Inductive Analysis of Security Protocols in Isabelle/HOL with Applications to Electronic Voting

Denis Frédéric Butin

B.Sc., M.Sc.

A dissertation submitted to Dublin City University in fulfilment of the requirements for the award of Doctor of Philosophy (Ph.D.) in the Faculty of Engineering and Computing, School of Computing

Supervisor: Dr. David Gray

September 2012

Declaration

I, Denis Frédéric Butin, hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed Student ID: 58118977 3 September 2012

Contents

Abstract

Security protocols are predefined sequences of message exchanges. Their uses over computer networks aim to provide certain guarantees to protocol participants. The sensitive nature of many applications resting on protocols encourages the use of formal methods to provide rigorous correctness proofs. This dissertation presents extensions to the Inductive Method for protocol verification in the Isabelle/HOL interactive theorem prover. The current state of the Inductive Method and of other protocol analysis techniques are reviewed. Protocol composition modelling in the Inductive Method is introduced and put in practice by holistically verifying the composition of a certification protocol with an authentication protocol. Unlike some existing approaches, we are not constrained by independence requirements or search space limitations. A special kind of identity-based signatures, auditable ones, are specified in the Inductive Method and integrated in an analysis of a recent ISO/IEC 9798-3 protocol. A side-by-side verification features both a version of the protocol with auditable identity-based signatures and a version with plain ones. The largest part of the thesis presents extensions for the verification of electronic voting protocols. Innovative specification and verification strategies are described. The crucial property of voter privacy, being the impossibility of knowing how a specific voter voted, is modelled as an unlinkability property between pieces of information. Unlinkability is then specified in the Inductive Method using novel message operators. An electronic voting protocol by Fujioka, Okamoto and Ohta is modelled in the Inductive Method. Its classic confidentiality properties are verified, followed by voter privacy. The approach is shown to be generic enough to be re-usable on other protocols while maintaining a coherent line of reasoning. We compare our work with the widespread process equivalence model and examine respective strengths.

Acknowledgements

I am very grateful to all the people who kindly supported me over the course of this thesis.

My first thank you goes to my supervisor, David Gray, who always took time for stimulating discussions, brainstorming and feedback. His assistance, help and experience were invaluable.

I was very fortunate over the course of this thesis to meet Giampaolo Bella, who has been deeply involved with the Inductive Method since its early stages. His technical expertise, enthusiasm and friendship meant a great deal to this work and lifted my spirits.

My teachers at Université François-Rabelais de Tours, through their clarity of exposition, raised my appetite for more science. Pierre Damphousse will be missed by many; his enthusiasm for science and life in general was contagious. He encouraged me to go for a PhD, and also organised the exchange semester at Université Laval in Québec, where Claude Levesque taught excellent cryptography and algebraic number theory classes and Jean-Marie de Koninck energetically introduced analytical number theory.

Jean Everson Martina and David Sinclair kindly accepted to examine this thesis. Thank you for taking the time to evaluate my work.

I enjoyed the inspiring research atmosphere at Università di Catania and interesting conversations with Rosario Giustolisi, Luigi Grillo, Gianpiero Costantino and Gabriele Lenzini. The Isabelle users mailing list provided a sense of community and many helpful tips by Lawrence Paulson, Jasmin Blanchette and others. The thorough documentation available on the Isabelle/HOL website helped me tame The Beast to some extent. Science Foundation Ireland made this PhD possible through the grant 08/RFP/CMS1347.

My labmates and friends of DCU helped make the lab and campus a fun and engaging place to work. Michèle Péporté and Frantisek Polach welcomed me when I first arrived at DCU, and the former never stopped providing great conversations and belgian chocolate. I was blessed with great flatmates during my stay on campus: Sharon Murphy, James Walsh, Ana Barat and Ubaldo Colmenar. I had the pleasure of sharing the crypto lab with Luis Julian Dominguez Perez, Chen Yu, Manuel Charlemagne, Ezekiel Kachisa and Hyun Sung Kim. Naomi Benger, despite her stellar productivity, always managed to make time for a tea & language break or fun barbecues and outings on weekends.

Once the crypto lab became empty, Marija Bezbradica, Karthika Raghavan and Alina Sîrbu prevented me from becoming a full-fledged hermit. Bénédicte Galtier, Flavie Lacroix, Morgane Croissant left Eire too soon. Phil Lenzenhuber and Andrej Blagojević made for challenging squash opponents and good friends. Low-cost airlines allowed me to visit my treasured friends in France and elsewhere unreasonably often.

Thank you to my parents, for being there.

Finally, a very special thank you to Melanie, who was by my side every step of the way.

List of Figures

List of Tables

Protocol Notation Syntax

Symbol	Meaning
$i.$	i^{th} protocol step
$A, B, C \ldots$	Agent names
$\{\!\| \cdot, \cdot \|\!\}$	Concatenation
$A \longrightarrow B \ : \ M$	A sends message M to B
K_A^+	Public key of agent A
K_A^-	Private key of agent A
$\{\!\| M \|\!\}_K$	Encryption of message M with key K
$Sig_A(M)$	Signature of message M by agent A
C_A^I	Certificate for A, issued by I
$C_A^{I_1 \ldots I_n}$	Certificate chain for A, with root certificate I_1

Table 1: Protocol notation syntax

Chapter 1

Introduction

During the last few decades, computer networks have become the communication tool of choice for most organisations, independently of companies' sector of activity. The number of Internet users throughout the world has expanded dramatically since the beginning of the century: over 2 billion people enjoyed online access in December 2011 [72], over an hundredfold increase from 1995. Networking is thus pervasive and omnipresent — not only for business use but also for personal communication, often of sensitive nature. Internet banking and private correspondence are examples of common tasks for which users rightfully demand security guarantees.

In many settings, it is hence essential that information be protected, notably from eavesdropping, forgery and impersonation.

1.1 The Need for Network Security

Cryptography provides families of tools seeking to prevent these attacks and to reach various security goals, the primary ones being:

- *Confidentiality* — a message can only be read by its intended recipient(s);
- *Integrity* — the message has not been altered during the course of its transmission;

- *Non-repudiation* — ensuring that an agent cannot deny an action (usually, sending a message or generating a digital signature);
- *Authentication* — guaranteeing the identity of one or several agents.

Often, cryptosystems (sets of cryptographic algorithms) are designed to meet several of these goals simultaneously. Recent research in the field has provided many other solutions, such as zero-knowledge proofs [62] (allowing an entity to prove knowledge of a secret without actually disclosing it) and secret sharing [111] (disseminating confidential information among several agents, allowing prevention of individual abuse and implying distribution of trust).

Two paradigms coexist in modern cryptography: the *symmetric (secret-key)* and *asymmetric (public-key)* settings. In symmetric cryptography, agents share a common key or a pair of keys in which one is easily derived from the other. It is then crucial that only the two communication partners know the key. In the asymmetric setting, each agent is associated with a private and a public key pair. The public key is openly disclosed to all network participants, whereas the private one remains the secret of the sole agent. Public-key encryption removes the need for an initial key exchange (or *key agreement*) found in secret-key settings. It also makes digital signatures possible — using an agent's public key, all other agents can verify the authenticity of a message she created.

Notwithstanding the abundance of research involving public-key cryptography and our focus on it in this work, it should be stressed that symmetric encryption retains a major role in contemporary cryptosystems. Since asymmetric encryption is slower and produces a larger resulting length increase (*ciphertext expansion*) than symmetric algorithms, a common method is to encrypt only a key asymmetrically, before using this key to encrypt a message with a symmetric algorithm. The two approaches are thus combined.

In most cases, public-key-based systems require a *Public Key Infrastructure* (PKI). A PKI is a framework designed to manage the mapping between identities and pub-

lic keys. The International Telecommunication Union issued a PKI standard recommendation, called X.509, in 1988 [117]. Certificate standards for X.509 were further profiled by the Internet Engineering Task Force (IETF) in a series of RFC documents, such as RFC2510 [71]. Commonly used protocols such as HTTPS, SSH and TLS implement X.509 certificates.

Despite the benefits they provide, PKIs imply some computational and logistical overhead. A recent evolution of asymmetric cryptography, called *identity-based cryptography*, reduces the need for PKIs by changing the way public keys are selected. Instead of being generated by agents and then requiring certification by a trusted third party (part of the PKI), public keys are simply derived from agents' identities by a process known to all. Here, *identity* is usually to be understood as a person's name, email address or social security number. Numerous other examples are possible; the point is that an agent's public key is directly available to anybody who knows her, without the need to contact a trusted third party first. Identity-based cryptography branches into *identity-based encryption* (IBE) and *identity-based signatures* (IBS). For IBE, a message is encrypted with the recipient's public key (i.e. a string representing his identity), and only decryptable using the recipient's secret key. For IBS, a signature is generated using the sender's secret key; the signature's validity can be checked by any agent aware of the sender's public key, i.e. her identity. Nevertheless, some form of infrastructure remains needed to certify the key generation center's (KGC) public key.

Protocols are predefined message exchanges between a finite set of actors, called *principals* or *agents*. Security protocols are designed to provide security guarantees through the use of cryptographic primitives. A protocol can be flawed even if the underlying cryptography is perfect; network communication is often insecure, and transmitting apparently uncritical unencrypted information can lead to impersonation, for instance through attacks combining previously transmitted message elements to create a new, fake one. This particular kind of attack, called a replay attack [63], is fairly common.

When evoking the correctness of security protocols in our work, we thus focus on flaws which are not due to weaknesses of the underlying cryptographic algorithms but rather inherent to the structure and contents of the message exchange sequence. Cryptographic algorithms are seen as black boxes and their security assumed perfect; this will be the case not only for encryption functions but also, e.g., for hash functions which will be assumed to be perfect one-way functions.

Protocols have been checked for correctness intuitively for a long time, but the availability of mechanical tools to carry out this task automatically or semi-automatically is recent. The two main approaches are model checking, which searches a finite space of protocol configurations for possible flaws, and (interactive) theorem proving. Due to computational constraints, model checking applied to security protocol is typically limited to a small population of agents. If no attack is found, there may still be possible attack scenarios against a larger system running the protocol. On the other hand, interactive theorem proving provides mathematical proof of a protocol's correctness and does not assume a finite number of message steps or agents.

Electronic voting is an application for which both new cryptographic primitives to define useful protocols and new techniques to formally analyse them are required. In many cases, e-voting protocols feature any number of voters interacting with so-called election officials, entities that gather data about voters and their ballots and perform the tasks leading to the announcement of election results. Since those protocols are meant to be used in official elections with far-reaching consequences, rigorous guarantees of correctness are expected by voters and (some) governments. Formal methods are invaluable in establishing correctness of protocols in general, and are therefore well-suited to analyse this particular class of protocols. However, the need to protect the privacy of voters while still allowing them to check election results calls for the use of cryptographic building blocks seldom used in protocols in general. Blind signatures and bit commitments are two examples of such cryptographic primitives. Blind signatures allow a user to sign a message without revealing its contents to the user. Intuitively, this mirrors the pen-and-paper signing of a carbon-lined envelope.

To verify e-voting protocols, new cryptographic primitives must therefore be integrated into the chosen formal tool. What is more, many properties expected from e-voting protocols cannot be specified directly using existing frameworks in their current state. Privacy-type properties exemplify this issue; this family of properties relates to the association between a voter and the voter's ballot. For instance, the first privacy property is often called voter privacy or ballot secrecy and is defined as the fact that how a particular voter voted is not revealed to anyone.

This thesis addresses the formal verification of security protocols via the Isabelle theorem prover, with a number of extensions to the verification framework to account for novel situations such as e-voting security goals, protocol composition and the use of new cryptographic primitives.

In the next sections, a more detailed presentation of required cryptographic building blocks (§1.2) and of identity-based cryptography (§1.3) will be given. An overview of security layers and protocol security follows (§1.4). The second part of this chapter will describe our motivations (§1.5), contributions (§1.6) to the subject and an outline of the thesis.

1.2 Cryptography

Security protocols rely on cryptographic primitives. We now give an overview of elementary cryptologic concepts and present some major developments of the last decades.

1.2.1 A Short History of Modern Cryptology

Information hiding techniques date back to centuries before the common era, and clever attack techniques such as frequency analysis already appeared at the beginning of the Middle Ages. Nevertheless, sound mathematical foundations for cryptology were only built in the twentieth century.

In his landmark 1949 paper, Shannon [113] applied information theory, statistics

and probability theory to a formalised view of cryptology, marking the beginning of the field's modern era. Among other fundamental contributions to the field, the publication defines the notion of information theoretic security and includes a proof of one-time pads' adherence to this strict notion.

In the seventies, the independent discovery of public-key cryptography by Ellis (at GCHQ, his work was classified at that time [50]) and by Diffie and Hellman [47] influenced the development of cryptography in profound ways. Two years after Diffie and Hellman's paper, the concept of public-key cryptography became a reality through Rivest, Shamir & Adleman's RSA algorithm, which is still ubiquitous over thirty years later.

A parallel development is the advent of quantum cryptography. The first theoretical step was taken by Wiesner [125], who in 1970 introduced the primitive later to be known as oblivious transfer (even though his paper was only published a decade later). In 1994, Shor [114] invented an algorithm making integer factorisation in polynomial time possible on a quantum computer.

1.2.2 Symmetric Cryptography

Symmetric cryptography involves (at least) two principals sharing a common key.

A *symmetric encryption scheme* consists of the following algorithms:

1. Keygen:
    - Input: Security parameter l
    - Output: key K
2. Encrypt:
    - Input: Message M; K
    - Output: Ciphertext $\{\!|M|\!\}_K$
3. Decrypt:

- Input: $\{\!|M|\!\}_K$; K
- Output: M

The key feature that remains in black-box modellings of symmetric cryptography is the fact that the same key is used for encryption and decryption. This raises the issue of how this key is distributed securely in the first place. The two main symmetric key exchange techniques are as follows:

1. Using the Diffie-Hellman (DH) key exchange protocol over a possibly insecure channel. The DH protocol's security rests on the difficulty of the discrete logarithm problem in the multiplicative subgroup $\mathbb{Z}_p^*$ of the field $\mathbb{F}_p$ (p prime). While discrete exponentiation is fast, no efficient algorithm for the inverse computation is known.

2. Encrypting the symmetric key with an asymmetric algorithm before transmitting it. The result is called a hybrid cryptosystem, and the process is known as key encapsulation.

The following two sections deal with the two symmetric ciphers which have been most widely used during the last decades. Both were defined as standards by the U.S. National Institute of Standards and Technology (NIST).

Classic symmetric algorithms DES is a symmetric block cipher based on a Feistel structure, widely used since its publication[1] as a standard by NIST in 1977 [90]. After numerous, increasingly efficient attacks it is now considered obsolete. Notably, in 2006, the COPACABANA FPGA array broke DES by brute force in little more than a week [77]. It had only stopped being a standard a year before, in 2005.

AES has been a NIST standard (as DES's successor) since 2001 [91] and has been implemented, in particular, in IPSec.[2] While it is — like DES — a block cipher, its structure is not a Feistel network but a substitution-permutation (SP) network. A major

[1] The document referenced in the bibliography is a later version of the standard.
[2] The Internet Protocol security protocol suite.

difference is that all bits are changed during each round. In the former case, only half of the bits are changed every round; the other half is merely moved. The consequence is higher diffusion in the SP structure. Indeed, full diffusion is achieved in AES after only two rounds; this means that after two rounds, every output bit depends on each input bit.

Differential cryptanalysis Differential cryptanalysis is a potent form of chosen-plaintext attack. Its focus is to deduce information about the secret key by analysing differences between ciphertexts resulting from a selection of well-chosen plaintexts. Biham and Shamir introduced differential cryptanalysis in 1990, in a paper [20] applying this technique to DES. The same authors also soon successfully used it against the FEAL block cipher [21].

Message Authentication Codes A *Message Authentication Code* (MAC) is a short tag attached to a message to guarantee its authenticity. It is based on symmetric cryptography: the sender and the receiver share a secret key. Most MACs make use of a hash function (§1.2.4). The corresponding asymmetric technique is called a digital signature: see §1.2.3 and §1.3.1.

1.2.3 Asymmetric Cryptography and PKIs

Unlike symmetric cryptography, asymmetric (also called public-key) cryptography relies on key pairs, where one key is private and the other is public. A *Certificate Authority* (CA) is involved — see §1.2.3.

An *asymmetric encryption scheme* consists of the following algorithms:

1. Keygen: (Run by recipient B)
    - Input: Security parameter l
    - Output: public key K_B^+; private key K_B^-
2. Register: (Run by CA after authenticating recipient B)

- Input: K_B^+
- Output: Certificate C_B^{CA}

3. Getkey: (Run by sender A)
   - Input: $C_B^{CA...I_n}$
   - Output: $\{K_B^+, \texttt{ INVALID}\}$
4. Encrypt: (Run by sender A)
   - Input: K_B^+; plaintext M
   - Output: Ciphertext C
5. Decrypt: (Run by recipient B)
   - Input: K_B^-; C
   - Output: M

The sender A obtains B's public key K_B^+ through a certificate chain connecting a certificate for B to the trusted root CA, and proceeds to create a ciphertext from the message M and K_B^+. B recovers M from the ciphertext using his private key K_B^-.

An *asymmetric signature scheme* consists of the following algorithms:

1. Keygen: (Run by signer A)
   - Input: Security parameter l
   - Output: public key K_A^+; private key K_A^-
2. Register: (Run by CA after authenticating A)
   - Input: K_A^+
   - Output: Certificate C_A^{CA}
3. Sign: (Run by signer A)
   - Input: Message M; K_A^-

- Output: Signature σ

4. Getkey: (Run by verifier B)
   - Input: $C_A^{CA...I_n}$
   - Output: $\{K_A^+,$ `INVALID`$\}$
5. Verify: (Run by verifier B)
   - Input: σ; K_A^+
   - Output: $\{$`VALID, INVALID`$\}$

A generates a signature on the message M using her private key K_A^-. The verifier B obtains A's public key K_A^+ through a certificate chain connecting a certificate for A to the trusted root CA, and uses K_A^+ to verify the signature.

RSA In 1978, two years after Diffie and Hellman introduced public-key cryptography, the first practical asymmetric cryptosystem was created by Rivest, Shamir and Adleman [103]. The main theoretical results used are the following:

1. The decision version of the integer factorisation problem is *hard*; in other words, multiplication of primes is considered a one-way function.
2. The RSA problem is *hard* — see below.
3. Fermat's Little Theorem: if p is prime, then $\forall a \in \mathbb{N},\ a^p \equiv a \pmod{p}$.

Saying that a problem is *hard* (or *intractable*), in this context, means that it cannot be solved efficiently by currently available (non-quantum) computational methods: no known non-quantum algorithm can solve it in polynomial time.

The first step of the scheme is to pick two large primes p and q, of comparable size. The resulting product, $n = pq$, should be at least 2048 bits long to be considered secure until 2020 and even longer if more lasting security is desired; see Lenstra's paper [80] for justifications.

Then the public key e is chosen; it has to fit the requirement

$$[1 < e < \varphi(n)] \wedge [\gcd(\varphi(n), e) = 1] \tag{1.1}$$

Where $\varphi(n)$, Euler's totient function, is the number of positive integers less than or equal to n and relatively prime (sharing no common factor except 1) to n. In particular, when p and q are prime (as in our current example), $\varphi(n) = (p-1)(q-1)$.

The (extended) Euclidean algorithm is then used to compute the private key d. It is the modular inverse of e mod $\varphi(n)$, i.e.

$$de \equiv 1 \pmod{\varphi(n)} \tag{1.2}$$

Suppose Alice wants to send the message x to Bob. First Bob needs to make his public key e available, along with n. Now Alice sends the encrypted message $M = x^e \pmod{n}$. Bob uses his private key d to compute $M^d = x^{de} \pmod{n}$. Because of (1.2) and Fermat's Little Theorem, this yields x and Bob recovers the message. Indeed, congruence $\pmod{pq}$ is equivalent to the conjunction of congruence $\pmod{p}$ and congruence $\pmod{q}$ here, since p and q are prime. It is easy to check each condition by noticing that $de - 1$ must be a multiple of $\varphi(n) = (p-1)(q-1)$.

Assuming hardness of integer factorisation is one requirement for the scheme to be secure. Indeed, recovering x from x^e involves determining d; d can be computed using the knowledge of e, but only given $(p-1)(q-1)$. Calculating this, in turn, is only possible when p and q are known, which reduces to factoring n.

The second assumption is hardness of the RSA problem, defined as follows by Rivest and Kaliski [102]:

Problem (RSA)**.** *Given an RSA public key* (n, e) *and a ciphertext* $M = x^e \pmod{n}$*, compute* x*.*

The RSA problem is not harder than the integer factorisation, since we just showed that being able to find p and q from n yields x. It is not known, however, if integer

factorisation is harder than the RSA problem. Notably, in 1998, Boneh and Venkatesan showed that breaking an RSA system with a *low exponent* is not equivalent to integer factorisation [30].

Public Key Infrastructures: Certificate Authorities and Webs of Trust Public Key Infrastructures (PKI) are frameworks providing the logistics for the use of public-key cryptography on a large scale. They allow a trusted entity to securely bind a public key to the name of its owner. Without a PKI, there is no guarantee that a public key actually belongs to its claimed owner. Two types of PKIs are widely used in practice.

1. The most commonly used system, X.509 [117], is based on certificate chains. The link between a user and his public key is guaranteed by a certificate belonging to a *Certificate Authority* (CA); the key/identity binding for *this* entity (the CA) is guaranteed by another certificate belonging to a higher-level CA. Every CA is certificated by its parent CA, until a *root CA* is reached. This root CA is either unsigned, or self-certified. Either way, if the root CA is trusted, the hierarchy of trust implies that its children CA can be trusted. In practice, certificates are signed with a CA's private key. The largest root CA as of 2012 is a commercial company, VeriSign. Certificates can also be cancelled by being placed on a (dynamic) certificate revocation list, to which the *CRL Distribution Points* certificate field points. Reasons for revocation include change of name and private key compromise. Certificates expire at the end of a validity period.

2. The second PKI paradigm is called a *web of trust*. It was first found in the celebrated Pretty Good Privacy (PGP) software, developed by Zimmermann in 1991 [61]. In contrast with a CA chain, webs of trust are highly decentralised and feature no authorities. Instead, each user possesses a keyring defining trusted correspondents, or *trusted introducers* as Zimmermann puts it. Various levels of trust can be assigned. When checking a certificate for validity, if the sender is not in the recipient's keyring, levels of trust assigned to the sender by other users in the recipient's keyring are combined to make a decision. In

uncertain cases, the recipient has to make an informed choice. Unfortunately, in practice, this choice is often ill-informed. Unlike X.509, the system is thus non-deterministic and certificates can be signed by more than one party. PGP certificates can also be revoked, either by the certificate's owner or by an entity explicitly designed as a *revoker* by the former.

When mentioning PKIs in our work, we always refer to infrastructures of the first kind.

1.2.4 One-way Functions and Hash Functions, the SHA Family

Hash functions are a commonly used example of one-way functions, i.e. functions which are easy to compute in one direction and hard to reverse. A hash function takes as input a bit string of any length, and outputs a fixed length bit string which can be seen as a digest of the message. An efficient hash function should have several properties:

1. Hashes can be computed quickly, even for long input messages;
2. It is infeasible to find two different messages resulting in identical hashes (*collisions* resistance);
3. It is infeasible, given a hash, to find a message yielding this exact hash (*preimage* resistance).

Hash functions are part of many cryptosystems, both signing and encrypting ones. They are also used for the purpose of integrity checking; when a user has finished downloading a file from a network, he can compute its hash, also known as the file's *checksum*. If the file provider distributes the checksum (for a given hash algorithm) along with the file, the user can verify if the file he obtained has been transmitted without errors.

The currently most used hash functions are those of the Secure Hash Algorithm (SHA) family. All of them were developed under the supervision of the NIST and are part of the Secure Hash Standard (SHS) [92].

SHA-1 SHA-1 is a hash function with an output of 160 bit, a 512 bit block size, and a Merkle-Damgård construction [42] to chain those blocks. It was specified as a standard in 1995 and is still widely used and considered usable by NIST despite numerous attacks, notably the one by de Cannière and Rechberger in 2006 [35].

SHA-2 Rather than being a single hash function, SHA-2 is a family of functions producing various digest sizes, from 224 to 512 bits. Recent collision attacks have been realised by Indesteege *et al.* [69] and Sanadhya & Sarkar [106].

SHA-3 SHA-3 is a new proposed NIST standard. A competition, first announced in 2007, is currently underway to choose the hash algorithm which will become known as SHA-3. As of August 2012, five finalist algorithms remain. A final decision is expected before the end of 2012 [93].

1.2.5 Provable Security of Public-key Schemes

For public-key schemes, security is generally proven by relating attacks on the cryptosystem to the resolution of a computational problem which is believed to be hard, such as integer factoring or the discrete logarithm problem. The most common approach is to assume that an adversary can break a cryptosystem, and then prove that, given this assumption, the adversary would be able to solve a classic computational problem. This is called a *security reduction*. Security is quantified by an asymptotic bound on computational power, in function of the key length. A reduction is called *tight* if there is little to no loss of efficiency compared to the computational problem; that is, if breaking the scheme is exactly as hard as solving the problem. In that case, the security parameter used for the scheme does not need to be larger than the one that would be used initially.

In the *Random Oracle Model*, hash functions are modelled by ideal random functions, with the additional condition that this kind of function, or oracle, outputs identical values for identical inputs. In the *Generic Group Model*, actual bit string representations of group elements, such as those of a given elliptic curve group, are modelled

by ideal random representations. While proofs in the Random Oracle Model and the Generic Group Model are useful indications as to a scheme's security, they do not constitute ultimate guarantees and have been controversial; see *The Random Oracle Methodology, Revisited* [34].

1.3 Identity-Based Cryptography

Identity-based (IB) schemes simplify public-key cryptography infrastructures by giving universal access to any user's public key, no longer requiring communication with a Certificate Authority to obtain it. In IB schemes, the public key of a user is easily deduced from his identity by a process known to all. This reduces communication overhead. Despite widespread broadband internet connection, many systems — such as embedded devices — can greatly benefit from reduced data transfer need. Key management is also simplified. Shamir's 1984 seminal article [112] proposed the basic principle of identity-based encryption (IBE), identity-based signatures (IBS) and provided a concrete scheme for IBS (based on RSA). A concrete scheme for IBE, however, appeared only 17 years later in Boneh and Franklin's scheme based on the Weil pairing [29]. IB cryptography is now commonly used in the industry. Notably, companies such as Trend Micro [119] and Voltage [120] distribute IB email encryption and toolkit software.

1.3.1 Identity-Based Signatures

Identity-based signatures are digital signatures that can be verified if the signer's identity is known.

Principles of IBS Alice (A) generates her signature σ (on the message M) from M and her secret key K_A^-, and sends $\{\!|M, \sigma|\!\}$ to Bob (B) through a (possibly insecure) communication channel. B validates (or rejects) σ by checking its validity using A's public key and the master public key master-key of the KGC.

The public key of a principal A is obtained from her (public) identity ID, and from the public system parameters params. A's secret key K_A^- is derived by the KGC using her identity ID and master-key, a master secret key only known to the KGC.

There is no key channel between the users, only between each user and the KGC. A user's public key is deduced directly from her identity by a public process, and her secret key is communicated directly to her by the KGC upon proof of identity. A's secret key is hence known to exactly two entities: A and KGC.

In contrast, in a secret-key setting, the (secret) key has to be transmitted separately on a secure channel; and in a public-key setting, the public key is also sent through a specific channel, although it does not have to be confidential, only possess integrity.

The KGC is responsible for delivering their secret key only to users who prove their identity to it, else the infrastructure is compromised. The seed has to remain secret, lest the entity getting a hold of it be able to generate secret keys for any user identity.

IBS algorithms We now describe the algorithms making up an IBS scheme.

1. Setup: (Run by KGC)
    - Input: Security parameter l
    - Output: System parameters params; master-key
2. Extract: (Run by KGC)
    - Input: params; master-key; ID
    - Output: Secret key K_A^- (transmitted to the signer)
3. Sign: (Run by signer)
    - Input: Message M; params; ID; K_A^-
    - Output: Signature σ
4. Verify: (Run by verifier)

- Input: σ; ID; M; params
- Output: {VALID, INVALID}

Properties of IBS

- The public keys of all users are deriveable to all peers, since they are directly computed (through a publicly known method, sometimes a hash function) from identities. This is the defining characteristic of IB cryptography. The KGC's master public key is also known to all. It therefore has to be published, relying on some limited form of PKI.
- K_A^- is communicated to A by the KGC upon request and authentication.
- It is impossible to sign a message on behalf of A without knowing K_A^-.
- The KGC knows all secret keys, since they are all generated directly from the master secret key and a user's identity. This is called *key escrow*.
- Hence, a message that decodes properly using A's public key (its identity) has been signed either by A or by the KGC. This provides *weak non-repudiation*, as A can refute a signature since it may have been produced by the KGC.
- To check a signature, the recipient needs exactly two additional elements: the master public key, and the sender's identity.

IBS implementation When introducing IBS in 1984, Shamir was able to propose an implementation based on RSA straight away. In 2002, Hess [66] developed an efficient IBS scheme using elliptic curves. The main mathematical tools used were the Weil and Tate pairings, maps which are part of a widely-used family of bilinear applications called *pairings*.

Shamir's IBS system To achieve IBS practically, Shamir [112] used RSA in the following way:

- Let i be the user's identity, m the message to be signed, n the (public) product of two large primes and e a (public) large prime such that $(e, \varphi(n)) = 1$. Then the secret key for i is g, such that $g^e = i \pmod{n}$. If RSA is secure, g can not be recovered from i. Note that $g = i^d \pmod{n}$, where d is an inverse of $e \pmod{\varphi(n)}$ and can be computed by the KGC like in the RSA scheme.

- The signer picks a random r and sets $t = r^e \pmod{n}$. He also computes $s = g \cdot r^{f(t,m)} \pmod{n}$, where $f(t, m)$ is a (public) one-way function. The signature is (s, t).

- The signature verification condition, then, is $s^e = i \cdot t^{f(t,m)} \pmod{n}$. Since $(e, \varphi(n)) = 1$, the condition reduces to $s = g \cdot r^{f(t,m)} \pmod{n}$.

Pairings A pairing is a map $e : \mathbb{G}_1 \times \mathbb{G}_1 \mapsto \mathbb{G}_2$ between cyclic groups $\mathbb{G}_1$ (additive) and $\mathbb{G}_2$ (multiplicative), both of large prime order q, possessing the following properties:

1. Bilinearity: $\forall\, a, b \in \mathbb{Z}_q^*, \forall\, P, Q \in \mathbb{G}_1, \quad e(aP, bQ) = e(P, Q)^{ab}$;

2. Non-degeneracy: $\exists\, P, Q \in \mathbb{G}_1 / \; e(P, Q) \neq 1$;

3. Computability: $\forall\, P, Q \in \mathbb{G}_1$, there is an efficient way to compute $e(P, Q)$.

In practice, $\mathbb{G}_1$ is typically a group of points on an elliptic curve over a finite field. Elliptic curves over which pairings provide efficient and secure schemes are said to be *pairing-friendly*. For a detailed definition, see the taxonomy by Freeman, Scott and Teske [57].

Hess's IBS system In 2002, Hess [66] proposed a more efficient signature scheme, following the introduction by Boneh and Franklin (see below) of pairings for identity-based cryptography. Its security is based on the assumed hardness of the Diffie-Hellman Problem (DHP); namely:

Problem (Diffie-Hellman)**.** *Let g be a generator of a cyclic group $\mathbb{G}$ of prime order q, and $a, b \in \mathbb{Z}_q$. Given g,g^a and g^b, find g^{ab}.*

Let e be a non-degenerate, bilinear pairing from $G \times G$ to V, where $(G, +)$ and $(V, \cdot)$ are cyclic groups of order q, q prime. P is a generator of G, and h, H are (public) hash functions. The scheme consists of the following algorithms:

1. Setup:
    - Input: Security parameter l
    - Output: System parameters params: (e, P, tP, H, h); master-key: t (randomly chosen by TA). tP is published.
2. Extract:
    - Input: params; master-key; ID
    - Output: Secret key $S_{ID} = tH(ID)$
3. Sign:
    - Input: Message m; params; ID; S_{ID}
    - Output: Signature $\sigma = (u, v) \in G_1 \times (Z/qZ)^{\times}$, with $r = e(P_1, P)^k$, $v = h(m, r)$, $u = vS_{ID} + kP_1$ (with P_1 and k chosen at random by Signer)
4. Verify:
    - Input: $\sigma = (u, v)$; ID; m; params
    - Output: {`VALID` if $v = h(m, r)$ with $r = e(u, P) \cdot e(H(ID), -tP)^v$, else `INVALID`}

A concise, efficient and provably secure scheme, it has been widely used and adapted since.

1.3.2 Identity-Based Encryption

Identity-based encryption works similarly to IBS; here, the identity of one agent is the encryption key.

IBE algorithms A set of four algorithms is used to described IBE schemes. They were defined in [29] as follows:

1. Setup: (Run by KGC)
   - Input: Security parameter l
   - Output: System parameters params; master-key
2. Extract: (Run by KGC)
   - Input: params; master-key; ID
   - Output: Secret key K_B^-
3. Encrypt: (Run by sender)
   - Input: params; ID; plaintext M
   - Output: Ciphertext C
4. Decrypt: (Run by recipient)
   - Input: params; K_B^-; C
   - Output: M

IBE implementation As mentioned earlier, the path from concept to implementation was straightforward for IBS but considerably longer for IBE. The two first concrete IBE schemes both appeared in 2001.

Cocks' IBE scheme In [37], Cocks described an implementation of IBE based on the *quadratic residuosity problem* (QRP). Recall the QRP:

Problem (Quadratic Residuosity)**.** [3] *Let $n = pq$ and x be integers, with p and q large*

[3] $\left(\frac{x}{n}\right)$, the Jacobi symbol of x and n, is equal to the product of the Legendre symbols of x and n's factors, i.e. p and q. Furthermore, the Legendre symbol is defined as such (r prime):

$$\left(\frac{x}{r}\right) = \begin{cases} -1 & \text{if } \nexists\, t \,.\, t^2 \equiv x \pmod{r} \\ 0 & \text{if } r|x \\ +1 & \text{if } \exists\, t \,.\, t^2 \equiv x \pmod{r} \end{cases}$$

primes and $\left(\frac{x}{n}\right) = 1$. *Is* x *a square mod* n*?*

The scheme works as follows: let p and q be two primes with $p = 3 \pmod 4$ and $q = 3 \pmod 4$. Let $m = pq$ be public, but p and q only known to an authority. Also assume the existence of a public hash function. It is applied to ID until the result is a value $a \pmod m$ with $\left(\frac{a}{m}\right) = 1$.

It follows from $\left(\frac{a}{m}\right) = 1$ that $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right)$, hence either a or $-a$ must be a square $\pmod m$. The authority computes the square root $\pmod m$ of whichever of those values is a square $\pmod m$, and gives the result r to Alice.

Now Bob generates a transport key, and sends it to Alice bit by bit, encrypted: Assume without loss of generality that $r^2 = a \pmod m$. Bob picks a random t such that $\left(\frac{t}{m}\right) = x$, with x being the current bit of the key that Bob wishes to send. Bob sends Alice $s = (t + a/t) \pmod m$.

Bob must replicate this process (creating overhead) using a different t because he cannot know whether Alice has the root of a or of $-a$.

Alice can recover x by computing $\left(\frac{s+2r}{m}\right)$, which is equal to $\left(\frac{t}{m}\right) = x$.

Cocks' scheme constitutes a proof of concept, but is not widely used because of the high degree of ciphertext expansion (every bit of plaintext produces a segment of ciphertext as long as the key). The following scheme is more efficient.

Bohne and Franklin's IBE scheme In [29], an IBE system using the Weil pairing is presented. An identity is represented as a point on an elliptic curve. The scheme's security is based on the hardness of a variant of the Diffie-Hellman problem:

Problem (Computational Bilinear Diffie-Hellman)**.** *Given* $(P, aP, bP, cP) \in \mathbb{G}_1^4$ *with* $a, b, c \in \mathbb{Z}_q^*$ *chosen at random, find* $e(P, P)^{abc}$.

Boneh and Franklin present two versions of the scheme in their paper, called BasicIdent and FullIdent. For the sake of simplicity, we present BasicIdent here. Let e be a non-degenerate, bilinear pairing from $G_1 \times G_1$ to G_2, where G_1 and G_2 are groups of order q, q prime. P is a random generator of G_1, and H_1, H_2 are (public) hash

functions. Let $\{0,1\}^n$ be the message space. BasicIdent consists of the following algorithms:

1. Setup: (Run by KGC)
   - Input: Security parameter k
   - Output: System parameters params = $(q, G_1, G_2, e, n, P, sP, H_1, H_2)$; master-key = s (picked at random)
2. Extract: (Run by KGC)
   - Input: params; master-key; ID
   - Output: Secret key $sH_1(ID)$
3. Encrypt: (Run by sender)
   - Input: params; ID; plaintext M
   - Output: Ciphertext $C = (rP, M + H_2(g_{ID}^r))$ with $g_{ID} = e(H_1(ID), sP)$
4. Decrypt: (Run by recipient)
   - Input: params; d_{ID}; $C = (U, V)$
   - Output: $M = V + H_2(e(d_{ID}, U))$

BasicIdent is secure in the IND-ID-CPA[4] sense in the Random Oracle Model, with a reduction to the Computational Bilinear Diffie-Hellman Problem. FullIdent, obtained by applying a transformation due to Fujisaki and Okamoto [59] is secure in the IND-ID-CCA[5] sense. IND-ID-CPA and IND-ID-CCA are based on the standard notions of Chosen-Plaintext security (IND-CPA) and Chosen Ciphertext security (IND-CCA), respectively; for detailed technical definitions, see [29].

[4] Adaptive Chosen Plaintext security for an Identity-based scheme.
[5] Adaptive Chosen Ciphertext security for an Identity-based scheme.

1.3.3 Provable Security of IBE and IBS

In [121], Waters introduced an efficient IBE scheme, the proof of which doesn't require the Random Oracle Model. In [74], Kiltz gives an extensive security comparison of IBS schemes up to 2008.

1.4 Protocol Security

Network security can be defined at several levels:

1. Hardware: computers, physical network links.
2. Cryptographic primitives. This includes encryption algorithms, hash functions and digital signatures. Respective examples of these primitives are RSA, SHA-1 and the ElGamal signature scheme.
3. Security protocols: predefined sets of message exchanges, making use of cryptographic primitives. Commonly used ones include the Internet Protocol Suite (TCP/IP), TLS, Kerberos.

Attacks have been carried out successfully at all of those levels.

The hardware level is vulnerable to so-called *side channel attacks*. This strategy is recent, dating back to Kocher's timing analysis in 1996 [75], which measured cryptographic algorithms' execution time to deduce critical information about keys. In the analogue world, similar strategies were already used decades earlier.

Some attacks on cryptographic primitives (DES, FEAL, the SHA hash functions) were mentioned in §1.2.2and §1.2.4.

The security issues we will focus on are those of security protocols. We are concerned about protocol design weaknesses. Attacks on protocols can succeed even when the composing cryptographic algorithms are unassailable. For this reason, we will consider cryptographic primitives as theoretical perfect black boxes.

1.4.1 Security Protocols

Security protocols are predefined sets of message exchanges that aim to guarantee specific security properties about the agents involved in them and what they send.

The common syntax for security protocols The most commonly used notation for discussing security protocols is the one used in Burrows *et al.*'s seminal paper [32].

We adopt the following conventions (see also Table 1):

- Principals are represented by upper-case letters. $A, B, \ldots$ represent normal users. S is a server or a trusted third party.
- Protocols are written as numbered sequential lists of message transmissions between sender A and desired recipient B. Each step has the following form:

$$N. \quad A \longrightarrow B \quad : \quad M$$

  Where N is an integer and M is the message.

- Private (K_A^-) and public (K_A^+) keys of a principal A are distinguished. If A and S share a symmetric key, it is denoted K_{AS}.
- Concatenation is represented by commas.
- Encryption of a concatenation of items is denoted by placing braces around those items, and the key as a subscript: $\{\!|M, A, B|\!\}_{K_{AS}}$ is the encryption of the concatenation of the message M and the name of the principals A and B, encrypted with the shared symmetric key K_{AS}.
- A signed message is represented as a message encrypted with an agent's private key, e.g. $\{\!|M|\!\}_{K_S^-}$ represents a signature on message M by agent S. We sometimes also write $Sig_A(M)$.
- Nonces (non-guessable numbers) generated by a principle B are denoted N_B.

As an example, consider the following line:

$$2. \quad B \longrightarrow S \quad : \quad B, N_B, \{\!|A, N_A|\!\}_{K_{BS}}$$

This is the second step of a security protocol, Paulson's modified version of the Yahalom protocol. The principal B sends a message to the server S, consisting of the conjunction of three elements: the name of the sending principal, a nonce created by him, and the encryption with a symmetric key shared by B and S of the identity of another principal, A, and a nonce created by her.

A good catalogue of security protocols is to be found at the Security Protocols Open Repository (SPORE) [73]. Some example protocols are as follows.

Needham-Schroeder Public-Key The Needham-Schroeder Public-Key protocol, proposed in [94], is meant to provide mutual authentication between two principals. In [82], Lowe showed that the protocol is susceptible to a replay attack and fixed it by including the responder's identity in step 6. Here is the specification of Lowe's version:

1. $A \longrightarrow S \quad : \quad A, B$
2. $S \longrightarrow A \quad : \quad \{\!|K_B^+, B|\!\}_{K_S^-}$
3. $\mathbf{A \longrightarrow B} \quad : \quad \{\!|\mathbf{N_A}, \mathbf{A}|\!\}_{\mathbf{K_B^+}}$
4. $B \longrightarrow S \quad : \quad B, A$
5. $S \longrightarrow B \quad : \quad \{\!|K_B^+, A|\!\}_{K_S^-}$
6. $\mathbf{B \longrightarrow A} \quad : \quad \{\!|\mathbf{N_A}, \mathbf{N_B}, \mathbf{B}|\!\}_{\mathbf{K_A^+}}$
7. $\mathbf{A \longrightarrow B} \quad : \quad \{\!|\mathbf{N_B}|\!\}_{\mathbf{K_B^+}}$

Since steps 1, 2, 4 and 5 are only concerned with obtaining the public key, a core version of the protocol consisting merely of steps 3, 6 and 7 is often used for formal verification, notably in Isabelle. In chapter 4, we will compose this protocol with a certification protocol and verify the resulting composed protocol holistically, i.e. as a whole.

Kerberos In a distributed client-server environment, the Kerberos protocol provides mutual authentication between clients and servers via a ticket-granting system relying on a trusted third party. Symmetric cryptography is used to encode tickets in the general case. Public-key cryptography (i.e., an RSA variant) can be used via an extension of the most recent version of the protocol. Kerberos IV and V were formally analysed with the Isabelle theorem prover — see Table 3.1.

Electronic payment protocols From 1996 to 2000, the standard protocol suite for online credit card transactions was Secure Electronic Transactions (SET). It was formally analysed with Isabelle, in several stages, by Bella *et al.* [19]. SET experienced limited success, and its successor (used by both Visa and MasterCard) is called 3-D Secure. In practice, most credit card payments on the internet are protected solely by TLS, which was not designed specifically as an e-commerce protocol.

1.5 Motivation

The motivation for this thesis is twofold. Firstly, existing work using the Inductive Method [14] (introduced in §3.2) and the nature of its specification in Isabelle/HOL call for further investigation into the method's potential and boundaries. Since its fundamentals assume little more than the standard Dolev-Yao model [48], a great deal of applications have not yet been modelled in Isabelle/HOL despite their relevance to formal correctness study. Because the underlying platform is a theorem prover operating with higher-order logic and comprehensive extensibility, no intrinsic limitation prevents the modelling of those applications. This generality is the major difference between the Inductive Method and automated tools; it is also a strength that should be exploited.

The second motivation for this work is a pragmatic need for complementary approaches to security protocol verification. The study of electronic voting protocols provides a striking example of the need for alternative analysis methods. Significant research has been done on security guarantees such as voter privacy, but various lim-

itations remain. While the Inductive Method may eventually reach limitations of its own on this topic, a thorough investigation of its capabilities is worthwhile. The same can be said for protocol composition and support of specific cryptographic primitives, such as auditable identity-based signatures.

In practice, the work arising from these considerations tends to advance both inquiries. Tackling new verification scenarios requires building extensions to the method, which eventually defines its potential more clearly.

1.6 Outline and Contributions

In this chapter, we have presented the general problem of network security, fundamentals of cryptography and the issue of security at the protocol level with blackbox cryptography. We also presented the motivations for this thesis. We now summarise the structure of the remainder of the thesis, with a focus on original contributions.

- Chapter 2 is about existing formal approaches to security protocol analysis. Existing methods are reviewed and compared.
- Chapter 3 presents Isabelle/HOL briefly and the Inductive Method in detail.
- Chapter 4 describes our first contribution [17], a joint work with Giampaolo Bella. The handling of protocol composition in the Inductive Method is demonstrated by the verification of a protocol obtained by composing a certification step with a well-known mutual authentication protocol by Needham and Schroeder. The specification of composed protocols, never used before in the Inductive Method, is shown to be straightforward and applicable in generality. The case study marks a first phase in the holistic specification of protocols with the PKI they rely upon.
- Presented in chapter 5, our second contribution is the use of auditable identity-based signatures [65], specified in conjunction with an ISO/IEC 9798-3 [70]

authentication protocol. A side-by-side specification and verification in the Inductive Method clarifies the additional guarantees enjoyed by protocols replacing plain signatures with the former. Modelling this specific kind of signatures required adapting the framework of the Inductive Method at the message operator level.

- In chapter 6, the most significant contribution [33] of this thesis is presented. The Inductive Method is extended to deal with the unique characteristics and properties of electronic voting protocols. This collaboration with Giampaolo Bella features the specification of blind signatures and of unlinkability, modelling voter privacy. These extensions are then used to verify the FOO [58] protocol as a case study. Both the extensions to the framework and the privacy proofs are substantial.

- The last part of this work, in chapter 7, provides a synthesis of the thesis and discusses both the general findings that emerged from our research and directions in which future work is proposed.

- In the appendix A, the Isabelle theories for this work are provided.

Chapter 2

Security Protocol Analysis

Over the last decades, numerous methods and tools for security protocol analysis have been developed. Before presenting in detail the method and tool used in the remainder of the thesis, we give a panoramic view of other existing approaches. Many available tools combine more than one approach; therefore, we start by listing the main approaches and detail tools later.

2.1 Approaches for the Analysis of Security Protocols

The field of security protocol analysis has developed quickly over the last twenty years; the pace of research even seems to be increasing currently [39, 88]. It is difficult to give a totally exhaustive picture of existing methods and tools; therefore our goal in this section is to focus on the most significant approaches.

2.1.1 BAN Logic

BAN reasoning, introduced in [32], is one of the earliest approaches to formal verification of security protocols. It is also called belief logic, since its formalism is partly based on what information principals are entitled to believe based on messages they received. It is now mainly of historical interest, since many flaws found later in protocol underwent BAN analysis undetected. The question whether the error resided in the

BAN logic itself or in an incorrect idealisation of the protocol provoked some controversy, but is of minor importance today since model checkers and theorem provers are favoured over BAN reasoning nowadays. Nevertheless, it generated a lot of interest and development in the field.

2.1.2 Model Checking

Model checking [52] refers to a set of formal methods based on finite state exploration. If one desires to check a model, for instance a security protocol, the first step is usually to create a simplified representation of it. Indeed, since all possible configurations are explored, the initial model is often too complex to be verified in full detail. This limitation is due to insufficient computing power and intrinsic theoretical constraints. Since the set of possible configurations often increases exponentially when the size of the model increases linearly, the only practical solution is to construct a simplified model which is still faithful to the defining characteristics of the initial one.

While model checkers can often help detecting flaws, they are not capable of guaranteeing the security of a protocol since only a simplified version of it is analysed. Symbolic model checking can reduce computational requirements, but ultimately it only provides information about the security of a simplified version of the protocol under scrutiny.

2.1.3 Strand Spaces

In [118], Thayer et al. introduce *strand spaces*. A strand is the sequence of messages sent and received by a single principal during a single protocol run. There is a specific strand for the attacker, capturing her (Dolev-Yao) abilities. A strand space is then the collection of all strands for a given protocol. *Bundles* are sets of communicating strands and "portions" of a strand space. A strand space is seen as a graph of nodes, where the nodes are network events. The security proofs are done by induction. Authentication is proven using a specific technique, called *authentication test*. Automation of the strand spaces method is provided in subsequent work by Song et

al [115]. Termination is not guaranteed.

2.1.4 Process Calculi and Horn Clauses

Combining process calculi with the representation of protocols as Horn clauses (finite disjunctions with one or less positive atoms) has recently proven to be a fruitful research direction. More precisely, cryptographic extensions of process calculi are used, with the applied pi calculus [1] being most typical.

Observational equivalence is a key notion for analysis of indistinguishability using process calculi. It can intuitively be described as follows: two processes P and Q are observationally equivalent if no observer process O can tell P and Q apart. In the language of process calculi, this is defined as the fact that for any process O, "the processes $P||O$ and $Q||O$ are equally able to emit on a given channel" [38]. We will provide a comparison between our approach and process calculi-based ones when looking at electronic voting security analysis.

Static Analysis In [28], Bodei et al. define a new process algebra, loosely based on the pi calculus but without channels. The focus is on the verification of authentication and secrecy via static analysis based on control flow analysis — originally a technique to approximate values during the execution of software written in functional languages. Over-approximation is used.

2.1.5 Interactive Theorem Proving

Theorem proving uses mathematical reasoning to determine if a protocol reaches its security goals. In most cases, the protocol need not be simplified, unlike in the case of model checking. One drawback is that significant human assistance is required, hence the name of *interactive* theorem proving. After our presentation of existing tools using the different approaches outlined in this section, we will focus in Chapter 3 on Isabelle, an interactive theorem prover suitable for security protocol correctness checking.

2.1.6 Automated Theorem Proving

While the theorem proving approach we will rely on is based on higher-order logic, first-order logic methods such as the ones used by Meadows [89] or Weidenbach [122] must be noted. Note that the SPASS tool used in [122] can be used with the Sledgehammer component in the Isabelle/HOL interactive theorem prover that we will introduce soon.

2.2 Tools for the Analysis of Security Protocols

2.2.1 FDR, FDR2 and CSP

The model checker FDR [104] was successfully used by Lowe [83] to find flaws in security protocols. The CSP process calculus [68] is used to model each principal as a process. An intermediate language, Casper [84], is compiled into CSP before being checked by FDR. Ryan and Schneider's book [105] is authoritative on the subject. As the state space has to be finite for model checking, protocols have to be simplified or restricted to be checked with FDR. For instance, for the verification of the Needham-Schroeder public key protocol (see §1.4.1), a single initiator and a single responder are assumed.

