DOI 10.1007/s00165-012-0246-3
BCS © 2012
Formal Aspects of Computing

Formal Aspects of Computing

Inductive study of confidentiality: for everyone

Giampaolo Bella[1,2]

[1] Software Technology Research Laboratory, De Montfort University, Leicester, UK
[2] Dipartimento di Matematica e Informatica, Università di Catania, Viale A. Doria 6, I-95125 Catania, Italy

Abstract. The Inductive Method is among the most established tools to analyse security protocols formally. It has successfully coped with large, deployed protocols, and its findings are widely published. However, perhaps due to its embedding in a theorem prover or to the lack of tutorial publications, it is at times criticised to require super-specialised skills and hence to be rather impractical. This paper aims at showing that criticism to be stereotypical. It pursues its aim by presenting the first tutorial-style paper to using the Inductive Method. This paper cannot cover every aspect of the method. It focuses on a key one, that is how the Inductive Method treats one of the main goals of security protocols: confidentiality against a threat model. The treatment of that goal, which may seem elegant in the Inductive Method, in fact forms a key aspect of all protocol analysis tools, hence the paper motivation rises still. With only standard skills as a requirement, the reader is guided step by step towards design and proof of significant confidentiality theorems. These are developed against two threat models, the standard Dolev–Yao and a more up-to-date one, the General Attacker, the latter turning out particularly useful also for didactic purposes.

Keywords: Security protocol, Inductive Method, Theorem proving, Isabelle, Threat model

1. Introduction

Along with the NRL Protocol Analyzer [Mea96], CSP with model checking [RSG+01] and Strand Spaces [THG99], the Inductive Method [Pau98] is among the longest-established machine assisted tools for the formal analysis of security protocols. Born in the mid 1990s with the aim of providing machine support for reasoning by belief logics [BAN89], it soon showed a personality of its own. It has been significantly developed and extended throughout the years [Bel07] to cope with virtually all security protocol varieties, such as industrial-strength protocols [BMP06], fair-exchange ones [BP06] and distance-bounding protocols [BCSS09]. The support that the interactive theorem prover Isabelle [NPW02] offers to the Inductive Method cannot do without human intervention but has recently been empowered with significant automation [Pau10]. Interactive work retains its importance of intuition booster in supporting the protocol analyst's most exploratory conjectures. The findings might then be implemented into more automated tools, such as Proverif [Bla11].

Correspondence and offprint requests to: G. Bella, E-mail: giamp@dmi.unict.it

Published online: 23 June 2012

The Inductive Method has at times received criticism as to require super-specialised skills and hence to be rather impractical. This paper aims at showing that criticism to be stereotypical, arguing that most skills required of the learner can be acquired through a standard Computer Science syllabus. However, the essence of such criticism may derive from the lack of hands-on, tutorial-style publications, which can effectively guide the reader, step by step, to study protocols inductively. This is the first paper written with such aim. The aim implies deep inspection of the proof goals that are encountered during the study, a practice that our teaching experience deems indispensable, whereas our community often banish: "Presentation of goal states does not conform to the idea of human-readable proof documents!" [Wen11, p. 44]. Another reason to our claim is that the Inductive Method is written in the traditional Isabelle style, as opposed to the more recent and human-friendly *Isar* (Intelligible Semi-Automated Reasoning) style.

This paper focuses on the confidentiality goal, perhaps the main one for most security protocols, and consists of three main parts. The first part reviews the existing account on confidentiality by detailing it in depth. It is conducted against the standard threat model for security protocol analysis, due to Dolev and Yao [DY83], where an attacker controls the network traffic but cannot do cryptanalysis. The second part, still developed against the standard threat model, presents an innovative, more comprehensive and systematic study of confidentiality, both in terms of specification of the relevant guarantees and of development of lines of reasoning to establish them. The final part conducts from scratch a confidentiality study against a more recent threat model where every agent *may* misbehave [ABCC09a, ABCC09b]. Among its pros, this forces the elicitation of all preconditions to a relevant guarantee, an important didactic exercise.

The best way to read this paper is to download and install Isabelle [URL11b], and then download and execute our theory files from our theory archive [Bel12], which contains the full scripts this paper refers to.

1.1. Prerequisites to the reading

It is important to state the set of minimal skills that the user should acquire before they venture in this reading:

1. **Logic** as the main specification language: first-order logic, an idea of quantifier generalisation for higher-order logic (HOL), classical reasoning, unification, resolution [NPW02].
2. **Set theory** as the main tool to make claims: the main set operators [Hrb99].
3. **Mathematical induction** as the main working principle: specification and proof [MVM09].
4. **Term rewriting** as the main computational paradigm: expression update by means of equations [FB99].
5. **Security protocols** as the main target: property enforcing over distributed communications [BM03].
6. **Isabelle** as the main computer program supporting the treatment: its implementation of various proof methods for resolution [Wen11].
7. **The Inductive Method** as the use of mathematical induction to tackle security protocols using a logical language: guarantee design and interpretation [Pau98].

While all skills outlined above are needed prerequisites, we believe that none of them are necessary to have at the top level of proficiency. Rather, a familiarity level could suffice at least for first reading. A by-product of the reading will be an appreciable increase of familiarity with the skills themselves. The process could be conveniently iterated.

The first five skills will not be treated in this manuscript, as they can be considered part of a standard Computer Science syllabus. This observation supports our claim that using the Inductive Method profitably requires only standard skills. However, Isabelle and the Inductive Method may not be considered standard skills if standard means a Bachelor's level. For example, it is our experience that they are only taught at post-graduate level. We shall therefore summarise them here.

1.2. Paper summary

This manuscript continues by outlining Isabelle (Sect. 2) and the Inductive Method (Sect. 3). It details the inductive study of confidentiality against a standard threat model, by revisiting what is available (Sect. 4) and by conducting it afresh (Sect. 5). After an evaluation of the findings, it details a novel study of confidentiality against a more audacious threat model (Sect. 6), and contrasts the findings with those discussed earlier. Then, it concludes (Sect. 7).

2. Isabelle

Isabelle is a generic, interactive theorem prover [Pau94]. *Generic* means that it supports reasoning in a variety of formal logics. Isabelle/HOL [NPW02] supports HOL, a typed formalism that allows quantification over functions, predicates and sets. *Interactive* means that it is not entirely automatic and, rather, requires human intervention. In a typical proof, the user directs Isabelle to perform a certain induction and then to simplify the resulting *subgoals* forming the *proof state*. Then, they can each be treated by classical reasoning, also appealing to available lemmas, and be reduced to other subgoals or closed. Failure to find a proof for a conjecture may simply mean that the user is not skilled enough; otherwise, it exhibits what in the system being modelled contradicts the conjecture. This often helps in locating a system bug.

The interaction with Isabelle takes place by *commands*, such as **apply** to launch a *proof method*, and **done** to state that a proof has terminated [Wen11]. The diagnostic command **thm**, followed by a theorem name, is useful to the analyst's inspection of the theorem, possibly to evaluate whether it is useful in the current proof state. The most typical proof methods, `rule`, `erule` and `drule`, implement resolution of the first subgoal in the proof state with theorems that are respectively introduction, elimination or destruction rules. As of today, Isabelle has reached a considerable level of automation. Its simplifier, which can be invoked by the proof method `simp`, combines rewriting with arithmetic decision procedures. Its automatic provers, can solve most subgoals. In particular, `clarify` does the obvious steps, `blast` implements a generic tableau prover, and `force` combines it with appeals to the simplifier. While they both apply to the first subgoal in the proof state, the `auto` method tackles the entire state. All provers except `clarify` can conveniently appeal to the lemmas that the user deems useful. The recent combination with fully automatic first-order-logic provers, which can be invoked after simplification, speeds up the developments considerably [Pau10].

The series of commands used to prove a theorem form the *proof script*. It coexists in a single file together with some specification of the target system, as we shall see below. Such a file forms an Isabelle *theory*, with the typical `.thy` extension. Theories can be built on top of each other, inheriting specifications and theorems developed in the ancestor theories. The next section demonstrates this.

Isabelle can be freely downloaded from the Internet [URL11b] under the Open Source Software BSD licence. It is available for most Unix platforms, such as Linux, MacOS X and Solaris. For Microsoft Windows platforms, it is sufficient to install a Linux-like environment such as Cygwin [URL11a] or a virtual Linux machine. Installing Isabelle reduces to unpacking a tar ball, which includes an ML compiler and runtime system, such as Poly/ML [URL11c]. It also includes the latest version of the Proof General graphical interface [URL11d], which allows the user to write the proof commands inside an editor window, and to navigate through it by means of intuitive buttons. Meanwhile, the proof state dynamically refreshes in a separate window. Each distribution of Isabelle comes with a repository of proofs, organised through a theory hierarchy. A new distribution is released every year to include the latest developments to the provers and optimisations to the proof scripts installed in the repository.

Isabelle has many sibling tools, notably PVS and Coq, and some comparison is available [Wie06]. Isabelle stands out for at least two reasons. One is that it allows the user to define logics of his own, the other is its deep integration with the most widely-adopted document preparation system. For example, all scripts quoted in the sequel of this manuscript have been mechanically translated from the theory files into LaTeX. This paper uses Isabelle/HOL because our treatment is built on HOL. Therefore, any other tool capable of handling the same formalism should be able to reproduce what follows.

3. The Inductive Method

The Inductive Method [Pau98] indicates an application of mathematical induction to reason on security protocols with the support of the theorem prover Isabelle. The method rests on a protocol model based on the notion of *trace*, that is a network history arising from a particular use of the protocol by various agents. A trace is a list of events, each event modelling a notable happening in the use of the protocol. For example, an event may be an agent's initiation of the protocol or another agent's reception of a protocol message. Another possible event is the attacker's malicious faking of a message that deviates from the formats that the protocol prescribes. Induction is

used to construct the protocol model as the set of all possible traces: having set the base of the induction through the empty trace being in the protocol model, an inductive rule states how to extend a trace of the model in such a way that the extension still belongs to the model. Typically, one inductive rule is defined for each of the protocol events of sending a message from an agent to another one.

One of the strengths of this approach is that the protocol model is unbounded. There are no set limits to the number of participating agents, the number of traces in the model, the length of each trace, and the number of sessions that can be interleaved. Most importantly, there is no limit to the activity of the attacker, termed the *spy*, which embodies the full potential of the Dolev–Yao threat model [DY83] (DY in brief). The spy can intercept all messages that are sent in the traffic, break down concatenated messages into components as well as open up ciphertexts of which she knows the corresponding key, and use them actively to build fake messages of any form. In addition, she is a registered agent and can therefore execute the protocol correctly using her own legal credentials such as private keys. There also is a set of compromised agents who collude with her revealing their credentials in such a way that the spy can act on their behalf. However, the spy cannot perform cryptanalysis, so that this work falls in the category of symbolic protocol analysis, as opposed to the category of computational protocol analysis.

Mathematical induction is also used as a proof principle in the traditional way. An inductive proof derives from applying the axiom of induction, stating that a property that holds of the base case and of all inductive steps also holds in general. Therefore, an inductive proof is the process of establishing that the preconditions of the axiom of induction, which are called subgoals, hold. In the case of security protocols, a proof by induction scrutinises all rules defining the protocol model. More precisely, the analyst conjectures a property of the protocol, and then matches it to each protocol rule in order to verify whether the rule preserves the conjectured property. When this is affirmative for all protocol rules, a proof by induction is found, hence a theorem states that the protocol establishes the property. We shall see through many examples below that, in proving confidentiality, the most complicated subgoal typically arises from the protocol rule that incorporates the threat model.

Chief security protocol goals, such as confidentiality and authentication, can be studied extensively by induction, but also other important ones [Bel07]. This manuscript focuses on confidentiality with a hands-on style. It must be observed that the size of the terms can become very big, and the length and complexity of the proof state often stretches human intuition. This is the reason why a theorem prover such as Isabelle helps considerably. There may be subgoals with a simple logical structure but spanning many pages. Isabelle will guide the analyst and possibly reduce or close automatically those subgoals. Proofs could otherwise be conducted by pen and paper using conditional rewriting techniques and classical reasoning.

The Inductive Method is written in HOL, however, we believe that many users who are familiar with first-order-logic may easily get to grips with the extended use of quantification. Technically, HOL is specified in theory file *HOL.thy*, which in turn is built on its intuitionistic fragment coming with *Pure.thy*—these are clearly explained in the reference manual [Wen11]. The HOL theory file comes under subdirectory `src/HOL` of the Isabelle repository [URL11b]. The same folder contains all theory files based on that logic. Each file contains the homonymous theory name. The Inductive Method is implemented in three theory files that are grouped under subdirectory `src/HOL/Auth`, along with the studied protocols. Each theory file contains the formalisation of related elements: protocol messages in *Message.thy*, protocol events in *Event.thy* and cryptographic primitives in *Public.thy*. The theory of protocol messages imports the various theories defining the logic language, and hence inherits various useful types and related functions. Each of the other two theories imports the previous one, and a protocol is then defined on top of *Public.thy*, as we shall see below through the outline of the main elements of each theory. Each theory file also contains the pertaining theorems. We omit them purposely from this resume and only recall those that are needed in the subsequent treatment. This is in line with the aim of this manuscript, which is not to present the available results but, rather, to detail how to construct the main ones.

The Inductive Method has always been presented concept by concept before, never through its three main theory files. Our experience indicates that going file by file favours the reader. A graphical representation of the interdependencies among all theories of interest in this manuscript [Bel12] is in Fig. 1. While the leaf theories are entirely new, the left branch is identical to the version that comes with the Isabelle repository [URL11b], with only minor modifications to the protocol theory *NS_Public_Bad.thy*. The right branch is an innovative variant of the left branch. All features and variants are discussed below, except for theory *Knowledge.thy*, which contains general knowledge equivalence results that are not fundamental to this paper.

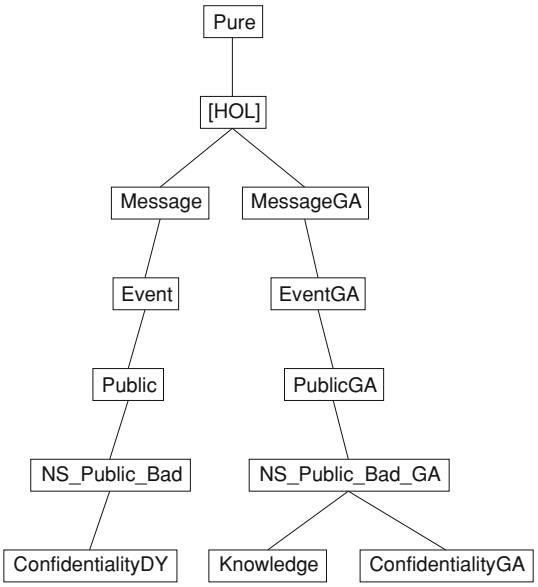

Fig. 1. Interdependency graph of all theories of interest in this manuscript

3.1. Message.thy

This theory file builds the main types that are useful to specify security protocols. There is a type *key*, which renames the natural numbers, for cryptographic keys:

```
types
  key = nat
```

Some functions to distinguish symmetric from asymmetric keys are introduced, and are straightforward. They are omitted here, while more important ones will be introduced in the subsequent theories.

