Part VI

Appendices

'BAN logic' does not uniquely identify one single article in which it is introduced. The relation between the many versions of the 'BAN paper' that exist is explained. We survey the major critisims of BAN logic in the literature.

Appendix A

Remarks to Authentication Logics

A.1 A Taxonomy of Versions of the BAN Paper

(Referred to on pages 47 and 56.)

The seminal paper "A Logic of Authentication" has a respectable number of versions. Its precursor, "Authentication: A Practical Study in Belief and Action" was presented at the second conference on Theoretical Aspects of Reasoning About Knowledge in March 1988 [BAN88, 18 pages]. Then, there is the DEC technical report, which was published in February 1989 and revised in February 1990 [BAN89a, 49 pages]. In April 1989, the work was submitted to the Royal Society of London, which published it in December 1989 [BAN89b, 39 pages]. Also in December 1989, a revised version of the article was presented on the twelfth ACM Symposium on Operating Systems Principles, which was also published in the ACM SIGOPS Operating Systems Review [BAN89c, 13 pages]. This led to a paper in the ACM Transactions on Computer Systems in February 1990 [BAN90a, 19 pages]. In May 1994, an appendix to the DEC technical report was published [BAN94, 10 pages].

The most notable distinction between these versions is that in the ACM-published versions and the DEC appendix, the notation of many operators has changed from symbols (e.g. $\models$) to linguistic terms (e.g. **believes**). These versions refer to the DEC technical report for full reference. The DEC technical report and the Royal Society version [BAN89a, BAN89b] should be considered the most complete versions, due to their size and the fact that these papers are most often used in self-references of the authors. Martín Abadi considers the Royal Society version the most definite one (on his homepage). These two versions of the article contain a Section 12, "On Hashing", which introduces and discusses the inference rule essential in Chapter 5 of this thesis. These two versions also contain a Section 13, "Semantics", which defines the partial semantics for BAN logic, used in Sect. 5.6 of this thesis.

A.2 A Short Survey of Critisisms on BAN Logic

(Referred to on page 54.)

The main criticisms of BAN logic regard the following properties of the logic:

1. the semantics,
2. the notion of belief,
3. the protocol idealization method,
4. the honesty assumption, and
5. the incompleteness.

These weaknesses of BAN logic have not been a reason to abandon the way of thinking introduced by Burrows, Abadi and Needham. Many researchers have tried to 'repair' BAN logic. The following pages we will give a short survey of the critiques and how and where they have been addressed in literature.

The first two critiques of the list above are, in more detail:

1. *BAN logic has no (well-defined) semantics*
   It is not really clear what the constructs in the logic actually represent.[1]
2. *BAN logic does not distinguish between possession and belief*
   Normally, one would like to distinguish between possessing a sentence (e.g. "Clapton is God") and believing such a sentence.

One of the first attempts to fix BAN logic was by Abadi and Tuttle, who introduced AT logic [AT91], which has a slightly better semantics than the original BAN logic. A second attempt was by Gong, Needham and Yahalom, who introduced GNY logic [GNY90]. GNY logic distinguishes between possession and belief, but has a semantics just about as poor as BAN logic.

Authentication logics with the goal to have a well-defined semantics *and* a clear distinction between possession and belief are VO logic [vO93], SVO logic [SvO94, SvO96], AUTLOG [KW94, WK96] and SVD logic [Dek00]. There are many crossbreeds of these logics, and the logics have heavily influenced one another. Often, it is relatively easy to translate protocols and formulae between these logics.

Except for 'repairs' of the original BAN logic, extensions have also been available in abundance. Here we will mention just a few of the extensions.

[1] For a compact treatment on the value of semantics for authentication logics, consult [Syv91].

Gaarder and Snekkenes added time-related concepts to BAN logic [GS91], and Syverson did something similar for AT logic [Syv93]. Kailar and Gligor extended BAN logic to make it less dependent on *jurisdiction*[2] [GKSG91, KG91].

