Introducing ASPECT[1] – a tool for checking protocol security

Brian Monahan
Trusted E-Services Laboratory
HP Laboratories Bristol
HPL-2002-246
September 16th , 2002*

E-mail: brian_monahan@hp.com

security protocols, software tools, strand spaces, formal methods

We have developed an efficient proof-of-concept prototype tool for security protocol validation called ASPECT. This prototype is designed to demonstrate the feasibility of our approach to security protocol checking and validation. Our approach can be characterised in terms of "flaw-detection for security protocols", much in the same way that programs in a programming language can be type checked. However, we go beyond approaches based solely upon static analysis by combining an initial static, passive analysis with an active search that attempts to uncover attacks upon the given protocol. Naturally, a successful attack indicates the presence of a security flaw in the protocol. More precisely, ASPECT takes a conventional security protocol description given in terms of message sequences between several parties, and analyses this statically in terms of defined high-level security goals (e.g. confidentiality, authorisation) to derive a number of conjectured security properties for the given protocol. ASPECT then attempts to find protocol flaws, if any, by trying to dynamically construct active attack patterns to disprove these conjectures. The question of whether the checking techniques used within ASPECT are *complete* is currently open and the subject of further research. In this introductory report, we illustrate what ASPECT does in terms of a worked example, where we develop a simple authentication protocol. This protocol is revised several times and ASPECT used to examine these revisions. As you would anticipate, ASPECT finds no flaws with the final revision of this protocol.

* Internal Accession Date Only
Approved for External Publication
[1] Automated Security Protocol Examination and Checking Tool

Introducing ASPECT[1] – a tool for checking protocol security

Brian Monahan (brian_monahan@hp.com)

Trusted E-Services Laboratory, HP Labs, Bristol, UK

May 2002

Abstract We have developed an efficient proof-of-concept prototype tool for security protocol validation called ASPECT. This prototype is designed to demonstrate the feasibility of our approach to security protocol checking and validation.

Our approach can be characterised in terms of "flaw-detection for security protocols", much in the same way that programs in a programming language can be type checked. However, we go beyond approaches based solely upon static analysis by combining an initial static, passive analysis with an active search that attempts to uncover attacks upon the given protocol. Naturally, a successful attack indicates the presence of a security flaw in the protocol.

More precisely, ASPECT takes a conventional security protocol description given in terms of message sequences between several parties, and analyses this statically in terms of defined high-level security goals (e.g. confidentiality, authorisation) to derive a number of conjectured security properties for the given protocol. ASPECT then attempts to find protocol flaws, if any, by trying to dynamically construct active attack patterns to disprove these conjectures. The question of whether the checking techniques used within ASPECT are *complete* is currently open and the subject of further research.

In this introductory report, we illustrate what ASPECT does in terms of a worked example, where we develop a simple authentication protocol. This protocol is revised several times and ASPECT used to examine these revisions. As you would anticipate, ASPECT finds no flaws with the final revision of this protocol.

[1] ASPECT stands for "Automated Security Protocol Examination and Checking Tool".

Contents

INTRODUCTION **3**

Security protocol engineering 3
Roadmap 3

1 PROTOCOL DESCRIPTIONS **4**

1.1 Protocol notation and its informal meaning 4
1.2 How to interpret message transfer events 7
1.3 Initial Conditions and Final Objectives 8

2 HOW TO ATTACK A PROTOCOL **10**

2.1 The attack model 10
2.2 Strand spaces and security analysis of protocols 12
2.3 The Middleperson Attack Strategy 13

3 WHAT DOES ASPECT DO? **14**

3.1 Passive Analysis 14
3.2 Active Analysis 15
3.3 Analytic coverage 15

4 WORKED EXAMPLES **16**

4.1 First version 16
4.2 Second version 19
4.3 Third version 22
4.4 Fourth version 23
4.5 Performance figures 24

CONCLUSIONS AND FURTHER WORK **25**

ACKNOWLEDGEMENTS **26**

REFERENCES **26**

APPENDIX: EXAMPLE REPORT **28**

Introduction

Protocols are ubiquitous in distributed systems – they are the expressions of distributed action between the communicating agents making up the system. To ensure that trust is established and maintained appropriately, the various agents involved need to use protocols enjoying security properties such as confidentiality and which provide security services such as authentication.

However, knowing the security requirements for a distributed system is one thing – it is quite another to ensure that an ensemble of agents and the various protocols between them will behave accordingly. Further discussion of these important, general issues may be found in [TvS02], [A01], [Sc96].

For definiteness, a security protocol is an abstract (multi-party) messaging protocol designed to fulfil particular security requirements, and is perhaps embedded within some other protocol. The remainder of this paper is concerned with security protocols and how ASPECT can be used to check them.

Security protocol engineering

The range of uses to which distributed systems are being put is expanding rapidly (c.f. Peer to Peer networking [P2P], MIT's Project Oxygen [Oxy]) – providing a source of fresh challenges for old and new security protocols to meet. However, there are already many security protocols widely available and in general use. Such broad adoption inevitably means that many of these have been adapted for duties they were not originally designed to perform – i.e. requirement creep. Thus, protocols will continue to evolve and their application is ever increasing and widening.

There would be little cause for concern if these protocols were *known* to cover all of their security requirements – both old and new. Unfortunately, this is far from being the case.

Relatively recently, techniques have started to emerge that can formally capture these requirements and then objectively assess protocols against them – see [TFHG99], [Lowe95], [Lowe96], [DLMS], [P98], [FA01], [CDMLS], [DM99], [SRI], [RS01] for example. Breakthrough decidability and complexity results concerning security analysis of protocols have also been found recently – see [DLMS], [SRI] [AL00], [RT01]. See [CS02] for a recent survey of these advances in security protocol analysis techniques.

Such advances offers the hope that engineering tools can be developed that can routinely check that protocols do meet their security requirements, under appropriate conditions (see [S99]). Such tools would enable more routine development of particular protocols that can be tailored to fit its security requirements more accurately. Standardisation would also benefit by enabling a verifiable classification of protocols against standard security requirements.

ASPECT is an early prototype contributing to the creation of effective protocol engineering tools.

Roadmap

The next section describes what security protocols are and how they are described in terms of message sequences. This section also describes how the message terms and keys are to be interpreted. The second section describes how security protocols might be attacked, in particular using the so-called *middleperson* attack strategy. Section 3 describes the top-level structure and processing model behind ASPECT, in particular introducing the main Passive and Active analysis phases. The main section of this paper is a series of worked examples based around the iterative development of a simple authentication protocol. Finally, we discuss further work and present our conclusions. An appendix contains an example report generated by ASPECT.

1 Protocol descriptions

Security protocols are conventionally described at an abstract level in terms of message sequence diagrams, together with some notation for the security related message elements, such as *nonce values*[2] and *encryption operations*. These diagrams specify a sequence of point-to-point message transfers between sending and receiving parties. Although each transfer is typically between two parties, several distinct parties are typically involved in the protocol overall.

Although the intuitive idea behind these conventional descriptions is fairly straightforward, it also seems that there are some subtly different variants in actual use. For example, the protocol notations employed in [A01], [HAC97], [Sc96], [TvS02], [TFHG99] are all similar but not identical. These notations are mostly used for conceptual description and none were particularly designed for formal computation or manipulation as such.

Other, more formal, notations for security protocols are available, such as CAPSL [CAPSL] and the Spi calculus [AG98] for example. However, these are more complex and expressive notations than we need for demonstrating the capabilities of the prototype ASPECT. We have instead used a simpler, more lightweight notation that is very much in keeping with the spirit of the notations used traditionally for protocol description. Because of this, it should be straightforward to translate existing security protocols into the format we use for input to ASPECT. However, producing tools similar to ASPECT that exploit these more expressive notations is a topic for future research.

1.1 Protocol notation and its informal meaning

A security messaging protocol, *sP*, can be described purely in terms of message sequence diagrams. A *message sequence diagram* consists of a sequence of message transfers. Each *message transfer* is a triple consisting of a message, *Msg*, which is transmitted from some sending party, *Send*, to another receiving party, *Rcv*. Message transfers are denoted thus:

$$Send \rightarrow Rcv : Msg$$

Fig 1 illustrates a short protocol definition consisting of two message transfers involving the parties *A* and *B*. This is used purely as an illustrative example here and elsewhere in this paper.

---

1. $A \rightarrow B$: $e\{ n(A), id(A) : PubK(B) \}, data(A)$
2. $B \rightarrow A$: $e\{ n(B), n(A), id(B) : SymK(A, B) \}$

Fig 1: A short example protocol

Furthermore, we require that the given protocol is *deterministic* – this means that the sender of any immediately following message was the receiver of the previous message in the protocol. Because there is a unique successor, all the message transfers of a deterministic protocol follow one another, forming a linear sequence.

Each protocol definition is considered a *pattern* of message transfers, defined over the set of particular parties (known as *principals*), as well as other message related entities, such as nonce values. Protocol definitions may thus be *instantiated* by binding the principals and these other entities to particular literal values.

[2] A *nonce* generally means "For the one occasion" – but here it means, particularly, "A number issued once". Although a nonce is only ever *issued* once, it may of course be transmitted several times during its lifetime.

The underlying abstract execution model we have in mind is to take some set of (actual) principals, *Prc*, and to consider sets of possible *traces* i.e. sequences of message transfers, between these principals. Using this concept of trace, we can define a *simple run* of a protocol *sP* to be a trace that exactly *matches* as an instance of *sP*. Thus, there are many possible simple runs of each protocol definition.

We further say that a *general run* of a protocol *sP* is a trace consisting of several simple runs joined together i.e. multiple instances. This therefore means that every simple run is a general run. The term *session* commonly means the same as "protocol run" (typically a simple run) and thus *multiple sessions* corresponds to "general protocol run".

1.1.1 Message Terms

Message Terms describe the abstract *structural content* of messages sent between principals. As such, they occur within protocol definitions, traces, etc. The structure of these terms is given by the grammar:

$$MT ::= \mathbf{id}(P) \mid \mathbf{data}(P, n) \mid \mathbf{n}(P, n) \mid K \mid \mathbf{e}\{ MT : K \} \mid MT_1, \ldots, MT_i$$

$$K ::= \mathbf{PubK}(P, n) \mid \mathbf{SecK}(P, n) \mid \mathbf{SymK}(A, B, n) \mid K^{-1}$$

where *P* denotes a principal (i.e. a name) and $n>0$ is a number literal denoting an *instance marker*. The purpose of these instance markers[3] is to allow distinct instances of the same kind of term (i.e. message or key) to occur within protocol runs, traces and protocol definitions.

As a convenient shorthand, we may identify items of the form *n(P)* with *n(P, 1)* and so on.

The non-terminals *MT* and *K* specify message and key terms accordingly. We explain the meaning of these terms informally:

***id**(P)* – **Identity claim**
: This term symbolically stands for a data value representing a *claim* of identity for principal *P*. Such a data value is not in itself cryptographically secure – it is perhaps convenient to think of it as a plaintext string containing some combination of a literal name, a URL, an ID number and so on.

***data**(P, n)* – **Data item**
: This term symbolically stands for a pure data value that is *originated* by principal *P*, and labelled by instance marker *n*. Typically, such values convey no security significance in themselves, and could therefore be ignored or deleted. However, they may serve as valuable *targets* for protocol attackers to capture and so we choose to include them within our protocol descriptions.