The tool has been continuously developed since its creation, and is now a commercial tool even though it is still freely available for academic use. It has been renamed to FDR2 [56].

2.2.2 AVISPA and the AVANTSSAR Platform

AVISPA [6], standing for Automated Validation of Internet Security Protocols and Applications, is a recent "push-button" automatic protocol verification tool. Protocols and conjectured properties must be specified in a formal language called HLPSL. It is based on a number of backends that use different analysis techniques, including model-checkers and tree automata [96]. The On-The-Fly Model-Checker (OFMC) [11], one of the backends, can also be used independently. Some of these backends, such as pro-

tocol falsification ones, operate on a bounded number of sessions. Abstraction-based verification [7] considers unbounded sessions. A substantial number of results have been made available in an online library (`www.avispa-project.org`). Since the end of the European AVISPA project, the same line of work has been continuing under the name of the AVANTSSAR platform [5] (`www.avantssar.eu`).

2.2.3 The NRL Protocol Analyzer and Maude-NPA

Meadow's NRL Protocol Analyzer (NPA) [89] is a versatile, hybrid interactive tool, part model checker, part theorem prover, based on unification. Security goals are proven by showing, via backwards search, that *insecure states* (configurations breaking the goals) cannot be reached. Its successor, Maude-NPA [53], can deal with some algebraic properties of cryptosystems, such as exponentiation and homomorphic encryption. Sequential protocol composition is also supported [54].

2.2.4 Scyther

Cremer introduced Scyther [41], a recent "push-button" tool. Like NPA and Maude-NPA, it assumes that a security property is violated and then checks if backwards search can lead to an admissible initial state. A pattern refinement algorithm is used: the role of each principal is characterised by a small number of execution patterns, which are classes of traces. See also §4.1 about Scyther's capabilities for analysing composed protocols.

2.2.5 LySatool

The LySatool [31] is based on the static analysis framework described earlier [28]. It is automated, and analyses protocols modelled in the LySa process calculus. A tool called Elyjah [97] makes it possible to directly analyse protocols implemented in java by translating them into LySa.

2.2.6 ProVerif and AKiSs

Automated tools such as ProVerif [22] or, more recently, AKiSs [36] can be used to assist with process equivalence analysis. ProVerif is used to check protocols represented by processes modelled in the applied pi calculus. It does not restrict the number of protocol sessions. A stronger condition than observational equivalence between processes is checked. Since the correctness criterion is an under-approximation, spurious attacks may be found in some cases. There is no risk for flawed protocols to be deemed correct, but correct protocols may be deemed flawed by the tool because of the approximation. Extensions have been created for ProVerif to support equational theories using XOR [79] and Diffie-Hellman exponentiation [78].

Automated analysis of voter privacy for electronic voting protocols Various approaches to checking voter privacy have been presented. Notably, Kremer and Ryan [76] presented an analysis with some manual parts. In 2008 [46], a fully automatic verification was done. However, a translation algorithm was used without formal proof of correctness. The next year, Delaune, Kremer and Ryan published a detailed analysis in which the number of voters is fixed, with a partially automated privacy proof [44]. New cryptographic primitives can be added easily to the tool via equational theories, but in some cases the resulting processes fail to terminate.

AKiSs, the most recent automated tool able to check privacy automatically, is also based on equivalence properties. However, a new kind of cryptographic process calculus is used and a different type of process equivalence is checked, called trace equivalence. Under- and over-approximations of trace equivalence are used to detect flawed protocols and verify correct ones, respectively. These equivalences are coarser and more fine-grained variants of trace equivalence. For detailed technical definitions, refer to [36].

The set of supported cryptographic primitives is broader than in ProVerif. For a specific class of processes, called *determinate*, a precise verification can be done. However, not all e-voting protocols fall in this class, in which case one of the approx-

imations must be used. The number of sessions must be bounded as it has critical impact on the computational cost.

2.2.7 tamarin

Another recent tool, tamarin [108], supports automatic, unbounded verification and falsification of protocols using Diffie-Hellman exponentiation. It is based on a constraint-solving algorithm. Equational theories are used to specify cryptography. Termination is not guaranteed. Unlike tools like ProVerif, tamarin is well-suited to the analysis of stateful protocols.

2.2.8 Tool Synthesis

Table 2.1 compares a number of key characteristics of the tools discussed in this section.

2.3 Discussion

In this chapter, we have given a review of existing security protocol analysis approaches and tools. A wide diversity exists, but the current trend is clearly in favour of process-calculi based methods and composite, automated tools. More and more of them support more exotic constructs such as XOR or Diffie-Hellman exponentiation [108]. The support for composed protocols analysis is spreading too, but most tools are restricted to sequential composition [54] or to modular protocols that do not interact strongly [41].

We will go down the road of interactive theorem proving. While it normally requires more analyst interaction, we intend to exploit its greater flexibility to apply an existing framework to new problems. The next chapter introduces the Isabelle interactive theorem prover and provides a detailed presentation of its application to security protocol verification, using higher-order logic.

	Isabelle	FDR2	AVANTSSAR	Maude-NPA	Scyther	LySatool	ProVerif	AKiSs	tamarin
Interactive	✓	×	×	×	×	×	×	×	×
Unbounded verification	✓	×	✓	✓	✓	✓	✓	×	✓
XOR support	×	×	✓	✓	×	×	✓	×	✓
Analysis of composed protocols	✓	×	✓	✓	✓	×	×	×	×
Currently under development	✓	✓	✓	✓	✓	×	✓	✓	✓

Table 2.1: Comparison of security protocol analysis tools

Chapter 3

Isabelle/HOL and the Inductive Method

We now turn to the approach that will be used for the remainder of this thesis: the Isabelle theorem prover and the Inductive Method for security protocol verification.

Isabelle is the latest descendant of a long line of theorem provers. A timeline of this family, starting with Milner's Logic for Computable Functions (LCF), is provided by Gordon in [64]. In the early seventies, a team led by Milner developed the LCF system, based on a type of λ-calculus introduced by Scott a few years earlier [110]. [1] The ML functional programming language was also developed as part of that effort, as a *M*eta*L*anguage of the LCF prover language. During the eighties, successive versions of HOL were developed internally at Cambridge. Independent versions emerged later, such as HOL-4 and HOL Light. Isabelle, introduced in 1989 by Paulson [98], uses theory libraries. Among them, the Higher-Order Logic (HOL) library is based on Cambridge's HOL system; LCF libraries are also available.

[1] Scott's paper was written in 1969 but only published in 1993.

3.1 Isabelle/HOL

Isabelle is a generic interactive theorem prover with a wide range of applications. It has been used for security protocol verification, but also to prove substantial mathematical theorems, for instance the prime number theorem stating $\lim_{x\to\infty} \frac{\pi(x)}{x/\ln(x)} = 1$ (where $\pi(x)$ is the prime-counting function) as well as fundamental results in logic. It is implemented in ML and features a metalogic supporting a variety of object logics. In this thesis, we will always talk about Isabelle/HOL, that is Isabelle with the higher-order logic object logic, which makes quantification over predicates possible.

A *theory* is a file written in the *Intelligible semi-automated reasoning* (Isar) proof language. It contains the detailed steps used by Isabelle to prove theorems. Isabelle comes with a large set of theories, i.e. existing proofs; those theory files are arranged in a hierarchy where each node is able to use results proven in parent nodes.

User interaction is needed to specify and prove theorems, even though a number of automatic tools are available. Isabelle's Proof General interface builds on Emacs and includes the X-Symbol package, which automatically formats token commands such as `\<union>` for display (in this case, $\cup$). We will always present the formatted version of Isar code. Our theory files are written for the Isabelle2011-1 release.

Sledgehammer Isabelle/HOL includes a component, called Sledgehammer [24], allowing the use of Automatic Theorem Provers (ATPs) and satisfiability-modulo-theories (SMT) solvers on a subgoal. The subgoal is first translated automatically into untyped first-order logic. Once a proof using the ATPs is found, it is reconstructed in the *metis* proof method and minimised. The ATPs that can currently be invoked through Sledgehammer include the E Equational Theorem Prover [109], SPASS [123], VAMPIRE [101] and WALDMEISTER [67]. CVC3 [9], Yices [49] and Z3 [43] are the available SMT solvers. If internet access is available, the ATPS can be used through System on TPTP [116] (Thousands of Problems for Theorem Provers), which allows remote simultaneous querying of many ATPs. The three cited SMT solvers can also be queried remotely, though on a different server. The possibility of combining Sledge-

hammer with SMT solvers is recent and remarkably efficient [26].

Nipick Another handy component in Isabelle/HOL is the Nitpick [27] counter-example finder. It can be invoked on subgoals or on entire lemmas. Since infinite types admit no finite model, Nitpick only inspects an approximation. False counter-examples may then be found. Nitpick uses the Kodkod backend, which builds on a SAT solver. Its user's guide is well-detailed [25].

3.2 The Inductive Method

The Inductive Method, created in 1996, is the application of Isabelle to the verification of security protocols. It was first applied to small protocols such as Needham-Schroeder, then developed to real-size ones: Kerberos version 4 and 5, the SET protocol suite, the Shoup-Rubin smartcard protocol and others. See Table 3.1 for a full list of protocols verified in the Inductive Method.

The main idea of the Inductive Method is that by simple mathematical induction, it is possible to model security protocols and reason about their goals.

The *Message* theory in the *Auth* folder provides the starting point for verifying security protocols.

3.2.1 Main Components

Let us first describe the atomic elements used in the Inductive Method.

Agents Agents are seen as processes. This abstracts away security issues such as security holes at the human / computer interaction level: agents are not seen as actual users.

Not only is the number of agents unbounded: every agent is also able to interleave any number of protocol sessions. Since proofs are carried out inductively, susceptibility to replay attacks is taken into account; this kind of flaw detection was actually one of the first practical results of the method. Agents can be either the Server (a

trusted agent which knows all shared keys and is never compromised), a friendly agent (identified by a natural number) or the Spy, who embodies the threat model:

datatype

$agent = Server \mid Friend\ nat \mid Spy$

Compromised agents belong to the *bad* set. The Spy is always in *bad* and the Server never is:

consts

$bad \quad :: agent\ set$

specification (bad)

$Spy\_in\_bad \quad [iff]: Spy \in bad$

$Server\_not\_bad\ [iff]: Server \notin bad$

Messages Messages in the Inductive Method are in practice finite sets built from the following elements:

- Agent names
- Guessable integers
- Unguessable integers (nonces)
- Cryptographic keys (symmetric or asymmetric, public or private)
- Message digests (hashes)
- Pairs of messages (this can be reused to form compound messages of any length)
- Ciphertexts, taking as arguments a cryptographic key and a message

Here is the corresponding code, to be found in the *Message* theory:

datatype

$msg = Agent\ \ agent$

$\mid Number\ nat$

| *Nonce nat*

| *Key key*

| *Hash msg*

| *MPair msg msg*

| *Crypt key msg*

Keys Cryptographic keys in Isabelle have the type *key* and are modelled as natural numbers that cannot be guessed:

type_synonym *key* = *nat*

The *invKey* function is an involution.[2] Its restriction to the domain of symmetric keys is equal to the identity function. It is useful to specify the properties of key pairs for asymmetric cryptography (see below).

invKey :: *key* ⇒ *key*

specification (*invKey*)

invKey [*simp*]: *invKey* (*invKey K*) = *K*

invKey_symmetric: *all_symmetric* ⟶ *invKey* = *id*

Symmetric keys are part of the set *symKeys*. *shrK* is a long-term key, shared between an agent and the Server. Symmetric keys are either shared keys (between peers) or session keys.

definition *symKeys* :: *key set* **where**

symKeys ≡ {*K*. *invKey K* = *K*}

consts

all_symmetric :: *bool* — true if all keys are symmetric

In the case of asymmetric keys, for encryption, key pairs are modelled by *priEK* and *pubEK*. For signature, *priSK* and *pubSK*:

datatype *keymode* = *Signature* | *Encryption*

[2] A function f such that $f \circ f = \mathrm{id}$.

consts

publicKey :: [*keymode*,*agent*] ⇒ *key*

abbreviation

pubEK :: *agent* ⇒ *key* **where** *pubEK* ≡ *publicKey Encryption*

pubSK :: *agent* ⇒ *key* **where** *pubSK* ≡ *publicKey Signature*

invKey turns an asymmetric key into its associated reciprocal key, for instance *invKey(priEK A) = pubEK A*:

privateKey :: [*keymode*, *agent*] ⇒ *key* **where** *privateKey b A* ≡ *invKey* (*publicKey b A*)

priEK :: *agent* ⇒ *key* **where** *priEK A* ≡ *privateKey Encryption A*

priSK :: *agent* ⇒ *key* **where** *priSK A* ≡ *privateKey Signature A*

When the distinction is not needed, *priK* and *pubK* are used:

pubK :: *agent* ⇒ *key* **where** *pubK A* ≡ *pubEK A*

priK :: *agent* ⇒ *key* **where** *priK A* ≡ *invKey* (*pubEK A*)

Generally speaking, encryption and signature keys are specified differently so distinction between keys with different purposes in a protocol is possible.

Message operators The *parts* operator, defined inductively, takes as input a message set and returns another message set. The output set consists of the initial set augmented with all the elements it contains, up to atomic ones. Even encrypted elements for which the decryption key is not available appear in the resulting set. *parts* can therefore be seen as the total information that an adversary with unlimited computing power could extract from a message set. More pragmatically, it is used to denote all elements appearing in network traffic. Its inductive definition from the *Message* theory is as follows:

inductive_set

parts :: *msg set* ⇒ *msg set*

for *H* :: *msg set*

where

Inj [*intro*]: $X \in H \Longrightarrow X \in parts\ H$

| *Fst*: $\{\!|X,Y|\!\} \in parts\ H \Longrightarrow X \in parts\ H$

| *Snd*: $\{\!|X,Y|\!\} \in parts\ H \Longrightarrow Y \in parts\ H$

| *Body*: $Crypt\ K\ X \in parts\ H \Longrightarrow X \in parts\ H$

Note that cryptographic keys that appear solely as encrypting keys, while not appearing in any message body, are not extracted. In other words, $K \in parts\ (Crypt\ K\ X)$ if and only if $K \in parts\ X$.

The *analz* operator bears close resemblance to the one we just described, but takes into account cryptography. It also takes as input a message set, and returns the set of everything that can be decrypted using cryptographic keys present in the set.

inductive_set

analz :: *msg set* ⇒ *msg set*

for *H* :: *msg set*

where

Inj [*intro*,*simp*] : $X \in H \Longrightarrow X \in analz\ H$

| *Fst*: $\{\!|X,Y|\!\} \in analz\ H \Longrightarrow X \in analz\ H$

| *Snd*: $\{\!|X,Y|\!\} \in analz\ H \Longrightarrow Y \in analz\ H$

| *Decrypt* [*dest*]: $[\![Crypt\ K\ X \in analz\ H; Key(invKey\ K) \in analz\ H]\!] \Longrightarrow X \in analz\ H$

As opposed to *parts* and *analz*, the *synth* operator specifies message building rather than message deconstruction. Taking into account available cryptographic keys, it inductively defines the set of messages that can be built from an initial message set. Ciphertexts and message hashes can be generated from available messages. Agent names and guessable numbers can be synthesised ex nihilo.

inductive_set

synth :: *msg set* ⇒ *msg set*

for *H* :: *msg set*

where

Inj [*intro*]: $X \in H \Longrightarrow X \in synth\ H$

| *Agent* [*intro*]: $Agent\ agt \in synth\ H$

| *Number* [*intro*]: *Number n* ∈ *synth H*

| *Hash* [*intro*]: *X* ∈ *synth H* ⟹ *Hash X* ∈ *synth H*

| *MPair* [*intro*]: ⟦*X* ∈ *synth H*; *Y* ∈ *synth H*⟧ ⟹ {|*X*,*Y*|} ∈ *synth H*

| *Crypt* [*intro*]: ⟦*X* ∈ *synth H*; *Key*(*K*) ∈ *H*⟧ ⟹ *Crypt K X* ∈ *synth H*

Initial knowledge Initial knowledge is data that is available to agents before any protocol has even begun. It consists of the keys that can reasonably be assumed known to various protocol participants. Usually, every agent knows all public encryption and signing keys, his own private encryption and signing keys and his shared key. The Server additionally knows *all* shared keys. The Spy additionally knows the private encryption and signing keys and shared keys of all compromised agents.

primrec *initState* **where**

initState_Server:

initState Server =

{*Key* (*priEK Server*), *Key* (*priSK Server*)} ∪

(*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*) ∪ (*Key* ‘ *range shrK*)

| *initState_Friend*:

initState (*Friend i*) =

{*Key* (*priEK*(*Friend i*)), *Key* (*priSK*(*Friend i*)), *Key* (*shrK*(*Friend i*))} ∪

(*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*)

| *initState_Spy*:

initState Spy =

(*Key* ‘ *invKey* ‘ *pubEK* ‘ *bad*) ∪ (*Key* ‘ *invKey* ‘ *pubSK* ‘ *bad*) ∪

(*Key* ‘ *shrK* ‘ *bad*) ∪ (*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*)

Events Three event types are initially available:

datatype

event = *Says agent agent msg* | *Gets agent msg* | *Notes agent msg*

The *Says* event models the sending of a message between agents. For instance, *Says A B* {*Nonce Na*} means that agent *A* sends agent *B* a message consisting solely

of the nonce *Na*. It is realistic to assume that not all messages reach their destination, hence message sending does not imply message reception in general. However, a sent message *can* be delivered as intended; this is modelled by the *Gets* events, which precisely represents the reception of a message by an agent. The *Notes* event models the addition of a message to an agent's knowledge, for later use. It can be seen as a private recording of information.

Knowledge The events we just described shape the knowledge of agents. The knowledge that an agent gains from a given event list is modelled as follows:

```
primrec knows :: agent ⇒ event list ⇒ msg set
where
 knows_Nil:  knows A [] = initState A
| knows_Cons:
   knows A (ev # evs) =
     (if A = Spy then
      (case ev of
        Says A' B X ⇒ insert X (knows Spy evs)
       | Gets A' X ⇒ knows Spy evs
       | Notes A' X  ⇒  if A' ∈ bad then insert X (knows Spy evs) else knows Spy evs)
      else
      (case ev of
        Says A' B X ⇒  if A'=A then insert X (knows A evs) else knows A evs
       | Gets A' X   ⇒  if A'=A then insert X (knows A evs) else knows A evs
       | Notes A' X  ⇒  if A'=A then insert X (knows A evs) else knows A evs))
```

It can be seen that the *knows* function accounts both for the knowledge of the Spy and for the knowledge of ordinary agents. The sharp sign # signifies concatenation. In the case of ordinary agents, their knowledge only changes if they send, receive or note something themselves. On the other hand, the Spy learns everything that is sent by anybody over the network. *Gets* messages do not influence the Spy's knowledge to prevent redundancy; not every *Says* is followed by a *Gets*, but there can be no

Gets without a corresponding prior *Says*. Indeed, *Gets* events can only come from the *Reception* rule that we will present soon (§3.2.1). All *Notes* events from compromised agents augment the Spy's knowledge, in line with the idea that they share what they learn with her. The abbreviation *spies* stands for *knows Spy*.

Freshness The *used* function specifies freshness of message components. A component is defined as being fresh if:

1. It is not part of anyone's initial knowledge;
2. It was not sent as part of any message;
3. It was not noted as part of any message.

primrec *used* :: *event list* ⇒ *msg set*
where
used_Nil: *used* [] = (⋃ *B*. *parts* (*initState B*))
| *used_Cons*: *used* (*ev* # *evs*) =
(*case ev of*
Says A B X ⇒ *parts* {*X*} ∪ *used evs*
| *Gets A X* ⇒ *used evs*
| *Notes A X* ⇒ *parts* {*X*} ∪ *used evs*)

Event lists (traces) One key concept is that of an event list, called *trace*. It can be seen as the history of network events, i.e. the list of messages exchanged between peers. Events are listed in reverse chronological order. The agent population is considered unbounded (mapped to natural numbers).

The set of all admissible traces under a specific protocol represents the formal protocol model. Proofs are carried out by induction on a generic trace of this model. Isabelle provides the mechanical support. It is semi-automatic, or interactive: the user has to specify goals and strategies to reach them, but Isabelle works out the details and provides easier subgoals if the methods provided are not sufficient to reach the ultimate goal.

The formal protocol model Every protocol step is modelled as an inductive rule, with preconditions and a postcondition. The protocol model is the set of all admissible traces built from those steps. The empty trace is also allowed, and modelled by the *Nil* event. Furthermore, a *Fake* rule accounts for message forgery by the Spy.

For a concrete example, we now take a look at the model of a classic protocol, BAN Kerberos. This is the abstract model of Kerberos that was analysed by Burrows, Abadi and Needham in [32], not a deployed version.

inductive_set *bankerb_gets* :: *event list set*

This defines the name of the inductive model; here, it is called *bankerb_gets*. *event list* is the type of an event trace. *bankerb_gets* is hence a set of event traces. A trace is modelled as an event list, and therefore the set of all these traces is a model for the protocol. *BKi* is the rule corresponding to the i-th protocol step.

Nil: $[] \in$ *bankerb_gets*

This is the "base case" of the inductive model; it merely means that the empty trace, that is to say a network history with no events, is part of the model — i.e., of possible traces.

| *Reception*:
$[\![evsr \in bankerb\_gets; Says\ A\ B\ X \in set\ evsr]\!] \Longrightarrow Gets\ B\ X \#\ evsr \in bankerb\_gets$

The *Reception* rule reflects the possibility for sent messages to be received.

| *Fake*: $[\![evsf \in bankerb\_gets;\ X \in synth\ (analz\ (knows\ Spy\ evsf))]\!]$
$\Longrightarrow Says\ Spy\ B\ X \#\ evsf \in bankerb\_gets$

This important *Fake* rule defines the Spy's properties, i.e. the threat model. The capacities given to the Spy through it are those of the Dolev-Yao model defined in [48].

The operator *analz* formalises breaking up messages into their elements. Hence *analz(spies evsf)* is the set of all message components that the Spy can obtain from observing network traffic. This includes decoding ciphertexts when the key is available to the Spy, but *not* cryptanalysis. Crypto is seen from a black-box point of view

and considered perfect as long as the key does not leak. Furthermore, the operator *synth* formalises the creation of new messages from available components. As a consequence, the set *synth (analz (spies evsf))* is the set of all messages that the Spy can build up from her observation of network events, excluding cryptanalysis. In a nutshell, this *Fake* rule means that if a trace *evsf* is already part of the model, then the trace obtained by inductively concatenating it with messages that the Spy can synthesise from network traffic observation is also part of the model.

It should be noted that in some different settings, e.g. smartcards, the threat model is extended beyond this simple rule: for instance, the Spy can additionally obtain the outputs of illegally usable smartcards [13].

$$|\ \textit{BK1}:\ \textit{evs1} \in \textit{bankerb\_gets} \implies \textit{Says A Server}\ \{\!|\textit{Agent A}, \textit{Agent B}|\!\}\ \#\ \textit{evs1}\ \in\ \textit{bankerb\_gets}$$

This models the first BAN Kerberos protocol step:

$$1.\quad A \longrightarrow S\quad :\quad \{\!|A, B|\!\}$$

In fact, while the above notation suggests that the sending of the message and its reception occur as one event, in the Inductive Method one event stands for the sending of a message, without guarantee of delivery, and another for its possible reception.

Keeping with the spirit of the inductive definition of traces, the concatenation of a trace which is already part of the model with this new protocol step is part of the model also. Since this is the first protocol step, there are no prerequisites.

$$\begin{aligned}|\ \textit{BK2}:\ &[\![\textit{evs2} \in \textit{bankerb\_gets};\ \textit{Key K} \notin \textit{used evs2};\ \textit{K} \in \textit{symKeys};\\ &\textit{Gets Server}\ \{\!|\textit{Agent A}, \textit{Agent B}|\!\} \in \textit{set evs2}]\!]\\ &\implies \textit{Says Server A}\ \ (\textit{Crypt}\ (\textit{shrK A})\\ &\quad \{\!|\textit{Number}\ (\textit{CT evs2}), \textit{Agent B}, \textit{Key K},\\ &\quad (\textit{Crypt}\ (\textit{shrK B})\ \{\!|\textit{Number}\ (\textit{CT evs2}), \textit{Agent A}, \textit{Key K}|\!\})|\!\})\\ &\quad \#\ \textit{evs2} \in \textit{bankerb\_gets}\end{aligned}$$

In this second protocol step, we have two prerequisites. The first one is related to the key K. *Key K* $\notin$ *used evs2* means that we require it to be fresh: K has not been used

before. It is a session key. It is also a symmetric key. The guessable integer *Number (CT evs2)* is a timestamp (see §3.2.4) standing for Current Time, used to control the validity of a session key. Lifetimes for keys and authenticators are defined in the protocol prologue (not shown here); if the difference between the current time and the timestamp is greater than the lifetime, the key or authenticator is considered expired and rejected.

The second prerequisite is the execution of the previous protocol step. There is a subtlety here: instead of requiring that A sent $Server$ the message $\{\!|A, B|\!\}$, we merely ask that the Server received it from *someone*. The justification for this is that in our threat model, the Spy can easily forge messages, and there is no immediate way for $Server$ to check for A's identity — indeed, goals such as authentication can only be provided by the protocol itself. Hence we simply require that an agent — hoping it was A, but not knowing for sure — sent the first protocol message.

```
| BK3: ⟦evs3 ∈ bankerb_gets;
        Gets A (Crypt (shrK A) {|Number Tk, Agent B, Key K, Ticket|}) ∈ set evs3;
        Says A Server {|Agent A, Agent B|} ∈ set evs3; ¬ expiredK Tk evs3⟧
      ⟹ Says A B {|Ticket, Crypt K {|Agent A, Number (CT evs3)|}|} # evs3 ∈ bankerb_gets
```

The postcondition of step three is a *Says* event performed by *A*. Accordingly, *A* can check the preconditions. The first one is *A*'s reception of the message that (presumably) the Server sent in the second step; the second one is the fact that *A* indeed started a session with the Server mentioning the same agent *B* as the one appearing in the *Gets* message. The third precondition is that the timestamp from the *Gets* message has not expired yet. If all these requirements are satisfied, the inductive rule associated with this protocol step enables *A* to start her communication with *B*. The message she sends is built from elements obtained through the *Gets* event; a new timestamp is also included.

```
| BK4: ⟦evs4 ∈ bankerb_gets;
        Gets B {|(Crypt (shrK B) {|Number Tk, Agent A, Key K|}),
                 (Crypt K {|Agent A, Number Ta|})|} ∈ set evs4;
```

$\neg$ *expiredK Tk evs4*; $\neg$ *expiredA Ta evs4*$\rrbracket$
$\Longrightarrow$ *Says B A* (*Crypt K* (*Number Ta*)) # *evs4* $\in$ *bankerb_gets*

Step four of the protocol model is similar, also featuring non-expiration conditions. Here *B* replies to *A*, provided he received a message matching his expectations.

| *Oops*: $\llbracket$*evso* $\in$ *bankerb_gets*;
Says Server A (*Crypt* (*shrK A*) $\{\!|$*Number Tk*, *Agent B*, *Key K*, *Ticket*$|\!\}$) $\in$ *set evso*;
expiredK Tk evso$\rrbracket$
$\Longrightarrow$ *Notes Spy* $\{\!|$*Number Tk*, *Key K*$|\!\}$ # *evso* $\in$ *bankerb_gets*

Finally, the appropriately named "Oops event" models the accidental loss of an expired (*expiredK Tk evso*) session key by an agent — see §3.2.4.

Protocol design principles One goal of security protocol formal analysis is to derive general guidelines which guarantee security. The road towards a set of simple rules directly guaranteeing security is long, but a number of common conceptual mistakes can be avoided reasonably easily. Early "robustness principles" were pointed out by Anderson and Needham [3].

As a concrete example, in the Needham-Schroeder public-key protocol, there is a lack of *explicitness*. Indeed, agent *B* does not quote his own identity in the second protocol message. Explicitness holds when "each message says exactly what it means without ambiguity" (Bella [14]). In other words, protocol steps should include all relevant information, even elements which seem obvious. This often means quoting the identities of all agents involved in a given step, or quoting associations between agents and session keys clearly. As few elements as possible should be left open to agent interpretation. Symbolically, the consequence is additional variable bindings that can prevent flaws due to unnecessary genericity. Interestingly, explicitness also makes developing formal proofs easier.

Another important concept is that of *viewpoint*: a formal guarantee is only useful to an agent if that agent can verify himself if the theorem assumptions hold. For instance, an agent can check whether a ciphertext has been encrypted with his public key, but has

no means of verifying whether the encryption was performed using a different agent's public key. Likewise, an agent knows whether he sent a given message, but cannot know whether the intended recipient actually obtained it. Thus a security property is always achieved *from an agent's viewpoint*. Sometimes the proof can be conducted on assumptions verifiable by one agent, but not by another one. In this case, different conclusions as to its correctness are made depending on whose viewpoint we are considering. This leads to the principle of *goal availability* for protocol analysis [15]. A goal of a security protocol is said to be available to a principal if the protocol's formal model contains a guarantee for the goal, and if the assumptions for that guarantee can be verified within the agent's minimal trust. The principle states that protocols should provide goal availability to its agents. Noticing lack of goal availability can be of significant help for discovering protocol weaknesses.

Other principles include avoiding unnecessary encryption, which does not always increase security, as the Kerberos IV example shows: the subsequent version of the Kerberos protocol disposes of double encryption, yet the main security properties still hold [14]. Synchronising network clocks is important too: in our ideal model, network clocks are considered perfectly coordinated but this is a substantial problem in real-life situations.

The threat model In the Dolev-Yao (DY) model [48], the Spy has total control over network traffic. She can prevent delivery, create new messages from past message components she intercepted during network traffic, note all message contents but cannot cryptanalyse. Nevertheless, the Spy may gain access to an encrypted message if an incautious agent leaks a private key.

Whether non-adherence to protocol design principles such as explicitness implies insecurity of a protocol depends on the chosen threat model. Indeed, the DY model is quite pessimistic; real-life network conditions can be less drastic. For instance, sometimes only eavesdropping is possible but no forgery. The threat model can be defined in Isabelle. For the protocols analysed in [14], the DY model is always used,

with the additional threat that the Spy can collaborate with compromised agents.

3.2.2 Goal Definition and Proving

Protocol goals can be mapped to trace properties: a security property is achieved if it remains true at every step. Indeed, protocol goals are defined as predicates on traces, and must be true for all trace in the inductive model. Their veracity is checked by proving statements about the inductive protocol model.

As an example, consider a proof of secrecy for the Needham-Schroeder public key protocol. This example is presented in [95]. Our goal is to prove that agent A's private key remains secret. This is equivalent to stating that A's private key is known to the Spy if and only if A belongs to *bad* (the set of compromised agents).

lemma *Spy_see_priEK* [*simp*]:

$evs \in ns\_public \Longrightarrow (Key\ (priEK\ A) \in parts\ (spies\ evs)) = (A \in bad)$

apply (*erule ns_public.induct*, *simp_all*)

The bracketed *[simp]* tells Isabelle that the lemma, once proven, shall be saved as a simplification rule; i.e., it will be reused by Isabelle if necessary when invoking simplification. The line starting with *apply* specifies which proving techniques Isabelle should use. In this case, the basic techniques of induction (*induct*) and simplification by rewriting and arithmetic decision (*simp_all*) are used.

This induction does not directly prove the goal, but generates several subgoals to be proven; in this case, one subgoal for each rule in *ns_public*. Since the key is not transmitted during the protocol steps, only the rule *Fake* remains to be proven:

1. $\bigwedge evsf\ X.$

$[\![evsf \in ns\_public;$

$(Key\ (priEK\ A) \in parts\ (knows\ Spy\ evsf)) = (A \in bad);$

$X \in synth\ (analz\ (knows\ Spy\ evsf))]\!]$

$\Longrightarrow (Key\ (priEK\ A) \in parts\ (insert\ X\ (knows\ Spy\ evsf))) = (A \in bad)$

This case is proven automatically by the *blast* automatic prover, a powerful first-order tool of Isabelle's classic reasoner:

apply *blast*

The Isabelle tutorial [95] and reference manual [124] provide full information on syntax and proving techniques.

Proofs are in the "natural" inductive style that would be used manually by humans. A middle ground is to be found between highly streamlined and compact proofs and ones that are more linear and easily readable by people. Hence limited use of automatic proof techniques is preferred to preserve intuition and meaningful semantics. In cases where automatic proof techniques are used, the subgoal that is solved should be in a sufficiently clear form, so as to make the implicit reasoning obvious.

3.2.3 Common Security Property Formalisations

We now review the formalisations of the most common security properties to give an impression of how theorems are expressed in the Inductive Method.

Regularity lemmas Regularity lemmas are statements about the inclusion of a message in network traffic. Since network traffic corresponds to the set *parts (spies evs)*, where *evs* is a trace of the examined protocol model, regularity lemmas state under which conditions a given message is in *parts (spies evs)*.

Spy_see_priEK, which we just saw, is an example of such a lemma: its meaning is that the private encryption key of an agent appears in network traffic if and only if the agent is compromised.

Confidentiality of a message Confidentiality lemmas are statements about the availability of a message to the Spy's knowledge, denoted by the set *analz (spies evs)*. It represents the knowledge that the Spy can learn from a trace, possibly decrypting ciphertexts for which she knows the decryption key.

$evs \in ns\_public \Longrightarrow (Key\ (priEK\ A) \in analz\ (spies\ evs)) = (A \in bad)$

In this example, for a given trace (this will be implied from now on), the private encryption key of an agent is confidential if and only if the agent is not compromised.

Reliability Reliability statements establish that the protocol model works as expected. They often describe causality of events.

$$\begin{aligned}&[\![Says\ A\ B\ \{\!|Ticket, Crypt\ K\ \{\!|Agent\ A, Number\ Ta|\!\}|\!\} \in set\ evs;\\ &\ A \notin bad; evs \in bankerb\_gets]\!]\\ &\Longrightarrow \exists\ Tk.\ Gets\ A\ (Crypt\ (shrK\ A)\ \{\!|Number\ Tk, Agent\ B, Key\ K, Ticket|\!\})\ \in set\ evs\end{aligned}$$

Here, honest agents behave as expected: if a protocol step requires another step as a precondition and the former happened, then the latter happened as well.

Freshness of a message component Freshness lemmas describe under which conditions an element appears for the first time. They often state that a nonce appearing in a particular message was not part of a different message also involving a nonce.

$$\begin{aligned}&evs \in ns\_public \Longrightarrow\\ &\quad Crypt\ (pubEK\ C)\ \{\!|NA', Nonce\ NA, Agent\ D|\!\} \in parts\ (spies\ evs) \longrightarrow\\ &\quad Crypt\ (pubEK\ B)\ \{\!|Nonce\ NA, Agent\ A|\!\} \in parts\ (spies\ evs) \longrightarrow\\ &\quad Nonce\ NA \in analz\ (spies\ evs)\end{aligned}$$

In the above example, if the nonce *NA* is confidential, then it can not be reused for some specific protocol steps. In other words, if the nonce appears in the two cited messages, at the specified spots, then the nonce is known to the Spy.

Unicity of a message up to multiple identical messages This kind of statement clarifies under which conditions messages with similar structures possess identical elements.

$$\begin{aligned}&[\![Crypt(pubEK\ B)\ \ \{\!|Nonce\ NA, Agent\ A\ |\!\} \in parts(spies\ evs);\\ &\ Crypt(pubEK\ B')\ \{\!|Nonce\ NA, Agent\ A'|\!\} \in parts(spies\ evs);\\ &\ Nonce\ NA \notin analz\ (spies\ evs); evs \in ns\_public]\!]\\ &\ \Longrightarrow A{=}A' \wedge B{=}B'\end{aligned}$$

If two messages with the quoted structure and with the same nonce appear in traffic, then either their other components are identical or the nonce is not confidential.

Strong unicity Sometimes, we want to express the fact that a given message only appears once. This strong unicity does not hold if multiple identical messages appear. For this, we use the dedicated *Unique* predicate:

$$\begin{array}{l}[\![Says\ Server\ A\ (Crypt\ (shrK\ A)\ \{\!|Number\ Tk, Agent\ B, Key\ K, Ticket|\!\})\in set\ evs;\\ \ evs\in bankerb\_gets]\!] \Longrightarrow\\ Unique\ Says\ Server\ A\ (Crypt\ (shrK\ A)\ \{\!|Number\ Tk, Agent\ B, Key\ K, Ticket|\!\})\ on\ evs\end{array}$$

Authentication of an agent Authentication of an agent is proven by showing that if a message with a specific structure appeared at all, it could only have been sent by that agent.

$$\begin{array}{l}[\![A\notin bad;\ B\notin bad;\ evs\in ns\_public]\!] \Longrightarrow\\ \ Crypt\ (pubEK\ A)\ \{\!|Nonce\ NA, Nonce\ NB, Agent\ B|\!\}\in parts\ (spies\ evs)\longrightarrow\\ \ Says\ A\ B\ (Crypt(pubEK\ B)\ \{\!|Nonce\ NA, Agent\ A|\!\})\in set\ evs\longrightarrow\\ \ Says\ B\ A\ (Crypt(pubEK\ A)\ \{\!|Nonce\ NA, Nonce\ NB, Agent\ B|\!\})\in set\ evs\end{array}$$

Here, assuming the involved agents act legally, if the message *Crypt (pubEK A)* $\{\!|$*Nonce NA, Nonce NB, Agent B*$|\!\}$ appears in the traffic at all, and if the message *Crypt (pubEK B)* $\{\!|$*Nonce NA, Agent A*$|\!\}$ was sent from A to B, then the former message was sent by B to A (and not by anyone else).

Order of events Statements about chronology are also possible; specifically, we can extract subtraces of traces by only considering the part of the trace up to a given event.

Key Kab $\notin$ *used (before (Says Server A (Crypt K* $\{\!|$ *Na, Agent B, Key Kab, X* $|\!\}$ *)) on evs)*

At the moment the Server issues the key *Kab*, it is still fresh. This uses a special function, *before*, which maps an event and a trace to the trace containing all events happening before said one.

Fair non-repudiation This kind of property is useful for protocols that ensure that two agents performing it either both obtain what they seek, or none of them do.

Quoting from [14], *If evs is a generic trace, NRO binds A to the sending of message m and NRR binds B to the reception of message m, then*

$NRO \in analz\ (knows\ B\ evs) \Leftrightarrow NRR \in analz\ (knows\ A\ evs)$

Notice that the lemma (unlike most others) includes no assumptions about agent honesty. However, in practice, one would not prove such an equivalence directly. We would rather establish for each agent a result such as *A_fairness_NRO* for the Zhou-Gollmann protocol [126]:

$$\begin{aligned}
&[\![con_K \in used\ evs;\ NRO \in parts\ (spies\ evs);\\
&\ con_K = Crypt\ (priK\ TTP)\ \{\!|Number\ f_con, Agent\ A, Agent\ B, Nonce\ L, Key\ K|\!\};\\
&\ NRO = Crypt\ (priK\ A)\ \{\!|Number\ f_nro, Agent\ B, Nonce\ L, Crypt\ K\ M|\!\};\\
&\ NRR = Crypt\ (priK\ B)\ \{\!|Number\ f_nrr, Agent\ A, Nonce\ L, Crypt\ K\ M|\!\};\\
&\ A \notin bad;\ evs \in zg]\!]\\
&\ \Longrightarrow Gets\ A\ \{\!|Number\ f_nrr, Agent\ A, Nonce\ L, NRR|\!\}\ \in set\ evs
\end{aligned}$$

Assume that A is honest. B does not have to be. If non-repudiation of origin *NRO* (an item promised to B) and another element *con_K* exist in the trace at all, then A received his promised non-repudiation of receipt item *NRR* at some stage. This guarantees fairness to A.

Key distribution Key distribution is specified by asserting that a key is in the *analz* set of the knowledge of the agent it is distributed to.

$$\begin{aligned}
&[\![Says\ A\ B\ \{\!|Nonce\ M, Agent\ A, Agent\ B,\\
&\qquad Crypt\ (shrK\ A)\ \{\!|Nonce\ Na, Nonce\ M, Agent\ A, Agent\ B|\!\}\ |\!\} \in set\ evs;\\
&\ Gets\ A\ \{\!|Nonce\ M, Crypt\ (shrK\ A)\ \{\!|Nonce\ Na, Key\ K|\!\}\ |\!\} \in set\ evs;\\
&\ A \notin bad; B \notin bad; evs \in orb]\!]\\
&\ \Longrightarrow Key\ K \in analz\ (knows\ B\ evs)
\end{aligned}$$

Validity of evidence Validity of evidence means that "an agent is given evidence sufficient to convince a third party of his peer's participation in the protocol" [14]. In practice, if an element was used at some stage, then it was used by a specific agent. Statements like this can be seen as a special kind of agent authentication.

$[\![con\_K \in used\ evs;$
$con\_K = Crypt\ (priK\ TTP)\ \{\!|Number\ f\_con, Agent\ A, Agent\ B, Nonce\ L, Key\ K|\!\};$
$evs \in zg]\!]$
$\Longrightarrow Notes\ TTP\ \{\!|\ Number\ f\_con, Agent\ A, Agent\ B, Nonce\ L, Key\ K, con\_K\ |\!\}\ \in set\ evs$

If a given term (here, *con_K*) exists at all in a trace, then it is known to *TTP*:

3.2.4 Existing Extensions to the Inductive Method

Since its creation, the Inductive Method has been extended in a number of directions. The following overview roughly respects the chronological order of those developments.

Higher-level protocols Some security protocols build on existing protocols. Abadi *et al.*'s certified e-mail protocol [2], for instance, uses SSL. The strategy adopted by Bella with the Inductive Method is reducing those protocols to black boxes and modelling their goals. To model authentication without confidentiality, preconditions akin to *Says B A X* are used; A can authenticate the sender, but the Spy can still intercept *X*.

To be more precise, consider a protocol step modelling the sending of a message by *A*. Assume her peer *B* is mentioned somewhere in the preconditions of the step. Assume a precondition involving a message that *A* received. Normally, this precondition would be a *Gets A X* event: *A* knows what she received, but has no way of determining the sender. In the case of an authenticated channel, this precondition is a *Says* event binding the sender to an agent that *A* already knows: here, agent *B*. Hence the precondition has the form *Says B A X*.

For confidentiality, the *Notes* event is used with three arguments, twice. For instance, the event $A \xrightarrow{\text{SSL}} B \ : \ M$ is formalised in two steps. First, the transmission of the message is modelled by *Notes A (Agent A, Agent B,M)*. This is written as a precondition to the message's *Reception* rule, which includes the event *Notes B (Agent A, Agent B,M)*.

Timestamps Timestamps are modelled by a discrete formalisation of time, corresponding to the position of an event in the trace; there is thus an implicit global clock. The empty trace is associated with the time zero; then, every event increases the clock by one unit. Since traces are concatenations of non-simultaneous events, the result is a timeline without ambiguity.

A message component is considered *fresh* if $t - t_* < l$, where t_* is the time of creation, t is the current time and l is the component's lifetime. A non-fresh message component is said to be *expired*. Session keys are considered *valid* exactly within their lifetime; afterwards, they are also called expired.

Physical properties Basin and his group [10, 107] extended the Inductive Method to support the verification of *physical* protocols, i.e. protocols that "establish properties of the network environment" such as physical position and time synchronisation. The framework was then applied to the analysis of distance bounding protocols and the TESLA broadcast authentication protocol.

Multicast protocols The *event* datatype was extended by Martina [85] to allow the specification of multicast (one-to-many) protocols. The rest of the framework, including user knowledge, was adapted to account for this extension. The result can be seen as a superset of what was previously available: unicast protocols are still specifiable using the resulting theories, and so are protocols mixing different casting modes.

Secret sharing In the same monograph, Martina presented a formalisation of threshold cryptography. The *Nonce* message element is used by splitting the set of nonces in three disjoint classes: shares, session identifiers and standard freshness tokens. In combination with the specification of multicast, this extension is then used to verify the Franklin-Reiter auction protocol, a byzantine agreement one.

Provable anonymity Li and Pang [81] analysed the onion routing protocol [100] using a definition of provable anonymity due to Garcia et al. [60]

Ceremonies Security ceremonies [51] are the natural generalisation of security protocols: what is out of band for a protocol is considered part of ceremonies involving it. In particular, user behaviour is a crucial part of security ceremonies. Their analysis can therefore be seen as a socio-technical problem. Bella [18] provides a formal framework in Isabelle/HOL for the human-computer interaction layer, in the spirit of the Inductive Method, and applies it to an Amazon privacy ceremony.

3.2.5 Protocols Verified So Far

The application of interactive theorem proving to security protocols, via Isabelle, was started by Paulson in 1996. Table 3.1 summarises, to the best of our knowledge, all published protocol verifications in this framework as of February 2012. NS means Needham-Schroeder.

Protocol	**Class**	**Year**	**Author(s)**
Yahalom	Key sharing, authentication	1996	Paulson
NS symmetric	Key sharing	1996	Paulson & Bella
Otway-Rees (with variants)	Authentication	1996	Paulson
Woo-Lam	Authentication	1996	Paulson
Otway-Bull	Authentication	1996	Paulson
NS asymmetric	Authentication	1997	Paulson
TLS	Multiple[1]	1997	Paulson
Kerberos IV	Mutual authentication	1998	Bella
Kerberos BAN	Mutual authentication	1998	Paulson & Bella
SET suite	Multiple[1]	2000+	Bella *et al.*
Abadi *et al.* certified e-mail	Accountability	2003	Bella *et al.*
Shoup-Rubin smartcard	Key distribution	2003	Bella
Zhou-Gollmann	Non-repudiation	2003	Paulson & Bella
Kerberos V	Mutual authentication	2007	Bella
TESLA	Broadcast authentication	2009	Schaller *et al.*
Meadows distance bounding	Physical	2009	Basin *et al.*
Multicast NS symmetric	Key sharing	2011	Martina
Franklin-Reiter	Byzantine	2011	Martina
Onion routing	Anonymising	2011	Li & Pang

Table 3.1: Security protocols verified in Isabelle so far

[1] Confidentiality, integrity, authentication

3.3 Discussion

We presented Isabelle/HOL and introduced all important features of the Inductive Method. Typical security property formalisations and a review of extensions to the Inductive Method were also provided. The length of this chapter mirrors the steep learning curve faced by new practitioners; a couple of months are typically required to be able to use it productively. However, the diversity of the protocols and goals that have been tackled successfully in this framework make the case for its continued use and application to new topics. The three next chapters deal with our contributions and all extend the Method in various ways.

Chapter 4

Protocol Composition Analysis Applied to Public Key Infrastructure

While the formal analysis of security protocols is mature and the quest towards making it fully automatic is progressing, a number of situations are not yet supported by automatic tools. In such cases, more interactive tools such as the Inductive Method can provide a solution. We start with the problem of verifying composed protocols holistically.

4.1 Security Protocol Composition

The challenges of protocol analysis are still significant, at least upon those protocols that are run *with* other protocols. In particular, security protocols are typically sequenced, as is the case of protocols for public-key registration, certification and actual use. They are often executed on top of one another in a stack fashion, as is the case of an SSL session taking place over an IPSec channel. Protocols may also be interleaved, perhaps with malicious aims, as is the case of a purchase transaction that is entwined with a banking session. We advocate that protocol analysis become more precise

(hence more reliable) upon these protocols, and therefore make explicit a number of preconditions whose validity is taken for granted.

We call *mix protocol* any protocol built by composition through protocol sequencing, stacking or interleaving. The analysis of mix protocols still appears out of reach for the automatic protocol analyser ProVerif [22] but not for Scyther [41], which can handle some. However protocols are composed, their interactions make isolated security analysis hazardous. Intuitively, protocols sharing components may influence security guarantees in unforeseen ways. In practice, the risk of maliciously interleaving the sessions of two different protocols was demonstrated by Cremers, the author of Scyther [40], who exposed a number of multi-protocol attacks.

Thanks to a compositionality theorem, Scyther can analyse two theorems separately and conclude that their parallel composition is correct. However, this result is limited to "specific classes of protocols that can be composed in certain ways" [4], namely classes whose protocols are "strongly independent". In fact, "ciphertexts, signatures, and message authentication tags originating in one protocol set will never be accepted by the other protocol set and vice versa". For protocols that do not meet these strong conditions, it is still possible to model their parallel composition and analyse it using brute force in Scyther in general. The resulting search space may however become too large for practical purposes.

Our approach can tackle protocols that interact a lot and in fact depend on each other, such as a public-key authentication protocol depending on a certification protocol by using its certificates — notably, the two protocols can be specified and studied separately, but they can use the same elements, such as keys and certificates. We are not constrained by search space issues because of the inductive nature of our approach.

The technicalities of ad hoc protocol tools may somewhat hide the operational features of mix protocols and limit their understanding to non-practitioners of the software. This explains the need to reason about these protocols using a mathematically-rooted and well-understood language such as induction, which is not constrained by completeness issues.

4.2 Specification and Verification of a Composed Protocol

The main finding of this work is how to formally analyse mix protocols using the Inductive Method. Although Isabelle requires user specialisation to conduct the proofs, the findings are intuitive thanks to the simplicity of induction. Therefore, they can be easily understood, for example, by mathematicians approaching protocol analysis without interest in learning specialised tools, also because most proofs can be reproduced using a pen and paper, depending on their size.

In particular, it is found that mix protocols can be specified holistically by using more than one inductive protocol definition, precisely one per protocol. In case of sequencing, a protocol step can be premised with particular conditions about a previous protocol. Typically, such conditions express the achievement of the main goals of the previous protocol, such as the distribution of a session key or the authentication of a peer. In abstract, logical terms, the case of stacking is the same, as the necessary premise conditions refer to the underlying protocol. In case of interleaving, the two (n in general) protocols refer to each other using the same mechanism, a feature that enforces the advancement of a protocol only upon condition that the other protocol advances too. This approach bears the potential to scale up easily to the specification of protocols obtained by composition.

This holistic approach is demonstrated here upon a mix protocol built by sequencing. First, a general certification protocol whereby an agent receives from a certification authority a certificate for herself and her peer is defined and studied inductively. Then, an authentication protocol is tackled. Notably, it is based on its peers' knowledge of the necessary certificates; hence on the successful completion of the underlying certification protocol. For the sake of demonstration, we chose the best-known authentication protocol, due to Needham and Schroeder, which has rarely been analysed in its full version including certification. While Meadows analysed it as a monolithic entity [87], we will treat certification separately from the remaining protocol but derive holistic guarantees about their sequential combination .