The important datatype *agent* defines the protocol participants. It features a trusted third party, which is often used in symmetric-key protocols, the agents who always conform to the protocol being specified and, additionally, the attacker. These are specified as type constructors of the datatype, respectively as *Server*, *Friend* and *Spy*. The second constructor is parametric over the natural numbers, so that the population of agents conforming to the protocol, called *friendly agents*, is unbounded:

```
datatype    agent = Server | Friend nat | Spy
```

The final essential datatype is for the protocol messages. It features seven constructors corresponding to seven message types respectively. All constructors are parametric over a type, except for the two constructors building concatenated messages and ciphertexts respectively, which take the two obvious parameters. The datatype is quoted here with the original comments coming from the theory file:

```
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
```

The three main operators to manipulate message sets can be introduced. With `::` being the definition symbol, the operator that extracts all message components is defined by induction as:

```
inductive_set
  parts :: "msg set ⇒ msg set"
  for H :: "msg set"
  where
     Inj [intro]:              "X ∈ H ⟹ X ∈ parts H"
  |  Fst:          "{|X,Y|}   ∈ parts H ⟹ X ∈ parts H"
  |  Snd:          "{|X,Y|}   ∈ parts H ⟹ Y ∈ parts H"
  |  Body:        "Crypt K X ∈ parts H ⟹ X ∈ parts H"
```

The **for**. . . **where** syntax is used to state the four rules defining how the `parts` operator applies to a generic message set. The first rule, named `Inj`, sets the base of the induction, stating that an element of the given message set also belongs to `parts` of the given set. The central rules, `Fst` and `Snd`, state that both components of a concatenated message can be extracted. The final rule, `Body`, establishes that also the body of a ciphertext can be taken out. The theorem prover stores each of these defining rules as a theorem for later invocation. Invocation can be performed by using the theorem names of interest, which with all inductive definitions are built methodically using a dotted notation: in this case for example, the resulting theorem names are `parts.Inj`, `parts.Fst`, `parts.Snd` and `parts.Body`.

In particular, it can be seen that the first rule is installed as an introduction rule, by `[intro]`, because it introduces the symbol being defined, that is `parts`. It implies that the automatic provers will by default appeal to it. This stable installation is at times risky because it may break an available fact up into others. Also, while it compacts the proof script on one hand, we believe it also complicates the human understanding of the proof by hiding the theorems that must necessarily be applied to close a subgoal. Therefore, we decide to avoid this practice in our treatment built on top of these basic theories, as the sequel of the paper demonstrates. In terms of syntax, the use of quotation marks can be observed to surround declarations, definitions and, in the following, lemma or theorem statements. They are omitted for clarity when used within the actual prose. Isabelle output, such as a response to an instance of command **thm**, also omits them.

The main feature of `parts` is that it treats ciphertexts as if they were broken. This has seemed (to some learners) to contradict the DY threat model. However, this operator is only used to refer to every component that appears on a trace, also termed *network traffic* on that trace. Also, stating that a message component does not belong to `parts` of a message set can be used as a strong confidentiality claim, as we shall see. For finer claims, there also is a similar operator, `analz`, defined analogously, with the only difference that ciphertexts can be opened only reasonably, that is if the corresponding key is available:

```
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"
```

Rule `Decrypt` insists on two preconditions to derive its postconditions. When more than one precondition must be stated, they are surrounded by double brackets and separated by semicolon. The rule refers to the inverse of the key that encrypts the stated ciphertext. The inverse is defined by a self-explanatory function, omitted here. The rule is general for both symmetric and asymmetric cryptography. In particular, with symmetric cryptography, a key and its inverse coincide. The extra constraint of this rule with respect to `parts.Body` gives the intuition that `analz H ⊆ parts H` for any message set `H`, a theorem named `analz_into_parts` that is often useful. Rule `Decrypt` is defined as a destruction rule, by `[dest]`, because it deletes an occurrence of the symbol being defined. Rule `Inj` is also made available to the simplifier.

The final message operator, `synth`, builds up messages from available components (typically obtained by means of `analz`). Its defining rules are all installed as introduction rules:

```
inductive_set
  synth :: "msg set ⇒ msg set"
  for H :: "msg set"
  where
    Inj    [intro]:   "X ∈ H ⟹ X ∈ synth H"
  | Agent  [intro]:   "Agent agt ∈ 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"
```

Agent names and numbers (such as timestamps) can be synthesized without any precondition, indicating that they are guessable. However, nonces cannot be synthesized. The remaining rules state that hashes, concatenated messages and ciphertexts can be built upon the preconditions that the necessary components are available.

As anticipated, `analz` and `synth` can be used in combination. For example, `synth(analz H))` is the set of messages that can be synthesized at will from the components obtained by analysing the given message set `H`. Once it is clear how to define `H` meaningfully, this mechanism will be profitably used to define the spy as a DY attacker.

3.2. Event.thy

This theory file models what can take place during the execution of a security protocol: the events of sending, receiving and noting down a message. It also presents the definitions of all related functions. Events are introduced as a datatype, with a type constructor for each:

```
datatype
  event = Says  agent agent msg
        | Gets  agent       msg
        | Notes agent       msg
```

The first constructor takes three types as parameters, `agent`, `agent` and `msg`, so that for example `Says A B X` formalises the act of an agent `A` who sends message `X` to agent `B`. Likewise, a `Gets B X` event indicates an agent `B`'s reception of message `X`. The final constructor introduces events of form `Notes A X`, formalising an agent who records a message, possibly for future use. When the threat model definition is complete, it will be appreciated that this event is used for private records.

It is now appropriate to introduce the set of agents, `bad`, who are compromised to the extent of revealing their secrets to the spy. This set is only declared at this stage, but is characterised later:

```
  bad    :: "agent set"
```

Two characterisations of this set are that the spy belongs to it but the server does not. Appropriate rules, omitted here, state these facts. Two additional characterisations come below. Remarkably, it is unnecessary to state precisely how many and what agents belong to `bad`, because it simply remains a parameter of the analysis. For example, typical theorem preconditions mention generic agents who do not belong to `bad`.

All functions are now available to begin defining agent knowledge. Without purely epistemic connotation, knowledge here indicates the messages that can be derived from participating in a protocol. Agents have some initial knowledge prior to the beginning of any protocol. This is declared here, because it forms a parameter of a subsequent function, but is defined in the next theory. Its name may be misleading towards *finite-state analysis* [McM93], when in fact it only indicates a set of messages:

```
consts
  initState :: "agent ⇒ msg set"
```

One of the fundamental functions is `knows` [Bel07], which formalises the knowledge that an agent derives from a given trace. It must then take two parameters, an agent and a trace, and return a message set. To allow for the DY threat model, the function makes sure that the spy knows all messages that are sent (by anyone), but also the notes of compromised agents. This establishes the third characterisation of the set `bad`: agents who do not belong to it will keep their notes private. It may surprise that the messages that are received do not expand the spy's knowledge. Every protocol model (Sect. 3.4) enforces reception only of those messages that were previously sent, a realistic constraint that supports the simplification of `knows`. Any agent different from the spy only knows the messages they send or receive. The function is defined by primitive recursion on the length of a trace. Rule `knows_Nil` treats the empty trace, and rule `knows_Cons` treats the recursive case of a trace whose first element is an event `ev`. Two elements of syntax are needed here. One is `#`, for the list cons operator. Another is `insert`, for union of a set with a singleton, so that `{X} ∪ H` can also be represented as `insert X H`. To understand the following definition fully, it must be observed that traces are built from right to left (Sect. 3.4):

```
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))"
```

The base case states that an agent knows their initial state since the beginning, hence `initState A` is the symbolic evaluation of `A`'s knowledge on the empty trace. This is also the first time that the trace type `event list` appears.

Three important sets of messages can now be built. Two of them let us formalise confidentiality. It can be stated for a message `m` over a trace `evs` as `m ∉ parts(knows Spy evs)`. It means that `m` does not feature as any of the components of the messages that the spy can extract from the trace, that is essentially the entire network traffic except private notes. This is a strong fact. Alternatively, a weaker fact that is still very meaningful is `m ∉ analz(knows Spy evs)`. It means that `m` cannot be derived from the analysis of all messages that the spy can extract from the trace.

The third important set is `synth(analz(knows Spy evs)`. It elegantly formalises the huge potential of the DY spy. She can intercept all traffic by means of function `knows`, analyse its components by `analz`, and build new messages using those components by `synth`. This set is unbounded, as it grows with the length of the trace it is built upon. Therefore, the formalisation does not limit the offensive potential of the spy. The protocol specification will rely on this set to allow for the spy's malicious behaviour while she engages with the protocol (Sect. 3.4).

Another important function helps to model freshness, `used`. It applies to a trace and returns a message set. Informally, it yields the components, extracted by means of `parts`, of all messages appearing in the trace, extended with the components of the initial states of all agents. Its formal definition is omitted. For a given trace `evs`, writing that a message `m` is such that `m ∉ used evs` means that `m` does not appear on `evs` at all, hence it signifies that it is fresh on the trace. This is very practical for issuing *dynamic secrets*, also called short-term secrets, such as session keys and nonces, which have not existed forever and are only built at some stage in the execution of a protocol: on a given trace for which fact `m ∉ used evs` is stated. Incidentally, messages should be addressed as secrets only after suitable guarantees establish that they are confidential, but terminology will often be abused.

3.3. Public.thy

This theory file introduces the functions formalising long-term keys: `pubEK` for public encryption keys, `pubSK` for public signature keys and `shrK` for keys shared with the server. Each function takes an agent as a parameter and returns a key. There also exist functions for the private halves, `priEK` and `priSK`. Their definitions are outside our focus, and are omitted here. Those for asymmetric keys are syntactic translations of a main function that takes as extra parameter the working mode, encryption or signature. All keys were unified in this single theory a few years after the method's inception. This explains why the theory name may not appear in line with its contents: the original treatment separated asymmetric keys in `Public.thy` and symmetric ones in `Shared.thy`, which does not exist anymore.

The main definition that this file features is for agents' initial states. It establishes that the server knows its private keys, all public keys, and also stores all shared keys. This can be defined compactly by appealing to the function `range`, which is self-explanatory, in combination with `‘`, the image operator. As a result, for example `Key ‘ range pubEK` is a set of keys. The definition also enforces that a friendly agent initially knows only her *static secrets*, also called long-term secrets, those that exist forever: her private keys and her key shared with the server. Moreover, the agent also has all public keys. As it can be expected, the spy has access to all public keys. The fourth

characterisation of the set `bad` is that, additionally, the spy has access to the private and shared keys of agents who are in the set. The constant `initState`, which was introduced in the previous theory file, is now overloaded to meet a definition by primitive recursion:

```
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) =
       {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)"

end
```

3.4. Example protocol

For didactic reasons, we choose the best known protocol as an example: the Needham–Schroeder public-key protocol subject to Lowe's man-in-the middle attack. It is perhaps the most published ever, and we expect this feature to make our study more accessible. It must be stressed that our study will be most general: either protocol-independent or else presenting general proof strategies that will be easy to instantiate over each protocol.

For the reader's convenience, we opt to quote the example protocol because it is very short, but avoid explanations that we deem part of the basic skills in security protocols. As customary, the early steps whereby agents fetch the public keys are omitted:

$$1.\quad A \to B : \{Na, A\}_{Kb}; \qquad 2.\quad B \to A : \{Na, Nb\}_{Ka}; \qquad 3.\quad A \to B : \{Nb\}_{Kb}$$

The protocol model is the constant `ns_public`, defined inductively as the set of all possible traces arising from the given protocol. It can be found in the theory file `NS_Public_Bad.thy` [Bel12] (the original theory version [URL11b] was developed prior to the introduction of the `Gets` event). The constant is defined by induction through the usual syntax. It is a set of traces, hence its type is a set of lists of events:

```
inductive_set ns_public :: "event list set" where
```

It can be seen that, contrarily to the definitions seen above for functions, no **for** statement is necessary because we are defining a constant. The defining rules follow **where**. In particular, the `Nil` rule sets the base of the induction by introducing the empty trace in the constant:

```
  Nil:  "[] ∈ ns_public"
```

All other rules are inductive, as they assume a trace of the protocol model and detail how to extend it (with an event that the protocol admits) into a trace that is stated to belong to the model too. In particular, two rules appear unchanged in every protocol model. One, `Fake`, formalises the DY spy. The rule states that, given a trace of the protocol and a message drawn from the spy's fakes, then the trace can be extended (from the left end) with the event whereby the spy sends that message to a generic agent:

```
| Fake: "⟦evsf ∈ ns_public;  X ∈ synth (analz (knows Spy evsf))⟧
       ⟹ Says Spy B X  # evsf ∈ ns_public"
```

Another universal rule, *Reception*, was introduced through the developments of the Inductive Method [Bel07] to state that a message that is sent can be received. Using the predefined *set* function, which extracts the events of a trace as a set, the rule insists on a trace where a *Says* event appears, and extends it with the corresponding *Gets* event to still obtain a trace of the model:

```
| Reception: "⟦evsr ∈ ns_public; Says A B X ∈ set evsr⟧
           ⟹ Gets B X # evsr ∈ ns_public"
```

Only the rules that specifically model the example protocol are left. Rule *NS1* models an initiator sending the initial protocol message at any stage, that is without particular preconditions except that the nonce inserted in the message is fresh. Rule *NS2* models the responder's reply, and so assumes another fresh nonce and also that the event whereby the responder receives the initial message exists in the trace. Finally, rule *NS3* models the initiator sending the final message, hence upon the assumptions that he sent the initial one and received the corresponding reply. The three rules can be presented compactly:

```
| NS1:  "⟦evs1 ∈ ns_public;  Nonce NA ∉ used evs1⟧
        ⟹ Says A B (Crypt (pubEK B) {|Nonce NA, Agent A|})
             # evs1  ∈  ns_public"

| NS2:  "⟦evs2 ∈ ns_public;  Nonce NB ∉ used evs2;
         Gets B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs2⟧
        ⟹ Says B A (Crypt (pubEK A) {|Nonce NA, Nonce NB|})
             # evs2  ∈  ns_public"

| NS3:  "⟦evs3 ∈ ns_public;
         Says A  B (Crypt (pubEK B) {|Nonce NA, Agent A|}) ∈ set evs3;
         Gets A (Crypt (pubEK A) {|Nonce NA, Nonce NB|}) ∈ set evs3⟧
        ⟹ Says A B (Crypt (pubEK B) (Nonce NB)) # evs3 ∈ ns_public"
```

The inductive specification of the example protocol is concluded. For completeness, it should be noted that each protocol rule is implicitly in the scope of a meta-level universal quantifier, $\bigwedge$, which will be sometimes made explicit below. Intuitively, the meta-universal establishes that *any* trace can be extended as each rule specifies. This is clearly a higher logical level than the object level, which is used to express the protocol properties. A practical implication is that all rules assume traces that could be called anything and, in particular, always with the same name because each rule is in the scope of a different quantifier. However, each rule customarily uses a trace variable name that identifies the rule. During the proofs, this helps the analyst pinpoint which protocol rule triggered the very subgoal under scrutiny.

Also this theory file, as the previous ones, contains a number of important theorems with their proofs. Most are omitted from this manuscript, except for the confidentiality ones, which will be detailed fully. Moreover, broader confidentiality guarantees will be studied and evaluated.

