Formal Analysis of Cryptographic Protocols

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

by

Aldrin John D'Souza

(Roll No. 01305009)

Supervisor

Prof. G. Sivakumar

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

December 2002

Acknowledgments

I take this opportunity to express my sincere thanks and deep sense of gratitude to my advisor **Prof. G. Sivakumar** for his valuable guidance and supervision in all phases of the project. I am also thankful to the **Center of Formal Design and Verification of Software** and the **Department of Computer Science, IIT Bombay** for providing the necessary facilities and support.

I believe, this work wouldn't have been possible without my parents blessings, I owe it all to them.

Aldrin John D'Souza

January, 2003.

Abstract

The process of designing a correct cryptographic protocol does not end with getting the cryptographic primitives right. The literature is full of protocols which were initially believed to be correct, and were later found to have flaws. Interestingly, most of these are structural flaws, i.e. the intruder can subvert the goals of the protocol without breaking the underlying crypto-system. Given the wide range of operations which the intruder uses to compose these attacks, it is very difficult for the designer to intuitively reason about these attacks. Formal methods of analysis should thus be applied before the protocols are put to use. Such an analysis involves developing property preserving abstractions of protocols, specification languages to express goals and assumptions, and procedures to decide whether the protocol achieves its intent. The Strand Space Model is one of the existing cryptographic protocol analysis mechanisms. Here, we describe how proofs in the strand spaces formalism can be generalized and applied to a range of protocols. We formalize our generalization in PVS and describe how a protocol description in a common specification language can be translated to theories, which can be used to prove the correctness of protocol properties.

Contents

1. Introduction

The need for formally analyzing cryptographic protocols is emphasized. A brief survey of the work done in this field is presented. An overview of the our work is given.

Security is all about providing guarantees. Authentication, for example, is the guarantee that we are indeed communicating with the party we think we are communicating with. Secrecy, on the other hand, is the guarantee that the messages we send, are intelligible only to the intended recipients. Providing these guarantees on the network is difficult, because all that we can observe with surety are local events, while these guarantees require us to reason about what might have happened across the network. This is where network security protocols come in.

A network security protocol - from the perspective of a party executing it, is a sequence of local events, enough to guarantee the occurrence of certain events across the network. These global inferences from local observations are enabled by using the properties of cryptographic transformations that are performed on the messages exchanged. For instance, if we send a encrypted message, we can infer than it makes sense only to those who posses the proper decryption key. The correctness of this inference, obviously, depends on our faith in the underlying crypto-system, but this is not the only factor.

The literature is full of examples of cryptographic protocols that were published, believed to be correct and later found to have flaws independent of the strength of the primitives used to implement them. In other words, even if the underlying crypto-system is perfect, penetrators can subvert the goals of these protocols. Any new protocol, must be thoroughly analyzed before being deployed, and this project deals with how such an analysis can be done using formal techniques of software verification.

1.1. Flaws and Attacks

One reason why formal analysis of cryptographic protocols is required is that, the attacks are usually hard to figure out intuitively. Cryptographic protocols rarely have more than four messages per party, but the options available to the penetrator are enough to take all possible executions of the protocol beyond intuitive reasoning. To realize this, consider the following symmetric key protocol due to Woo and Lam. (It is assumed that all principals share keys with a fixed key server S).

1. $\mathsf{A} \longrightarrow \mathsf{B}$: A
2. $\mathsf{B} \longrightarrow \mathsf{A}$: N_B
3. $\mathsf{A} \longrightarrow \mathsf{B}$: $\{N_B\}_{K_{AS}}$
4. $\mathsf{B} \longrightarrow \mathsf{S}$: $\{A, \{N_B\}_{K_{AS}}\}_{K_{BS}}$
5. $\mathsf{S} \longrightarrow \mathsf{B}$: $\{N_B\}_{K_{BS}}$

Figure 1.1.: Woo-Lam Authentication Protocol

The authors propose that the protocol structure is enough to provide the guarantee - *"whenever a responder finishes execution of the protocol, the initiator of the protocol is in fact the principal claimed in the initial message."* This is how they informally reason the correctness of the protocol:

A claims its identity in message 1; B provides a nonce challenge in message 2; A returns this challenge encrypted under K_{AS} in message 3; B passes on this message to S for verification, bound with A's name encrypted under K_{BS} in message 4; S decrypts $\{N_B\}_{K_{AS}}$ using the key it shares with A and re-encrypts the result under B's key and sends it to B in the last message. If B gets back $\{N_B\}_{K_{BS}}$ from S, it should be convinced that A has responded to the nonce challenge, since only A and S know the key K_{AS}.

So far so good, but as we introduce the penetrator in the system, things start happening. Consider the following run of the protocol. The penetrator P succeeds in making B believe, that it is A by starting two concurrent sessions. It uses the information that it derives from one of the sessions to fool B into believing that it is A in the other session.

1.	P $\longrightarrow$ B :	A
1'.	P $\longrightarrow$ B :	P
2.	B $\longrightarrow$ A :	N_B
2'.	B $\longrightarrow$ P :	N'_B
3.	P $\longrightarrow$ B :	$\{N_B\}_{K_{PS}}$
3'.	P $\longrightarrow$ B :	$\{N_B\}_{K_{PS}}$
4.	B $\longrightarrow$ S :	$\{A, \{N_B\}_{K_{PS}}\}_{K_{BS}}$
4'.	B $\longrightarrow$ S :	$\{P, \{N_B\}_{K_{PS}}\}_{K_{BS}}$
5.	S $\longrightarrow$ B :	$\{N''_B\}_{K_{BS}}$
5'.	S $\longrightarrow$ B :	$\{N_B\}_{K_{BS}}$

Figure 1.2.: Attack on Woo-Lam Authentication Protocol

Given this attack, we can reason what is wrong with the protocol. The last message is intended to be a reply to the query that is presented by B to S in the fourth message, but nothing in the protocol structure links the two messages. The protocol implicitly associates N_B to the claimed identity, and it is precisely this implicit association that P uses to subvert the protocol goal. The structure of the protocol is flawed!

The attack can be avoided if the protocol makes the relation between B's query and S's response explicit in the last message by replacing it with $\{A, N_B\}_{K_{BS}}$. Cryptographic operations aren't wholly cheap, and the designer is tempted to do away with parts of the messages which he thinks aren't necessary for the correctness of the protocol. While such an optimization is definitely welcome, the effects of removing information from messages must be thoroughly examined.

Apart from the fact that reasoning intuitively about these protocols is difficult and error prone, the use of formal methods is essential for another reason. These protocols are usually components of some larger systems, and the critical nature of the applications that these systems support makes it essential to produce proofs of correctness before these protocols are deployed. Systems may be unacceptable to the users in absence of such proofs of correctness. This is indeed one of the reasons, why protocols like SET could never find widespread usage.

1.2. Formal Analysis - A Brief Survey

Formal analysis of cryptographic protocols has been a field of active research in recent years, several formalisms and mechanisms have been developed and have been used to show that protocols are correct. Most of the work done falls into one of the following categories -

- Logics of Knowledge and Belief
- Discrete State based Algebraic Systems
- Complexity Theoretic Analysis

The logical paradigm, makes use of logics of belief or knowledge to prove correctness of protocols. The framework comprises of a set of constructs to express the beliefs of protocol participants and a set of inference rules to specify how these beliefs change as the protocol proceeds. BAN logic[15], introduced by Burrows et. al. is the most successful attempt that falls under this style of analysis. The approach enjoys the advantages of being intuitive and easy to use, but the system is abstracted at a very high level, and several issues like concurrency are ignored.

Complexity theoretic analysis on the other hand, reduce the problem of attacking the protocol to some computationally hard problem, thus proving that it is impossible for a penetrator to attack the protocol. This approach is flexible enough to model the protocol at any level of detail, but can not be generalized or automated.

Most of the recent work done is based on algebraic systems. Here, cryptographic primitives are idealized as types with certain properties, and the protocol is considered to be a system that uses the properties of these types to provide the required guarantees. Correctness of protocols is proved either by searching the execution space for attacks or by proving mathematically that it is impossible for the penetrator to subvert the goals of the protocol. These correspond to the two approaches of analysis namely, *attack construction* and *proof generation.*

1.2.1. Attack Constructors

Millen's Interrogator, Longley and Rigby's search tool, are examples of specialized exhaustive model checkers which attempt to construct attacks. General purpose model checkers have also been used - Lowe, for instance used FDR to uncover a flaw in the Needham-Schroeder protocol, Mitchell used Murϕ to analyze SSL. Each of these tools, however search the execution space exhaustively, and require the number of protocol participants to be specified before the search begins. Recently, Song developed Athena, which models the protocol state symbolically and can search for attacks with arbitrary number of participants. Another noteworthy tool is Meadows NRL Protocol Analyzer. Like Song's tool, the NRL analyzer constructs attacks backward (starting from an undesirable state and searching for states that reachable from the initial state) and thus does not require initial number of participants to be specified.

1.2.2. Proof Generators

Paulson used Isabelle to generate inductive proofs of correctness of cryptographic protocols[17]. Millen used PVS as a proof assistant to obtain proofs based on Paulson's ideas[18]. Schneider uses rank functions to reason about the possibility of events occurring in the protocol execution [19]. Thayer et.al. recently came up with a graph theoretic model of causality in a protocol, called strand spaces. Strand spaces form the theoretical base of our tool, and are described in detail in the following chapters. Abadi and Gordon extend Miller's pi-calculus, to what they call, spi-calculus to prove secrecy properties of cryptographic protocols[16].

Having described the problem we are trying to deal with, and the ways it has been dealt with in the past, we present the scope of our work in the next section. Our survey is not exhaustive. The reader is referred to [8, 10, 11] for a more detailed description of the work done in the field.

1.3. Scope of Our Work

The goal of our project is to come up with an tool that a cryptographic protocol designer can use to analyze protocols that he designs. The following are the features of our tool

- Graphical specification of cryptographic protocols.
- Read and generate CAPSL specifications.
- Generation of PVS theories for generating proofs.

Strand spaces form the theoretical base of our tool. Most of the work on proof generation for cryptographic protocol properties is based on Paulson's inductive analysis. To the best of our knowledge proofs based on strand spaces haven't been automated. We have developed a framework where PVS can be used to assist in proving properties of protocols based on the strand spaces formalism.