***n**(P, n)* – **Random nonce value**
: This term symbolically stands for a randomised value that is *originated* by principal *P*, and labelled by instance marker *n*. Nonce values are used to uniquely identify protocol runs. This means that such a value must have *uniquely originated* from principal *A* in every run of the protocol.

K – **Key data item**
: This term symbolically stands for a data value representing a key (see below).

[3] Instance markers can be globally renumbered without a change in meaning, just as long as existing distinctions between terms are maintained. In particular, there is *no* specific ordering implied by the actual numbers used as instance markers.

***e*{ *MT : K* } – Encrypted item**

This term symbolically stands for a (typically) opaque block of data, derived by encrypting the given message data specified by *MT*, under the key specified by *K*. Obviously, the key itself is not literally embedded in the message at this point.

The idea is that the internal content, the value *M*, can only be extracted from the given block by a principal possessing the *inverse key*, K^{-1} (see below). If the principal does not have this inverse key, then the block appears to be completely opaque – it cannot be opened.

$MT_1, \ldots, MT_i$ – **Sequences**

Sequences of message term items can also form message terms. We implicitly use associativity to "flatten" all such literal sequences occurring in protocol descriptions. This normalisation step is justified because any protocol whose security properties *rely* upon how a sequence is nested should be regarded as vulnerable. Normalisation effectively ensures that such subtle dependencies do not arise.

1.1.2 Key terms

Keys are used when encrypting data and may appear literally as data within messages (see above). We further assume that *decryption* is subsumed into the way that particular keys are used. Each key value, *Key*, is regarded as having an (unique) *inverse* key, specified by Key^{-1}, with which to *decrypt* messages e.g.:

$$e\{\ e\{\ Msg : Key\ \} : Key^{-1}\ \} = Msg$$

Additionally, we assume that key inverse is an *involution*: $(Key^{-1})^{-1} = Key$.

Using inverse keys in this way implies we need not introduce an additional decryption operation explicitly into our present model. We may replace any would-be use of a decryption operation by an encryption operation using the corresponding key inverse instead. A future extension might be to add an explicit decryption operator.

The different types of key values are either *asymmetric* or *symmetric* and can be:

PubK(P, n) – **Public key value**

This term denotes an asymmetric publicly available key, that is potentially *widely known* to be *owned* by principal *A*. This is labelled by instance marker *n*. It's inverse is the secret key, *SecK(P, n)*.

SecK(P, n) – **Secret key value**

This term denotes an asymmetric secret key that is *uniquely owned* by principal *A*. This is labelled by instance marker *n*. It's inverse is the public key, *PubK(P, n)*.

SymK(P, Q, n) – **Symmetric key value**

This term denotes a symmetric secret key that is simultaneously *owned* by principals *P* and *Q*. This is labelled by instance marker *n*. It's inverse is itself, the symmetric key *SymK(A, B, n)*. Also, the order in which the owners is given is irrelevant i.e. *SymK(A, B, n) = SymK(B, A, n)*.

Each type of key has different ownership, usage and inverse characteristics has been conveniently collected here as **Table 1**, for clarity:

	Ownership	Usage	Inverse
PubK(P, n)	*P*	public	*SecK(P, n)*
SecK(P, n)	*P*	*P*	*PubK(P, n)*
SymK(P, Q, n)	$P \wedge Q$	$P \vee Q$	*SymK(P, Q, n)*

Table 1: Key properties and characteristics

Finally, since protocols do not contain variables denoting keys, the key inverse operator $(\_)^{-1}$ can be systematically eliminated from all expressions involving keys.

1.2 How to interpret message transfer events

Message transfer events specify the structure of the message sent by the sender and the expected structure of the message received by the receiver. Essentially, the message part of a message transfer represents a pattern to be matched in some way. However, this pattern is interpreted slightly differently in each case.

1.2.1 Sending messages

The sending principal constructs messages that match to the pattern term either using locally generated material or by using acquired material sent earlier during the protocol (e.g. sending nonce responses). Specifically, encrypted items could be locally constructed, involving directly accessible keys for encryption.

Alternatively, they may be opaque encrypted items that had been received earlier in the protocol and are forwarded onwards intact as part of the next message sent. This permits key distribution protocols (such as Kerberos, Ottway-Rees [HAC97], [Sc96]) that operate by forwarding encrypted key-containing blocks among the principals involved.

1.2.2 Receiving messages

The receiving principal uses the message term to decompose the incoming message into its constituent parts. The protocol definition is used to determine which items sent earlier should match the instances actually sent on this occasion. Items seen for the first time (within each protocol run) are regarded as binding occurrences.

The notation for encrypted items, *e{Msg : Key},* specifies which key was used for encryption – thus, the receiver can only inspect the encrypted message content if it has direct access to Key^{-1}, the corresponding decryption key. If this decryption key is inaccessible, the encrypted item is retained for possible use later e.g. key distribution protocols.

1.2.3 Further constraints: The Perfect Encryption and Unique Origination assumptions

To allow analysis of encryption items, we additionally require the *Perfect Encryption* assumption:

$$e\{ Msg_1 : Key_1 \} = e\{ Msg_2 : Key_2 \} \iff (Msg_1 = Msg_2) \wedge (Key_1 = Key_2)$$

Broadly, this asserts that no two encryption items could ever be *accidentally* equal – both the message and the keys must be correspondingly equal for the encryption blocks to be equal. In particular, if the keys used are *different* then the encryption items are always *distinct*, no matter what the messages happen to be. In practice, this is quite a reasonable assumption.

The *Unique Origination* assumption says that certain classes of *owned* terms (such as nonce items) should always *originate* at a single, unique source within each protocol. This means that the *first*

occurrence of a particular instance must be within the body of some particular message sent by the appropriate *owning* principal. For example, a nonce item *n(A)* should always occur first within some message sent by principal *A*. Thus, the *unique origination* condition models the fact that all such terms are *generated* at a unique point in each protocol run (by honest principals).

1.3 Initial Conditions and Final Objectives

A protocol represents a coordinated pattern of behaviour that the participants perform within some context to achieve some shared effect. Accordingly, protocols may place initial requirements upon their participants so that their final shared objectives can be achieved.

1.3.1 Calculating the initial conditions

Fortunately, given a protocol definition, it is reasonably straightforward to determine the *minimum initial requirements* for the (honest) principals, thus enabling them to carry out each of their protocol actions.

Starting from the first sender, look at the messages sent and determine what is needed to construct the message, based upon what is already known by that party. Any item that is not constructible from already known (or originated) pieces must be provided initially. Received messages also need to be decomposed to extract items for potential reuse or for matching purposes. By way of illustration, we examine the simple protocol defined earlier in Fig 1.

To send the first message, *e{ n(A), id(A) : PubK(B) }, data(A)*, it is clear that *A* has to use *B*'s public key *PubK(B)*. Therefore, as this is the first message, *A* had to have this key data available initially. We reasonably assume that A must also know the claimed identity, *id(B)*, for *B*. Naturally, this also conversely implies that principal *B* must also have the key *PubK(B)* and the corresponding secret key, *SecK(B)*. Using this, principal *B* can decode the message received and acquire the nonce value, *n(A)*. *B* also notes the identity claim *id(A)* that was embedded in the encrypted item. Thus, *B* learns that the sender of this message is claiming to be *A*.

To construct the second protocol message, *e{ n(B), n(A), id(B) : SymK(A, B) }*, the principal *B* needs to know items *n(A), n(B), id(B)* and the symmetric key *SymK(A,B)*. The item *n(A)* was learnt from the previous message and both *n(B), id(B)* are either originated by *B* or already known to *B*. This leaves the symmetric key, *SymK(A,B)* which therefore has to be already known to *B* initially, in order to make the protocol work. If *B* has this key initially, then so too does the other owner, *A*. Given that both principals *A* and *B* share a common symmetric key this naturally implies that B already knew the identity claim for *A*. Thus, the occurrence of *id(A)* from the first message must belong to the set of identity claims already known to *B* – it cannot be an identity claim from a principal not already known to *B*.

Finally, principal *A* can open the encrypted item in the second message because we now know that they possess the symmetric key *SymK(A,B)*. This means that *A* acquires the nonce *n(B)* and *id(B)*. Since *A* initially knew *id(B)*, this occurrence can be compared for equality with the identity claim that *A* originally sent.

The outcome of all this is:

1. *A* initially knows *id(A), id(B), PubK(B)* and *SymK(A, B)*.
2. *B* initially knows *id(B), id(A), PubK(B), SecK(B)* and *SymK(A, B)*.
3. The principal *A* does not need their own public key, even though B did.
4. Both principals *A* and *B* are "well known" to each other – as they both initially possess a common symmetric key. Such a key would have to be issued to both parties prior to running this protocol, either from one of them or from some trusted third party.

5. Each party learnt nonce values from the other – these can be used to check for replayed messages i.e. freshness.

1.3.2 Generic security objectives

We assume that protocols are written to fulfil some security objectives such as key distribution, confidentiality, authentication and origination integrity. For our purposes here, we shall just focus on confidentiality and authentication objectives. Extending our approach here to other security objectives seems tractable and is the subject of current research.

The idea here is to characterise these objectives in a general, structural manner. This will provide an effective means to *match* protocol definitions against these generic objectives.

The outcome of this matching process will be a specific set of required goals to be met by the particular protocol. This will prove useful when checking protocols, as these instantiated protocol goals can then play the role of *targets* for the attacker to explicitly *acquire* or "break".

The next subsection discusses the important issue of explicitly bounding the number of protocol runs within which the attacker has to acquire at least one target. The remaining subsections then informally discuss how confidentiality and authentication objectives are realised.

1.3.3 Bounding the number of sessions

It turns out to be important to explicitly bound the number of sessions that the attacker would need to acquire at least one target. For example, it is known that the general reachability problem for security protocols (with unlimited number of sessions permitted to the attacker) is undecidable [DLMS]. The same paper further shows the decidability of the same reachability problem (with unbounded numbers of sessions) but for a restricted class of protocols (i.e. nonce free). The decidability of the bounded case was reported in [AL00]. These results imply that the number of sessions involved is a critical parameter – and needs to be explicitly considered during checking.

For our work here, we only consider the case of simple runs of protocols i.e. single sessions.

This is less of a restriction then might first be thought, since we could always obtain the effect of multiple sessions by building a more complex protocol description that explicitly "unrolls" and instantiates several sessions to form a single one. However, in terms of convenience and flexibility for checking, it is better to give explicit support for multiple sessions, including the ability to examine and check combinations of several protocols together (e.g. unintended services).

Naturally, all of the security objectives under consideration should be regarded as having been qualified by the number of sessions that an attacker has available to capture some targets. It is conceivable that a protocol has no attacks for n sessions, only to find that an attack exists for $(n+1)$ sessions.

1.3.4 Valuable items

To formulate both confidentiality and authentication objectives precisely, we first need the concept of *valuable item*. A *valuable item* is simply an item that is never transmitted in an exposed manner. An item is *exposed* if it is transmitted in clear or in an encrypted form using a *weak* key (i.e. a key that turns out to have been compromised or is easily determined). Thus, exposure can be retrospective because future key compromises can make previous communications vulnerable.

1.3.5 Confidentiality objective

The general confidentiality objective is easily stated – a protocol possesses confidentiality if and only if no valuable data leaks to an unauthorised party i.e. an attacker.

At first glance, this looks almost tautological since an item is considered valuable precisely when it isn't exposed. However, direct exposure is a particularly brutal form of loss of confidentiality.

Instead, we are interested if apparently secure items i.e. the valuable items, could nonetheless be captured.