Analyzing a protocol by hand is a tedious task, and some of the logics have been designed in such a way that computer-aided analysis can be performed. In particular, AUTLOG is implemented in PROLOG [KW94], and SVD has an implementation in the Isabelle theorem prover [Dek00][3]. Also, model checkers have been used for protocol analysis, most notably by Lowe who found a major error in the Needham-Schroeder Public-Key protocol (NSPK) in this way [Low96].

Authentication logics have been tied to radically different approaches for analyzing security protocols. One of them is the strand-space methodology [THG98, THG99]. Van Oorschot used Strand-spaces to create yet another semantics [Syv00], and Jacobs used it to create a new logic with an accompanying implementation in the theorem prover PVS [Jac04][4].

The semantics and the notion of belief have not been the only aspects inviting criticism. The remaining three important avenues of criticism are:

3. *BAN logic has a rather vague 'protocol idealization method'*
   The process of translating a protocol into the language of the logic is poorly described and depends on 'intuitions an intentions'. Clearly, a rigorous description would be better.

4. *BAN logic assumes honest principals*
   Within BAN logic, it is impossible to model principals which state lies. This limits the kind of protocol flaws that can be found using BAN logic, as lying can sometimes be relatively easy and computationally cheap.[5]

5. *BAN logic is incomplete*
   There are protocol errors which BAN logic fails to identify; in fact, the class of such protocol errors contains some rather obvious errors.

Critiques on the protocol idealization method of BAN logic have appeared in [MB93, WK96]. In general, one can say that this problem has been addressed in almost all authentication logics. Criticism number 4 (honesty), stated in [GKSG91], boils down to criticism number 2 (possession and belief), and has been resolved in various ways in most authentication logics.

Incompleteness of BAN logic has been demonstrated by Nessett [Nes90], who showed a protocol which has a very obvious flaw, which cannot be de-

[2] Jurisdiction is the concept that a specific principal in a protocol has designated authority over some statements. Making BAN logic less dependent on jurisdiction therefore widens the class of protocols that can be analyzed using BAN logic.

[3] For background on Isabelle, consult [NPW02].

[4] For background on PVS, consult [ORSvH95].

[5] Thus, BAN logic supposedly is a logic in the Honest-But-Curious (HBC) attacker model, instead of in the Dolev-Yao threat model.

tected in BAN logic[6]. In some circumstances, incompleteness of BAN logic is also due to incorrect protocol idealization [BM94].

Solving the problem of incompleteness of authentication logics is difficult for a number of reasons. One of the challenging problems is that while strengthening the logic, the modeled inference capabilities (computational resources) of principals should remain the same. Thus, the inference capabilities of the principals should be constrained (as in [ABV01]). This approach has led to preliminary completeness results for authentication logics [CD05b, CD05a].

Though the original BAN logic has serious limitations, the way of reasoning is useful. The way of reasoning should not be abandoned because of the flaws in the original logic [HPvdM03]. Moreover, recent results show that it is possible to create a computational justification of authentication logics [AR02].

[6] Burrows, Abadi and Needham replied to this critique essentially by stating that they never claimed this would be the case [BAN90b]. Moreover, they felt the urge to humiliate Nessett by calling one of his assumptions "absurd" and questioning his "wit of a man to notice" [BAN90b, page 40].

In this thesis, we use an extension of GNY logic to analyze the T-1 protocol. We summarize the formal language and inference rules of GNY logic, as presented in [GNY90]. Parts of the language we do not use are omitted.

Appendix B

Summary of GNY Logic

(Referred to extensively from Chapters 4, 6 and 9.)

In this appendix, we summarize the formal language and inference rules of GNY logic [GNY90]. Parts of the logic that we do not use are omitted.