4. Confidentiality against Dolev–Yao: existing study revisited

The study discussed in this section forms our theory file *ConfidentialityDY.thy* [Bel12]. Its aim is to revisit in great detail the study of confidentiality that currently comes with the repository.

Following the protocol specification, the next and most complex phase is the actual formal reasoning about the model. With theorem proving, formal reasoning boils down to stating conjectures and attempting to prove them. Proving a conjecture about a protocol model requires closing n subgoals corresponding respectively to the n rules defining the model. It turns out that all subgoals are simple to close with all protocols currently in the repository [URL11b], except the one corresponding to the *Fake* case:

> "The *Fake* case usually survives, but it can be proved by a standard argument involving the properties of *synth* and *analz*. This argument can be programmed as a tactic, which works for all protocols investigated" [Pau98, Sect. 4.5].

The proof method that the excerpt addresses as "tactic" using old terminology is called *spy_analz*, and is implemented in file *Message.thy*, outlined above (Sect. 3.1). Our interest is beyond debugging its code, that is understanding what proof strategy it takes. Its gist is the application of a lemma chosen among two crucial ones, named *Fake_parts_insert* and *Fake_analz_insert* [URL11b]. Unfortunately, these have never been discussed in detail through publications. Nonetheless, we strongly believe that understanding how these lemmas contribute to proofs through *spy_analz* is crucial to digest how to reason on confidentiality against DY by induction. Paulson published the only relevant comment:

> "(...) *synth(parts H)* and *synth(analz H)* appear to be irreducible. (...) we can derive a bound on what the enemy can say:
>
> $$\frac{X \in synth(analz\ H)}{parts(\{X\} \cup H) \subseteq synth(analz\ H) \cup parts\ H}$$
>
> *H* is typically the set of all messages sent during a trace. The rule eliminates the fraudulent message X, yielding an upper bound on *parts({X} ∪ H)*. Typically, *parts H* will be bounded by an induction hypothesis. There is an analogous rule for *analz*" [Pau98, Sect. 3.3].

The practical implications of this lemma do not seem obvious. In particular, while the fake messages are extracted from the set *synth(analz H)*, the upper bound is given on the set *parts({X} ∪ H)*, but the relation between these two sets is not straightforward. And it remains unclear how the induction hypothesis available on *parts H* can help. Moreover, this lemma is only *Fake_parts_insert*, which is the simpler of the two. It would therefore seem that the proof of the *Fake* subgoal in the confidentiality proofs has never been given the complete treatment that it deserves.

4.1. Static secrets

Most surprisingly, *spy_analz* is apparently irrelevant to all confidentiality proofs of static secrets, something that seems to contradict the excerpts seen above. For example, the confidentiality of private keys for our example protocol, as found in theory file *NS_Public_Bad.thy* [URL11b], are as follows:

```
lemma Spy_see_priEK [simp]:
  "evs ∈ ns_public ⟹ (Key (priEK A) ∈ parts (knows Spy evs)) = (A ∈ bad)"
apply (erule ns_public.induct, auto)
done

lemma Spy_analz_priEK [simp]:
  "evs ∈ ns_public ⟹ (Key (priEK A) ∈ analz (knows Spy evs)) = (A ∈ bad)"
apply auto
done
```

These guarantees signify that the protocol never prescribes its participants to send their private keys in the network traffic as message components. More precisely, the first lemma states that a private key appears as component of a message in the traffic if and only if the owner of the key is compromised. Conversely, when the owner is compromised, the spy knows her key since the beginning, by definition of *initState* (Sect. 3.3). The second lemma expresses a more stringent property in terms of *analz*. The first proof terminates by one command that applies two methods: induction over the protocol and the *auto* method. The second proof is even shorter.

These proofs are rather cryptic, and our goal precisely is to detail them, also to investigate the irrelevance of *spy_analz*, which in fact is only apparent. Hence, we restart them step by step, beginning with lemma *Spy_see_priEK*. We replace the *auto* method by simplification:

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

Induction over the protocol is a customary start. It is launched by applying the *erule* proof method with a lemma that Isabelle builds automatically for each inductively defined set, in this case called *ns_public.induct*. That lemma is in the form of an elimination rule, hence the proof method. The reader may view it by prefixing it with command **thm**. Intuitively, the lemma implements the check of the conjecture against all rules defining the protocol model. After simplification, all subgoals are closed, except the one corresponding to the *Fake* case, shown in Fig. 2. The inductive style of the proof is clear, as the equality in the preconditions shows the inductive hypothesis, while the one in the postconditions is the extended case to be proven. It may be useful to recall that an equality represents a logical co-implication, hence if one fact is available then also the other one holds.

The proof continues through the following script:

```
apply (cases "A ∈ bad")
apply blast
```

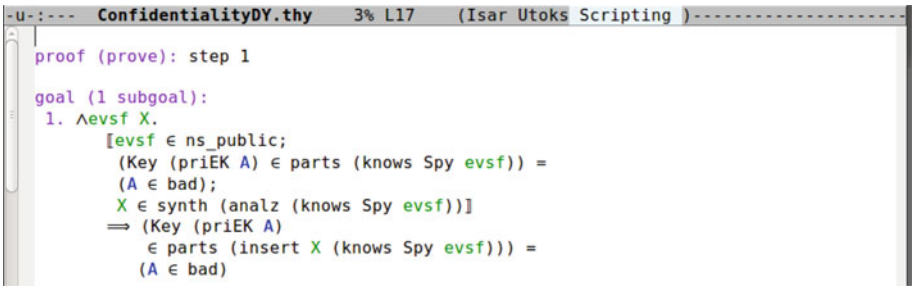

Fig. 2. Proving the *Fake* subgoal of *Spy_see_priEK*: level 1

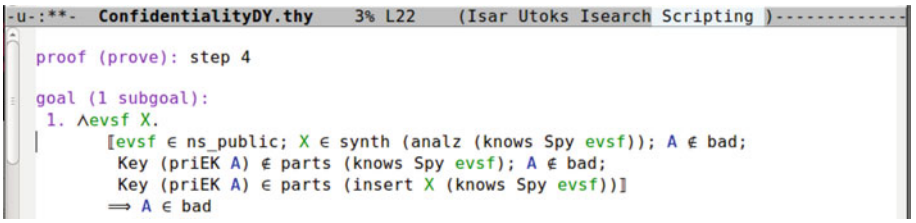

Fig. 3. Proving the *Fake* subgoal of *Spy_see_priEK*: level 4

The first command applies a case analysis based upon `A` being compromised or not, splitting up the subgoal into two subgoals. If `A` is compromised, then the inductive hypothesis and the monotonicity of `parts` would close the subgoal. However, the `blast` method invoked by the second command suffices without installation of further lemmas about `parts`. In fact, they are not needed thanks to a simpler proof: because `A` is compromised, her private key is known to the spy since the beginning (Sect. 3.3), which produces the necessary contradiction without additional lemmas. To distill this line of reasoning, the reader may want to polish the subgoal additionally before blasting it. For example, theorem `ccontr: (¬ P ⟹ False) ⟹ P`, which can be inspected by launching **thm** `ccontr`, is a useful introduction rule. It can be invoked by command **apply** `(rule ccontr)`. Followed by another application of the simplifier by **apply** `simp`, it would leave the subgoal with `False` as a postcondition and facts `Key (priEK A) ∉ parts (insert X (knows Spy evsf))` and `A ∈ bad` in the preconditions, a contradiction.

The remaining subgoal, which concerns the case when `A` is not compromised, can be clarified by applying the appropriate method:

apply `clarify`

The outcome becomes rather compact, and is depicted in Fig. 3. It is not obvious how to derive a contradiction here. Because of the monotonicity of `parts`, it seems perfectly plausible that `A`'s private key appears in no messages in the traffic on `evsf` but does appear in some message if that traffic is extended with one of the spy's fakes `X`. Here, the lemma `Fake_parts_insert` seen above is useful, and in particular its version resolved with the canonical subset destruction rule `subsetD: ⟦A ⊆ B; c ∈ A⟧ ⟹ c ∈ B`. It can be inspected customarily by:

thm `Fake_parts_insert [THEN subsetD]:`
`⟦X1 ∈ synth (analz H1); c ∈ parts (insert X1 H1)⟧ ⟹ c ∈ synth (analz H1) ∪ parts H1`

It can then be applied to the current proof state by:

apply `(drule Fake_parts_insert [THEN subsetD], simp)`

The appeal to the simplifier helps binding the variables—for example, variable `Y` must be instantiated as `Key (priEK A)`—leading to Fig. 4. This subgoal can be closed as follows:

apply `(blast dest:analz_into_parts)`

The command applies `blast` with the extra lemma `analz_into_parts` mentioned above (Sect. 3.1). More in detail, blasting applies the definition of set union, introducing a disjunction whose second disjunct already leads to a contradiction along with the second precondition: `A`'s private key cannot at the same time appear and fail to appear in the traffic on `evsf`. The first disjunct, `Key (priEK A) ∈ synth(analz(knows Spy evsf))`, is simplified by stripping off the application of `synth` because no private key can be synthesized by definition of `synth` (Sect. 3.1). An application of lemma `analz_into_parts` produces also in this subgoal an instance of the contradiction just seen about `A`'s private key, and the whole proof terminates.

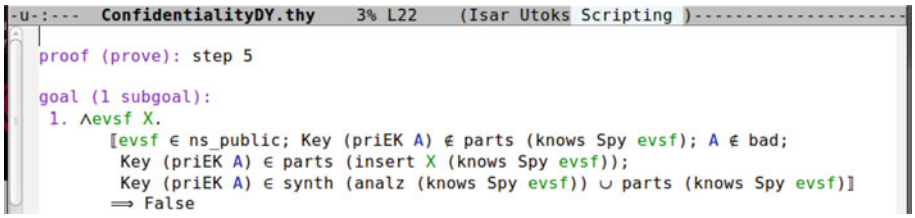

Fig. 4. Proving the *Fake* subgoal of *Spy_see_priEK*: level 5

At this stage, the last subgoal, depicted in Fig. 4, has been closed, and Isabelle prints the corresponding message *No subgoals!*. By the axiom of induction, it can be stated that the proof is complete. This is done by command:

done

As a result, *Spy_see_priEK* is stored as a lemma for future invocation.

Also the proof of *Spy_analz_priEK* deserves some extra insight. It is a simple corollary of the previous lemma. A rather intelligible proof script would be:

apply *(auto simp: Spy_see_priEK dest: analz_into_parts)*

The *auto* method alone would automatically solve the right-to-left implication by definitions of *initState* and *analz*. By installing the rewriting rule just proven and a recurrent lemma, it also solves the other implication.

After a detailed explanation of these confidentiality proofs, it should be clearer why they seem to be the only confidentiality proofs terminating with no need to invoke *spy_analz*. The *auto* method can solve the *Fake* subgoal in the repository proof of *Spy_see_priEK* because lemma *Fake_parts_insert* was stably installed before. By contrast, our proof explicitly invokes all necessary lemmas. It becomes clear that *Fake_parts_insert* is necessary in any case. In fact, also *spy_analz* could then terminate the proof of *Spy_see_priEK* after the first command, that is before the case analysis, as the reader may easily try.

It may also be insightful to prove *Spy_analz_priEK* directly [Bel12]. The same proof as that of the other lemma only requires two adaptations. One is to launch the simplifier by appealing to a specific set of rewriting rules for the symbolic evaluation of *analz* over the union of sets of keys, called *analz_image_freshK_simps*. This set, which can be inspected routinely by means of the **thm** command, serves to close the *Nil* case. The other adaptation is to replace *Fake_parts_insert* with *Fake_analz_insert*, which will be discussed in the next section. It becomes clear that the reason why *spy_analz* is apparently unnecessary for the confidentiality proofs about static secrets is the choice of syntactically streamlining the proof scripts in the Isabelle repository. By contrast, we have seen that the actual reasoning behind *spy_analz* is necessary. It must be stressed that the full line of reasoning just described applies to all static secrets of all protocols analysed so far [URL11b].

We remark that, coherently with our didactic aims, we have avoided the stable installation of lemmas in the respective sets that the simplifier or the automatic provers can access. These installations are normally done by a **declare** command such as **declare** *Spy_see_priEK [simp]*, or **declare** *analz_into_parts [dest]*. We stress that in our experience the reduced readability of the proof script pays the user back in terms of increased proof comprehension.

4.2. Dynamic secrets

The treatment continues with a detailed account on the confidentiality proof of the initiator's nonce in the Needham–Schroeder protocol [URL11b], a line of reasoning can be reused for all other dynamic secrets:

lemma *Spy_not_see_NA:*
"⟦Says A B (Crypt(pubEK B) {|Nonce NA, Agent A|}) ∈ set evs;
A ∉ bad; B ∉ bad; evs ∈ ns_public⟧
⟹ Nonce NA ∉ analz (knows Spy evs)"
apply *(erule rev_mp, erule ns_public.induct)*
apply *simp_all*

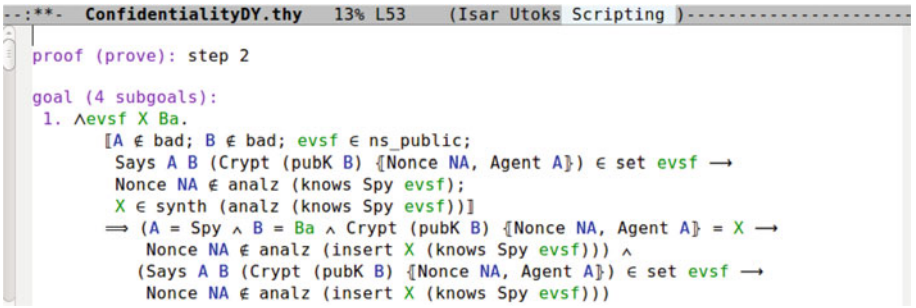

Fig. 5. Proving *Spy_not_see_NA*: level 2

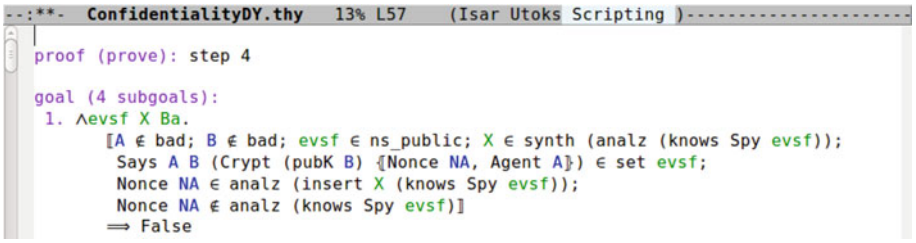

Fig. 6. Proving *Spy_not_see_NA*: level 4

The theorem relies on appearance of the event formalising the first protocol step on a generic trace *evs* of the protocol model *ns_public*. This binds the nonce *NA*. If both agents *A* and *B* are uncompromised, then *NA* is confidential on the trace.

The proof begins with a command that brings the main precondition in the inductive hypothesis. It applies the reverse *modus ponens* lemma, *rev_mp:* ⟦*P*; *P* ⟶ *Q*⟧ ⟹ *Q*, often used below. Then, it launches induction over the protocol model. The second command simplifies the entire proof state, closing the subgoals where term-rewriting suffices. Figure 5 isolates the subgoal caused by the *Fake* rule.