The second finding is a general treatment of certification, which can be reused for all public-key protocols tackled so far. Agents that had to refer to each public key as if their owner was magically known can now use a public key accordingly to the corresponding certificate, which is signed by the authority. By reflecting exactly what happens in the real world, this reaches our aim of a more precise and reliable analysis of the authentication protocol. The depth of the certificate chain is kept to one level for simplicity, but it can be naturally generalised to many levels by considering more modular protocols. Recall the presentation of PKI seen earlier (§1.2.3).

Summary This chapter continues by presenting the certificate distribution and authentication protocols featured in a running example (§4.2.1). The main guarantees for our case study are discussed (§4.2.2). More details on the formal specifications and proofs follow (§4.2.3). Some conclusion terminate the treatment, and outline the potential for future applications arising from the scalability of the approach (§4.4).

4.2.1 Specification

Our research begins with the analysis of the generic certificate distribution protocol in Figure 4.1. Its function is to tell agents, on request, which public keys are associated with their peers. The standard protocol notation is adopted, and the reader's familiarity with it is assumed. An agent A contacts a certification authority *CA* to obtain public-key certificates for her and her intended peer B. In a subsequent protocol, A may use her certificate to forward it to her peer, and needs her peer's to meet a very basic requirement: knowing what public key to use with her peer to ensure that only he can decrypt her traffic. Public knowledge of the *CA*'s public key — needed by agents to validate certificates — is assumed, but could be treated as the outcome of a previous protocol in the sequence.[1] It is also assumed that the CA knows all public keys. For regular agents, the usual initial knowledge setup was left unchanged, but the protocol narrative does not rely on it since all main steps depend on the certification distribution

[1] That process, however, has to stop at some stage and ultimately rely on a locally available certificate.

protocol.

$$\begin{array}{lll} 1. & A \longrightarrow S & : \quad \{\!|A, B|\!\} \\ 2. & S \longrightarrow A & : \quad \{\!|K_A^+, A|\!\}_{K_S^-}, \{\!|K_B^+, B|\!\}_{K_S^-} \end{array}$$

Figure 4.1: A generic certificate distribution protocol

The most published protocol ever is (yet again) quoted in Figure 4.2 for the reader's convenience. It is the public-key Needham-Schroeder protocol [94] with Lowe's fix [83] of repeating B's identity next to the nonce pair in a message. This is the full version that includes the certificate distribution steps, which are, irrespectively of their importance, usually simplified away. We present the protocol with appropriate line spacing in order to emphasise that the first two steps provide the initiator with her peer's certificate, as the trusted server S is offering the certification service. Then, steps 4 and 5 are homologous for B.

$$\begin{array}{lll} 1. & A \longrightarrow S & : \quad \{\!|A, B|\!\} \\ 2. & S \longrightarrow A & : \quad \{\!|K_B^+, B|\!\}_{K_S^-} \\ 3. & A \longrightarrow B & : \quad \{\!|N_a, A|\!\}_{K_B^+} \\ 4. & B \longrightarrow S & : \quad \{\!|B, A|\!\} \\ 5. & S \longrightarrow B & : \quad \{\!|K_A^+, A|\!\}_{K_S^-} \\ 6. & B \longrightarrow A & : \quad \{\!|N_a, N_b, B|\!\}_{K_A^+} \\ 7. & A \longrightarrow B & : \quad \{\!|N_b|\!\}_{K_B^+} \end{array}$$

Figure 4.2: The full public-key Needham-Schroeder protocol with Lowe's fix

The certificate distribution protocol we model, as shown in Figure 4.1, subsumes the distribution steps from Figure 4.2 and does a bit more: in addition to the certificate of an agent's peer, it also sends the agent its own certificate. It can be studied using traditional techniques. The inductive specification of its main event is quoted here (*cert* is the name of the inductive protocol model).

$$\begin{aligned} &| \textit{Cert2}: [\![\textit{evsc2} \in \textit{cert}; \textit{Gets CA} \{\!|\textit{Agent A}, \textit{Agent B}|\!\} \in \textit{set evsc2}; \ A \neq B]\!] \\ &\quad \Longrightarrow \textit{Says CA A} \{\!|\textit{Crypt} (\textit{priSK CA}) \{\!|\textit{Key} (\textit{pubEK A}), \textit{Agent A}|\!\}, \\ &\qquad\qquad \textit{Crypt} (\textit{priSK CA}) \{\!|\textit{Key} (\textit{pubEK B}), \textit{Agent B}|\!\}|\!\} \end{aligned}$$

$\# evsc2 \in cert$

It can be seen that upon receiving a valid certificate request, *CA* replies to the agent first quoted in the message by sending two certificates: one for each agent quoted. The authority only checks that it is issuing certificates for two different agents — else, either the requesting agent only obtains its own certificate, or it receives no answer (since the reply is sent to the first agent name quoted in the initial message). A signature by *CA* is indicated by *Crypt (priSK CA)*.

The specification phase can now tackle the authentication protocol, and can refer to the certification one. Figure 4.3 shows the associated Isabelle theory hierarchy.

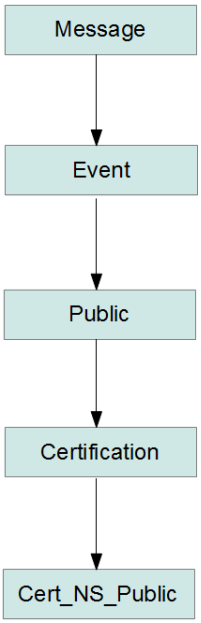

Figure 4.3: Theory hierarchy for the composed protocol

Let *NA* be a fresh nonce and assume a certificate mentioning agent *B* is part of *A*'s knowledge derived from the certification protocol. Assume also that *A* obtained *B*'s certificate on some trace of the certification protocol. Then the first message of the authentication protocol is sent by *A* to *B*: the concatenation of *NA* and of the sender's name, all encrypted with the key found in the certificate for *B*. The inductive model for the authentication protocol can be found in §A.1.2.

Notably, it is the first time that the specification of a protocol with the Inductive

Method needs to make assumptions on traces of two different protocols, here *evsca* from the protocol specified by *cert*, and *evs1* from the protocol being specified by *ns_public*. Precisely, the assumption on *evsca* serves to bind the key that *A* uses to build the new message.

Similarly, if *B* has received a message of format *NS1* and knows a certificate for *A*, then he picks another fresh nonce *NB* and sends it, encrypted, to *A*, along with the previously received nonce and *B*'s identity.

The explicit reference to traces belonging to the two protocols is visible also in the second step. In line with the previous step, the assumption on *evscb* serves to bind the key that *B* uses to build the new message.

In the third and final step, if *A* has sent the first message and received the the second message, he sends a new message to *B*, quoting the nonce *B* provided, and using the same encryption key ("*K*") as in the first message. *A* implicitly decrypts the received message using her private key *priEK A*.

According to the principle of *guarantee availability* [15], only one of the encrypting keys, *pubEK A*, can be specified. Because this rule defines an action of *A*'s, she can check that the message she gets is sealed under her public key. By contrast, she cannot check that the message she sent her peer was encrypted with his public key *pubEK B*: this must be proven in the model; hence the reference to a generic key *K*. In fact, such result holds because the event is traced back to when it was issued in the first protocol step, when access to the right certificate was assumed (only relying on trusting the CA). Quoting *pubEK B* explicitly in the preconditions of this protocol step, as is done in the initial formalisation, violates guarantee availability because it expects too much from *A* in terms of knowledge. *A* should only assume, at this stage, that she is reusing the encryption key from step one.

4.2.2 Results

The generic certification protocol (Figure 4.1) has been analysed using the standard techniques available in the Inductive Method. Despite the presence of an active at-

tacker, the standard Dolev-Yao [48], the protocol succeeds in establishing its intended security properties. The corresponding guarantees are summarised and discussed here, while their proofs are deferred (§4.2.3).

Theorem 1 (Says_CA_cert)**.** *The certification protocol establishes the following security properties.*

- *A message that the certification authority sends contains two different well-formed certificates.*
- *Each certificate contains the public key of the agent that the certificate mentions.*

This theorem holds because the authority cannot misbehave (it is assumed that the keys sent by the authority are the actual public keys of the associated agents). Its structure is not new because other protocols analysed so far rest on similar assumptions; however, its significance is innovative. It formalises for the first time that a bitstring contained in a certificate is the public key of the agent mentioned beside it. Formally, a generic key K found inside a certificate next to a generic agent name A implies that $K = pubEK\ A$. To enable the protocol participants to appeal to this theorem, another guarantee is needed.

Theorem 2 (cert_authentic_agent)**.** *The certification protocol establishes the following security properties.*

- *A well-formed certificate that a generic agent obtains originated with the certification authority.*
- *The authority may have sent it as either first or second component of its message.*

By combining this theorem with the previous, a generic agent can conclude that the mentioned key is the public key of the agent mentioned next to it. Despite the brevity of the statement, theorems of this form have never been proven before. It generalises to a generic agent the existing authenticity guarantees [14] confirming the originator of messages available to the attacker. This generalisation has required novel proof strategies involving a number of case splits, which are detailed later.

The main findings about the authentication protocol stem from each agent's check of their peer's certificate prior to sending the relevant protocol message, as we shall see (§4.2.3). The main confidentiality guarantees about the exchanged nonces can thus be expanded by stating the form of the encrypting key via an appeal to Theorem 1.

Theorem 3 (Spy_not_see_NA)**.** *The authentication protocol, if executed after the certification protocol by honest agents, establishes the following security properties.*

- *The first message that an initiator agent sends to a responder agent is such that:*
  - *it contains a confidential initiator's nonce;*
  - *it is encrypted by the responder's public key.*

Theorem 4 (Spy_not_see_NB)**.** *The authentication protocol, if executed after the certification protocol by honest agents, establishes the following security properties.*

- *The reply that a responder agent sends to an initiator agent is such that:*
  - *it contains a confidential responder's nonce;*
  - *it is encrypted by the initiator's public key.*

These guarantees confirm what happens in the real world. When honest agents engage in the authentication protocol, each of them does not have to blindly assume that they are using the right key. By contrast, each inherits a guarantee from the preceding certification protocol that they precisely are using the public key belonging to the intended peer. This level of detail was not available before our work.

4.2.3 Details of the Findings

This section outlines the actual theorems proven in Isabelle to support the less formal versions discussed above (§4.2.2).

Theorem 1 derives from the combination of the two following theorems.

theorem *Says_CA_cert1*:

$[\![Says\ CA\ A\ \{\!|Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ A|\!\},\ certB|\!\}\in set\ evs; evs\in cert]\!]$

$\Longrightarrow K = pubEK\ A$

The statement can be read as follows. For any sequence of events following the certification protocol, if *CA* sends a message containing a specific certificate and another component, then the mentioned key is the public encryption key of the agent mentioned beside the key. A version for the other certificate can be proven too.

theorem *Says_CA_cert2*:

$[\![Says\ CA\ A\ \{\!|certA, Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ B|\!\}|\!\}\ \in set\ evs; evs \in cert]\!]$

$\Longrightarrow K = pubEK\ B \wedge A \neq B$

It can be observed that this theorem, contrarily to the previous, specifies the agent pair; hence it can conclude that they are different by leveraging on the assumption stated by rule *Cert2* seen above.

Theorem 2 rests on the following innovative statement.

theorem *cert_authentic_agent*:

$[\![Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ B|\!\} \in parts(knows\ A\ evs); evs \in cert]\!]$

$\Longrightarrow (\exists\ D\ certB.\ Says\ CA\ D\ \{\!|certB, Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ B|\!\}|\!\}\ \in set\ evs) \vee$

$(\exists\ certB.\ Says\ CA\ B\ \{\!|Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ B|\!\}, certB|\!\}\ \in set\ evs)$

This result requires particular attention because it is the first significant fact proven *upon assuming* something about the knowledge of a generic agent. By contrast, existing proofs only consider the knowledge of the Spy. Here, *A* can be either the Spy or a regular agent. As a consequence, the proof features two successive inductions on the certification protocol model. First, assume *A* is the Spy. In this case, induction and simplification leave us with two remaining subgoals: the *Fake* case, which embodies the threat model, and the subgoal arising from the second protocol step. Those two cases are treated by classic methods, *spy_analz* and *blast*.

If *A* is an honest agent, that is different from the Spy, the proof is more intricate and was unexplored before the present effort. We must perform a number of case splits and reason about how protocol events modify agent knowledge. In existing Inductive Method theories, simplification lemmas are provided to deal with changes to the Spy's knowledge upon occurrence of protocol events, but the case of regular agents is not spelled out fully. The reasoning is therefore performed along the proof by expanding

the definition of *knows* as appropriate, since available automatic proof techniques only reason about the Spy's knowledge.

The case of the *Reception* case is particularly interesting. We first perform a case split on whether a certificate for *B* appeared in the traffic — that is, it exists in the Spy's knowledge. If such is the case, a traditional authenticity result about the certificate (*cert_authentic*, omitted here for brevity) can be applied and allows us to conclude. Otherwise our subgoal still features the premise *Crypt (priSK CA)* $\{$*Key K, Agent B*$\}$ $\in$ *parts (knows A (Gets Ba X # evsr))* and we must differentiate between the scenarios $A \neq Ba$ and $A = Ba$. In the former case, the *Gets* event cannot influence *A*'s knowledge because honest agent do not see all traffic; hence we obtain a contradiction. Else, *Crypt (priSK CA) Key K, Agent B* must be in *parts (insert X (knows A evsr))*. Since it is not in *parts (knows A evsr)*, it must be in *parts*$\{X\}$, but *X* appears in a *Says* event — hence it must be known to the Spy, a contradiction.

It is interesting to investigate what happens when we consider a simpler certificate distribution protocol that only returns one certificate in its second step (namely the certificate of the agent's peer, not of the agent itself). It turns out that the proof of *cert_authentic_agent* remains the same. The explanation is that the line of reasoning does not rely on case splits about the certificates, but rather on the interaction between the knowledge of agents. The main effect of the aforementioned change is that *Says_CA_cert1* and *Says_CA_cert2*, mentioned earlier, can be collapsed into a single lemma. Subsequent theorems that relied on those two lemmas then only require the unified one.

Theorem 3 can now be explained.

theorem *Spy_not_see_NA*:
$[\![$*Says A B* (*Crypt K* $\{\!|$*Nonce NA*, *Agent A*$|\!\}$) $\in$ *set evs*;
A $\notin$ *bad*; *B* $\notin$ *bad*; *evs* $\in$ *ns_public*$]\!]$
$\Longrightarrow$ *Nonce NA* $\notin$ *analz* (*knows Spy evs*) $\wedge$ *K* = *pubEK B*

This establishes the confidentiality of the initiator's nonce in the authentication protocol. It resembles the guarantee that can be found in the Isabelle repository [16]: if

the protocol step *NS1* takes place and A and B are honest, then the nonce from *NS1* remains secret. In addition, it specifies K to be B's public key. Notably, this requires appeals to the guarantees about the other protocol, the certification one. Thanks to the premise about trace *evsca* in rule *NS1* seen above, theorem *cert_authentic_agent* can be applied. Then, the combination of *Says_CA_cert1* and *Says_CA_cert2* pinpoints the contents of the certificates.

Similarly, Theorem 4 derives from the following result.

theorem *Spy_not_see_NB* :
$[\![Says\ B\ A\ (Crypt\ K\ \{\!|Nonce\ NA, Nonce\ NB, Agent\ B|\!\})\in set\ evs;$
$A\notin bad; B\notin bad; evs\in ns\_public]\!]$
$\Longrightarrow Nonce\ NB\notin analz\ (knows\ Spy\ evs)\wedge K = pubEK\ A$

This result can be commented similarly to the previous; however, the results about the certification protocol, starting with *cert_authentic_agent*, can be applied thanks to the premise about trace *evscb* in rule *NS2* described earlier.

4.3 Other Protocol Composition Configurations

So far, we presented a general methodology for analysing composed protocols through our case study featuring two protocols composed by sequencing. What of other scenarios? Even without additional case studies, some general considerations can be made. In turn, this section addresses sequences of more than two protocols and intertwined protocols.

4.3.1 Generalised Protocol Sequencing

In this chapter's case study, we analysed a composed protocol built by sequencing of a certificate distribution one and an authentication one. As in the theory files, let us denote the steps of the certificate distribution protocol by *Cert1* and *Cert2* and the steps of the authentication protocol by *NS1*, *NS2* and *NS3*. The dependencies between the steps can then be pictured as in Figure 4.4. When two steps are connected by an arrow,

the step at the arrow's head requires the step at its tail to have occurred.

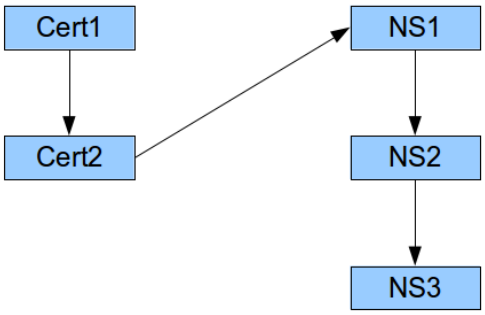

Figure 4.4: Protocol step dependencies for the example composed protocol

The step *NS2* could also be said to depend on *Cert2* directly. However, since it also relies on *NS1* which depends on *Cert2* too, we omit the arrow from *Cert2* to *NS2*. Dependencies between protocol steps can be seen syntactically as preconditions in the inductive rules modelling them: for instance, one of the preconditions of the rule for *NS1* requires the reception of a message which can only come from *Cert2*. In terms of proofs, the sequenced nature of the protocol composition is reflected by the fact that all lemmas requiring induction over the *cert* model can be proven before the model for *ns_public* is even defined.

It is now quite straightforward to reason about sequencing of more than two protocols. More precisely, consider n protocols $P_1 \dots P_n$, where P_{i+1} can only start when P_i has finished ($0 < i < n$). We denote $p_{i1} \dots p_{ik_i}$ the k_i steps of P_i ($0 < i < n + 1$). The dependency graph is represented in Figure 4.5.

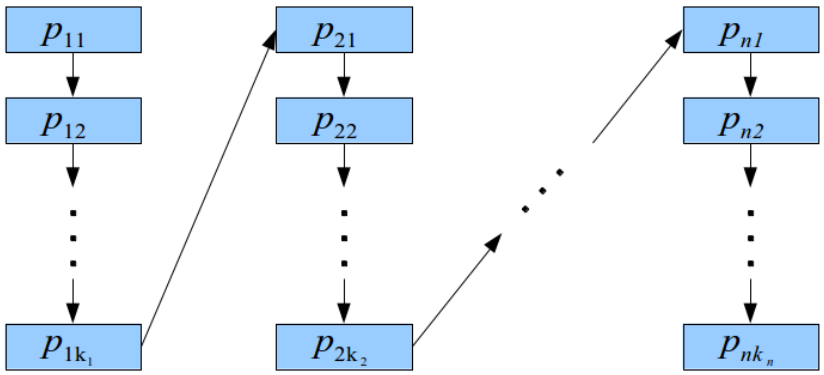

Figure 4.5: Protocol step dependencies for an arbitrary sequenced protocol

When analysing such a protocol, first properties about P_1 are proven independently. Analysing P_2, which depends on P_1, is then exactly the configuration of the case study earlier in this chapter. Once this is done, properties of P_2 have been established, and P_3 can be examined in turn the same way. Proving lemmas about P_n, which is only possible at the end, is then equivalent to proving properties of the composed protocol (since it depends implicitly on every other protocol in the sequence).

4.3.2 Intertwined Protocols

One can also consider a composed protocol where dependencies go both ways, as in Figure 4.6.

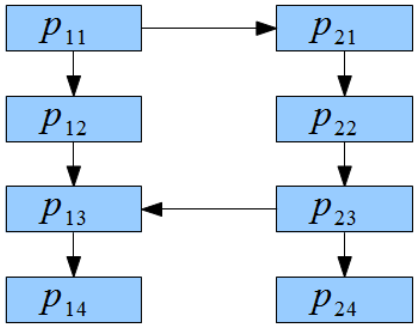

Figure 4.6: Protocol step dependencies for two entwined protocols

Here the protocols P_1 (with steps p_{1i}) and P_2 (with steps p_{2i}) interact to form an entwined, composed protocol. The dependency shown in Figure 4.6 allows the completion of the combined protocol run; for instance, the sequence of steps $p_{11} - p_{12} - p_{21} - p_{22} - p_{23} - p_{13} - p_{14} - p_{24}$ is allowed.

This kind of composed protocol can be analysed by exploiting the dependencies; namely, reliability lemmas (see §3.2.3) can be proven. If a given step has happened, then the steps it requires must have occurred as well. Nevertheless, in this particular configuration, both composing protocols can terminate before the other one has. The fact that two inductive definitions refer to each other causes no intrinsic issue.

Not all dependencies between the steps of two protocols make sense. The obvious example is when a step from the first protocol and a step from the second one refer to

each other. Such circular dependency makes the composed protocol not only impossible to analyse but, more importantly, meaningless since it can never run to completion. Another unsoundly composed protocol is pictured in Figure 4.7.

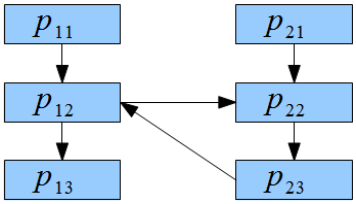

Figure 4.7: Protocol step dependencies for two unsoundly entwined protocols

Neither composing protocol can run to completion: p_{12} cannot occur before p_{22} because p_{12} requires p_{23}, and p_{22} cannot occur before p_{12} because p_{22} requires p_{12}. In terms of formal analysis, malformed composed protocols can be detected by the fact that so-called "possibility properties" cannot be proven. Possibility properties refer to a specific kind of reliability lemma, often proven at the very beginning of a protocol analysis in the Inductive Method, aiming to establish that a protocol model can yield a complete protocol run.

Barring such ill-formed dependencies between protocol steps, intertwined composed protocols can be analysed by modelling them as inductive models that refer to each other. Additional case studies are desirable to support this approach.

4.4 Discussion

We have described the formal modelling of mix protocols and their holistic analysis in the interactive theorem prover Isabelle/HOL. Sequenced, stacked and interleaved protocols can be specified and verified in a framework with rigorous foundations using inductive mathematical reasoning. In our running example, we have analysed a sequence featuring a generic key certification protocol and a simple authentication protocol. For the specific case of key certification, our approach is scalable to a full PKI model featuring multiple levels of trust. The adopted strategy for mix protocol specification translates into a proving process featuring novel situations. Those could

be tackled effectively with the mechanical support of the interactive theorem prover. While automatic protocol analysers are making progress, more general tools like Isabelle/HOL and the Inductive Method provide invaluable flexibility for reasoning in detail about common and uncommon protocol combinations. Case studies for intricate protocol interactions and mixes of more than two protocols are our natural next objects of study. The Isabelle locales [8] system could be used to permit more generic proofs. Part of this work was published in [17].

Chapter 5

Modelling an ISO/IEC 9798-3 Protocol Using Auditable Identity-Based Signatures

In the previous chapter, we looked at the issue of verifying composed protocols with the Inductive Method. Another commonly needed feature is the possibility of analysing protocols that require more exotic cryptographic primitives than the ones specified in *Message*. We now consider the case of a special kind of digital signatures: auditable identity-based signatures (AIBS). After presenting the general features of AIBS (§5.1), we introduce an ISO protocol that will be adapted to use this cryptographic primitive (§5.2). A side-by-side specification of AIBS and IBS versions of the protocol (§5.3), their comparative analysis (§5.4) and a concluding discussion (§5.5) follow.

5.1 Auditable Identity-Based Signatures

AIBS are a type of IBS providing stronger non-repudiation qualities. After presenting some context, properties of basic IBS and AIBS are listed.

Principles of auditable IBS AIBS [65] provide stronger non-repudiation by addressing the issue of key escrow. The details of the underlying cryptographic scheme are omitted here and can be found in the aforementioned report. An *audit step* allows a third party to verify that a given signature has been issued by the user herself and not by the KGC. This provides strong non-repudiation.

The audit step is separate from signature validation, and optional. Audit requires the transmission of additional information — if it became a required part of the validity check, the scheme wouldn't be identity-based anymore.

Upon request, thanks to the audit step, the KGC can prove that a signature has been provided by a user and not created by itself. The scheme rests upon the existence of a second key pair (A^+_{ID} ,A^-_{ID}), generated by the user ID. The usual key pair (D^+_{ID}, D^-_{ID}) exists as for standard IB signatures. The message M is signed by both secret keys, and the resulting signature is denoted $\{\!|M|\!\}_{D^-_{ID},A^-_{ID}}$.

A user should only be able to sign with a registered audit key-pair: indeed, she does not gain direct access to D^-_{ID}, but only to D^-_{ID} "protected" (which does not mean encrypted — for details, see [65]) by A^+_{ID}. This requires her to use D^-_{ID} in conjunction with A^-_{ID}.

Additional properties provided by auditable IBS

- ID cannot sign with the usual secret key D^-_{ID} without using the private audit key A^-_{ID} in conjunction, through a key package system. The signing key is not made available explicitly but can be used via the package provided the corresponding private audit key is known.
- Only ID knows A^-_{ID}; not the KGC.
- ID can sign thanks to the key package $\{\!|\{\!|D^-_{ID}|\!\}|\!\}_{A^+_{ID}}$, but never knows D^-_{ID} on its own.
- The signature validity check does not involve the audit key pair, and is done as usual. Hence only the message M, its signature $\{\!|M|\!\}_{D^-_{ID},A^-_{ID}}$ and the sender's

public key D^+_{ID} are needed.

- The audit step shows if the resulting signature was produced using A^-_{ID}. If this is not the case, ID did not sign, since he cannot sign without using A^-_{ID}. If so, the signature was forged by the KGC, since only these two entities know D^-_{ID}. To perform the audit step, the acting third party needs M and $\{\!|M|\!\}_{D^-_{ID},A^-_{ID}}$, as in the validity check step, but A^+_{ID} instead of D^+_{ID} .

5.2 The ISO/IEC 9798-3 Protocol Suite

In ISO/IEC 9798-3, the International Organization for Standardization presents a number of authentication protocols. The 2010 amendment [70], to which we refer here, introduces additional protocols. A good deal of those protocols have been formally analysed before [12]; however, our focus here is more on the extension of the Inductive Method with new cryptographic primitives than on the protocols themselves.

The specific protocol used in the remainder of this chapter is the first protocol from the 2010 amendment, *Five-pass mutual authentication with TTP, initiated by A*, Option 1. It is called 9798-3-6-1 in [12]. Its steps are as follows:

1. A picks a fresh nonce N_a and sends B the message $\{\!|A, N_a, T_1|\!\}$.
2. Upon reception of A's message, B picks another fresh nonce N_b and sends A the message $\{\!|B, N_a, N_b, T_3, Sig_B(B, N_a, N_b, A, T_2)|\!\}$.
3. Upon reception of B's reply, A picks yet another fresh nonce N'_a and sends the TTP the message $\{\!|N'_a, N_b, A, B, T_4|\!\}$.
4. Upon reception of A's message, the TTP sends A the message

$$\{\!|A, K^+_A, B, K^+_B, Sig_{TTP}(N'_a, B, K^+_B, T_6), Sig_{TTP}(N_b, A, K^+_A, T_5), T_7|\!\}$$

5. Upon reception of the TTP's reply, A sends B the message

$$\{\!|T_9, A, K_A^+, Sig_{TTP}(N_b, A, K_A^+, T_5), Sig_A(N_b, N_a, B, A, T_8)|\!\}$$

The T_i are text fields, specified in the standard as being optional. They are not meant to be confidential, and their contents and purpose are not stated in ISO/IEC 9798-3. Some message components, such as public keys, may be redundant in an IBS setting. We elect to keep them, preferring to risk redundancy than a lack of explicitness.

5.3 Side-by-side Specification of IBS and AIBS Variants of an ISO/IEC 9798-3 Protocol

The protocol presented in the previous section is a suitable testbed to compare features of IBS and AIBS: it is recent, uses digital signatures and is standardised. We set out to create two formal models of the protocol for the two types of signatures in order to highlight the benefits of AIBS. The basic goals of the protocol, such as authentication, are not our focus here.

Figure 5.1 shows the hierarchy of Isabelle/HOL theories arising from this analysis. The AIBS trunk is more independent of existing theories because new message components are required to account for the key package datatype.

In both the IBS and the AIBS version, the Trusted Third Party (TTP) is simply what is commonly called the Server:

abbreviation

TTP :: *agent* **where** *TTP* == *Server*

However, the TTP and the TA are two different entities. Indeed, the TA is specified as a fixed friendly agent; this way, we can control whether it is honest or not. Recall that, on the other hand, the Server is never in *bad*, by definition.

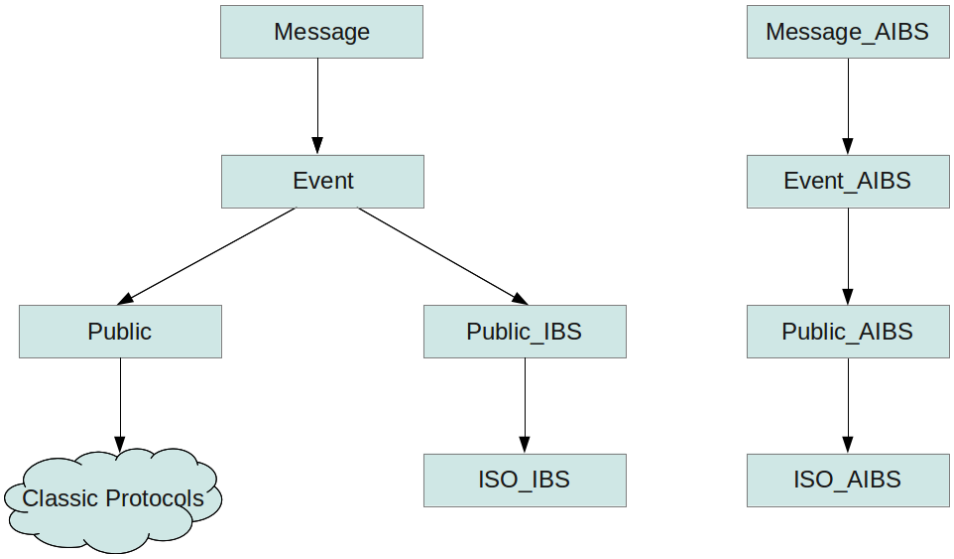

Figure 5.1: Theory hierarchy for the IBS and AIBS versions of the ISO/IEC 9798-3 protocol

abbreviation

TA :: *agent* **where** *TA* == *Friend 0*

5.3.1 Specifying the IBS Version

Rules about the initial knowledge of cryptographic keys are defined in the *Public* theory. *Public_IBS* is similar to it: the main difference is key escrow, modelled by the Server's knowledge of all secret signing keys. Lemmas about initial knowledge, found in the same theory, were adapted accordingly. The initial knowledge of agents includes their own secret keys.

Earlier attempts included the modification of *Message* to account for the fact that in the context of IBS, identities of agents yield their public keys. Our initial strategy consisted in adding a *CryptID agent msg* component to the *msg* datatype, which then yielded a *agentsFor* set, similar to *keysFor* but returning a set of agents instead of a set of keys.

This approach was dropped due to its lack of operational significance, especially for a comparative analysis of signature structure. We are more interested in what additional information the two kinds of signature convey than in the details of the

public keys required to open them. As a consequence, specifying IBS boils down to formalising signatures with the same operator rules as usual, but different initial key knowledge: in *Public_IBS*, *initState TA* notably contains *Key ‘ range priSK*. The situation for AIBS is different— see below.

The inductive model of the IBS version of the ISO/IEC 9798-3 protocol we consider can be found in §A.2.2.[1]

5.3.2 Specifying the AIBS Version

In the case of AIBS, signature generation requires a key package that hides the signing key from the signer. The message datatype was therefore extended to include a new component, the key package. A new datatype is defined first:

datatype *pack* = *Pack key key*

Now the following line can be added to the definition of the *msg* datatype in the *Message* theory:

| *Pkg pack*

The order in which the keys are used in the *Pack* notation does not matter as long as it is consistent. We chose to set the second key argument as the hidden signing key.

An initial attempt was made to include all rules governing the behaviour of the new component directly in *Message*. The resulting changes to the message operators broke relationships between them that are required for crucial tactics to work. We finally defined synthesis rules as part of the protocol model itself. The theories *Message*, *Event* and *Public* still required changes due to the presence of this new message component.[2] As for IBS, the initial knowledge of agents is also different from the one found in the legacy *Public* theory. As in the IBS case, the TA knows all secret signing keys. On the other hand, as opposed to the IBS case, agents no longer know their own signing keys

[1] We omit *Nil*, *Fake* and *Reception* since they were discussed earlier in this thesis and do not change. We will always do this for the protocols in this thesis. Text fields were modelled as Numbers, which can always be synthesised (they are “guessable”).

[2] In the case of Event, no real change was needed; only the *import* command needed tweaking.

in the AIBS version. They only have access to key packages that allow them to sign while the signing key remains hidden from them. *initState* was adapted accordingly.

The following inductive rules, part of the protocol model in *ISO_AIBS*, specify the interaction between keys, key packages and signatures. The *KeyPack* rule models the fact that if the attacker knows the KGC's master secret key, then she is able to generate any key package. What is meant by "key package" is the element obtained by protecting an agent's secret signing key with the corresponding audit key:

$$\begin{aligned} &| \mathit{KeyPack}: [\![\mathit{evsk} \in \mathit{iso}; \mathit{Key}(\mathit{priSK}\ \mathit{TA}) \in \mathit{analz}\ (\mathit{spies}\ \mathit{evsk})]\!] \\ &\qquad \Longrightarrow \mathit{Notes}\ \mathit{Spy}\ (\mathit{Pkg}\ (\mathit{KP}\ A\ B))\ \#\ \mathit{evsk} \in \mathit{iso} \end{aligned}$$

The *SigGen* rule shows the specification of auditable identity-based signatures: they feature a redundant structure in which the signed message is included twice in the main signature body: once as is, and once as part of a ciphertext encrypted by a private audit key. By convention, we arbitrarily specify audit encryption keys as the *priEK* type and normal signing keys as the *priSK* type. The purpose of this rule is to specify under which conditions the attacker can forge an auditable identity-based signature. A key package and the corresponding private audit key are the crucial ingredients:

$$\begin{aligned} &| \mathit{SigGen}: [\![\mathit{evss} \in \mathit{iso}; X \in \mathit{synth}(\mathit{analz}\ (\mathit{spies}\ \mathit{evss})); \mathit{Key}\ (\mathit{priEK}\ A) \in \mathit{analz}\ (\mathit{spies}\ \mathit{evss}); \\ &\qquad \mathit{Pkg}\ (\mathit{KP}\ A\ B) \in \mathit{analz}\ (\mathit{spies}\ \mathit{evss})]\!] \\ &\qquad \Longrightarrow \mathit{Notes}\ \mathit{Spy}\ (\mathit{Crypt}\ (\mathit{priSK}\ B)\ \{\!|\mathit{Crypt}\ (\mathit{priEK}\ A)\ X, X|\!\}) \# \mathit{evss} \in \mathit{iso} \end{aligned}$$

The rest of the protocol model adapts the one for IBS by replacing the *Crypt (priSK A) M* IBS structure by the *Crypt (priSK A)* $\{\!|$*Crypt (priEK A) M, M*$|\!\}$ AIBS structure. It can be found in §A.2.6.

5.4 Comparative Analysis

Our main aim was to formalise the fact that different signature structures reveal different types of information. The main benefit of AIBS is to make the signatures auditable,

i.e. provide more fine-grained authentication. Thanks to the audit key-pair, it is possible to differenciate signatures generated by their legitimate signer from signatures forged by the TA.

The main tool for the comparison is the function *candidates*, defined as follows:

definition *candidates* :: *agent* ⇒ *event list* ⇒ *agent set* **where**
candidates A evs ≡
{*C*. *C* ≠ *Spy* ∧
(∃ *Y Z*. *Crypt* (*priSK A*) *Y* ∈ *parts* (*knows Spy evs*) ∧
(*A* = *C* ∨ *Key* (*priSK A*) ∈ *initState C*) ∧
(*Crypt* (*priSK C*) *Z* ∈ *parts* {*Y*} ∨ *Crypt* (*priEK C*) *Z* ∈ *parts* {*Y*}))}

It takes as input an agent name *A* and a trace *evs* and returns a set of agents — the set of those, different from the Spy, that can sign on *A*'s behalf and leave a trace as a ciphertext encrypted with one of their secret keys inside a signature by *A* on *evs*.

Note that this definition does not ignore the Spy's action: she can still produce signatures using keys from compromised agents.

5.4.1 Findings for the IBS Version

The first result derives directly from the specification of initial knowledge of agents: if A or the TA are bad, then A's secret signing key is known to the Spy too. Else, it is only known to A and the TA:

lemma *priSK_knowledge*:
⟦*Key* (*priSK A*) ∈ *initState D*; *A* ∉ *bad*; *TA* ∉ *bad*⟧ ⟹ *D* = *TA* ∨ *D* = *A*

Even though our focus on this chapter is not on the properties of the ISO protocol itself, we give a glimpse of its authentication properties. The fact that *B* authenticates *A* is expressed as follows:

theorem *B_auth_A*:
⟦{|*Number Text9*, *Agent A*, *Key* (*pubSK A*),
Crypt (*priSK TTP*) {|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|},

$$\begin{aligned}
&\mathit{Crypt}\ (\mathit{priSK}\ A)\ \{\!|\mathit{Nonce\ Nb}, \mathit{Nonce\ Na}, \mathit{Agent\ B},\\
&\qquad\qquad \mathit{Agent\ A}, \mathit{Number\ Text8}|\!\}\ \in \mathit{parts}\ (\mathit{spies}\ \mathit{evs});\\
&\mathit{Says\ B\ A}\ \ \{\!|\mathit{Agent\ B}, \mathit{Nonce\ Na}, \mathit{Nonce\ Nb}, \mathit{Number\ Text3},\\
&\qquad\quad \mathit{Crypt}\ (\mathit{priSK\ B})\ \{\!|\mathit{Agent\ B}, \mathit{Nonce\ Na}, \mathit{Nonce\ Nb},\\
&\qquad\quad \mathit{Agent\ A}, \mathit{Number\ Text2}|\!\}\ |\!\}\ \in \mathit{set\ evs};\\
&A \notin \mathit{bad}; \mathit{TA} \notin \mathit{bad}; \mathit{evs} \in \mathit{iso}[\![\\
&\Longrightarrow \mathit{Says\ A\ B}\ \{\!|\mathit{Agent\ A}, \mathit{Nonce\ Na}, \mathit{Number\ Text1}|\!\}\ \in \mathit{set\ evs}
\end{aligned}$$

The first precondition refers to *ISO5*, the second one to *ISO2*, and the postcondition to *ISO1*. Its proofs requires a few other lemmas, among which is a weaker authentication result:

lemma *sig_A_origin*:

$$\begin{aligned}
&[\![\mathit{Crypt}\ (\mathit{priSK\ A})\ \{\!|\mathit{Nonce\ Nb}, \mathit{Nonce\ Na}, \mathit{Agent\ B}, \mathit{Agent\ A},\\
&\qquad\qquad \mathit{Number\ Text8}|\!\}\ \in \mathit{parts}\ (\mathit{spies}\ \mathit{evs});\\
&A \notin \mathit{bad}; \mathit{TA} \notin \mathit{bad}; \mathit{evs} \in \mathit{iso}[\![\\
&\Longrightarrow \mathit{Says\ A\ B}\ \{\!|\mathit{Number\ Text9}, \mathit{Agent\ A}, \mathit{Key}\ (\mathit{pubSK\ A}),\\
&\qquad\quad \mathit{Crypt}\ (\mathit{priSK\ TTP})\ \{\!|\mathit{Nonce\ Nb}, \mathit{Agent\ A}, \mathit{Key}\ (\mathit{pubSK\ A}), \mathit{Number\ Text5}|\!\},\\
&\qquad\quad \mathit{Crypt}\ (\mathit{priSK\ A})\ \{\!|\mathit{Nonce\ Nb}, \mathit{Nonce\ Na}, \mathit{Agent\ B},\\
&\qquad\qquad\qquad \mathit{Agent\ A}, \mathit{Number\ Text8}|\!\}\ |\!\}\ \in \mathit{set\ evs}
\end{aligned}$$

Here, the postcondition refers to *ISO5* and the precondition to its second half. If both *A* and the TA are assumed to be honest, the *candidates* function over *A* returns an empty set for any trace in the IBS case. This was to be expected: this version features only plain IBS signatures, the structure of which does not permit any auditing. A signature generated by *A* cannot be told apart from a signature forged by the TA, since both know *A*'s secret signing key.

theorem *candidates_IBS_none*:

$$[\![D \in \mathit{candidates\ A\ evs}; \mathit{TA} \notin \mathit{bad}; A \notin \mathit{bad}; \mathit{evs} \in \mathit{iso}]\!] \Longrightarrow \mathit{False}$$

As well as an induction over *D*, the proof requires the lemma *signature_form*, which states that messages of a given structure never appear in the network traffic:

lemma *signature_form*:

$[\![Crypt\ (priSK\ A)\ Y \in parts\ (spies\ evs);$

$Crypt\ (priSK\ TA)\ Z \in parts\ \{Y\} \vee Crypt\ (priEK\ TA)\ Z \in parts\ \{Y\} \vee$

$Crypt\ (priSK\ A)\ Z \in parts\ \{Y\} \vee Crypt\ (priEK\ A)\ Z \in parts\ \{Y\};$

$A \notin bad; TA \notin bad; evs \in iso]\!]$

$\Longrightarrow False$

Its proof consists of an induction over the protocol steps, combined with classic methods such as *spy_analz* and *force*.

5.4.2 Findings for the AIBS Version

In the AIBS version, if *candidates* contains an element and both the TA and A are honest, then that element must be A:

theorem *candidates_AIBS_A*:

$[\![D \in candidates\ A\ evs; TA \notin bad; A \notin bad; evs \in iso]\!] \Longrightarrow D = A$

If the TA is not required to be honest, we can still quantify the set. In that case, *candidates A* may additionally contain TA:

theorem *candidates_AIBS_A_TA*:

$[\![D \in candidates\ A\ evs; A \notin bad; evs \in iso]\!] \Longrightarrow D = A \vee D = TA$

The proofs of these two theorems rely on the same techniques used for *candidates_IBS_none*, as well as the use of the classical reasoner and an additional lemma about the regularity of key packages:

lemma *Pack_conf*:

$[\![Pkg\ (Pack\ (pubK\ A)\ (priSK\ B)) \in parts\ (knows\ Spy\ evs); TA \notin bad; evs \in iso]\!] \Longrightarrow False$

Can a tighter result be obtained? Attempting to prove a variant of the theorem *candidates_AIBS_A_TA* in which the conclusion is only $D = A$ leads us to a subgoal presenting the scenario where a signature by A appears in a trace, with the signature containing a second signature encrypted with *Crypt (priSK TA)*. This can clearly not

be discarded: if the TA is dishonest (which is not forbidden by the preconditions of *candidates_AIBS_A_TA*), the Spy knows both the TA's and A's secret signing key and can therefore forge such messages.

Furthermore, the following statement shows that a regular agent can indeed be in her own *candidates* set. It analyses an arbitrary trace concatenated with the kind of structure effectively used in the protocol:

theorem *candidates_AIBS_possible*:
$[\![A \neq Spy; ev = Says\ A\ B\ (Crypt\ (priSK\ A)\ (Crypt\ (priEK\ A)\ X))\ \#\ evs]\!]$
$\Longrightarrow A \in candidates\ A\ ev$

Finally, the lemma *Fake_generates_AIBS* is a possibility result showing that if the TA is bad, the Spy can indeed forge an AIBS that verifies but does not audit:

lemma *Fake_generates_AIBS*:
$TA \in bad \Longrightarrow\ \exists\ D\ B\ N.\ \exists\ evs \in iso.$
$Says\ Spy\ D\ (Crypt\ (priSK\ B)\ \{\!|Crypt\ (priEK\ TA)\ (Number\ N), Number\ N|\!\})\ \in set\ evs \wedge$
$B \neq TA$

5.5 Discussion

This chapter recalls the features of auditable identity-based signatures and features them in a comparative analysis of a recent ISO/IEC 9798-3 authentication protocol. Both a version with identity-based signatures and a version with auditable identity-based signatures are specified and analysed. Using a common framework for trace analysis allows us to highlight the impact of signature structure choice. The flexibility of the Inductive Method is confirmed, both for new cryptographic primitives specification and for alternative trace analysis methods. A number of approaches were tried for the modelling of the new cryptographic primitives; shifting their specification from the imported theory files to the protocol model itself seems to be the most efficient strategy.

Chapter 6

Formally Analysing an Electronic Voting Scheme Using Blind Signatures

The use of electronic voting (e-voting) for official elections is on the rise across the world. Security protocols claiming properties that protect voters and guarantee regular elections require formal scrutiny because of their sensitive nature. One key objective of e-voting protocols is to hide the way a particular voter votes. Most recent efforts to advance formal verification of this property are based on process equivalence. Despite substantial progress, limitations remain on aspects such as termination of supporting tools or simplification of protocol models.

The benefits of specifying privacy in an interactive theorem prover have never been explored until now. Existing research on the topic exploits other tools, which do not provide comparable levels of interaction. Isabelle [99], a generic theorem prover, is flexible enough when used with higher-order logic to allow new classes of security properties to be analysed in the framework provided by the Inductive Method [14]. Its extensions for dealing with voter privacy are described and demonstrated on a classic protocol in this chapter. They required new proof techniques and lines of reasoning,

whose development in turn demanded substantial effort. Nevertheless, their application to other protocols is expected to be straightforward, as has been the case for the confidentiality argument [86] for example, with most of the proof scripts adapted for new protocols without significant effort. Automated tools are ideal for checking conjectures about protocols quickly. However, the interactive nature of the Inductive Method yields, also with e-voting, greater support to the analyst's understanding of the protocol intricacies than the one fully automated tools offer today.

The most notable findings in this area stem from formalising the protocols with a process algebra and encoding the privacy properties by process equivalence [44]. As detailed below, process equivalence supports a notion of *indistinguishability* between two situations where a voter voted, respectively, for two different candidates. This implies that an observer cannot discern the two situations being formalised. In line with the operational semantics of the protocols specified by the Inductive Method, we develop an operational encoding of privacy based on *unlinkability* of voter with vote, focusing on the associations that an active attacker can derive from intercepting the protocol traffic. For example, if Alice sent her vote for Bob to the election administrator as a clear-text, then the attacker would build the association Alice-Bob.

However, actual protocol messages are complicated nestings of advanced cryptographic operations (which are still assumed to be secure), so that the attacker's inspection is far from straightforward. This inspection is formalised by the innovative *association analyser* operator *aanalz* — naming is consistent with the existing terminology. Also, the attacker can intelligently merge associations when they have at least one element in common, similarly to an investigator relating Alice to a crime scene because she wears the same shoe size as that of a shoe print in the scene. This merge is formalised by the innovative *association synthesiser* operator *asynth*. When it is impossible to build, by means of analysis and then synthesis, an association that features both voter and vote, then there is unlinkability of voter with vote, hence the protocol enforces voter privacy (about their vote). Conversely, the protocol violates voter privacy, irrespectively of how many other voters cast that vote.

An outline of the indistinguishability and unlinkability approaches to modelling privacy (§6.3) leads to our extensions to the Inductive Method to account for privacy specification and analysis (§6.3.2). These extensions are then demonstrated on the classic FOO [58] e-voting protocol through its inductive specification (§6.6.2) and verification of voter privacy (§6.7). Conclusions and future work conclude the chapter (§6.9).

6.1 The Spread of Electronic Voting

When compared to classic methods, electronic voting can offer a number of benefits. The most obvious one is efficiency: automated counting methods can drastically reduce the time and manpower required to compute election results. Money can be saved, and the risk of human counting errors eliminated. A different, maybe even more important advantage is increased monitoring possibilities by the voters themselves. Ideally, e-voting protocols aim to give voters sufficient information to check that their vote was indeed taken into account, while still preserving their privacy. In many cases, these guarantees should hold even without assumptions about the honesty of election officials. What is more, outsiders not taking part in the election should be able to check that the published results reflect the votes that were actually cast.

Severely flawed voting machines from the Diebold company were infamously and massively used for official elections in the United States [55]. Despite public concern over these kinds of cases, e-voting is spreading quickly across the world. Ireland partially used it for the 2002 general election.[1] In 2005, Estonia allowed internet voting for municipal elections nationwide. In 2007, it became the first country in the world using internet voting for parliamentary elections. Brazil, Canada, the United Kingdom and Switzerland also use it for official elections. Often, lack of verifiability means voters have to trust an all-powerful government about the results.

As more people in the world are affected by e-voting, rigorous analysis of such

[1] However, following analysis of the system to be used, it was decided "not to proceed with the implementation of electronic voting" in 2009 due to the lack of security guarantees.

protocols and their claimed goals is more important than ever. We turn next to some of the most common properties of e-voting protocols.

6.2 Common Properties of Electronic Voting Protocols

The following four properties are commonly studied in e-voting protocols and are of particular interest to the protocol we will analyse. Other properties, such as universal verifiability, are relevant for some e-voting protocols. These definitions are due to Mark Ryan [44].

Fairness: No early results can be obtained which could influence the remaining voters.

Eligibility: Only legitimate voters can vote, and only once.

Individual verifiability: A voter can verify that her vote was correctly counted.

Voter privacy: How a particular voter voted is not revealed to anyone.

This property is sometimes called ballot secrecy. Votes may or may not be published at the end of an election, so it is not the confidentiality of the vote in itself that matters but its association with the voter who cast it. In other words, the way a voter votes should not be discoverable by anyone, even after the count. A caveat on this definition is the exclusion of the marginal case where all voters vote identically. Similarly, we exclude the case where all voters but one voted identically — in that case, the person who voted differently also knows how everyone else voted.

6.3 Alternative Formal Approach to Voter Privacy Analysis

6.3.1 Indistinguishability for E-voting Protocol Analysis

A common way of modelling privacy involves showing the indistinguishability between two situations:

1. Voter V_a votes x and Voter V_b votes y

2. Voter V_a votes y and Voter V_b votes x

Indistinguishability here means that when V_a and V_b swap their votes, no party (including the trusted parties running the election) can distinguish between situations 1 and 2.

We now turn to a new model based on the unlinkability between two pieces of information.

6.3.2 Unlinkability

In contrast to the indistinguishability modelling of privacy, an operational view reflects the natural threat model of an attacker monitoring all network traffic and using the data she can extract to associate a voter with a vote. Initially, the attacker decomposes each individual message, and records all plaintexts and ciphertexts for which keys are available. She can also associate these with the intended recipient agent of the message. For each protocol event whereby an agent sends a message to another agent, this analysis gives the attacker a set of (components of) messages, namely an association. Moreover, if the communication channel is not anonymous, then the attacker can also extend the association just gathered by storing the identity of the sender.