1.4. Outline

We formally define the problem of cryptographic protocol analysis in the next chapter, formalizing the notions of ideal cryptography, the penetrator and the correctness of a protocol. The specification language for cryptographic protocols CAPSL , is described in Chapter 3. The theoretical base of our tool, viz. strand spaces in explained in considerable detail in Chapter 4. Strategies for proving protocol properties are described in Chapter 5. PVS theories for automatic proofs are described in Chapter 6.3.

2. The Problem

The scope of our analysis is defined. Notions of ideal cryptography, correctness, penetrator capabilities are formalized.

Our problem can be informally put as - given a protocol and the guarantees that it claims to provide, establish whether or not the protocol achieves its intent in the presence of an active penetrator, under the assumption that the underlying cryptographic primitives are ideal. This problem statement is incomplete unless we formalize the notions of ideal cryptography, the capabilities of the intruder and the guarantees that protocols attempt to provide. We begin with our model of cryptography.

2.1. Ideal Cryptography

We treat cryptographic primitives as *black boxes* which work exactly as the protocol designer expects them to. They are formalized as operations defined on a set of messages. Consider $\mathcal{A}$, the set of all possible messages that can be exchanged in a protocol run. We will call elements of $\mathcal{A}$ - terms. Terms can belong to the following subsets of $\mathcal{A}$ -

- $\mathcal{T} \subseteq \mathcal{A}$, the set of plain-text components like text and nonces.
- $\mathcal{N} \subseteq \mathcal{T}$, the set of principal names.

- $\mathcal{K} \subseteq \mathcal{A}$, the set of cryptographic keys, disjoint from $\mathcal{T}$

Next, we define the cryptographic primitives as operations on the set $\mathcal{A}$. We will restrict ourselves to the following primitives -

- $\mathsf{inv} : \mathcal{K} \rightarrow \mathcal{K}$
- $\mathsf{encr} : \mathcal{K} \times \mathcal{A} \rightarrow \mathcal{A}$
- $\mathsf{join} : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$
- $\mathsf{K} : \mathcal{N} \rightarrow \mathcal{K}$
- $\mathsf{K} : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{K}$

inv maps keys to their respective inverses. We will denote $\mathsf{inv}(K)$ by K^{-1}. As a shorthand, we extend inv to set of keys and denote the the set of inverses of keys in a set S by S^{-1}. encr denotes encryption of a term using a key from $\mathcal{K}$. We will denote $\mathsf{encr}(K, m)$ by $\{m\}_K$. join denotes concatenation of two terms. We will denote $\mathsf{join}(a, b)$ by $a\,b$. K is overloaded, when given a single principal name it maps it to the public key corresponding to the principal, we will denote $\mathsf{K}(A)$ by K_A, and when given two arguments, it maps to the symmetric key that the two principal share. We will denote $\mathsf{K}(A, B)$ by K_{AB}.

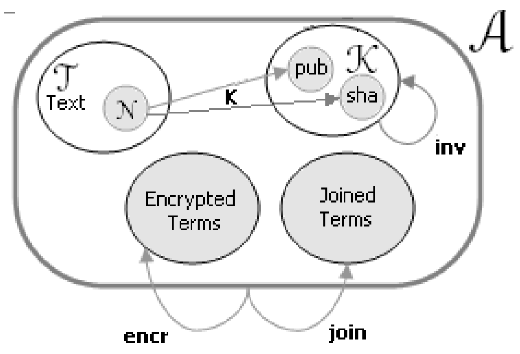


Figure 2.1.: The Term Algebra

Figure 2.1 gives a pictorial description of the term algebra and also the intuitive meaning of the freeness assumption that we describe next.

2.1.1. Freeness Assumptions

By abstracting cryptography as an algebra, we restricted ourselves to deal with messages as results of operations performed over atomic components and not as bit-streams. The freeness assumption[1] restricts us further. The assumption ensures that there is only one representation of a given message. In particular, the following can be inferred from the freeness assumption -

- A cipher-text can be regarded as a cipher-text in only one way. Formally, $\{m\}_K = \{m'\}_{K'} \Rightarrow m = m' \wedge K = K'$
- A concatenation of two terms can be interpreted in only one way. Formally, $m_0\, m_1 = m'_0\, m'_1 \Rightarrow m_0 = m'_0 \wedge m_1 = m'_1$
- A concatenation of two terms can not be treated as a valid cipher-text. Formally $m_0\, m_1 \neq \{m'_0\}_{K'}$
- A concatenation of two terms can not be taken as a key or a plain-text. Formally, $m_0\, m_1 \notin K \cup T$
- A cipher-text can not be treated as a key or a plain-text. Formally, $\{m'_0\}_{K'} \notin K \cup T$

The most important consequence of these assumptions is that we are ignoring what are called *type-flaw attacks*. These attacks corresponds to situations where the penetrator succeeds in making a participant read a term as some other term. Avoiding type-flaws is mostly an implemention issue, and we, being more concerned with the structure of protocols, will ignore them in our analysis.

[1] $\mathcal{A}$ is an algebra freely generated from $\mathcal{T}$ and $\mathcal{K}$ by the two operators encr and join.

2.2. The Penetrator

The model of the attacker that we consider is was introduced by Dolev and Yao[6] and is thus called the Dolev-Yao Intruder. It is capable of the following -

- Read all network traffic.
- Alter or destroy messages.
- Create new messages.
- Do anything that a legitimate principal is capable of doing.

Apart from these capabilities, another factor which defines the strength of the penetrator is its knowledge. The penetrator knows some terms to start with, and it can learn new terms from the messages that are sent over the network. The ability to learn is however limited by the rules of the cryptographic primitives. For instance, if some protocol participant sends a term $\{m\}_K$ then the intruder can learn m out of it only if it knows K^{-1}. The penetrator can also originate a message containing terms that it knows or can create. To account for dishonest participants, the penetrator is assumed to posses all capabilities of a normal participant. For instance, it can posses a valid name, an associated asymmetric key-pair, and can even share keys with a key server, if any.

2.3. Protocol Guarantees

Security protocols attempt to provide certain guarantees to the participants, in this section we formally define these guarantees and describe what it means for a protocol to be correct. In particular, we describe the meanings of authentication and secrecy.

2.3.1. Authentication

In [24], Lowe comes up with a hierarchy of authentication specifications. Our interpretation of authentication guarantee corresponds to that of Non-Injective Agreement in Lowe's paper. We say a protocol guarantees authentication of B to A iff whenever A completes a run of the protocol, apparently with B, then B has previously been running the protocol, apparently with A, and the two agents agree on some data values, as required by the protocol. Note that A and B are parameters to the specification rather than specific principals.

2.3.2. Secrecy

Secrecy is the requirement that some term doesn't fall into the hands of the penetrator. The formal definition of depends on how we model the penetrator. It is assumed that the penetrator emits values that it knows, so verifying whether the penetrator can emit a given value in the protocol run is equivalent to verifying if the protocol keeps that value secret.

3. Specification

The syntax and semantics of the specification language CAPSL *are illustrated with an example. The features of the intermediate language are described.*

In most cases, the system description that the analysis mechanism requires does not intuitively correspond to the analyst's perception. It is the task of the specification language to provide the analyst with abstractions closer to what he thinks the system is, and then translate these abstractions to the formalisms that the analysis mechanism works on. In this chapter, we describe CAPSL, the language which does this job for cryptographic protocols.

As we mentioned, many mechanisms for analyzing cryptographic protocols exist. In spite of being used successfully for analyzing protocols, it is difficult for analysts other than the original developers to translate conventional protocol descriptions to the formalisms that these mechanisms require. Also, an analyst wishing to use more than one mechanism has to translate between the specification languages of the two. To summarize, the problems that a cryptographic protocol analyst faces are -

- Translating a published description of the protocol to the formalism required by the analysis mechanism is difficult.

- Using more than one analysis mechanisms requires translation between the formalisms underlying the mechanisms.

It is precisely to tackle these problems CAPSL[9] was developed.

3.1. CAPSL

CAPSL stands for Common Authentication Protocol Specification Language. It was introduced by Jonathan Millen, as a common specification language that all analysis mechanisms can use. The idea is to include all the information needed for analysis in the specification, and allow a "connector" to selectively choose and translate the part that its underlying mechanism uses for the analysis. This is made possible by dividing the specification into three modules:

- *Type Specification Module* - Declare and describe cryptographic types, operators and other functions axiomatically.
- *Protocol Specification Module* - Declare the assumptions, messages and goals of the protocol.
- *Environment Specification Module* - Declare run specific information about the protocol.

The types and operations that most cryptographic protocols use are provided in a prelude, which is expected to be included in all CAPSL specifications. So unless a protocol uses special functions or types, the typespec module can be omitted. Environment specifications are used by tools which search the execution space of the protocol for attacks. This module is optional and can be omitted if such a search is not to be attempted. The most important part of a specification is the protocol module, describing the messages, the assumptions and the goals of the protocol.

3.1.1. Type Specifications

Messages in cryptographic protocols are constructed by cryptographically transforming the message components. Analysis mechanisms use the properties of these transforming operations to provide guarantees about the possible execution of the protocol. Properties of types and these transformation operations are declared in the `typespec` module. A `typespec` module consists of some declarations, followed optionally by some axioms. Typespecs usually introduce a new type and some functions defined on it, but in some cases they merely extend an existing typespec by defining new functions on existing types.

Listing 3.1: CASPL Typespecs

```
TYPESPEC PKEY;
IMPORTS FIELD;
TYPES Pkey;
VARIABLES PK1, PKI1: Pkey;
FUNCTIONS
 keypair(Pkey, Pkey): Boolean, COMM;
END;
```

```
TYPESPEC SPKE;
IMPORTS PKEY;
FUNCTIONS
    ped(Pkey, Atom): Atom;
    ped(Pkey, Field): Field;
AXIOMS
 IF keypair(PK1,PKI1) THEN
   ped(PKI1, ped(PK1,X1))=X1
 ENDIF;
END;
```

Listing 1.1. shows two `typespecs`. The first declares a new type `Pkey`, and a function `keypair` on it. The function returns a boolean value. The `COMM` in the declaration specifies that the function is commutative. The second typespec imports the types defined in `PKEY` using the `IMPORTS` declaration and defines more functions on the type. The declaration `ped(Pkey, Atom):Atom` declares a function `ped` with parameters of type `Pkey` and `Atom` respectively, and a return type `Atom`. Properties of the functions are specified in the `AXIOMS` definition. For instance, the property shown here says that encrypting a term encrypted in a key, with its pair gives us the term again.