The repository proof terminates with another application of the *spy_analz* method. A step-by-step alternative proof is studied here. It can be observed that the postcondition of the subgoal is a conjunction whose first conjunct, itself an implication, holds because its precondition is false: it cannot be *A=Spy* because of the subgoal precondition *A∉ bad*. So, the subgoal can be split up into two subgoals, respectively with one conjunct as a postcondition, appropriately using the introduction rule *conjI:* ⟦*P*; *Q*⟧ ⟹ *P* ∧ *Q*. This lets us close the first one immediately:

apply *(rule conjI)*
apply *clarify*

It is worth remarking that the first conjunct was the outcome of simplification of the case when the *Says* event in the inductive hypothesis corresponds with the event introduced by the *Fake* rule—hence the three equalities. It is remarkable that this subgoal can be closed by contradiction due to the strong assumption *A∉ bad*. This indicates that the guarantee certainly does not apply to a compromised agent, as it is typical in DY reasoning. It shall be seen (Sect. 6) that removing this assumption, which makes sense against a different threat model, complicates the line of reasoning significantly, requiring novel proof strategies.

Then, we clarify the remaining half of the subgoal, which describes the case when the *Says* event in the inductive hypothesis already appeared in the given trace *evsf*:

apply *clarify*

This produces the subgoal in Fig. 6, which is small but not simple. The hidden core of *spy_analz* can help, the pair of lemmas *Fake_parts_insert* and *Fake_analz_insert*. Having seen the former above, we inspect the latter here by the usual command:

thm *Fake_analz_insert:*
X ∈ *synth(analz G)* ⟹ *analz(insert X H)* ⊆ *synth(analz G)* ∪ *analz(G* ∪ *H)*

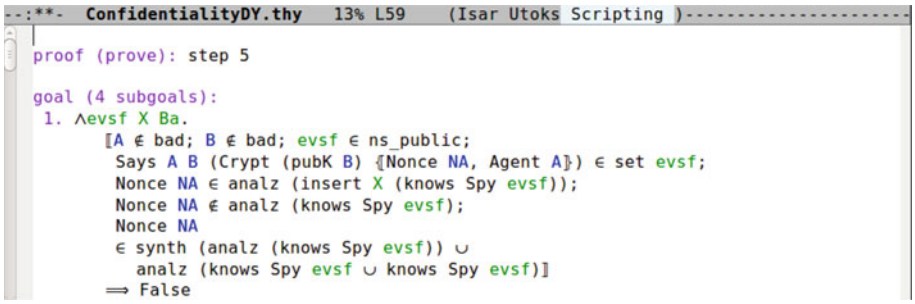

Fig. 7. Proving *Spy_not_see_NA*: level 5

Its version resolved with the subset destruction rule, which is more useful here, may also seem (slightly) more intelligible. Its contribution remains the elimination of a term involving *analz* over a message set expanded with the insertion of another element, which is usually problematic for symbolic evaluation due to the definition of the message operator. The postcondition is rather clean, though long:

thm *Fake_analz_insert [THEN subsetD]:*
⟦X1 ∈ synth (analz G1); c ∈ analz (insert X1 H1)⟧ ⟹ c ∈ synth (analz G1) ∪ analz (G1 ∪ H1)

This theorem can be applied to the current proof state by:

apply *(drule Fake_analz_insert [THEN subsetD], simp)*

The appeal to the simplifier helps binding the variables—for example, variable *Y* must be instantiated as *Nonce NA*—producing the result shown in Fig. 7. It can be seen that *Fake_analz_insert* is more general than what is actually needed here, both its variables *G* and *H* being instantiated as *knows Spy evs*. However, this feature helps closing the proof, but will no longer be available against a different threat model (Sect. 6), ruling out the applicability of *Fake_analz_insert* in that case. Now we launch:

apply *simp*

It closes the subgoal. More precisely, the internal occurrence of the union operator is removed because it is applied to the same sets; the external occurrence of the union operator then simplifies away as a disjunction. Because no nonce can be synthesized, fact *Nonce NA ∈ synth(analz(knows Spy evsf))* is further simplified by stripping off the application of *synth*. Both disjuncts produce a contradiction with the last-but-one precondition.

The other subgoals can be closed easily by invoking two lemmas (from *NS_Public_Bad.thy*) establishing respectively that a nonce is never reissued by its issuer or by the other peer. The *+* suffix tells Isabelle to apply the method to the entire proof state:

apply *(blast dest: unique_NA intro: no_nonce_NS1_NS2)+*
done

We remark once more that the description just given is not limited to a specific theorem, but applies to all confidentiality proofs about dynamic secrets in the Isabelle repository [URL11b]. We conducted the analogous study of the other nonce, the one that the responder invents, at the same level of detail [Bel12] but omit it from this presentation.

5. Confidentiality against Dolev–Yao: novel study

Also the study discussed in this section is part of our theory file *ConfidentialityDY.thy* [Bel12].

At this stage, the reader has seen the full details of the study of confidentiality that is currently available within the Inductive Method. These have been given by spelling out two paradigmatic theorems. In fact, we remark that *Spy_see_priEK*, or its analogue for the other static secrets, can be found towards the beginning of each theory file for security protocols coming with the repository. Likewise, we have seen that the major effort in

proving *Spy_not_see_NA* derives from the *Fake* case, that is from having to verify the stated conjecture against the formalised threat model. By contrast, the other cases are always more intuitive. The repository will confirm that these comments apply to all dynamic secrets studied so far, except those of the SET protocol [BMP06].

However, out of personal experience not only with research on the Inductive Method but also with teaching it, we believe that a full account on confidentiality must be broader than what is currently available. We then motivate, define and meet two inter-related requirements for our formal analysis. Teaching practice indicates that they are important for the reader's understanding both of the method and of the protocols that they intend to study.

Let us focus more on the threat model. Although it is clear where it plays its role in the protocol specification, as vastly studied above, it is somewhat hard-coded in the treatment—in fact, we shall see that modifying it requires changing the full underlying theories (Sect. 6). This feature demands the analyst's care in clarifying what sort of confidentiality guarantees can be proved due exactly to the threat model, regardless of the specific protocol under analysis: for example the DY spy cannot guess someone else's static secret. These guarantees can be interpreted as to implicitly express the limitations of the threat model. But there are guarantees that can be proved specifically of certain security protocols, such as the confidentiality of a nonce issued during a session of the protocol. These signify and confirm that protocols attempt to additionally counter the threat model in order to obtain specific goals. In particular, with the guarantees seen above (Sect. 4), one must note that the generic trace that is the main subject of each theorem is in fact constrained to belong to the protocol model *ns_public* to realise that those guarantees are not protocol-independent. The reader then may wonder: can they, and to what extent, be made protocol-independent? We therefore define the following requirement for our analysis.

1. The reader should be facilitated in distinguishing what guarantees can be proved against the threat model *in general*, that is independently from the studied protocols, from those that only hold of a particular protocol.

A related issue is how to design confidentiality guarantees specifically for static or for dynamic secrets. Because static secrets exist prior to executing any protocol, one may think that they should be studied protocol-independently. However, this paper discussed *Spy_see_priEK*, a relevant protocol-dependent guarantee about static secrets. Likewise, because dynamic secrets are only invented at runtime, one may think that they can only be studied protocol-dependently. But the Isabelle repository [URL11b] features all four combinations of guarantees and secrets. The reader may then wonder: protocol-independent guarantees are more appealing, so why do we need protocol-dependent confidentiality guarantees at all? We therefore define the following requirement for our analysis.

2. The reader should be facilitated in understanding that relevant confidentiality guarantees on static secrets are not necessarily protocol-independent and that relevant confidentiality guarantees on dynamic secrets are not necessarily protocol-dependent, and also facilitated in evaluating their strength.

The second requirement appears to specify the first for what pertains to confidentiality. The sequel of this section demonstrates how to meet them. We anticipate that protocol-independent guarantees can be formulated profitably but may turn out weaker than protocol-dependent ones.

It can be conjectured that all static secrets are treated in the same fashion, and hence may enjoy the same guarantees. Static secrets can be defined as a function *staticSecret* that takes an agent as parameter and yields their long-term secrets. Currently, three such secrets are formalised, the private encryption key, the private signature key and a symmetric key shared with the server. The following syntax seems self-explanatory, starting with a declaration line followed by a simple definition line:

```
definition staticSecret :: "agent ⇒ msg set" where
 [simp]: "staticSecret A ≡ {Key (priEK A), Key (priSK A), Key (shrK A)}"
```

It can be noted that the function definition, which Isabelle automatically names *staticSecret_def*, is installed in the simplifier's set of available rules by means of the traditional prefix *[simp]*. This function will be useful to generalise to all static secrets those guarantees that have been stated about a single long-term key so far.

A glimpse at the graphical interface was provided in the previous section through a few screenshots. Here, we only use inline subgoal presentation to save space. Subgoals are mechanically translated into LaTeX code by using the antiquotation mechanism [Wen11, p. 41].

5.1. Protocol-independent study

In order to meet the first of the two analysis requirements set above, we begin with a protocol-independent study of confidentiality. To conduct it comprehensively, the analyst could enumerate the events that may lead to the spy's violation of confidentiality of the message. In practice, the study can be conducted by assuming the spy's knowledge of a message and enumerating the events that this fact determines. While the enumeration may help intuition, the contrapositive of such theorem amounts to a guarantee of confidentiality. Let us begin with static secrets by conjecturing the following theorem:

```
lemma staticSecret_parts_Spy:
"⟦m ∈ parts (knows Spy evs); m ∈ staticSecret A⟧ ⟹
 A ∈ bad ∨
 (∃C B X. Says C B X ∈ set evs ∧ m ∈ parts{X}) ∨
 (∃C Y. Notes C Y ∈ set evs ∧ C ∈ bad ∧ m ∈ parts{Y})"
```

The two preconditions indicate a generic static secret of `A`'s known to the spy. Knowledge here is expressed in terms of `parts`, rather than in terms of `analz` as it is more customary, because the contrapositive becomes stronger. The statement says that if the spy knows some agent's static secret, then either that agent is bad, or someone sent a message featuring the static secret, or someone who is bad noted it down. Notably, it relies on a generic, that is unconstrained, trace and therefore is protocol-independent.

We begin the proof with the following four commands, explained later:

```
apply (erule rev_mp)
apply (induct_tac "evs")
apply force
apply (induct_tac "a")
```

The main precondition, the spy's knowledge of `m`, is first brought into the inductive formula. Then, induction is applied over the trace `evs`, resulting in two subgoals, one for the empty trace, one for an inductive trace whose head event is a generic `a`. The former can be closed by the `force` method, which appeals to `staticSecret_def`. The reader can empirically verify that appeal by command **apply** `(force simp del:staticSecret_def)`, which fails, instead of **apply** `force`. The remaining command treats the inductive trace by expanding the event `a` into its three possible options, that is a `Says`, a `Gets` or a `Notes` event.

The first of the three resulting subgoals is shown here. It concerns the `Says` event:

```
1. ⋀a list agent1 agent2 msg.
      ⟦m ∈ staticSecret A;
       m ∈ parts (knows Spy list) ⟶
       A ∈ bad ∨
       (∃C B X. Says C B X ∈ set list ∧ m ∈ parts {X}) ∨
       (∃C Y. Notes C Y ∈ set list ∧ C ∈ bad ∧ m ∈ parts {Y})⟧
      ⟹ m ∈ parts (knows Spy (Says agent1 agent2 msg # list)) ⟶
          A ∈ bad ∨
          (∃C B X.
              Says C B X ∈ set (Says agent1 agent2 msg # list) ∧
              m ∈ parts {X}) ∨
          (∃C Y. Notes C Y ∈ set (Says agent1 agent2 msg # list) ∧
                 C ∈ bad ∧ m ∈ parts {Y})
```

The inductive formula among the preconditions, and the thesis featuring a `Says agent1 agent2 msg` event can be spotted. It is convenient to push the precondition of the postcondition among the preconditions of the theorem, which corresponds to lifting the precondition from the object level, the protocol, to the meta level, the proof. It can be done via an appeal to theorem `impI: (P ⟹ Q) ⟹ P ⟶ Q` by resolution with the subgoal:

```
apply (rule impI)
apply simp
```

The canonical simplifier invocation symbolically evaluates term `knows Spy (Says agent1 agent2 msg # list)` extracting `msg`. Also, the other two set memberships featuring the `Says` event are evaluated:

```
1. ⋀list agent1 agent2 msg.
     ⟦m = Key (priEK A) ∨ m = Key (priSK A) ∨ m = Key (shrK A);
      m ∈ parts (knows Spy list) ⟶
      A ∈ bad ∨
      (∃C B X. Says C B X ∈ set list ∧ m ∈ parts {X}) ∨
      (∃C Y. Notes C Y ∈ set list ∧ C ∈ bad ∧ m ∈ parts {Y});
      m ∈ parts (insert msg (knows Spy list))⟧
     ⟹ A ∈ bad ∨
         (∃C B X.
             (C = agent1 ∧ B = agent2 ∧ X = msg ∨ Says C B X ∈ set list) ∧
             m ∈ parts {X}) ∨
         (∃C Y. Notes C Y ∈ set list ∧ C ∈ bad ∧ m ∈ parts {Y})
```

Fact `m ∈ parts (insert msg (knows Spy list))` can be simplified further by a manipulation of a basic result, which can be easily viewed:

```
thm parts_insert [THEN equalityD1, THEN subsetD]
    c ∈ parts (insert X2 H2) ⟹ c ∈ parts {X2} ∪ parts H2
```

It can applied as a destruction rule:

```
apply (drule parts_insert [THEN equalityD1, THEN subsetD])
```

The resulting proof state is still of three subgoals. The first becomes:

```
1. ⋀list agent1 agent2 msg.
     ⟦m = Key (priEK A) ∨ m = Key (priSK A) ∨ m = Key (shrK A);
      m ∈ parts (knows Spy list) ⟶
      A ∈ bad ∨
      (∃C B X. Says C B X ∈ set list ∧ m ∈ parts {X}) ∨
      (∃C Y. Notes C Y ∈ set list ∧ C ∈ bad ∧ m ∈ parts {Y});
      m ∈ parts {msg} ∪ parts (knows Spy list)⟧
     ⟹ A ∈ bad ∨
         (∃C B X.
             (C = agent1 ∧ B = agent2 ∧ X = msg ∨ Says C B X ∈ set list) ∧
             m ∈ parts {X}) ∨
         (∃C Y. Notes C Y ∈ set list ∧ C ∈ bad ∧ m ∈ parts {Y})
```

The last command has reduced the last of the available preconditions to `m ∈ parts {msg} ∪ parts (knows Spy list)`. This implies that either `m ∈ parts {msg}` or `m ∈ parts (knows Spy list)`. In the first case, because `msg` is meta-universally quantified, it can trigger the inductive hypothesis, which is the second of the preconditions, hence the thesis. In the second case, the inductive hypothesis is triggered straightforwardly, hence the thesis. The theorem prover conducts this classical reasoning in one single command, which closes the subgoal:

```
apply blast
```

Alternatively, the reader may follow the line of reasoning more closely by launching **apply** `(simp only: Un_iff)`, which applies the set union definition transforming the precondition in the disjunction `m ∈ parts {msg} ∨ m ∈ parts (knows Spy list)`. Then, this can then be split up into two facts, hence two subgoals, by **apply** `(erule disjE)`. Each subgoal can then be blasted. We remind that inspecting the content of a theorem by prefixing its name with command **thm** may be useful to evaluate whether to apply the theorem.