Another way of saying this is that a protocol possesses confidentiality when the only items that an attacker could have captured are those that are already exposed.

1.3.6 Authentication objective

The general authentication objective is harder to define – especially as we want a definition that is also amenable to direct checking. However, the intuition behind authentication is nonetheless straightforward – principal A authenticates principal *B* whenever *A* can verify that the identity claim made by *B* is *valid* i.e. that *B* is the principal identified by the identity claim.

In cryptographic terms, the process of validating identity involves producing and checking certain data that only a particular principal could have originated. This in turn reduces to the demonstrable use of some keys or other data that only they could possess. For example, digital signing a document with your secret private key proves that you had sufficient access to the document (with overwhelming probability) because no one else could use your private key.

A more pragmatic definition is thus – principal A authenticates principal B whenever:

- *B* presents his own identity claim data to *A*
- *B* sends a valuable item to *A* whose construction exploited some resource solely available to *B*. Moreover, *A* has to be able to independently examine the construction and ensure that this resource unique to *B* was indeed used.

A typical way of implementing the above uses the idea of *challenge-response* combined with use of some kind of secret by *B*. Generally speaking, *A* sends a challenge value to *B*, and then *B* performs a local computation upon *A*'s challenge, making use of *B*'s secret. *B* sends the result back to *A*, whereupon *A* checks that this response is consistent with the original challenge. If this checks OK, then *A* can authenticate *B* after all.

2 How to attack a protocol

We now turn to the business of attacking protocols. Our first subsection introduces the attack model defining the attackers general capabilities – i.e. the rules of the game. This also allows us to say what is expected of an *honest* participant. Next, strand spaces [TFHG99] are introduced as a conceptual framework for exploring potential attacks according to the attack model. In the third subsection, we discuss the middleperson[4] attack strategy.

2.1 The attack model

The idea behind the attack model we use (inspired from [TFHG99], [P98], [DY83]) is to characterise *what* the attacker may do without *prescribing* a mechanism saying exactly *how* to do it – this is a powerful concept.

The objective of an attacker is to gain some *reward* from a target by accessing or acquiring capabilities or information the target does not otherwise permit them to have. Usually, such a reward is only of any value if the target remains sufficiently unaware that the attacker has gained it – at least until the attacker has exploited the reward. It is therefore important that the attacker acquires its reward *covertly* without alerting the target that it has done so.

The attacker is assumed to:

[4] Also known as a "man-in-the-middle" or "active" attack.

- Interact with their targets purely through the act of sending and receiving messages
- Possess arbitrary memory capacity – they never forget information they have received or inferred.
- Only use keys they have directly acquired, either known initially or extracted from messages.

All keys are assumed *strong* and so considered uncrackable and unguessable by any player. This means that the attacker cannot just freely use an arbitrary key owned by someone else. The only way the attacker can use keys owned by others is to have acquired *direct possession* of them somehow. Naturally, the attacker directly possesses any keys they already own or have generated themselves.

Finally, to enable the attacker to fully interact with their chosen targets, they may:

- Pull apart any received messages, using all the corresponding keys for *decryption* that the attacker has directly available. The attacker has access to the protocol definition and so can interpret message terms to determine the significance of the pieces and which keys are needed to do this.

  As for every principal, recall that opaque encrypted items that aren't immediately decryptable are retained for later re-examination and reuse. This means that newly revealed keys can be tried out to see if they unlock any of the remaining opaque material. This allows the attacker to dynamically exploit exposures and thus to potentially attack explicit key distribution protocols.

- Construct *arbitrary* message terms using:
  - Message terms obtained by analysing already received messages, as above.
  - Message terms freshly generated by the attacker directly.
  - Encoded message blocks, constructed using *(encryption)* keys that the attacker has direct possession of.

An extension of this attack model considers protocols acting in combination and the notion of *unintended services* provided by protocols. An attacker sets up a situation in which (possibly partial) runs of one protocol yields information that the attacker can usefully exploit in attacking another protocol – thus playing one protocol off against another.

2.1.1 Honest Principals verses Intruders and Attackers

Since our attack model characterises what the intruders and attackers could do, this leads to wondering if anything comparable can be said about the non-attackers – namely, the *honest* principals.

Within a protocol definition, the principals are assumed honest – that is, they follow the protocol as themselves and as no one else. Thus, an *honest principal* never knowingly deceives other principals or makes false claims of identity etc. In particular, honest principals will:

- Follow the protocol exactly as stated, without illegally adopting the *role* of some other principal. Of course, an honest principal can adopt any role it has been legitimately granted, perhaps by delegation from some authority.
- Always generate fresh nonce values as required by the protocol and so never "replay" any nonce values generated by other parties as their own.
- Only use keys that they legitimately have access to, as required by the protocol definition.

- Expect all *other* principals to behave in an honest manner.

2.2 Strand spaces and security analysis of protocols

Strand spaces were originally introduced in [TFHG99] to provide a conceptual framework for examining and analysing the security behaviour of protocols. ASPECT also uses this conceptual framework as a starting point for performing its analytic calculations.

The broad idea behind strand spaces is to model the message *activity* of each principal as a linear strand consisting of sequences of send and receive events (i.e. individual histories). As such, strands can represent arbitrary interleavings of message events as viewed from each participant.

Traces consisting of sequences of message transfer events can be straightforwardly translated into a corresponding strand space. Accordingly, there are strand spaces that represent general runs of a protocol.

An important insight of strand spaces is that they emphasise the logical separation of participants in a distributed system. Principals can only directly observe those events that happen to them. Any knowledge involving other participants has to be gleaned and inferred from the evidence obtained from received events that were sent by others.

We illustrate our form of strand spaces below in **Fig 2**. This diagram contains a strand space, represented graphically, corresponding to a simple run of the simple protocol given in **Fig 1**:

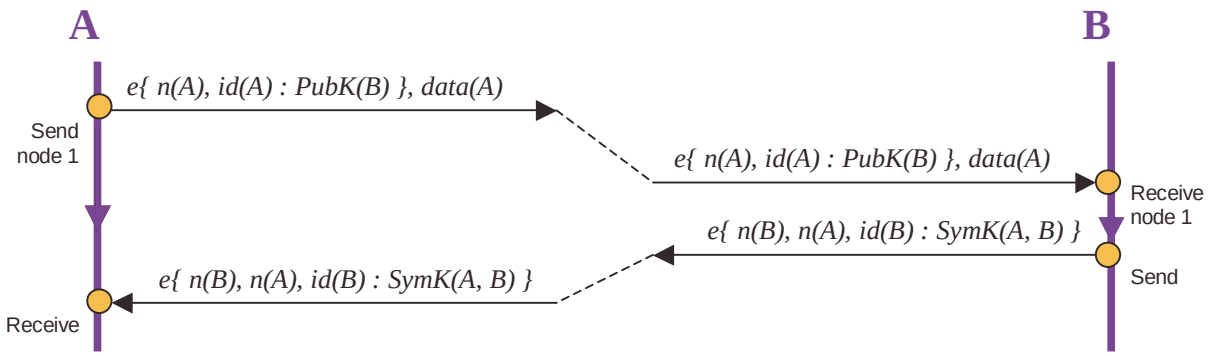

Fig 2: Strand space corresponding to a simple run of the protocol from Fig 1.

The dotted lines in **Fig 2** informally indicate that, although linked, the send and receive events in different strands are not *necessarily* tied to each other. There is the possibility they could be linked to other strands that also happen to match the messages.

By allowing the send and receive events to be weakly linked in this manner, it is possible to envisage how other strands could be *interposed* to somehow interfere with the intended message flow.

Thus, the insertion of additional strands into general protocol runs forms an effective basis for systematically finding attacks on protocols.

Strand spaces also provide a useful conceptual model for calculating the flow of information between different parties.

2.3 The Middleperson Attack Strategy

The attackers main objective is to violate at least one of the security objectives of the protocol, all without getting caught in the act – that is, being discovered by one of the honest principals.

The middleperson attack strategy amounts to *insinuating* extra strands representing the attacker to gain appropriate reward ***and*** without alerting the honest parties. By way of illustration, **Fig 3** below abstractly presents a protocol definition with some middleperson attacks.

What then is a reasonable set of initial conditions, in terms of accessible resources, that enables an attacker to do this? If the attacker already knows everyone's keys, including their secret keys, then arbitrary masquerade attacks are easily feasible. This is a degenerate situation and has no further interest.

The question of what attacks are possible only becomes interesting when we try to find the *smallest* set of the attackers initial conditions that could permit attacks to occur. In which case, there may be more than one set of initial conditions that does this, leading in turn to several distinct attack scenarios.

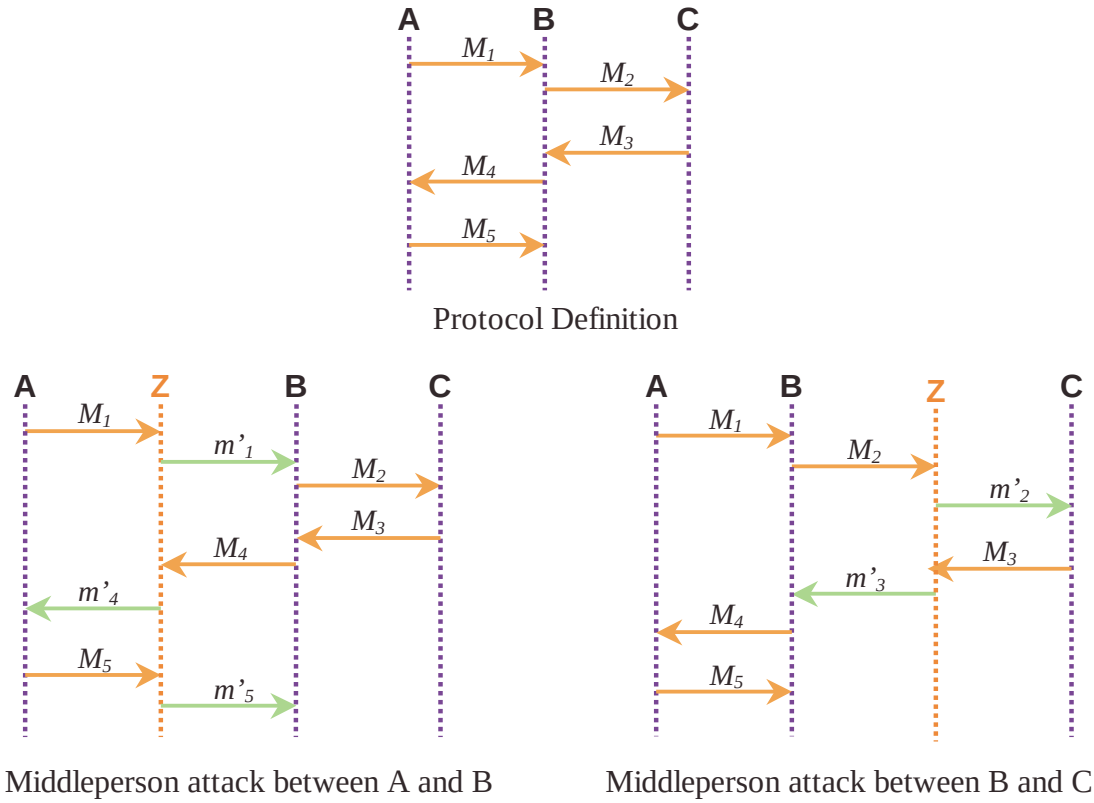

Fig 3: Illustration of a multi-party protocol with a couple of middleperson attacks

2.3.1 You knowing your enemy …and your enemy knowing you right back!

If the attacker cannot know secrets "by magic", then perhaps there are other ways in which the attacker can interact, giving them sufficient advantage to mount an attack.