3.1.2. Protocol Specifications

The protocol specification is the most important part of a CAPSL specification. Instead of listing the syntactic details, we illustrate the main features of CAPSL specification by the means of the adjacent example. For details the reader can refer the CAPSL project report[14].

```
PROTOCOL WooLam;
VARIABLES
  S : Server;
  A, B : Client;
  Nb : Nonce;
  F : Field;
  Kbs: Skey;
DENOTES
  Kbs = ssk(S,B): S;
  Kbs = csk(B): B;
ASSUMPTIONS
  HOLDS A: B;
  HOLDS B: S;
MESSAGES
  1. A -> B: A;
  2. B -> A: Nb;
  3. A -> B: {Nb}csk(A)%F;
  4. B -> S: B,{A, F%{Nb}ssk(S,A)}Kbs;
  5. S -> B: {Nb}Kbs;
GOALS
  PRECEDES A: B | Nb;
END;
```

Woo-Lam Protocol
CAPSL Specification

. Variables

Protocol specifications begin with a `PROTOCOL` declaration, which is followed by the protocol identifier. The `VARIABLES` declaration is used to declare message fields and their corresponding types. `Server` and `Client` are types defined in the prelude. A `DENOTES` declaration allows a variable to be defined as the value of an expression. This is helpful in reducing the clutter in the messages section, when the protocol uses tickets or messages with complex structure. The declaration `DENOTES Kbs = ssk(S,B): S;` specifies that principal S, obtains the value of the variable `Kbs`, by computing the result of the function `ssk` with its values of `S` and `B`. The `DENOTES` specification is optional, and can be omitted for simpler protocols.

Consider the third message, where `A` sends a message encrypted with the key it shares with `S` to `B`. Since `B` doesn't know the encryption key, there is no way it can interpret the contents of the message. It is however required to forward this term to the server, and such forwarding of uninterpreted message components is done using the `%` notation.

Intuitively, a `%` specifies that the senders and the recipient's view of the same message are different due to the lack of some information which one party does not posses. CAPSL borrows the `%` notation from Lowe's Casper.

. Assumptions

Protocols assumptions about the knowledge and the capabilities of the of the participating principals are made explicit in the `ASSUMPTIONS` clause. The declaration `HOLDS A: B` specifies, that on starting its run of the protocol, principal `A` knows the value of the protocol variable `B`. This in effect, makes explicit the assumption that `A` knows whom it wants to talk to.

. Messages

The messages exchanged in the protocol are specified in the `MESSAGES` section. A message declaration starts with a message id, is followed by a `<sender> -> receiver:` construct, and ends with an expression (which is the message to be sent). Some notational conventions allow the expression to be written in more intuitive form. For example, message components can be concatenated together, by enclosing them in `{}`, similarly encryption of a term `a` with a key `K` can be expressed as `{a}K`. For instance, the `{Nb}csk(A)` in the third message is a shortcut for `ped(csk(A), Nb)` where `ped` is the encryption function defined in the prelude. `csk`(client shared key) is another function from the prelude, mapping names to keys that principals share with a key server.

. Goals

Protocol specifications usually end with a specification of the `GOALS` of the protocol. Goals are specified using `PRECEDES`, `AGREE` and `SECRET` keywords. Secrecy assertions take the form `SECRET V`, specifying that the value of the protocol variable V should not be disclosed to the penetrator. A precedence assertion of the form `PRECEDES A,B | V`, specifies that when principal `A` finishes its run of the protocol, there should exists some run of the protocol for `B`'s role, which agrees on the values of `A`, `B` and `V`. This notion corresponds to Lowe's non-injective agreement. Agreement assertions are precedence claims with no claims on the existence of other party executions. `AGREE A,B: V|W` states that if `A` and `B` agree on `V` then they should agree on `W` too.

3.1.3. Environment Specification

Environment specifications provide run-time information that is needed by attack constructors to set up the search for unwanted executions. Typically the information provided here includes the a list of agents that are expected to be part of the execution, and the a list of terms that are assumed to be known to the penetrator, Agent settings are given using `AGENT` declarations, which are actually assignment of values to the protocol variables. Penetrator knowledge is specified using `EXPOSED` declaration, which encloses a list of terms that form the initial knowledge of the penetrator.

Our description of CAPSL here is incomplete, and was intended to give a feel of the specifications of protocols. There's a lot more to the language syntactically as well as semantically, and the reader can see the project web page[12], for a detailed description of the language.

3.2. Intermediate Language

CAPSL provides abstractions that make it easy to translate published descriptions of protocols to the specification required by analysis mechanisms. These abstractions, however need to be translated to the formalism that the underlying analysis requires. Like connectors translating CAPSL specifications for analyzers based on multi-set rewriting systems[13] we use an intermediate language.

Our intermediate language is described as an XML vocabulary, whose DTD is given Appendix-A. The language has two purposes, firstly it acts as a connector to CAPSL, filtering those parts of the specification which our analysis mechanism can make use of, and secondly it serves as the language that our GUI writes specification to and reads specifications from.

The message specification in CAPSL, is none better than the usual "Alice-Bob" style of specifying protocols. Given the fact that the intruder can originate or flush messages, the `{A -> B: A}` notation doesn't make a lot of sense, since we are niether sure of the sender nor of the receiver. A better approach would be to specify the interface of each participant i.e. what messages it sends and receives, without associating these messages with any other principal as is done by the intermediate language.

As can be done using the `DENOTES` declaration in CAPSL , we declare all terms used in messages before hand. Protocol roles are parameterized, as required for representing them as parameterized strands. In short, the following are the features of the intermediate language.

- Separate definitions for terms, roles and goals.
- Terms declared once and are reused throughout.
- Interfaces of the participants are specified separately, instead of the "Alice-Bob" style.

4. Modeling

The strand spaces formalism is described. The behavior of the protocol participants, the penetrator, and the correctness criteria are expressed in the formalism.

To reason about the correctness of cryptographic protocols, we need a mathematical model to represent their execution. Several formalisms have been developed for this purpose, some of which are used specifically for modeling cryptographic protocols, while others are generic enough to be used for the analysis of other systems as well. Our work is based on a formalism, which models protocol execution as strands. A run of the protocol is modeled as a collection of strands, called a strand space. Protocol properties are expressed as predicates on this structure and they are verified by reasoning inductively.

4.1. Strand Spaces

Events in a cryptographic protocols are causally dependent on other events. Strand spaces[1] are a graph theoretic model of this causal dependence. The basic building block of this model is a strand. Informally, a strand is a sequence of events, that represents either a protocol execution by a honest principal or a sequence of actions taken by the penetrator.

Recall that, $\mathcal{A}$ is the set of all possible messages that can be exchanged in a protocol run, if we denote the sending of a term $t \in \mathcal{A}$ by $+t$ and its receipt by $-t$, the execution of a protocol by a given participant can be modeled as a string in $(\pm\mathcal{A})^*$. A term with its direction prefixed to it is called a signed term. Intuitively, a signed term corresponds to the event of sending or receiving the term by a protocol participant or the intruder. A strand is string of signed terms, and a strand space is a set Σ together with a trace mapping $tr : \Sigma \rightarrow (\pm\mathcal{A})^*$. Figure 4.1 shows a strand space with four strands.

4.1.1. Nodes

A node is a pair (s, i) where $s \in \Sigma$ and $0 \leq i \leq |tr(s)|$. Given a node $n = (s, i)$, $strand(n) = s$ and $index(n) = i$. Also, $term(n)$ is the i^{th} signed term in the trace of s, and $unsterm(n)$ is the same term without the sign. For two nodes n_1 and n_2 we say $n_1 \Rightarrow n_2$ iff $n_1 = (s, i)$ and $n_2 = (s, i+1)$. $\Rightarrow^+$ is the transitive closure of $\Rightarrow$. Intuitively, the $\Rightarrow^+$ relation captures the fact that n_1 precedes n_2 on the same strand. For two nodes n_1 and n_2 we say $n_1 \rightarrow n_2$ if $term(n_1) = +t$ and $term(n_2) = -t$ form some term $t \in \mathcal{A}$. Intuitively, $n_1 \rightarrow n_2$ means that n_2 receives a term that is sent by n_1.

We define $\preceq$ as the reflexive and transitive closure of $(\Rightarrow \cup \rightarrow)$, modeling the causal dependency of events in the protocol execution. For example, in Figure 4.1 $(s_4, 0) \preceq (s_1, 1)$ captures the fact that for the event corresponding to $(s_1, 1)$ to happen, the event corresponding to $(s_4, 0)$ must happen first. To be a correct model of a protocol execution, it is necessary that a node be included only if all nodes that it causally depends on are already included. This requirement is enforced in what we call a bundle.

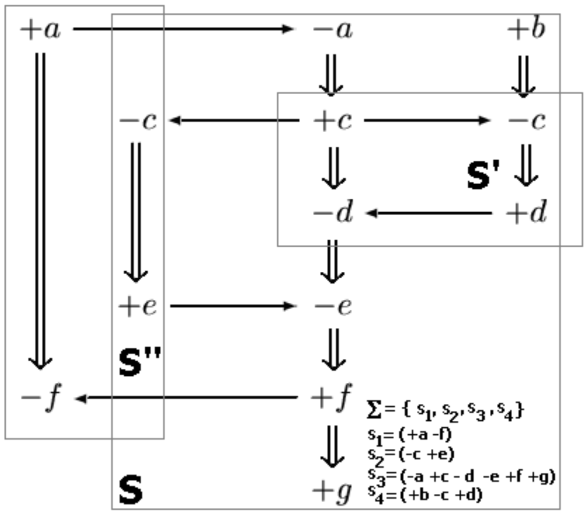


Figure 4.1.: A Strand Space

4.1.2. Bundles

Let N be the set of nodes in a strand space Σ, and let $\rightarrow_{\mathcal{C}} \subseteq \rightarrow$ and $\Rightarrow_{\mathcal{C}} \subseteq \Rightarrow$. A bundle is a subgraph $\mathcal{C} = \langle \mathsf{N}_{\mathcal{C}}, (\Rightarrow_{\mathcal{C}} \cup \rightarrow_{\mathcal{C}}) \rangle$ of the graph $\langle \mathsf{N}\ (\Rightarrow \cup \rightarrow) \rangle$ with the following properties

- $\mathcal{C}$ is finite.
- If $n_2 \in N_{\mathcal{C}}$ and $term(n_2)$ is negative, then there is a unique n_1 such that $n_1 \rightarrow_{\mathcal{C}} n_2$.
- If $n_2 \in N_{\mathcal{C}}$ and $n_1 \Rightarrow n_2$ then $n_1 \Rightarrow_{\mathcal{C}} n_2$
- $\mathcal{C}$ is acyclic.