The remaining proof state has two subgoals. The first, omitted here, concerns the `Gets` event. Because this event does not modify the spy's knowledge (Sect. 3.2), term `knows Spy (Gets agent msg # list)` gets evaluated as `knows Spy list`, hence the inductive hypothesis closes the subgoal. The simplifier can do this:

```
apply simp
```

The proof is now left with the single subgoal concerning the `Notes` event. Because this event extends the spy's knowledge only in case of bad agents (Sect. 3.2), the same proofs for the previous cases can be combined by an additional case study:

```
apply (rule impI)
apply simp
apply (case_tac "agent∉bad")
```

The proof state now exhibits the case of an agent who is not bad. Because this does not change the spy's knowledge, symbolic evaluation is straightforward as in the *Gets* case, so that the inductive hypothesis closes the subgoal. However, in this case we need both simplification, to reduce an if-then-else expression deriving from *knows_Cons* (Sect. 3.2), and classical reasoning to apply the inductive hypothesis:

```
apply simp
apply blast
```

The remaining case, pertaining to a bad agent, is identical to that for the *Says* case, and the proof terminates:

```
apply simp
apply (drule parts_insert [THEN equalityD1, THEN subsetD])
apply blast
done
```

Having proved this theorem, we have built evidence of how a static secret can be leaked. As observed, taking the contrapositive of the theorem gives us a strong confidentiality guarantee on static secrets. Note that the theorem cannot be proved using *analz* rather than *parts* in the postconditions. For example, if *Says C B X* appears in a trace with *X = Crypt (pubEK B) m*, it follows that *m* $\notin$ *analz{X}*.

We now move on to investigating whether this guarantee can be extended to generic messages, therefore including dynamic secrets. This is in line with the second of our analysis requirements set above. We begin with an empirical analysis. What happens if we remove precondition *m* $\in$ *staticSecret A* from the preconditions of the theorem just seen? To hold over a generic trace, the conjecture we are building should account in particular for the empty trace. With static secrets, it was correctly conjectured that their owner were bad. With generic messages, no restriction seems plausible, and thus we conjecture that the spy knew them since the beginning. Here is the formalisation of this conjecture:

```
lemma secret_parts_Spy:
"m ∈ parts (knows Spy evs)  ⟹
 m ∈ initState Spy ∨
 (∃C B X. Says C B X ∈ set evs ∧ m ∈ parts{X}) ∨
 (∃C Y. Notes C Y ∈ set evs ∧ C ∈ bad ∧ m ∈ parts{Y})"
```

It may seem surprising that this can be proved by the same script as that for the previous theorem. However, the reason is clear: in line with the intuition just given, the first disjunct of the postcondition accounts for the case when the stated trace is the empty one. The corresponding subgoal appears after applying induction over *evs*. It can be closed by the *force* method, as seen, but in fact also by simply invoking the simplifier through *simp*. This is because rewritings are fewer this time: term *parts (knows Spy [])* gets rewritten as *parts (initState Spy)*, then as *initState Spy*.

Also, the proof can be followed more easily [Bel12] by replacing each invocation of the simplifier in the sequel of the script with **apply** *(simp del: initState_Spy)*. This will avoid the symbolic evaluation of the spy's initial state into a (large) set of keys, keeping term *initState Spy* folded and hence the proof state more compact.

Once more, the contrapositive of this theorem gives a valuable confidentiality guarantee. One of its implications is that if an agent invents a dynamic secret and never sends or notes it down inside a message, because that secret certainly was not in the spy's initial knowledge, then it is confidential. This is the first protocol-independent confidentiality guarantee about a generic message, and in particular about a dynamic secret. As mentioned at the beginning of the section, it also expresses a boundary beyond which the threat model can never go, hence helps the reader's mental picture of it.

Arguably, we also need confidentiality guarantees for secrets that are sent over the network. While this clearly rules out the applicability of the contrapositive of *secret_parts_Spy* as it violates one of its preconditions, it also naturally imposes a protocol-dependent study: when secrets are sent over the network, they are in fact sent according to specific rules that are supposed to keep them *secure*. Those rules form an actual protocol.

Before moving on to protocol-dependent study, we observe for completeness that there are versions of these theorems where the main precondition is expressed in terms of *analz* rather than *parts*, proved thanks to *analz_into_parts*. Omitted here, they are called respectively *staticSecret_analz_Spy* and *secret_analz_Spy*, and can be found in our theory file *ConfidentialityDY.thy* [Bel12]. However, their contrapositives become weaker. Also, the converse of *staticSecret_parts_Spy* does not hold because not every component of every message that is sent or noted is a static secret, while the converse of *secret_parts_Spy* does. Here omitted, it can be found as *secret_parts_Spy_converse* in our theory.

5.2. Protocol-dependent study

A premise of the study that we are going to begin here is that *Spy_see_priEK* can be easily generalised to all static secrets:

```
lemma NS_Spy_see_staticSecret:
 "⟦m ∈ staticSecret A; evs ∈ ns_public⟧ ⟹
   m ∈ parts(knows Spy evs) = (A ∈ bad)"
```

Not surprisingly, the proof script is almost identical to that of *Spy_see_priEK*, except for what is explained later. The strategy taken is the traditional, *direct* launch of induction over the very protocol being analysed, followed by simplification and then extra care to close the *Fake* subgoal:

```
apply (erule ns_public.induct, simp_all)
prefer 2
apply (cases "A:bad")
apply blast
apply clarify
apply (drule Fake_parts_insert [THEN subsetD], simp)
```

The first subgoal in the proof state exhibits the *Fake* case, which shows much similarity with the corresponding subgoal of the other theorem (Fig. 4) except for the generality of *m*:

```
1. ⋀evsf X.
      ⟦m = Key (priEK A) ∨ m = Key (priSK A) ∨ m = Key (shrK A);
       evsf ∈ ns_public; A ∉ bad; m ∉ parts (knows Spy evsf);
       m ∈ parts (insert X (knows Spy evsf));
       m ∈ synth (analz (knows Spy evsf)) ∪ parts (knows Spy evsf)⟧
      ⟹ A ∈ bad
```

The few syntactical differences with the proof script of *Spy_see_priEK* are easy to justify. Due to the necessity to expand *staticSecret* and to reason about the three deriving opportunities, the *simp_all* method closes no subgoal. All can be closed by *force*, except for the *Fake* subgoal, which requires the special treatment spelled out above (Sect. 4). Hence the use of command **prefer** to put the *Fake* subgoal at the beginning of the proof state, then treat it customarily, and finally close the entire proof by means of the *force* method applied to all subgoals:

```
apply (blast dest: analz_into_parts)
apply force+
done
```

It can be appreciated that the guarantees of confidentiality existing prior to our study [URL11b] are all protocol-dependent (Sect. 4). To clarify the relation between them and the guarantees just presented (Sect. 5.1), there still is protocol-dependent analysis to conduct. Precisely, its aim is to assess whether and to what extent the protocol-independent guarantee *staticSecret_parts_Spy* can be specialised, by means of protocol-dependent subsidiary lemmas, to resemble or equal *NS_Spy_see_staticSecret*. We address this alternative proof strategy, which is detailed in the following, as *specialisation*. It is not a priori obvious whether the specialisation proof strategy may yield a guarantee that is stronger, weaker or equal to the outcome of the direct proof strategy; further, in case of equal guarantee, it is not obvious what strategy would be quicker. Addressing these issues is the purpose of the sequel of this section.

As for dynamic secrets, the specialisation strategy would fail due to previous observations (Sect. 5.1): pinpointing a dynamic secret that is used in a protocol falsifies the preconditions of the contrapositive of *secret_parts_Spy*. This means that the specialisation proof strategy cannot succeed in proving *Spy_not_see_NA* or any similar guarantee about the dynamic secret.

We begin the assessment of the specialisation proof strategy for static secrets by observing the postcondition of *staticSecret_parts_Spy*. It is a disjunction of three facts, indicating all possible alternatives insisting on the generality of the stated trace *evs*. The key question here is whether those alternatives can be reduced when the stated trace is a particular one, that is one belonging to the protocol model *ns_public* (Sect. 3.4). The first disjunct can never be ruled out because of the existing threat model: the spy sees bad agents' static secrets since the beginning, prior to any protocol session. By inspecting the definition of constant *ns_public*, it can be seen that no *Notes* event are ever introduced. This is confirmed via the following lemma:

```
lemma NS_no_Notes:
 "evs ∈ ns_public ⟹ Notes A X ∉ set evs"
```

Not surprisingly, its proof is straightforward, as it applies induction over the protocol and then simplifies all subgoals. We recall that the practical use of induction is to check if all rules defining the protocol preserve a given conjecture:

```
apply (erule ns_public.induct)
apply simp_all
done
```

In consequence, `staticSecret_parts_Spy` can be specialised over our example protocol. The following theorem takes a trace of the protocol and concludes without the disjunct corresponding to the `Notes` event:

```
lemma NS_staticSecret_parts_Spy_weak:
"⟦m ∈ parts (knows Spy evs); m ∈ staticSecret A;
  evs ∈ ns_public⟧ ⟹ A ∈ bad ∨
 (∃C B X. Says C B X ∈ set evs ∧ m ∈ parts{X})"
```

As it can be expected, its proof combines the protocol-independent `staticSecret_parts_Spy` with `NS_no_Notes` by the `blast` method:

```
apply (blast dest: staticSecret_parts_Spy NS_no_Notes)
done
```

This guarantee illustrates the line of reasoning we are following. However, we aim at strengthening it further, having `NS_Spy_see_staticSecret` in mind. Therefore, we must study specific conditions of the example protocol upon which a `Says` event can be ruled out. It can be proved, coherently with the threat model, that no agent except the spy would include their static secrets in any message they send out in the traffic:

```
lemma NS_Says_staticSecret:
 "⟦Says A B X ∈ set evs; m ∈ staticSecret C; m ∈ parts{X};
   evs ∈ ns_public⟧ ⟹ A=Spy"
```

Also this guarantee is easy to prove via induction, this time followed by the `force` method to resolve the matching between a static secret and the message that each protocol rule introduces:

```
apply (erule rev_mp)
apply (erule ns_public.induct)
apply force+
done
```

A way to specialise `NS_staticSecret_parts_Spy_weak` further towards `NS_Spy_see_staticSecret` would be to link the second disjoint of its postcondition to the first one, which we understood can never be ruled out against the current threat model. To do this, we need a guarantee linking a `Says` event that introduces a static secret to the owner of the static secret. Therefore, `NS_Says_staticSecret` cannot suffice because it links a `Says` event that introduces a static secret to the event originator, the spy, not to the owner of the static secret. However, we shall see that it will be useful.

Things are now clear to conjecture what would be needed:

```
lemma NS_Says_Spy_staticSecret:
 "⟦Says Spy B X ∈ set evs; m ∈ parts{X};
   m ∈ staticSecret A; evs ∈ ns_public⟧ ⟹ A ∈ bad"
```

This establishes a fact about the owner of the static secret that the stated event features. For didactic reasons, its proof is deferred to the end of the section.

The two guarantees just given help us specialise `NS_staticSecret_parts_Spy_weak` further, precisely by tracking the second disjunct of the postcondition down to the first one. So, the specialisation continues as follows:

```
lemma NS_staticSecret_parts_Spy:
"⟦m ∈ parts (knows Spy evs); m ∈ staticSecret A;
  evs ∈ ns_public⟧ ⟹ A ∈ bad"
```

The proof can be conducted in a forward style. It begins by appealing to the protocol-independent guarantee that is being specialised. Because the guarantee only applies to a static secret, it builds a subgoal aimed at proving `m ∈ staticSecret A`. That fact is available among the current preconditions, so the subgoal can be closed by assumption:

```
apply (drule staticSecret_parts_Spy)
apply assumption
```

The postcondition of the guarantee just appealed to is now available among the current preconditions, as the proof state clearly shows:

```
1. ⟦m ∈ staticSecret A; evs ∈ ns_public;
    A ∈ bad ∨
    (∃C B X. Says C B X ∈ set evs ∧ m ∈ parts {X}) ∨
    (∃C Y. Notes C Y ∈ set evs ∧ C ∈ bad ∧ m ∈ parts {Y})⟧
   ⟹ A ∈ bad
```

Each disjunct in the preconditions demands proving the postcondition. The first disjunct can be isolated and the corresponding subgoal closed as follows:

```
apply (erule disjE)
apply assumption
```

The next disjunct corresponds to a `Says` event. It can be isolated customarily, and is then useful to strip off the existential quantification by instantiating the three variables:

```
apply (erule disjE)
apply (erule exE)+
```

The remaining proof state is rather intelligible:

```
1. ⋀C B X.
     ⟦m ∈ staticSecret A; evs ∈ ns_public;
      Says C B X ∈ set evs ∧ m ∈ parts {X}⟧
     ⟹ A ∈ bad
2. ⟦m ∈ staticSecret A; evs ∈ ns_public;
    ∃C Y. Notes C Y ∈ set evs ∧ C ∈ bad ∧ m ∈ parts {Y}⟧
   ⟹ A ∈ bad
```

The first subgoal can be tackled by a case study over `C`. If it is the spy, then `NS_Says_Spy_staticSecret` closes it, otherwise `NS_Says_staticSecret` closes the subgoal. The final subgoal, which concerns a `Notes` event, can be closed in the same fashion, by appealing to `NS_no_Notes`. The proof terminates:

```
apply (case_tac "C=Spy")
apply (blast dest: NS_Says_Spy_staticSecret)
apply (blast dest: NS_Says_staticSecret)
apply (blast dest: NS_no_Notes)
done
```

Compared to the target guarantee `NS_Spy_see_staticSecret`, which is an equality, the guarantee just studied expresses its non-trivial implication. In fact, the target guarantee can be proved by the single command **apply** `(force dest: NS_staticSecret_parts_Spy)`, which the reader can try.

Our study has achieved its aim. The specialisation proof strategy reaches the same confidentiality guarantee on static secrets obtained through the direct strategy. However, to evaluate whether it is quicker, hence preferable, we must complete the presentation with the proof of `NS_Says_Spy_staticSecret`, whose evaluation may be surprising (Sect. 5.3).

Incidentally, the same one-line proof script used above can be used to prove a purely set-theoretic version that hides the set candidate. Arguably, the subset operator has higher priority than equality:

```
lemma NS_staticSecret_subset_parts_knows_Spy:
"evs ∈ ns_public ⟹
 staticSecret A ⊆ parts (knows Spy evs) = (A ∈ bad)"
apply (force dest: NS_staticSecret_parts_Spy)
done
```

For completeness of the presentation, we note that there are versions of the last two guarantees [Bel12] where `analz` replaces `parts`, here omitted.

5.3. Evaluation

In summary, it was found that `NS_Spy_see_staticSecret`, which generalises `Spy_see_priEK` to any static secret, can be proved both using the direct and the specialisation proof strategy. However, the two strategies overlap at some point, as we set out to explain here.