One way this can happen is for the attacker to have acquired some *legitimate* role known to *some* of the honest parties. The attacker then gathers information somehow to exploit this role. Armed with this information, the attacker then mounts an attack upon some other unsuspecting party. Usually the victim is deceived into treating the attacker as a known honest party – a masquerade.

For our purposes, we consider adding the assumption that the attacker is actually known to one or more of the honest principals. Another principal would then be chosen as a victim. The attack attempt is then made, with the attacker impersonating one of the honest parties to the victim.

The effect of considering this is to increase the possible number of ways of attacking a protocol. This is reflected in the sets of possible initial conditions for the attacker to use.

Supposing that the attacker, *Z*, is attempting to masquerade as principal *A* to principal *B*, we may further assume that *Z* initially knows any *public* information that principal *A* also knew initially (i.e. not their secret keys). Moreover, we can also assume that *Z* acts as a principal in the protocol and thus has the same kind of resources that other, honest, principals have common access to.

We further require that the only secret data any principal (including attackers) can directly know initially is their own. Any protocol that fails to meet this basic requirement is flawed.

3 What does ASPECT do?

In earlier sections, we have broadly described what security protocols are and how to attack them. ASPECT is an efficient proof-of-concept prototype tool that examines protocol definitions and tries to find middleperson attacks by interposing an attacker strand into a protocol run.

Conceptually, ASPECT involves two main phases – Passive Analysis followed by Active Analysis. The block diagram in **Fig 4** describes the main process flow for ASPECT. The next two sections describe the main sub-components of ASPECT.

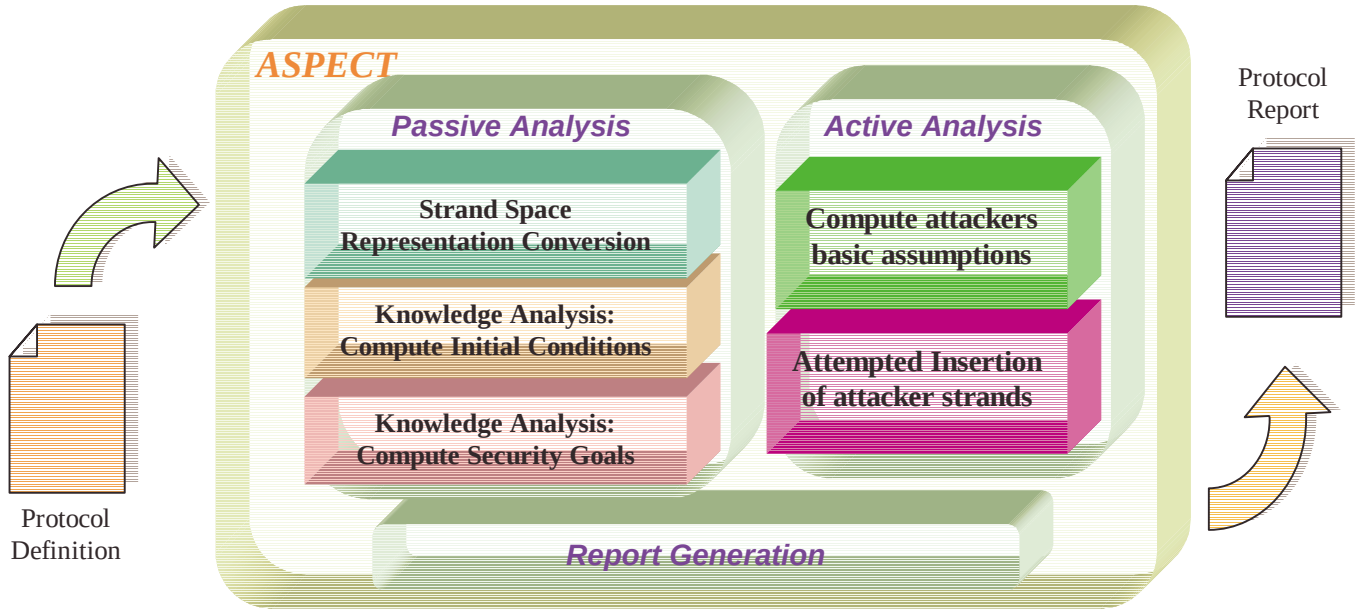

Fig 4: Block diagram describing the top-level structure of ASPECT.

3.1 Passive Analysis

This computes static, structural features of the protocol description. In particular, this determines what each principal directly knows at each stage. From this knowledge, the initial conditions for each (honest) principal can be determined (c.f. *weakest pre-conditions*).

Based on this, passive analysis proposes a number of *conjectured* security goals based upon a comparison of standard high-level security objectives (confidentiality and authentication) and the detailed structure of the protocol (see **Section 1.2**).

We emphasise that although a security protocol should have some security goals (all of which should be met), the exact form of these goals necessarily depends upon the protocol structure and the high-level security objectives that they match and are derived from. It is certainly possible that a given protocol only matches some of the security objectives, whilst not matching others. For example, a given protocol could yield the confidentiality objectives, without also yielding the authentication objectives.

Thus, the passive analysis involves both a dependency analysis and a matching process to determine both initial conditions and the conjectured security goals.

3.2 Active Analysis

This takes the protocol plus the outcomes from its passive analysis and attempts to *disprove* the conjectured security goals proposed. The main strategy used is to try to construct *middleperson* attack strands. To do this, we add a principal representing the attacker, called Z. The Active Analysis then attempts to insinuate a strand for Z into the message flow in such a way that at least one of the conjectured security goals is broken. This typically involves Z capturing some particular, valuable data item. Such data might be used for authentication purposes, thus allowing the attacker to break the authentication goals of the protocol.

Thus, the active analysis is in effect performing a form of reachability analysis that tries to find (complete) protocol runs containing a successful attack strand. However, as mentioned earlier in section 2.2.3, ASPECT is restricted to considering attacks that can be completed within a single run of the protocol.

3.3 Analytic coverage

Finally, in the absence of a completeness result, we **cannot** claim that if ASPECT finds no flaws in a protocol, then there were no flaws to find i.e. protocol correctness. Nevertheless, ASPECT does subject protocols to a critical examination capable of uncovering security flaws. When no flaws are found by ASPECT, then we may improve our confidence in the security of the protocol. Further work is required to characterise more precisely the class of errors that ASPECT is capable of detecting completely.

4 Worked examples

We show what ASPECT can do by means of a sequence of worked examples. Our examples are all based around the familiar Needham-Schroeder-Lowe public-key authentication protocol (see [NS78], [L95], [L96]). In the worked examples below, the principals A and B are taken to be *honest* and the attacker is denoted by Z.

The set-up goes as follows. Imagine a protocol engineer working on adapting a protocol to some new purpose. They may have looked up an existing protocol from somewhere but it doesn't quite fit the situation they have in mind. Now read on …

4.1 First version

The starting point is the simple two-party protocol as given in **Fig 5**:

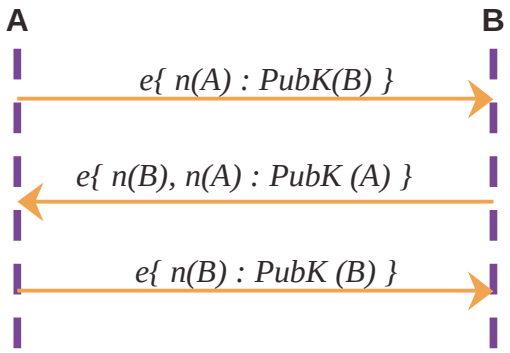

Fig 5: First version of our example protocol

Briefly, the first message is sent by *A* and consists of a nonce value originated by *A*, encrypted using *B*'s public key. Therefore, the only principal that can decrypt this block is *B*. Upon receipt, *B* can extract the nonce value from *A*. The second message is sent by *B* and consists of an encrypted pair of nonce values – a nonce originated by *B* and the nonce sent in the previous message by *A*. The encryption key is *A*'s public key. Finally, the third message is sent by *A* and just consists of the nonce generated by *B*, encrypted using *B*'s public key.

Each of the messages sent could only be decrypted by their respective recipients. The content of the second message proves to *A* that *A*'s own nonce was successfully extracted from the first protocol message that was sent by *A*. The content of the third message proves to *B* that *B*'s own nonce was also successfully extracted from the second protocol message as sent by *B*.

However, this protocol merely exchanges three encrypted items – no evidence is sent that ties any of these messages to the principals involved. Thus, no authentication is possible with this protocol.

4.1.1. Passive Analysis

Passive Analysis of the above finds no errors and determines the following by inference and pattern-matching:

1. Initially, principal *A* directly knows: *id(A), id(B), PubK(B), PubK(A), SecK(A)*

   Initially, principal *B* directly knows: *id(B), id(A), PubK(A), PubK(B), SecK(B)*
2. Both *n(A)* and *n(B)* were only sent in encrypted form, where the decryption keys required are secret.

3. The value *n(A)* was exchanged in a challenge-response pattern between *A* and *B* (i.e. the value was sent from *A* using encryption that only *B* could decode and was later returned back to *A* in a secure and consistent manner..

   Similarly, the value *n(B)* was exchanged in a challenge-response pattern between *B* and *A*.

4. As a security goal, the values *n(A)* and *n(B)* are regarded as being valuable since they are always sent in encrypted form, where the decryption keys required are secret.

   Additionally, the secret keys *SecK(A)* and *SecK(B)* are regarded as being valuable (to their respective owners).

The initial conditions for A and B were each listed under 1 above, and the conjectured security goals are simply to keep the valuable material secure, as listed under 4. This information can be summarised in diagrammatic form, as in **Fig 6** below.

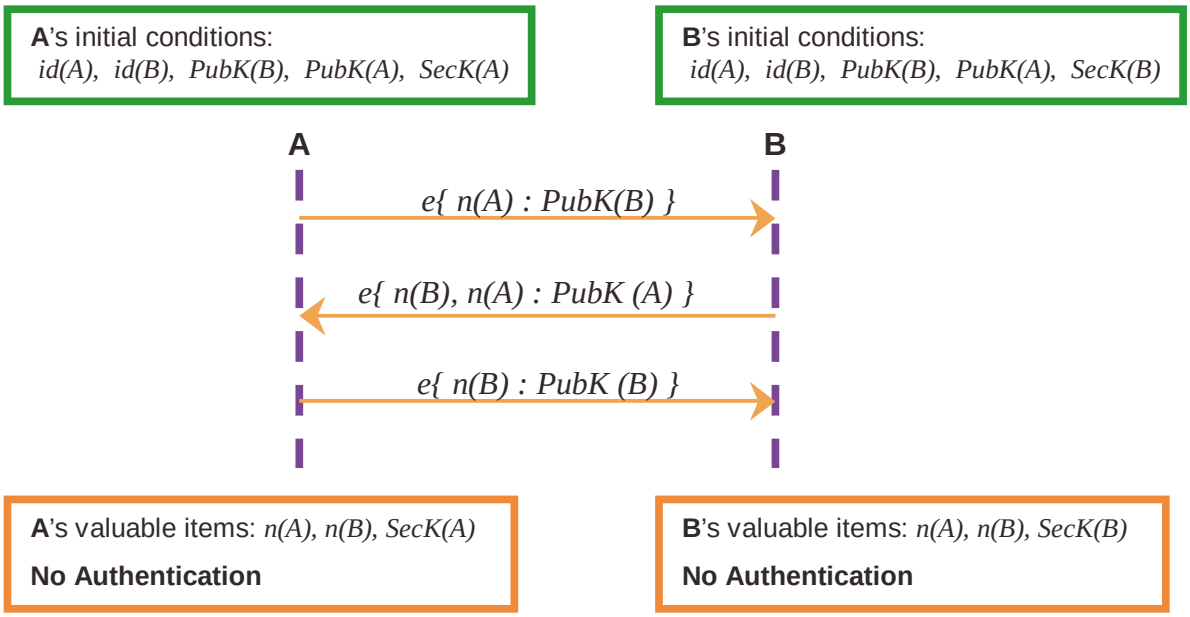

Fig 6: First protocol annotated by outcomes from Passive Analysis.