Intuitively, the definition insures that a node is included only if all nodes that causally precede it are already included. The graph in Figure 4.1 constitutes a bundle, so do the nodes and edges enclosed in the rectangle labeled S. The graphs contained in the rectangles labeled S' and S'' are not a bundles because they are not causally closed. S' is not closed under $\Rightarrow$. (c is included and $-a \Rightarrow +c$ but $-a$ is not included) Similarly S'' is not closed under $\rightarrow$. ($-f$ is included, but no node that sends f is).

Strands in a bundle must be up wards closed, but it is not necessary for them to be completely in the bundle. We define the $\mathcal{C}$-height of a strand in a bundle C as the index of the last node that is included in the bundle.

If $\mathcal{C}$ is a bundle, the reflexive and transitive closure of $(\Rightarrow_{\mathcal{C}} \cup \rightarrow_{\mathcal{C}})$ is a partial order denoted by $\preceq_{\mathcal{C}}$. Moreover every non empty subset of the nodes in $\mathcal{C}$ has $\preceq_{\mathcal{C}}$ minimal members. This well-foundedness of bundles allows us to prove properties inductively.

4.1.3. Subterms, Interms and Ideals

We have defined cryptographic primitives as operation on $\mathcal{A}$, in this section we add more structure to $\mathcal{A}$ with the help of a subterm relation. We also introduce the notion of an ideal which will be used to obtain bounds on the capabilities of the penetrator.

We say a term t_1 is a subterm of another term t_2 ($t_1 \sqsubset t_2$) if starting with t_1 we can construct t_2 by repeatedly concatenating with arbitrary terms or encrypting with arbitrary keys. Formally, the subterm relation relation($\sqsubset$) is defined inductively as the smallest relation such that:

- $a \sqsubset a$
- $a \sqsubset \{g\}_K$ if $a \sqsubset g$
- $a \sqsubset g\,h$ if $a \sqsubset g \vee a \sqsubset h$

Note that $K \sqsubset \{g\}_K$ only if $K \sqsubset g$.

Subterms of a term are terms that can be extracted out of it, if we knew the inverses of all the keys that are used in those terms. If none of the keys are known, the subterm relation reduces to the interm relation($\Subset$). In other words, a term $a_1 \Subset a_2$ if a_1 can be extracted from a_2 without having to decrypt anything. Formally,

- $a \Subset t$ for $t \in \mathsf{T}$ iff $a = t$; or

- $a \Subset k$ for $k \in \mathsf{K}$ iff $a = k$; or
- $a \Subset \{g\}_k$ iff $a = \{g\}_k$; or
- $a \Subset gh$ iff $a \Subset g \vee a \Subset h \vee a = gh$.

If $a_1 \Subset a_2$ then, a_1 is called a *component* of a_2.

The subterm relation with a specific set of keys in our hands reduces to what we call an ideal. If k$\subseteq \mathcal{K}$, a k-ideal of $\mathcal{A}$ is a subset I of A such that for all $h \in I$, $g \in A$ and $K \in k$

- $h\,g, g\,h \in I$
- $\{h\}_K \in I$.

The smallest k-ideal containing h is denoted as $\mathsf{I}_k[h]$, and the smallest k-ideal containing $S \subseteq A$ is denoted as $\mathsf{I}_k[S]$. Intuitively, $\mathsf{I}_k[S]$ is the smallest set containing S closed under concatenation with arbitrary terms, and encryption with keys in k.

Note that $g \sqsubset h$ iff $h \in \mathsf{I}_{\mathcal{K}}[h]$.

4.1.4. Origination

We give some definitions first -

- A term t occurs at a node n, if $t \sqsubset term(n)$.
- A node n is an entrypoint for a set of terms I if, $term(n) = +t$ for some $t \in I$ and for all $n' \Rightarrow^+ n$ $term(n') \notin I$.
- An unsigned term originates on n iff n is an entrypoint for the set $I = \{t' : t \sqsubset t'\}$.
- An unsigned term is uniquely originating if it originates at only one node.

Intuitively, a term occurs at a node if the node either sends or receives a term which contains it. A node is an entrypoint for a set of terms iff it sends a term contained in the set, and no node before it on the strand, either sends or receives any term in the set. An originating node for a term is the entrypoint of the set of terms containing the given term. Intuitively, originating nodes are those sending nodes on a strand that send a term without receiving it from somewhere for the first time. A term is uniquely originating iff it is introduced into the strand space at a unique node. Uniquely originating terms serve as nonces and session keys in protocols.

4.1.5. The Penetrator

Two things decide the power of the penetrator - first the keys he possesses and second, his ability to manipulate terms. Kp denotes the keys that the penetrator knows initially, and these include public keys of all principals, the private keys of the penetrator, and all keys that the penetrator shares with other principals. It also contains keys that it might have obtained by some cryptanalysis.

The second constituent of the penetrator's power are the atomic operations that enable him to flush messages, generate well known messages, concatenate messages together, and apply cryptographic transformations using the keys that he knows. Attacks result when the penetrator succeeds in composing these atomic actions together to achieve something that is undesirable. The following set of penetrator traces give the penetrator just the same powers as mentioned in the previous chapter.

4.2. Modeling Protocols

Unlike the penetrator traces, which stay the same, honest principals send and receive terms, the form of which, is governed by the protocol. The messages exchanged are

M	Text Message	$\langle +t \rangle$ where $t \in T$
F	Flushing	$\langle -g \rangle$
T	Tee	$\langle -g, +g, +g \rangle$
C	Concatenation	$\langle -g, -h, +gh \rangle$
S	Separation	$\langle -gh, +g, +h \rangle$
K	Key	$\langle +K \rangle$ where $K \in \mathsf{Kp}$
E	Encryption	$\langle -K, -h, +\{h\}_K \rangle$
C	Decryption	$\langle -K^{-1}, -\{h\}_K, +h \rangle$

Figure 4.2.: Penetrator Traces

however parameterized, and the actual term sent depends on the bindings that are provided to the protocol parameters when a participant begins execution. The strand space model, allows us to represent this parameterized execution as a parameterized strand.

4.2.1. Parameterized Strands

A parameterized strand is a set of strands, and members of this set can be obtained by assigning values to the parameters. We illustrate the idea using the Otway-Rees protocol, shown in Figure 6.4. The protocol has three participants - the initiator, the responder and a fixed key server

1. A $\longrightarrow$ B : $M\,A\,B\,\{N_a, M, A, B\}_{K_{AS}}$
2. B $\longrightarrow$ S : $M\,A\,B\,\{N_a, M, A, B\}_{K_{AS}}\,\{N_b, M, A, B\}_{K_{BS}}$
3. S $\longrightarrow$ B : $M\,\{N_a K\}_{K_{AS}}\,\{N_b K\}_{K_{BS}}$
4. B $\longrightarrow$ A : $M\,\{N_a K\}_{K_{AS}}$

Figure 4.3.: Otway-Rees Protocol

The initiator's interface is parameterized by the following: its own identity, A; the identity of the respondent, B; the text component, M; the identity of the server, S; the value of the nonce, N_a and the value of the session key, K. Specific instances of the initiator's role send messages of the form shown in the figure, with the parameters replaced by the values bound to them for that run. The values may get bound to the parameters before the role starts execution, or they might be learned from the messages received. For instance B is bound to a value before the initiator starts its run, while K gets its value only after the last message is received. However, irrespective of the actual values, the interface of the initiator is a trace of the form - $\langle\ +M\ A\ B\ \{N_a, M, A, B\}_{K_{AS}}\ -M\ \{N_a K\}_{K_{AS}}\ \rangle$.

Thus, we can use $\mathsf{Init}[A, B, S, N_a, M, K]$ to represent all possible initiator strands. A specific run of the initiator role, would have some assignments to the parameters, but the strcuture of the trace would remain the same. This notation also allows us express traces with some common feature, for instance $\mathsf{Init}[*, *, *, N_{a_0}, **]$ is the set of all initiator strands, which use a specific vallue of the nonce N_a.

Extending the same to other participants, we can express the entire strand space of the Otway-Rees protocol with the following parameterized strands.

$\mathsf{Init}[A, B, S, N_a, M, K] =$

$\langle\ +M\ A\ B\ \{N_a\ M\ A\ B\}_{K_{AS}},\ -M\ \{N_a\ K\}_{K_{AS}}\ \rangle$

$\mathsf{Resp}[A, B, S, N_b, M, K, H, H'] =$

$\langle\ -M\ A\ B\ H\ ,\ +M\ A\ B\ H\ \{N_b\ M\ A\ B\}_{K_{BS}}\ ,\ -M\ H'\ \{N_b\ K\}_{K_{BS}}\ ,\ +M\ H'\rangle$

$\mathsf{Serv}[A, B, S, N_a, N_b, M, K] =$

$\langle\ -M\ A\ B\ \{N_a\ M\ A\ B\}_{K_{AS}}\ \{N_b, M, A, B\}_{K_{BS}}\ ,\ +M\ \{N_a, K\}_{K_{AS}}\{N_b, K\}_{K_{BS}}\rangle$

4.2.2. Freshness

While parameterized strands represent the interface behavior succinctly, they still don't express the freshness guarantees of nonces and session keys. This is done using the notion of origination discussed in Section 4.1.4. For instance, specifying that N_b in the Woo-Lam Protocol originates uniquely at the second node of the responder strand, establishes it as a nonce with the property that it occurs at no sending node(except the second node of the responder strand) in the strand space unless there is a receiving node that precedes the sending node. Moreover, the requirement that nonces be generated fresh every time can be expressed as $|Resp[*, *, *, N_b, *]| \leq 1$.

Thus, the notion of origination models the requirements of session keys and nonces appropriately, and this beautiful modeling of freshness is one of the strengths of the strand spaces model.