B.1 Formal Language

A formula is a name used to refer to a bit string, which would have a particular value in a protocol run. Let X and Y range over formulae, and $+K$ and $-K$ over public keys and private keys respectively. The following are also formulae:

(X, Y)	conjunction of two formulae. We treat conjunctions as sets with properties such as associativity and commutativity.
$\{X\}_{+K}$	public-key (asymmetric) encryption.
$\{X\}_{-K}$	private-key (asymmetric) signature We assume a cryptosystem for which $\{\{X\}_{+K}\}_{-K} = X$ holds (i.e., encryption), and for which also $\{\{X\}_{-K}\}_{+K} = X$ holds (i.e., signatures, e.g. RSA [RSA78]).
$H(X)$	the hash value of X obtained by application of a strongly collision-free cryptographic hash function (CRHF).
$*X$	a not-originated-here formula. A formula X is a not-originated-here formula if a principal receives X without having sent X itself before. Thus, a formula is a not-originated-here formula for a principal P, if it is not a replay of one of P's previously sent messages.

Assertions reflect properties of formulae. Let P and Q be principals. The following are basic assertions:

$P \triangleleft X$	P *is told* formula X. P receives X, possibly after performing some computation such as decryption.
$P \ni X$	P *possesses*, or is capable of possessing, formula X. At a particular state of a run, this (X) includes all the formulae P has been told, all the formulae he started the session with, and all the ones he has generated in that run. In addition P possesses, or is capable of possessing, everything that is computable from the formulae he already possesses.
$P \mid\sim X$	P *once conveyed* formula X. X can be a message itself or some content computable from such a message, i.e., a formula can be conveyed implicitly.
$P \mid\equiv \sharp(X)$	P *believes*, or is entitled to believe, that formula X is *fresh*. That is, X has not been used for the same purpose at any time before the current run of the protocol.
$P \mid\equiv \phi(X)$	P *believes*, or is entitled to believe, that formula X is *recognizable*. That is, P would recognize X if P has certain expectations about the contents of X before actually receiving X. P may recognize a particular value or a particular structure.
$P \mid\equiv P \overset{S}{\leftrightarrow} Q$	P *believes*, or is entitled to believe, that S is a suitable *secret* for P and Q. They may properly use it to mutually prove identity. They may also use it as, or derive from it, a key to communicate. S will never be discovered by any principal except P and Q.
$P \mid\equiv \overset{+K}{\mapsto} Q$	P *believes*, or is entitled to believe, that $+K$ is a suitable *public key* for Q, i.e., the matching secret key $-K$ will never be discovered by any principal except Q.

Let C range over assertions. The following are also assertions:

$P \mid\equiv C$	P *believes*, or is entitled to believe, that formula C *holds*.
C_1, C_2	conjunctions. We treat conjunctions as sets with properties such as associativity and commutativity.

B.2 Inference Rules

We repeat all inference rules from [GNY90] used in this thesis. The rule names correspond to the names used in the original article. The first letter of the name intends to reflect the category of the rule: **T** is about being **t**old, **P** about **p**ossession, **F** about **f**reshness, **R** about **r**ecognizability, and **I** about message interpretation. Some of the inference rules (**P2**, **F1**, **I3**) have more allowable conclusions than used in this thesis. These unused conclusions are omitted.

An inference rule that applies to formula X also applies to $*X$, though not necessarily vice versa. If $\frac{C1}{C2}$ is an inference rule then for any principal P so is $\frac{P \mid\equiv C1}{P \mid\equiv C2}$.