4.1.2. Active Analysis

Even though public keys were used for encryption, it turns out that there are quite a few attacks possible. Each of the attacks[5] found will occur as some form of masquerade where one of *A* or *B* is known to the attacker, *Z*, and the other is the victim.

The first attack that ASPECT uncovers automatically is as follows:

1. Assume that the following hold:
   a. Principal *A* knows *Z* (and so considers *Z* to be honest).
   b. Principal *A*'s initial conditions are: *id(A), id(Z), PubK(Z), PubK(A), SecK(A)*.
   c. Principal *B*'s initial conditions are: *id(B), id(A), PubK(A), PubK(B), SecK(B)*.
   d. Attacker Z's initial conditions are: *id(Z), id(B), PubK(B), PubK(Z), SecK(Z)*.

[5] Even under the same attack conditions, it turns out that there could be several possible attacks, each with different protocol runs.

The attacker *Z* can now masquerade as *A* to *B*, according to the run given in **Fig 7**.

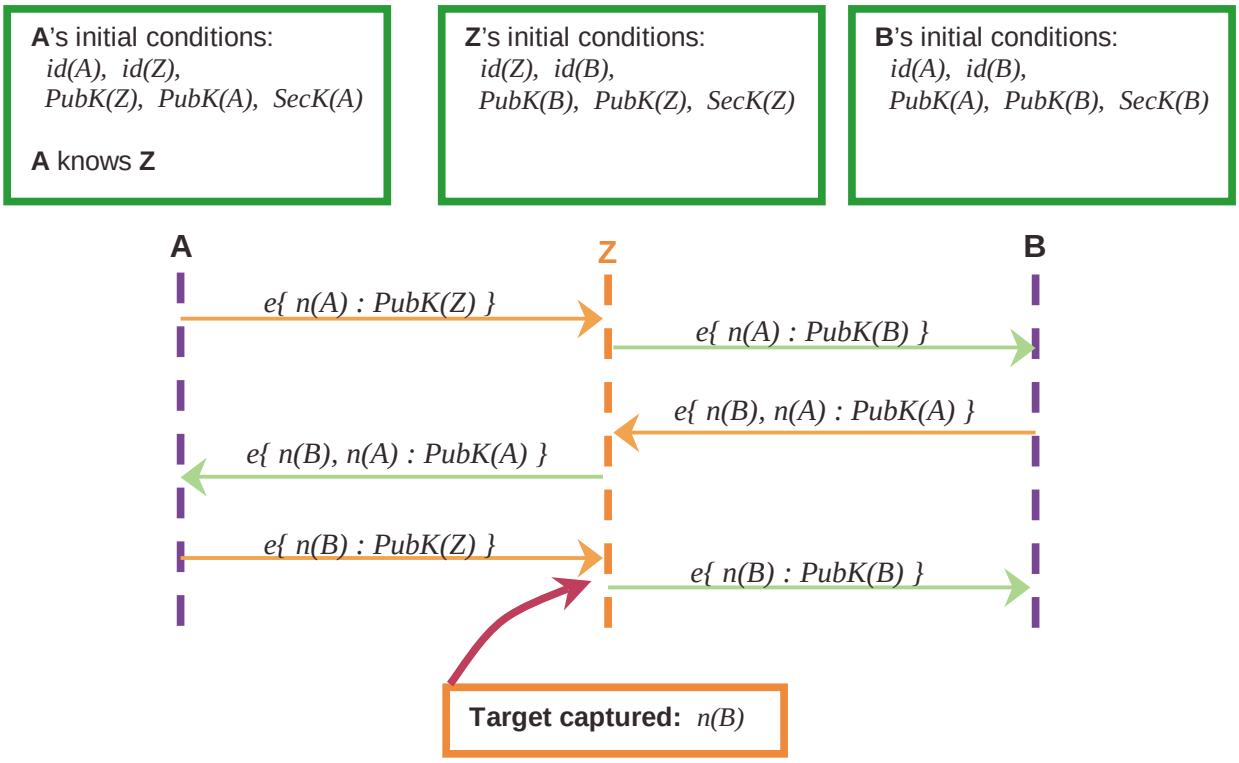

Fig 7: First attack found automatically by ASPECT

Although *A* regards *n(A)* as being valuable, *Z* is known to *A* and so *n(A)* is a commonly known value between them. (If *A* had been aware of it, *A* might be more concerned that *Z* had "leaked" this value to another principal, *B*). Consequently, *n(A)* is not considered a sufficient reward by the attacker *Z*. The real prize for *Z* is in capturing *B*'s valued nonce *n(B)* at the penultimate step.

2. The second attack found against this protocol is as given in **Fig 7**. In this case, *B* knows *Z* (and so considers *Z* to be honest). The attacker *Z* can now masquerade as *B* to *A*.

   This time, the attacker *Z* regards the nonce item *n(A)* issued by the victim, *A*, to be the valued reward. Because *B* knows *Z*, the nonce item *n(B)* is considered by *Z* to be a commonly known item.

It is interesting to compare these two attacks. Firstly, the victims and the rewards gained are different in each case. Secondly, the attacker acquires the reward item at different stages. In the first attack, the reward is acquired at the penultimate step, whereas in the second attack, the reward is learnt after *B* sends the first response. Finally, the attacks also differ in the amount of re-encryption that the attacker needs to do. In the first attack, this is needed whenever *Z* sends to *B* (i.e. twice), whereas in the second attack, re-encryption is only needed once when *Z* sends to *A*.

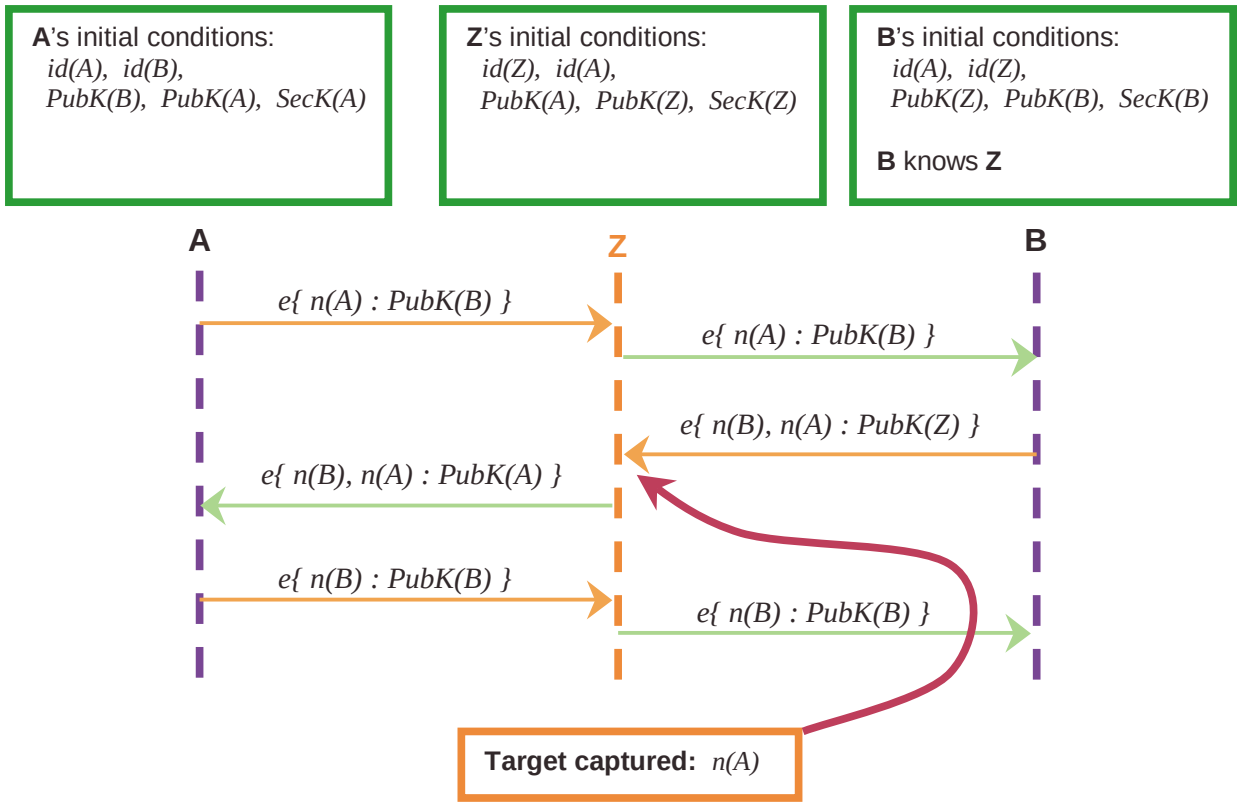

Fig 8: The second attack found by ASPECT

In both attacks, the attacker continues to the end of the protocol, even though the attacker has already acquired the reward. If this did not happen, the run would end prematurely and constitute detection by one or other of the honest parties. In general, the reward item captured by the attacker only has value purely because the victim remains blissfully unaware that the attacker now possesses it.

Overall, ASPECT discovers seven (different) attacks against this simple protocol.

4.2 Second version

The first version of our simple protocol didn't check out too well – so our protocol engineer needs to fix and adapt it. We notice that ASPECT didn't recognise the previous protocol as an authentication protocol – no identity claims were exchanged. If we add these claims to the protocol, then perhaps that will make some difference. Lets try it out and see …

The second version is as given in **Fig 9**.

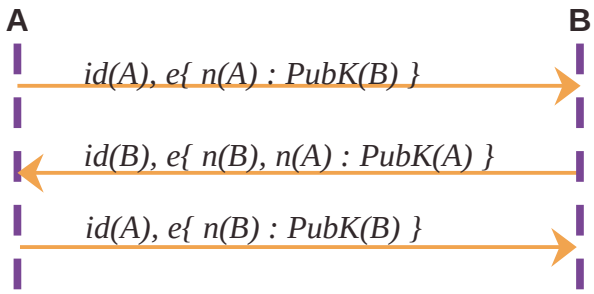

Fig 9: Second version of protocol

The experienced protocol developer will no doubt see that these identity claims are not sufficiently "tied" to the messages for secure authentication. Anyway, lets see what ASPECT makes of it.

4.2.1 Passive Analysis

Passive Analysis of the above finds no errors and is mostly identical to the passive analysis of the previous protocol. The outcomes are mostly summarised in **Fig 10**.