4.3. Modeling Correctness

4.3.1. Secrecy

Typically, protocols claim to guarantee secrecy of terms like session keys and nonces. If we collect all terms to be kept secret in a set S, proving that the protocol guarantees secrecy is equivalent to proving that the predicate $\mathcal{S}(\phi, S)$ holds, where

$$\mathcal{S}(\phi, S) = \forall\, \mathcal{C}\ \forall\, s\ \forall\, n\ \phi(s) \wedge n \in \mathcal{C} \Rightarrow term(n) \notin \mathsf{I}_k[S]$$

Here, $k = (\mathcal{K} \setminus S)^{-1}$. $\mathcal{C}$ is a variable ranging over bundles, s ranges over strands, and n over nodes. $\phi(s)$ contains assumptions about the keys being uncompromised and assumptions about the origination of the terms being exchanged.

We begin our explanation of how this predicate captures the requirements of Section 2.3.2, with the claim that no key in S is in Kp. It makes little sense to talk about the secrecy of terms(keys) that are already known to the penetrator. Actually, ϕ will contain assumptions about keys in S not being in Kp. In other words, we begin with the penetrator not knowing any key in S.

For the penetrator to learn any value in S, some node must send a term that is encrypted by keys whose inverse the penetrator might know. All we are sure of, is that the penetrator does not know any key in S ($\phi(s)$ ensures that) but we do not know what other keys are known to it. Any node sending terms encrypted with the inverse of keys ($\mathcal{K} \setminus S$) can risk the secrecy of terms in S. Recall that $\mathsf{I}_k[S]$ is the set of all terms that can be formed by concatenating terms in S with arbitrary terms and encrypting with keys in k. Our choice of k is precisely those keys whose inverse the penetrator *might* posses. So proving that no node ever emits terms contained in $\mathsf{I}_k[S]$ is equivalent to proving that the protocol never sends its secrets unprotected.

4.3.2. Authentication

Like we did for the secrecy guarantee, we will translate the requirements of authentication mentioned in Section 2.3.1 to our model. Authentication guarantees can be expressed using the following predicate

$$A(i, j, \phi, \psi) = \forall \mathcal{C}\ \forall s\ \exists s'\ \mathcal{C}\text{-height}(s) = i \land \phi(s) \Rightarrow \mathcal{C}\text{-height}(s') = j \land \psi(s, s')$$

Intuitively, the implication claims about the presence of certain strands if certain other strands are already in the bundle. Translated to normal language, this is the guarantee of certain events happening given that some events happened, which is precisely what authentication is all about.

$\phi(s)$ specifies properties that the strand s should posses. As in the secrecy requirement, it also contains assumptions about terms being uniquely originating, and keys being uncompromised. The conclusion $\psi(s, s')$ specifies properties that the strand s' should posses(s' is always assumed to be a regular strand). It will also include the agreement requirements, i.e. what parameters of s and s' should have the same values.

This concludes our discussion of how cryptographic protocols, penetrator and the correctness requirements are modeled in the strand spaces formalism. While we have stated the correctness requirements in the last section, we haven't described of the way the proofs of these implications must be attempted. This forms the subject of the next chapter.

5. Proving Protocol Correctness

A general strategy for proving protocol properties is described. The use of the partial orders $\preceq$ and $\sqsubset$ is illustrate. Secrecy and authentication guarantees of Otway-Rees Protocol are proved.

A bundle models one possible execution of a protocol. When we prove a property for all bundles, we establish it for all executions of the protocol. We identified properties that we are interested in in the last chapter. In this chapter we discuss how proofs of properties can be constructed in the strand spaces formalism.

Proofs in the strand spaces framework, exploit two partial orderings, namely the subterm relation $\sqsubset$ between terms and the $\preceq$ relation between nodes. Inductive arguments over the $\preceq$ relation are based on the minimal nodes, while those over $\sqsubset$ relation are based on ideals. Before describing strategies to prove protocol properties we provide insights on how proofs can use these partial orders.

5.1. Using $\preceq$

Suppose ψ is a predicate on nodes, and suppose we want to prove the following statement over all bundles $\mathcal{C}$

$$\forall \mathcal{C}\, \forall n : n \in \mathcal{C} \Rightarrow \neg\, \psi$$

In words, this is the same as saying that nodes with certain features(those required by ψ) do not belong to any bundle, which in turn is equivalent to saying that that certain events never occur in protocol runs. The proof will use the well-foundedness of bundles. Recall that, every non empty subset of the nodes in a bundle must have $\preceq$-minimal nodes. If we assume the set of nodes satisfying ψ as non-empty, we must have $\preceq$-minimal nodes in the set. If we prove that no node can satisfy ψ and the requirements of a minimal node simultaneously, we prove that S is empty. In step iv. of the general

i. Let $S = \{n : node \mid \psi(n)\ holds\}$.
ii. The goal is to prove that S is empty, assume that it isn't.
iii. If S is non-empty, is must have a $\preceq$-minimal, say min.
iv. Consider all possibilities of min.
v. If no node can be min, we have a contradiction, and S is empty.

Figure 5.1.: A Generic Proof Strategy

strategy, we are required to consider both regular and penetrator nodes. Considering all these nodes might seem demanding at first, but it actually isn't. Given that we have a fixed penetrator model and that the regular strands hardly have more than four nodes, we are never required to consider a very large number of nodes. Moreover, we have a few results that reduce the number of candidate nodes even further.

If ψ has the additional property -

$$\forall m, m'(unsterm(m) = unsterm(m')) \Rightarrow (\psi(m) \text{ iff } \psi(m'))$$

Then all $\preceq$-minimal nodes of S are positive nodes, which means we can ignore all negative nodes in step iv. This proposition can be easily proved using the fact that for every negative node in the bundle, there is a positive node which precedes it. In addition, most ψ that we consider satisfy this property.

5.2. Using $\sqsubset$

We reason about the nodes using the $\preceq$ relation, in a similar way we reason about terms, using the $\sqsubset$ relation. The notion of an ideal corresponds to the notion of a minimal node in the case of the $\preceq$ relation. Ideals, allow us to obtain bounds on the capabilities of the intruder.

Recall that a k-ideal of a set S, denoted by $\mathsf{I}_k[S]$ is the smallest set containing S, that is closed under concatenation with arbitrary terms and encryption with keys in k. Depending on our choice of k, and ideal can represent the set of terms that a penetrator can create or interpret. We would like to reason about terms that the penetrator can create because authentication guarantees require that certain terms originate on regular nodes only. On the contrary, we would like to reason about the terms that the penetrator can interpret for providing secrecy guarantees. The notion of ideals is useful in both these situations.

The predicate ψ of the generic proof strategy usually involves ideals, and it is using the properties of ideals, we rule out most candidate nodes considered in step iv. In addition, ideals provide us generic bounds on the capabilities of the intruder, allowing us to completely skip penetrator nodes if the ideal structure is enough for the bound to apply. We mention one of such bounds, without a proof.

If $\mathcal{K} = S \cup k^{-1}$ and $S \cap \mathsf{Kp}$ is empty, then the entry points of $\mathsf{I}_k[S]$ (if any) are regular.

5.3. Proving Secrecy

Consider the Otway-Rees protocol, described in Figure 6.4. The strands of the protocol participants are given in §4.2.1. Suppose we wish to prove that the protocol preserves the secrecy of the session key K exchanged and also does not disclose the shared keys K_{AB} and K_{BS}. We restate the secrecy requirement of §4.3.1 -

$$\mathcal{S}(\phi, S) = \forall\, \mathcal{C}\, \forall\, s\, \forall\, n\ \phi(s) \wedge n \in \mathcal{C} \Rightarrow term(n) \notin \mathsf{I}_k[S]$$

The set of secrets S in our case is is $\{K, K_{BS}, K_{AS}\}$. ϕ is expected to contain assumptions about the origination of terms, penetrator knowledge and the structure of the strands. We include the following assumptions in ϕ

- K is uniquely originating.
- $K \notin \mathsf{Kp}$.
- $K \notin \{K_{AS} : A \in \mathcal{N}\}$
- $K = K^{-1}$

The generic proof strategy with $\psi(n) = term(n) \in \mathsf{I}_k[S]$ can be used to prove the property. We give a very brief proof outline next.

Firstly, note that the conditions required by the bound mentioned in the previous section are satisfied. This rules out any penetrator nodes from being minimal. All we need to check now is if any of the regular nodes, are minimal in the set. The structure of the strands and the assumptions in ϕ do not let that happen too. This proves that the set is empty, and thus the property holds.

5.4. Proving Authentication

If a principal sends a term encrypted by some other principals key and later receives it back in some cryptographically altered form, and if we know that the keys required for such a transformation are not accessible to the intruder, we can be sure that some regular principal must have operated upon it. An inference of this kind is called an authentication test[2]. We use these to prove authentication guarantees.

5.4.1. Some Notation

Recall that a_1 is a component of a_2 if $a_1 \Subset a_2$. A term is new at a node n, if it is a component of $term(n)$ and not a component of $term(n')$ if $n' \Rightarrow^+ n$. New components result from some cryptographic transformations. An edge $n_1 \Rightarrow^+ n_2$ is a transformed edge, for $a \in \mathcal{A}$ if n_1 is positive, n_2 is negative, $a \sqsubset n_1$ and there is a new component t_2 of n_2 such that $a \sqsubset t_2$. If n_1 is negative and n_2 is positive, the edge is called a transforming edge.

A term $t = \{h\}_K$ is a test component for a in n, if $a \sqsubset t$ and t is a component of n and t is not a proper subterm of any regular node in the strand space.

The edge $n_0 \Rightarrow^+ n_1$ is a test for a, if a uniquely originates at n_0, and $n_0 \Rightarrow^+ n_1$ is a transformed edge for a.

The edge $n_0 \Rightarrow^+ n_1$ is an outgoing test for a in $t = \{h\}_K$ if it is a test for a in which, K is inaccessible to the intruder; a does not occur in any component of n_0 other than t and t is a test component for a in n_0.

The edge $n_0 \Rightarrow^+ n_1$ is an incoming test for a in $t = \{h\}_K$ if it is a test for a in which, K is inaccessible to the intruder; and t is a test component for a in n_1.

An outgoing test, formally expresses the fact that a principal sends a freshly generated message encrypted by a key that is not known to the intruder and gets it back in an altered form. An outgoing test challenges the other principal to decrypt the test component. On the other hand, for an incoming test, the challenge is to encrypt a term, which is generated fresh and sent by the principal at n_0.

Let P be the set of keys including those in Kp and those that are learned during protocol executions. If we know what keys belong to P, then we can reason about the possible bundle structures which can contain these tests. This brings us to the main results of this section, the authentication tests.