However, it is not sufficient to inspect in isolation each of the messages sent in the traffic. A voter's identity V may appear "near" an element m that is later to be extracted again, this time in conjunction with the vote N_v. In this case, such a common element m provides the link between voter V and vote N_v. An attacker monitoring the network sees messages as discrete entities and can exploit the shared context of elements extracted from one given message. This process of combining sets of associations builds up an association synthesis. When all possible protocol scenarios are taken into account, establishing voter privacy boils down to inspecting the synthesised set for the presence of a voter's vote.

The only pieces of information that should not be treated as a possible link to

synthesise new associations are those that can be linked to *all* voters, such as the name of the precise election officials that a protocol prescribes. Because their identities appear in each and every protocol session, using one of them as a link would lead to the synthesis of insignificant, that is, privacy-irrelevant, associations. For example, an investigator will not call up every human being as suspect of murder simply upon the basis that everyone could pull a trigger. We shall see that with the FOO protocol, the administrator and the collector are omnipresent, hence must be ruled out to synthesise significant associations.

Without setting bounds on the number of agents, sessions, or message nesting depth, the number of different associations that the attacker can synthesise is very large. Precisely, an unbounded number of associations can be derived by observing a full trace, due to the fact that its length is unbounded. This size limits the tool support that traditional finite-state search can offer, since merely having an unbounded number of agents is already often a problem. As experienced before with other goals [14], inductive reasoning bypasses the size constraints also with the analysis of associations. In this model, associations are derived from traces, so more than one protocol may be involved.

Simple association sets Before specifying unlinkability in the Inductive Method, let us see how it works on an example trace. Consider the message exchanges (in chronological order) in Figure 6.1.

$$\begin{array}{lcl} 1. \quad A \longrightarrow B & : & A, B \\ 2. \quad B \longrightarrow A & : & \{\!|B, A, N_B|\!\}_{K_A^+} \\ 3. \quad A \longrightarrow C & : & \{\!|A, \{\!|N_B|\!\}_K, N_A|\!\}_{K_A^-} \end{array}$$

Figure 6.1: An example network history

What associations could an attacker derive when observing this trace? He would, for each message, list everything he could possibly extract from it, using if necessary the keys he knows. If a ciphertext is part of a message, it is automatically part of the association set. Whether the plaintext it contains is also a member of the association set

depends on the availability of the decryption key to the attacker. If concatenated data is in the message, both the concatenation and its elements belong to the association set. Assume the channels are not anonymous — then the identities of senders and recipients will be included too.

As a side note, in the more exotic setting of two competing Dolev-Yao attackers, association sets could quickly become meaningless as each attacker forges artifical associations to sabotage his rival's knowledge.

Formally, those rules mirror the *analz* operator (see §3.2.1):

- $X \in msg \Longrightarrow X \in associations(msg)$
- $\{\!|X|\!\}_{K^+} \in msg \wedge K^-$ known $\Longrightarrow X \in associations(msg)$
- $\{\!|X, Y|\!\} \in msg \Longrightarrow X \in associations(msg)$
- $\{\!|X, Y|\!\} \in msg \Longrightarrow Y \in associations(msg)$

Let A, B and C be three different agents. Assume that the private keys of *A* and *B* are only known to themselves, i.e. they are uncompromised. The resulting association sets, shown in Figure 6.2, are presented assuming that the message sender and receiver can be extracted from the message headers, observed as source and destination of messages on the network.

1. $\{A, B\}$
2. $\{B, A, \{\!|B, A, N_B|\!\}_{K_A^+}\}$
3. $\{A, C, \{\!|A, \{\!|N_B|\!\}_K, N_A|\!\}_{K_A^-}, \{A, \{\!|N_B|\!\}_K, N_A\}, \{\!|N_B|\!\}_K, N_A, N_B?\}$

Figure 6.2: Individual association sets generated from an example network history

The first step reveals nothing more than the name of the two peers. In the second step, the ciphertext is extracted but not its contents since an observer (which is not A himself since we assumed the agents to be uncompromised![2]) cannot normally inspect a message encrypted with the private key of an honest agent. More information

[2]Recall we are building association sets from the Spy's point of view.

can be extracted from the third message. On the one hand, what is inside the main ciphertext is accessible because A's public key is not confidential. On the other hand, the question mark after the last element of the third set indicates that the presence of N_B in this association set depends on the confidentiality of the key K. Both the set $\{A, \{\!|N_B|\!\}_K, N_A\}$ and its components are part of the association set because messages are decomposed into their elements inductively.

None of the three association sets feature both B and N_B; the first two feature B and not N_B, and the third one does not feature B. It may or may not include N_B, depending on the confidentiality of K. If K is confidential, then N_B is never visible to an attacker and cannot therefore be linked to anyone. If K is known to the attacker, she can extract N_B from the ciphertext containing it in the third message. Therefore N_B is in the third association set.

Pairwise association synthesis We may now ask whether B can be linked to N_B via pairwise association synthesis. Both the second and the third association set mention the name of agent A. This element in common provides a link between the elements in the two sets. B and N_B are linked via agent A.

Consider the following definition of a pairwise set synthesis set:

- $a_1 \in \Diamond \wedge a_2 \in \Diamond \wedge (\exists m \mid m \in a_1 \wedge m \in a_2) \Longrightarrow a_1 \cup a_2 \in pairsynth(\Diamond)$

When $\Diamond$ is a set of sets, $pairsynth(\Diamond)$ is too. Note that $a_1 = a_2$ is allowed, hence any non-empty set in $\Diamond$ ends up in $pairsynth(\Diamond)$.

The sets belonging to the pairwise association synthesis set are shown in Figure 6.3. The first three sets are exactly the ones from 6.2, and the two other ones are $1 \cup 3$ and $2 \cup 3$, respectively.

Transitive association synthesis A more general way of deriving association syntheses is to recursively join association sets. Consider the following definition of a transitive set synthesis set:

- $a_1 \in \Diamond \Longrightarrow a_1 \in transynth(\Diamond)$

1. $\{A, B\}$
2. $\{B, A, \{\!|B, A, N_B|\!\}_{K_A^+}\}$
3. $\{A, C, \{\!|A, \{\!|N_B|\!\}_K, N_A|\!\}_{K_A^-}, \{A, \{\!|N_B|\!\}_K, N_A\}, \{\!|N_B|\!\}_K, N_A, N_B?\}$
4. $\{A, B, C, \{\!|A, \{\!|N_B|\!\}_K, N_A|\!\}_{K_A^-}, \{A, \{\!|N_B|\!\}_K, N_A\}, \{\!|N_B|\!\}_K, N_A, N_B?\}$
5. $\{A, B, C, \{\!|B, A, N_B|\!\}_{K_A^+}, \{\!|A, \{\!|N_B|\!\}_K, N_A|\!\}_{K_A^-}, \{A, \{\!|N_B|\!\}_K, N_A\}, \{\!|N_B|\!\}_K, N_A, N_B?\}$

Figure 6.3: Pairwise association synthesis sets generated from an example network history

- $a_1 \in transynth(\Diamond) \wedge a_2 \in transynth(\Diamond) \wedge (\exists m \mid m \in a_1 \wedge m \in a_2)$ $\Longrightarrow a_1 \cup a_2 \in transynth(\Diamond)$

In other words, the number of links between two association sets should not be limited (but finite). As before, any pairwise union of association sets sharing an element should be in the association synthesis set, but additionally unions of sets which are already in the association synthesis set should be in it as well, provided they have a common element. In our running example, the largest transitive association synthesis set is equal to the largest pairwise association synthesis set in Figure 6.3 — but, in general, transitive association synthesis yields bigger sets than pairwise synthesis and hence models a stronger notion of unlinkability (because more rounds of association synthesis are being performed). For instance, there exists a transitive association synthesis set containing both u and z if the following are association sets:

1. $\{u, v\}$
2. $\{v, w\}$
3. $\{w, x\}$
4. $\{x, y\}$
5. $\{y, z\}$

Indeed, 1 and 2 share v, so $1 \cup 2 = \{u, v, w\}$ is an association synthesis set. For a similar reason, $4 \cup 5 = \{x, y, z\}$ also is. Since 3 is an association set, it is also an association synthesis set. It has an element in common with $1 \cup 2$, namely w. Hence $1 \cup 2 \cup 3 = \{u, v, w, x\}$ is an association synthesis set. It shares the element x with $4 \cup 5$, so finally the union of all those sets is an association synthesis set and it contains

both u and z. Conversely, there is no pairwise association synthesis set containing both u and z, but there is one containing both u and w, another one containing both v and x, and another one with x and z (and of course y).

6.4 Modelling Electronic Voting Protocols in the Inductive Method

The analysis of associations requires a new message operator, *analzplus*. It is built on the traditional *analz* message operator, endowed with an external message set providing extra decryption keys:

inductive_set

analzplus :: *msg set* ⇒ *msg set* ⇒ *msg set*

for *H* :: *msg set* **and** *ks* :: *msg set*

where

Inj [*intro*,*simp*]: $X \in H \Longrightarrow X \in analzplus\ H\ ks$

| *Fst*: $\{\!|X,Y|\!\} \in analzplus\ H\ ks \Longrightarrow X \in analzplus\ H\ ks$

| *Snd*: $\{\!|X,Y|\!\} \in analzplus\ H\ ks \Longrightarrow Y \in analzplus\ H\ ks$

| *Decrypt* [*dest*]: $[\![Crypt\ K\ X \in analzplus\ H\ ks; Key\ (invKey\ K) \in analzplus\ H\ ks]\!] \Longrightarrow X \in analzplus\ H\ ks$

| *Decrypt2* [*dest*]: $[\![Crypt\ K\ X \in analzplus\ H\ ks; Key\ (invKey\ K) \in ks]\!] \Longrightarrow X \in analzplus\ H\ ks$

In particular, the new operator is useful to formalise everything, namely the set of all message components, that the attacker can extract from a single message sent in the traffic by hammering it with the entire knowledge she has acquired on an entire trace. It can be seen as a localised version of *analz*, and features an additional *Decrypt* rule. For a message *X* and a trace *evs*, this set can be defined as $analzplus\ \{X\}\ (analz(knows\ Spy\ evs))$.

Using *analzplus*, the message association analyser *aanalz* can be defined inductively. Only *Says* events influence it. Indeed, each *Gets* message reception event follows a message sending event *Says*, so its message body was already processed by

aanalz. *Notes* events correspond to private recording of data by agents, yielding no observable information for the attacker.

```
primrec aanalz :: agent ⇒ event list ⇒ msg set set
where
 aanalz_Nil:  aanalz A [] = {}
| aanalz_Cons:
 aanalz A (ev # evs) =
 (if A = Spy then
  (case ev of
    Says A' B X ⇒
     (if A' ∈ bad  then aanalz Spy evs
      else if isAnms X
          then insert  ({Agent B} ∪ (analzplus {X} (analz(knows Spy evs)))) (aanalz Spy evs)
          else insert (({Agent B} ∪ {Agent A'} ∪ (analzplus {X} (analz(knows Spy evs))))
         (aanalz Spy evs) )
   | Gets A' X ⇒ aanalz Spy evs
   | Notes A' X ⇒ aanalz Spy evs)
  else aanalz A evs)
```

The definition indicates, among other aspects, that only the attacker can analyse associations. Regular agents cannot observe the entire traffic, so we focus on the point of view of a Dolev-Yao Spy who controls the network. Since she has access to more information, establishing unlinkability from her point of view is necessarily stronger than proving it from the point of view of an agent who sees less.[3]

Also, she will neglect the associations created by compromised agents, thus including those that she may have created, by sending out specific messages. We are interested in the information leaked by honest agents in the normal course of the protocol. The Spy can make up any message, and therefore any association if we were to extract them from her messages. It can also be seen that the sender identity is extracted

[3]This situation somewhat conflicts with the usual approach to guarantee availability for regular agents. We aim to establish a security property enjoyed by regular agents, yet they cannot observe the entire traffic like the Spy does.

only for messages that are not sent anonymously.

The *isAnms* predicate holds of messages with a specific form that we conventionally interpret to signify anonymity:

definition *isAnms* :: *msg* ⇒ *bool* **where**
isAnms M ≡ (∃ *Y*. *M* = *MPair* (*Number anms*) *Y*)

Anonymous channels are specified by defining a function to replace *Says* when needed. We are conventionally defining an anonymous message by means of a precise message format — the actual message is prepended with a constant number:

consts *anms* :: *nat*
definition *Anms* :: [*agent*, *agent*, *msg*] ⇒ *event* **where**
Anms A B X ≡ *Says A B* {|*Number anms*, *X*|}

Since the number *anms* is defined as a constant before, protocols involving messages with numbers can still use this specification, provided numbers used in other ways are stated to be different from *anms*. Creating a new datatype instead would involve changes to the message operators.

The pairwise association synthesiser *asynth* can be introduced now. Its definition is not tied to *aanalz*, but it will always be used in conjunction with it for our purposes. Specifically, we will examine the contents of the set *asynth (aanalz Spy evs)*, where *evs* is a generic protocol history. The *asynth* operator introduces a new association as the union of association sets that share a common element:

inductive_set
asynth :: *msg set set* ⇒ *msg set set*
for *as* :: *msg set set*
where
asynth_Build [*intro*]:
⟦*a1* ∈ *as*; *a2* ∈ *as*; *m* ∈ *a1*; *m* ∈ *a2*; *m* ≠ *Agent Adm*; *m* ≠ *Agent Col*⟧
⟹ *a1* ∪ *a2* ∈ *asynth as*

As noted above, the definition insists that a common element is not a piece of information that can be linked to *all* voters — for instance, the name of election officials since they appear in every step.[4] The version below can be used for protocols that define two election officials, here called *Adm* (for administrator) and *Col* (for collector), in line with the subsequent case study. In the protocol we will work with, the administrator and the collector check the validity of ballots and count the resulting votes. We will generalise *asynth* to a transitive version later.

6.5 The FOO Protocol

The Fujioka, Okamoto and Ohta (FOO) [58] protocol features two election officials called administrator and collector and involves bit commitments as well as blind signatures. It features six phases that will give rise to as many protocol steps:

1. *Preparation*: The voter V picks a vote N_v, builds $\{\!|N_v|\!\}_c$ using the commitment key c, and blinds this vote commitment using the blinding factor b. V then signs the resulting blinded commitment $\{\!|\{\!|N_v|\!\}_c|\!\}_b$ and sends it to the administrator along with V's identity.

2. *Administration*: Upon reception of a signed, blinded commitment, the administrator extracts the blinded commitment from the signature and checks that the alongside quoted agent name is equal to the signer of the blinded commitment. If such is the case and the agent has not voted before, the administrator returns the message $\{\!|\{\!|\{\!|N_v|\!\}_c|\!\}_b|\!\}_{K^-_{Adm}}$ to V, now signed by the former. The administrator also records V's name.

3. *Voting*: If V obtained the administrator's reply, V unblinds it to obtain $\{\!|\{\!|N_v|\!\}_c|\!\}_{K^-_{Adm}}$ and sends the resulting plain signature of the commitment to the collector over an anonymous channel.

[4] To make the definition of *asynth* fully generic, we could define a set of such elements, say *alwayslinked*, and reference it in the definition of *asynth*. The *asynth* definition would then be constant across protocols; only the contents of *alwayslinked* would vary.

4. *Collecting*: The collector checks the signature and publishes the enclosed vote commitment $\{\!|N_v|\!\}_c$ on a bulletin board once all votes have been received.

5. *Opening*: Once all commitments have been published, V sends c over an anonymous channel so that N_v can be revealed.

6. *Counting*: Upon reception of V's key, the collector publishes N_v.

6.6 Specifying the FOO Protocol and Blind Signatures

We are now ready to specify the steps of the protocol in the Inductive Method. The first step is to model blind signatures, which have not been specified in it before.

6.6.1 Blind Signatures

Blind signatures are a cryptographic primitive often found in e-voting protocols, and in particular in the FOO protocol. We specify them for the first time in the Inductive Method, as an inductive rule in the protocol model. First, we describe an initial, unsuccessful attempt to specify blind signatures as a change to the message datatype.

Message datatype modification attempt Since blind signatures constitute a new cryptographic primitive in the Inductive Method, our initial specification attempt was based on an extension of the *msg* datatype with the following line:

| *Blind key msg*

We were wary of using axioms. This quote from the Isabelle/HOL tutorial [95] warns user specifically about their dangers:

"The philosophy in HOL is to refrain from asserting arbitrary axioms (such as function definitions whose totality has not been proven) because they quickly lead to inconsistencies. Instead, fixed constructs for introducing types and functions are offered (such as *datatype* and *primrec*) which are guaranteed to preserve consistency."

It seemed natural to introduce this new *Blind* message component and to adapt the framework theory files — *Message*, *Event* and *Public* — accordingly.

Despite numerous trials, we were not able to obtain a specification in terms of the message operators for which all usual lemmas from the three theories just quoted hold. More precisely, extending the message datatype implies extending *parts*, *analz* and *synth*, the inductive message operators. Adapting their definitions is straightforward. Proving basic lemmas for each operator also is. The bottleneck appears for lemmas stating results about the combinations between different message operators. Specifically, we were not able to prove the following lemma for any specification of blind signatures:

lemma *parts_synth* [*simp*]: $parts\ (synth\ H) = parts\ H \cup synth\ H$

This lemma shows properties about the relationship between *parts* and *synth*, which the asymmetric structure of blind signatures broke in all of our trials. Furthermore, the *parts_synth* lemma can absolutely not be discarded because it is needed to prove *Fake_parts_insert*, which is ultimately needed in the *spy_analz* tactic. The *spy_analz* tactic itself is generally needed when reasoning about *Fake* cases, i.e. about the threat model. *parts_synth* is therefore an essential building block of the framework, and we cannot afford to lose it.

Let us be more precise about the nature of the issues that we faced by examining one example specification.

When writing a blind signature (with signing key *T*, on message *X*, with blinding factor *K*) as *Blind K (Crypt T X)*, [5] the issue arises from the *synth* rule modelling blind signature generation from a blinded message and a (signing) key:

$[\![Blind\ K\ X \in synth\ H;\ Key(T) \in H]\!] \Longrightarrow Blind\ K\ (Crypt\ T\ X) \in synth\ H$

To prove *parts_synth*, we need to show that if *Blind K (Crypt T X)* is in $parts\ H \cup synth\ H$, then *parts(Blind K (Crypt T X))* also is.

[5] The same issue appears when blind signatures are written as *Crypt T (Blind K X)*.

But we need an *analz* rule to obtain plain signatures from blinded ones, given the blinding factor:

$[\![Blind\ K\ X \in analz\ H;\ Key(K) \in analz\ H]\!] \Longrightarrow X \in analz\ H$

Because of the requirement *analz_into_parts*, this implies a corresponding rule in *parts*:

$Blind\ K\ X \in parts\ H \Longrightarrow X \in parts\ H$

Now, *parts(Blind K (Crypt T X)) = Crypt T X*. If *Blind K (Crypt T X)* is in *parts H*, *Crypt T X* also is by the aforementioned rule. But if *Blind K (Crypt T X)* is in *synth H*, we have a problem. Either it is also in *H*; in this case it is in *parts*, and everything is fine. But if it is not in *H*, then we can only conclude that *Blind K X* is in *synth H* and *T* is in *H*. This does not allow us to prove that *Crypt T X* is in *parts H* $\cup$ *synth H*. Indeed, for *Crypt T X* to be in *parts H*, we would need a new and counter-intuitive *parts* rule yielding it from *Blind K X* and *T*; this was tested, and broke other lemmas. And for *Crypt T X* to be in *synth*, we would need *X* to be in *synth*; but from *Blind K X* in *synth H*, we can only obtain *X* in *parts H*, which does not imply *X* in *H*.

It can be concluded that the expected relationships between message operators in the existing theories imply some level of symmetry in message components. Theoretical justification of the underlying issues can be provided by noting that adding new primitives with behaviour akin to blind signatures to the *msg* datatype can yield a context-sensitive language.

However, as the next paragraph shows, cryptographic primitives which do not fulfil these constraints can still be specified at the protocol model level.

Blind signatures as a protocol step The final approach, which proved much simpler, consists in specifying blind signatures as a protocol step. Whereas the blinding process works just like a normal signature with a symmetric key, the reverse process, unblinding, exhibits an asymmetry that warrants this extra rule. The Spy gains knowl-

edge of a plain signature if she knows the corresponding blinded signature and blinding factor, modelled as a symmetric key:

| *Unblinding*:
⟦*evsb* ∈ *foo*; *Crypt* (*priSK V*) *BSBody* ∈ *analz* (*spies evsb*);
BSBody = *Crypt b* (*Crypt c* (*Nonce N*)); *b* ∈ *symKeys*; *Key b* ∈ *analz* (*spies evsb*)⟧
⟹ *Notes Spy* (*Crypt* (*priSK V*) (*Crypt c* (*Nonce N*))) # *evsb* ∈ *foo*

Unblinding is also performed, implicitly, by agents in the third protocol step.

6.6.2 Inductive Protocol Model

Figure 6.4 shows the Isabelle theory hierarchy for FOO and the privacy extensions to the Inductive Method.

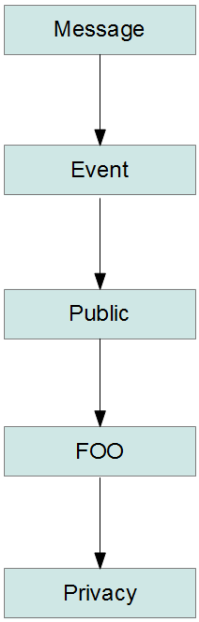

Figure 6.4: Theory hierarchy for the verification of the FOO protocol with the privacy framework

abbreviation *Adm* :: *agent* **where** *Adm* ≡ *Friend 0*

abbreviation *Col* :: *agent* **where** *Col* ≡ *Friend* (*Suc*(*0*))

Administrator and collector are introduced as translations *Adm* and *Col* of specific agents.

The FOO protocol as seen above (§6.5) can then be specified by the six inductive rules in §A.3.1.

The first protocol step involves a regular agent: he can be compromised, but cannot be one of the election officials. He picks fresh keys for blinding and commitment. The ballot, represented by a nonce, must also be fresh. In the conclusion of the predicate, two *Notes* event are appended after the *Says* event. If they are omitted, the keys are still considered fresh even though they have just been used.[6] Hence the *Notes* events are simply present to artificially specify the loss of freshness for b and c.

In the second step, the administrator reacts to the voter's message only if the voter's name has not been recorded already. This is specified by the premise *Notes Adm (Agent V)* $\notin$ *set evs2*. The administrator has received a message containing a ciphertext he cannot open: it is protected by the blinding factor, which only the voter knows. Hence the precondition over the structure of the received message only specifies that some ciphertext of length one was signed by the agent. There can be any number of encryption layers, but we assume that the administrator can verify that it contains only one atomic component. This condition is specified by stating that there is no message concatenation, denoted *MPair*, in *parts* of the ciphertext. Describing the contents of what the administrator should receive from an honest agent more precisely would not be in line with what the administrator can actually check. The genericity of this step (and of step *EV4* below) has huge implications in terms of proof complexity, but is necessary to be realistic about the assumptions that agents can make. In the postcondition of the rule, the administrator notes the name of the agent, preventing it from voting twice.

The third step features a return to explicitness, because V knows his blinding factor and can therefore decipher what he receives entirely, provided it follows the prescribed structure. If the received message does not follow that structure, it is discarded and yields no reaction. The postcondition includes the first use of the *Anms* event, which prevents the collector from extracting V's identity from the message header.

[6] This unfortunate fact is due to the definition of *used*, which builds on *parts*. Keys used solely for encryption are not extracted by *parts*. Session keys are not in any initial knowledge, hence they are not considered "used" by default in this situation.

Message four is similar to the second one in that the collector can only make generic assumptions about the message usually received from *V*. It consequently causes the same proving difficulties as message two. Publication on a bulletin board is specified by a message sent from an agent, here the collector, to himself. Note that the inductive nature of the model prevents us from formalising the vote collecting deadline: ideally, the collector should only publish commitments once all votes have been received. This separation of phases is not modelled here.

In step five, if *V* sees his commitment on the bulletin board, he reveals his commitment key by sending it over an anonymous channel. The condition *Key c* $\in$ *analz (knows V evs5)* is a crucial one: without it, nothing prevents the Spy from using this protocol step to send (and hence know) a key she did not know in the first place! Indeed, the Spy can act as a regular agent and therefore perform the first protocol step, thus fulfilling the requirements for step five.

Finally, in the sixth and last message, the collector publishes *V*'s vote provided this has not been done already.

6.7 Formal Verification

The inductive model of FOO has been specified. To inspect the security properties of the protocol, statements about the protocol model must be defined and proven. We start with a classic treatment.

6.7.1 Main Classic Results

Studying advanced properties of the protocol required starting with a classic confidentiality treatment that was more complex than expected. This section will emphasise the main results and their proofs.

lemma *unique_Nv2*:

$[\![$*Crypt c* (*Nonce Nv*) $\in$ *parts* (*spies evs*); *Crypt c* (*Nonce Nva*) $\in$ *parts* (*spies evs*);
Key c $\notin$ *analz* (*spies evs*); *evs* $\in$ *foo*$]\!]$

$\Longrightarrow Nv{=}Nva$

The first lemma is a unicity result about ballots. If two ballot commitments with the same commitment key c appear in the network traffic, either the commitment key is not confidential or the ballots are identical.

theorem *Spy_see_c* [*simp*]:
$[\![Says\ V\ Adm\ \{\!|Agent\ V, Crypt\ (priSK\ V)\ (Crypt\ b\ \ (Crypt\ c\ (Nonce\ Nv)))|\!\}\in set\ evs;$
$Anms\ V\ Col\ (Key\ c)\notin set\ evs;\ \ V\notin bad; evs\in foo]\!]$
$\Longrightarrow Key\ c\notin parts\ (spies\ evs)$

This theorem, *Spy_see_c*, tells us that any commitment key used by an honest voter in an instance of the first protocol step does not appear in the protocol traffic as long as the voter has not disclosed it to the collector in step 5.

lemma *unique_c*:
$[\![Crypt\ c\ (Nonce\ Nv)\in parts(spies\ evs);\ \ Crypt\ ca\ (Nonce\ Nv)\in parts(spies\ evs);$
$Nonce\ Nv\notin analz\ (spies\ evs); evs\in foo]\!]$
$\Longrightarrow c{=}ca$

As for ballots, we also obtain a unicity result about commitment keys. Assume two ballot commitments on the same ballot have appeared in the network traffic. Then either this ballot is known to the Spy, or the commitment keys are equal.

We now turn to a strong confidentiality result about the blinding factor b:

theorem *Spy_see_b2* [*simp*]:
$[\![Crypt\ (priSK\ V)\ (Crypt\ b\ (Crypt\ c\ (Nonce\ Nv)))\in parts\ (spies\ evs); V\neq Adm;$
$V\notin bad; evs\in foo]\!]$
$\Longrightarrow Key\ b\notin parts\ (spies\ evs)$

Assume it was used to blind a ballot commitment, which then appeared in the traffic signed by an honest, regular voter. Then the blinding factor itself was never sent over the network in any form (even encrypted).

The following theorem may seem obscure and not so useful. In fact, it was not only crucial to prove the strong confidentiality result about *Nv* (see below), but also required the greatest proof effort of all the classic results about FOO.

theorem *Nonce_secrecy*:

$evs \in foo \Longrightarrow$

$(\forall KK.\ KK \subseteq -\ (range\ shrK) \longrightarrow$

$(\forall K \in KK.\ K \in symKeys \longrightarrow \neg\ KeyWithNonce\ K\ Nv\ evs) \longrightarrow$

$(Nonce\ Nv \in analz\ (Key`KK \cup (knows\ Spy\ evs))) =$

$(Nonce\ Nv \in analz\ (knows\ Spy\ evs)))$

It states that adding keys of a certain class to the Spy's knowledge does not influence her knowledge of a ballot. We will be more precise soon.

This theorem was needed because of our strategy of using session keys. The two different types of implication in the theorem statement, "$\Longrightarrow$" and "$\longrightarrow$", refer to meta-level implication and object-level implication (see §3.1), respectively. Meta-level implication represents inference at the theorem level. The *KeyWithNonce* predicate used here was inspired by the one used by Paulson in his verification of the Yahalom theorem. We define it as follows:

definition $KeyWithNonce :: [key, nat, event\ list] \Rightarrow bool$ **where**

$KeyWithNonce\ c\ Nv\ evs \equiv$

$\exists\ V\ B\ b.\ Says\ V\ B\ \{|Agent\ V, Crypt\ (priSK\ V)\ (Crypt\ b\ (Crypt\ c\ (Nonce\ Nv)))|\} \in set\ evs$

It can be seen that stating *KeyWithNonce c Nv evs* is equivalent to stating that *c* and *Nv* can be found as commitment key and ballot, respectively, in an instance of the first protocol message. They are thus associated. We can now describe the statement of *Nonce_secrecy* more clearly: all keys that are not shared keys, that are symmetric and that do not appear in conjunction with a nonce in an instance of the first protocol message do not influence the Spy's knowledge of that nonce.

Once *Nonce_secrecy* is proven, we are able to verify the confidentiality of any ballot used by an honest voter in the first protocol step as long as the associated commitment key is confidential:

theorem *Nv_secrecy* [*simp*]:
⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;
Key c ∉ *analz* (*spies evs*); *V* ∉ *bad*; *evs* ∈ *foo*⟧
⟹ *Nonce Nv* ∉ *analz* (*spies evs*)

6.7.2 Main Privacy Results

The following theorem, *foo_V_privacy_asynth*, is the culmination of the entire proof process and states that the FOO protocol guarantees pairwise unlinkability to all honest voters that started the protocol.

theorem *foo_V_privacy_asynth*:
⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs;*
a ∈ (*asynth* (*aanalz Spy evs*));
Nonce Nv ∈ *a*; *V* ∉ *bad*; *V* ≠ *Adm*; *V* ≠ *Col*; *evs* ∈ *foo*⟧
⟹ *Agent V* ∉ *a*

More precisely, assume that the regular, honest voter V sent the administrator a message in line with the first step of the protocol, containing a blinded commitment on the vote *Nv*. Also assume that this very vote is in a message set of association syntheses. Then the name of V is not in that set.

Before turning to the proof itself, we focus on the most important proof elements, which are mainly results about associations.

A fundamental result is *foo_V_privacy_aanalz*, which looks similar to the theorem *foo_V_privacy_asynth*.

theorem *foo_V_privacy_aanalz*:
⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;
a ∈ (*aanalz Spy evs*); *Nonce Nv* ∈ *a*; *V* ∉ *bad*; *evs* ∈ *foo*⟧
⟹ *Agent V* ∉ *a*

However, whereas the latter is a statement about *asynth*, hence about association synthesis, the former only considers sets belonging to *aanalz*, that is associations aris-

ing from individual messages. Whenever an honest voter performed the first step of the protocol, the voter's identity and vote cannot be found in the same association set.

The lemma called *asynth_insert* is a direct consequence of the definition of *asynth* quoted in 6.3.2.

lemma *asynth_insert*:

$a \in asynth(insert\ a1\ as) \Longrightarrow$

$(a{=}a1 \vee$

$a \in asynth\ as \vee$

$(\exists\, a2\ m.\ a2 \in as \wedge a = a1 \cup a2 \wedge m \in a1 \wedge m \in a2 \wedge m \neq Agent\ Adm \wedge m \neq Agent\ Col))$

By introducing the various cases that an application of *asynth* may imply, it provides a useful rewrite rule for expressions involving the operator name.

The next three theorems allow more precise reasoning about messages that contain encryption. They are all concerned with the situation where a message yields an association set containing at least one ciphertext. They are necessary for dealing with situations where protocol messages are not completely specified. For instance, an agent may have to transmit an encrypted commitment without even being able to check that the commitment is actually about a vote. In those situations, protocol step specification must model agents' limited knowledge when dealing with sealed messages. However, even when the complete contents of a ciphertext are not known, a number of scenarios can be distinguished. A number of encryption key values and partial knowledge of the ciphertext contents lead to contradictions. For instance, classic results about the protocol reveal that a blinding key used by an honest agent always remains confidential. Possible configurations are therefore made explicit in the following results.

lemma *aanalz_PR*:

$[\![a \in aanalz\ Spy\ evs;\ Crypt\ P\ R \in a;\ evs \in foo]\!] \Longrightarrow$

$(Agent\ Col \notin a \vee$

$(Agent\ V \in a \longrightarrow V \in bad \vee V = Col) \vee$

$(Nonce\ Nv \notin parts\ \{R\})) \wedge$

$((Nonce\ Nv \notin a) \vee$

$(Key\ (invKey\ P) \in analz\ (spies\ evs) \wedge Agent\ V \notin parts\ \{R\}))$

Lemma *aanalz_PR* states properties about the possible forms of a generic ciphertext appearing in any association. Its conclusion is expressed as a conjunction between two predicates that are themselves disjunctions. The first conjunct relates to the presence of an agent name in the association. If the name of the collector appears in the association and any nonce (a vote) is an atomic component of R, then no agents that are both honest and different from the collector can also be in a. The second conjunct states that if any nonce is part of the association, then the Spy must be able to decrypt the ciphertext and no agent name can be an atomic component of R.

lemma *aanalz_AdmPR_V_Nparts*:
$[\![a \in aanalz\ Spy\ evs;\ Crypt\ (priSK\ Adm)\ (Crypt\ P\ R) \in a;\ evs \in foo]\!]$
$\Longrightarrow Nonce\ Nv \notin parts\ \{R\} \vee$
$Key\ (invKey\ P) \notin analz\ (knows\ Spy\ evs) \vee$
$(Agent\ V \in a \longrightarrow V \in bad \vee V = Adm \vee V = Col)$

Then, lemma *aanalz_AdmPR_V_Nparts* relates to the specific case when a ciphertext signed by the administrator is in an association. It establishes a disjunction: either no nonce is an atomic component of the ciphertext's body, or the Spy cannot open the ciphertext inside the signature, or there is no regular, honest agent name in the association.

lemma *aanalz_Adm*:
$[\![Says\ V\ Adm\ \{\!|Agent\ V, Crypt\ (priSK\ V)\ (Crypt\ b\ (Crypt\ c\ (Nonce\ Nv)))|\!\} \in set\ evs;$
$a \in aanalz\ Spy\ evs;\ Agent\ Col \notin a;\ Agent\ V \in a;\ V \notin bad;$
$Crypt\ P\ R \in a;\ Nonce\ Nv \in parts\ \{R\};\ evs \in foo]\!]$
$\Longrightarrow (P = priSK\ V \vee P = priSK\ Adm \vee P = b) \wedge$
$(P \neq priSK\ Adm \vee R = Crypt\ b\ (Crypt\ c\ (Nonce\ Nv)))$

Finally, lemma *aanalz_Adm* is still about associations containing a ciphertext. Like *foo_V_privacy_aanalz*, it binds the variables involved in a version of the first step of the protocol. Assume an association contains the name of an honest agent who already

sent a message corresponding to step one. Also assume it contains a ciphertext *Crypt P R*, and that the nonce from step one is in *parts* of *R*. If the name of the collector is absent from the association, then the following conclusions hold:

- If *P* is neither the signing key of the voter mentioned in the precondition nor the signing key of the administrator, then it must be the blinding factor;
- If *P* is the administrator's signing key, then the body of the ciphertext is exactly the body of the message signed by the voter in the bound first message.

6.7.3 Proof of the Main Theorem

Proving privacy by *foo_V_privacy_asynth* is done, as usual in the Inductive Method, by induction on the protocol model. The full proof of this theorem can be found in §A.3.2.

Every protocol step generates a subgoal. When all subgoals are closed, the theorem is proven. Developing the proof required considerable effort. After eliminating redundancies and streamlining, its number of step was reduced. It will be shown that despite its length, the proving strategy is general, hence reusable for different protocols.

Induction and simplification leaves us with seven subgoals: the six protocol steps, plus *Fake*. The *Fake* is closed thanks to the classic reasoner *blast*. Its proof is simple because messages sent by dishonest agents do not yield associations. Intuitively, the goal of the Spy is to extract plausible associations, not make up new ones. However, it keeps its traditional Dolev-Yao attacker role and influences all usual theorems proven for the protocol; those are used in the privacy proof. What is more, the Spy indirectly influences the associations by sending forged messages to which agents react. For instance, in the second step, the administrator is not able to inspect the received ciphertext and therefore even forwards illegitimate content stemming from messages from the Spy.

The subgoal arising from *EV1* is first simplified by remarking that fresh keys (the blinding factor and commitment key) can never be known to the Spy — they cannot

yet be in the set *analz (knows Spy evs1)*. We then perform a case split about the agent *Va* involved in the instance of the first protocol step generated by this subgoal. If *Va* is dishonest (a member of the *bad* set), then the message it sent yields no new association and the subgoal concludes thanks to the inductive hypothesis. If *Va* is an honest agent, we must apply, for the first time, *asynth_insert*. This lemma is of recurrent use throughout the proof because it allows us to split the *asynth* set. For instance, this stage of the proof features the following precondition:

a ∈ *asynth (insert*
{{|*Agent Va, Crypt (priSK Va) (Crypt ba (Crypt ca (Nonce Nva)))*|},
Agent Va, Crypt (priSK Va) (Crypt ba (Crypt ca (Nonce Nva))),
Agent Va, Agent Adm, Crypt ba (Crypt ca (Nonce Nva))}
(aanalz Spy evs1))

Let us call X the set such that *a* ∈ *asynth (insert X (aanalz Spy evs1))*.

Applying *asynth_insert* leaves us with three possibilities:

1. *a* = *X*.
2. *a* ∈ *asynth (aanalz Spy evs1)*.
3. There exists *a2* in *aanalz Spy evs1* and an element *m* such that *a* is the union of *a2* and X and *m* is both in X and in *a2*.

The inductive hypothesis tells us that *Nv* is in *a* and X contains no nonces, so the first disjunction is excluded. The second disjunction is eliminated thanks to the lemma *nv_fresh_a2*, not quoted here, which states that fresh nonces do not appear in association syntheses.

If *a* is a union, more precision is required. First, if the agent *V* from the inductive hypothesis and the agent *Va* introduced by the induction are different, then $V \notin X$ and therefore *V* must be in *a2*. Since *Nv* is also in *a2*, *foo_V_privacy_aanalz* leads us to a contradiction.

Otherwise, *V* = *Va*. If *Nv* and *Nva* are equal, *Nv* must be fresh like *Nva*. The auxiliary lemma *aanalz_traffic*, according to which elements in associations which are not

agent names must have appeared in the traffic, solves this case (fresh elements never appeared in network traffic). On the other hand, if $Nv \neq Nva$, the element in common m can be any of the elements in X. We appeal to another lemma, *association_Nv*. It is specifically tailored for this subgoal, used only here, and shows that an association containing a nonce cannot additionally contain any of the possibilities for m listed here except for V. Together with *foo_V_privacy_aanalz*, that takes care precisely of the case $m = V$, this solves the subgoal.

The use of *asynth_insert* to split the association synthesis is a technique used for all subgoals of the theorem. It turns out that the third disjunction generates the bulk of the proving work for the remaining subgoals. We will therefore focus on it. It requires taking a close look at the structure of sets in *aanalz*.

Subgoals arising from protocol steps two and four are much larger than the other ones because of the generic specification of the steps. For instance, in the second protocol step, the administrator has received a signed ciphertext from the voter. The administrator can extract the ciphertext from the signature, but has no means in general to look inside. We only assume that it is possible to know that the ciphertext contains no more than one atomic component, by inspecting its length. However, the precise nature of the plaintext is unknown in general and this generality in the specification of the inductive step explains the additional complexity of the proof. It is necessary if the precondition is to be realistic. Likewise, in step four, the collector receives a signed ciphertext that he cannot open in general. The concrete consequence in terms of association syntheses is that potential common elements m are not listed explicitly in the goal preconditions. Instead of belonging to a finite set of bound variables, only partial information is known about them. For instance, we may only know that an element m can be deduced from some ciphertext via *analzplus*. By contrast, for non-generic protocol steps, we obtain an explicit set and the proof is much easier.

A number of results about elements in *aanalz* are available, such as *aanalz_PR* and *aanalz_Adm*. These theorems are stated with weak premises and offer a number of conclusions as disjunctions. The most systematic proof strategy is therefore to perform

case splits about the ciphertext contained in *aanalz*. As this is done, one can reason more precisely about encryption keys and plaintexts until a contradiction is reached thanks to the aforementioned results. One crucial distinction is whether the name of the collector appears in the association. If such is the case, the elements in *aanalz* arose from the collection step *EV4*. Conversely, if *Agent Col* $\notin$ *a2*, the association set in the precondition was generated by another protocol step. The encryption key *P* from the ciphertext *Crypt P R*, assumed to be in an association, is then compared in turn to the voter's signing key, the administrator's signing key, and to the blinding factor. Contradictions are reached in every case. The value of the payload *R* is also compared with the voter's blinded vote commitment *Crypt b (Crypt c (Nonce Nv))*. Those different situations obviously refer to various ciphertext values naturally generated by the protocol steps. In essence, the proving strategy amounts to zooming in sufficiently into the various possible association configurations to uncover contradictions that are not apparent at a more general level.

The outline of this proving strategy is not dependent on a given protocol. Let us recall the important steps:

1. For every subgoal, split the association syntheses set *asynth* using *asynth_insert*.

2. The subgoals arising from explicit protocol steps are straightforward to close because the set of potential common elements *m* becomes explicit as well.

3. For more general subgoals, case splits about the possible values of initially generic ciphertexts are combined with lemmas describing their structure in associations in a systematic way.

6.7.4 Proof of the Supporting Theorems

Rather than describing the full proof of every theorem required for the privacy one, we focus on *aanalz_PR* due to space constraints. It is of repeated use in the privacy proof, appearing in it eleven times, and its proof exemplifies the kind of reasoning required for the other supporting theorems. Recall its statement from earlier (6.7.2); it details

the form of elements of *aanalz* that contain a ciphertext.

As expected, complications arise again from the generic steps, namely *EV2* and *EV4*. As the other subgoals are easier to prove, let us concentrate on *EV2*, as the proof for *EV4* is similar.

We require the following subsidiary results in addition to standard lemmas from the existing Inductive Method framework:

- *analzplus_into_parts*: Elements in the set *analzplus X ks* (recall *ks* is the external key set) are in *parts X*.

- *no_pairs*: If a message contains only atomic components and already contains an agent name in its *parts* set, a number of other elements cannot be in the *parts* set. Specifically:

$$
\begin{aligned}
&[\![\mathit{Agent\ Va} \in \mathit{parts}\{R\}; \forall A\ B.\ \{\!|A, B|\!\} \notin \mathit{parts}\ \{R\};\\
&\ \mathit{Crypt}\ (\mathit{priSK}\ V)\ (\mathit{Crypt}\ \mathit{ca}\ (\mathit{Nonce}\ \mathit{Nv})) \in \mathit{parts}\ \{R\} \vee\\
&\ \mathit{Crypt}\ (\mathit{priSK}\ V)\ (\mathit{Crypt}\ b\ (\mathit{Crypt}\ c\ (\mathit{Nonce}\ \mathit{Nv}))) \in \mathit{parts}\ \{R\} \vee\\
&\ \mathit{Crypt}\ b\ (\mathit{Crypt}\ c\ (\mathit{Nonce}\ \mathit{Nv})) \in \mathit{parts}\ \{R\} \vee\\
&\ \mathit{Number}\ N \in \mathit{parts}\ \{R\} \vee\\
&\ \mathit{Nonce}\ \mathit{Nv}\ \in \mathit{parts}\ \{R\} \vee\\
&\ (\mathit{Agent}\ T \in \mathit{parts}\ \{R\} \wedge T \neq \mathit{Va}) \vee\\
&\ \mathit{Crypt}\ \mathit{ca}\ (\mathit{Nonce}\ \mathit{Nv}) \in \mathit{parts}\{R\}]\!] \Longrightarrow \mathit{False}
\end{aligned}
$$

- *analzplus_Nv*: Assume an *analzplus Q (analz H)* set contains a ciphertext and a nonce. If *parts Q* contains only atomic components, then the decryption key of the ciphertext must be in *analz (insert Q H)*.

We can now describe the proof of the second subgoal. The case where the administrator (*Adm*) is dishonest is closed easily using a standard property of *analz* and the fact that he is the message sender in *EV2*. Else, we must distinguish cases on the basis of the origin of the association in the inductive hypothesis. The first possibility is that it was generated by the *EV2* message introduced by the induction. In that case, the key *P* in the *aanalz_PR* theorem statement could either be the administrator's signing

key, or the encryption key of the ciphertext that he signed. A third possibility is that the entire *Crypt P R* is embedded deeper in the signed ciphertext. Let us examine each possibility in turn.

In the first case, we must show that an agent name (either the collector or a regular voter) and a nonce cannot be present in *analzplus R* at the same time. This is shown by combining *analzplus_into_parts* and *no_pairs*. In the second case, the ciphertext *Crypt P R* from the theorem precondition is exactly the ciphertext *Crypt Pa Ra* signed by the administrator in this version of the second protocol step. Disentangling the precondition conjunction leads to the same scenario and an additional one that entails proving that if the inverse of key *P* is known to the attacker in the first place, it is all the more known to her after getting hold of *R*.

If *Crypt P R* is embedded in the ciphertext generated by the administrator, we must perform a few additional case splits but the line of reasoning is the same, with the additional use of *analzplus_Nv* which allows us to reason about the properties of *analzplus*.

It is clear that even though specifying the possible forms of elements in *aanalz* requires inspecting a number of scenarios, the proving process is straightforward once some crucial building blocks are established. Notably, the three subsidiary results we listed earlier (*analzplus_into_parts* and so on) are stated without any reference to the FOO protocol — they are protocol-independent and can be reused directly.

6.8 Comparison

In this section, we compare features of our approach with the ones of two established methods, relying on the ProVerif and AKiSs tools.

Precision Since ProVerif systematically uses under-approximations[7], it is not precise in general. While it will not deem a flawed protocol correct, it may fail to verify a correct protocol due to the detection of a false attack. For AKiSs, precision depends

[7]This means that a stronger property than the desired one is checked.

on the class of the process modelling the protocol. Those which can be modelled using determinate processes can be checked directly for observational equivalence and trace equivalence ($\approx_t$) because of their coincidence with coarse trace equivalence ($\approx_{ct}$) in that class. On the other hand, some e-voting protocols, particularly those using phases like FOO, must be under-approximated by fine-grained trace equivalence ($\approx_{ft}$) because they do not lead to determinate processes. In that case, like for ProVerif, the risk of spurious attack detection remains. Unlinkability in the Inductive Method is treated by inductive theorem proving, so there can be no false positives: when it doesn't hold, lemmas stating it cannot be proven. However, precision requires using the generalised association synthesiser. When unlinkability cannot be established, attacks are not given explicitly but examination of the remaining subgoal allows the user to trace back the flaw to the problematic protocol step. This interactivity gives greater insight into the protocol's intricacies but requires more effort than the aforementioned tools. Another aspect of precision is session bounding. While the Inductive Method and ProVerif both support unlimited protocol sessions, the computational cost of AKiSs blows up exponentially when more sessions are interleaved.

Automation vs. interaction AKiSs is fully automated. Privacy in ProVerif was checked by hand in [76]. An automatic verification in ProVerif was presented in [46], but a translation algorithm is involved and its correctness is not formally proven. In [44], a ProVerif privacy proof for a fixed number of voters is partially automated. The Inductive Method is mechanised in Isabelle/HOL, an interactive theorem-prover. As noted above, this requires user interaction at proof development time, whereas ProVerif and AKiSs require it at tool development time.

Termination Termination is guaranteed neither in ProVerif nor AKiSs. While ProVerif features a resolution method that ensures termination for a specific syntactic transformation of protocol models, this method is limited to secrecy and authentication properties [23]. Since the Inductive Method is based on induction, there is no termination issue if the user is sufficiently proficient.

Supported cryptographic primitives Associative/commutative (A/C) operators such as exponentiation or XOR are currently supported by only one of the tools, ProVerif [79], despite their usefulness for protocol modelling. Blind signatures, which are commonly used in e-voting protocols, are supported by all three tools compared in this chapter. AKiSs supports a larger variety of cryptographic primitives than ProVerif and the current version of the Inductive Method. In [36], Chadha et al. conjecture that all primitives that can be modelled in a rewrite system fulfilling a specific convergence property are supported. Notably, trapdoor commitments can be modelled. By contrast, supporting new cryptographic primitives in the Inductive Method requires modelling them through additional rules using existing primitives or changing the underlying framework. The latter option requires more work since specific assumptions about the interaction of operators are made in the theory *Message*, which is always imported. On the other hand, major modifications have been performed successfully, for instance in [86], demonstrating the flexibility of the Method. New cryptographic primitives can be added easily to the applied pi calculus by devising new equational theories, but the resulting model may be beyond the scope of the tool's automatic analysis [45].

Efficiency The authors of [36] state that the automated analysis of privacy for FOO requires a few minutes on a modern laptop. Loading and verifying all FOO theories in Isabelle/HOL takes a similar time. However, our FOO theory also establishes other useful security guarantees about the protocol. Another metric is the respective human effort necessary to formalise protocols and their goals before the actual analysis. This aspect remains to be quantified precisely in our approach, but the modelling step of the verification process is clearly the quickest, usually done in a few days. On the other hand, the actual proof process is a matter of weeks if the framework exists, and the framework development itself took us months.

Synthesis Table 6.1 summarises our comparison but does not aim for exhaustivity.

[1] Those that can be modelled in an optimally reducing convergent rewrite system.

	ProVerif	AKiSs	Inductive Method
Precision	Precise for determinate processes, else under-approximation	Under-approximation	Precise for unbounded association synthesis
Operation	Partial automation	Full automation	Interactive proving
Unbounded sessions	Yes	No	Yes
Systematic termination	No	No	Yes, assuming user proficiency
Supported cryptographic primitives	Usual + Blind signatures, bit commitments, proxy reencryption	Large set conjectured[1]	Usual + Blind signatures, bit commitments

Table 6.1: Synthesis of characteristics of mechanised FOO privacy analyses

6.9 Discussion

We have presented the first interactive theorem proving-based analysis of voter privacy, to offer an alternative means of investigation to consolidated work based on process equivalence. Privacy is modelled as an unlinkability property between a voter and her ballot. Extensions to the Inductive Method are implemented in Isabelle/HOL to specify associations between elements and combinations of associations that share a common element.

The initial proof development effort was significant, but a coherent line of reasoning emerges from the proof. This general strategy and a number of protocol-independent results about the new operators support the case of re-usability for other e-voting protocols. Interactive proofs entail a level of clarity about protocol scenarios that is unavailable from automatic tools. The inductive nature of our specification eliminates termination issues or inherent size limitations. While the benefits of automated tools are clear, our approach sheds a complementary light on voter privacy by its operational view.