We begin by presenting a subsidiary lemma establishing that a static secret cannot be synthesized. Hence, it can be synthesized from a set of messages if and only if it belongs to the set:

```
lemma staticSecret_synth_eq:
"m ∈ staticSecret A ⟹ (m ∈ synth H) = (m ∈ H)"
apply force
done
```

Its proof is so short because the guarantee specifies for static secrets an existing theorem, `Key_synth_eq`, which expresses the same guarantee for a generic key. It can be found in theory `Message.thy` [URL11b]. Because it is installed in the simpset, the set of rewriting rules available to the simplifier, the `force` method alone closes the proof by unfolding the definition of `staticSecret`, invoking the simplifier and reasoning about the three possible keys that turn up.

Going back to our main task, theorem `NS_Says_Spy_staticSecret` still remains to be proved. It is convenient to quote it again here:

```
lemma NS_Says_Spy_staticSecret:
 "⟦Says Spy B X ∈ set evs; m ∈ parts{X};
   m ∈ staticSecret A; evs ∈ ns_public⟧ ⟹ A ∈ bad"
```

As it is customary, the inductive formula can be prepared and then induction can be launched:

```
apply (erule rev_mp, erule ns_public.induct)
```

It is convenient to simplify all subgoals without expanding the definition of `staticSecret` in order to avoid further subgoal splits. This solves the `Nil` case, and exposes a proof state of four subgoals, corresponding respectively to the `Fake`, `NS1`, `NS2` and `NS3` protocol rules. The application of the `clarify` method then cleans the `Fake` subgoal:

```
apply (simp_all del: staticSecret_def)
apply clarify
```

We focus on the `Fake` subgoal, omitting the others:

```
1. ⋀evsf Xa Ba.
      ⟦m ∈ parts {X}; m ∈ staticSecret A; evsf ∈ ns_public;
       X ∈ synth (analz (knows Spy evsf)); Says Spy B X ∉ set evsf⟧
      ⟹ A ∈ bad
```

It was not without considerable pondering that we realised it can be tackled by theorem `Fake_parts_sing: X ∈ synth (analz H) ⟹ parts {X} ⊆ synth (analz H) ∪ parts H`, which comes with file `Message.thy`. Incidentally, this guarantee was only adopted in our proofs about the accountability protocols [BP06]. In particular, the version resolved with the subset destruction rule is useful here:

```
thm Fake_parts_sing [THEN subsetD]
   ⟦X1 ∈ synth (analz H1); c ∈ parts {X1}⟧ ⟹ c ∈ synth (analz H1) ∪ parts H1
```

We apply it followed by an appeal to the simplifier to bind the variables:

```
apply (drule Fake_parts_sing [THEN subsetD], simp)
```

The subgoal takes a compact though meaningful form, very similar to the subgoal encountered during the direct proof of `NS_Spy_see_staticSecret` (beginning of Sect. 5.2), except for two differences. One is the definition of `staticSecret`, which is folded here:

```
1. ⋀evsf Xa Ba.
      ⟦m ∈ parts {X}; m ∈ staticSecret A; evsf ∈ ns_public;
       Says Spy B X ∉ set evsf;
       m ∈ synth (analz (knows Spy evsf)) ∪ parts (knows Spy evsf)⟧
      ⟹ A ∈ bad
```

The other difference is more important. It is the inductive hypothesis, which is:

- `m ∉ parts (knows Spy evsf)` in the direct proof of `NS_Spy_see_staticSecret`;
- `Says Spy B X ∉ set evsf`, with the extra fact that `m ∈ parts {X}` in the current proof, which is for theorem `NS_Says_Spy_staticSecret`.

The implications are serious. To distill the available facts yet more, the simplifier can be invoked again having care to avoid the use of `staticSecret_def` so that lemma `staticSecret_synth_eq` can be applied:

```
apply (simp del: staticSecret_def add: staticSecret_synth_eq)
```

The outcome becomes very compact:

```
1. ⋀evsf.
     ⟦m ∈ parts {X}; m ∈ staticSecret A; evsf ∈ ns_public;
      Says Spy B X ∉ set evsf; m ∈ parts (knows Spy evsf)⟧
     ⟹ A ∈ bad
```

It is now clear that the current subgoal would exhibit a contradiction with fact `m ∉ parts (knows Spy evsf)`. By contrast, it does not exhibit a contradiction as it stands, that is with its own inductive hypothesis. It describes a plausible scenario where the spy has potential access to some static secret `m` of `A`'s, but has not yet sent it out inside a message. Hence, this subgoal cannot be closed in its current form. It requires extra reasoning to relate the spy's potentiality to access an agent's private key to the postcondition, that is to whether the agent is bad.

Interestingly, this is exactly what `NS_Spy_see_staticSecret` does! It can profitably be applied to this subgoal and to the rest of the proof state through a script that we omit [Bel12], concluding the current proof. An evaluation of the findings at this stage is most important. In particular, we must understand how we stepped again over `NS_Spy_see_staticSecret` while we were conducting an alternative proof strategy to obtain it, that is through specialisation of protocol-independent guarantees, which appeared to succeed (end of Sect. 5.2). Perhaps that success was only apparent?

Halfway through the specialisation (Sect. 5.2), we proved:

```
lemma NS_staticSecret_parts_Spy_weak:
"⟦m ∈ parts (knows Spy evs); m ∈ staticSecret A;
  evs ∈ ns_public⟧ ⟹ A ∈ bad ∨
 (∃C B X. Says C B X ∈ set evs ∧ m ∈ parts{X})"
```

We observed that the first disjunct of the postcondition is minimal in the current threat model, and then set out to study a law to link the second disjunct to the first. In doing so, we stated `NS_Says_Spy_staticSecret` and deferred its proof till this section, where we realise that proving it requires having proved `NS_Spy_see_staticSecret`. The reason is that, it turns out that, when assuming `m` to be a static secret of `A`'s, no proof strategy could be found to reduce `Says C B X ∈ set evs ∧ m ∈ parts{X}` to `A ∈ bad` if not via `m ∈ parts (knows Spy evs)`. This means that the `Says` event must be first reduced to the canonical fact involving the `parts` operator. However, that fact also happens to be the main precondition of `NS_Spy_see_staticSecret`. In summary, the alternative, specialisation strategy we have investigated to prove `NS_Spy_see_staticSecret` fails because it soon brings out the main precondition of `NS_Spy_see_staticSecret`, leaving only the known, direct strategy to enforce its postcondition.

This conclusion can be seen metaphorically. While the direct proof strategy goes from London to Rome on some route, the specialisation proof strategy attempts to find a different route. However, having left London, it soon ends up back in London, facing again the known route to reach Rome. Despite our experience of one and a half decades in this area, we still find this insight somewhat surprising, and managed to appreciate the intuition behind it only after the proof experiments had showed its full technicalities. This is one of the wonders of inductive reasoning with theorem proving support.

6. Confidentiality study against the General Attacker

Thanks to equivalence results stating that n collaborating attackers bear the same offensive capabilities as a single attacker [CLC04], Dolev–Yao has become the standard threat model for security protocol analysis. Intuitively, it suffices to find any attack that may involve collaborative participation of any number of attackers who all aim at that very attack. In addition to a common aim, collaboration also implies knowledge sharing. The last decade showed ferment on the definition of new threat models, such as models optimised for finite-state analysis [LLB01, AC05, BKP10], models for specific applications [BBF10, CW09], configurable models [BC10], and DY-equivalent models [CD06]. The General Attacker (GA in brief) is a threat model where anybody has total network control and *may* misbehave arbitrarily, but there is no predefined knowledge sharing [ABCC09a, ABCC09b], and everyone acts for their own profit. It models more up-to-date scenarios where people use offensive capabilities and compete with each other, such as over eBay, during online auctions, or among the internal divisions of the same institution.

GA has made it possible to discover scenarios where different attackers achieve different, personal profit. For example, even with a simple protocol, all participants may exploit someone's misbehaviour to attack each

other [ABCC09a, ABCC09b]. Moreover, an agent's attack on Google's version of SAML single sign-on protocol [ACC+08] can be iterated successfully by someone else against that agent [ABCC10] (precisely, the latter result adopted a variant of GA where agents guard their static secrets). It is clear that if all agents misbehave, then they are not following any protocol and there would be no actual protocol analysis to conduct. But the aims of analysing protocols against GA are different, at least threesome.

Support the analysis of attacks by independent agents It is not rare that a protocol is found vulnerable, that is to suffer an attack against DY, when it is already deployed. The redesign and redeployment phases that would arise are typically costly and lengthy. If there also were other attacks by other agents, possibly against the first attacker, that is a form of *retaliation* [ABCC09b], then the scenario could realistically reach a favourable equilibrium such as with con games. GA could therefore be used after an attack is found against DY to investigate realistic consequences involving other agents' misbehaviour. By contrast, this is impossible against DY, where all agents except the attacker can only follow the protocol.

Question the realism of DY attacks Traditional protocol analysis has made us accustomed to reading attack traces that make perfect sense, and to concluding that a protocol must be redesigned. Fiazza et al. observe that those attacks are realistic in the assumption that the attacker knows that nobody else misbehaves, that is she knows the threat model [FPV11]. How plausible this prerequisite is in present protocol applications is at least questionable. As an example, they analyse a classical protocol attack in the setting of two non-collaborating attackers, showing that attackers may no longer have a successful strategy to succeed in attacking the protocol. In particular, one attacker will not be able to realise at the end of the trace whether her own attack has succeeded or not.

Elicit hidden preconditions of protocol guarantees We are used to conditional guarantees where a postcondition can be benefited of only if the preconditions are met. Abstractly, each guarantee has form $[\![p_1, \ldots, p_n]\!] \Longrightarrow q$. However, this guarantee only is meaningful if rooted in an underlying model, which is a set of assumptions concerning the protocol modelling in the broad sense and, in particular, the threat model. If we denote that set as $\mathcal{M}_{DY}$, the complete guarantee we have is $\mathcal{M}_{DY} \models [\![p_1, \ldots, p_n]\!] \Longrightarrow q$. Because DY is more restrictive than GA on most agents' behaviour, using GA lets us bring a number of preconditions hidden in $\mathcal{M}_{DY}$ out at the object level of proof, so that the guarantee takes the form $\mathcal{M}_{GA} \models [\![p_1, \ldots, p_n, p_{n+1}, \ldots, p_m]\!] \Longrightarrow q$, and hence remains more complete when the model is dropped. For example, guarantees against GA typically require clear preconditions that the entire population of agents does not misbehave, as we shall see below.

6.1. Modelling the General Attacker

Due to its generality, modelling the GA threat model appears to be a challenge for formal methods. However, with the Inductive Method, the actual modelling turns out simple. It is the subsequent formal reasoning what requires large care. All theory files encoding the method must be updated, though not significantly. They can be downloaded from our theory archive [Bel12].

The theory of protocol messages is updated in file `MessageGA.thy` with a simplification of the datatype of agents. A free type could work too, but the datatype is kept for historical reasons:

datatype — We only allow for any number of friendly agents
```
agent = Friend nat
```

Another modification is that the set `bad` is entirely disposed with. This is in line with the proviso of non-collaboration underlying GA. In fact, it makes the model more flexible because any form of collaboration (or collusion) can be easily defined at protocol specification level by means of dedicated protocol specification rules. For example, if an agent wanted to share her private key with another one, a protocol rule could introduce event `Says A B (Crypt (pubEK B) Key (priEK A))`, or `Notes B (Key (priEK A))`.

The theory of protocol events is updated in file *EventGA.thy* with a simpler definition of *knows*. Simplicity derives from the uniform treatment of all agents, each seeing the entire network traffic and their own notes, which are kept private:

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

Uniformity of agent treatment also simplifies the third theory, which is updated in *PublicGA.thy* with a more compact definition of initial state. Each agent only knows her own static secrets and all public keys:

```
overloading
  initState ≡ initState
begin

primrec initState where

  initState_Friend:
    "initState (Friend i) =
       {Key (priEK (Friend i)), Key (priSK (Friend i)), Key (shrK (Friend i))} ∪
       (Key ‘ range pubEK) ∪ (Key ‘ range pubSK)"
end
```

Many subsidiary lemmas must be updated throughout the three theories in the obvious way. They are beyond the focus of this paper, and can be found with our theory files [Bel12]. Our focus now turns to the actual confidentiality guarantees that can be designed and proved against GA. They are all grouped in file *ConfidentialityGA.thy*.

6.2. Protocol-independent study

Static secrets can be defined as seen above (Sect. 5) in order not to limit the treatment to private encryption keys. We set off to investigate whether the protocol-independent study conducted against DY (Sect. 5.1) scales up to the GA threat model. In brief, we shall see that the theorem statements can be easily upgraded, and the corresponding proof strategies scale up. The proof script is at time simplified due to the absence of *bad*, which avoids a case split.

Let us focus on *staticSecret_parts_Spy* (Sect. 5.1). Aiming at proving guarantees on knowledge of an agent with respect to another agent's, it is clear that what was the spy must now be a generic agent. So, the theorem should be stated upon a static secret of some agent *A* and about the knowledge of some agent *C*. Because the set *bad* does not exist anymore, it seems that the only circumstance for the latter to know the static secret initially, that is on the empty trace, is that the two agents coincide. The remaining part of the guarantee, concerning events whereby the secret can be leaked, continues to make sense against GA. Therefore, the conjecture can be expressed as:

```
lemma staticSecret_parts_agent:
 "⟦m ∈ parts (knows C evs); m ∈ staticSecret A⟧ ⟹
   A=C ∨
  (∃D E X. Says D E X ∈ set evs ∧ m ∈ parts{X}) ∨
  (∃Y. Notes C Y ∈ set evs ∧ m ∈ parts{Y})"
```

This can be proved. It is the very first guarantee ever proved to relate an agent's knowledge with the static secrets of another agent. Precisely, it reads as follows. When some agent *C* has some agent *A*'s static secret in his view of the traffic, then there are three options: *C* and *A* are the same agent; or someone sent the static secret in a message over the network; or *C* annotated that secret as part of some message. Note that the message sender of the middle case cannot be specified because in general anyone may send anything if not guided by a specific protocol.