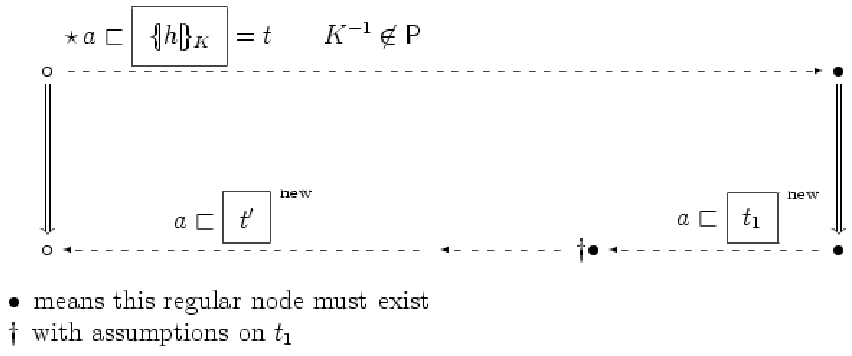
5.4.2. Authentication Tests

Having defined the required notation, we describe the main theorems which will be used to prove authentication guarantees of protocols.

. Outgoing Authentication Test

Let $\mathcal{C}$ be a bundle with $n' \in \mathcal{C}$, and let $n_0 \Rightarrow^+ n_1$ be an outgoing test for a in t, then

- There exist regular nodes $m, m' \in \mathcal{C}$ such that t is a component of m and $m \Rightarrow^+ m'$ is a transforming edge for a.
- Suppose in addition, that a occurs only in component $t_1 = \{h_1\}_{K_1}$ of m', that t' is not a proper subterm of any regular component, and that K_1^{-1} is not accessible to the intruder, then there is a negative regular node n'' with t_1 as component.



$\star a \sqsubset \boxed{\{\!|h|\!\}_K} = t$
$K^{-1} \notin \mathsf{P}$
$a \sqsubset \boxed{t'}$
new
$a \sqsubset \boxed{t_1}$
new
†
● means this regular node must exist
† with assumptions on t_1

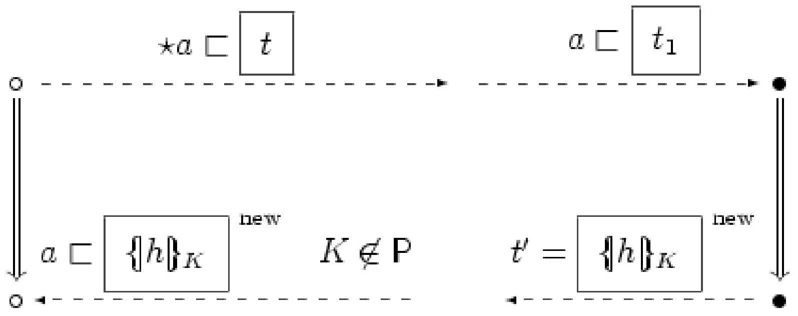
Figure 5.2.: The Outgoing Authentication Test

The intuitive meaning of this theorem is expressed in Figure 5.2. In the figure nodes marked with empty circles are n and n' and those marked by filled circles are m and m'. The theorem says that if a bundle has nodes n and n' which satisfy the conditions then it will also have nodes m, m' and n''. The next theorem, uses incoming tests to offer a similar guarantee.

. Incoming Authentication Test

Let $\mathcal{C}$ be a bundle, with $n' \in \mathcal{C}$, and let $n \Rightarrow^{+} n'$ be an incoming test for a in t'. Then these exists regular nodes $m, m' \in \mathcal{C}$ such that t' is a component of m' and $m \Rightarrow^{+} m'$ is a transforming edge for a.

As shown in Figure 5.3, this test allows us to infer existence of a regular transforming edge in a protocol where a nonce is sent in plain-text, and is expected to be replied encrypted.



$\star a \sqsubset \boxed{t}$
$a \sqsubset \boxed{t_1}$
$a \sqsubset \boxed{\{\!|h|\!\}_K}$ new
$K \notin \mathsf{P}$
$t' = \boxed{\{\!|h|\!\}_K}$ new

Figure 5.3.: The Incoming Authentication Test

. Unsolicited Authentication Test

In protocols which involve more than two participants, like those with key servers, a weaker test is often useful. This test uses the fact that reception of a term encrypted by some key which the intruder doesn't know, provides some hints about the node which originated the term.

A negative node n is an unsolicited test for $t = \{h\}_K$ if t is a component for any a in n and $K \notin P$ We can use an unsolicited test to obtain the following theorem, also called the unsolicited authentication test.

Let $\mathcal{C}$ be a bundle, with $n \in \mathcal{C}$, and let n be an unsolicited test for $t = \{h\}_K$. Then there exists a positive regular node $m \in \mathcal{C}$ such that t is a component of m.

5.5. Using Tests

The authentication tests, have been proved in [2]. In this section we will use them to prove them to prove authentication guarantees of the form mentioned in §2.3.1.

$$A(i, j, \phi, \psi) = \forall \mathcal{C}\ \forall s\ \exists s'\ \mathcal{C}\text{-height}(s) = i \wedge \phi(s) \Rightarrow \mathcal{C}\text{-height}(s') = j \wedge \psi(s, s')$$

One authentication requirement of the Otway-Rees protocol can be obtained by the following assignments - $\phi(s) = s \in Serv[A, B, S, N_a, N_b, M, *]$, i=1, $\psi(s, s') = s' \in Init[A, B, S, N_a, M, *]$, $j = 1$. This guarantee in plain words says that if the server has received a request for a session key, then the request indeed comes from an honest participant with the same bindings.

This can be proved, by verifying that the term $\{N_a MAB\}_{K_{AS}}$ is an unsolicited test and thus by the unsolicited authentication test theorem, there has to exist a regular node which sends it. Considering all the positive regular nodes we find that the only node that is capable of sending it is the first node on the initiator strand. Thus the strand with the same bindings exist.

Another authentication requirement of the Otway-Rees protocol can be obtained by these assigning these values to the general property - $\phi(s) = s \in Init[A, B, N_a, , M, K]$, i=2, $\psi(s, s') = s' \in Serv[A, B, S, N_a, *, M, K]$, $j = 2$. This can be read as - if the initiator ends its protocol execution, the keys that it gets originates on a server strand.

This can be proved by verifying that the the first two nodes on the initiator strand constitute an outgoing test for N_a in $\{N_a MAB\}_{K_{AS}}$. The outgoing test theorem ensures that there will a regular transforming edge for N_a. Again, considering the nodes will show that this edge can lie on no strand other than $s \in Serv[A, B, S, N_a, *, M, K]$.

This ends our discussion of how proofs of properties can be obtained. As is evident from most of the proofs we outlined, a lot of arguments are repeated again and again. Also, a lot of the arguments are independent of the actual protocol under consideration. This allows us to develop a general proof mechanism in PVS. The next chapter describes the details of the same.

6. PVS Theories

We describe a translation scheme from protocol descriptions to PVS theories. We also illustrate our PVS formalization of the strand spaces proof mechanism

The overall design of our translation scheme is shown in Figure 6.1. CAPSL specifications are first translated to an intermediate language representation which are then converted to PVS theories required from proving properties of protocols. Our PVS formalization

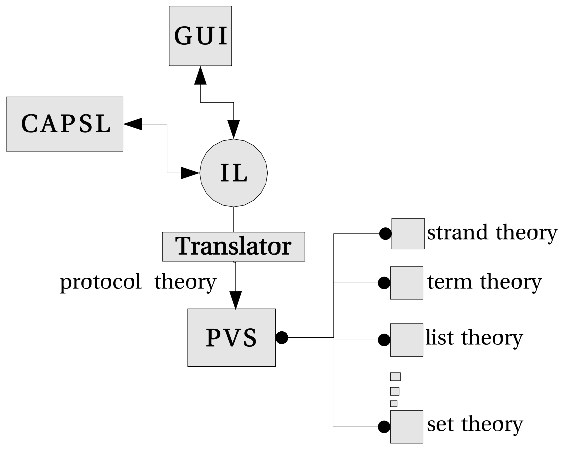


Figure 6.1.: Automated Generation of Strand Space Theories

separates the protocol dependent arguments used in proofs from the independent ones. The protocol independent arguments reside in theories that are included along with the core theories that PVS provides, allowing us to reuse these theories over protocols. Protocol specific arguments are contained in theories that are are generated automatically from the protocol description.

6.1. Protocol Independent Theories

The message algebra, the penetrator model, and the basic notions of strand spaces do not depend on the protocol under consideration. These are formalized as separate theories which protocol specific theories can use. We concentrate on proofs of secrecy properties in this chapter, and thus we describe only those portions of the theories that we will need later in the example proofs. The theories, in their entirety are expressive enough to handle any proof described in or based on the techniques in [1].

6.1.1. Messages

The set of all possible messages exchanged during a protocol run include messages of primitive types like text or keys and also compound types like encrypted or concatenated messages. In general, a message is an instance of a type `crypt_term`, which is defined as a PVS data-type. Messages of more specialized types can be created by using their corresponding constructors.

```
crypt_term: DATATYPE
WITH SUBTYPES key, text, name, encrypt, concat
BEGIN
  text(id: nat): text?: text
  key(id: nat): key?: key
  name(id: nat): name?: name
  encr(plain: crypt_term, enckey: key): encrypt?: encrypt
  join(a: crypt_term, b: crypt_term): concat?: concat
END crypt_term
```

Equality of terms is defined inductively using the `==` operator. Note that the definition takes into consideration the *freeness* assumptions described in §2.1.

```
==(x, y: crypt_term):  INDUCTIVE boolean =
CASES y
  OF  text(tid): text?(x) AND id(x) = tid,
      key(tid): key?(x) AND id(x) = tid,
      name(tid): name?(x) AND id(x) = tid,
      encr(enctext, enckey):
       encrypt?(x) AND
        (enctext == plain(x)) AND
         (enckey == enckey(x)),
      join(first, second):
       concat?(x) AND
        (first == a(x)) AND (second == b(x))
  ENDCASES
```

6.1.2. Penetrator Capabilities

The keys that the penetrator knows are modeled as a predicate on the set of keys `penetrated: pred[key]`. The penetrator's capability to learn terms given the set of keys he knows, is modeled using the notion of *ideals* described in §4.1.3.