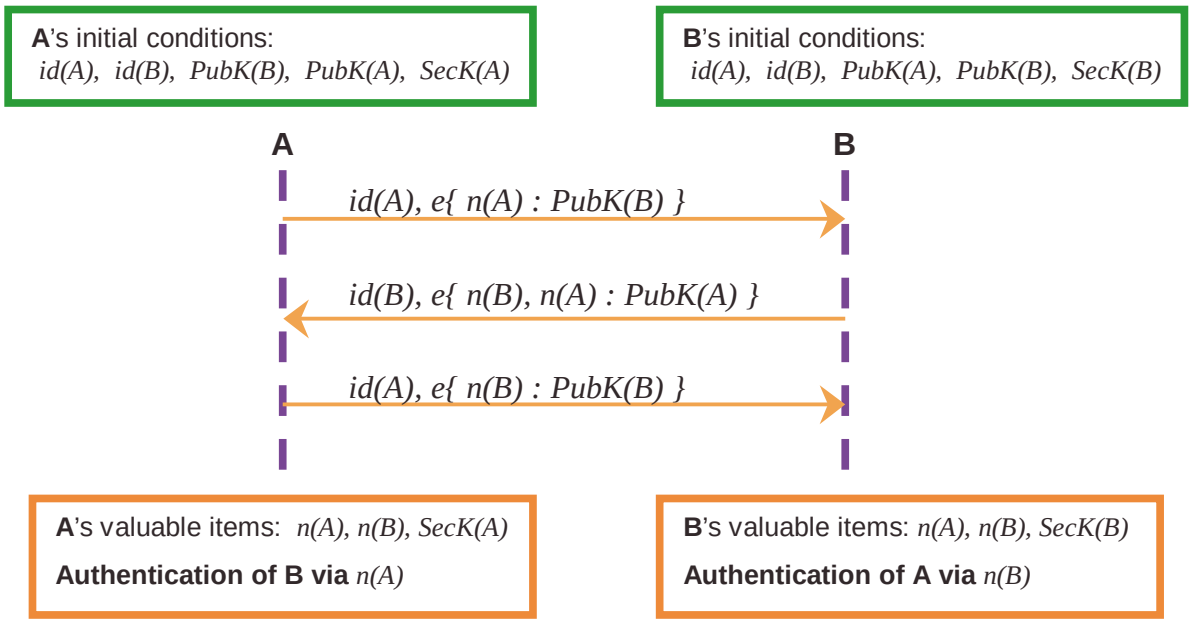

Fig 10: Second version annotated with outcomes from Passive Analysis.

The main difference is that, as hoped, the protocol might be an authentication protocol, since:

1. *A* is regarded as authenticating *B* via valuable item, *n(A)*. This is because *B* makes an identity claim to *A* **and** there is a challenge-response pattern between *A* and *B* that is **witnessed** by a valuable data item, *n(A)*.

   Similarly, *B* is regarded as authenticating *A* via valuable item, *n(B)*.

The conjectured security goals are

a. Keep the valuable material secure.

b. Establish mutual authentication between A and B and vice versa, as listed by 5.

4.2.2 Active Analysis

It turns out that, as suspected, there is a masquerade attack on this revised protocol – and this is found by ASPECT. The attack is given in **Fig 11**. Principal *A* knows *Z* (and so considers *Z* to be honest), and the attacker *Z* can now masquerade as *A* to *B*.

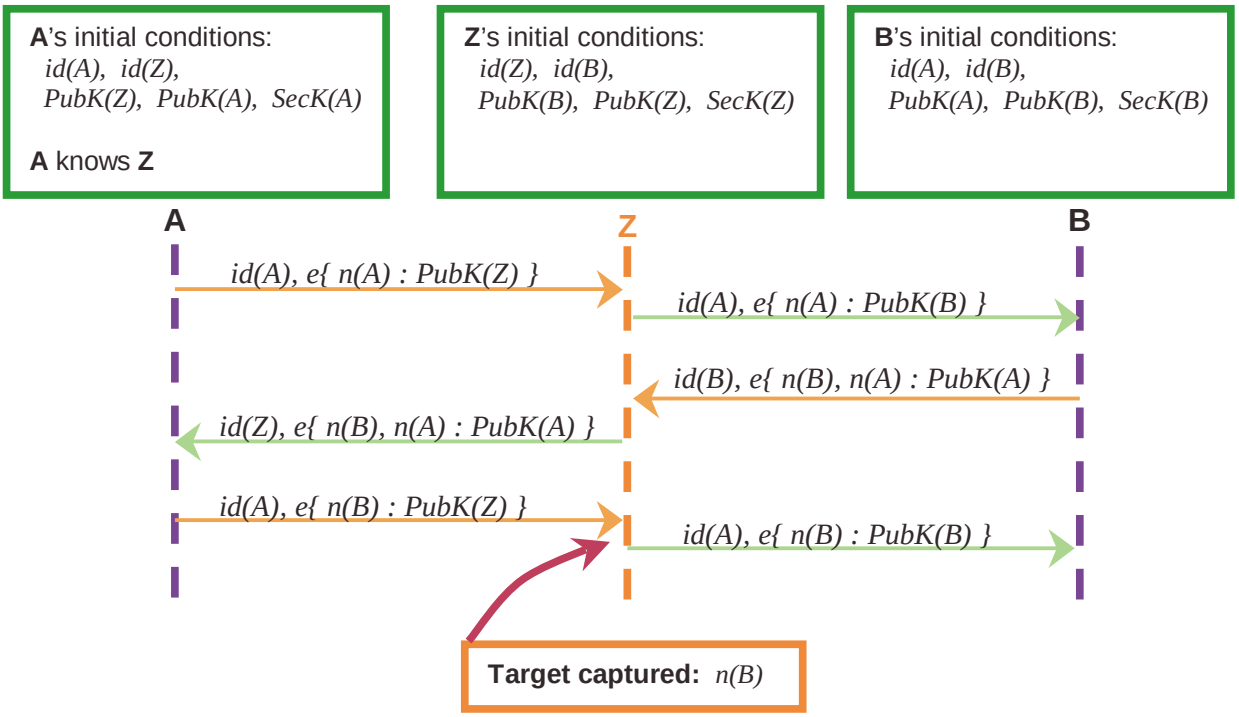

Fig 11: First attack on second protocol

This is very similar to the initial attack on the earlier protocol. Again, *Z*'s target is *B*. However, the bar is now higher, since the protocol is required to achieve authentication. Thus, the attackers goal is to succeed in impersonating *A* to *B* – it is clearly insufficient just to capture secure data when one of the honest principals detects something wrong due to a failed authentication check. At this point, the attacker has failed because the attack has been uncovered.

Now, the protocol correctly authenticates *Z* to *A* as *Z*, because, as far as *A* can observe, the required identity-claims match and the challenge-response was explicitly witnessed by nonce *n(A)*. Indeed, even though *Z* calmly serves *B*'s nonce value as *Z*'s own, *A* cannot discriminate or observe any difference – random bits generated by one principal are just like random bits generated by any other.

However, the real problem is that *Z* is authenticated to *B* as *A*, because, as far as *B* can observe, the required identity-claims match and the challenge-response was also explicitly witnessed, this time by nonce *n(B)*. Naturally, to achieve this, Z had to capture the valuable nonce term *n(B)*.

Thus, this protocol did not correctly authenticate the principals – the attacker succeeded and the protocol is therefore flawed.

In fact, ASPECT finds two attacks in total. The second attack discovered is very similar to the corresponding attack on the previous protocol – the only point to say is that, in this attack, the attacker *Z* is successfully authenticated as *B* to *A*, instead.

A remaining question is what happened to the other 5 successful attacks upon the original protocol? The short answer is that the improved protocol now has to meet a greater challenge – namely, authentication. In the remaining attacks, the attacker failed to capture any valuable data teams and/or failed to achieve a successfully deceptive authentication.

4.3 Third version

Our protocol engineer now thinks that they are on the right track – at least, the second version is now considered as an authentication protocol of some kind. Unfortunately, it didn't quite work. The attack on the second protocol showed how the attacker could modify the first identity-claim to push his attack forward. Perhaps we should make that hard for the attacker to do something with. Why not push the first identity-claim inside the encrypted item, which is after all opaque to them? Lets try that out …

The third version of the protocol is as given in **Fig 12** below.

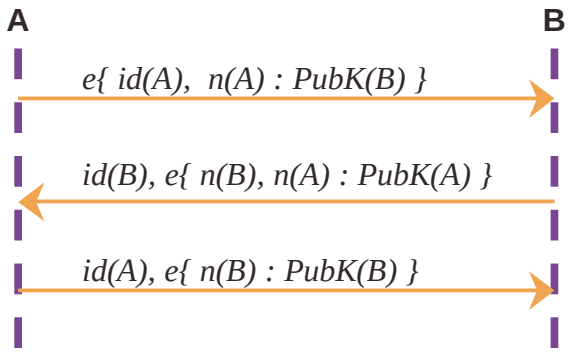

Fig 12: Third version of protocol

To save space and avoid tedious repetition, we shall instead summarise how the outcomes of the analyses differ from what we have already seen so far.

4.3.1 Passive Analysis

There is ***no*** apparent change in outcome for Passive Analysis between the second and the third versions – they are *identical* from the point of view of Passive Analysis!

The reason for this is straightforward – the only change between the second and third protocol is that *id(A)* is moved inside the encrypted item in the first protocol message. But the remaining occurrence of *id(A)* is not encrypted and is thus exposed. Hence, *id(A)* is not securely transmitted in this protocol, and cannot be treated as a valuable data item.

Furthermore, the presence of identity-claims for both *A* and *B* and the challenge-response patterns for *n(A)* and *n(B)* are exactly the same in each version. Consequently, this leads to exactly the same conjectures of security objective being made in both cases.

4.3.2 Active Analysis

Even though Passive Analysis found no differences, Active Analysis does uncover differences between the second and third versions. Only a single attack is found for the current version, whereas two different attacks were found on the previous version. Again, the attack found here is very similar to the original attack found earlier. In particular, the attacker *Z* succeeded in impersonating *A* to *B*.

We can now see the main difference between the two versions of the protocol – by moving the identity-claim *id(A)* into the first encrypted item, this effectively prevented the *second* attack, but not the first.

The previously successful attack fails in this case because the attacker cannot modify the identity-claim from *A* since it is now embedded inside an opaque encrypted item. When *B* subsequently

decrypts this item, *B* discovers that the identity-claim is from *A* and not from *Z* after all. Unfortunately, *B* doesn't have a public-key from *A* – and so B cannot make the next move in the protocol. This would-be attack now fails.

Even if *B* had already known *A* directly in this case, *B* would just have used *A*'s known public-key to encrypt the second message – but then *Z* wouldn't have a way into the message stream. Again, the attack fails.

4.4 Fourth version

Our protocol engineer now realises that the attacker gained advantage by manipulating either of the identity-claims – not just the first. One of the previous attacks was foiled by encrypting one of the identity-claims – perhaps that tactic might help here since then the attacker has fewer things to manipulate. Lets try this one out …

The fourth version of the protocol is given in **Fig 13** below

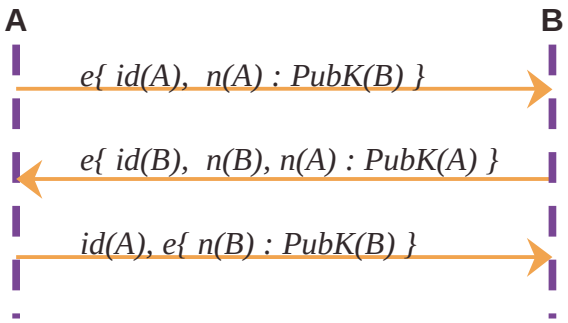

Fig 13: Fourth version of protocol

As before, we summarise how the outcomes of the analyses differ from what we have seen earlier.