Other privacy-type properties such as receipt-freeness and coercion-resistance ought

to be specified in the Inductive Method. Additionally, e-voting protocols that are not amenable to analysis in the process equivalence model must be studied in our framework to investigate its domain of applicability. We would also like to program some of the recurring proof steps as ML tactics. Part of this work was published in [33].

Chapter 7

Discussion

We contributed to the formal analysis of security protocols by extending the Inductive Method in a number of ways. After outlining the problem of network security and issues specific to security protocols, we presented a selection of existing protocol analysis techniques, with an emphasis on the Inductive Method and its implementation in the Isabelle/HOL theorem prover.

A way to verify composed protocols in the Inductive Method was then presented. The holistic analysis of a certification protocol composed with a simple authentication protocol was performed to present empirical evidence.

We then turned to the problem of cryptographic primitive specification, exemplified by a comparative analysis of two versions of an ISO/IEC 9798-3 protocol. One of these versions featured auditable identity-based signatures.

Our main contribution, the extension of the Inductive Method to deal with electronic voting protocols, was then introduced. Another new cryptographic primitive, blind signatures, was specified as a protocol step after an initial attempt to specify it in the message datatype. Privacy, seen as unlinkability between two pieces of information, was introduced and compared to the indistinguishability approach. We then described our model of the FOO protocol. After proving a number of classic confidentiality properties about the protocol model, we focused on unlinkability.

7.1 Domain of Applicability of the Inductive Method

General considerations about the Inductive Method can be made from this work. A progressive realisation was the difficulty in tailoring the message datatype and operators in the *Message* theory to our needs. New applications frequently require modelling cryptographic primitives that were never used before in the framework. The natural way of specifying additional cryptographic primitives is to adapt the part of the framework that prescribes the rules governing them: the inductive definitions of the *analz*, *parts* and *synth* message operators. Those definitions, however, cannot be changed in isolation: they are all interdependent and many important lemmas describe their properties. These lemmas are then necessary for the proving process to deal with the threat model.

To be more specific, the *spy_analz* tactic (very often used when trying to close the subgoal arising from the *Fake* protocol rule) relies on lemmas, such as *Fake_parts_insert*, that themselves build upon other lemmas. For instance, *Fake_parts_insert* requires the lemma *parts_synth* which prescribes a specific relationship between *parts* and *synth*, namely *parts (synth H) = parts H* $\cup$ *synth H*. This type of expected relationship between message operators has often been the decisive bottleneck in our modification attempts. One cannot simply dispose of lemmas like *parts_synth*, as they provide the automation that is needed for a reasonably speedy proving process. Programming substitute tactics for the *Fake* case could solve this issue, but seems like a major undertaking. Worse, it may even be impossible to prove security goals that would otherwise hold if such major changes are made to operator interplay.

The alternative, then, is to only perform changes that preserve operator dependencies. Inevitably, the range of modifications that can be done are limited if one is to keep the current framework. Empiric evidence is provided by the fact that two of the new cryptographic primitives that we needed to model in the scope of this thesis did not preserve these dependencies.

However, as stated earlier, changing imported theories is not the only way of intro-

ducing new cryptographic primitives. Specifying them as part of the protocol model, as we did for blind signatures, can be much easier and greatly limit the risks of unwanted side-effects. Determining the range of primitives that can be specified in such a way remains to be done, but it seems reasonable to try this more straightforward approach first, before any framework changes. In terms of costs, specifying cryptographic primitives in the protocol model is less aesthetically satisfying: the extension is less integrated in the framework. We did not encounter other negative consequences. Integration still seems sufficiently tight for the threat model to be kept intact.

Our work also shows that the Inductive Method can be applied to previously unexplored protocol classes, such as composed or electronic voting ones.

7.2 Future Work

Much work remains to make the capabilities of the Inductive Method more comprehensive. One task would consist in gathering more formal evidence for the hypothesis just discussed — that cryptographic primitive specification at the protocol level is not detrimental to the validity of the threat model. This involves ascertaining that there are no cases were essential rewriting is not being done due to the less tight integration between the new cryptographic primitive and the definition of *msg*.

7.2.1 Protocol Composition

In the case of protocol composition, inductive models involving a larger number of protocols should be analysed (our case study in Chapter 4 featured only two). To get a sense of the boundaries of this approach, it would be profitable to focus on composed protocols that are too complex to be analysed automatically by tools such as Scyther, and too intricate to benefit from the composition theorem described by Cremers.

7.2.2 Electronic Voting

Major work also remains to be done for broader support of electronic voting protocols and their properties. In this thesis, we only addressed voter privacy; natural avenues for further research involve considering receipt-freeness and coercion-resistance. Those properties are similar but are expected to require different modellings. For instance, in the case of coercion-resistance, the target security property is that a voter cannot prove to a coercer how he voted, even assuming cooperation of the voter with the coercer. This cooperation aspect could be modelled by allowing the voter to share private information with the coercer — here we are reminded of the *bad* set (agents in *bad* share their private knowledge with the Spy).

Furthermore, much light would be shed on the potential of our developments by analysing other electronic voting protocols. As in the case of protocol composition, one should focus on what is currently out of reach of existing methods. Since process equivalence methods are currently the most common, investigating the protocols that these methods are currently unable to tackle, such as the ones featuring state changes, would reveal how much complementarity exists between the two strategies.

Generalising the Association Synthesiser The *asynth* operator should be generalised to take into account unlimited layers of association synthesis. The definition presented earlier does not account for transitive associations:

- $\{\!|$*Agent V, Number T*$|\!\}$ in *as* and $\{\!|$*Number T, Nonce N*$|\!\}$ in *as* implies $\{\!|$*Agent V, Number T, Nonce N*$|\!\}$ in *asynth as* (the union is performed for *m* = *Number T*).
- However, $\{\!|$*Agent V, Number T*$|\!\}$ in *as* and $\{\!|$*Number T, Crypt K X*$|\!\}$ in *as* and $\{\!|$*Crypt K X, Nonce N*$|\!\}$ in *as* does not imply $\{\!|$*Agent V, Number T, Crypt K X, Nonce N*$|\!\}$ in *asynth as*.

A natural generalised definition could be as follows:

inductive_set

asynth :: *msg set set* $\Rightarrow$ *msg set set*

for *as* :: *msg set set*

where

asynth_Base [*intro*]: $a \in as \Longrightarrow a \in asynth\ as$

| *asynth_Build* [*intro*]: $[\![a1 \in asynth\ as; a2 \in asynth\ as; m \in a1; m \in a2;$
$m \neq Agent\ Adm; m \neq Agent\ Col]\!]$
$\Longrightarrow a1 \cup a2 \in asynth\ as$

In practice, we cannot prove unlinkability using this definition because statements over *asynth(aanalz)* never get broken down to statements over *aanalz*. The reason is that any element in *asynth* may be the union of other elements in *asynth*. Rewriting seems to loop for this reason. An alternative definition is therefore needed. Preliminary tests were more successful with definitions such as this one:

inductive_set

asynth2 :: *msg set set* $\Rightarrow$ *msg set set*

for *as* :: *msg set set*

where

asynth2_Base [*intro*]: $a \in as \Longrightarrow a \in asynth2\ as$

| *asynth2_Build* [*intro*]: $[\![a1 \in asynth2\ as; a2 \in asynth2\ as; m \in a1; m \in a2; \forall\ X.\ as \neq \{X\};$
$\sim (a1 \subseteq a2);\ \sim (a2 \subseteq a1); m \neq Agent\ Adm; m \neq Agent\ Col;$
$\forall\ z.\ z \in a1 \cup a2 \longrightarrow\ (\exists\ f.\ f \in as \wedge z \in f)]\!]$
$\Longrightarrow a1 \cup a2 \in asynth2\ as$

The crucial precondition here is the last one: every element appearing in one of the association synthesis sets must find its origin somewhere in the initial base set. Another important precondition (the fifth one) states that the *Build* rule is not applicable to the situation where *as* is a singleton. This behaviour is justified by the fact that there is no point building unions over a set of sets if that set is a singleton.

While this version of a generalised *asynth* was fairly successful on simple case studies, proof complexity grew quickly once we tried more advanced examples. This was somewhat surprising giving the relative simplicity of the operator. It is not clear what additional tweaks are needed to obtain a generalised *asynth* operator that can be

used efficiently for unlinkability proofs of full protocols. We plan to continue investigating this question.

7.2.3 Framework Evolution

A number of evolutions of the Inductive Method were suggested by our work. Some were mentioned earlier in this thesis explicitly, some only in private conversation.

Threat models diversity The Dolev-Yao attacker model should be one available threat models out of a few, not the only one. The Dolev-Yao model was created at a time when computer networks were structured very differently from now. Current network configurations such as the Internet call for a threat model with more than one attacker. Whether attackers cooperate or compete should also be optional. A more modern incarnation of the Inductive Method would feature all these threat models, and allow the analyst to select the one more consistent with the real-life implementations of the protocol under analysis.

Cryptographic primitives support The question of cryptographic primitives specification should be settled. Either it is accepted that all new primitives can be modelled at the protocol level without negative impact, and then this should be the canonical strategy; or the framework theories should be revamped to include a much broader selection of cryptographic primitives, to cover at larger fraction of their use in current protocols. Special constructions like XOR and exponentiation cannot be specified in the Inductive Method as of now; adding support for them would be one more significant improvement.

Specialisation More generally speaking, the way forward for protocol verification may be a clearer identification of strengths and weaknesses of available methods and tools, conducive to a more marked specialisation of the latter. For the Inductive Method, this would mean focusing more narrowly than now on protocols or threat models that other tools struggle with. The research community of the field would

benefit from increased productivity, and the specific benefits of various tools would become more explicit.

7.3 Conclusion

The formal analysis of security protocols is a broad topic. New applications frequently appear, bringing with them the need for new security guarantees, models and specifications. The field has shown much progress recently, but no single tool is capable of dealing with all protocol classes and properties. A need for specialisation therefore remains. In this thesis, we extended the Inductive Method's domain of application to deal with new cryptographic primitives, protocol composition and aspects of electronic voting. While requiring more user interaction at the time of proof development than some other formal methods, interactive theorem proving provides detailed insight into protocols and their properties. The technicalities of formal methods should not divert from the concrete, real-world need for security guarantees, especially for sensitive applications such as electronic voting. With its flexibility and specificity, the Inductive Method bears potential for further advancements of the field.

Appendix A

Isabelle Theories

For adaptations of existing theories such as *Message*, *Event* or *Public*, only the main modified fragments are shown here.

A.1 Proofs for the Protocol Composition Case Study

A.1.1 Certification.thy

theory *Certification* **imports** *Public* **begin**

abbreviation

CA :: *agent* **where** *CA* == *Server*

inductive_set *cert* :: *event list set*

where

Nil: [] ∈ *cert*

| *Fake*: ⟦*evsf* ∈ *cert*; *X* ∈ *synth* (*analz* (*knows Spy evsf*))⟧
⟹ *Says Spy B X* # *evsf* ∈ *cert*

| *Cert1*: *evsc1* ∈ *cert*
⟹ *Says A CA* {|*Agent A*, *Agent B*|} # *evsc1* ∈ *cert*

| *Cert2*: ⟦*evsc2* ∈ *cert*; *Gets CA* {|*Agent A*, *Agent B*|} ∈ *set evsc2*;
A ≠ *B*⟧

$\Longrightarrow$ *Says CA A*

$\{\!|Crypt\ (priSK\ CA)\ \{\!|Key\ (pubEK\ A), Agent\ A|\!\},$

$Crypt\ (priSK\ CA)\ \{\!|Key\ (pubEK\ B), Agent\ B|\!\}\}$

$\#\ evsc2 \in cert$

| *Reception*:

$[\![evsr \in cert; Says\ A\ B\ X \in set\ evsr]\!]$

$\Longrightarrow Gets\ B\ X\ \#\ evsr \in cert$

lemma $A \neq B \Longrightarrow \exists\, evs \in cert.\ Says\ CA\ A\ \{\!|Crypt\ (priSK\ CA)\ \{\!|Key\ (pubEK\ A), Agent\ A|\!\},$

$Crypt\ (priSK\ CA)\ \{\!|Key\ (pubEK\ B), Agent\ B|\!\}\} \in set\ evs$

apply (*intro exI bexI*)

apply (*rule_tac* [2] *cert.Nil* [*THEN cert.Cert1*, *THEN cert.Reception*,

THEN cert.Cert2], *possibility*)

done

lemma *Says_CA_cert1*:

$[\![Says\ CA\ A\ \{\!|Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ A|\!\},$

$certB|\!\} \in set\ evs; evs \in cert]\!]$

$\Longrightarrow K = pubEK\ A$

apply (*erule rev_mp*)

apply (*erule cert.induct*, *simp_all*)

done

lemma *Says_CA_cert2*:

$[\![Says\ CA\ A\ \{\!|certA, Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ B|\!\}\}$

$\in set\ evs; evs \in cert]\!]$

$\Longrightarrow K = pubEK\ B \wedge A \neq B$

apply (*erule rev_mp*)

apply (*erule cert.induct*, *simp_all*)

done

lemma *Says_CA_cert*:

⟦*Says CA A* {|*Crypt* (*priSK CA*) {|*Key K1*, *Agent A*|},

Crypt (*priSK CA*) {|*Key K2*, *Agent B*|}|} ∈ *set evs*;

evs ∈ *cert* ⟧

⟹ *K1* = *pubEK A* ∧ *K2* = *pubEK B* ∧ *A* ≠ *B*

apply (*blast dest*: *Says_CA_cert1 Says_CA_cert2*)

done

declare *analz_into_parts* [*dest*]

declare *Fake_parts_insert_in_Un* [*dest*]

lemma *Spy_see_priSK* [*simp*]:

evs ∈ *cert* ⟹ (*Key* (*priSK A*) ∈ *parts* (*knows Spy evs*)) = (*A* ∈ *bad*)

by (*erule cert.induct*, *simp_all*, *force*)

lemma *Spy_analz_priSK* [*simp*]:

evs ∈ *cert* ⟹ (*Key* (*priSK A*) ∈ *analz* (*knows Spy evs*)) = (*A* ∈ *bad*)

by *auto*

lemma *cert_authentic*:

⟦*Crypt* (*priSK CA*) {|*Key K*, *Agent A*|} ∈ *parts*(*knows Spy evs*);

evs ∈ *cert*⟧

⟹ (∃ *certB*.

Says CA A {|*Crypt* (*priSK CA*) {|*Key K*, *Agent A*|}, *certB*|}

∈ *set evs*)

∨

(∃ *B certB*.

Says CA B {|*certB*, *Crypt* (*priSK CA*) {|*Key K*, *Agent A*|}|}

∈ *set evs*)

apply (*erule rev_mp*)

apply (*erule cert.induct*, *simp_all*)

apply (*blast dest*: *Spy_see_priSK*)

apply *blast*

done

lemma *Gets_imp_Says* :

$[\![\ Gets\ B\ X \in set\ evs;\ evs \in cert\ ]\!] \Longrightarrow \exists A.\ Says\ A\ B\ X \in set\ evs$

apply (*erule rev_mp*)

apply (*erule cert.induct*, *auto*)

done

lemma *Gets_imp_knows_Spy*:

$[\![\ Gets\ B\ X \in set\ evs;\ evs \in cert\ ]\!] \Longrightarrow X \in knows\ Spy\ evs$

apply (*blast dest*!: *Gets_imp_Says Says_imp_knows_Spy*)

done

lemma *Gets_cert_authentic_Fake*:

$[\![Gets\ A\ (Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ B|\!\}) \in set\ evs;$

$evs: cert]\!]$

$\Longrightarrow K = pubEK\ B$

apply (*blast dest*: *Gets_imp_knows_Spy* [*THEN parts.Inj*]

cert_authentic

Says_CA_cert1 Says_CA_cert2)

done

lemma *Says_knows*:

$A \neq Spy \Longrightarrow knows\ A\ (Says\ Spy\ B\ X\ \#\ evs) = knows\ A\ evs$

apply (*unfold knows_def*)

apply *auto*

done

lemma *Says_knows2*:

$A \neq Spy \wedge A \neq B \Longrightarrow knows\ A\ (Says\ B\ C\ X\ \#\ evs) = knows\ A\ evs$

apply (*unfold knows_def*)

apply *auto*

done

lemma *Gets_knows*:

$A \neq Spy \wedge A \neq Ba \Longrightarrow knows\ A\ (Gets\ Ba\ X\ \#\ evs) = knows\ A\ evs$

apply (*unfold knows_def*)

apply *auto*

done

lemma *cert_authentic_agent*:

$[\![Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ B|\!\} \in parts(knows\ A\ evs);$

$evs \in cert]\!]$

$\Longrightarrow (\exists\ D\ certB.$

$Says\ CA\ D\ \{\!|certB, Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ B|\!\}|\!\}$

$\in set\ evs)$

$\vee$

$(\exists\ certB.$

$Says\ CA\ B\ \{\!|Crypt\ (priSK\ CA)\ \{\!|Key\ K, Agent\ B|\!\}, certB|\!\}$

$\in set\ evs)$

apply (*erule rev_mp*)

apply (*case_tac A = Spy*)

apply (*erule cert.induct*, *simp_all*)

apply *spy_analz*

apply *blast*

apply (*erule cert.induct*, *simp_all*)

apply (*metis Crypt_notin_initState*)

apply *clarsimp*

apply (*metis Says_knows*)

apply (*case_tac Aa≠A*)

apply (*metis Says_knows2*)

apply (*simp add*: *knows_Says*)

apply (*case_tac A = CA*)

apply (*simp add*: *knows_Says*)

apply *blast*

apply (*metis Says_knows2*)

apply (*case_tac Crypt* (*priSK CA*) {|*Key K*, *Agent B*|} ∈ *parts* (*knows Spy evsr*))

apply (*drule cert_authentic*, *assumption*)

apply *blast*

apply *clarsimp*

apply (*case_tac A* ≠ *Ba*)

apply (*metis Gets_knows*)

apply (*case_tac Crypt* (*priSK CA*) {|*Key K*, *Agent B*|} ∈ *parts* (*insert X* (*knows A evsr*)))

apply *clarsimp*

apply (*blast dest*: *Says_imp_parts_knows_Spy parts_cut*)

apply (*metis knows_Gets*)

done

end

A.1.2 Cert_NS_Public.thy

theory *Cert_NS_Public* **imports** *Certification* **begin**

inductive_set *ns_public* :: *event list set*

where

| *NS1*: ⟦*evs1* ∈ *ns_public*; *Nonce NA* ∉ *used evs1*; *evsca* ∈ *cert*;
Crypt (*priSK CA*) {|*Key K*, *Agent B*|} ∈ *parts*(*knows A evsca*)⟧
⟹ *Says A B* (*Crypt K* {|*Nonce NA*, *Agent A*|}) # *evs1* ∈ *ns_public*

| *NS2*: ⟦*evs2* ∈ *ns_public*; *Nonce NB* ∉ *used evs2*; *evscb* ∈ *cert*;
Gets B (*Crypt* (*pubEK B*) {|*Nonce NA*, *Agent A*|}) ∈ *set evs2*;
Crypt (*priSK CA*) {|*Key K*, *Agent A*|} ∈ *parts*(*knows B evscb*)⟧
⟹ *Says B A* (*Crypt K* {|*Nonce NA*, *Nonce NB*, *Agent B*|}) # *evs2* ∈ *ns_public*

| *NS3*: ⟦*evs3* ∈ *ns_public*;
Says A B (*Crypt K* {|*Nonce NA*, *Agent A*|}) ∈ *set evs3*;

$Gets\ A\ (Crypt\ (pubEK\ A)\ \{|Nonce\ NA, Nonce\ NB, Agent\ B|\}) \in set\ evs3\rrbracket$
$\Longrightarrow Says\ A\ B\ (Crypt\ K\ (Nonce\ NB))\ \#\ evs3 \in ns\_public$

declare *knows_Spy_partsEs* [*elim*]

declare *knows_Spy_partsEs* [*elim*]

declare *analz_into_parts* [*dest*]

declare *Fake_parts_insert_in_Un* [*dest*]

declare *image_eq_UN* [*simp*]

lemma *Spy_see_priEK* [*simp*]:

$evs \in ns\_public \Longrightarrow (Key\ (priEK\ A) \in parts\ (knows\ Spy\ evs)) = (A \in bad)$

by (*erule ns_public.induct*, *auto*)

lemma *Spy_analz_priEK* [*simp*]:

$evs \in ns\_public \Longrightarrow (Key\ (priEK\ A) \in analz\ (knows\ Spy\ evs)) = (A \in bad)$

by *auto*

lemma *no_nonce_NS1_NS2* [*rule_format*]:

$evs \in ns\_public \Longrightarrow$
$Crypt\ (pubEK\ C)\ \{|NA', Nonce\ NA, Agent\ D|\}$
$\in parts\ (knows\ Spy\ evs) \longrightarrow$
$Crypt\ (pubEK\ B)\ \{|Nonce\ NA, Agent\ A|\}$
$\in parts\ (knows\ Spy\ evs) \longrightarrow$
$Nonce\ NA \in analz\ (knows\ Spy\ evs)$

apply (*erule ns_public.induct*, *simp_all*)

apply (*blast intro*: *analz_insertI*)+

done

lemma *unique_NA*:

$\llbracket Crypt(pubEK\ B)\ \{|Nonce\ NA, Agent\ A|\} \in parts(knows\ Spy\ evs);$
$Crypt(pubEK\ C)\ \{|Nonce\ NA, Agent\ D|\} \in parts(knows\ Spy\ evs);$
$Nonce\ NA \notin analz\ (knows\ Spy\ evs);\ evs \in ns\_public\rrbracket$
$\Longrightarrow A=D \wedge B=C$

apply (*erule rev_mp*, *erule rev_mp*, *erule rev_mp*)

apply (*erule ns_public.induct*, *simp_all*)

apply (*blast intro*: *analz_insertI*)+

done

lemma *Says_A_pubEK_B*:

⟦*Says A B* (*Crypt K* {|*Nonce NA*, *Agent A*|}) ∈ *set evs*;

A ∉ *bad*; *evs* ∈ *ns_public* ⟧

⟹ *K* = *pubEK B*

apply (*erule rev_mp*)

apply (*erule ns_public.induct*, *simp_all*, *spy_analz*)

apply (*case_tac Says A B* (*Crypt K* {|*Nonce NA*, *Agent A*|}) ∈ *set evs1*)

apply *blast*

apply *clarsimp*

apply (*drule cert_authentic_agent*, *assumption*)

apply (*erule disjE*)

apply (*blast dest*: *Says_CA_cert1 Says_CA_cert2*)+

done

lemma *Says_B_pubEK_A*:

⟦*Says B A* (*Crypt K* {|*Nonce NA*, *Nonce NB*, *Agent B*|}) ∈ *set evs*;

B ∉ *bad*; *evs* ∈ *ns_public* ⟧

⟹ *K* = *pubEK A*

apply (*erule rev_mp*)

apply (*erule ns_public.induct*, *simp_all*, *spy_analz*)

apply (*case_tac Says B A* (*Crypt K* {|*Nonce NA*, *Nonce NB*, *Agent B*|}) ∈ *set evs2*)

apply *blast*

apply *clarsimp*

apply (*drule cert_authentic_agent*, *assumption*)

apply (*erule disjE*)

apply (*blast dest*: *Says_CA_cert1 Says_CA_cert2*)+

done

theorem *Spy_not_see_NA*:

⟦*Says A B* (*Crypt K* {|*Nonce NA*, *Agent A*|}) ∈ *set evs*;

A ∉ *bad*; *B* ∉ *bad*; *evs* ∈ *ns_public*⟧

⟹ *Nonce NA* ∉ *analz* (*knows Spy evs*) ∧ *K* = *pubEK B*

apply (*erule rev_mp*)

apply (*erule ns_public.induct*, *simp_all*, *spy_analz*)

apply (*case_tac Says A B* (*Crypt K* {|*Nonce NA*, *Agent A*|}) ∈ *set evs1*)

apply *blast*

apply *clarsimp*

apply (*case_tac Ba*=*B*)

apply *clarsimp*

apply (*case_tac Ka* = *pubK B*)

apply *clarsimp*

apply *blast*

apply *clarsimp*

apply (*drule cert_authentic_agent*, *assumption*)

apply (*blast dest*: *Says_CA_cert1 Says_CA_cert2*)

apply *blast*

apply *clarsimp*

apply (*case_tac NA* = *NB*)

apply *clarsimp*

apply *blast*

apply *clarsimp*

apply (*case_tac Aa* = *A*)

apply *clarsimp*

apply (*case_tac K* = *pubK A*)

apply *clarsimp*

apply (*drule cert_authentic_agent*, *assumption*)

apply (*blast dest*: *Says_CA_cert1 Says_CA_cert2*)

apply (*blast dest*: *unique_NA*)

apply (*blast dest*: *no_nonce_NS1_NS2*)

done

lemma *A_trusts_NS2_lemma* [*rule_format*]:

⟦*A* ∉ *bad*; *B* ∉ *bad*; *evs* ∈ *ns_public*⟧

⟹ *Crypt* (*pubEK A*) {|*Nonce NA*, *Nonce NB*, *Agent B*|} ∈ *parts* (*knows Spy evs*) ⟶

Says A B (*Crypt*(*pubEK B*) {|*Nonce NA*, *Agent A*|}) ∈ *set evs* ⟶

Says B A (*Crypt*(*pubEK A*) {|*Nonce NA*, *Nonce NB*, *Agent B*|}) ∈ *set evs*

apply (*erule ns_public.induct*, *simp_all*)

apply *clarsimp*

apply (*drule Fake_parts_insert_in_Un*, *assumption*)

apply (*blast dest*: *Spy_not_see_NA*)

apply *blast*

apply (*blast dest*: *Spy_not_see_NA unique_NA*)

done

theorem *A_trusts_NS2*:

⟦*Says A B* (*Crypt*(*pubEK B*) {|*Nonce NA*, *Agent A*|}) ∈ *set evs*;

Says B′ A (*Crypt*(*pubEK A*) {|*Nonce NA*, *Nonce NB*, *Agent B*|}) ∈ *set evs*;

A ∉ *bad*; *B* ∉ *bad*; *evs* ∈ *ns_public*⟧

⟹ *Says B A* (*Crypt*(*pubEK A*) {|*Nonce NA*, *Nonce NB*, *Agent B*|}) ∈ *set evs*

by (*blast intro*: *A_trusts_NS2_lemma*)

lemma *B_trusts_NS1* [*rule_format*]:

evs ∈ *ns_public*

⟹ *Crypt* (*pubEK B*) {|*Nonce NA*, *Agent A*|} ∈ *parts* (*knows Spy evs*) ⟶

Nonce NA ∉ *analz* (*knows Spy evs*) ⟶

Says A B (*Crypt* (*pubEK B*) {|*Nonce NA*, *Agent A*|}) ∈ *set evs*

apply (*erule ns_public.induct*, *simp_all*)

apply (*blast intro*!: *analz_insertI*)

apply *auto*

apply (*drule cert_authentic_agent*) **apply** *assumption*

apply *safe*

apply (*blast dest*: *Says_CA_cert2*)

apply (*blast dest*: *Says_CA_cert1*)

done

lemma *unique_NB* [*dest*]:

⟦*Crypt*(*pubEK A*) {|*Nonce NA*, *Nonce NB*, *Agent B*|} ∈ *parts*(*knows Spy evs*);

Crypt(*pubEK A'*) {|*Nonce NA'*, *Nonce NB*, *Agent B'*|} ∈ *parts*(*knows Spy evs*);

Nonce NB ∉ *analz* (*knows Spy evs*); *evs* ∈ *ns_public*⟧

⟹ *A*=*A'* ∧ *NA*=*NA'* ∧ *B*=*B'*

apply (*erule rev_mp*, *erule rev_mp*, *erule rev_mp*)

apply (*erule ns_public.induct*, *simp_all*)

apply (*blast intro*: *analz_insertI*)+

done

theorem *Spy_not_see_NB* [*dest*]:

⟦*Says B A* (*Crypt K* {|*Nonce NA*, *Nonce NB*, *Agent B*|}) ∈ *set evs*;

A ∉ *bad*; *B* ∉ *bad*; *evs* ∈ *ns_public*⟧

⟹ *Nonce NB* ∉ *analz* (*knows Spy evs*) ∧ *K* = *pubEK A*

apply (*erule rev_mp*)

apply (*erule ns_public.induct*, *simp_all*, *spy_analz*)

apply *blast*

apply (*case_tac Says B A* (*Crypt K* {|*Nonce NA*, *Nonce NB*, *Agent B*|}) ∈ *set evs2*)

apply *clarsimp*

apply (*case_tac Key* (*invKey K*) ∈ *analz* (*knows Spy evs2*))

apply *clarsimp*

apply (*blast dest*: *no_nonce_NS1_NS2*)

apply *clarsimp*

apply *clarsimp*

apply (*case_tac Key* (*invKey Ka*) ∈ *analz* (*knows Spy evs2*))

apply *clarsimp*

apply (*case_tac K* = *pubK A*)

apply *clarsimp*

apply (*blast dest*: *Says_CA_cert1 Says_CA_cert2 cert_authentic_agent*)

apply *clarsimp*

apply (*case_tac K* = *pubK A*)

apply *clarsimp*

apply *blast*

apply (*blast dest*: *Says_CA_cert1 Says_CA_cert2 cert_authentic_agent*)

apply (*blast dest*: *Says_imp_analz_Spy Spy_not_see_NA*)+

done

lemma *B_trusts_NS3_lemma* [*rule_format*]:

⟦*A* ∉ *bad*; *B* ∉ *bad*; *evs* ∈ *ns_public*⟧ ⟹

Crypt (*pubEK B*) (*Nonce NB*) ∈ *parts* (*knows Spy evs*) ⟶

Says B A (*Crypt* (*pubEK A*) {|*Nonce NA*, *Nonce NB*, *Agent B*|}) ∈ *set evs* ⟶

Says A B (*Crypt* (*pubEK B*) (*Nonce NB*)) ∈ *set evs*

by (*erule ns_public.induct*, *auto*)

theorem *B_trusts_NS3*:

⟦*Says B A* (*Crypt* (*pubEK A*) {|*Nonce NA*, *Nonce NB*, *Agent B*|}) ∈ *set evs*;

Says A' B (*Crypt* (*pubEK B*) (*Nonce NB*)) ∈ *set evs*;

A ∉ *bad*; *B* ∉ *bad*; *evs* ∈ *ns_public*⟧

⟹ *Says A B* (*Crypt* (*pubEK B*) (*Nonce NB*)) ∈ *set evs*

by (*blast intro*: *B_trusts_NS3_lemma*)

theorem *B_trusts_protocol*:

⟦*A* ∉ *bad*; *B* ∉ *bad*; *evs* ∈ *ns_public*⟧ ⟹

Crypt (*pubEK B*) (*Nonce NB*) ∈ *parts* (*knows Spy evs*) ⟶

Says B A (*Crypt* (*pubEK A*) {|*Nonce NA*, *Nonce NB*, *Agent B*|}) ∈ *set evs* ⟶

Says A B (*Crypt* (*pubEK B*) {|*Nonce NA*, *Agent A*|}) ∈ *set evs*

by (*erule ns_public.induct*, *auto*)

end

A.2 Proofs for the ISO/IEC 9798-3 Protocol with AIBS

A.2.1 Public_IBS.thy

theory *Public_IBS*

imports *Event*

abbreviation

TA :: *agent* **where**

TA == *Friend 0*

overloading

initState ≡ *initState*

begin

primrec *initState* **where**

initState_Server:

initState Server =

{*Key* (*priEK Server*), *Key* (*priSK Server*)} ∪

(*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*) ∪ (*Key* ‘ *range shrK*)

| *initState_Friend*:

initState (*Friend i*) =

(*if* (*i* = *0*)

then {*Key* (*priEK TA*), *Key* (*shrK TA*)} ∪

(*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*) ∪ (*Key* ‘ *range priSK*)

else {*Key* (*priEK*(*Friend i*)), *Key* (*priSK*(*Friend i*)), *Key* (*shrK*(*Friend i*))} ∪

(*Key* ‘ *range pubEK*) ∪ (*Key* ‘ *range pubSK*))

| *initState_Spy*:

initState Spy =

(*if* (*TA* ∈ *bad*)

then $(Key\ `\ invKey\ `\ pubEK\ `\ bad) \cup$

$(Key\ `\ range\ pubEK) \cup (Key\ `\ range\ pubSK) \cup (Key\ `\ shrK\ `\ bad) \cup$

$(Key\ `\ range\ \ priSK)$

else $(Key\ `\ invKey\ `\ pubEK\ `\ bad) \cup$

$(Key\ `\ invKey\ `\ pubSK\ `\ bad)\ \cup\ (Key\ `\ shrK\ `\ bad) \cup$

$(Key\ `\ range\ pubEK) \cup (Key\ `\ range\ pubSK))$

A.2.2 ISO_IBS.thy

theory *ISO_IBS* **imports** *Public_IBS* **begin**

abbreviation

TTP :: *agent* **where** *TTP* == *Server*

abbreviation *Text1* :: *nat* **where** *Text1* == *1*

abbreviation *Text2* :: *nat* **where** *Text2* == *2*

abbreviation *Text3* :: *nat* **where** *Text3* == *3*

abbreviation *Text4* :: *nat* **where** *Text4* == *4*

abbreviation *Text5* :: *nat* **where** *Text5* == *5*

abbreviation *Text6* :: *nat* **where** *Text6* == *6*

abbreviation *Text7* :: *nat* **where** *Text7* == *7*

abbreviation *Text8* :: *nat* **where** *Text8* == *8*

abbreviation *Text9* :: *nat* **where** *Text9* == *9*

| *ISO1*: $[\![evs1 \in iso;\ Nonce\ Na \notin used\ evs1]\!]$

$\Longrightarrow Says\ A\ B\ \{\!|Agent\ A, Nonce\ Na, Number\ Text1|\!\}\ \#\ evs1 \in iso$

| *ISO2*: $[\![evs2 \in iso;\ Nonce\ Nb \notin used\ evs2; A \neq TA; B \neq TA;$

$Gets\ B\ \ \{\!|Agent\ A, Nonce\ Na, Number\ Text1|\!\} \in set\ evs2]\!]$

$\Longrightarrow Says\ B\ A\ \ \{\!|Agent\ B, Nonce\ Na, Nonce\ Nb, Number\ Text3,$

$Crypt\ (priSK\ B)\ \{\!|Agent\ B, Nonce\ Na, Nonce\ Nb, Agent\ A,$

$Number\ Text2|\!\}\!|\!\}\ \ \#\ evs2 \in iso$

| *ISO3*: $[\![evs3 \in iso; Nonce\ Na' \notin used\ evs3; A \neq TA; B \neq TA;$

$Says\ A\ B\ \{\!|Agent\ A, Nonce\ Na, Number\ Text1|\!\} \in set\ evs3;$

$$\begin{aligned}
&\mathit{Gets}\ A\ \{\!|\mathit{Agent}\ B, \mathit{Nonce}\ \mathit{Na}, \mathit{Nonce}\ \mathit{Nb}, \mathit{Number}\ \mathit{Text3},\\
&\qquad \mathit{Crypt}\ (\mathit{priSK}\ B)\ \{\!|\mathit{Agent}\ B, \mathit{Nonce}\ \mathit{Na}, \mathit{Nonce}\ \mathit{Nb},\ \mathit{Agent}\ A,\\
&\qquad\qquad \mathit{Number}\ \mathit{Text2}|\!\}\ |\!\} \in \mathit{set}\ \mathit{evs3}[\!]\\
&\Longrightarrow \mathit{Says}\ A\ \mathit{TTP}\ \{\!|\mathit{Nonce}\ \mathit{Na}', \mathit{Nonce}\ \mathit{Nb}, \mathit{Agent}\ A, \mathit{Agent}\ B, \mathit{Number}\ \mathit{Text4}|\!\}\\
&\quad \#\ \mathit{evs3} \in \mathit{iso}
\end{aligned}$$

$$\begin{aligned}
|\ \mathit{ISO4}:\ &[\![\mathit{evs4} \in \mathit{iso}; A \neq B; A \neq \mathit{TA}; B \neq \mathit{TA};\\
&\mathit{Gets}\ \mathit{TTP}\ \{\!|\mathit{Nonce}\ \mathit{Na}', \mathit{Nonce}\ \mathit{Nb}, \mathit{Agent}\ A, \mathit{Agent}\ B, \mathit{Number}\ \mathit{Text4}|\!\} \in \mathit{set}\ \mathit{evs4}[\!]\\
&\Longrightarrow \mathit{Says}\ \mathit{TTP}\ A\ \{\!|\mathit{Agent}\ A, \mathit{Key}\ (\mathit{pubSK}\ A), \mathit{Agent}\ B, \mathit{Key}\ (\mathit{pubSK}\ B),\\
&\qquad \mathit{Crypt}\ (\mathit{priSK}\ \mathit{TTP})\ \{\!|\mathit{Nonce}\ \mathit{Na}', \mathit{Agent}\ B,\\
&\qquad\qquad \mathit{Key}\ (\mathit{pubSK}\ B), \mathit{Number}\ \mathit{Text6}|\!\},\\
&\qquad \mathit{Crypt}\ (\mathit{priSK}\ \mathit{TTP})\ \{\!|\mathit{Nonce}\ \mathit{Nb}, \mathit{Agent}\ A,\\
&\qquad\qquad \mathit{Key}\ (\mathit{pubSK}\ A), \mathit{Number}\ \mathit{Text5}|\!\},\\
&\qquad \mathit{Number}\ \mathit{Text7}|\!\}\ \#\ \mathit{evs4} \in \mathit{iso}
\end{aligned}$$

$$\begin{aligned}
|\ \mathit{ISO5}:\ &[\![\mathit{evs5} \in \mathit{iso}; A \neq \mathit{TA}; B \neq \mathit{TA};\\
&\mathit{Says}\ A\ B\ \{\!|\mathit{Agent}\ A, \mathit{Nonce}\ \mathit{Na}, \mathit{Number}\ \mathit{Text1}|\!\} \in \mathit{set}\ \mathit{evs5};\\
&\mathit{Gets}\ A\ \ \{\!|\mathit{Agent}\ A, \mathit{Key}\ (\mathit{pubSK}\ A), \mathit{Agent}\ B, \mathit{Key}\ (\mathit{pubSK}\ B),\\
&\qquad \mathit{Crypt}\ (\mathit{priSK}\ \mathit{TTP})\ \{\!|\mathit{Nonce}\ \mathit{Na}', \mathit{Agent}\ B,\\
&\qquad\qquad \mathit{Key}\ (\mathit{pubSK}\ B), \mathit{Number}\ \mathit{Text6}|\!\},\\
&\qquad \mathit{Crypt}\ (\mathit{priSK}\ \mathit{TTP})\ \{\!|\mathit{Nonce}\ \mathit{Nb}, \mathit{Agent}\ A,\\
&\qquad\qquad \mathit{Key}\ (\mathit{pubSK}\ A), \mathit{Number}\ \mathit{Text5}|\!\},\\
&\qquad \mathit{Number}\ \mathit{Text7}|\!\} \in \mathit{set}\ \mathit{evs5}[\!]\\
&\Longrightarrow \mathit{Says}\ A\ B\ \{\!|\mathit{Number}\ \mathit{Text9}, \mathit{Agent}\ A, \mathit{Key}\ (\mathit{pubSK}\ A),\\
&\qquad \mathit{Crypt}\ (\mathit{priSK}\ \mathit{TTP})\ \{\!|\mathit{Nonce}\ \mathit{Nb}, \mathit{Agent}\ A,\\
&\qquad\qquad \mathit{Key}\ (\mathit{pubSK}\ A), \mathit{Number}\ \mathit{Text5}|\!\},\\
&\qquad \mathit{Crypt}\ (\mathit{priSK}\ A)\ \{\!|\mathit{Nonce}\ \mathit{Nb}, \mathit{Nonce}\ \mathit{Na},\\
&\qquad\qquad \mathit{Agent}\ B, \mathit{Agent}\ A, \mathit{Number}\ \mathit{Text8}|\!\}\ |\!\}\ \#\ \mathit{evs5} \in \mathit{iso}
\end{aligned}$$

declare *knows_Spy_partsEs* [*elim*]

declare *knows_Spy_partsEs* [*elim*]

declare *analz_into_parts* [*dest*]

declare *Fake_parts_insert_in_Un* [*dest*]

declare *image_eq_UN* [*simp*]

lemma $A \neq TA \wedge B \neq TA \wedge A \neq B \Longrightarrow \exists\, Nb\ Na.\ \exists\, evs \in iso.$

$Says\ A\ B\ \{|Number\ Text9, Agent\ A, Key\ (pubSK\ A),$

$Crypt\ (priSK\ TTP)\ \{|Nonce\ Nb,\ Agent\ A, Key\ (pubSK\ A), Number\ Text5|\},$

$Crypt\ (priSK\ A)\ \{|Nonce\ Nb, Nonce\ Na, Agent\ B, Agent\ A, Number\ Text8|\}|\}$

$\in set\ evs$

apply $(intro\ exI\ bexI)$

apply $(rule\_tac\ [2]\ iso.Nil\ [THEN\ iso.ISO1, THEN\ iso.Reception, THEN\ iso.ISO2,$

$THEN\ iso.Reception, THEN\ iso.ISO3, THEN\ iso.Reception,$

$THEN\ iso.ISO4, THEN\ iso.Reception, THEN\ iso.ISO5], possibility)$

apply *auto*

done

lemma *Spy_see_priSK* [*simp*]:

$evs \in iso \Longrightarrow (Key\ (priSK\ A) \in parts\ (spies\ evs)) = (A \in bad \vee TA \in bad)$

apply $(erule\ iso.induct, auto)$

by $(metis\ parts\_insertI)$

lemma *Spy_analz_priSK* [*simp*]:

$evs \in iso \Longrightarrow (Key\ (priSK\ A) \in analz\ (spies\ evs)) = (A \in bad \vee TA \in bad)$

by $(metis\ Spy\_see\_priSK\ Spy\_spies\_bad\_privateKey\ analz.Inj$

$analz\_into\_parts\ initState\_subset\_knows\ priSK\_initState3\ subsetD)$

lemma *prikeys*: $evs \in iso \Longrightarrow Key\ (priSK\ A) \in parts(knows\ TA\ evs)$

apply $(metis\ initState\_subset\_knows\ parts.Inj\ priSK\_initState\ subsetD)$

done

lemma *priSK_knowledge*:

$[\![Key\ (priSK\ A) \in initState\ D; A \notin bad; TA \notin bad]\!] \Longrightarrow D = TA \vee D = A$

apply $(induct\ D)$

apply *auto*

apply $(case\_tac\ nat = 0)$

apply *auto*

done

lemma *sig_B_origin*:

⟦*Crypt* (*priSK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|} ∈ *parts* (*spies evs*);

B ∉ *bad*; *TA* ∉ *bad*; *evs* ∈ *iso*⟧

⟹ *Says B A* {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Number Text3*,

Crypt (*priSK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*,

Agent A, *Number Text2*|}|} ∈ *set evs*

apply (*erule rev_mp*)

apply (*erule iso.induct*)

apply *simp_all*

apply *spy_analz*

done

lemma *sig_A_origin*:

⟦*Crypt* (*priSK A*) {|*Nonce Nb*, *Nonce Na*, *Agent B*, *Agent A*, *Number Text8*|} ∈ *parts* (*spies evs*);

A ∉ *bad*; *TA* ∉ *bad*; *evs* ∈ *iso*⟧

⟹ *Says A B* {|*Number Text9*, *Agent A*, *Key* (*pubSK A*),

Crypt (*priSK TTP*) {|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|},

Crypt (*priSK A*) {|*Nonce Nb*, *Nonce Na*, *Agent B*,

Agent A, *Number Text8*|}|} ∈ *set evs*

apply (*erule rev_mp*)

apply (*erule iso.induct*)

apply *simp_all*

apply *spy_analz*

done

lemma *sig_TTP_origin*:

⟦*Crypt* (*priSK TTP*) {|*Nonce Na'*, *Agent B*, *Key* (*pubSK B*), *Number Text6*|} ∈ *parts* (*spies evs*);

TA ∉ *bad*; *evs* ∈ *iso*⟧

⟹ ∃ *A Nb*. *Says TTP A* {|*Agent A*, *Key* (*pubSK A*), *Agent B*, *Key* (*pubSK B*),

Crypt (*priSK TTP*) {|*Nonce Na'*, *Agent B*,

Key (*pubSK B*), *Number Text6*⦄,
Crypt (*priSK TTP*) ⦃*Nonce Nb*, *Agent A*,
Key (*pubSK A*), *Number Text5*⦄,
Number Text7⦄ ∈ *set evs*

apply (*erule rev_mp*)
apply (*erule iso.induct*)
apply *simp_all*
apply *spy_analz*
apply *blast*
done

lemma *ISO5_imp_ISO1*:
⟦*Says A B* ⦃*Number Text9*, *Agent A*, *Key* (*pubSK A*),
Crypt (*priSK TTP*) ⦃*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*⦄,
Crypt (*priSK A*) ⦃*Nonce Nb*, *Nonce Na*, *Agent B*,
Agent A, *Number Text8*⦄⦄ ∈ *set evs*;
A ∉ *bad*; *evs* ∈ *iso*⟧
⟹ *Says A B* ⦃*Agent A*, *Nonce Na*, *Number Text1*⦄ ∈ *set evs*
apply (*erule rev_mp*)
apply (*erule iso.induct*)
apply *simp_all*
apply *spy_analz*
done
lemma *Fake_insert*:
⟦*Crypt* (*priSK A*) *Y* ∈ *parts* (*insert X* (*knows Spy evs*));
X ∈ *synth* (*analz* (*knows Spy evs*));
A ∉ *bad*; *TA* ∉ *bad*; *evs* ∈ *iso*⟧
⟹ *Crypt* (*priSK A*) *Y* ∈ *parts* (*spies evs*)
apply *spy_analz*
done

theorem *A_auth_B*:

⟦{|*Agent B*, *Nonce Na*, *Nonce Nb*, *Number Text3*,
Crypt (*priSK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*,
Agent A, *Number Text2*|}|} ∈ *parts* (*spies evs*);
Says A B {|*Agent A*, *Nonce Na*, *Number Text1*|} ∈ *set evs*;
B ∉ *bad*; *TA* ∉ *bad*; *evs* ∈ *iso*⟧
⟹ *Says B A* {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Number Text3*,
Crypt (*priSK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*,
Agent A, *Number Text2*|}|} ∈ *set evs*

apply (*blast dest*: *sig_B_origin*)

done

theorem *B_auth_A*:

⟦{|*Number Text9*, *Agent A*, *Key* (*pubSK A*),
Crypt (*priSK TTP*) {|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|},
Crypt (*priSK A*) {|*Nonce Nb*, *Nonce Na*, *Agent B*,
Agent A, *Number Text8*|}|} ∈ *parts* (*spies evs*);
Says B A {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Number Text3*,
Crypt (*priSK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*,
Agent A, *Number Text2*|}|} ∈ *set evs*;
A ∉ *bad*; *TA* ∉ *bad*; *evs* ∈ *iso*⟧
⟹ *Says A B* {|*Agent A*, *Nonce Na*, *Number Text1*|} ∈ *set evs*

apply (*erule rev_mp*, *erule rev_mp*)

apply (*erule iso.induct*)

apply *simp_all*

apply (*case_tac B=Spy*)

apply (*case_tac A=B*, *clarsimp*)

apply (*case_tac* {|*Agent Spy*, *Nonce Na*, *Nonce Nb*, *Number Text3*,
Crypt (*priSK Spy*) {|*Agent Spy*, *Nonce Na*, *Nonce Nb*,
Agent A, *Number Text2*|}|} = *X*)

apply *clarsimp*

apply (*metis sig_A_origin ISO5_imp_ISO1 One_nat_def*)

apply (*blast dest*: *Fake_insert sig_A_origin*)+

done

definition *candidates* :: *agent* ⇒ *event list* ⇒ *agent set* **where**

candidates A evs ≡

{*C*. *C* ≠ *Spy* ∧

(∃ *Y Z*. *Crypt* (*priSK A*) *Y* ∈ *parts* (*knows Spy evs*) ∧

(*A* = *C* ∨ *Key* (*priSK A*) ∈ *initState C*) ∧

(*Crypt* (*priSK C*) *Z* ∈ *parts* {*Y*} ∨ *Crypt* (*priEK C*) *Z* ∈ *parts* {*Y*}))}

lemma *signature_form*:

⟦*Crypt* (*priSK A*) *Y* ∈ *parts* (*spies evs*);

Crypt (*priSK TA*) *Z* ∈ *parts* {*Y*} ∨ *Crypt* (*priEK TA*) *Z* ∈ *parts* {*Y*} ∨

Crypt (*priSK A*) *Z* ∈ *parts* {*Y*} ∨ *Crypt* (*priEK A*) *Z* ∈ *parts* {*Y*};

A ∉ *bad*; *TA* ∉ *bad*; *evs* ∈ *iso*⟧

⟹ *False*

apply (*erule rev_mp*)

apply (*erule iso.induct*)

apply *simp_all*

apply *spy_analz*

apply *force*+

done

theorem *candidates_IBS_none*:

⟦*D* ∈ *candidates A evs*; *TA* ∉ *bad*; *A* ∉ *bad*; *evs* ∈ *iso*⟧ ⟹ *False*

apply (*unfold candidates_def*)

apply (*induct D*)

apply (*force intro*: *Server_not_bad dest*!: *signature_form*)

apply (*case_tac nat* = *0*)

apply (*force dest*!: *signature_form*)

apply (*force dest*!: *signature_form*)

apply *simp*

done

end

A.2.3 Message_AIBS.thy

Message_AIBS

imports *Main*

abbreviation

TA :: *agent* **where**

TA == *Friend 0*

datatype *pack* = *Pack key key*

datatype

msg = *Agent agent* — Agent names

| *Number nat* — Ordinary integers, timestamps, ...

| *Nonce nat* — Unguessable nonces

| *Key key* — Crypto keys

| *Hash msg* — Hashing

| *MPair msg msg* — Compound messages

| *Crypt key msg* — Encryption, public- or shared-key

| *Pkg pack* — Key package (second key is hidden SK)

abbreviation

KP :: [*agent*, *agent*] => *pack* **where**

KP A B == *Pack* (*pubEK A*) (*priSK B*)

inductive_set

parts :: *msg set* => *msg set*

for *H* :: *msg set*

where

Inj [*intro*]: $X \in H \Longrightarrow X \in parts\ H$

| *Fst*: $\{|X,Y|\} \in parts\ H \Longrightarrow X \in parts\ H$

| *Snd*: $\{|X,Y|\} \in parts\ H \Longrightarrow Y \in parts\ H$

| *Body*: $Crypt\ K\ X \in parts\ H \Longrightarrow X \in parts\ H$

lemma *Protect_image_eq* [*simp*]: (*Pkg x* ∈ *Pkg*‘*A*) = (*x*∈*A*)

by *auto*

lemma *Protect_Key_image_eq* [*simp*]: (*Pkg R* ∉ *Key* ‘ *A*)

by *auto*

lemma *keysFor_insert_Pkg* [*simp*]:

keysFor (*insert* (*Pkg R*) *H*) = *keysFor H*

by (*unfold keysFor_def*, *blast*)

lemma *parts_insert_Protect* [*simp*]:

parts (*insert* (*Pkg S*) *H*)

= *insert* (*Pkg S*) (*parts H*)

inductive_set

analz :: *msg set* => *msg set*

for *H* :: *msg set*

where

Inj [*intro*,*simp*] : *X* ∈ *H* ==> *X* ∈ *analz H*

| *Fst*: {|*X*,*Y*|} ∈ *analz H* ==> *X* ∈ *analz H*

| *Snd*: {|*X*,*Y*|} ∈ *analz H* ==> *Y* ∈ *analz H*

| *Decrypt* [*dest*]:

[|*Crypt K X* ∈ *analz H*; *Key*(*invKey K*): *analz H*|] ==> *X* ∈ *analz H*

lemma *analz_insert_Protect* [*simp*]:

analz (*insert* (*Pkg* (*KP R T*)) *H*) =

insert (*Pkg* (*KP R T*)) (*analz H*)

apply (*rule analz_insert_eq_I*)

by (*erule analz.induct*, *auto*)

inductive_set

synth :: *msg set* => *msg set*

for *H* :: *msg set*

where

Inj [*intro*]: $X \in H ==> X \in synth\ H$

| *Agent* [*intro*]: $Agent\ agt \in synth\ H$

| *Number* [*intro*]: $Number\ n \in synth\ H$

| *Hash* [*intro*]: $X \in synth\ H ==> Hash\ X \in synth\ H$

| *MPair* [*intro*]: $[|X \in synth\ H;\ Y \in synth\ H|] ==> \{|X,Y|\} \in synth\ H$

| *Crypt* [*intro*]: $[|X \in synth\ H;\ Key(K) \in H|] ==> Crypt\ K\ X \in synth\ H$

inductive_simps *synth_simps* [*iff*]:

$Nonce\ n \in synth\ H$

$Key\ K \in synth\ H$

$Hash\ X \in synth\ H$

$\{|X,Y|\} \in synth\ H$

$Crypt\ K\ X \in synth\ H$

$Pkg\ R \in synth\ H$

A.2.4 Event_AIBS.thy

This theory is the same as the traditional *Event.thy* — the only difference is that it importes *Message_AIBS.thy* instead of *Message.thy*.