Recall that a k-ideal of a term is the set of terms that can be constructed by concatenating terms already in the set with arbitrary terms or encrypting them with keys in k. Alternatively, k-ideal of a term t - $\mathsf{I}_k[t]$ is the set of terms from which t can be obtained if inverses of keys in k are known. The notion of an ideal, and the set `penetrated` allow us to succinctly express the set of terms that could potentially leak some secret. We use the set `secrecy_ideal` for this purpose.

```
secrecy_ideal(S: set[crypt_term], x: crypt_term):  INDUCTIVE
boolean =
S(x) OR
 (EXISTS (y, z: crypt_term):
    x == join(y, z) AND
     (secrecy_ideal(S, y) OR secrecy_ideal(S, z)))
  OR
  (EXISTS (y: crypt_term, k: key):
     x == encr(y, k) AND
      secrecy_ideal(S, y) AND penetrated(inv(k)))
```

Intuitively, when a principal sends a term term belonging to the `secrecy_ideal` corresponding to a set of secrets S, it sends some element of S without protecting it with proper keys. Thus, proving that a protocol preserves secrecy of a set of terms is equivalent to proving that no protocol principal ever sends a term that belongs to the `secrecy_ideal`.

Before we describe how such a proof can be attempted, recall that, the $\preceq$ relation models the causal dependencies of events in a protocol and that a *bundle* is a causally complete sub-graph of the strand space of a protocol. Intuitively, a bundle models one possible execution of a protocol so proving a property for all bundles establishes it for all executions of the protocol. Also recall that a bundle is well-founded i.e. all possible subset of nodes in a bundle posses $\preceq$-minimal nodes, which are nodes that are not causally dependent on other nodes of the subset. These minimal nodes form the base case of the inductive arguments that we use in proofs

6.1.3. Proving Secrecy

Thus we have reduced the problem of proving that a protocol preserves secrecy of a given set of terms to that of proving that the set of nodes, which send terms in the `secrecy_ideal` corresponding to the set, is empty. We begin by assuming that the set is non-empty and thus, by the well-foundedness of bundles, must have $\preceq$-minimal nodes. Next using induction on the structure of the strands of the protocol, if we can show that no node can satisfy the conditions required to be in the set and those required to be a minimal node, simultaneously, we reach a contradiction, thus proving that S is empty.

We will need to consider both regular and penetrator nodes to apply the strategy described above. While regular nodes depend on the protocol we analyze, the penetrator nodes are fixed and can be included in the protocol-independent part of the overall theory. Moreover, as mentioned in §5.1, using results from [1] we can reduce the number of nodes to be considered even further. (Note that these results can be proved in PVS using the axioms of our theories).In particular we use the result - *If m is minimal in $\{m : node \mid term(m) \in I\}$ then m is an entrypoint for I.* Recall that a node is an entrypoint for a set of terms I if it sends a term in I while no node that precedes it on its strand sends or receives any term in I. So, given a fixed set of penetrator strands, it is possible for us to selectively choose only those nodes which can possibly be entry-points. We define a formulae `no_penetrator_entrypoint` which returns true iff there is no penetrator node capable of being the entrypoint of give set of terms.

```
no_penetrator_entrypoint(I: set[crypt_term]): boolean =
  NOT ((EXISTS (t: text): I(t)) OR
   (EXISTS (k: key): penetrated(k) AND I(k)) OR
    (EXISTS (j: concat):
       ((NOT I(a(j))) AND (NOT I(b(j))) AND I(j)) OR
        (NOT I(j)) AND I(a(j)))
     OR
     (EXISTS (e: encrypt):
        (NOT I(plain(e))) AND (NOT I(enckey(e))) AND I(e) OR
         ((NOT I(e)) AND
           (NOT I(inv(enckey(e)))) AND
            penetrated(inv(enckey(e))) AND (I(plain(e))))))
```

Note that we consider only positive nodes on the penetrator strands(since negative nodes can not be entry-points), and every time we include the terms corresponding to a node in the set, we check whether terms of nodes preceding it on the same strand do not belong to the set. A similar `no_regular_entrypoint` formula is expected to be generated by the translation mechanism from the protocol specification. Thus, if we denote the set of secrets by `Secret`, then the set of terms that could potentially leak the secrets would be `LeakingTerms` and proving that no node leaks secrets is equivalent to proving `NoLeaks`.

```
Secret: set[crypt_term] = .. % filled by translator %

LeakingTerms: set[crypt_term] =
  LAMBDA (t: crypt_term): secrecy_ideal(Secret, t)

NoLeaks: THEOREM no_penetrator_entrypoint(LeakingTerms)
                   AND no_regular_entrypoint(LeakingTerms)
```

Thus, all we need now is the formula `no_regular_entrypoint`, which is generated from the protocol specification.

6.2. The Translation

We illustrate the translation using a simple protocol shown below, that was used by Nessett to demonstrate a flaw in the BAN logic analysis. The protocol claims to distribute a "good" session key between parties A and B, and clearly does not achieve this since encryption with the private key of A does not preserve the secrecy of the session key. We will use this simple protocol to illustrate our translation procedure and also show how a proof attempt of this protocol pin-points the flaw in its structure.

1. A $\longrightarrow$ B : $\{N_a, K\}_{K_A^{-1}}$
2. B $\longrightarrow$ A : $\{N_b\}_K$

The protocol can be specified in CAPSL or using a GUI, and the specification is then translated to an intermediate language description, which is defined as an XML vocabulary. A part of the specification for the current protocol, expressing the structure of messages that the protocol uses, is shown below.

```
<?xml version='1.0' ?>
<!DOCTYPE protocol SYSTEM 'crypa.dtd'>
<protocol label='nesset'>
  <terms>
   <name label='A'/>
   <name label='B'/>
   <text label='Na'/>
   <text label='Nb'/>
   <key label='Ka' type='private' principals='A'/>
   <key label='K'/>
   <encrypt label='one' terms='Na K' key='Ka'/>
   <encrypt label='two' terms='Nb' key = 'K'/>
  </terms>
  <roles> ... </roles>
  <goals> ... </goals>
</protocol>
```

The message structure can be converted to the PVS representation using the following strategy.

```
if tag is (key or text or name)
        print '@label:<tag>=<tag>(<unique_id>)'
if tag is encrypt
        print '@label:encrypt=encr(<join(@terms)>,@key)'
```

In the pseudo-code we use `tag` to denote the current element, `@` to access values of attributes of the current element and we enclose names in `< >` to denote that they must be treated as a functions whose return values must be inserted. Also, `unique_id` is a function returning a identifier that hasn't been used before and `join` is a function generating the PVS expression for a term which is concatenation of the terms in the argument string. For example, a call `join("A B C")` returns `join(A,join(B,C))`, which is the `crypt_term` corresponding to the concatenation of the terms in the string. We also use a construct `for-every` to apply a set of statements for every child element with the specified name. For instance `for-every role {print @label}` would print the value of the label attribute of all `role` children of the current node.

The remaining specification describes the protocol roles using the `role` element.

```
<protocol>
 <terms> ... </terms>
 <roles>
  <role>
    label='Init' self='A' parameters='A B Na Nb K'
    knows='B' originates='Na'>
   <send terms='one'/>
   <receive terms='two'/>
  </role>
  <role
    label='Resp' self='B' parameters='A B Na Nb K'
    originates='Nb'>
   <receive terms='one'/>
   <send terms='two'/>
  </role>
 </roles>
 <goals><secret terms='K'/></goals>
</protocol>
```

The role element has many attributes which are not required for proving secrecy properties which are our current concern. All that we need to prove whether or not a protocol preserves its secrets is the function `no_regular_entrypoint` which is true only if no regular node can be an entrypoint of the given set of terms. Such a function can be constructed using the terms that a participant sends and receives. The following translation results in an expression which is true if some regular node can be an entrypoint of a set of terms `I`.

```
for-every role
 for-every send
  print 'I(@terms)'
  for-every preceding-sibling
    print '(NOT I(@terms))'
  endfor
 endfor
 print 'OR'
endfor
```

The assumptions about the penetrator knowledge are translated next using the following strategy. All keys that the protocol claims to be keep secret are also assumed to be uncompromised.

```
print 'phi:AXIOM'
 for-every key
  if @type= (private OR session OR shared)
    print 'NOT penetrated(@label)'
  if @type='public'
    print 'NOT penetrated(inv(@label))'
 end-for
```

Lastly, the set `Secret` mentioned in the previous section is generated using the following translation

```
print 'Secret:set[crypt_term]=LAMBDA(t:crypt_term)'
 for-every token in secret@terms
  print 't == <token>'
 endfor
```

The complete PVS file generated using the translation mentioned in this section is given next. The theory `message_algebra` which this theory imports includes the protocol independent axioms used by the proofs.

```
nesset: THEORY
 BEGIN
  IMPORTING message_algebra
  A: name = name(1)
  B: name = name(2)
  Na: text = text(3)
  Nb: text = text(4)
  K: key = key(1)
  Ka: key = key(2)
  one: encrypt = encr(join(Na, K), Ka)
  two: encrypt = encr(Nb, K)
  Secret: set[crypt_term] = LAMBDA (t: crypt_term): (t == K)
  phi: AXIOM NOT (penetrated(inv(Ka)) OR penetrated(K))
  LeakingTerms: set[crypt_term] =
    LAMBDA (t: crypt_term): secrecy_ideal(Secret, t)
  no_regular_entrypoint(I: set[crypt_term]): boolean =
      NOT ((I(one)) OR (I(two) AND (NOT I(one))))
  empty_set(s: set[crypt_term]): boolean =
      no_regular_entrypoint(s) AND no_penetrator_entrypoint(s)
  secret1: THEOREM empty_set(LeakingTerms)
END nesset
```

6.3. Proof in PVS

Having generated the required theories, we can attempt the proof of the claim that the protocol guarantees secrecy of the session key. As it turns out, the proof gets stuck at the following sequent

```
[-1]  secrecy_ideal(Secret, join(text(3), key(1)))
[-2]  penetrated(inv(key(2)))
  |-------

Rule?
```

Here, `key(2)` is the key used to encrypt the session key. Note that the formula `[-1]` holds, since `key(1)` corresponds to the session key which is in the `Secret` set. So the success of the proof attempt boils down to whether or not `penetrated(inv(key(2))` holds, which does not. In other words, the proof attempts pin points the flaw in the structure of the protocol.