As with its DY analogue, the lemma can be weakened expressing the precondition in terms of *analz* instead of *parts* thanks to the mentioned lemma *analz_into_parts*. It cannot, however, be strengthened by having *analz* in the postcondition because *A*'s secret cannot be analysed from a message should it appear in the message only encrypted under some key that cannot be analysed from the message. Our proof begins customarily, by preparation of the inductive formula and application of induction to the trace *evs*:

```
apply (erule rev_mp)
apply (induct_tac evs)
```

The outcome features two goals corresponding respectively to the base case when *evs* is the empty trace and to the inductive case when *evs* extends a given trace with an event `a`:

```
1. m ∈ staticSecret A ⟹
   m ∈ parts (knows C []) ⟶
   A = C ∨
   (∃D E X. Says D E X ∈ set [] ∧ m ∈ parts {X}) ∨
   (∃Y. Notes C Y ∈ set [] ∧ m ∈ parts {Y})
2. ⋀a list.
      ⟦m ∈ staticSecret A;
       m ∈ parts (knows C list) ⟶
       A = C ∨
       (∃D E X. Says D E X ∈ set list ∧ m ∈ parts {X}) ∨
       (∃Y. Notes C Y ∈ set list ∧ m ∈ parts {Y})⟧
      ⟹ m ∈ parts (knows C (a # list)) ⟶
          A = C ∨
          (∃D E X. Says D E X ∈ set (a # list) ∧ m ∈ parts {X}) ∨
          (∃Y. Notes C Y ∈ set (a # list) ∧ m ∈ parts {Y})
```

The first subgoal can be closed with the help of the new lemma *staticSecretA_notin_parts_initStateB: m ∈ staticSecret A ⟹ m ∈ parts(initState B) = (A = B)*. Although this lemma also appears to hold against DY for all agents who are not the spy, it is not surprising that it has not been studied before, as no reasoning on generic agents' knowledge has been conducted so far:

apply *(simp add: staticSecretA_notin_parts_initStateB)*

After termination of the first subgoal, it seems convenient to apply induction to event `a` of the remaining subgoal, and in fact most of the proof of *staticSecret_parts_Spy* can be reused:

apply *(induct_tac a)*
apply *(rule impI)*
apply *simp*
apply *(drule parts_insert [THEN equalityD1, THEN subsetD])*
apply *blast*
apply *simp*

The last two commands closed respectively the subgoal corresponding to the *Says* event and the one for the *Gets* event, so that only the one for the *Notes* event remains. Its treatment can be simplified as:

apply *simp*
apply *clarify*

The *clarify* method following simplification establishes that when *A = C* the inductive hypothesis closes the corresponding subgoal. By contrast, it could not handle automatically the corresponding case against DY, *A ∈ bad*, hence the need for an explicit case analysis in the proof of *staticSecret_parts_Spy*. The remaining subgoal therefore builds around *A ≠ C*:

```
1. ⋀list agent msg.
      ⟦m ∈ staticSecret A;
       m ∈ parts (knows C list) ⟶
       A = C ∨
       (∃D E X. Says D E X ∈ set list ∧ m ∈ parts {X}) ∨
       (∃Y. Notes C Y ∈ set list ∧ m ∈ parts {Y});
       m ∈ parts (insert msg (knows C list)); A ≠ C;
       ¬ (∃Y. (Y = msg ∨ Notes C Y ∈ set list) ∧ m ∈ parts {Y})⟧
      ⟹ ∃D E X. Says D E X ∈ set list ∧ m ∈ parts {X}
```

It can be treated as expected, and the proof terminates:

apply *(drule parts_insert [THEN equalityD1, THEN subsetD])*
apply *blast*
done

Also against GA, the result can be extended to generic secrets:

lemma *secret_parts_agent:*
"m ∈ parts (knows C evs) ⟹ m ∈ initState C ∨
(∃A B X. Says A B X ∈ set evs ∧ m ∈ parts{X}) ∨
(∃Y. Notes C Y ∈ set evs ∧ m ∈ parts{Y})"

The proof, here omitted but available [Bel12], is the same as that of the previous theorem with one exception. In case of empty trace, we must state that applying *parts* to an initial state yields nothing but the initial state, a lemma that is simple to prove, *parts_initState: parts(initState C) = initState C*. With the experience gained thus far, it is no longer surprising that such a guarantee was never investigated against DY, where it also holds. Of course, also this lemma can be weakened by having *analz* instead of *parts* in the precondition.

6.3. Protocol-dependent study

A protocol-dependent study is conducted over the model of our example protocol. The protocol model must account for GA rather than for DY this time. This can be done easily by a simple upgrade to the *Fake* rule in order to ensure that a generic agent may send out any of the messages she can fake. With our example protocol, we need:

```
| Fake: "⟦evsf ∈ ns_public;  X ∈ synth (analz (knows A evsf))⟧
        ⟹ Says A B X  # evsf ∈ ns_public"
```

Notably, the rule insists on the set *synth(analz(knows A evs))*, formalising all fake messages that a generic agent *A* may synthesize from her own analysis of the whole traffic. This new rule replaces the old one seen above (Sect. 3.4). The example protocol specified against GA can be found in the theory file *NS_Public_Bad_GA.thy*.

This section intends to investigate whether the protocol-dependent study conducted against DY scales up to the new threat model. The anticipation is that not only did the proof effort increase, but also the theorem statements require substantial care, as we shall see.

6.3.1. Static secrets

The first investigation aims at whether the specialisation proof strategy can be useful against GA. As in the previous case, the answer for dynamic secrets is negative. Focusing on static secrets, we perform some specialisation:

```
lemma NS_staticSecret_parts_agent_weak:
 "⟦m ∈ parts (knows C evs); m ∈ staticSecret A;
   evs ∈ ns_public⟧ ⟹
  A=C ∨ (∃D E X. Says D E X ∈ set evs ∧ m ∈ parts{X})"
```

It may seem surprising that theorem *NS_no_Notes* continues to hold against GA. This is due to the fact that neither the *Fake* rule nor the protocol specification rules prescribe anyone to perform that event. Thus, the weakly-specialised theorem can be proved in one go, by an appeal to this theorem and to the one that is being specialised:

```
apply (blast dest: NS_no_Notes staticSecret_parts_agent)
done
```

Due to the generality of the GA threat model, the only guarantee similar to *NS_Says_staticSecret* and *NS_Says_Spy_staticSecret*, which in this circumstance both helped against DY, reads as: if someone sends the static secret of someone else inside a message, then either the two agents coincide, or the owner of the secret also sent her static secret inside another message. It means that any abuse on someone's static secrets may only derive from the owner's own abuse. In trying to formalise such a guarantee, we begin observing a trace *evs* such that *Says D E X ∈ set evs ∧ m ∈ parts{X}*, for a generic agent pair and message *X*. This obviously implies that *m ∈ parts (knows C evs)* for any *C*. Therefore, as with DY, the specialisation proof strategy collapses—over the very preconditions of the theorem to specialise.

Before switching to the direct proof strategy, a corollary can be established. If an agent knows a static secret of someone else, then anybody knows it. This stresses that all agents have the same potential in GA:

```
lemma NS_staticSecret_parts_agent_parts:
"⟦m ∈ parts (knows C evs); m ∈ staticSecret A; A≠C; evs ∈ ns_public⟧ ⟹
  m ∈ parts(knows D evs)"
```

The proof is straightforward. It appeals to lemma *Says_imp_knows: Says A' B X ∈ set evs ⟹ X ∈ knows A evs*, which comes with the theory file *EventGA.thy* to generalise a result previously seen about the spy in DY:

```
apply (blast dest: NS_staticSecret_parts_agent_weak Says_imp_knows [THEN parts.Inj] parts_trans)
done
```

An alternative proof replaces the appeal to *NS_staticSecret_parts_agent_weak* by a joint appeal to the guarantees that were used to prove it, *staticSecret_parts_agent* and *NS_no_Notes*.

Let us begin to follow the direct proof strategy. The target guarantee can be formalised by binding the potential sender of a static secret to its owner:

```
lemma NS_staticSecret_parts_agent:
 "⟦m ∈ parts (knows C evs); m ∈ staticSecret A;
   C≠A; evs ∈ ns_public⟧
 ⟹ ∃ B X. Says A B X ∈ set evs ∧ m ∈ parts {X}"
```

The proof begins customarily:

```
apply (erule rev_mp, erule ns_public.induct)
apply (simp add: staticSecretA_notin_parts_initStateB)
apply simp
apply clarify
apply (drule parts_insert [THEN equalityD1, THEN subsetD])
apply (case_tac "Aa=A")
apply clarify
```

After the second command closes, not surprisingly, the case for the empty trace, the proof state exhibits the subgoal for the *Fake* case as its first. It is treated almost routinely, but a case study becomes necessary to assess coincidence of the stated agent, *A*, with the agent who launched the *Fake* rule, *Aa*. After some polishing, the two cases look like (omitting the rest) this:

```
1. ⋀evsf X Aa B.
      ⟦m ∈ staticSecret A; C ≠ A; evsf ∈ ns_public;
       m ∈ parts (knows C evsf) ⟶
       (∃B X. Says A B X ∈ set evsf ∧ m ∈ parts {X});
       X ∈ synth (analz (knows A evsf));
       m ∈ parts {X} ∪ parts (knows C evsf)⟧
      ⟹ ∃Ba Xa.
            (A = A ∧ Ba = B ∧ Xa = X ∨ Says A Ba Xa ∈ set evsf) ∧
            m ∈ parts {Xa}
2. ⋀evsf X Aa B.
      ⟦m ∈ staticSecret A; C ≠ A; evsf ∈ ns_public;
       m ∈ parts (knows C evsf) ⟶
       (∃B X. Says A B X ∈ set evsf ∧ m ∈ parts {X});
       X ∈ synth (analz (knows Aa evsf));
       m ∈ parts {X} ∪ parts (knows C evsf); Aa ≠ A⟧
      ⟹ ∃Ba Xa.
            (A = Aa ∧ Ba = B ∧ Xa = X ∨ Says A Ba Xa ∈ set evsf) ∧
            m ∈ parts {Xa}
```

The preconditions of the first subgoal feature an application of the set union operator (originating from simplification of a *parts* expression). As seen above, this can be translated into a disjunction of two facts about set membership. If the first disjunct holds, then the thesis follows; if the second disjunct holds, then the thesis follows by application of the induction hypothesis. Blasting confirms this line of reasoning by closing the subgoal. It is useful to polish the other one routinely, so that the proof script continues as:

```
apply blast
apply simp
apply clarify
```

The only surviving *Fake* subgoal now is:

```
1. ⋀evsf X Aa.
      ⟦m ∈ staticSecret A; C ≠ A; evsf ∈ ns_public;
       X ∈ synth (analz (knows Aa evsf)); Aa ≠ A;
       m ∉ parts (knows C evsf); m ∈ parts {X}⟧
      ⟹ ∃B X. Says A B X ∈ set evsf ∧ m ∈ parts {X}
```

Following the same patterns seen above (Sect. 5.3), this proof needs another of the rare applications of *Fake_parts_sing* followed by simplification by *staticSecret_no_synth*, which also holds against GA:

```
apply (drule Fake_parts_sing [THEN subsetD], simp)
apply (simp add: staticSecret_no_synth)
```

The subgoal remains as:

```
1. ⋀evsf X Aa.
     ⟦m ∈ staticSecret A; C ≠ A; evsf ∈ ns_public; Aa ≠ A;
      m ∉ parts (knows C evsf); m ∈ parts {X}; m ∈ parts (knows Aa evsf)⟧
     ⟹ ∃B X. Says A B X ∈ set evsf ∧ m ∈ parts {X}
```

This subgoal is unseen: it rests on messages known to three different agents. But the guarantee presented above, `NS_staticSecret_parts_agent_parts` can be used to close it! Having dealt with the various `Fake` cases, all other subgoals are closed easily, in one command:

```
apply (blast dest: NS_staticSecret_parts_agent_parts)
apply (force simp add: staticSecret_def)+
done
```

The guarantee just proved can be easily put in the canonical equational form:

```
lemma NS_agent_see_staticSecret:
 "⟦m ∈ staticSecret A; C≠A; evs ∈ ns_public⟧
 ⟹ m ∈ parts (knows C evs) = (∃ B X. Says A B X ∈ set evs ∧ m ∈ parts {X})"
apply (force dest: NS_staticSecret_parts_agent Says_imp_knows [THEN parts.Inj] intro: parts_trans)
done
```

The originality of the guarantees presented here, often involving relations on the knowledge of various agents, rules out the applicability of `spy_analz`. The proof strategies developed upon DY could in fact be reused only to a certain extent against GA.

6.3.2. Dynamic secrets

A general lemma, inspired by subsequent protocol proofs, which turns out to be protocol-independent, must be studied first. It expresses some strong conditions to reduce fact `c ∈ analz(insert Z H)`, which otherwise does not in general seem reducible:

```
lemma analz_insert_analz:
 "⟦c ∉ parts{Z}; ∀K. Key K ∉ parts{Z}; c ∈ analz(insert Z H)⟧
   ⟹ c ∈ analz H"
```

Two assumptions are necessary about the inserted element `Z`. One is that it does not contain `c` among its components. The informal justification of this fact is that, no matter what set `H` is `Z` inserted in, such insertion will not enable the analysis of `c` from `Z`. Likewise, such insertion will not enable the analysis of `c` from `H` thanks to the other assumption that `Z` features no key among its components. The two assumptions appear to be minimal about `Z`. We shall see below that, when the lemma is applied, `Z` is the fake message introduced by means of the protocol rule `Fake`. The proof can be conducted by induction on `analz`:

```
apply (erule rev_mp, erule rev_mp)
apply (erule analz.induct)
```

The fourth subgoal, about the analysis of a cyphertext, is the most interesting. We focus on it by:

```
prefer 4
apply clarify
```

Clarification leaves it as:

```
1. ⋀K X. ⟦Crypt K X ∈ analz (insert Z H);
           Key (invKey K) ∈ analz (insert Z H); ∀K. Key K ∉ parts {Z};
           X ∉ parts {Z}; Crypt K X ∉ parts {Z} ⟶ Crypt K X ∈ analz H;
           Key (invKey K) ∉ parts {Z} ⟶ Key (invKey K) ∈ analz H⟧
          ⟹ X ∈ analz H
```

Precondition `X ∉ parts{Z}` and lemma `parts.Body: Crypt K X ∈ parts H ⟹ X ∈ parts H` together imply that `Crypt K X ∉ parts{Z}`. This fact along with the last-but-one precondition imply that `Crypt K X ∈ analz{H}`. If we also had that `Key (invKey K) ∈ analz H`, we could derive the thesis by lemma `analz.Decrypt: ⟦Crypt K X ∈ analz H; Key(invKey K) ∈ analz H⟧ ⟹ X ∈ analz H`. The missing fact derives from the third and the last preconditions. The other subgoals can be closed easily, hence the proof script ends with:

```
apply (blast dest: parts.Body analz.Decrypt)
apply blast+
done
```

A confidentiality guarantee over a dynamic secret of the example protocol against GA can be stated at this point. The secret is a nonce. The statement, which is rather complicated, is explained afterwards:

```
lemma Agent_not_analz_NA:
     "⟦Key (priEK B) ∉ analz(knows C evs);
       Key (priEK A) ∉ analz(knows C evs);
       ∀S R Y. Says S R Y ∈ set evs ⟶
        Y = Crypt (pubEK B) {|Nonce NA, Agent A|} ∨
        Y = Crypt (pubEK A) {|Nonce NA, Nonce NB|} ∨
        Nonce NA ∉ parts{Y} ∧ (∀K. Key K ∉ parts{Y});
       C≠A; C≠B;  evs ∈ ns_public⟧
      ⟹ Nonce NA ∉ analz (knows C evs)"
```

This statement had to be designed with great care, and not without a number of failed attempts. Its structure resembles that of the contrapositive of the guarantee for static secrets seen above, `NS_staticSecret_parts_agent`, but the preconditions are significantly strengthened:

1. The `Says` event gets a third quantification over the sender variable as anyone can send a nonce, as opposed to `A`'s private key, which `A` alone can be proved to be able to send.
2. The message variable `Y` is restricted as to include no keys, because a nonce might in general appear encrypted under a key.
3. Alternative facts on `Y` are that it may take the form of the two protocol messages legitimately featuring the nonce, hence the preconditions on secrecy of the two encrypting private keys.