4.4.1 Passive Analysis

The only change over previous outcomes is that the identity-claim, *id(B)*, is now regarded as a valuable item by both *A* and *B*, since it's only occurrence in the protocol lies within an encrypted item whose decryption keys are secret. The item *id(B)* is considered to be valuable even though both principals already directly knew it from the start. The concept of "valuableness" here models the concern with which an item is handled by the principals.

4.4.2 Active Analysis

As anticipated, the big difference here is that Active Analysis finds no flaw in the above version of the protocol. The one attack remaining from the previous version has now been defeated!

It is instructive to see how the above change prevents the final remaining attack. The problem for the attacker, *Z*, is that the second protocol message cannot now be tampered with – because *Z* cannot now decrypt the block, without otherwise deceiving *A* to do so as an *unintended service*.

In particular, the attacker *Z* needs to change the identity-claim data, *id(B)*, now embedded in the encrypted item for the attack to succeed. To see this, consider what would happen if *A* did receive the second protocol message originally from *B*, and forwarded via *Z*.

After decrypting it, the identity-claim from *B* is now a dead giveaway since *A* did not previously know *B* and certainly does not have access to *B*'s public key. In particular, *A* had previously sent the first message using *Z*'s public-key and hence *A* had to know both *id(Z)* and *PubK(Z)*. Because

the public keys associated with *Z* and *B* are necessarily distinct, their identity claims *id(Z)* and *id(B)* are in turn necessarily distinct. Accordingly, *A* can now see an unavoidable clash.

There is a further, but minor, refinement – the identity-claim *id(A)* is still not considered to be secure or valuable because its occurrence in the third protocol message is still *in clear*. This can be remedied in one of two ways – either by deleting it or by moving it inside the associated encrypted item. ASPECT finds no flaws with either of these alternatives.

4.5 Performance figures

The ASPECT prototype processes all of the worked examples above in a rapid, efficient and completely automatic manner. ASPECT takes as input a description of the protocol to be checked and produces a report containing the outcomes of the Passive and Active analyses.

Even though ASPECT was implemented as a proof-of-concept prototype, the dramatic performance obtained demonstrates the effectiveness of our implementation strategy. The timing information given below in **Table 2** was obtained by running ASPECT on a HP OmniBook laptop (running at approx. 700 MHz) with 250Mb store and 11.5Gb hard disk.

The amount of actual store dynamically used appears to have been very small – a pessimistic estimate suggests that around 4 MB was actually consumed based upon system statistics that gave the amount of store garbage collected/reclaimed and the current store allocation at the end of each run.

	Overall Time (to 1st attack)	**Overall time (for all attacks)**
Example 3.1	0.21 secs	5.96 secs (7 attacks)
Example 3.2	0.4 secs	16.28 secs (2 attacks)
Example 3.3	0.32 secs	8.34 secs (1 attack)
Example 3.4	- N/A -	4.58 secs (NO attacks)

Table 2: Performance timings for the worked examples

It is indeed encouraging that simple protocols such as these can be analysed quickly and effectively. Although of some interest, it is also true that these results only represent a small, finite sample. It is already known that the bounded reachability problem for security protocols is (only!) NP-complete [RT01]. Although this result may not be directly applicable to ASPECT, this suggests that, as protocol examples increase in complexity, their analysis will eventually take overwhelmingly more resources to process. For example, increasing the number of distinct nonce values exchanged also increases the number of choices available to the attacker, most likely making protocol analysis much harder to perform.

Further investigation is called for – to more precisely characterise the worst-case and average-case complexity performance of tools such as these and to consider different trade-offs in the analytic complexity of the protocols themselves.

Conclusions and further work

We have introduced a proof-of-concept security protocol analysis tool called ASPECT. We have illustrated its effective operation through a linked sequence of worked examples. Prototypes such as ASPECT offer some hope that engineering tools for checking security protocols could become a practical reality in the not-too-distant future.

There are a number of interesting and promising ways to extend and develop our work:

1. The main question concerns whether the underlying Active Analysis algorithms are *complete*. Completeness here means that if no attacks were uncovered by the analysis, then there were no attacks to find. The converse to this implies that if an attack did exist, then the analysis must report that fact.

   The basic theoretical issue involves showing that the finite set of attack attempts that ASPECT discovers effectively covers all the possible ways of constructing an attack strand. If that is the case, then any actual attack strand would be covered by at least one of these attack attempts.

   Now, there is encouraging evidence that the underlying algorithms used by ASPECT will in fact turn out to be complete. A suitable mathematical framework inspired by Strand Spaces for exploring questions such as these can enable the development of a better understanding of the interactions between different roles. An exploration of this evidence leading hopefully to a proof of completeness is work-in-progress for a forthcoming report.

2. Extending ASPECT to support the analysis of a more expressive range of protocol elements such as hashing, digital signatures, key exchange functions, non-atomic key expressions and the like. An important addition is to explicitly include logical test expressions within protocol descriptions, thus allowing the underlying decision structure for a protocol to be stated more explicitly.

   Such extensions naturally bring with them the need to extend the range of security property to check for e.g. integrity, forward security, etc.

3. Include the ability to *pre-specify* the expected characteristics of a given protocol – i.e. allow the user to provide a specification as an explicit statement of required properties. We anticipate that this can be smoothly integrated with our Passive Analysis approach.

   A further extension is to include options concerning the type of attack to be considered. For example, if a protocol run can reveal persistent, confidential data (e.g. non-nonce items) then the protocol run does not need to complete – the attacker gains reward as soon as the confidential data is captured.

4. At present, ASPECT only deals with *single session* attacks. An interesting extension would be to provide explicit support for analysing multiple session attacks. With this support available, this may help in analysing interactions between several protocols c.f. the so-called *interleaving attacks* and *unintended services*.

5. Another avenue is to investigate concerns relaxing the *determinism* constraint on security protocols – this would open up a much wider range of potential applications where strict synchrony cannot be guaranteed, but where weaker assumptions can be.

Our initial investigations were inspired by the Strand Space approach developed by Joshua Guttman and his colleagues at MITRE [TFHG99]. The concepts underlying Strand Spaces provide a natural structure in which to formulate security properties, focusing upon the notions of *agents* as represented by strands that possess certain *roles* and capabilities for action. These concepts provide

a persistent framework in which to model what is *learned* (or *acquired*) by agents at each stage, as a result of message transfers between the individual participants.

In short, the security properties of protocols are less to do with concurrent behaviour interactions, and far more to do with the interactions between the different *roles* held by participating agents and what is *expected* of them by other participants.

Acknowledgements

This work has benefited from the many helpful conversations and provocative comments from fellow researchers in HP Labs and the Trust, Security and Privacy group in particular.

I thank both Martin Sadler and Keith Harrison for actively supporting and encouraging the research into protocol engineering tools and related theory. This project would never have happened without their initial insights and appreciation of the research opportunities and challenge. Pete Bramhall provided much project support and sage advice – thanks also to Pete for reengineering the acronym for ASPECT! Along Lin contributed comments at an early stage concerning representations of protocol definitions within Prolog.

This report has been improved by comments and remarks from my colleagues Adrian Baldwin, Pete Bramhall, Liquin Chen, Patrick Goldsack, Antonio Lain, Stephen Crane, Siani Pearson, Martin Sadler, and David Soldera.

References

[A01] R. Anderson, Security Engineering, Wiley, 2001

[AG98] M. Abadi, A.Gordon, A calculus for cryptographic protocols: the Spi Calculus, DEC-SRC Tech. Report SRC-149, 1998

[AL00] R. Amadio and D. Lugiez, On the reachability problem in cryptographic protocols, in *CONCUR* (2000), vol. 1877 of LNCS, Springer-Verlag, 380-394.

[BAN] M. Burrows, M. Abadi, and R. Needham, A logic of authentication, in *Proceedings of the Royal Society of London A*, 426:233-271, 1989. Also publ. (condensed) in *ACM Transactions on Computer Systems*, 8(1): 18-36, February 1990.

[CAPSL] G.Denker, J.Millen, CAPSL intermediate language, In *Proc. FLoC Workshop on Formal Methods and Security Protocols*, 1999

[CJ97] J. Clarke, and J. Jacob, A survey of authentication protocol literature, http://www.cs.york.ac.uk/~jac/papers/drareview.ps, 1997.

[CS02] H. Comon and V. Shmatikov, Is it possible to decide whether a cryptographic protocol is secure or not?, To appear in *Journal of Telecommunications and Information Technology*, 2002.

[CDMLS] I. Cervesato, N. Durgin, J. Mitchell, P. Lincoln, A. Scedrov, Relating Strands and Multiset Rewriting for Security Protocol Analysis, In *Proc. 15th IEEE Computer Security Foundations Workshop* (2000), 35-51.

[DLMS] N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov, Undecidability of bounded security protocols, In *Proc. FLOC Workshop on Formal Methods in Security Protocols*, Trento, Italy, 1999.

[DM99] N.A.Durgin and J. C. Mitchell, Analysis of Security Protocols, In *Calculational System Design, Series F: Computer and Systems Sciences*, Vol 173, IOS Press, 1999.

[DY83] D. Dolev, and A.Yao, On the security of public key protocols, Technical Report No. STAN-CS-81-854, Dept. of Computer Science, Stanford University, May 1981. Also in *Transactions on Information Theory*, 29(2):198-208, 1983.

[FA01] M. Fiore, and M. Abadi, Computing symbolic models for verifying cryptographic models, in *14th IEEE Computer Security Foundations Workshop* (2001), pp. 160-173.

[HAC97] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.

[L95] G. Lowe, An attack on the Needham-Schroeder public key authentication protocol, *Information Processing Letters*, 56(3):131 – 136, November 1995.

[L96] G. Lowe, Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR, In *TACAS* (1996), vol. 1055, LNCS, Springer-Verlag, 147-166.

[NS78] R. Needham and M. Schroeder, Using encryption for authentication in large networks of computers. *Communications of the ACM*, 12(12):993-999, 1978.

[Oxy] M. Dertouzos, The Future of Computing, *Scientific American*, 281(2), 36-39, August 1999.

[P98] L. Paulson, The inductive approach to verifying cryptographic protocols, Journal of Computer Security, 6, 1(1998), 85-128.

[P2P] A. Oram (ed), Peer-To-Peer Harnessing the Benefits of Disruptive Technologies, O'Reilly & Associates, Inc., March 2001.

[RS01] P. Ryan and S. Schneider, Modelling and Analysis of Security Protocols, Addison-Wesley, 2001.

[RT01] M.Rusinowitch, and M. Turuani, Protocol Insecurity with Finite Number of Sessions is NP-complete, In *Proc. 14th IEEE Computer Security Foundations Workshop* (2001), 174-187.

[S99] D. Song, Athena: a new efficient automatic checker for security protocol analysis, In *12th IEEE Computer Security Foundations Workshop* (1999), 192-202.

[Sc96] B. Schneier, Applied Cryptography, 2nd Ed., Wiley, 1996.

[SRI] J. Millen and V. Shmatikov, Constraint solving for bounded process cryptographic protocol analysis, in *Proc. 8th ACM Conference on Computer and Communications Security*, ACM, 2001.

[TFHG99] F. J. Thayer Fábrega, J. C. Herzog, J. D. Guttman, Strand Spaces: Proving Security Protocols Correct, in *Journal of Computer Security*, 7:191-230, 1999.

[TvS02] A. S. Tanenbaum, M. van Steen, Distributed Systems – Principles and Paradigms, Prentice Hall, 2002.