6.4. Otway-Rees Example

The intermediate language representation of the Otway-Rees protocol and its generated PVS theory is is given in Appendix-B. Note that both the intermediate representation and the PVS theory cleanly model the fact that the responder forwards components which it can not interpret, for the lack of the required key. A proof attempt provides useful insights on what properties of the uninterpreted terms the responder must be capable of checking for the protocol to guarantee secrecy.

7. Conclusion

The PVS formalization of the strand spaces proof mechanism and the generic translation mechanism are the two contributions of this report. The description of our PVS theories given here is incomplete. The theories can be used to prove much more than just the secrecy guarantees. For instance proofs of lemmas and propositions in [1] have been successfully attempted. Also, the formulas `no_regular_entrypoint` and `no_penetrator_entrypoint` that we generate from the protocol specification are general, and they work with sets of terms, other than ideals. In other words, proof of any property that can be expressed as a predicate on the terms that nodes send or receive can be attempted using our theories.

Proofs in the strand space mechanism are more intuitive than proofs which use other mechanisms[18]. Moreover, using already proved results[2] we can extend our theories to attempt proofs of properties like authentication, which haven't been proved in an automated setting. We have successfully proved authentication properties of a few protocols, and will soon generalize our theories for the same.

Bibliography

[1] J. Thayer, J. Herzog, J. Guttman. Strand Spaces: Proving Security Protocols Correct. Journal of Computer Security, Volume 7, Issue 2-3:191–230, 1999.

[2] Joshua D. Guttman, F. Javier Thayer Fábrega. Authentication tests and the Structure of Bundles. Journal of Theoretical Computer Science, 2001.

[3] D. X. Song, S. Berezin, A Perrig. Athena: A Novel Approach to Efficient Automatic Security Protocol Analysis. Journal of Computer Security, Volume 9, Issue 1-2:47–74, 2001.

[4] A. Perrig, D. Song A First Step Towards the Automatic Generation of Security Protocols In Symposium on Network and Distributed Systems Security (NDSS) 2000.

[5] Joshua D. Guttman Protocol Design via the Authentication Tests Appears in Computer Security Foundations Workshop, 2002

[6] D. Dolev and A. Yao On the security of public-key protocols IEEE Transactions on Information Theory, 2(29), 1983

[7] Catherine Meadows The NRL Protocol Analyzer: An Overview Journal of Logic Programming, Volume 26(2), 113–131,1996

[8] Stefanos Gritzalis, Diomidis Spinellis, Panagiotis Georgiadis. Security Protocols over Open Networks and Distributed Systems: Formal Methods for their Analysis, Design, and Verification. Computer Communications, 22(8):695–707, 1999.

[9] J. Clark and J. Jacob A Survey of Authentication Protocol Literature Available via http://www.cs.york.ac.uk/jac/papers/drareview.ps.gz

[10] Catherine Meadows Formal Verification of Cryptographic Protocols: A Survey ASIACRYPT: International Conference on the Theory and Application of Cryptology

[11] Levente Buttyan Formal methods in the design of cryptographic protocols (state of the art)

[9] G. Denker, J. Millen CAPSL and CIL Language Design: A Common Authentication Protocol Specification Language and Its Intermediate Language CSL Report SRI-CSL-9902, Computer Science Laboratory, International, Menlo Park CA 94025, 1999.

[11] Martin Abadi Security Protocols and Specifications Foundations of Software Science and Computation Structures: Second International Conference, FOSSACS '99. Vol-1578:1–13, 1999.

[12] CAPSL - Webpage http://www.csl.sri.com/users/millen/capsl/

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

[14] G. Denker and J. Millen, CAPSL Integrated Protocol Environment , DARPA Information Survivability Conference (DISCEX 2000).207–221, 2000. IEEE Computer Society

[15] M. Burrows and M. Abadi and R. Needham A Logic of Authentication ACM Transactions on Computer Systems, 8(1)18–36,1990

[16] M. Abadi and A. Gordon A calculus for cryptographic protocols: the Spi Calculus Digital Systems Research Center, SRC-149, 1998

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

[18] H. Rueß and J. Millen Local Secrecy for State-Based Models Formal Methods in Computer Security, CAV workshop. Chicago, July 2000.

[19] Steve Schneider Verifying Authentication Protocols with CSP PCSFW: Proceedings of The 10th Computer Security Foundations Workshop, IEEE Computer Society Press, 1997

[24] Gawin Lowe A Hierarchy of Authentication Specifications PCSFW: Proceedings of The 10th Computer Security Foundations Workshop, IEEE Computer Society Press, 1997

[25] Owre, S. and Shankar, N. and Rushby, J.M. and Stringer-Calvert, D.W.J. PVS Language Reference,Version 2.4 CSL,SRI. December 2001

[26] Owre, S. and Shankar, N. and Rushby, J.M. and Stringer-Calvert, D.W.J. PVS Prover Guide, Version 2.4 CSL,SRI. November 2001

[27] Owre, S. and Shankar, N. and Rushby, J.M. and Stringer-Calvert, D.W.J. PVS System Guide,Version 2.4 CSL,SRI. December 2001

A. Intermediate Language DTD

```
<!ENTITY % iden-attr 'label ID #REQUIRED
                     title CDATA #IMPLIED
                     desc CDATA #IMPLIED'>
<!ELEMENT protocol (terms,roles,goals)>
 <!ATTLIST protocol %iden-attr;
  title CDATA #IMPLIED>
<!ELEMENT terms (name|key|text|encrypt|term)*>
<!ELEMENT term EMPTY>
 <!ATTLIST term %iden-attr;
   interpret IDREF #REQUIRED>
<!ELEMENT text EMPTY>
 <!ATTLIST text %iden-attr;>
<!ELEMENT name EMPTY>
 <!ATTLIST name %iden-attr;>
<!ELEMENT key EMPTY>
 <!ATTLIST key %iden-attr;
         type (session|shared|public|private) "session"
         principals IDREFS #IMPLIED>
<!ELEMENT encrypt EMPTY>
 <!ATTLIST encrypt %iden-attr;
         terms IDREFS #REQUIRED
         key IDREF #REQUIRED>
<!ELEMENT roles (role)*>
<!ELEMENT role (send|receive)*>
 <!ATTLIST role %iden-attr;
   self IDREF #REQUIRED
   parameters IDREFS #REQUIRED
   knows IDREFS #IMPLIED
   originates IDREFS #IMPLIED>
<!ELEMENT send EMPTY>
 <!ATTLIST send terms IDREFS #IMPLIED>
<!ELEMENT receive EMPTY>
 <!ATTLIST receive terms IDREFS #IMPLIED>
<!ELEMENT goals (agreement|secret)*>
<!ELEMENT agreement EMPTY>
 <!ATTLIST agreement
   on-finish IDREF #IMPLIED
         agrees-on IDREFS #IMPLIED
         with IDREF #IMPLIED>
<!ELEMENT secret EMPTY>
 <!ATTLIST secret
   terms IDREFS #IMPLIED>
```

B. Otway Rees Example

B.1. Intermediate Language Specification

```
<?xml version='1.0' ?>
<!DOCTYPE protocol SYSTEM 'crypa.dtd'>
<protocol label='otwayrees'>
  <terms>
   <name label='A'/>
   <name label='B'/>
   <name label='S'/>
   <text label='M'/>
   <text label='Na'/>
   <text label='Nb'/>
   <key label='K'/>
   <key label='Kas' type='shared' principals='A S'/>
   <key label='Kbs' type='shared' principals='B S'/>
   <term label='H' interpret='one'/>
   <term label='H1' interpret='four'/>
   <encrypt label='one' terms='Na M A B' key='Kas'/>
   <encrypt label='two' terms='Nb M A B' key='Kbs'/>
   <encrypt label='three' terms='Na K' key='Kas'/>
   <encrypt label='four' terms='Nb K' key='Kbs'/>
  </terms>
 <roles>
  <role label='Init' self='A'
        parameters='A B S M Na K' knows='S B' originates='Na'>
   <send terms='M A B one'/>
   <receive terms='M three'/>
  </role>
  <role label='Resp' self='B'
        parameters='A B S M Nb K H H1' knows='S' originates='Nb'>
   <receive terms='M A B H'/>
   <send terms='M A B H two'/>
   <receive terms='M H1 four'/>
   <send terms='M H1'/>
  </role>
  <role label='Serv' self='S'
        parameters='A B S M Na Nb K' originates='K'>
   <receive terms='M A B one two'/>
   <send terms='M three four'/>
  </role>
 </roles>
 <goals> <secret terms='K'/> </goals>
```

```
</protocol>
```

B.2. PVS Theory

```
otway: THEORY
 BEGIN
  IMPORTING message_algebra
  A: name = name(1)
  B: name = name(2)
  S: name = name(3)
  M: text = text(4)
  Na: text = text(5)
  Nb: text = text(6)
  K: key = key(1)
  Kas: key = key(2)
  Kbs: key = key(3)
  H: crypt_term
  H1: crypt_term
  four: encrypt = encr(join(Nb, K), Kbs)
  one: encrypt = encr(join(Na, join(M, join(A, B))), Kas)
  three: encrypt = encr(join(Na, K), Kas)
  two: encrypt = encr(join(Nb, join(M, join(A, B))), Kbs)
  Secret: set[crypt_term] = LAMBDA (t: crypt_term): t == K
  phi: AXIOM NOT (penetrated(K)
                  OR penetrated(Kbs)
                  OR penetrated(Kas))
  inverses: AXIOM inv(K)=K AND inv(Kas)=Kas AND inv(Kbs)=Kbs
  LeakingTerms: set[crypt_term] =
    LAMBDA (t: crypt_term): secrecy_ideal(Secret, t)

  no_regular_entrypoint(I: set[crypt_term]): boolean =
   NOT ((I(join(M, join(three, four))) AND
      (NOT I(join(M, join(A, join(B, join(one, two)))))))
      OR
      (I(join(M, join(A, join(B, one)))) OR
       (I(join(M, join(A, join(B, join(H, two))))) AND
         (NOT I(join(M, join(A, join(B, H))))))
        OR
        (I(join(M, H1)) AND
          (NOT I(join(M, join(A, join(B, H))))) AND
           (NOT I(join(M, join(A, join(B, join(H, two)))))) AND
            (NOT I(join(M, join(H1, four))))))
  empty_set(s: set[crypt_term]): boolean =
      no_regular_entrypoint(s) AND no_penetrator_entrypoint(s)
  secret3: THEOREM empty_set(LeakingTerms)
 END otway
```