The proofs of this theorem and of the one it resembles are entirely different. Because a dynamic secret is typically exposed to the traffic inside a ciphertext, it must be studied in terms of the `analz` operator. In fact, its appearance in one of the legitimate messages is necessary to pinpoint it. The original development proof required more than 30 commands, but its current version is significantly streamlined. Its early commands are well understood. A lemma enforcing that initial states never contain nonces is needed [Bel12]:

```
apply (erule rev_mp, erule rev_mp, erule rev_mp, erule ns_public.induct)
apply (simp add: nonce_notin_analz_initState)
apply clarify
apply simp
apply (drule subset_insertI [THEN analz_mono, THEN contra_subsetD])+
```

After the first two commands, we face the `Fake` subgoal, where most effort is expected. We make it more readable by `clarify` and `simp`, and then do some manipulation using a theorem obtained by resolving three basic ones—it can be inspected by prefixing it with **thm** as often seen above. It is applied twice, thanks to the `+`, producing the two preconditions on private keys that can be seen as the last ones in the current subgoal:

```
1. ⋀evsf X Aa Ba.
      ⟦C ≠ A; C ≠ B; evsf ∈ ns_public;
       Key (priEK A) ∉ analz (knows C evsf) ⟶
       Key (priEK B) ∉ analz (knows C evsf) ⟶
       Nonce NA ∉ analz (knows C evsf);
       X ∈ synth (analz (knows Aa evsf));
       ∀S R Y.
          (S = Aa ∧ R = Ba ∧ Y = X ⟶
           X = Crypt (pubEK B) {|Nonce NA, Agent A|} ∨
           X = Crypt (pubEK A) {|Nonce NA, Nonce NB|} ∨
           Nonce NA ∉ parts {X} ∧ (∀K. Key K ∉ parts {X})) ∧
          (Says S R Y ∈ set evsf ⟶
           Y = Crypt (pubEK B) {|Nonce NA, Agent A|} ∨
           Y = Crypt (pubEK A) {|Nonce NA, Nonce NB|} ∨
           Nonce NA ∉ parts {Y} ∧ (∀K. Key K ∉ parts {Y}));
       Nonce NA ∈ analz (insert X (knows C evsf));
       Key (priEK A) ∉ analz (knows C evsf);
       Key (priEK B) ∉ analz (knows C evsf)⟧
      ⟹ False
```

There are many ways to proceed. Seeking the most didactic, we opt for simplifying the most complex precondition manually. We observe that the quantified formula is a conjunction. The first conjunct signifies that the quantified event corresponds to the `Fake` event, as confirmed by the equalities `S = Aa ∧ R = Ba ∧ Y = X` produced by the

simplifier. The second conjunct is the opposite case, hence the quantified event already appeared on the given trace `evsf`. To get rid of the second, we introduce the first alone via command `subgoal_tac`, and then launch the simplifier. It will dispose with the original, larger precondition. This line of reasoning is unsafe because it deletes a conjunct, but it works in this case as we know that conjunct to be irrelevant to our reasoning. Hence, we launch:

```
apply (subgoal_tac "∀S R Y.
          (S = Aa ∧ R = Ba ∧ Y = X ⟶
           X = Crypt (pubEK B) {|Nonce NA, Agent A|} ∨
           X = Crypt (pubEK A) {|Nonce NA, Nonce NB|} ∨
           Nonce NA ∉ parts {X} ∧ (∀K. Key K ∉ parts {X}))")
```

Of course, for the proof to be sound, the prover asks us to prove upon the original preconditions the fact that we manually introduced. This request forms an additional subgoal, coming right after the one under focus, that is as number 2. Confirming that our fact did not falsify the proof, we can easily close that subgoal by blasing it:

```
prefer 2
apply blast
```

The main subgoal can now be simplified:

```
apply simp
```

As anticipated, this leads to a more neat scenario:

```
1. ⋀evsf X Aa.
      ⟦C ≠ A; C ≠ B; evsf ∈ ns_public; Nonce NA ∉ analz (knows C evsf);
       X ∈ synth (analz (knows Aa evsf));
       ∀S R Y.
          Says S R Y ∈ set evsf ⟶
          Y = Crypt (pubEK B) {|Nonce NA, Agent A|} ∨
          Y = Crypt (pubEK A) {|Nonce NA, Nonce NB|} ∨
          Nonce NA ∉ parts {Y} ∧ (∀K. Key K ∉ parts {Y});
       Nonce NA ∈ analz (insert X (knows C evsf));
       Key (priEK A) ∉ analz (knows C evsf);
       Key (priEK B) ∉ analz (knows C evsf);
       X = Crypt (pubEK B) {|Nonce NA, Agent A|} ∨
       X = Crypt (pubEK A) {|Nonce NA, Nonce NB|} ∨
       Nonce NA ∉ parts {X} ∧ (∀K. Key K ∉ parts {X})⟧
      ⟹ False
```

The current `Fake` subgoal can be finally closed by a single command:

```
apply (force dest!: analz_insert_analz)
```

It appeals as a safe destruction rule, as ensured by the `!` symbol, to the reduction lemma for `analz`. Safety here is necessary because the last precondition is a disjunction of three facts, which in turn signify three corresponding subgoals. If the last disjunct holds, then lemma `analz_insert_analz` can close the proof producing the necessary contradiction, that `NA` is and at the same time is not analysable by `C`. This needs fact `Nonce NA ∈ analz(insert X (knows C evsf))`, which an unsafe destruction would not backtrack to. However, the same fact is necessary to terminate the proof also if one of the other two disjuncts holds. Precisely, that fact is simplified stripping off the inserted `X` whose form is available in the preconditions, hence leading to the same contradiction.

This line of reasoning may more easily be followed by the following alternative proof script:

```
apply (erule disjE) apply simp
apply (erule disjE) apply simp
apply (blast dest: analz_insert_analz)
```

Because the simplifier is not needed over the third subgoal (corresponding to the third disjunct seen above), the `blast` method can replace `force`. The lemma no longer needs be invoked as a safe destruction rule: it leads to a contradiction using preconditions that are only needed once because the subgoal is reduced.

The remaining subgoals can be closed in one go, and the proof terminates:

```
apply auto
done
```

Compared to its analogue for DY, `Spy_not_see_NA` (Sect. 4.2), this theorem seems more general because its postcondition applies to a generic agent. However, its preconditions are much stronger. On one hand, we interpret this as a positive outcome of the use of GA: the generality of the threat model forces the analyst to elicit all necessary preconditions for a useful fact to hold. These express the extent to what the agents who are involved may misbehave without compromising the postcondition. Clearly, the theorem statement gets more complex, for example with additional quantifiers, but Isabelle can handle them efficiently. The proof runtime in fact is not significantly increased.

Our guarantee must not be confused with the theorem that found Lowe's attack, which was a study of confidentiality of the responder's nonce against DY [URL11b]. Our guarantee concerns the initiator's nonce, whose leak marks the inception of a retaliation attack in a scenario with two malicious agents, hence against GA. While the retaliation finding [ABCC09b] stemmed from an optimised and targeted model-checking specification, `Agent_not_analz_NA` is designed to establish precisely the minimal facts that rule out the inception of the retaliation attack.

Finally, it does not seem possible to weaken the strong theorem precondition $\forall K.\ Key\ K \notin parts\{Y\}$, at least using the current message operators. It may be worth studying a more detailed operator to transform a local condition upon the message variable `Y` into the confidentiality postcondition of the theorem. It would seem enough to specify that `Y` does not contain those keys that may help analyse the dynamic secret. However, such definition may involve reasoning on subtraces, and hence forms non-straighforward future work.

7. Conclusions

A tutorial-style paper on the Inductive Method has never been written so far. We have been more interested in describing the actual protocol guarantees that were found, possibly only outlining the proof strategies to establish them. Our teaching experience shows that the learners face a conceptual gap between what they read in the research papers and what they execute in the theory files that come with the Isabelle repository [URL11b]. The Isabelle publications, such as the tutorial [NPW02] and the reference manual [Wen11] do explain each command in detail, but are not concerned with the conceptualisation of security properties and the development of their proof strategies. Hence the need for this manuscript, where only some significant results on security protocols are presented, but are described down to the last feature.

The focus has been kept on what is perhaps the main protocol property, confidentiality, from the standpoint of a protocol expert who is an Inductive Method learner. The existing account formed against the Dolev–Yao threat model has been revisited with the aim of detailing it fully. A novel, more systematic study has been presented, not without results that may have seemed exciting at first but in the end failed to strengthen the existing ones. This didactic experience was followed by another one through the confidentiality study against the General Attacker. The analyst was forced to explicitly state all preconditions to limit a potentially misbehaving population of agents in order to establish a security property. We believe that reading this paper and experiencing through the accompanying theory files makes the learning curve of the Inductive Method milder.

Acknowledgements

I am so very grateful to a number of colleagues with whom I shared over the years some of the ideas underlying this paper: Wihem Arsac, Stefano Bistarelli, Denis Butin, Antonio Cau, Xavier Chantry, Cas Cremers, Luca Compagna, Rosario Giustolisi, Helge Janicke, Jean Martina, Fabio Massacci, Larry Paulson, Salvatore Riccobene, Luca Viganò, Sasa Radomirovic, and Peter Ryan. Alfio Martini gave a useful tip to mechanically display proof states in LaTeX.

References

[ABCC09a] Arsac W, Bella G, Chantry X, Compagna L (2009) Attacking each other. In: Proceedings of the 17th international workshop on security protocols (CIWSP'09). LNCS Series. Springer, Berlin (in press)

[ABCC09b] Arsac W, Bella G, Chantry X, Compagna L (2009) Validating security protocols under the general attacker. In: Degano P, Viganò L (eds) Proceedings of the joint workshop on automated reasoning for security protocol analysis and issues in the theory of security (ARSPA-WITS'09). LNCS, vol 5511. Springer, Berlin, pp 34–51

[ABCC10] Arsac W, Bella G, Chantry X, Compagna L (2010) Multi-attacker protocol validation. Springer Int J Automa Reason 46(3–4):353–388
[AC05] Armando A, Compagna L (2005) An optimized intruder model for SAT-based model-checking of security protocols. In: Proceedings of the workshop on automated reasoning for security protocol analysis (ARSPA'04). ENTCS, vol 125. Elsevier, Amsterdam, pp 91–108
[ACC+08] Armando A, Carbone R, Cuellar J, Tobarra L, Compagna L (2008) Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In: Proceedings of FMSE. ACM Press, New York
[BAN89] Burrows M, Abadi M, Needham RM (1989) A logic of authentication. In: Proceedings of the Royal Society of London, vol 426, pp 233–271
[BBF10] Benenson Z, Blass E-O, Freiling FC (2010) Attacker models for wireless sensor networks. Inf Technol 52(6):320–324
[BC10] Basin DA, Cremers CJF (2010) Modeling and analyzing security in the presence of compromising adversaries. In: Gritzalis D, Preneel B, Theoharidou M (eds) Proceedings of the 15th European symposium on research in computer security (ESORICS'10). LNCS, vol 6345. Springer, Berlin, pp 340–356
[BCSS09] Basin D, Capkun S, Schaller P, Schmidt B (2009) 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. Springer, Berlin, pp 1–22. http://dx.doi.org/10.1007/978-3-642-03359-9_1
[Bel07] Bella G (2007) Formal correctness of security protocols. Information security and cryptography. Springer, Berlin
[Bel12] Bella G (2012) Inductive study of confidentiality. Archive of Formal Proofs, vol 2012. http://afp.sourceforge.net/entries/Inductive_Confidentiality.shtml
[BKP10] Basagiannis S, Katsaros P, Pombortsis A (2010) An intruder model with message inspection for model checking security protocols. Comput Secur 29(1):16–34
[Bla11] Blanchet B (2011) Proverif: Cryptographic protocol verifier in the formal model. http://www.proverif.ens.fr/
[BM03] Boyd C, Mathuria A (2003) Protocols for authentication and key establishment. Information security and cryptography. Springer, Berlin
[BMP06] Bella G, Massacci F, Paulson LC (2006) Verifying the SET purchase protocols. J Autom Reason 36(1–2):5–37
[BP06] Bella G, Paulson LC (2006) Accountability protocols: formalized and verified. ACM Trans Inf Syst Secur 9(2):1–24
[CD06] Cederquist J, Dashti MT (2006) An intruder model for verifying liveness in security protocols. In: Proceedings of the 4th ACM workshop on formal methods in security (FMSE'06). ACM Press, New York, pp 23–32. http://doi.acm.org/10.1145/1180337.1180340
[CLC04] Comon-Lundh H, Cortier V (2004) Security properties: two agents are sufficient. Sci Comput Program 50(1–3):51–71
[CW09] Cordasco J, Wetzel S (2009) An attacker model for MANET routing security. In Proceedings of the 2nd ACM conference on wireless network security (WiSec'09). ACM Press, New York, pp 87–94. http://doi.acm.org/10.1145/1514274.1514288
[DY83] Dolev D, Yao A (1983) On the security of public-key protocols. IEEE Trans Inf Theory 2(29):198–208
[FB99] Nipkow T, Baader F (1999) Term rewriting and all that. Cambridge University Press, London
[FPV11] Camilla Fiazza M, Peroli M, Viganò L (2011) Attack interference in non-collaborative scenarios for security protocol analysis. In: Proceedings of the international conference on security and cryptography (Secrypt'11)
[Hrb99] Hrbacek K (1999) Introduction to set theory. CRC Press, Boca Raton
[LLB01] Liu D, Li X, Bai Y (2001) An intelligent intruder model for security protocol analysis. In: Qing S, Okamoto T, Zhou J (eds) Information and communications security. LNCS, vol 2229. Springer, Berlin, pp 13–22
[McM93] McMillan K (1993) Symbolic model checking. Kluwer Academic Publisher, Dordrecht
[Mea96] Meadows CA (1996) The NRL protocol analyzer: an overview. J Logic Program 26(2):113–131
[MVM09] Miller FP, Vandome AF, McBrewster J (2009) Mathematical induction: mathematical proof, mathematical logic. Alphascript Publishing, Mauritius
[NPW02] Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL: a proof assistant for higher-order logic. LNCS tutorial, vol 2283. Springer, Berlin
[Pau94] Paulson LC (1994) Isabelle: a generic theorem prover. LNCS, vol 828. Springer, Berlin
[Pau98] Paulson LC (1998) The inductive approach to verifying cryptographic protocols. J Comput Secur 6:85–128
[Pau10] Paulson LC (2010) Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. http://www.cl.cam.ac.uk/~lp15/papers/Automation/paar.pdf
[RSG+01] Ryan PYA, Schneider S, Goldsmith M, Lowe G, Roscoe AW (2001) Modelling and analysis of security protocols. Addison-Wesley, Reading
[THG99] Thayer FJ, Herzog JC, Guttman JD (1999) Strand spaces: proving security protocols correct. J Comput Secur 7:191–220
[URL11a] (2011) Cygwin: a Linux-like environment for Windows. http://www.cygwin.com
[URL11b] (2011) Isabelle download page. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/download.html
[URL11c] (2011) Poly/ML: a full implementation of Standard ML. http://www.polyml.org
[URL11d] (2011) Proof General: a generic interface for proof assistants. http://proofgeneral.inf.ed.ac.uk
[Wen11] Wenzel M (2011) The Isabelle/Isar reference manual. http://isabelle.in.tum.de/doc/isar-ref.pdf
[Wie06] Wiedijk F (ed) (2006) The seventeen provers of the world. LNAI, vol 3600. Springer, Heidelberg

Received 16 October 2011
Revised 27 March 2012
Accepted 22 April 2012 by Eerke Boiten