Appendix: Example report

The report listed below was generated by ASPECT and analyses the worked example protocol discussed in **Section 4.1**.

```
ASPECT - Version 0.3 - January 2002
Proof-of-concept version (Passive & Active Analysis)
(C) Hewlett-Packard 2002.

Report file: c:/users/brimon/prolog/pchk.rep
Report generated at Tue Jan 22 13:08:26 2002

Principals : a, b

Protocol definition (in diagram form) :
---------------------------------------
     1.   a ---> b      :   E{ N(a) : PubK[b] }.
     2.   b ---> a      :   E{ N(b), N(a) : PubK[a] }.
     3.   a ---> b      :   E{ N(b) : PubK[b] }.


+-------------------------+
|                         |
| Passive analysis report |
|                         |
+-------------------------+


Initial Assumptions Discovered:
-------------------------------

   Principal a :
      Assumed identities: id(a) id(b)
      Assumed keys: SecK[a] PubK[a] PubK[b]

   Principal b :
      Assumed identities: id(a) id(b)
      Assumed keys: SecK[b] PubK[a] PubK[b]


Freshly generated and issued items - nonces and keys (computed)
---------------------------------------------------------------

   Principal a :
      Nonce terms generated: N(a)
      No keys were freshly generated.
      No keys were issued.

   Principal b :
      Nonce terms generated: N(b)
      No keys were freshly generated.
      No keys were issued.


Securely received message items (computed)
------------------------------------------

   Principal a :
         Item: N(a)
         Encryption keys used:
            PubK[a]

         Item: N(b)
         Encryption keys used:
            PubK[a]


   Principal b :
         Item: N(a)
         Encryption keys used:
            PubK[b]

         Item: N(b)
         Encryption keys used:
            PubK[b]


Valuable message items:
-----------------------

   Principal a :
```

```
      N(a)
      N(b)
      SecK[a]

   Principal b :
      N(a)
      N(b)
      SecK[b]


Authentications:
----------------

   Principal a :
      No authentications

   Principal b :
      No authentications


+-------------------------------------------------------------+
|                                                             |
| No errors were found by Passive Analysis of this protocol. |
|                                                             |
+-------------------------------------------------------------+


-------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------


          //////////////////////////
          /                      /
          / Active Analysis report /
          /                      /
          //////////////////////////


Honest Principals    : a, b
Dishonest Principals : Z


--------------------------------------------------------------------------


Attack information:
-------------------
Attacker 'Z' masquerades as 'a' to 'b'.
Attacker 'Z' is known to 'a'.


Assumptions:
------------

   Principal Z :
      Identities: id(Z) id(b)
      Keys:       SecK[Z] PubK[Z] PubK[b]

   Principal a :
      Identities: id(Z) id(a)
      Keys:       SecK[a] PubK[Z] PubK[a]

   Principal b :
      Identities: id(a) id(b)
      Keys:       SecK[b] PubK[a] PubK[b]


Attack transactions:
--------------------

**** SUCCESSFUL ATTACK FOUND ****
     1.   a ---> Z      :   E{ N(a) : PubK[Z] }.
     2.   Z(a) ---> b   :   E{ N(a) : PubK[b] }.
     3.   b ---> Z(a)   :   E{ N(b), N(a) : PubK[a] }.
     4.   Z ---> a      :   E{ N(b), N(a) : PubK[a] }.
     5.   a ---> Z      :   E{ N(b) : PubK[Z] }.
     6.   Z(a) ---> b   :   E{ N(b) : PubK[b] }.

Captured valued items:
----------------------
   Captured item 'N(a)' was known to [a, b]
   Captured item 'N(b)' was known to [a, b]


--------------------------------------------------------------------------


Attack information:
-------------------
Attacker 'Z' masquerades as 'a' to 'b'.


Assumptions:
------------

   Principal Z :
      Identities: id(Z) id(b)
      Keys:       SecK[Z] PubK[Z] PubK[b]

   Principal a :
      Identities: id(a) id(b)
      Keys:       SecK[a] PubK[a] PubK[b]

   Principal b :
      Identities: id(a) id(b)
      Keys:       SecK[b] PubK[a] PubK[b]


Attack transactions:
--------------------

--------------------------------------------------------------------------


Attack information:
```

```
------------------
Attacker 'Z' masquerades as 'b' to 'a'.
Attacker 'Z' is known to 'b'.


Assumptions:
------------

   Principal Z :
      Identities: id(Z) id(a)
      Keys:       SecK[Z] PubK[Z] PubK[a]

   Principal a :
      Identities: id(a) id(b)
      Keys:       SecK[a] PubK[a] PubK[b]

   Principal b :
      Identities: id(Z) id(b)
      Keys:       SecK[b] PubK[Z] PubK[b]


Attack transactions:
--------------------

**** SUCCESSFUL ATTACK FOUND ****
     1.   a ---> Z(b)   :   E{ N(a) : PubK[b] }.
     2.   Z ---> b      :   E{ N(a) : PubK[b] }.
     3.   b ---> Z      :   E{ N(b), N(a) : PubK[Z] }.
     4.   Z(b) ---> a   :   E{ N(b), N(a) : PubK[a] }.
     5.   a ---> Z(b)   :   E{ N(b) : PubK[b] }.
     6.   Z ---> b      :   E{ N(b) : PubK[b] }.

Captured valued items:
----------------------
   Captured item 'N(a)' was known to [a, b]
   Captured item 'N(b)' was known to [a, b]


------------------------------------------------------------------------


Attack information:
-------------------
Attacker 'Z' masquerades as 'b' to 'a'.


Assumptions:
------------

   Principal Z :
      Identities: id(Z) id(a)
      Keys:       SecK[Z] PubK[Z] PubK[a]

   Principal a :
      Identities: id(a) id(b)
      Keys:       SecK[a] PubK[a] PubK[b]

   Principal b :
      Identities: id(a) id(b)
      Keys:       SecK[b] PubK[a] PubK[b]



Attack transactions:
--------------------

------------------------------------------------------------------------


Attack information:
-------------------
Attacker 'Z' masquerades as 'a' to 'b'.
Attacker 'Z' masquerades as 'b' to 'a'.
Attacker 'Z' is known to 'a'.


Assumptions:
------------

   Principal Z :
      Identities: id(Z) id(a) id(b)
      Keys:       SecK[Z] PubK[Z] PubK[a] PubK[b]

   Principal a :
      Identities: id(Z) id(a)
      Keys:       SecK[a] PubK[Z] PubK[a]
```

```
   Principal b :
      Identities: id(a) id(b)
      Keys:       SecK[b] PubK[a] PubK[b]

Attack transactions:
--------------------

**** SUCCESSFUL ATTACK FOUND ****
     1.   a ---> Z      :   E{ N(a) : PubK[Z] }.
     2.   Z(a) ---> b   :   E{ N(a) : PubK[b] }.
     3.   b ---> Z(a)   :   E{ N(b), N(a) : PubK[a] }.
     4.   Z ---> a      :   E{ N(b), N(a) : PubK[a] }.
     5.   a ---> Z      :   E{ N(b) : PubK[Z] }.
     6.   Z(a) ---> b   :   E{ N(b) : PubK[b] }.

Captured valued items:
----------------------
   Captured item 'N(a)' was known to [a, b]
   Captured item 'N(b)' was known to [a, b]


**** SUCCESSFUL ATTACK FOUND ****
     1.   a ---> Z      :   E{ N(a) : PubK[Z] }.
     2.   Z(a) ---> b   :   E{ N(a) : PubK[b] }.
     3.   b ---> Z(a)   :   E{ N(b), N(a) : PubK[a] }.
     4.   Z ---> a      :   E{ N(Z), N(a) : PubK[a] }.
     5.   a ---> Z      :   E{ N(Z) : PubK[Z] }.
     6.   Z(a) ---> b   :   E{ N(b) : PubK[b] }.

Captured valued items:
----------------------
   Captured item 'N(a)' was known to [a, b]
   Captured item 'N(b)' was known to [a, b]


------------------------------------------------------------------------------

Attack information:
-------------------
Attacker 'Z' masquerades as 'a' to 'b'.
Attacker 'Z' masquerades as 'b' to 'a'.
Attacker 'Z' is known to 'b'.


Assumptions:
------------

   Principal Z :
      Identities: id(Z) id(a) id(b)
      Keys:       SecK[Z] PubK[Z] PubK[a] PubK[b]

   Principal a :
      Identities: id(a) id(b)
      Keys:       SecK[a] PubK[a] PubK[b]

   Principal b :
      Identities: id(Z) id(b)
      Keys:       SecK[b] PubK[Z] PubK[b]


Attack transactions:
--------------------

**** SUCCESSFUL ATTACK FOUND ****
     1.   a ---> Z(b)   :   E{ N(a) : PubK[b] }.
     2.   Z ---> b      :   E{ N(a) : PubK[b] }.
     3.   b ---> Z      :   E{ N(b), N(a) : PubK[Z] }.
     4.   Z(b) ---> a   :   E{ N(b), N(a) : PubK[a] }.
     5.   a ---> Z(b)   :   E{ N(b) : PubK[b] }.
     6.   Z ---> b      :   E{ N(b) : PubK[b] }.

Captured valued items:
----------------------
   Captured item 'N(a)' was known to [a, b]
   Captured item 'N(b)' was known to [a, b]



**** SUCCESSFUL ATTACK FOUND ****
     1.   a ---> Z(b)   :   E{ N(a) : PubK[b] }.
     2.   Z ---> b      :   E{ N(Z) : PubK[b] }.
     3.   b ---> Z      :   E{ N(b), N(Z) : PubK[Z] }.
     4.   Z(b) ---> a   :   E{ N(b), N(a) : PubK[a] }.
     5.   a ---> Z(b)   :   E{ N(b) : PubK[b] }.
     6.   Z ---> b      :   E{ N(b) : PubK[b] }.

Captured valued items:
```

```
----------------------
   Captured item 'N(a)' was known to [a, b]
   Captured item 'N(b)' was known to [a, b]


**** SUCCESSFUL ATTACK FOUND ****
     1.   a ---> Z(b)   :   E{ N(a) : PubK[b] }.
     2.   Z ---> b      :   E{ N(Z) : PubK[b] }.
     3.   b ---> Z      :   E{ N(b), N(Z) : PubK[Z] }.
     4.   Z(b) ---> a   :   E{ N(Z), N(a) : PubK[a] }.
     5.   a ---> Z(b)   :   E{ N(Z) : PubK[b] }.
     6.   Z ---> b      :   E{ N(b) : PubK[b] }.

Captured valued items:
----------------------
   Captured item 'N(a)' was known to [a, b]
   Captured item 'N(b)' was known to [a, b]


------------------------------------------------------------------------------


Attack information:
-------------------
Attacker 'Z' masquerades as 'a' to 'b'.
Attacker 'Z' masquerades as 'b' to 'a'.


Assumptions:
------------

   Principal Z :
      Identities: id(Z) id(a) id(b)
      Keys:       SecK[Z] PubK[Z] PubK[a] PubK[b]

   Principal a :
      Identities: id(a) id(b)
      Keys:       SecK[a] PubK[a] PubK[b]

   Principal b :
      Identities: id(a) id(b)
      Keys:       SecK[b] PubK[a] PubK[b]


Attack transactions:
--------------------


+------------------------------------------------------------------------+
|                                                                        |
| 7 successful attacks were found by Active Analysis of the protocol.    |
|                                                                        |
+------------------------------------------------------------------------+
```