A.2.5 Public_AIBS.thy

theory *Public_AIBS*

imports *Event_AIBS*

overloading

$initState \equiv initState$

begin

primrec *initState* **where**

initState_Server:

$initState\ Server =$

$\{Key\ (priEK\ Server)\} \cup$

$(Key\ `\ range\ pubEK) \cup (Key\ `\ range\ pubSK)$

| *initState_Friend*:

initState $(Friend\ i) =$

$(if\ (i = 0)$

then $\{Key\ (priEK\ TA)\} \cup$

$(Key\ `\ range\ pubEK) \cup (Key\ `\ range\ pubSK) \cup (Key\ `\ range\ \ priSK)$

else $\{Key\ (priEK(Friend\ i))\} \cup$

$(Key\ `\ range\ pubEK) \cup (Key\ `\ range\ pubSK))$

| *initState_Spy*:

initState Spy $=$

$(if\ (TA \in bad)$

then $(Key\ `\ invKey\ `\ pubEK\ `\ bad) \cup$

$(Key\ `\ range\ pubEK) \cup (Key\ `\ range\ pubSK) \cup$

$(Key\ `\ range\ \ priSK)$

else $(Key\ `\ invKey\ `\ pubEK\ `\ bad) \cup$

$(Key\ `\ range\ pubEK) \cup (Key\ `\ range\ pubSK))$

end

lemma *TA_knows_nil* : *initState* $(TA) = \{Key\ (priEK\ TA)\} \cup$

$(Key\ `\ range\ pubEK) \cup (Key\ `\ range\ pubSK) \cup (Key\ `\ range\ \ priSK)$

by *auto*

lemma *priK_in_initState* [*iff*]: $Key\ (privateKey\ Encryption\ A) \in initState\ A$

by $(cases\ A, auto)$

lemma *priSK_initState* [*iff*]: $Key\ (privateKey\ Signature\ A) \in initState\ TA$

by $(cases\ A, auto)$

lemma *priSK_initState2* [*iff*]: $A \neq TA \wedge A \neq Spy \Longrightarrow Key\ (privateKey\ Signature\ A) \notin initState\ A$

by $(cases\ A, auto)$

lemma *priSK_initState3* [*iff*]: *TA* : *bad* $\Longrightarrow$ *Key* (*privateKey Signature A*) : *initState Spy*

by (*cases A*, *auto*)

lemma *publicKey_in_initState* [*iff*]: *Key* (*publicKey b A*) $\in$ *initState B*

apply (*cases b*, *auto*)

apply (*cases B*, *auto*)+

done

lemma *Spy_spies_bad_privateKey* [*intro*!]:

A $\in$ *bad* ==> *Key* (*privateKey Encryption A*) $\in$ *spies evs*

apply (*induct_tac evs*)

apply (*auto simp add*: *imageI knows_Cons split add*: *event.split*)

done

lemma *Spy_spies_bad_privateKeyTA* [*intro*!]:

TA $\in$ *bad* ==> *Key* (*privateKey Signature A*) $\in$ *spies evs*

apply (*induct_tac evs*)

apply (*auto simp add*: *imageI knows_Cons split add*: *event.split*)

done

lemma *privateKey_into_usedTA* [*iff*]: *TA* : *bad* $\Longrightarrow$ *Key* (*privateKey Signature A*) $\in$ *used evs*

apply (*rule initState_into_used*)

apply *auto*

done

A.2.6 ISO_AIBS.thy

theory *ISO_AIBS* **imports** *Public_AIBS* **begin**

abbreviation

TTP :: *agent* **where** *TTP* == *Server*

abbreviation *Text1* :: *nat* **where** *Text1* == *1*

abbreviation *Text2* :: *nat* **where** *Text2* == *2*

abbreviation *Text3* :: *nat* **where** *Text3* == *3*

abbreviation *Text4* :: *nat* **where** *Text4* == *4*

abbreviation *Text5* :: *nat* **where** *Text5* == *5*

abbreviation *Text6* :: *nat* **where** *Text6* == *6*

abbreviation *Text7* :: *nat* **where** *Text7* == *7*

abbreviation *Text8* :: *nat* **where** *Text8* == *8*

abbreviation *Text9* :: *nat* **where** *Text9* == *9*

inductive_set *iso* :: *event list set*

where

| *KeyPack*: ⟦*evsk* ∈ *iso*; *Key*(*priSK TA*) ∈ *analz* (*spies evsk*)⟧

⟹ *Notes Spy* (*Pkg* (*KP A B*)) # *evsk* ∈ *iso*

| *SigGen*: ⟦*evss* ∈ *iso*; *X* ∈ *synth*(*analz* (*spies evss*)); *Key* (*priEK A*) ∈ *analz* (*spies evss*);

Pkg (*KP A B*) ∈ *analz* (*spies evss*)⟧

⟹ *Notes Spy* (*Crypt* (*priSK B*) {|*Crypt* (*priEK A*) *X*, *X*|}) # *evss* ∈ *iso*

| *ISO1*: ⟦*evs1* ∈ *iso*; *Nonce Na* ∉ *used evs1*; *A* ≠ *TA*; *B* ≠ *TA*⟧

⟹ *Says A B* {|*Agent A*, *Nonce Na*, *Number Text1*|}

evs1 ∈ *iso*

| *ISO2*: ⟦*evs2* ∈ *iso*; *Nonce Nb* ∉ *used evs2*; *A* ≠ *TTP*; *B* ≠ *TTP*; *A* ≠ *TA*; *B* ≠ *TA*;

Gets B {|*Agent A*, *Nonce Na*, *Number Text1*|} ∈ *set evs2*⟧

⟹ *Says B A* {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Number Text3*,

Crypt (*priSK B*) {|*Crypt* (*priEK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|},

{|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|}|}

evs2 ∈ *iso*

| *ISO3*: ⟦*evs3* ∈ *iso*; *Nonce Na'* ∉ *used evs3*; *A* ≠ *TTP*; *B* ≠ *TTP*; *A* ≠ *TA*; *B* ≠ *TA*;

Says A B {|*Agent A*, *Nonce Na*, *Number Text1*|} ∈ *set evs3*;

Gets A {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Number Text3*,

Crypt (*priSK B*) {|*Crypt* (*priEK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|},

{|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|}| }

∈ *set evs3*⟧

⟹ *Says A TTP* {|*Nonce Na'*, *Nonce Nb*, *Agent A*,

Agent B, *Number Text4*|} # *evs3* ∈ *iso*

| *ISO4*: ⟦*evs4* ∈ *iso*; *A* ≠ *B*; *A* ≠ *TA*; *B* ≠ *TA*;

Gets TTP {|*Nonce Na'*, *Nonce Nb*, *Agent A*, *Agent B*, *Number Text4*|} ∈ *set evs4*⟧

⟹ *Says TTP A* {|*Agent A*, *Key* (*pubSK A*), *Agent B*, *Key* (*pubSK B*),

Crypt (*priSK TTP*) {|*Crypt* (*priEK TTP*) {|*Nonce Na'*, *Agent B*, *Key* (*pubSK B*), *Number Text6*|},

{|*Nonce Na'*, *Agent B*, *Key* (*pubSK B*), *Number Text6*|}|},

Crypt (*priSK TTP*) {|*Crypt* (*priEK TTP*) {|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|},

{|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|}|},

Number Text7|} # *evs4* ∈ *iso*

| *ISO5*: ⟦*evs5* ∈ *iso*; *A* ≠ *TTP*; *B* ≠ *TTP*; *A* ≠ *TA*; *B* ≠ *TA*;

Says A B {|*Agent A*, *Nonce Na*, *Number Text1*|} ∈ *set evs5*;

Gets A {|*Agent A*, *Key* (*pubSK A*), *Agent B*, *Key* (*pubSK B*),

Crypt (*priSK TTP*) {|*Crypt* (*priEK TTP*) {|*Nonce Na'*, *Agent B*, *Key* (*pubSK B*), *Number Text6*|},

{|*Nonce Na'*, *Agent B*, *Key* (*pubSK B*), *Number Text6*|}|},

Crypt (*priSK TTP*) {|*Crypt* (*priEK TTP*) {|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|},

{|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|}|},

Number Text7|} ∈ *set evs5*⟧

⟹ *Says A B* {|*Number Text9*, *Agent A*, *Key* (*pubSK A*),

Crypt (*priSK TTP*) {|*Crypt* (*priEK TTP*) {|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|},

{|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|}|},

Crypt (*priSK A*) {|*Crypt* (*priEK A*) {|*Nonce Nb*, *Nonce Na*, *Agent B*, *Agent A*, *Number Text8*|},

{|*Nonce Nb*, *Nonce Na*, *Agent B*, *Agent A*, *Number Text8*|}|}

|}

evs5 ∈ *iso*

declare *knows_Spy_partsEs* [*elim*]

declare *knows_Spy_partsEs* [*elim*]

declare *analz_into_parts* [*dest*]

declare *Fake_parts_insert_in_Un* [*dest*]

declare *image_eq_UN* [*simp*]

lemma $A \neq TTP \wedge B \neq TTP \wedge A \neq B \wedge A \neq TA \wedge B \neq TA \Longrightarrow \exists Nb\ Na.\ \exists evs \in iso.$

$Says\ A\ B\ \{|Number\ Text9, Agent\ A, Key\ (pubSK\ A),$

$Crypt\ (priSK\ TTP)\ \{|Crypt\ (priEK\ TTP)\ \{|Nonce\ Nb, Agent\ A, Key\ (pubSK\ A),$

$Number\ Text5|\},$

$\{|Nonce\ Nb, Agent\ A, Key\ (pubSK\ A), Number\ Text5|\}|\},$

$Crypt\ (priSK\ A)\ \{|Crypt\ (priEK\ A)\ \{|Nonce\ Nb, Nonce\ Na, Agent\ B, Agent\ A,$

$Number\ Text8|\},$

$\{|Nonce\ Nb, Nonce\ Na, Agent\ B, Agent\ A, Number\ Text8|\}|\}$

$|\} \in set\ evs$

apply $(intro\ exI\ bexI)$

apply $(rule\_tac\ [2]\ iso.Nil\ [THEN\ iso.ISO1, THEN\ iso.Reception, THEN\ iso.ISO2,$

$THEN\ iso.Reception, THEN\ iso.ISO3, THEN\ iso.Reception,$

$THEN\ iso.ISO4, THEN\ iso.Reception, THEN\ iso.ISO5], possibility)$

apply *auto*

done

lemma *Spy_see_priSK* [*simp*]:

$evs \in iso \Longrightarrow (Key\ (priSK\ A) \in parts\ (spies\ evs)) = (TA \in bad)$

apply $(erule\ iso.induct)$

apply *auto*

done

lemma *Spy_see_priEK* [*simp*]:

$evs \in iso \Longrightarrow (Key\ (priEK\ A) \in parts\ (spies\ evs)) = (A \in bad)$

apply $(erule\ iso.induct)$

apply *auto*

done

lemma *Pack_conf*:

$[\![Pkg\ (Pack\ (pubK\ A)\ (priSK\ B)) \in parts\ (knows\ Spy\ evs); TA \notin bad; evs \in iso]\!] \Longrightarrow False$

apply $(erule\ rev\_mp)$

apply $(erule\ iso.induct)$

apply *simp_all*

apply *auto*

done

lemma *TA_never*:

⟦*Crypt* (*priSK TA*) *Z* ∈ *parts* (*spies evs*) ∨ *Crypt* (*priEK TA*) *Z* ∈ *parts* (*spies evs*);

TA ∉ *bad*; *A* ≠ *TA*; *evs* ∈ *iso*⟧

⟹ *False*

apply (*erule rev_mp*)

apply (*erule iso.induct*)

apply *simp_all*

defer

apply (*rule conjI*)

apply *clarsimp*

apply *force*

apply *force*

apply (*rule conjI*)

apply *clarsimp*

defer

apply (*rule conjI*)

apply *clarsimp*

apply (*smt Crypt_synth_eq Fake_parts_insert_in_Un Spy_see_priSK UnE analz_disj_parts*)

apply *clarsimp*

apply (*rule conjI*)

apply *clarsimp*

apply (*metis Spy_see_priEK not_parts_not_analz*)

apply *clarsimp*

apply (*smt Crypt_synth_eq Fake_parts_insert_in_Un Spy_see_priEK UnE analz_disj_parts*)

apply (*blast dest*: *Pack_conf*)

done

lemma *Spy_analz_priSK* [*simp*]:

evs ∈ *iso* ⟹ (*Key* (*priSK A*) ∈ *analz* (*spies evs*)) = (*TA* ∈ *bad*)

by *auto*

lemma *prikeys*: *evs* ∈ *iso* ⟹ *Key* (*priSK A*) ∈ *parts*(*knows TA evs*)

apply (*metis initState_subset_knows parts.Inj priSK_initState subsetD*)

done

lemma *priSK_knowledge*:

⟦*Key* (*priSK A*) ∈ *initState D*⟧ ⟹ *D* = *TA* ∨ *D* ∈ *bad*

apply (*induct D*)

apply *force*

apply (*case_tac nat* = *0*)

apply *clarsimp*

apply *clarsimp*

apply *force*

done

lemma *sig_B_origin*:

⟦*Crypt* (*priSK B*) {|*Crypt* (*priEK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|},

{|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|}| ∈ *parts* (*spies evs*); *B* ∉ *bad*;

TA ∉ *bad*; *evs* ∈ *iso*⟧

⟹ *Says B A* {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Number Text3*,

Crypt (*priSK B*) {|*Crypt* (*priEK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|},

{|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|}|} ∈ *set evs*

apply (*erule rev_mp*)

apply (*erule iso.induct*)

apply *simp_all*

apply *auto*[*1*]

apply *spy_analz*

done

lemma *sig_A_origin*:

⟦*Crypt* (*priSK A*) {|*Crypt* (*priEK A*) {|*Nonce Nb*, *Nonce Na*, *Agent B*, *Agent A*, *Number Text8*|},

$\{|Nonce\ Nb, Nonce\ Na, Agent\ B, Agent\ A, Number\ Text8|\} \in parts\ (spies\ evs);$

$A \notin bad; TA \notin bad; evs \in iso\rrbracket$

$\Longrightarrow Says\ A\ B\ \{|Number\ Text9, Agent\ A, Key\ (pubSK\ A),$

$Crypt\ (priSK\ TTP)\ \{|Crypt\ (priEK\ TTP)\ \{|Nonce\ Nb, Agent\ A, Key\ (pubSK\ A),$
$Number\ Text5|\},$

$\{|Nonce\ Nb, Agent\ A, Key\ (pubSK\ A), Number\ Text5|\}|\},$

$Crypt\ (priSK\ A)\ \{|Crypt\ (priEK\ A)\ \{|Nonce\ Nb, Nonce\ Na, Agent\ B, Agent\ A, Number$
$Text8|\},$

$\{|Nonce\ Nb, Nonce\ Na, Agent\ B, Agent\ A, Number\ Text8|\}|\} \in set\ evs$

apply $(erule\ rev\_mp)$

apply $(erule\ iso.induct)$

apply $simp\_all$

apply $auto[1]$

apply spy_analz

done

lemma sig_TTP_origin:

$\llbracket Crypt\ (priSK\ TTP)\ \{|Crypt\ (priEK\ TTP)\ \{|Nonce\ Na', Agent\ B, Key\ (pubSK\ B), Number$
$Text6|\},$

$\{|Nonce\ Na', Agent\ B, Key\ (pubSK\ B), Number\ Text6|\}|\} \in parts\ (spies\ evs);$

$TA \notin bad;$

$evs \in iso\rrbracket$

$\Longrightarrow \exists\ A\ Nb.\ Says\ TTP\ A\ \{|Agent\ A, Key\ (pubSK\ A), Agent\ B, Key\ (pubSK\ B),$

$Crypt\ (priSK\ TTP)\ \{|Crypt\ (priEK\ TTP)\ \{|Nonce\ Na', Agent\ B, Key\ (pubSK$
$B), Number\ Text6|\},$

$\{|Nonce\ Na', Agent\ B, Key\ (pubSK\ B), Number\ Text6|\}|\},$

$Crypt\ (priSK\ TTP)\ \{|Crypt\ (priEK\ TTP)\ \{|Nonce\ Nb, Agent\ A, Key\ (pubSK\ A),$
$Number\ Text5|\},$

$\{|Nonce\ Nb, Agent\ A, Key\ (pubSK\ A), Number\ Text5|\}|\},$

$Number\ Text7|\} \in set\ evs$

apply $(erule\ rev\_mp)$

apply $(erule\ iso.induct)$

apply $simp\_all$

apply *auto*[*1*]

apply *spy_analz*

apply *blast*

done

lemma *ISO5_imp_ISO1*:

⟦*Says A B* {|*Number Text9*, *Agent A*, *Key* (*pubSK A*),

Crypt (*priSK TTP*) {|*Crypt* (*priEK TTP*) {|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|},

{|*Nonce Nb*, *Agent A*, *Key* (*pubSK A*), *Number Text5*|}|},

Crypt (*priSK A*) {|*Crypt* (*priEK A*) {|*Nonce Nb*, *Nonce Na*, *Agent B*, *Agent A*, *Number Text8*|},

{|*Nonce Nb*, *Nonce Na*, *Agent B*, *Agent A*, *Number Text8*|}|} ∈ *set evs*;

A ∉ *bad*; *evs* ∈ *iso*⟧

⟹ *Says A B* {|*Agent A*, *Nonce Na*, *Number Text1*|} ∈ *set evs*

apply (*erule rev_mp*)

apply (*erule iso.induct*)

apply *simp_all*

apply *spy_analz*

done

lemma *Fake_insert*:

⟦*Crypt* (*priSK A*) *Y* ∈ *parts* (*insert X* (*knows Spy evs*));

X ∈ *synth* (*analz* (*knows Spy evs*));

TA ∉ *bad*; *evs* ∈ *iso*⟧

⟹ *Crypt* (*priSK A*) *Y* ∈ *parts* (*spies evs*)

apply *spy_analz*

done

theorem *A_auth_B*:

⟦{|*Agent B*, *Nonce Na*, *Nonce Nb*, *Number Text3*,

Crypt (*priSK B*) {|*Crypt* (*priEK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|},

{|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|}|} ∈ *parts* (*spies evs*);

Says A B {|*Agent A*, *Nonce Na*, *Number Text1*|} ∈ *set evs*;

TA ∉ *bad*; *B* ∉ *bad*; *evs* ∈ *iso*⟧

⟹ *Says B A* {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Number Text3*,

Crypt (*priSK B*) {|*Crypt* (*priEK B*) {|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|},

{|*Agent B*, *Nonce Na*, *Nonce Nb*, *Agent A*, *Number Text2*|}|} ∈ *set evs*

apply (*blast dest*: *sig_B_origin*)

done

lemma *signature_form*:

⟦*Crypt* (*priSK A*) *Y* ∈ *parts* (*spies evs*);

Crypt (*priSK TA*) *Z* ∈ *parts* {*Y*} ∨ *Crypt* (*priEK TA*) *Z* ∈ *parts* {*Y*};

A ∉ *bad*; *TA* ∉ *bad*; *evs* ∈ *iso*⟧

⟹ *False*

apply (*erule rev_mp*)

apply (*erule iso.induct*)

apply *simp_all*

apply (*metis Pack_conf not_parts_not_analz*)

apply *spy_analz*

apply *auto*

done

lemma *not_in_msg_A*:

⟦*Crypt* (*priSK A*) *Y* ∈ *parts* (*spies evs*);

∃ *Z*. (*Crypt* (*priSK TA*) *Z* ∈ *parts* {*Y*} ∨

Crypt (*priEK TA*) *Z* ∈ *parts* {*Y*} ∨

Crypt (*pubK TA*) *Z* ∈ *parts* {*Y*} ∨

Crypt (*pubSK TA*) *Z* ∈ *parts* {*Y*}) ∨

Agent TA ∈ *parts*{*Y*};

TA ∉ *bad*; *evs* ∈ *iso*⟧

⟹ *False*

apply (*erule rev_mp*)

apply (*erule iso.induct*)

apply *simp_all*

apply (*blast dest*: *Pack_conf not_parts_not_analz*)

apply *spy_analz*

apply *force*+

done

lemma *possible*:

⟦*TA* ∈ *bad*; *evs* ∈ *iso*⟧ ⟹ *Key* (*priSK A*) ∈ *analz* (*spies evs*)

apply *auto*

done

lemma *TA_can_generate_sig*:

⟦*TA* ∈ *bad*; *evs* ∈ *iso*⟧

⟹ *Crypt* (*priSK A*) {|*Crypt* (*priEK TA*) (*Number N*), *Number N*|} ∈ *synth*(*analz* (*spies evs*))

apply *auto*

done

lemma *Fake_generates_AIBS*:

TA ∈ *bad* ⟹ ∃ *D B N*. ∃ *evs* ∈ *iso*.

Says Spy D (*Crypt* (*priSK B*) {|*Crypt* (*priEK TA*) (*Number N*), *Number N*|}) ∈ *set evs* ∧ *B* ≠ *TA*

apply (*intro exI bexI*)

apply (*rule_tac* [2] *iso.Nil* [*THEN iso.Fake*], *possibility*)

apply *auto*

done

definition *candidates* :: *agent* ⇒ *event list* ⇒ *agent set* **where**

candidates A evs ≡

{*C*. *C* ≠ *Spy* ∧

(∃ *Y Z*. *Crypt* (*priSK A*) *Y* ∈ *parts* (*knows Spy evs*) ∧

(*A* = *C* ∨ *Key* (*priSK A*) ∈ *initState C*) ∧

(*Crypt* (*priSK C*) *Z* ∈ *parts* {*Y*} ∨ *Crypt* (*priEK C*) *Z* ∈ *parts* {*Y*}))}

theorem *candidates_AIBS_none*:

$[\![D \in candidates\ A\ evs; TA \notin bad; A \neq TTP; A \neq TA; A \notin bad; evs \in iso]\!] \Longrightarrow False$

oops

theorem *candidates_AIBS_A*:

$[\![D \in candidates\ A\ evs; TA \notin bad; A \notin bad; evs \in iso]\!] \Longrightarrow D = A$

apply (*unfold candidates_def*)

apply (*induct D*)

apply *force*

apply (*case_tac nat = 0*)

apply (*force dest*!: *signature_form*)

apply *auto*

done

theorem *candidates_AIBS_A_TA*:

$[\![D \in candidates\ A\ evs; A \notin bad; evs \in iso]\!] \Longrightarrow D = A \vee D = TA$

apply (*unfold candidates_def*)

apply (*induct D*)

apply *auto*

done

theorem *candidates_AIBS_possible*:

$[\![A \neq Spy; ev = Says\ A\ B\ (Crypt\ (priSK\ A)\ (Crypt\ (priEK\ A)\ X))\ \#\ evs]\!]$

$\Longrightarrow A \in candidates\ A\ ev$

apply (*unfold candidates_def*)

apply *auto*

done

end

A.3 Proofs for the FOO Protocol

A.3.1 Foo.thy

theory *Foo* **imports** *Public* **begin**

abbreviation

Adm :: *agent* **where**

Adm ≡ *Friend 0*

abbreviation

Col :: *agent* **where**

Col ≡ *Friend* (*Suc*(*0*))

consts *anms* :: *nat*

definition *Anms* :: [*agent*, *agent*, *msg*] ⇒ *event* **where**

Anms A B X ≡ *Says A B* {|*Number anms*, *X*|}

definition *isAnms* :: *msg* => *bool* **where**

isAnms M ≡ (∃ *Y*. *M* = *MPair* (*Number anms*) *Y*)

lemma *isAnms_check*:

isAnms {|*Number anms*, *X*|} ∧ ¬ *isAnms* (*Crypt K Y*) ∧ ¬ *isAnms* {|*Agent V*, *Z*|}

by (*unfold isAnms_def*, *auto*)

inductive_set *foo* :: *event list set*

where

| *Unblinding*:

⟦*evsb* ∈ *foo*; *Crypt* (*priSK V*) *BSBody* ∈ *analz* (*spies evsb*);

BSBody = *Crypt b* (*Crypt c* (*Nonce N*));

b ∈ *symKeys*; *Key b* ∈ *analz* (*spies evsb*)⟧

⟹ *Notes Spy* (*Crypt* (*priSK V*) (*Crypt c* (*Nonce N*))) # *evsb* ∈ *foo*

| *EV1*:

⟦*evs1* ∈ *foo*; *V* ≠ *Adm*; *V* ≠ *Col*; *c* ∈ *symKeys*; *Key c* ∉ *used evs1*;

b ∈ *symKeys*; *Key b* ∉ *used evs1*; *b* ≠*c*; *Nonce Nv* ∉ *used evs1*⟧

$\Longrightarrow$ *Says V Adm* $\{\!|$*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))$|\!\}$

Notes V (*Key c*) # *Notes V* (*Key b*) # *evs1* $\in$ *foo*

| *EV2*:

$[\![$*evs2* $\in$ *foo*; *V* $\neq$ *Adm*; *V* $\neq$ *Col*; *Notes Adm* (*Agent V*) $\notin$ *set evs2*;

Gets Adm $\{\!|$*Agent V*, *Crypt* (*priSK V*) *BSBody*$|\!\}$ $\in$ *set evs2*;

BSBody = *Crypt P R*; $\forall$ *X Y*. *MPair X Y* $\notin$ *parts*$\{$*BSBody*$\}$$]\!]$

$\Longrightarrow$ *Says Adm V* (*Crypt* (*priSK Adm*) *BSBody*)

Notes Adm (*Agent V*) # *evs2* $\in$ *foo*

| *EV3*:

$[\![$*evs3* $\in$ *foo*; *Says V Adm* $\{\!|$*Agent V*,

Crypt (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))$|\!\}$ $\in$ *set evs3*;

Gets V (*Crypt* (*priSK Adm*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))) $\in$ *set evs3*$]\!]$

$\Longrightarrow$ *Anms V Col* (*Crypt* (*priSK Adm*) (*Crypt c* (*Nonce Nv*))) # *evs3* $\in$ *foo*

| *EV4*:

$[\![$*evs4* $\in$ *foo*; *V* $\neq$ *Adm*; *V* $\neq$ *Col*; *Says Col Col CX* $\notin$ *set evs4*;

Gets Col $\{\!|$*Number anms*, *Crypt* (*priSK Adm*) *CX*$|\!\}$ $\in$ *set evs4*;

CX = *Crypt P R*; $\forall$ *X Y*. *MPair X Y* $\notin$ *parts*$\{$*CX*$\}$$]\!]$

$\Longrightarrow$ *Says Col Col CX* # *evs4* $\in$ *foo*

| *EV5*:

$[\![$*evs5* $\in$ *foo*; *Says V Adm* $\{\!|$*Agent V*,

Crypt (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))$|\!\}$ $\in$ *set evs5*;

Gets Col (*Crypt c* (*Nonce Nv*)) $\in$ *set evs5*; *Key c* $\in$ *analz* (*knows V evs5*);

c $\notin$ *range shrK*; *c* $\in$ *symKeys*$]\!]$

$\Longrightarrow$ *Anms V Col* (*Key c*) # *evs5* $\in$ *foo*

| *EV6*:

$[\![$*evs6* $\in$ *foo*; *Gets Col* $\{\!|$*Number anms*, *Key c*$|\!\}$ $\in$ *set evs6*;

Gets Col (*Crypt c* (*Nonce Nv*)) $\in$ *set evs6*; *Says Col Col* (*Nonce Nv*) $\notin$ *set evs6*$]\!]$

$\Longrightarrow$ *Says Col Col* (*Nonce Nv*) # *evs6* $\in$ *foo*

declare *knows_Spy_partsEs* [*elim*]

declare *analz_into_parts* [*dest*]

declare *Fake_parts_insert_in_Un* [*dest*]

declare *image_eq_UN* [*simp*]

declare *Anms_def* [*simp*]

definition *KeyWithNonce* :: [*key*, *nat*, *event list*] => *bool* **where**
KeyWithNonce c Nv evs ≡
∃ *V B b*.
Says V B {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|}
∈ *set evs*

lemma *Gets_imp_Says*:
⟦*Gets B X* ∈ *set evs*; *evs* ∈ *foo*⟧ ⟹ ∃ *D*. *Says D B X* ∈ *set evs*
apply (*erule rev_mp*)
apply (*erule foo.induct*)
apply *auto*
done

lemma *Gets_imp_knows_Spy*:
⟦*Gets B X* ∈ *set evs*; *evs* ∈ *foo*⟧ ⟹ *X* ∈ *knows Spy evs*
apply (*blast dest*!: *Gets_imp_Says Says_imp_knows_Spy*)
done

lemma *Gets_imp_parts_knows_Spy* [*dest*]:
⟦*Gets D X* ∈ *set evs*; *evs* ∈ *foo*⟧ ⟹ *X* ∈ *parts*(*spies evs*)
apply (*drule Gets_imp_Says*, *simp*)
by (*blast dest*: *Says_imp_knows_Spy parts.Inj*)

lemma *Gets_imp_knows*:
⟦*Gets B X* ∈ *set evs*; *evs* ∈ *foo*⟧ ⟹ *X* ∈ *knows B evs*
apply (*case_tac B*=*Spy*)
by (*blast dest*: *Gets_imp_knows_Spy Gets_imp_knows_agents*)+

lemma *CX_analz*:
⟦*Gets* (*Friend* (*Suc 0*)) {|*Number anms*, *Crypt* (*priSK Adm*) *CX*|} ∈ *set evs*; *evs* ∈ *foo*⟧
⟹ *CX* ∈ *analz* (*spies evs*)

apply (*drule Gets_imp_Says*, *auto*)

by (*drule Says_imp_analz_Spy*, *auto*)

lemma *new_keys_not_used* [*simp*]:

⟦*Key K* ∉ *used evs*; *K* ∈ *symKeys*; *evs* ∈ *foo*⟧

⟹ *K* ∉ *keysFor* (*parts* (*spies evs*))

apply (*erule rev_mp*, *erule foo.induct*)

apply (*simp_all*,*auto*)

apply (*force dest*!: *keysFor_parts_insert*)

apply (*blast dest*: *Crypt_imp_keysFor*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply (*blast dest*: *Crypt_imp_keysFor*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply *auto*

apply (*force dest*: *parts_cut_eq*)

apply (*blast dest*: *Crypt_imp_keysFor*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply *auto*

apply (*force dest*: *parts_cut_eq*)

apply (*blast dest*: *Crypt_imp_keysFor*)

apply (*blast dest*: *CX_analz Crypt_imp_keysFor*)

apply (*force dest*!: *Gets_imp_parts_knows_Spy*)

done

lemma *V_struct* [*simp*]:

⟦*Says V A* {|*Agent V*, *Crypt* (*priSK V*) *BSBody* |} ∈ *set evs*;

V ∉ *bad*; *evs* ∈ *foo*⟧

⟹ ∃ *b c Nv*. *BSBody* = *Crypt b* (*Crypt c* (*Nonce Nv*))

∧ *c* ∈ *symKeys* ∧ *b* ∈ *symKeys*

by (*erule rev_mp*, *erule foo.induct*, *simp_all*, *spy_analz*)

lemma *Spy_see_priK* [*simp*]:

evs ∈ *foo* ⟹ (*Key* (*priSK V*) ∈ *parts* (*spies evs*)) = (*V* ∈ *bad*)

apply (*erule foo.induct*, *simp_all*)

apply (*blast dest*: *parts_cut_eq*)

apply (*blast dest*: *parts_cut_eq*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply *auto*

apply (*force dest*: *parts_cut_eq*)

apply (*blast dest*: *parts_cut_eq*)+

done

lemma *priSKV_nosynth*:

⟦*Crypt* (*priSK V*) *X* ∈ *synth* (*analz* (*knows Spy evs*));

V ∉ *bad*; *evs* ∈ *foo*⟧

⟹ *Crypt* (*priSK V*) *X* ∈ *analz* (*knows Spy evs*)

by *auto*

lemma *EV2_analz*:

⟦*Gets Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt K X*)|} ∈ *set evs*;

evs ∈ *foo*⟧

⟹ (*X* ∉ *analz*(*spies evs*) ∧ *Key*(*invKey*(*K*)) ∉ *analz*(*spies evs*)) ∨

(*X* ∈ *analz*(*spies evs*)) ∨

(∃ *c Nv*. *X* = *Crypt c* (*Nonce Nv*) ∧ *K* ∈ *symKeys* ∧ *c* ∈ *symKeys*)

apply (*metis Gets_imp_knows_Spy analz.simps invKey invKey_K spies_pubK*)

done

lemma *Crypt_used_imp_spies*:

⟦*Crypt K X* ∈ *used evs*; *evs* ∈ *foo*⟧ ⟹ *Crypt K X* ∈ *parts* (*spies evs*)

apply (*erule rev_mp*, *erule foo.induct*)

apply (*simp_all add*: *parts_insert_knows_A*)

apply (*blast dest*: *parts_cut parts_insertI*)

done

lemma *EV2_an*:

⟦*Gets Adm* {|*Agent V*, *Crypt* (*priSK V*) *X*|} ∈ *set evs*; *evs* ∈ *foo*⟧ ⟹ *X* ∈ *parts* (*spies evs*)

by (*drule Gets_imp_Says*, *auto*)

lemma *new_keys_not_analzd*:

⟦*K* ∈ *symKeys*; *evs* ∈ *foo*; *Key K* ∉ *used evs*⟧

⟹ *K* ∉ *keysFor* (*analz* (*knows Spy evs*))

by (*blast dest*: *new_keys_not_used intro*: *keysFor_mono* [*THEN subsetD*])

lemma *unique_Nv2*:

⟦*Crypt c* (*Nonce Nv*) ∈ *parts* (*spies evs*);

Crypt c (*Nonce Nva*) ∈ *parts* (*spies evs*);

Key c ∉ *analz* (*spies evs*); *evs* ∈ *foo*⟧

⟹ *Nv*=*Nva*

apply (*erule rev_mp*,*erule rev_mp*,*erule rev_mp*)

apply (*erule foo.induct*, *simp_all*)

apply (*blast dest*: *analz_insertI*)

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used analz_insertI*)

apply (*rule conjI*, *clarsimp*, *rule conjI*, *clarsimp*, *rule conjI*)

apply (*force*, *clarsimp*, *rule conjI*, *force*, *clarsimp*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply (*case_tac Key c* ∉ *analz* (*knows Spy evs2*), *clarsimp*)

apply (*case_tac Crypt c* (*Nonce Nva*) ∈ *parts* (*knows Spy evs2*), *clarsimp*)

apply (*metis parts_cut_eq spies_partsEs*(*2*), *clarsimp*)

apply (*metis parts_cut_eq spies_partsEs*(*2*), *clarsimp*)

apply (*blast dest*: *analz_insertI*)

apply (*clarsimp*, *rule conjI*, *clarsimp*, *force*, *clarsimp*, *rule conjI*)

apply (*force*, *clarsimp*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply (*metis MPair_parts parts_cut_eq spies_partsEs*(*2*))

apply (*clarsimp*, *rule conjI*, *clarsimp*, *rule conjI*, *force*, *clarsimp*)

apply (*rule conjI*, *force*, *clarsimp*)

apply (*drule EV2_an*, *assumption*)

apply (*metis analz_insertI parts_cut_eq spies_partsEs*(*2*))

apply (*clarsimp*, *rule conjI*, *force*, *clarsimp*, *rule conjI*, *force*, *clarsimp*)

apply (*drule EV2_an*, *assumption*)

apply (*metis parts_cut_eq spies_partsEs*(2))

apply (*force*, *clarsimp*, *rule conjI*, *clarsimp*, *rule conjI*, *force*)

apply (*clarsimp*, *rule conjI*, *force*, *clarsimp*)

apply (*drule CX_analz*, *assumption*)

apply (*metis analz.Decrypt analz_insertI analz_into_parts parts_cut_eq*)

apply (*clarsimp*, *rule conjI*, *force*, *clarsimp*, *rule conjI*, *force*, *clarsimp*)

apply (*drule CX_analz*, *assumption*)

apply (*metis analz_into_parts parts_cut_eq spies_partsEs*(2))

by (*blast dest*: *analz_insertI*)

lemma *unique_Nv*:

⟦*Crypt c* (*Nonce Nv*) ∈ *parts* (*spies evs*);

Crypt c (*Nonce Nva*) ∈ *parts* (*spies evs*);

Key c ∉ *parts* (*spies evs*); *evs* ∈ *foo*⟧

⟹ *Nv*=*Nva*

by (*blast dest*: *unique_Nv2 analz_into_parts*)

lemma *b_fixed*:

⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

Key b ∉ *parts* (*knows Spy evs*); *V* ∉ *bad*; *evs* ∈ *foo*⟧

⟹ *Crypt b* (*Nonce Nv'*) ∉ *parts* (*spies evs*)

apply (*erule rev_mp*,*erule rev_mp*)

apply (*erule foo.induct*)

apply *simp_all*

apply (*case_tac V* = *Spy*)

apply *blast*

apply *clarsimp*

apply (*drule Fake_parts_insert* [*THEN subsetD*], *simp*)

apply *clarsimp*

apply (*blast dest*: *parts_insertI*)

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply (*metis MPair_parts parts_cut_eq spies_partsEs*(2))

apply (*clarsimp*, *blast*)

apply *clarsimp*

apply *auto*

apply (*blast dest*: *parts_insertI*)

apply (*case_tac Crypt* (*priSK Adm*) (*Crypt P R*) ∈ *parts* (*spies evs4*))

apply (*blast dest*: *parts_cut_eq*)+

done

theorem *Spy_see_b* [*simp*]:

⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

V ∉ *bad*; *evs* ∈ *foo*⟧

⟹ *Key b* ∉ *parts* (*spies evs*)

apply (*erule rev_mp*, *erule foo.induct*, *simp_all*)

apply *spy_analz*

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply (*rule conjI*, *force*, *force*)

apply (*blast dest*: *parts_cut_eq*)

apply (*blast dest*: *b_fixed*)

done

theorem *b_secrecy* [*dest*]:

⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

V ∉ *bad*; *evs* ∈ *foo*⟧

⟹ *Key b* ∉ *analz* (*spies evs*)

by (*blast dest*: *Spy_see_b*)

lemma *EV1_alternative*:

⟦*Says Va Adm* {|*Agent Va*, *Crypt* (*priSK Va*) (*Crypt ba* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

Says V Adm {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

Key c ∉ parts (spies evs); V ∉ bad; evs ∈ foo⟧

⟹ Va = V ∨ Va = Spy

apply (*erule rev_mp*, *erule rev_mp*,*erule rev_mp*)

apply (*erule foo.induct*, *simp_all*)

apply (*blast dest*: *parts_insertI*)+

done

theorem *Spy_see_c* [*simp*]:

⟦Says V Adm {|Agent V, Crypt (priSK V) (Crypt b

(Crypt c (Nonce Nv)))|} ∈ set evs;

Anms V Col (Key c) ∉ set evs;

V ∉ bad; evs ∈ foo⟧

⟹ Key c ∉ parts (spies evs)

apply (*erule rev_mp*,*erule rev_mp*,*erule rev_mp*)

apply (*erule foo.induct*, *simp_all*)

apply *spy_analz*

apply *auto*

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

apply (*frule EV2_an*, *assumption*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply *auto*

apply (*frule EV2_an*, *assumption*)

apply (*blast dest*: *parts_cut_eq*)

apply (*blast dest*: *parts_cut_eq*)

apply (*blast dest*: *EV1_alternative unique_Nv*)

done

theorem *c_secrecy* [*simp*]:

⟦Says V Adm {|Agent V, Crypt (priSK V) (Crypt b

(Crypt c (Nonce Nv)))|} ∈ set evs;

Anms V Col (Key c) ∉ set evs; V ∉ bad; evs ∈ foo⟧

⟹ Key c ∉ analz (spies evs)

by (*blast dest*: *Spy_see_c*)

lemma *double_fresh_insert*:
⟦*Nonce Nv* ∈ *analz* (*insert* (*Key c*) (*insert* (*Key b*) (*spies evs*)));
Key b ∉ *used evs*; *b* ∈ *symKeys*; *Key c* ∉ *used evs*; *c* ∈ *symKeys*;
evs ∈ *foo*⟧
⟹ *Nonce Nv* ∈ *analz* (*knows Spy evs*)
apply (*drule new_keys_not_analzd*, *assumption*, *assumption*)+
by *auto*

lemma *unique_c*:
⟦*Crypt c* (*Nonce Nv*) ∈ *parts*(*spies evs*);
Crypt ca (*Nonce Nv*) ∈ *parts*(*spies evs*);
Nonce Nv ∉ *analz* (*spies evs*); *evs* ∈ *foo*⟧
⟹ *c*=*ca*
apply (*erule rev_mp*, *erule rev_mp*, *erule rev_mp*)
apply (*erule foo.induct*, *simp_all*)
prefer *4*
apply (*frule Gets_imp_parts_knows_Spy*, *assumption*)
apply (*clarsimp*, *rule conjI*, *clarsimp*)
apply (*metis analz_insertI parts_cut_eq spies_partsEs*(*2*))
apply (*metis analz_insertI parts_cut_eq spies_partsEs*(*2*))
prefer *5*
apply (*rule conjI*)
apply (*drule CX_analz*, *assumption*)
apply (*metis analz_conj_parts analz_insertI parts_cut_eq spies_partsEs*(*2*))
apply (*drule CX_analz*, *assumption*)
apply (*metis analz_conj_parts parts_cut_eq spies_partsEs*(*2*))
by (*blast dest*: *analz_insertI parts_cut_eq*)+

lemma *EV5_msg_in_parts_spies*:
⟦*Says V A* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b X*)|} ∈ *set evs*⟧
⟹ *X* ∈ *parts* (*spies evs*)

by *auto*

lemma *pushing3*:

Key K ∈ *analz* (*insert X* (*insert* (*Agent V*) (*Key ' KK* ∪ *knows Spy evs*))) ⟹

Key K ∈ *analz* (*insert X* (*Key ' KK* ∪ *knows Spy evs*))

by (*metis analz_insert_Agent insertE insert_commute msg.simps*(*12*))

lemma *analz_image_freshK* [*rule_format*]:

evs ∈ *foo* ⟹

∀ *K KK*. *KK* ⊆ −(*range shrK*) ⟶

(*Key K* ∈ *analz* (*Key'KK* ∪ (*spies evs*))) =

(*K* ∈ *KK* ∨ *Key K* ∈ *analz* (*spies evs*))

apply (*erule foo.induct*) **prefer** *8*

apply (*thin_tac CX* = *Crypt P R*) **prefer** *7*

apply (*thin_tac BSBody* = *Crypt P R*)

apply (*unfold Anms_def*)

apply *analz_freshK*

defer defer

apply *spy_analz* **defer**

apply (*drule CX_analz*, *assumption*)

apply (*metis analz_cut analz_image_freshK_simps*(*61*) *analz_insertI*)

apply (*case_tac Adm* ∉ *bad*)

apply (*drule Gets_imp_knows_Spy*, *assumption*)

apply (*case_tac BSBody* ∈ *analz* (*spies evs2*))

apply (*metis analz_cut analz_image_freshK_simps*(*61*) *analz_insertI*)

apply (*case_tac Crypt* (*priSK V*) *BSBody* ∈ *analz* (*spies evs2*))

apply (*metis analz.Decrypt analz_spies_pubK invKey*)

apply (*metis MPair_analz analz.Inj*)

apply (*auto simp del*: *image_insert image_eq_UN*

simp add: *analz_image_freshK_simps*)

apply (*case_tac BSBody* ∈ *analz* (*spies evs2*))

apply (*metis analz_cut analz_image_freshK_simps*(*61*) *analz_insertI pushing3*)

by (*metis Gets_imp_knows_Spy MPair_analz analz.Decrypt analz.Inj analz_spies_pubK invKey*)

lemma *analz_insert_freshK*:

⟦*evs* ∈ *foo*; *KAB* ∉ *range shrK*⟧ ⟹

(*Key K* ∈ *analz* (*insert* (*Key KAB*) (*spies evs*))) =

(*K* = *KAB* ∨ *Key K* ∈ *analz* (*spies evs*))

by (*simp only*: *analz_image_freshK analz_image_freshK_simps*)

lemma *c_sym*:

⟦*Crypt c* (*Nonce Nv*) ∈ *parts*(*spies evs*);

Nonce Nv ∉ *analz* (*spies evs*); *evs* ∈ *foo*⟧

⟹ *c* ∈ *symKeys*

apply (*erule rev_mp*,*erule rev_mp*, *erule foo.induct*, *simp_all*)

apply *spy_analz*

apply (*blast dest*: *parts_cut_eq analz_insertI*)

apply (*blast dest*: *parts_cut_eq analz_insertI*)

defer

apply (*blast dest*: *parts_cut_eq analz_insertI*)

apply (*blast dest*: *parts_cut_eq analz_insertI*)

apply (*blast dest*: *parts_cut_eq analz_insertI*)

apply (*frule EV2_an*, *assumption*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply (*rule conjI*, *clarsimp*, *rule conjI*, *clarsimp*, *rule conjI*)

apply (*blast dest*: *analz_insertI*, *clarsimp*)

apply (*metis analz_insertI parts_cut_eq spies_partsEs*(2))

apply *clarsimp*

apply (*metis parts_cut_eq spies_partsEs*(2))

apply (*metis analz_insertI parts_cut_eq spies_partsEs*(2))

done

lemma *Nonce_secrecy_lemma*:

P ⟶ (*X* ∈ *analz* (*G Un H*)) ⟶ (*X* ∈ *analz H*) ⟹

$P \longrightarrow (X \in analz\ (G\ Un\ H)) = (X \in analz\ H)$

by (*blast intro*: *analz_mono* [*THEN subsetD*])

lemma *KeyWithNonceI*:

⟦*Says V B* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*⟧

⟹ *KeyWithNonce c Nv evs*

by (*unfold KeyWithNonce_def*, *blast*)

lemma *KeyWithNonce_Says* [*simp*]:

KeyWithNonce c Nv (*Says V B X* # *evs*) =

(∃ *b*. *X* = {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|}

∨ *KeyWithNonce c Nv evs*)

by (*simp add*: *KeyWithNonce_def*, *blast*)

lemma *KeyWithNonce_Notes* [*simp*]:

KeyWithNonce c Nv (*Notes B X* # *evs*) = *KeyWithNonce c Nv evs*

by (*simp add*: *KeyWithNonce_def*)

lemma *KeyWithNonce_Gets* [*simp*]:

KeyWithNonce c Nv (*Gets B X* # *evs*) = *KeyWithNonce c Nv evs*

by (*simp add*: *KeyWithNonce_def*)

A fresh key cannot be associated with any nonce (with respect to a given trace).

New argument wrt Yahalom because of new freshness argument on commitment key

lemma *fresh_not_KeyWithNonce*:

⟦*Key c* ∉ *used evs*; *c* ∈ *symKeys*; *evs* ∈ *foo*⟧ ⟹ ¬ *KeyWithNonce c Nv evs*

apply (*simp add*: *KeyWithNonce_def*)

by (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

lemma *EV1_analz*:

⟦*Gets Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce N*)))|}

∈ *set evs*; *evs* ∈ *foo*⟧

⟹ *Crypt b* (*Crypt c* (*Nonce N*)) ∈ *analz* (*spies evs*)

by (*force dest*: *Gets_imp_Says Says_imp_analz_Spy*)

lemma *Notes_analz_c_lemma*:

Crypt c (*Nonce Nv*) ∈ *analz* (*knows Spy evs*) ⟹

Crypt c (*Nonce Nv*) ∈ *analz* (*insert* (*Key c*) (*knows Spy evs*))

by (*blast dest*: *analz_insertI*)

lemma *Notes_analz_c_lemma2*:

Crypt c (*Crypt c* (*Nonce Nv*)) ∈ *analz* (*knows Spy evs*) ⟹

Crypt c (*Crypt c* (*Nonce Nv*)) ∈ *analz* (*insert* (*Key c*) (*knows Spy evs*))

by (*blast dest*: *analz_insertI*)

lemma *Notes_analz_c_lemma3*:

Crypt b (*Crypt c* (*Nonce Nv*)) ∈ *analz* (*knows Spy evs*) ⟹

Crypt b (*Crypt c* (*Nonce Nv*)) ∈ *analz* (*insert* (*Key b*) (*knows Spy evs*))

by (*blast dest*: *analz_insertI*)

lemma *Notes_analz_c*:

⟦*Gets V* (*Crypt c* (*Nonce Nv*)) ∈ *set evs*; *evs* ∈ *foo*⟧

⟹ *Nonce Nv* ∈ *analz* (*insert* (*Key c*) (*spies evs*))

apply (*case_tac Crypt c* (*Nonce Nv*) ∈ *analz* (*spies evs*))

prefer *2*

apply (*blast dest*: *Gets_imp_knows_Spy*)

apply (*case_tac c* ∈ *symKeys*)

apply (*metis analz.Inj analz_Decrypt' analz_insertI insertI1*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply (*blast dest*: *analz_insertI c_sym*)

done

lemma *KWN_Nv*:

⟦*KeyWithNonce K Nv evs*; *Key K* ∉ *analz* (*spies evs*); *Nva* ≠ *Nv*; *evs* ∈ *foo*⟧

⟹ ¬ *KeyWithNonce K Nva evs*

apply (*unfold KeyWithNonce_def*)

by (*blast dest*: *EV5_msg_in_parts_spies unique_Nv2*)

lemma *KWN_K*:

⟦*KeyWithNonce K Nv evs*; *K≠ K'*; *Nonce Nv ∉ analz* (*spies evs*); *evs ∈ foo*⟧

⟹ ¬ *KeyWithNonce K' Nv evs*

apply (*unfold KeyWithNonce_def*)

by (*blast dest*: *EV5_msg_in_parts_spies unique_c*)

lemma *EV2_Nv_analz*:

⟦*evs ∈ foo*;

Gets Adm {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce N*)))|} ∈ *set evs*;

c ∈ symKeys; *b ∈ symKeys*; *Key c ∈ analz* (*knows Spy evs*)⟧

⟹ *Nonce N ∈ analz* (*insert* (*Key b*) (*knows Spy evs*))

apply (*drule EV1_analz*, *assumption*)

by (*blast dest*: *analz_insertI analz_Decrypt'*)

lemma *EV2_Nv_analzbis*:

⟦*evs ∈ foo*;

Gets Adm {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce N*)))|} ∈ *set evs*;

c ∈ symKeys; *b ∈ symKeys*; *Key b ∈ analz* (*knows Spy evs*)⟧

⟹ *Nonce N ∈ analz* (*insert* (*Key c*) (*knows Spy evs*))

apply (*drule EV1_analz*, *assumption*)

by (*blast dest*: *analz_insertI analz_Decrypt'*)

lemma *EV2_Nv_analzter*:

⟦*evs ∈ foo*;

Gets Adm {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce N*)))|} ∈ *set evs*;

c ∈ symKeys; *b ∈ symKeys*⟧

⟹ *Nonce N ∈ analz* (*insert* (*Key c*) (*insert* (*Key b*) (*knows Spy evs*)))

apply (*drule EV1_analz*, *assumption*)

by (*blast dest*: *analz_insertI analz_Decrypt'*)

lemma *Nonce_analz_cut*:

⟦*Nonce Nv* ∈ *analz* (*Key* ‘ *insert c KK* ∪ *knows Spy evs*); *Key c* ∈ *analz* (*spies evs*)⟧ ⟹
Nonce Nv ∈ *analz* (*Key* ‘ *KK* ∪ *knows Spy evs*)

by (*auto*, *blast dest*:*analz_cut analz_image_freshK_simps*(*61*))

lemma *pushing*:

(*Nonce Nv* ∈ *analz* (*insert X* (*insert* (*Agent V*) (*Key* ‘ *KK* ∪ *knows Spy evs*))))

= (*Nonce Nv* ∈ *analz* (*insert X* (*Key* ‘ *KK* ∪ *knows Spy evs*)))

apply *auto*

apply (*metis analz_insert_Agent insertE insert_commute msg.simps*(*11*))

by (*drule analz_insertI*, *simp only*: *insert_commute*)

lemma *pushing_bis*:

(*Nonce Nv* ∈ *analz* (*insert X* (*insert* (*Agent V*) (*knows Spy evs*)))) =

(*Nonce Nv* ∈ *analz* (*insert X* (*knows Spy evs*)))

apply *safe*

apply (*metis analz_insert_Agent insertE insert_commute msg.simps*(*11*))

by (*drule analz_insertI*, *simp only*: *insert_commute*)

lemma *Nv_extract*:

⟦*Gets Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt K* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

K ∈ *symKeys*; *c* ∈ *symKeys*; *evs* ∈ *foo*⟧

⟹ *Nonce Nv* ∈ *analz* (*insert* (*Key K*) (*insert* (*Key c*) (*knows Spy evs*)))

apply (*drule EV2_Nv_analzter*, *assumption*+)

by (*simp only*: *insert_commute*)

lemma *Nv_extract2*:

⟦*Gets Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt K* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

K ∈ *symKeys*; *c* ∈ *symKeys*; *Key c* ∈ *analz* (*knows Spy evs*); *evs* ∈ *foo*⟧

⟹ *Nonce Nv* ∈ *analz* (*insert* (*Key K*) (*knows Spy evs*))

by (*blast dest*: *EV2_Nv_analz*)

lemma *Nv_extract3_lemma*:

⟦*Gets Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt K X*)|} ∈ *set evs*; *evs* ∈ *foo*⟧

$\Longrightarrow X \in analz\ (insert\ (Key\ (invKey\ K))\ (spies\ evs))$

apply (*drule Gets_imp_knows_Spy*, *assumption*)

apply (*case_tac Crypt K X* $\in$ *analz* (*knows Spy evs*))

prefer *2*

apply (*metis analz.simps analz_spies_pubK invKey*)

apply (*case_tac Crypt K X* $\in$ *analz* (*insert* (*Key* (*invKey K*)) (*knows Spy evs*)))

apply *auto*

by (*blast dest*: *analz_insertI*)

lemma *Nv_extract3*:

$[\![Gets\ Adm\ \{\!|Agent\ V, Crypt\ (priSK\ V)\ (Crypt\ K\ X)|\!\} \in set\ evs;$

$Nonce\ Nv \in analz\ (insert\ X\ (Key\ `\ KK \cup knows\ Spy\ evs));\ evs \in foo]\!]$

$\Longrightarrow Nonce\ Nv \in analz\ (insert\ (Key\ (invKey\ K))\ (Key\ `\ KK \cup knows\ Spy\ evs))$

apply (*drule Nv_extract3_lemma*, *assumption*)

apply (*erule analz.induct*, *auto*)

by (*erule analz.induct*, *blast*+)

lemma *priSKV_nosynth2*:

$[\![parts\ (insert\ X\ (knows\ Spy\ evs)) \subseteq synth\ (analz\ (knows\ Spy\ evs)) \cup parts\ (knows\ Spy\ evs);$

$Crypt\ (priSK\ V)\ (Crypt\ P\ R) \in parts\ (insert\ X\ (knows\ Spy\ evs));$

$V \notin bad;\ evs \in foo]\!]$

$\Longrightarrow Crypt\ (priSK\ V)\ (Crypt\ P\ R) \in parts\ (knows\ Spy\ evs)$

by (*force dest*: *priSKV_nosynth*)

lemma *b_fixed2*:

$[\![Crypt\ (priSK\ V)\ (Crypt\ b\ (Crypt\ c\ (Nonce\ Nv))) \in parts\ (spies\ evs);$

$Key\ b \notin parts\ (knows\ Spy\ evs);\ V \neq Adm;\ V \notin bad;\ evs \in foo]\!]$

$\Longrightarrow Crypt\ b\ (Nonce\ Nv') \notin parts\ (spies\ evs)$

apply (*erule rev_mp*, *erule rev_mp*, *erule foo.induct*, *simp_all*)

apply (*case_tac V* $=$ *Spy*, *blast*, *clarsimp*)

apply (*case_tac Key b* $\notin$ *parts* (*knows Spy evsf*), *clarsimp*)

apply (*case_tac Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*))) $\in$ *parts* (*knows Spy evsf*))

apply (*clarsimp*, *blast*, *clarsimp*)

apply (*force dest*!: *Fake_parts_insert priSKV_nosynth2*)

apply (*clarsimp*, *blast dest*: *parts_insertI*)

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

apply (*safe*, *simp_all*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*+)

apply (*blast dest*: *parts_insertI*)

defer **defer**

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*+)

apply (*metis MPair_parts parts_cut_eq spies_partsEs*(2))

prefer *5*

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*+)

apply (*metis parts.Fst parts.Snd parts_cut_eq spies_partsEs*(2))

by (*blast dest*: *parts_insertI CX_analz parts_cut_eq*)+

theorem *Spy_see_b2* [*simp*]:

⟦*Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*))) ∈ *parts* (*spies evs*); *V* ≠ *Adm*;

V ∉ *bad*; *evs* ∈ *foo*⟧

⟹ *Key b* ∉ *parts* (*spies evs*)

apply (*erule rev_mp*, *erule foo.induct*, *simp_all*)

apply (*drule Fake_parts_insert*, *clarsimp*)

apply (*case_tac Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*))) ∈ *parts* (*knows Spy evsf*))

apply (*clarsimp*, *blast*, *clarsimp*)

apply (*blast dest*: *priSKV_nosynth2*)

apply (*metis Crypt_imp_invKey_keysFor invKey_K new_keys_not_used spies_partsEs*(2) *usedI*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*, *clarsimp*)

apply (*rule conjI*, *clarsimp*, *rule conjI*, *clarsimp*, *force*)

apply (*clarsimp*, *metis parts_cut_eq spies_partsEs*(2))

apply (*clarsimp*, *rule conjI*, *force*)

by (*blast dest*: *parts_cut_eq b_fixed2*)+

theorem *b_secrecy2* [*dest*]:

⟦*Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*))) ∈ *parts*(*spies evs*); *V* ≠ *Adm*;

$V \notin bad; evs \in foo\rrbracket \Longrightarrow Key\ b \notin analz\ (spies\ evs)$

by (*blast dest*: *Spy_see_b2*)

lemma *forme*:

$\llbracket Crypt\ (priSK\ V)\ X \in parts\ (spies\ evs); V \neq Adm; evs \in foo\rrbracket$

$\Longrightarrow V \in bad \vee (\exists\ b\ c\ N.\ X = Crypt\ b\ (Crypt\ c\ (Nonce\ N)) \wedge c \in symKeys \wedge b \in symKeys)$

apply (*erule rev_mp*, *erule foo.induct*, *simp_all*)

apply (*force dest*!: *Fake_parts_insert_in_Un priSKV_nosynth*)

defer **apply** *blast* **defer** **apply** *blast* **defer** **apply** *blast*

apply (*case_tac Adm* $\in$ *bad*, *clarsimp*, *rule conjI*)

apply (*clarsimp*, *blast*, *clarsimp*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply *force*

by (*blast dest*: *parts_cut_eq*)+

lemma *EV1_analz_bis*:

$\llbracket Gets\ Adm\ \{\!|Agent\ V, Crypt\ (priSK\ V)\ X|\!\} \in set\ evs; evs \in foo\rrbracket$

$\Longrightarrow X \in analz\ (knows\ Spy\ evs)$

apply (*drule Gets_imp_knows_Spy*, *assumption*)

apply (*case_tac Crypt* $(priSK\ V)\ X \in analz\ (spies\ evs)$)

apply *force*

by (*metis analz.simps*)

lemma *uniq2*:

$\llbracket Gets\ Adm\ \{\!|Agent\ V, Crypt\ (priSK\ V)\ (Crypt\ P\ R)|\!\} \in set\ evs; V \neq Adm; evs \in foo\rrbracket$

$\Longrightarrow (\exists c\ N.\ R = Crypt\ c\ (Nonce\ N) \wedge Key\ P \notin analz\ (knows\ Spy\ evs) \wedge P \in symKeys \wedge c \in symKeys)$

$\vee\ V \in bad$

apply (*erule rev_mp*, *erule foo.induct*, *auto*)

apply (*spy_analz*, *blast dest*: *forme*)

apply (*case_tac* $P = c$)

apply (*blast dest*: *Crypt_imp_keysFor EV1_analz_bis new_keys_not_analzd*)

apply (*case_tac* $P = b$)

apply (*blast dest*: *Crypt_imp_keysFor EV1_analz_bis new_keys_not_analzd*)

apply (*case_tac Key P* ∈ *analz* (*insert* (*Key b*) (*knows Spy evs1*)))

apply (*metis Crypt_imp_keysFor EV1_analz_bis analz_insert_Key insertE new_keys_not_analzd*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply (*case_tac Key P* ∈ *parts* (*insert* (*Key c*) (*insert* (*Key b*) (*knows Spy evs1*))))

prefer *2* **apply** (*blast*, *force*)

apply (*case_tac R* : *analz* (*insert* (*Key* (*invKey Pa*)) (*spies evs2*)))

prefer *2* **apply** (*blast dest*: *Nv_extract3_lemma*)

apply (*metis analz_insert_Agent analz_insert_eq insertCI insertE msg.simps*(*12*))

apply (*blast dest*: *Nv_extract3_lemma analz_insert_eq*)

apply (*blast dest*: *CX_analz analz_cut*)

apply (*case_tac P=c*, *clarsimp*)

by (*blast dest*: *Spy_see_b2 b_fixed2*, *blast dest*: *analz_insert_freshK*)

lemma *b_fixed3*:

⟦*Crypt* (*priSK V*) (*Crypt b R*) ∈ *parts* (*spies evs*);

Key b ∉ *parts* (*knows Spy evs*); *Nonce Nv* ∈ *parts*{*R*}; *V* ≠ *Adm*;

V ∉ *bad*; *evs* ∈ *foo*⟧ ⟹ *Crypt b* (*Nonce Nv'*) ∉ *parts* (*spies evs*)

apply (*erule rev_mp*,*erule rev_mp*, *erule rev_mp*, *erule foo.induct*)

apply (*simp_all*, *case_tac V* = *Spy*, *blast*, *clarsimp*)

apply (*case_tac Key b* ∉ *parts* (*knows Spy evsf*), *clarsimp*)

apply (*case_tac Crypt* (*priSK V*) (*Crypt b R*) ∈ *parts* (*knows Spy evsf*))

apply (*clarsimp*, *blast*, *clarsimp*)

apply (*blast dest*: *Fake_parts_insert priSKV_nosynth2*)

apply (*clarsimp*, *blast dest*: *parts_insertI*)

apply (*case_tac Nonce Nv* ∈ *parts* {*R*}, *clarsimp*)

apply (*case_tac Crypt* (*priSK V*) (*Crypt b R*) ∈ *parts* (*knows Spy evsb*), *clarsimp*)

apply (*force dest*!: *analz_into_parts*)

apply *clarsimp*

apply (*case_tac Nv* ≠ *Nv'*)

apply (*blast dest*: *unique_Nv*, *blast*)

apply (*blast dest*: *new_keys_not_used*)

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

apply (*case_tac Adm* ∈ *bad*, *clarsimp*)

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply (*rule conjI*, *clarsimp*, *rule conjI*, *force*, *clarsimp*)

apply (*metis parts_cut_eq spies_partsEs*(2))

apply (*clarsimp*, *rule conjI*, *force*, *clarsimp*)

apply (*metis parts.Body parts_cut_eq*)

apply (*metis EV2_an parts_cut_eq spies_partsEs*(2))

apply (*blast dest*: *CX_analz parts_cut_eq parts_insertI*)

apply (*clarsimp*, *rule conjI*, *clarsimp*, *rule conjI*, *clarsimp*)

apply (*blast dest*: *CX_analz*)

apply (*metis CX_analz analz_into_parts parts_cut_eq parts_insertI spies_partsEs*(2))+

done

theorem *Spy_see_b3* [*simp*]:

⟦*Crypt* (*priSK V*) (*Crypt b R*) ∈ *parts* (*spies evs*);

Nonce Nv ∈ *parts*{*R*}; *V* ≠ *Adm*; *V* ∉ *bad*; *evs* ∈ *foo*⟧

⟹ *Key b* ∉ *parts* (*spies evs*)

apply (*erule rev_mp*, *erule rev_mp*, *erule foo.induct*, *simp_all*)

apply (*drule Fake_parts_insert*, *clarsimp*)

apply (*blast dest*: *priSKV_nosynth2*, *blast*)

apply (*case_tac Va* ∈ *bad*, *clarsimp*)

apply (*blast dest*: *Crypt_imp_keysFor new_keys_not_used*)

apply (*clarsimp*, *blast*)

apply (*metis EV2_an Gets_imp_parts_knows_Spy parts.Fst parts_cut_eq spies_partsEs*(2))

by (*blast dest*: *parts_cut_eq EV5_msg_in_parts_spies b_fixed3*)+

lemma *b_sym*:

⟦*Crypt* (*priSK V*) (*Crypt b R*) ∈ *parts* (*spies evs*); *V* ≠ *Adm*;

V ∉ *bad*; *evs* ∈ *foo*⟧ ⟹ *b* ∈ *symKeys*

apply (*erule rev_mp*, *erule foo.induct*, *simp_all*)

apply (*blast dest*: *Fake_parts_insert priSKV_nosynth2*, *blast*)

apply (*metis EV2_an Gets_imp_parts_knows_Spy parts.Fst parts_cut_eq spies_partsEs*(2))

by (*blast dest*: *parts_cut_eq*)

lemma *V_unique*:

⟦*Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*))) ∈ *parts* (*spies evs*);

Crypt (*priSK V*) *X* ∈ *parts* (*spies evs*); *Nonce Nv* ∈ *parts* {*X*};

X ≠ *Crypt b* (*Crypt c* (*Nonce Nv*));

V ∉ *bad*; *V* ≠ *Adm*; *evs* ∈ *foo*⟧ ⟹ *False*

apply (*erule rev_mp*, *erule rev_mp*, *erule foo.induct*, *simp_all*)

apply (*force dest*!: *Fake_parts_insert_in_Un analz_into_parts*)

apply (*clarsimp*, *blast*)

apply (*rule conjI*, *force*)

apply (*force dest*!: *parts_cut*)

apply (*metis Gets_imp_parts_knows_Spy MPair_parts parts_cut_eq spies_partsEs*(2))

apply (*metis EV5_msg_in_parts_spies*)

apply (*metis parts.Snd Gets_imp_parts_knows_Spy parts_cut_eq spies_partsEs*(2))

done

lemma *EV1_Key_alternative*:

⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt P* (*Nonce Nv*)))|} ∈ *set evs*;

evs ∈ *foo*⟧

⟹ *V* = *Spy* ∨ (*V* ∉ *bad*) ∨ (*V* ∈ *bad* ∧ *Key P* ∈ *analz* (*knows Spy evs*))

apply (*erule rev_mp*, *erule foo.induct*, *auto*)

by (*blast dest*: *analz_insertI*)+

lemma *analz_insert_Key3* [*simp*]:

K ∉ *keysFor* (*analz H*) ∧ *L* ∉ *keysFor* (*analz H*) ⟹

analz (*insert* (*Key K*) (*insert* (*Key L*) *H*)) = *insert* (*Key K*) (*insert* (*Key L*) (*analz H*))

by *auto*

theorem *Nonce_secrecy*:

evs ∈ *foo* ⟹

(∀ *KK*. *KK* ⊆ − (*range shrK*) ⟶

(∀ *K* ∈ *KK*. *K* ∈ *symKeys* ⟶ ¬ *KeyWithNonce K Nv evs*) ⟶

(*Nonce Nv* ∈ *analz* (*Key*‘*KK* ∪ (*knows Spy evs*))) =

(*Nonce Nv* ∈ *analz* (*knows Spy evs*)))

apply (*erule foo.induct*) **prefer** *8*

apply (*thin_tac CX* = *Crypt P R*) **defer**

apply (*unfold Anms_def*)

apply (*safe del*: *impI intro*!: *Nonce_secrecy_lemma* [*THEN impI*])

prefer *4*

apply (*simp del*: *image_eq_UN add*: *analz_image_freshK_simps*)

apply (*metis analz.Decrypt analz_Decrypt' analz_image_freshK_simps*(*61*) *analz_spies_pubK invKey*)

apply (*simp_all del*: *image_insert image_eq_UN*
add: *analz_image_freshK analz_image_freshK_simps fresh_not_KeyWithNonce*)

apply *spy_analz*

apply *fast* **defer**

apply (*frule EV5_msg_in_parts_spies*)

apply (*metis KeyWithNonceI c_sym invKey_K*)

prefer *2*

apply (*drule CX_analz*, *assumption*)

apply (*metis analz_cut analz_image_freshK_simps*(*61*) *analz_insertI*)

apply (*case_tac Nv=Nva*)

apply (*drule Notes_analz_c*, *assumption*)

apply (*simp del*: *image_insert image_eq_UN add*: *insert_Key_singleton*)

apply (*case_tac Nonce Nv* ∈ *analz*(*spies evs5*))

apply (*metis analz_insertI insert_Key_singleton*)

apply (*case_tac Key c* ∈ *analz*(*spies evs5*))

apply (*blast dest*: *Nonce_analz_cut*)

apply (*drule KeyWithNonceI*, *drule KWN_Nv*)

apply (*simp del*: *image_insert image_eq_UN*)+

apply (*case_tac Adm* ∈ *bad*)

apply (*simp_all del*: *image_insert image_eq_UN add*: *fresh_not_KeyWithNonce*)

apply (*case_tac Key* (*invKey P*) ∈ *analz* (*knows Spy evs2*))

apply (*simp_all del*: *image_insert image_eq_UN add*: *pushing*)

apply (*case_tac R* ∈ *analz* (*knows Spy evs2*))

apply (*simp_all del*: *image_insert image_eq_UN add*: *analz_image_freshK_simps*)

apply (*drule Nv_extract3_lemma*, *assumption*, *metis analz_insert_eq*)

defer

apply (*case_tac Key* (*invKey P*) ∈ *analz* (*knows Spy evs2*))

apply (*simp_all del*: *image_insert image_eq_UN*)

apply (*case_tac R* ∈ *analz* (*knows Spy evs2*))

apply (*simp_all del*: *image_insert image_eq_UN add*: *analz_image_freshK_simps*)

apply (*blast dest*: *EV1_analz_bis*)

apply (*frule_tac EV2_analz*, *assumption*)

apply (*elim disjE exE*)

apply (*simp_all del*: *image_insert image_eq_UN add*: *analz_image_freshK analz_image_freshK_simps*)

defer

apply *safe*

apply (*simp_all del*: *image_insert image_eq_UN*)

apply (*drule Nv_extract*, *assumption*+)

apply (*simp del*: *image_insert image_eq_UN add*: *analz_image_freshK_simps*)

apply (*case_tac* {*c*, *P*} ⊆ − *range shrK*, *blast*, *blast*)

apply (*drule Nv_extract2*, *assumption*+)

apply (*simp_all del*: *image_insert image_eq_UN add*: *analz_image_freshK_simps*)

apply (*case_tac* {*P*} ⊆ − *range shrK*, *blast*, *blast*)

apply (*frule_tac EV2_analz*, *assumption*)

apply (*simp_all del*: *image_insert image_eq_UN*)

apply (*drule Nv_extract3*, *assumption*+)

apply (*simp_all del*: *image_insert image_eq_UN add*: *analz_image_freshK_simps*)

apply (*simp add*: *insert_absorb*)

apply (*drule Nv_extract3*, *assumption*+)

apply (*simp_all del*: *image_insert image_eq_UN add*: *analz_image_freshK_simps*)

by (*auto simp add*: *insert_absorb*)

theorem *single_Nonce_secrecy*:

evs ∈ *foo* ⟹

(¬ *KeyWithNonce c Nv evs* ⟶

c ∉ *range shrK* ⟶

(*Nonce Nv* ∈ *analz* (*insert* (*Key c*) (*knows Spy evs*))) =

(*Nonce Nv* ∈ *analz* (*knows Spy evs*)))

by (*simp del*: *image_insert image_eq_UN*

add: *analz_image_freshK_simps Nonce_secrecy*)

lemma *single_Nonce_secrecy_2*:

⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

Nonce Nv ∉ *analz* (*spies evs*);

V ∉ *bad*; *ca* ∉ *range shrK*; *ca* ≠ *c*; *evs* ∈ *foo*⟧

⟹ (*Nonce Nv* ∈ *analz* (*insert* (*Key ca*) (*knows Spy evs*))) =

(*Nonce Nv* ∈ *analz* (*knows Spy evs*))

by (*blast dest*: *KeyWithNonceI single_Nonce_secrecy KWN_K*)

theorem *Spy_see_Nv* [*simp*]:

⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

Key c ∉ *parts* (*spies evs*); *V* ∉ *bad*; *evs* ∈ *foo*⟧

⟹ *Nonce Nv* ∉ *analz* (*spies evs*)

apply (*erule rev_mp*,*erule rev_mp*,*erule rev_mp*)

apply (*erule foo.induct*, *simp_all*)

apply (*drule Fake_analz_eq*, *blast dest*: *parts_insertI*)

apply (*metis analz.Decrypt analz_spies_pubK invKey invKey_K*)

apply (*blast dest*: *analz_insertI double_fresh_insert*)

apply (*drule EV1_analz_bis*, *assumption*)

apply (*simp add*: *pushing_bis*)

apply (*metis analz.Decrypt analz_insert_eq parts_insertI*)

apply (*clarsimp*, *case_tac c=ca*, *clarsimp*)

apply (*case_tac c* ∈ *symKeys*, *force*)

apply (*blast dest*: *c_sym*, *blast dest*: *unique_c*)

apply (*drule CX_analz*, *assumption*)

apply (*metis analz.Decrypt analz_insert_eq parts_insertI*)

apply (*case_tac Nv=Nva*)

apply (*blast dest*: *EV5_msg_in_parts_spies unique_c*)

apply (*blast dest*: *KWN_K single_Nonce_secrecy KeyWithNonceI*)

apply (*drule Gets_imp_knows_Spy*, *assumption*)

apply (*drule Notes_analz_c*, *assumption*)

apply (*auto simp add*: *insert_absorb*)

apply (*case_tac Key ca* ∈ *analz* (*spies evs6*))

apply (*blast dest*: *analz_cut*)

apply (*case_tac* {|*Number anms*, *Key ca*|} ∈ *analz* (*spies evs6*))

by *force*+

theorem *Nv_secrecy* [*simp*]:

⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

Key c ∉ *analz* (*spies evs*); *V* ∉ *bad*; *evs* ∈ *foo*⟧

⟹ *Nonce Nv* ∉ *analz* (*spies evs*)

apply (*erule rev_mp*,*erule rev_mp*,*erule rev_mp*)

apply (*erule foo.induct*, *simp_all*)

apply (*blast dest*: *Fake_analz_eq analz_insertI*)

apply (*metis analz.Decrypt analz_spies_pubK invKey invKey_K*)

apply (*blast dest*: *analz_insertI double_fresh_insert*)

apply (*drule EV1_analz_bis*, *assumption*)

apply (*simp add*: *pushing_bis*)

apply (*blast dest*: *analz_cut analz_insertI*)

apply (*clarsimp*, *case_tac c=ca*, *clarsimp*)

apply (*case_tac c* ∈ *symKeys*, *force*)

apply (*blast dest*: *c_sym*, *blast dest*: *unique_c*)

apply (*drule CX_analz*, *assumption*)
apply (*metis analz.Decrypt analz_insert_eq*)

apply (*case_tac Nv=Nva*) **apply** *auto*
apply (*blast dest*: *analz_insertI*) **apply** (*blast dest*: *KWN_K KeyWithNonceI*)
apply (*blast dest*: *analz_insertI*) **apply** (*drule KeyWithNonceI*)
apply (*blast dest*: *KWN_Nv analz_insert_eq single_Nonce_secrecy*)

apply (*drule Gets_imp_knows_Spy*, *assumption*)
apply (*drule Notes_analz_c*, *assumption*)
apply (*case_tac* {|*Number anms*, *Key ca*|} ∈ *analz* (*spies evs6*))
by (*blast dest*: *analz_cut*, *force*)

theorem *Nv_secrecy_relaxed* [*simp*]:
⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;
Anms V Col (*Key c*) ∉ *set evs*; *V* ∉ *bad*; *evs* ∈ *foo*⟧
⟹ *Nonce Nv* ∉ *analz* (*spies evs*)
by (*blast dest*: *Nv_secrecy c_secrecy*)

definition

Unique :: [*event*, *event list*] ⇒ *bool* (*Unique* _ *on* _ [*0*, *50*] *50*)
where (*Unique ev on evs*) = (*ev* ∉ *set* (*tl* (*dropWhile* (% *z*. *z* ≠ *ev*) *evs*)))

lemma *Notes_Unique*:
⟦*Notes Adm* (*Agent V*) ∈ *set evs*; *evs* ∈ *foo*⟧
⟹ *Unique* (*Notes Adm* (*Agent V*)) *on evs*
apply (*erule rev_mp*, *erule foo.induct*)
by (*simp_all add*: *Unique_def*, *blast*)

lemma *EV2_Notes*:

⟦*Says Adm V* (*Crypt* (*priSK Adm*) (*Crypt b* (*Crypt c* (*Nonce N*)))) ∈ *set evs*;
evs ∈ *foo*⟧
⟹ *Notes Adm* (*Agent V*) ∈ *set evs*
by (*erule rev_mp*, *erule foo.induct*, *simp_all*)

lemma *unique_fwd*:
⟦*Unique* (*Notes Adm* (*Agent V*)) *on evs*;
Says Adm V (*Crypt* (*priSK Adm*) (*Crypt b* (*Crypt c* (*Nonce N*)))) ∈ *set evs*;
evs ∈ *foo*⟧
⟹ *Unique* (*Says Adm V* (*Crypt* (*priSK Adm*) (*Crypt b* (*Crypt c* (*Nonce N*))))) *on evs*
apply (*erule rev_mp*, *erule rev_mp*, *erule foo.induct*)
by (*simp_all add*: *Unique_def*, *blast dest*: *EV2_Notes*)

lemma *no_two_votes*:
⟦*Says Adm V* (*Crypt* (*priSK Adm*) (*Crypt b* (*Crypt c* (*Nonce N*)))) ∈ *set evs*;
Says Adm V (*Crypt* (*priSK Adm*) (*Crypt d* (*Crypt e* (*Nonce Nv*)))) ∈ *set evs*;
evs ∈ *foo*⟧ ⟹ *Nv* = *N* ∧ *c* = *e* ∧ *b* = *d*
apply (*erule rev_mp*, *erule rev_mp*, *erule foo.induct*)
apply (*simp_all*, *auto*)
by (*blast dest*: *EV2_Notes*)+

lemma *Adm_sign_once*:
⟦*Says Adm V* (*Crypt* (*priSK Adm*) (*Crypt b* (*Crypt c* (*Nonce N*)))) ∈ *set evs*;
evs ∈ *foo*⟧ ⟹
Unique (*Says Adm V* (*Crypt* (*priSK Adm*) (*Crypt b* (*Crypt c* (*Nonce N*))))) *on evs*
by (*blast dest*: *unique_fwd Notes_Unique EV2_Notes*)

theorem *verifiability_lemma*:
⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;
Anms V Col (*Key c*) ∈ *set evs*;
V ∉ *bad*; *evs* ∈ *foo*⟧
⟹ *Gets Col* (*Crypt c* (*Nonce Nv*)) ∈ *set evs*
apply (*erule rev_mp*, *erule rev_mp*, *erule rev_mp*, *erule foo.induct*)

apply (*simp_all*, *spy_analz*, *auto*)

apply (*drule Says_imp_analz_Spy*)

apply *auto*

apply (*case_tac Key c* ∈ *analz* (*spies evs5*), *simp*)

apply (*frule EV5_msg_in_parts_spies*)

by (*blast dest*: *unique_Nv2*)

theorem *verifiability*:

⟦*Anms V Col* (*Crypt* (*priSK Adm*) (*Crypt c* (*Nonce Nv*))) ∈ *set evs*;

Anms V Col (*Key c*) ∈ *set evs*; *V* ≠ *Col*;

V ∉ *bad*; *evs* ∈ *foo*⟧

⟹ *Gets Col* (*Crypt c* (*Nonce Nv*)) ∈ *set evs*

apply (*erule rev_mp*, *erule rev_mp*, *erule rev_mp*, *erule foo.induct*)

apply (*simp_all*, *spy_analz*, *auto*)

apply (*drule verifiability_lemma*)

apply *auto*

apply (*frule EV5_msg_in_parts_spies*)

apply (*frule unique_Nv2*, *assumption*)

apply (*force dest*: *c_secrecy*, *assumption*)

apply *auto*

apply (*case_tac Key c* ∉ *analz* (*spies evs5*))

apply (*blast dest*: *unique_Nv2*, *force*)

done

end

A.3.2 Privacy.thy

theory *Privacy* **imports** *Foo* **begin**

inductive_set

analzplus :: *msg set* ⇒ *msg set* ⇒ *msg set*

for *H* :: *msg set* **and** *ks* :: *msg set*

where

Inj [*intro*,*simp*]: $X \in H \Longrightarrow X \in analzplus\ H\ ks$

| *Fst*: $\{\!|X,Y|\!\} \in analzplus\ H\ ks \Longrightarrow X \in analzplus\ H\ ks$

| *Snd*: $\{\!|X,Y|\!\} \in analzplus\ H\ ks \Longrightarrow Y \in analzplus\ H\ ks$

| *Decrypt* [*dest*]: $[\![Crypt\ K\ X \in analzplus\ H\ ks; Key\ (invKey\ K) \in analzplus\ H\ ks]\!]$
$\Longrightarrow X \in analzplus\ H\ ks$

| *Decrypt2* [*dest*]: $[\![Crypt\ K\ X \in analzplus\ H\ ks; Key\ (invKey\ K) \in ks]\!]$
$\Longrightarrow X \in analzplus\ H\ ks$

lemma *analzplus_mono*: $G \subseteq H \Longrightarrow analzplus\ G\ ks \subseteq analzplus\ H\ ks$

apply *auto*

apply (*erule analzplus.induct*)

apply (*auto dest*: *analzplus.Fst analzplus.Snd*)

done

lemma *MPair_analzplus* [*elim*!]:

$[\![\ \{\!|X,Y|\!\} \in analzplus\ H\ ks;$

$[\![\ X \in analzplus\ H\ ks; Y \in analzplus\ H\ ks\ ]\!] \Longrightarrow P$

$]\!] \Longrightarrow P$

by (*blast dest*: *analzplus.Fst analzplus.Snd*)

lemma *analzplus_increasing*: $H \subseteq analzplus\ H\ ks$

by *blast*

lemma *analz_analzplus*: $analz\ H \subseteq analzplus\ H\ ks$

apply *auto*

apply (*erule analz.induct*)

apply *auto*

done

lemma *analzplus_empty* [*simp*]: $analzplus\ \{\}\ ks = \{\}$

apply *safe*

apply (*erule analzplus.induct*, *blast*+)

done

lemmas *analzplus_insertI* =

subset_insertI [*THEN analzplus_mono*, *THEN* [2] *rev_subsetD*, *standard*]

lemma *analzplus_insert*: *insert X* (*analzplus H ks*) ⊆ *analzplus*(*insert X H*) *ks*

by (*blast intro*: *analzplus_mono* [*THEN* [2] *rev_subsetD*])

lemmas *analzplus_insert_eq_I* = *equalityI* [*OF subsetI analzplus_insert*]

lemma *analzplus_insert_Agent* [*simp*]:

analzplus (*insert* (*Agent agt*) *H*) *ks* = *insert* (*Agent agt*) (*analzplus H ks*)

apply (*rule analzplus_insert_eq_I*)

apply (*erule analzplus.induct*, *auto*)

done

lemma *analzplus_insert_Nonce* [*simp*]:

analzplus (*insert* (*Nonce N*) *H*) *ks* = *insert* (*Nonce N*) (*analzplus H ks*)

apply (*rule analzplus_insert_eq_I*)

apply (*erule analzplus.induct*, *auto*)

done

lemma *analzplus_insert_Number* [*simp*]:

analzplus (*insert* (*Number N*) *H*) *ks* = *insert* (*Number N*) (*analzplus H ks*)

apply (*rule analzplus_insert_eq_I*)

apply (*erule analzplus.induct*, *auto*)

done

lemma *analzplus_insert_Hash* [*simp*]:

analzplus (*insert* (*Hash X*) *H*) *ks* = *insert* (*Hash X*) (*analzplus H ks*)

apply (*rule analzplus_insert_eq_I*)

apply (*erule analzplus.induct*, *auto*)

done

lemma *analzplus_insert_Key* [*simp*]:

$K \notin keysFor\ (analzplus\ H\ ks) \Longrightarrow$

$analzplus\ (insert\ (Key\ K)\ H)\ ks = insert\ (Key\ K)\ (analzplus\ H\ ks)$

apply (*unfold keysFor_def*)

apply (*rule analzplus_insert_eq_I*)

apply (*erule analzplus.induct*, *auto*)

done

lemma *analzplus_insert_MPair* [*simp*]:

$analzplus\ (insert\ \{|X,Y|\}\ H)\ ks =$

$insert\ \{|X,Y|\}\ (analzplus\ (insert\ X\ (insert\ Y\ H))\ ks)$

apply (*rule equalityI*)

apply (*rule subsetI*)

apply (*erule analzplus.induct*, *auto*)

apply (*erule analzplus.induct*)

apply (*blast intro*: *analzplus.Fst analzplus.Snd*)+

done

lemma *analzplus_insert_Crypt*:

$[\![Key\ (invKey\ K) \notin analzplus\ H\ ks; Key\ (invKey\ K) \notin ks]\!]$

$\Longrightarrow analzplus\ (insert\ (Crypt\ K\ X)\ H)\ ks = insert\ (Crypt\ K\ X)\ (analzplus\ H\ ks)$

apply (*rule analzplus_insert_eq_I*)

apply (*erule analzplus.induct*, *auto*)

done

lemma *lemma1_analzplus*: $Key\ (invKey\ K) \in analzplus\ H\ ks \Longrightarrow$

$analzplus\ (insert\ (Crypt\ K\ X)\ H)\ ks \subseteq$

$insert\ (Crypt\ K\ X)\ (analzplus\ (insert\ X\ H)\ ks)$

apply (*rule subsetI*)

apply (*erule_tac x = x* **in** *analzplus.induct*, *auto*)

done

lemma *lemma2_analzplus*: *Key* (*invKey K*) ∈ *analzplus H ks* ⟹
insert (*Crypt K X*) (*analzplus* (*insert X H*) *ks*) ⊆
analzplus (*insert* (*Crypt K X*) *H*) *ks*
apply *auto*
apply (*erule_tac x* = *x* **in** *analzplus.induct*, *auto*)
apply (*blast intro*: *analzplus_insertI analzplus.Decrypt*)
done

lemma *analzplus_insert_Decrypt*:
Key (*invKey K*) ∈ *analzplus H ks* ⟹
analzplus (*insert* (*Crypt K X*) *H*) *ks* =
insert (*Crypt K X*) (*analzplus* (*insert X H*) *ks*)
by (*intro equalityI lemma1_analzplus lemma2_analzplus*)

lemma *lemma1_analzplus2*: *Key* (*invKey K*) ∈ *ks* ⟹
analzplus (*insert* (*Crypt K X*) *H*) *ks* ⊆
insert (*Crypt K X*) (*analzplus* (*insert X H*) *ks*)
apply (*rule subsetI*)
apply (*erule_tac x* = *x* **in** *analzplus.induct*, *auto*)
done

lemma *lemma2_analzplus2*: *Key* (*invKey K*) ∈ *ks* ⟹
insert (*Crypt K X*) (*analzplus* (*insert X H*) *ks*) ⊆
analzplus (*insert* (*Crypt K X*) *H*) *ks*
apply *auto*
apply (*erule_tac x* = *x* **in** *analzplus.induct*, *auto*)
apply (*blast intro*: *analzplus_insertI analzplus.Decrypt2*)
done

lemma *analzplus_insert_Decrypt2*:
Key (*invKey K*) ∈ *ks* ⟹

analzplus (*insert* (*Crypt K X*) *H*) *ks* =

insert (*Crypt K X*) (*analzplus* (*insert X H*) *ks*)

by (*intro equalityI lemma1_analzplus2 lemma2_analzplus2*)

lemma *analzplus_Crypt_if* [*simp*]:

analzplus (*insert* (*Crypt K X*) *H*) *ks* =

(*if* (*Key* (*invKey K*) ∈ *analzplus H ks*) ∨ (*Key* (*invKey K*) ∈ *ks*)

then insert (*Crypt K X*) (*analzplus* (*insert X H*) *ks*)

else insert (*Crypt K X*) (*analzplus H ks*))

by (*simp add*: *analzplus_insert_Crypt*

analzplus_insert_Decrypt analzplus_insert_Decrypt2)

lemma *analzplus_image_Key* [*simp*]: *analzplus* (*Key‘N*) *ks* = *Key‘N*

apply *auto*

apply (*erule analzplus.induct*, *auto*)

done

lemma

parts

{{|*Agent V*, *Nonce Nv*|}}

= {{|*Agent V*, *Nonce Nv*|}, *Agent V*, *Nonce Nv*}

by *auto*

lemma

analz

{{|*Agent V*, *Nonce Nv*|}}

= {{|*Agent V*, *Nonce Nv*|}, *Agent V*, *Nonce Nv*}

by *auto*

lemma

analzplus

{{|*Agent V*, *Nonce Nv*|}}

ks = {{|*Agent V*, *Nonce Nv*|}, *Agent V*, *Nonce Nv*}

by *auto*

lemma

analzplus

$\{\{|Agent\ V, Crypt\ c\ (Nonce\ Nv)|\}\}\ ks =$

$(if\ (Key\ (invKey\ c) \in analzplus\ \{\{|Agent\ V, Crypt\ c\ (Nonce\ Nv)|\}\}\ ks)$

$\vee\ (Key\ (invKey\ c) \in ks)$

then $\{\{|Agent\ V, Crypt\ c\ (Nonce\ Nv)|\},$

$Agent\ V, Crypt\ c\ (Nonce\ Nv), Nonce\ Nv\}$

else $\{\{|Agent\ V, Crypt\ c\ (Nonce\ Nv)|\},$

$Agent\ V, Crypt\ c\ (Nonce\ Nv)\})$

by *auto*

lemma *analz_subset_analzplus*: $analz\ \{X\} \subseteq analzplus\ \{X\}\ ks$

apply (*rule subsetI*) **apply** (*erule analz.induct*, *blast*+)

done

lemma *analzplus_subset_parts*: $analzplus\ \{X\}\ ks \subseteq parts\ \{X\}$

apply (*rule subsetI*) **apply** (*erule analzplus.induct*, *blast*+)

done

lemmas *analzplus_into_parts* = *analzplus_subset_parts* [*THEN subsetD*, *standard*]

lemmas *analz_into_analzplus* = *analz_subset_analzplus* [*THEN subsetD*, *standard*]

primrec *aanalz* :: *agent* => *event list* => *msg set set*

where

aanalz_Nil: $aanalz\ A\ [] = \{\}$

| *aanalz_Cons*:

$aanalz\ A\ (ev\ \#\ evs) =$

(*if* $A = Spy$ *then*

(*case ev of*

Says A′ B X ⇒

(*if A′* ∈ *bad then aanalz Spy evs*

else if isAnms X

then insert ({*Agent B*} ∪ (*analzplus* {*X*} (*analz*(*knows Spy evs*)))) (*aanalz Spy evs*)

else insert ({*Agent B*} ∪ {*Agent A′*} ∪ (*analzplus* {*X*} (*analz*(*knows Spy evs*)))) (*aanalz Spy evs*)

)

| *Gets A′ X* ⇒ *aanalz Spy evs*

| *Notes A′ X* ⇒ *aanalz Spy evs*)

else aanalz A evs)

lemma *aanalz_empty* [*simp*]: *aanalz A* [] = {}

by *simp*

lemma *aanalz_Says* [*simp*]: *aanalz Spy* (*Says A B X* # *evs*) =

(*if A*∈*bad then aanalz Spy evs else*

if isAnms X

then insert ({*Agent B*} ∪ (*analzplus* {*X*} (*analz*(*knows Spy evs*)))) (*aanalz Spy evs*)

else insert ({*Agent B*} ∪ {*Agent A*} *Un* (*analzplus* {*X*} (*analz*(*knows Spy evs*)))) (*aanalz Spy evs*))

by *simp*

lemma *aanalz_Gets* [*simp*]: *aanalz Spy* (*Gets A X* # *evs*) = *aanalz Spy evs*

by *simp*

lemma *aanalz_Notes* [*simp*]: *aanalz Spy* (*Notes A X* # *evs*) =

aanalz Spy evs

by *simp*

lemma *aanalz_priSK*:

⟦*a* ∈ *aanalz Spy evs*; *Crypt* (*priSK A*) *X* ∈ *a*; *X* ∉ *a*; *evs* ∈ *foo*⟧

$\Longrightarrow$ *False*

by (*erule rev_mp*, *erule foo.induct*, *auto simp add*: *isAnms_def*)

inductive_set

asynth :: *msg set set* $\Rightarrow$ *msg set set*

for *as* :: *msg set set*

where

asynth_Build [*intro*]: $[\![a1 \in as; a2 \in as; m \in a1; m \in a2;$
$m \neq Agent\ Adm; m \neq Agent\ Col]\!]$
$\Longrightarrow a1 \cup a2 \in asynth\ as$

lemma *asynth_empty* [*simp*]: *asynth* $\{\} = \{\}$

apply *safe*

apply (*erule asynth.induct*, *simp*+)

done

lemma *asynth_emptyE* [*elim*!]: $X \in asynth\ \{\} \Longrightarrow P$

by *simp*

lemma *asynth_insert*:

$a \in asynth(insert\ a1\ as) \Longrightarrow$

$(a{=}a1 \vee$

$a \in asynth\ as \vee$

$(\exists a2\ m.\ a2 \in as \wedge a = a1 \cup a2 \wedge m \in a1 \wedge m \in a2 \wedge$

$m \neq Agent\ Adm \wedge m \neq Agent\ Col))$

by (*erule asynth.induct*, *blast*)

lemma *not_says*:

$[\![Says\ A\ Adm\ X \in set\ evs; A = Col \vee A = Adm; evs \in foo]\!] \Longrightarrow False$

by (*erule rev_mp*, *erule foo.induct*, *auto*)

lemma *no_pairs_nonces*:

$[\![Nonce\ N \in parts\ \{R\}; N \neq Nva; \forall A\ B.\ \{\!|A, B|\!\} \notin parts\ \{R\};$

$Crypt\ b\ (Crypt\ c\ (Nonce\ Nva)) \in parts\ \{R\} \vee$

$Crypt\ (priSK\ V)\ (Crypt\ b\ (Crypt\ c\ (Nonce\ Nva))) \in parts\ \{R\}]\!]$

$\Longrightarrow False$

by $(induct\ R, auto)$

lemma *no_pairs*:

$[\![Agent\ Va \in parts\{R\}; \forall A\ B.\ \{\!|A, B|\!\} \notin parts\ \{R\};$

$Crypt\ (priSK\ V)\ (Crypt\ ca\ (Nonce\ Nv)) \in parts\ \{R\} \vee$

$Crypt\ (priSK\ V)\ (Crypt\ b\ (Crypt\ c\ (Nonce\ Nv))) \in parts\ \{R\} \vee$

$Crypt\ b\ (Crypt\ c\ (Nonce\ Nv)) \in parts\ \{R\} \vee$

$Number\ N \in parts\ \{R\} \vee$

$Nonce\ Nv\ \in parts\ \{R\} \vee$

$(Agent\ T \in parts\ \{R\} \wedge T\ \neq Va) \vee$

$Crypt\ ca\ (Nonce\ Nv) \in parts\{R\}]\!] \Longrightarrow False$

by $(induct\ R, auto)$

lemma *analzplus_Nv*:

$[\![Nonce\ Nv \in analzplus\ \{Q\}\ (analz\ H);$

$Key\ (invKey\ P) \notin analz\ (insert\ Q\ H);$

$Crypt\ P\ R \in analzplus\ \{Q\}\ (analz\ H);$

$\forall X\ Y.\ \{\!|X, Y|\!\} \notin parts\ \{Q\}]\!]$

$\Longrightarrow False$

apply $(induct\ Q, auto, case\_tac\ Key\ (invKey\ nat) \in analz\ H)$

by $(force\ intro: analz\_insertI)+$

lemma *analzplus_embed*:

$[\![Agent\ V \in analzplus\ \{Q\}\ (analz\ H);$

$Nonce\ Nv \in parts\ \{R\};$

$Crypt\ (priSK\ Adm)\ (Crypt\ P\ R) \in analzplus\ \{Q\}\ (analz\ H) \vee$

$Crypt\ P\ R \in analzplus\ \{Q\}\ (analz\ H);$

$\forall X\ Y.\ \{|X, Y|\} \notin parts\ \{Q\}]$

$\Longrightarrow False$

apply $(induct\ Q, auto)$

apply $(case\_tac\ Key\ (invKey\ nat) \in analz\ H)$

apply $(force\ intro: analz\_insertI, force\ intro: analz\_insertI)$

apply $(case\_tac\ Key\ (invKey\ nat) \in analz\ H, clarsimp)$

apply $(blast\ dest: analzplus\_into\_parts\ no\_pairs, force)$

done

lemma *aanalz_Col_Adm*:

$[\![a \in aanalz\ Spy\ evs; Agent\ Col \notin a; Agent\ Adm \notin a; evs \in foo]\!] \Longrightarrow False$

by $(erule\ rev\_mp, erule\ foo.induct, auto\ simp\ add: isAnms\_def)$

lemma *aanalz_traffic*:

$[\![a \in aanalz\ Spy\ evs; X \notin parts\ (spies\ evs); X \in a; \forall\ A.\ X \neq Agent\ A; evs \in foo]\!]$

$\Longrightarrow False$

apply $(erule\ rev\_mp, erule\ rev\_mp, erule\ foo.induct, simp\_all\ add: isAnms\_def)$

apply $(blast\ dest: analzplus\_into\_parts\ parts\_cut\ parts\_insertI)+$

done

theorem *foo_V_privacy_aanalz*:

$[\![Says\ V\ Adm\ \{|Agent\ V, Crypt\ (priSK\ V)\ (Crypt\ b\ (Crypt\ c\ (Nonce\ Nv)))|\} \in set\ evs;$

$a \in (aanalz\ Spy\ evs); Nonce\ Nv \in a; V \notin bad; evs \in foo]\!]$

$\Longrightarrow Agent\ V \notin a$

apply $(erule\ rev\_mp, erule\ rev\_mp, erule\ foo.induct, simp\_all\ add: isAnms\_def)$

apply *force*

apply $(blast\ dest: aanalz\_traffic)$

apply *auto*

apply $(blast\ dest: not\_says\ analzplus\_into\_parts\ no\_pairs)$

apply $(drule\ Gets\_imp\_parts\_knows\_Spy, assumption)$

apply $(drule\ analzplus\_into\_parts)$

apply $(case\_tac\ P \in symKeys)$

apply $(force\ dest!: Spy\_see\_b3)$

apply (*blast dest*: *b_sym*)

apply (*blast dest*: *not_says analzplus_into_parts no_pairs*)+

done

theorem *foo_privacy_aanalz*:

⟦*a* ∈ (*aanalz Spy evs*); *Nonce Nv* ∈ *a*; *evs* ∈ *foo*⟧

⟹ (*V* ∉ *bad* ∧ *V* ≠ *Col* ∧ *V* ≠ *Adm* ⟶ *Agent V* ∉ *a*)

apply (*erule rev_mp*, *erule foo.induct*, *simp_all add*: *isAnms_def*)

apply *force*

apply *auto*

apply (*drule Gets_imp_parts_knows_Spy*, *assumption*)

apply (*drule analzplus_into_parts*)

apply (*case_tac P* ∈ *symKeys*)

apply *force*

apply (*blast dest*: *b_sym*)

by (*blast dest*: *not_says analzplus_into_parts no_pairs*)+

lemma *nv_fresh_a2*:

⟦*Nonce Nv* ∉ *used evs*; *a* ∈ *asynth*(*aanalz Spy evs*); *evs* ∈ *foo*⟧ ⟹ *Nonce Nv* ∉ *a*

apply (*erule rev_mp*, *erule rev_mp*, *erule foo.induct*, *simp_all add*: *isAnms_def*)

apply (*blast dest*: *analzplus_into_parts asynth_insert aanalz_traffic*)+

done

lemma *association_Nv*:

⟦*a* ∈ *aanalz Spy evs*; *Nonce Nv* ∈ *a*; *m* ∈ *a*; *Nv* ≠ *Nva*;

m ∈ {⦃*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nva*)))⦄,

Crypt (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nva*))),

Crypt b (*Crypt c* (*Nonce Nva*))}; *evs* ∈ *foo*⟧

⟹ *False*

apply (*erule rev_mp*, *erule foo.induct*, *simp_all add*: *isAnms_def*)

apply *blast*

apply *clarsimp*

apply (*rule conjI*)

apply *clarsimp*

apply (*erule disjE*, *clarsimp*)

apply (*erule disjE*, *clarsimp*)

apply (*case_tac Key* (*invKey c*) ∈ *analz* (*knows Spy evs2*))

apply *clarsimp* **apply** *clarsimp*

apply *clarsimp*

apply (*erule disjE*, *clarsimp*)

apply (*blast dest*: *analzplus_into_parts no_pairs_nonces*)

apply (*erule disjE*, *clarsimp*)

apply (*erule disjE*, *clarsimp*)

apply (*erule disjE*, *clarsimp*)

apply (*case_tac Key* (*invKey b*) ∈ *analz* (*knows Spy evs2*))

apply *clarsimp*

apply (*case_tac Key* (*invKey c*) ∈ *analz* (*knows Spy evs2*))

apply *clarsimp* **apply** *clarsimp* **apply** *clarsimp*

apply (*blast dest*: *analzplus_into_parts no_pairs_nonces*)

apply (*erule disjE*, *clarsimp*)

apply (*erule disjE*, *clarsimp*)

apply (*case_tac Key* (*invKey c*) ∈ *analz* (*knows Spy evs2*))

apply *clarsimp* **apply** *clarsimp*

apply (*blast dest*: *analzplus_into_parts no_pairs_nonces*)

apply (*blast dest*: *analzplus_into_parts no_pairs_nonces*)

apply *blast*

apply (*rule conjI*, *clarsimp*)

apply (*erule disjE*, *clarsimp*)

apply (*erule disjE*, *clarsimp*)

apply (*blast dest*: *analzplus_into_parts no_pairs*)

apply (*erule disjE*, *clarsimp*)

apply (*erule disjE*, *clarsimp*)

apply (*case_tac Key* (*invKey b*) ∈ *analz* (*knows Spy evs4*))

apply *clarsimp*

apply (*case_tac Key* (*invKey c*) ∈ *analz* (*knows Spy evs4*))

apply *clarsimp* **apply** *clarsimp*

apply *clarsimp*

apply *clarsimp*

apply (*case_tac Key* (*invKey c*) ∈ *analz* (*knows Spy evs4*))

apply *clarsimp* **apply** *clarsimp*

apply (*blast dest*: *analzplus_into_parts no_pairs_nonces*)

apply (*blast dest*: *analzplus_into_parts no_pairs_nonces*)

apply *blast*+

done

lemma *association_Nv_2*:

⟦*a* ∈ *aanalz Spy evs*; *Agent V* ∈ *a*; *m* ∈ *a*;

V ≠ *Col*; *V* ≠ *Adm*; *V* ∉ *bad*;

m ∈ {|*Number anms*, *Crypt* (*priSK Adm*) (*Crypt c* (*Nonce Nv*))|}, *Number anms*,

Crypt (*priSK Adm*) (*Crypt c* (*Nonce Nv*)), *Crypt c* (*Nonce Nv*)};

evs ∈ *foo*⟧ ⟹ *False*

apply (*erule rev_mp*, *erule foo.induct*, *simp_all add*: *isAnms_def*)

apply (*blast*, *erule disjE*, *case_tac Key* (*invKey P*) ∈ *analz* (*knows Spy evs2*))

apply *clarsimp*

apply (*erule disjE*)

apply *clarsimp*

apply (*force dest*!: *uniq2*)

apply (*blast dest*: *analzplus_into_parts no_pairs*)

apply (*clarsimp*, *rule conjI*, *clarsimp*, *erule disjE*)

apply (*force dest*!: *uniq2*)

apply (*blast dest*: *analzplus_into_parts no_pairs*)

apply (*force dest*!: *uniq2*)

apply (*blast dest*: *analzplus_into_parts no_pairs*)+

done

lemma *Double_crypt_used*:

⟦*a* ∈ *aanalz Spy evs*; *Key P* ∉ *used evs*; *P* ∈ *symKeys*;

Crypt (priSK Adm) (Crypt P R) ∈ a; evs ∈ foo⟧

⟹ False

by (*blast dest*: *Crypt_imp_keysFor new_keys_not_used aanalz_traffic*)

lemma *m_crypt1*:

⟦*Nonce Nv ∈ parts {R}*; *∀ X Y. {|X, Y|} ∉ parts {R}*;

(m ∈ parts {R} ∧ (∀ A B. m ≠ Crypt A B) ∧ m ≠ Nonce Nv) ∨

(Crypt A B ∈ parts {R} ∧ Nonce Nv ∉ parts {B})⟧

⟹ False

by (*induct R*, *auto*)

lemma *m_crypt2*:

⟦*Agent V ∈ parts {R}*; *∀ X Y. {|X, Y|} ∉ parts {R}*;

(m ∈ parts {R} ∧ m ≠ Agent V ∧ (∀ A B. m ≠ Crypt A B)) ∨

(Crypt A B ∈ parts {R} ∧ Agent V ∉ parts {B})⟧

⟹ False

by (*induct R*, *auto*)

lemma *aanalz_Adm*:

⟦*Says V Adm {|Agent V*, *Crypt (priSK V) (Crypt b (Crypt c (Nonce Nv)))|} ∈ set evs*;

a ∈ aanalz Spy evs; *Agent Col ∉ a*; *Agent V ∈ a*; *V ∉ bad*;

Crypt P R ∈ a; *Nonce Nv ∈ parts {R}*; *evs ∈ foo⟧*

⟹ (P = priSK V ∨ P = priSK Adm ∨ P = b) ∧

(P ≠ priSK Adm ∨ R = Crypt b (Crypt c (Nonce Nv)))

apply (*erule rev_mp*,*erule rev_mp*, *erule foo.induct*, *simp_all add*: *isAnms_def*)

apply *spy_analz*

apply (*case_tac Key ca ∈ analz (knows Spy evs1)*, *clarsimp*)

apply (*blast dest*: *new_keys_not_used*)

apply (*case_tac Key ba ∈ analz (knows Spy evs1)*, *clarsimp*)

apply (*blast dest*: *new_keys_not_used*)

apply *clarsimp*

apply (*rule conjI*)

apply *force*

apply *clarsimp*

apply (*rule conjI*)

apply (*force dest*: *EV5_msg_in_parts_spies not_says*)

apply (*blast dest*: *aanalz_traffic parts_cut*)

apply (*rule conjI*)

apply *clarsimp*

apply (*erule disjE*)

apply (*blast dest*: *not_says*)

apply (*erule disjE*)

apply *clarsimp*

apply (*erule disjE*)

apply (*force dest*!: *uniq2*)

apply (*blast dest*: *analzplus_into_parts m_crypt2*)

apply (*erule disjE*)

apply (*force dest*!: *uniq2*)

apply (*erule disjE*)

apply (*blast dest*: *analzplus_into_parts m_crypt2*)

apply (*blast dest*: *analzplus_embed*)

apply *clarsimp*

apply (*erule disjE*)

apply (*blast dest*: *not_says*)

apply (*erule disjE*, *clarsimp*)

apply (*case_tac Crypt* (*priSK V*) (*Crypt P R*) ∈ *parts* (*spies evs2*) ∧
Crypt (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*))) ∈ *parts* (*spies evs2*))

apply (*force dest*!: *V_unique*)

apply *blast*

apply (*rule conjI*)

apply (*case_tac Crypt* (*priSK V*) (*Crypt P R*) ∈ *parts* (*spies evs2*) ∧
Crypt (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*))) ∈ *parts* (*spies evs2*))

apply (*force dest*!: *V_unique*)

apply *blast*

apply (*case_tac Crypt* (*priSK V*) (*Crypt P R*) ∈ *parts* (*spies evs2*) ∧
Crypt (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*))) ∈ *parts* (*spies evs2*))

apply (*force dest*!: *V_unique*)

apply *blast*+

done

lemma *aanalz_EV1_AdmPR*:

⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt P* (*Nonce Nva*)))|} ∈ *set evs*;

a ∈ *aanalz Spy evs*; *Crypt* (*priSK Adm*) (*Crypt P R*) ∈ *a*; *Va* ≠ *Adm*; *Va* ≠ *Col*;

Agent Va ∈ *a*; *V* ∉ *bad*; *Nonce Nv* ∈ *parts* {*R*};

evs ∈ *foo*; *Va* ∉ *bad*⟧ ⟹ *False*

apply (*erule rev_mp*, *erule rev_mp*, *erule foo.induct*, *simp_all add*: *isAnms_def*)

apply *blast*

apply (*force dest*!: *Double_crypt_used*)

apply (*case_tac Key* (*invKey Pa*) ∈ *analz* (*knows Spy evs2*), *clarsimp*)

apply (*erule disjE*, *clarsimp*, *erule disjE*, *clarsimp*)

apply (*force dest*!: *uniq2*)

apply (*blast dest*: *analzplus_into_parts no_pairs*)

apply (*erule disjE*, *clarsimp*, *erule disjE*, *clarsimp*)

apply (*force dest*!: *uniq2*, *force dest*!: *uniq2*)

apply (*erule disjE*, *clarsimp*)

apply (*case_tac Key*(*invKey P*) ∈ *analz* (*knows Spy evs2*))

apply (*force dest*!: *analzplus_into_parts no_pairs*)

apply (*force dest*: *analzplus_into_parts no_pairs*)

apply (*blast dest*: *analzplus_embed*)

apply (*blast dest*: *uniq2 Spy_see_b2 b_fixed2*)

apply *blast*

apply (*blast dest*: *analzplus_embed*)

apply *blast*+

done

lemma *aanalz_PR*:

⟦*a* ∈ *aanalz Spy evs*; *Crypt P R* ∈ *a*; *evs* ∈ *foo*⟧ ⟹

(*Agent Col* ∉ *a* ∨

(*Agent V* ∈ *a* ⟶ *V* ∈ *bad* ∨ *V* = *Col*) ∨

$(Nonce\ Nv \notin parts\ \{R\})) \wedge$

$((Nonce\ Nv \notin a) \vee$

$(Key\ (invKey\ P) \in analz\ (spies\ evs) \wedge Agent\ V \notin parts\ \{R\}))$

apply $(erule\ rev\_mp, erule\ foo.induct, simp\_all\ add: isAnms\_def)$

apply $(blast\ dest: analz\_insertI)$

apply $(blast\ dest: analz\_insertI)$

apply *clarsimp*

apply $(rule\ conjI)$

apply $(blast\ dest: analz\_insertI)$

apply *clarsimp*

apply $(rule\ conjI)$

apply *clarsimp*

apply $(erule\ disjE)$

apply *clarsimp*

apply $(blast\ dest: analzplus\_into\_parts\ no\_pairs)$

apply $(erule\ disjE)$

apply $(blast\ dest: analzplus\_into\_parts\ no\_pairs\ analz\_insertI)$

apply $(rule\ conjI)$

apply *clarsimp*

apply $(blast\ dest: analzplus\_into\_parts\ no\_pairs\ parts\_cut)$

apply $(rule\ conjI)$

apply *clarsimp*

apply $(blast\ dest: analzplus\_into\_parts\ no\_pairs\ parts\_cut)$

apply $(rule\ conjI)$

apply $(blast\ dest: analzplus\_into\_parts\ no\_pairs)$

apply $(case\_tac\ Key\ (invKey\ P) \in analz\ (insert\ Ra\ (knows\ Spy\ evs2)))$

apply *clarsimp*

apply $(blast\ dest: analzplus\_into\_parts\ no\_pairs\ parts\_cut)$

apply $(blast\ dest: analzplus\_Nv)$

apply $(blast\ dest: analz\_insertI)$

apply *force*

apply *clarsimp*

apply (*rule conjI*)

apply *clarsimp*

apply (*blast dest*: *analz_insertI*)

apply *clarsimp*

apply (*case_tac a* ∈ *aanalz Spy evs4*)

apply *clarsimp*

apply (*blast dest*: *analz_insertI*)

apply *clarsimp*

apply (*erule disjE*)

apply (*blast dest*: *analzplus_into_parts no_pairs analz_insertI*)

apply (*rule conjI*)

apply *clarsimp*

apply (*blast dest*: *analzplus_into_parts no_pairs parts_cut*)

apply *clarsimp*

apply (*rule conjI*)

apply (*blast dest*: *analzplus_Nv*)

apply *clarsimp*

apply (*blast dest*: *analzplus_into_parts no_pairs parts_cut*)

apply (*blast dest*: *analz_insertI*)

apply *clarsimp*

done

lemma *aanalz_AdmPR_V_Nparts*:

⟦*a* ∈ *aanalz Spy evs*; *Crypt* (*priSK Adm*) (*Crypt P R*) ∈ *a*; *evs* ∈ *foo*⟧

⟹ *Nonce Nv* ∉ *parts* {*R*} ∨

Key (*invKey P*) ∉ *analz* (*knows Spy evs*) ∨

(*Agent V* ∈ *a* ⟶ *V* ∈ *bad* ∨ *V* = *Adm* ∨ *V* = *Col*)

apply (*erule rev_mp*, *erule foo.induct*, *simp_all add*: *isAnms_def*)

apply *spy_analz*

apply (*case_tac Key c* ∈ *analz* (*knows Spy evs1*) ∨ *Key b* ∈ *analz* (*knows Spy evs1*))

apply (*blast dest*: *new_keys_not_used*, *clarsimp*)

apply *safe*

apply *auto* **defer**

apply (*frule EV1_analz_bis*, *assumption*)

apply (*drule Gets_imp_knows_Spy*, *assumption*)

apply (*case_tac Crypt* (*priSK V*) (*Crypt P R*) ∈ *parts* (*spies evs2*))

apply (*case_tac P* ∈ *symKeys*)

apply (*force dest*!: *Spy_see_b3*)

apply (*blast dest*: *b_sym*)

apply (*blast dest*: *parts.Inj*)

apply (*blast dest*: *analzplus_into_parts no_pairs*)

apply (*case_tac Key* (*invKey P*) ∈ *analz* (*knows Spy evs2*), *clarsimp*)

apply (*force dest*!: *uniq2*)

apply *clarsimp*

apply (*case_tac Key* (*invKey P*) ∈ *analz* (*knows Spy evs2*), *clarsimp*)

apply (*blast dest*: *analzplus_into_parts no_pairs*)

apply *clarsimp*

apply (*force dest*!: *uniq2*)

apply (*blast dest*: *analzplus_embed*)

apply (*case_tac Key* (*invKey P*) ∈ *analz* (*insert Ra* (*knows Spy evs2*)))

apply (*blast dest*: *Nv_extract3_lemma analz_cut*)

apply (*case_tac Agent Va* ∈ *analz* (*spies evs2*))

apply (*case_tac Ra* ∈ *analz* (*insert* (*Key* (*invKey Pa*)) (*spies evs2*)))

apply (*simp add*: *analz_insert_eq*)

apply (*blast dest*: *Nv_extract3_lemma*)

apply (*blast dest*: *Gets_imp_Says Says_imp_analz_Spy*)

apply (*force dest*!: *uniq2*)

apply (*blast dest*: *analzplus_embed*)

apply (*blast dest*: *Nv_extract3_lemma analz_cut*)

apply (*case_tac Key* (*invKey P*) ∈ *analz* (*knows Spy evs4*), *clarsimp*)

apply (*blast dest*: *analzplus_into_parts no_pairs*)

apply *clarsimp*

apply (*blast dest*: *analzplus_embed*)

apply (*blast dest*: *CX_analz analz_cut*)

apply (*blast dest*: *analzplus_embed*)

apply (*blast dest*: *CX_analz analz_cut*)

apply (*case_tac P=c*, *clarsimp*)

apply (*blast dest*: *EV1_Key_alternative aanalz_EV1_AdmPR*)

apply (*case_tac Key c* $\notin$ *used* (*evs5*))

apply (*blast dest*: *KeyWithNonceI fresh_not_KeyWithNonce*)

apply (*simp add*: *analz_insert_freshK*, *force*)

apply (*case_tac P=c*)

apply (*blast dest*: *Double_crypt_used*)

apply (*case_tac P=b*)

apply (*blast dest*: *Double_crypt_used*)

apply (*simp add*: *new_keys_not_analzd*)

apply *force*

done

lemma *aanalz_Vabad_V_imp_Notes*:

⟦*a* ∈ *aanalz Spy evs*; *Va* ≠ *Adm*; *V* ≠ *Adm*; *V* ≠ *Col*; *V* ∉ *bad*;

Agent Va ∈ *a*; *Agent V* ∈ *a*; *Va* ∈ *bad*; *evs* ∈ *foo*⟧

⟹ *Notes Adm* (*Agent Va*) ∈ *set evs*

apply (*erule rev_mp*, *erule foo.induct*, *simp_all add*: *isAnms_def*)

apply (*blast dest*: *uniq2 analzplus_into_parts no_pairs*)+

done

lemma *aanalz_V_Nv_imp_Notes*:

⟦*a* ∈ *aanalz Spy evs*; *V* ≠ *Col*; *V* ≠ *Adm*; *Agent V* ∈ *a*; *Nonce Nv* ∈ *a*; *evs* ∈ *foo*⟧

⟹ *Notes Adm* (*Agent V*) ∈ *set evs*

apply (*erule rev_mp*, *erule foo.induct*, *simp_all add*: *isAnms_def*)

apply (*blast dest*: *uniq2 analzplus_into_parts no_pairs*)+

done

Main privacy theorem

theorem *foo_V_privacy_asynth*:

⟦*Says V Adm* {|*Agent V*, *Crypt* (*priSK V*) (*Crypt b* (*Crypt c* (*Nonce Nv*)))|} ∈ *set evs*;

$a \in (\mathit{asynth}\ (\mathit{aanalz}\ \mathit{Spy}\ \mathit{evs}))$;

$\mathit{Nonce}\ \mathit{Nv} \in a; V \notin \mathit{bad}; V \neq \mathit{Adm}; V \neq \mathit{Col}; \mathit{evs} \in \mathit{foo}]\!]$

$\Longrightarrow \mathit{Agent}\ V \notin a$

apply $(\mathit{erule}\ \mathit{rev\_mp}, \mathit{erule}\ \mathit{rev\_mp}, \mathit{erule}\ \mathit{foo.induct}, \mathit{simp\_all}\ \mathit{add}\colon \mathit{isAnms\_def})$

apply blast

apply $(\mathit{case\_tac}\ \mathit{Key}\ \mathit{ca} \in \mathit{analz}\ (\mathit{knows}\ \mathit{Spy}\ \mathit{evs1}) \vee \mathit{Key}\ \mathit{ba} \in \mathit{analz}\ (\mathit{knows}\ \mathit{Spy}\ \mathit{evs1}))$

apply blast

apply $\mathit{clarsimp}$

apply $(\mathit{case\_tac}\ \mathit{Va} \in \mathit{bad})$

apply $\mathit{clarsimp}$

apply $\mathit{clarsimp}$

apply $(\mathit{drule}\ \mathit{asynth\_insert}, \mathit{erule}\ \mathit{disjE})$

apply simp

apply $(\mathit{erule}\ \mathit{disjE})$

apply $(\mathit{blast}\ \mathit{dest}\colon \mathit{nv\_fresh\_a2})$

apply $(\mathit{case\_tac}\ V{\neq}\mathit{Va})$

apply $(\mathit{blast}\ \mathit{dest}\colon \mathit{foo\_V\_privacy\_aanalz})$

apply $(\mathit{case\_tac}\ \mathit{Nv}{=}\mathit{Nva})$

apply $(\mathit{blast}\ \mathit{dest}\colon \mathit{aanalz\_traffic})$

apply $(\mathit{blast}\ \mathit{dest}\colon \mathit{association\_Nv}\ \mathit{foo\_V\_privacy\_aanalz})$

apply $(\mathit{case\_tac}\ b \in \mathit{symKeys})$

apply $(\mathit{case\_tac}\ a \in \mathit{asynth}\ (\mathit{aanalz}\ \mathit{Spy}\ \mathit{evs2}), \mathit{simp})$

apply $\mathit{clarsimp}$

apply $(\mathit{case\_tac}\ \mathit{Key}\ (\mathit{invKey}\ P) \in \mathit{analz}\ (\mathit{knows}\ \mathit{Spy}\ \mathit{evs2}))$

apply $(\mathit{case\_tac}\ \mathit{Va} \notin \mathit{bad} \vee \mathit{Va}{=}V)$

apply $(\mathit{force}\ \mathit{dest}!\colon \mathit{uniq2})$

apply $\mathit{clarsimp}$

apply $(\mathit{drule}\ \mathit{asynth\_insert}, \mathit{erule}\ \mathit{disjE})$

apply $(\mathit{blast}\ \mathit{dest}\colon \mathit{analzplus\_into\_parts}\ \mathit{no\_pairs})$

apply $(\mathit{erule}\ \mathit{disjE}, \mathit{blast})$

apply *clarsimp*

apply (*erule disjE*)

apply (*erule disjE*)

apply (*blast dest*: *analzplus_into_parts no_pairs*)

apply (*erule disjE*)

apply (*blast dest*: *analzplus_into_parts aanalz_AdmPR_V_Nparts*)

apply (*erule disjE*)

apply *clarsimp*

apply (*simp add*: *aanalz_Vabad_V_imp_Notes*)

apply (*erule disjE*)

apply *clarsimp*

apply (*case_tac Agent Col* $\in$ *a2*)

apply (*blast dest*: *aanalz_PR analzplus_into_parts*)

apply (*case_tac P* $\neq$ *priSK V*)

apply (*case_tac P* $=$ *b*, *clarsimp*)

apply *force*

apply (*case_tac P* $=$ *priSK Adm*)

apply (*case_tac R* $=$ *Crypt b* (*Crypt c* (*Nonce Nv*)), *clarsimp*)

apply (*case_tac Key* (*invKey b*) $\in$ *analz* (*knows Spy evs2*), *clarsimp*)

apply *force*

apply *force*

apply (*case_tac* $\exists$ *A B*. *R* $=$ *Crypt A B*, *clarsimp*)

apply (*case_tac Key* (*invKey A*) $\in$ *analz* (*knows Spy evs2*), *clarsimp*)

apply (*blast dest*: *analzplus_into_parts aanalz_AdmPR_V_Nparts*)

apply (*force dest*: *analzplus_into_parts aanalz_AdmPR_V_Nparts*)

apply (*blast dest*: *aanalz_Adm analzplus_into_parts*)

apply (*blast dest*: *analzplus_into_parts aanalz_Adm*)

apply *clarsimp*

apply (*case_tac R* $=$ *Crypt b* (*Crypt c* (*Nonce Nv*)), *clarsimp*)

apply (*case_tac Key* (*invKey b*) $\in$ *analz* (*knows Spy evs2*), *clarsimp*)

apply *force*

apply *clarsimp*

apply (*blast dest*: *analzplus_into_parts V_unique*)

apply (*case_tac* ∃ *A B*. *m* = *Crypt A B*, *clarsimp*)

apply (*case_tac Nonce Nv* ∈ *parts* {*B*})

apply (*case_tac Agent Col* ∈ *a2*)

apply (*blast dest*: *analzplus_into_parts aanalz_PR*)

apply (*case_tac A* = *priSK V*, *clarsimp*)

apply (*case_tac B* = *Crypt b* (*Crypt c* (*Nonce Nv*)), *clarsimp*)

apply (*drule EV1_analz_bis*, *assumption*)

apply (*case_tac Crypt b* (*Crypt c* (*Nonce Nv*)) ∈ *analzplus* {*R*} (*analz* (*knows Spy evs2*)))

apply (*case_tac Key* (*invKey b*) ∉ *analz* (*insert R* (*spies evs2*)))

apply (*blast dest*: *analzplus_Nv*)

apply *force*

apply (*drule analzplus.Decrypt2*, *force*, *simp*)

apply (*blast dest*: *V_unique aanalz_traffic*)

apply (*case_tac A* = *b*, *clarsimp*)

apply (*case_tac R* ∈ *analz* (*insert* (*Key* (*invKey P*)) (*spies evs2*)))

apply (*case_tac Key* (*invKey b*) ∉ *analz* (*insert R* (*spies evs2*)))

apply (*blast dest*: *analzplus_Nv*)

apply *clarsimp*

apply (*blast dest*: *analz_cut*)

apply (*blast dest*: *Nv_extract3_lemma*)

apply (*case_tac A* = *priSK Adm*)

apply (*case_tac B* = *Crypt b* (*Crypt c* (*Nonce Nv*)))

apply (*clarsimp*, *drule EV1_analz_bis*, *assumption*)

apply (*case_tac Key* (*invKey b*) ∉ *analz* (*insert R* (*spies evs2*)))

apply (*blast dest*: *analz_cut analzplus_Nv*)

apply (*force dest*: *analz_insert_eq*)

apply (*force dest*!: *aanalz_Adm*)

apply (*force dest*!: *aanalz_Adm*)

apply (*blast dest*: *analzplus_into_parts m_crypt1*)

apply *clarsimp*

apply (*blast dest*: *foo_V_privacy_aanalz analzplus_into_parts m_crypt1*)

apply (*erule disjE*)

apply (*erule disjE*)

apply (*blast dest*: *aanalz_PR aanalz_priSK analzplus_into_parts*)

apply (*erule disjE*)

apply (*simp add*: *aanalz_V_Nv_imp_Notes*)

apply (*erule disjE*)

apply (*blast dest*: *aanalz_PR analzplus_into_parts*)

apply (*case_tac* $\exists$ *A B*. *m* = *Crypt A B*, *clarsimp*)

apply (*blast dest*: *aanalz_PR analzplus_into_parts m_crypt2*)

apply *clarsimp*

apply (*blast dest*: *foo_V_privacy_aanalz analzplus_into_parts m_crypt2*)

apply (*blast dest*: *foo_V_privacy_aanalz*)

apply *clarsimp*

apply (*drule asynth_insert*)

apply (*blast dest*: *aanalz_priSK aanalz_PR foo_V_privacy_aanalz*)

apply (*blast dest*: *b_sym*)

apply (*case_tac Key* (*invKey ca*) $\notin$ *analz* (*knows Spy evs3*), *clarsimp*)

apply (*blast dest*: *asynth_insert foo_V_privacy_aanalz*)

apply (*clarsimp*, *drule asynth_insert*)

apply (*blast dest*: *association_Nv_2 foo_V_privacy_aanalz*)

apply (*case_tac b* $\in$ *symKeys*)

apply (*case_tac Key* (*invKey P*) $\in$ *analz* (*knows Spy evs4*), *clarsimp*)

apply (*drule asynth_insert*, *erule disjE*)

apply (*blast dest*: *analzplus_into_parts no_pairs*)

apply (*erule disjE*, *blast*)

apply *clarsimp*

apply (*erule disjE*)

apply (*erule disjE*)

apply (*blast dest*: *analzplus_into_parts no_pairs*, *erule disjE*)

apply *clarsimp*

apply (*case_tac R* = *Crypt b* (*Crypt c* (*Nonce Nv*)))

apply (*case_tac Key* (*invKey b*) ∈ *analz* (*knows Spy evs4*))

apply *force*

apply *simp*

apply (*case_tac Agent Col* ∈ *a2*)

apply (*blast dest*: *aanalz_PR analzplus_into_parts*)

apply (*case_tac P* = *priSK Adm*)

apply (*blast dest*: *aanalz_Adm analzplus_into_parts*)

apply (*case_tac P* = *priSK V*)

apply (*blast dest*: *V_unique analzplus_into_parts*)

apply (*case_tac P* = *b*)

apply *force*

apply (*blast dest*: *aanalz_Adm analzplus_into_parts*)

apply (*case_tac* ∃ *A B*. *m* = *Crypt A B*, *clarsimp*)

apply (*case_tac Nonce Nv* ∈ *parts* {*B*})

apply (*case_tac Agent Col* ∈ *a2*)

apply (*blast dest*: *aanalz_PR*)

apply (*case_tac A* = *priSK V*, *clarsimp*)

apply (*case_tac B* = *Crypt b* (*Crypt c* (*Nonce Nv*)), *clarsimp*)

apply (*case_tac Key* (*invKey b*) ∉ *analz* (*insert R* (*spies evs4*)))

apply (*blast dest*: *analz_cut analzplus_Nv CX_analz*)

apply (*force dest*!: *CX_analz*)

apply (*blast dest*: *aanalz_traffic V_unique*)

apply (*case_tac A* = *b*, *clarsimp*)

apply (*case_tac Key* (*invKey b*) ∉ *analz* (*insert R* (*spies evs4*)))

apply (*blast dest*: *analzplus_Nv*)

apply *clarsimp*

apply (*blast dest*!: *analz_cut CX_analz*)

apply (*case_tac A* = *priSK Adm*)

apply (*case_tac B* = *Crypt b* (*Crypt c* (*Nonce Nv*)))

apply (*case_tac Key* (*invKey b*) ∉ *analz* (*insert R* (*spies evs4*)))

apply (*blast dest*: *analz_cut analzplus_Nv CX_analz*)

apply (*force dest*!: *CX_analz*)

apply (*blast dest*: *aanalz_Adm*)

apply (*blast dest*: *aanalz_Adm*)

apply (*blast dest*: *analzplus_into_parts m_crypt1*)

apply *clarsimp*

apply (*blast dest*: *analzplus_into_parts m_crypt1 foo_V_privacy_aanalz*)

apply (*erule disjE*)

apply (*erule disjE*)

apply (*blast dest*: *analzplus_into_parts aanalz_PR*)

apply (*case_tac* $\exists$ *A B*. *m* = *Crypt A B*, *clarsimp*)

apply (*blast dest*: *aanalz_PR analzplus_into_parts m_crypt2*)

apply *clarsimp*

apply (*blast dest*: *foo_V_privacy_aanalz analzplus_into_parts m_crypt2*)

apply (*blast dest*: *foo_V_privacy_aanalz*)

apply (*blast dest*: *asynth_insert aanalz_PR*)

apply (*blast dest*: *b_sym*)

apply (*blast dest*: *asynth_insert foo_V_privacy_aanalz*)

apply (*force dest*!: *asynth_insert foo_V_privacy_aanalz*)

done

end

Bibliography

[1] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *Proc. of the 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115. ACM Press, 2001.

[2] M. Abadi, N. Glew, B. Horne, and B. Pinkas. Certified Email with a Light On-line Trusted Third Party: Design And Implementation, 2002.

[3] R. Anderson and R. M. Needham. Programming Satan's Computer. In J. Van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, LNCS 1000, pages 426–441. Springer, 1995.

[4] S. Andova, C. Cremers, K. Gjøsteen, S. Mauw, S. Mjølsnes, and S. Radomirović. A framework for compositional verification of security protocols. *Information and Computation*, 206:425–459, February 2008.

[5] A. Armando, W. Arsac, T. Avanesov, M. Barletta, A. Calvi, A. Cappai, R. Carbone, Y. Chevalier, L. Compagna, J. Cuéllar, G. Erzse, S. Frau, M. Minea, S. Mödersheim, D. von Oheimb, G. Pellegrino, S. E. Ponta, M. Rocchetto, M. Rusinowitch, M. T. Dashti, M. Turuani, and L. Viganò. The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures. In C. Flanagan and B. König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2012.

[6] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Heám, J. Mantovani, S. Mödersheim, D. von Ohe-

imb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In K. Etessami and S. K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005.

[7] A. Armando and L. Compagna. Abstraction-Driven SAT-based Analysis of Security Protocols. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 301–302. Springer, 2004.

[8] C. Ballarin. Tutorial to Locales and Locale Interpretation, 2012. `http://www.cl.cam.ac.uk/research/hvg/isabelle/dist/Isabelle2012/doc/locales.pdf` [Accessed 31 August 2012].

[9] C. Barrett and C. Tinelli. CVC3. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 298–302. Springer-Verlag, 2007.

[10] D. Basin, S. Capkun, P. Schaller, and B. Schmidt. Let's Get Physical: Models and Methods for Real-World Security Protocols. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 1–22, Berlin, Heidelberg, 2009. Springer.

[11] D. Basin, S. Mödersheim, and L. Viganò. An On-the-Fly Model-Checker for Security Protocol Analysis. In E. Snekkenes and D. Gollmann, editors, *Proc. of the 8th European Symposium on Research in Computer Security (ESORICS'03)*, volume 2808 of Lecture Notes in Computer Science, pages 253–270. Springer, 2003.

[12] D. A. Basin, C. J. F. Cremers, and S. Meier. Provably Repairing the ISO/IEC 9798 Standard for Entity Authentication. In P. Degano and J. D. Guttman, ed-

itors, *POST*, volume 7215 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2012.

[13] G. Bella. Inductive Verification of Smart Card Protocols. *Journal of Computer Security*, 11(1):87–132, 2003.

[14] G. Bella. *Formal Correctness of Security Protocols*. Information Security and Cryptography. Springer, 2007.

[15] G. Bella. The principle of guarantee availability for security protocol analysis. *Int. J. Inf. Secur.*, 9:83–97, April 2010.

[16] G. Bella, F. Blanqui, and L. C. Paulson. Theory libraries for Isabelle2012, 2012. `http://isabelle.in.tum.de/library/HOL/Auth` [Accessed 21 August 2012].

[17] G. Bella, D. Butin, and D. Gray. Holistic Analysis of Mix Protocols. In *7th International Conference on Information Assurance and Security (IAS 2011)*, pages 338–343, 2011.

[18] G. Bella and L. Coles-Kemp. Layered Analysis of Security Ceremonies. In *Proc. of The 27th IFIP International Information Security and Privacy Conference (IFIP SEC2012)*. Springer, 2012.

[19] G. Bella, F. Massacci, and L. C. Paulson. An Overview of the Verification of SET. *International Journal of Information Security*, 4(1-2):17–28, 2005.

[20] E. Biham and A. Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In A. Menezes and S. A. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[21] E. Biham and A. Shamir. Differential Cryptoanalysis of FEAL and N-Hash. In *EUROCRYPT*, pages 1–16, 1991.

[22] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96. IEEE Press, 1998.

[23] B. Blanchet and A. Podelski. Verification of Cryptographic Protocols: Tagging Enforces Termination. *Theoretical Computer Science*, 333(1-2):67–90, Mar. 2005. Special issue FoSSaCS'03.

[24] J. C. Blanchette. Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL, 2012. `http://isabelle.in.tum.de/doc/sledgehammer.pdf` [Accessed 21 August 2012].

[25] J. C. Blanchette. Picking Nits: A User's Guide to Nitpick for Isabelle/HOL, 2012. `http://isabelle.in.tum.de/doc/nitpick.pdf` [Accessed 21 August 2012].

[26] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Børner and V. Sofronie-Stokkermans, editors, *Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.

[27] J. C. Blanchette and T. Nipkow. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[28] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Static Validation of Security Protocols. *Journal of Computer Security*, 13:2005, 2005.

[29] D. Boneh and M. K. Franklin. Identity-Based Encryption from the Weil Pairing. In J. Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2001.

[30] D. Boneh and R. Venkatesan. Breaking RSA May Not Be Equivalent to Factoring. In *EUROCRYPT*, pages 59–71, 1998.

[31] M. Buchholtz. User's Guide for the LySatool version 2.01, 2005. `http://www2.imm.dtu.dk/cs_LySa/lysatool/lysatool-2.01.pdf` [Accessed 31 August 2012].

[32] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8:18–36, 1990.

[33] D. Butin and G. Bella. Verifying Privacy by Little Interaction and No Process Equivalence. In *Proceedings of the International Conference on Security and Cryptography (SECRYPT)*, 2012. To appear.

[34] R. Canetti, O. Goldreich, and S. Halevi. The Random Oracle Methodology, Revisited (Preliminary Version). In *STOC*, pages 209–218, 1998.

[35] C. D. Cannière and C. Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In X. Lai and K. Chen, editors, *ASIACRYPT*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[36] R. Chadha, c. Ciobâcă, and S. Kremer. Automated verification of equivalence properties of cryptographic protocols. In *Proceedings of the 21st European conference on Programming Languages and Systems*, ESOP'12, pages 108–127, Berlin, Heidelberg, 2012. Springer-Verlag.

[37] C. Cocks. An Identity Based Encryption Scheme based on Quadratic Residues. In *In IMA Int. Conf*, pages 360–363. Springer-Verlag, 2001.

[38] H. Comon-Lundh and V. Cortier. Computational Soundness of Observational Equivalence. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 109–118, New York, NY, USA, 2008. ACM.

[39] V. Cortier and S. Kremer, editors. *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*. IOS Press, 2011.

[40] C. Cremers. Feasibility of Multi-Protocol Attacks. In *Proc. of The First International Conference on Availability, Reliability and Security (ARES)*, pages 287–294, Vienna, Austria, 2006. IEEE Computer Society.

[41] C. Cremers. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In *Proc. of the 20th International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of Lecture Notes in Computer Science, pages 414–418. Springer, 2008.

[42] I. Damgård. A Design Principle for Hash Functions. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 416–427, London, UK, 1990. Springer-Verlag.

[43] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[44] S. Delaune, S. Kremer, and M. Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.

[45] S. Delaune, S. Kremer, and M. Ryan. Verifying privacy-type properties of electronic voting protocols: A taster. In *Towards Trustworthy Elections – New Directions in Electronic Voting*, volume 6000 of *Lecture Notes in Computer Science*, pages 289–309. Springer, 2010.

[46] S. Delaune, M. Ryan, and B. Smyth. Automatic verification of privacy properties in the applied pi calculus. *Syntax*, 263/2008:263–278, 2008.

[47] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[48] D. Dolev and A. C.-C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[49] B. Dutertre and L. de Moura. The Yices SMT solver, 2006. `http://yices.csl.sri.com/tool-paper.pdf` [Accessed 28 August 2012].

[50] J. Ellis. The story of non-secret encryption, 1987. `http://web.archive.org/web/20030610193721/http://www.cesg.gov.uk/ellisdox.ps` [Accessed 27 August 2012].

[51] C. Ellison. Ceremony Design and Analysis. *IACR eprint archive*, 399:2007, 2007.

[52] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and J. van Leeuwen, editors, *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.

[53] S. Escobar, C. Meadows, and J. Meseguer. A Rewriting-Based Inference System for the NRL Protocol Analyzer: Grammar Generation. In *Proceedings of the 2005 ACM workshop on Formal methods in security engineering*, FMSE '05, pages 1–12, New York, NY, USA, 2005. ACM.

[54] S. Escobar, C. Meadows, J. Meseguer, and S. Santiago. Sequential Protocol Composition in Maude-NPA. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *Computer Security – ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 303–318. Springer Berlin Heidelberg, 2010.

[55] A. J. Feldman, J. A. Halderman, and E. W. Felten. Security Analysis of the Diebold AccuVote-TS Voting Machine. In *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[56] Formal Systems. Failures-Divergence Refinement - FDR2 User Manual, 2012. http://www.cs.ox.ac.uk/projects/concurrency-tools/download/fdr2manual-2.94.pdf [Accessed 21 August 2012].

[57] D. Freeman, M. Scott, and E. Teske. A taxonomy of pairing friendly elliptic curves. Cryptology ePrint Archive, Report 2006/372, 2006. http://eprint.iacr.org/2006/372 [Accessed 21 August 2012].

[58] A. Fujioka, T. Okamoto, and K. Ohta. A Practical Secret Voting Scheme for Large Scale Elections. In *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques: Advances in Cryptology*, ASIACRYPT, pages 244–251. Springer-Verlag, 1993.

[59] E. Fujisaki and T. Okamoto. Secure Integration of Asymmetric and Symmetric Encryption Schemes. In M. J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

[60] F. D. Garcia, I. Hasuo, W. Pieters, and P. van Rossum. Provable Anonymity. In *Proceedings of the 2005 ACM workshop on Formal methods in security engineering*, FMSE '05, pages 63–72. ACM, 2005.

[61] S. L. Garfinkel. *PGP - Pretty Good Privacy: Encryption for Everyone*. O'Reilly, 1995.

[62] S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[63] L. Gong. Variations on the Themes of Message Freshness and Replay - or the Difficulty in Devising Formal Methods to Analyze Cryptographic Protocols. In *In Proceedings of the Computer Security Foundations Workshop VI*, pages 131–136. IEEE Computer Society Press, 1993.

[64] M. J. C. Gordon. *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, chapter 6. From LCF to HOL: A Short History. MIT Press, May 2000.

[65] D. Gray. Auditable Identity-Based Signatures with Strong Non-repudiation Properties. Dublin City University School of Computing, Working Paper CA-0307, 2007. www.computing.dcu.ie/wpapers/2007/0307.pdf [Accessed 21 August 2012].

[66] F. Hess. Efficient Identity Based Signature Schemes Based on Pairings. In K. Nyberg and H. M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 310–324. Springer, 2002.

[67] T. Hillenbrand, A. Buch, R. Vogt, and B. Löchner. WALDMEISTER – High-Performance Equational Deduction. *Journal of Automated Reasoning*, 18:265–270, 1997. 10.1023/A:1005872405899.

[68] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[69] S. Indesteege, F. Mendel, B. Preneel, and C. Rechberger. Collisions and other Non-Random Properties for Step-Reduced SHA-256. In *Selected Areas in Cryptography*, volume 5381 of *Lecture Notes in Computer Science*, pages 276–293. Springer, 2008.

[70] International Organization for Standardization, Genève, Switzerland. ISO/IEC 9798-3:1998/Amd.1:2010,Information technology - Security techniques - Entity Authentication - Part 3: Mechanisms using digital signature techniques. Amendment 1, 2010.

[71] Internet Engineering Task Force. Internet X.509 public key infrastructure certificate management protocols, 1999. http://www.ietf.org/rfc/rfc2510.txt [Accessed 21 August 2012].

[72] Internet World Stats. World internet usage and population statistics, 2011. http://www.internetworldstats.com/stats.htm [Accessed 30 August 2012].

[73] F. Jacquemard. Security Protocols Open Repository, 2009. http://www.lsv.ens-cachan.fr/Software/spore/index.html [Accessed 21 August 2012].

[74] E. Kiltz and G. Neven. *Identity-Based Cryptography*, chapter Identity-Based Signatures, pages 31–44. IOS Press, 2008.

[75] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[76] S. Kremer and M. Ryan. Analysis of an Electronic Voting Protocol in the Applied Pi Calculus. In *In Proc. 14th European Symposium On Programming (ESOP'05), volume 3444 of Lecture Notes in Computer Science*, pages 186–200. Springer, 2005.

[77] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, A. Rupp, and M. Schimmler. How to Break DES for 8,980€. In *International Workshop on Special-Purpose Hardware for Attacking Cryptographic Systems — SHARCS'06, Cologne, Germany*, 2006.

[78] R. Küsters and T. Truderung. Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation. pages 157–171, July 2009.

[79] R. Küsters and T. Truderung. Reducing Protocol Analysis with XOR to the XOR-Free Case in the Horn Theory Based Approach. *Journal of Automated Reasoning*, 46:325–352, 2011. 10.1007/s10817-010-9188-8.

[80] A. K. Lenstra and E. R. Verheul. Selecting Cryptographic Key Sizes. *Journal of Cryptology*, 14:255–293, 1999.

[81] Y. Li and J. Pang. An Inductive Approach to Provable Anonymity. In *Proceedings of The 6th Conference on Availability, Reliability and Security (ARES'11)*, pages 454–459. IEEE CS, 2011.

[82] G. Lowe. An Attack on the Needham-Schroeder Public-key Authentication Protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[83] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-key Protocol using CSP and FDR. In T. Margaria and B. Steffen, editors, *Proc of the 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of Lecture Notes in Computer Science, pages 147–166. Springer, 1996.

[84] G. Lowe. Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[85] J. E. Martina. Verification of Security Protocols Based on Multicast Communication. Ph.D. Thesis, Cambridge University Computer Laboratory, 2011.

[86] J. E. Martina and L. C. Paulson. Verifying Multicast-Based Security Protocols Using the Inductive Method. In *Workshop on Formal Methods and Cryptography (CryptoForma 2011)*, 2011.

[87] C. Meadows. Analyzing the Needham-Schroeder Public-Key Protocol: A Comparison of Two Approaches. In *ESORICS*, pages 351–364, 1996.

[88] C. A. Meadows. Formal Verification of Cryptographic Protocols: A Survey. In *Advances in Cryptology (ASIACRYPT 94)*, LNCS 917, pages 133–150. Springer, 1995.

[89] C. A. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[90] National Bureau of Standards (NBS). Data Encryption Standard (FIPS PUB 46-3), 1999. `http://csrc.nist.gov/publications/fips/archive/fips46-3/fips46-3.pdf` [Accessed 29 August 2012].

[91] National Institute for Science and Technology (NIST). Advanced Encryption Standard (FIPS PUB 197), 2001. `http://www.csrc.nist.gov/`

publications/fips/fips197/fips-197.pdf [Accessed 23 August 2012].

[92] National Institute for Science and Technology (NIST). Secure Hash Standard (SHS) (FIPS PUB 180-4), 2012. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf [Accessed 29 August 2012].

[93] National Institute for Science and Technology (NIST). Tentative Timeline of the Development of New Hash Functions, 2012. http://csrc.nist.gov/groups/ST/hash/timeline.html [Accessed 23 August 2012].

[94] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[95] T. Nipkow, L. C. Paulson, and M. Wenzel. Tutorial on Isabelle/HOL, 2012. http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/tutorial.pdf [Accessed 23 August 2012].

[96] F. Oehl, G. Cécé, O. Kouchnarenko, and D. Sinclair. Automatic Approximation for the Verification of Cryptographic Protocols. In A. E. Abdallah, P. Ryan, and S. Schneider, editors, *FASec*, volume 2629 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2002.

[97] N. O'Shea. Using Elyjah to analyse Java implementations of cryptographic protocols. *Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, pages 221–226, 2008.

[98] L. C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5, 1989.

[99] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. volume 828 of Lecture Notes in Computer Science. Springer, 1994.

[100] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous Connections and Onion Routing. *IEEE Journal on Selected Areas in Communications*, 16:482–494, 1998.

[101] A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2,3):91–110, Aug. 2002.

[102] R. Rivest and B. Kaliski. RSA problem. In *Encyclopedia of Cryptography and Security*, 2003.

[103] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[104] A. W. Roscoe. Model-Checking CSP. In A. W. Roscoe, editor, *A Classical Mind, Essays in Honour of C. A. R. Hoare*, pages 353–378. Prentice-Hall, 1994.

[105] P. Y. A. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and A. W. Roscoe. *Modelling and Analysis of Security Protocols*. Addison-Wesley, 2001.

[106] S. K. Sanadhya and P. Sarkar. Deterministic Constructions of 21-Step Collisions for the SHA-2 Hash Family. In T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, editors, *ISC*, volume 5222 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2008.

[107] P. Schaller, B. Schmidt, D. Basin, and S. Capkun. Modeling and Verifying Physical Properties of Security Protocols for Wireless Networks. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 109–123, Washington, DC, USA, 2009. IEEE Computer Society.

[108] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *Proceedings of the 2012 25th IEEE Computer Security Foundations Symposium (CSF'12)*, 2012.

[109] S. Schulz. E – A Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002.

[110] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci.*, 121(1-2):411–440, 1993.

[111] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[112] A. Shamir. Identity-based cryptosystems and signature schemes. In *CRYPTO*, pages 47–53, 1984.

[113] C. E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28:656–715, 1949.

[114] P. W. Shor. Polynominal Time Algorithms for Discrete Logarithms and Factoring on a Quantum Computer. In L. M. Adleman and M.-D. A. Huang, editors, *ANTS*, volume 877 of *Lecture Notes in Computer Science*, page 289. Springer, 1994.

[115] D. Song, S. Berezin, and A. Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9:2001, 2001.

[116] G. Sutcliffe. System on TPTP, 2009. `http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP` [Accessed 23 August 2012].

[117] Telecommunication Standardization Sector (ITU-T). Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks, 2005. `http://www.itu.int/rec/T-REC-X.509/en` [Accessed 23 August 2012].

[118] F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand Spaces: Proving Security Protocols Correct. *Journal of Computer Security*, 7:191–220, 1999.

[119] Trend Micro. Email encryption client, 2009. http://uk.trendmicro.com/imperia/md/content/uk/products/datasheets/ds01_tmee_080609gb.pdf [Accessed 23 August 2012].

[120] Voltage Security. Voltage identity-based encryption, 2012. http://www.voltage.com/technology/ibe.htm [Accessed 23 August 2012].

[121] B. Waters. Efficient Identity-Based Encryption Without Random Oracles. In R. Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 114–127. Springer, 2005.

[122] C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In H. Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 1999.

[123] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS Version 3.5. In R. A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.

[124] M. Wenzel. The Isabelle/Isar Reference Manual, 2012. http://isabelle.in.tum.de/doc/isar-ref.pdf [Accessed 23 August 2012].

[125] S. Wiesner. Conjugate Coding. *SIGACT News*, 15(1):78–88, 1983.

[126] J. Zhou and D. Gollmann. A Fair Non-repudiation Protocol. In *Proc. of the 15th IEEE Symposium on Security and Privacy (SSP'96)*, pages 55–61. IEEE Press, 1996.