T1	$\frac{P \triangleleft *X}{P \triangleleft X}$	Being told a 'not-originated-here' formula is a special case of being told a formula.
T2	$\frac{P \triangleleft (X, Y)}{P \triangleleft X}$	Being told a formula implies being told each of its concatenated components.
T6	$\frac{P \triangleleft \{X\}_{-K},\; P \ni +K}{P \triangleleft X}$	If a principal is told a formula encrypted with a private key and he possesses the corresponding public key then he is considered to have also been told the decrypted contents of that formula. This rule only holds for public-key systems with the property $\{\{X\}_{-K}\}_{+K} = X$ (e.g. RSA [RSA78]).
P1	$\frac{P \triangleleft X}{P \ni X}$	A principal is capable of possessing anything he is told.
P2	$\frac{P \ni X,\; P \ni Y}{P \ni (X, Y)}$	If a principal possesses two formulae then he is capable of possessing the formula constructed by concatenating the two formulae.
P3	$\frac{P \ni (X, Y)}{P \ni X}$	If a principal possesses a formula then he is capable of possessing any one of the concatenated components of that formula.
P4	$\frac{P \ni X}{P \ni H(X)}$	If a principal possesses a formula then he is capable of possessing the hash value of that formula obtained by application of a strongly. collision-free (collision resistant) cryptographic hash function (CRHF)
P8	$\frac{P \ni -K,\; P \ni X}{P \ni \{X\}_{-K}}$	If a principal possesses a formula and a private key then he is capable of possessing the decryption of that formula with that key (i.e., the cryptographically signed formula).
F1	$\frac{P \mid\equiv \sharp(X)}{P \mid\equiv \sharp(X, Y)}$	If a principal believes a formula X is fresh, then he is entitled to believe that any formula of which X is a component is fresh.

R6 $$\frac{P \ni H(X)}{P \models \phi(X)}$$ If P possesses a formula $H(X)$, then he is entitled to believe that X is recognizable.

I3 $$\frac{P \lhd *H(X,S), \quad P \ni (X,S), \quad P \models P \overset{S}{\leftrightarrow} Q, \quad P \models \sharp(X,S)}{P \models Q \mid\!\sim (X,S)}$$

Suppose that for principal P all of the following hold: (1) P receives a formula consisting of the hash value of X and S marked with a not-originated-here sign; (2) P possesses S and X; (3) P believes that S is a suitable secret for himself and Q; (4) P believes that either X or S is fresh. Then P is entitled to believe that Q once conveyed the formula X concatenated with S.[1]

I4 $$\frac{P \lhd \{X\}_{-K}, \quad P \ni +K, \quad P \models \overset{+K}{\mapsto} Q, \quad P \models \phi(X)}{P \models Q \mid\!\sim X, \quad P \models Q \mid\!\sim \{X\}_{-K}}$$

If P sees a signed message $\{X\}_{-K}$, knows the public key $+K$, knows the corresponding private key $-K$ belongs to Q, and recognizes X to be a message, then P is entitled to believe that Q once conveyed the signed message $\{X\}_{-K}$, and thus also once conveyed the message X itself.

I6 $$\frac{P \models Q \mid\!\sim X, \quad P \models \sharp(X)}{P \models Q \ni X}$$ If P believes that Q once conveyed formula X and P believes that X is fresh, then P is entitled to believe that Q possesses X.

[1] It should be noted that rule **I3** as given here differs slightly from the definition in [GNY90], which uses the notation $\langle S\rangle$ in some places instead of S to denote that S is used for identification. This is non-essential and only syntactic sugar. For readability of the proofs, these brackets have been omitted throughout this thesis.

There are many articles closely related to knowledge authentication which deserve some detailed comments because they contain minor (or major) flaws, while the comments are outside of the scope of Chapter 8 (Knowledge Authentication). These comments are bundled here.

Appendix C

Remarks to Knowledge Authentication

C.1 The 'French Approach'

(Referred to on page 115.)

In Table 8.1, a list of protocols is shown. The protocols presented in [DQB95, QBA$^+$98a, QBA$^+$98b, QAD00, Ber04, CC04] are omitted from this table. These omitted protocols are mainly protocols developed in the medical domain, and it is rather difficult to qualify these protocols without being harsh and impolite. In short: it seems that the thought that the mere application of cryptography would solve all problems prevented a clear formulation of the threats for which the protocols should offer a solution. We acknowledge that these are very strong claims. Nevertheless, they seem appropriate. Some examples:

- In [QBA$^+$98a, QBA$^+$98b] quality assesment of the protocols is performed, quoting figures on sensitivity and specificity. This type of assesments implies that privacy and validity (as defined in Section 2.7 and summarized at the start of this section) are not considered for granted.
- In [QBA$^+$98a, QAD00] the term "cryptology" is used where 'cryptography' is supposedly intended.
- In [Ber04] a protocol is devised that should be "zero-knowledge" but it is never explained what zero-knowledge actually means, nor contains the paper any reference to a paper about zero-knowledge.
- In [CC04] it is stated that the use of a MAC can prevent a dictionary attack. This is only the case when one assumes the principals who know

the key to the MAC are essentially honest — this makes the whole excercise of devising a secure protocol for computing set relations useless.

These publications sometimes use terms like "minimal knowledge", which can be considered a confession that some knowledge (other than the set sizes) is leaked. It suggests that zero-knowledge is impossible.

C.2 On the Probabilistic Communication Complexity of Set Intersection

(Referred to on page 118.)

In the context of sparse sets, it is appropriate to clarify an often misinterpreted result by Kalyanasundaram, Schnitger and Razborov (from here on: KSR) [KS92, Raz92][1]. For example, in [FNP04] it is claimed that the result of KSR implies that the lower bound for communication complexity of the secure computation of set intersection is at least proportional in $|n|$. All men are mortal ([KS92, Raz92]), therefore Socrates is mortal ([FNP04]), so it seems.

KSR show that the probabilistic communication complexity of disjointness is $\Theta(n)$ (where $n = |\Omega|$)[2]. This result applies to a problem which is more general than the problems described in Chapter 8. In particular:

1. The result of KSR applies to the communication complexity of a *problem*, and not to the communication complexity of a *particular protocol*. In particular, they assume the principals have unlimited computational power. Thus, the *efficiency* property as mentioned at the start of Section 8.5 does not apply to the computational resources, only to the communication resources.

2. The problem analyzed by KSR does not include *privacy* or *validity* concerns: the principals are implicitly assumed to be honest, and do not care whether their 'private' information is disclosed. One could say the honest adversary model is assumed.

3. The problem analyzed by KSR applies to the *disjointness* problem (in terms of Figure 8.2: f_{disj}), and not to the *intersection* problem (f_{int}). The intersection problem is more difficult than the disjunction problem[3].

[1] The result is oringinally published by Kalyanasundaram and Schnitger in [KS92]. It has been greatly simplified by Razborov [Raz92].

[2] It is difficult to use consistent notation which is not misleading. As we have defined our domain of set items to be Ω, we might as well write $\Theta(|\Omega|)$, but Ω has also a meaning in complexity. Therefore, we introduce n. We admit this notation is far from optimal.

[3] If, for some X and Y one knows $f_{\text{int}}(X,Y)$, one can easily infer $f_{\text{disj}}(X,Y)$. Observe that

$$f_{\text{disj}}(X,Y) = \begin{cases} 1 & \text{if} \quad f_{\text{int}}(X,Y) = \emptyset \\ 0 & \text{if} \quad f_{\text{int}}(X,Y) \neq \emptyset \end{cases}$$

The opposite is not the case: one cannot always infer $f_{\text{int}}(X,Y)$ from $f_{\text{disj}}(X,Y)$.

4. In the problem analyzed by KSR, no assumptions are made on the set sizes.

It is tempting to project these results to the problems described in Chapter 8. As the result by KSR is more general, these results would, so it seems, also apply to the problems analyzed in Chapter 8.

The generality of the result by KSR is misleading. The first three observations listed above seem to warrant a projection of the results of KSR to secure computation of set relations. The fourth observation, that there is no assumption on the set sizes, prevents projection of their results to secure computation of set relations. With extra assumptions on the set sizes, it is possible to construct protocols which are more efficient than the lower bound derived by KSR. In particular, if the sets cover only a sparse fraction of the domain Ω, it is possible to devise protocols whose communication complexity is below $|\Omega|$. And in fact, the protocols presented in [FNP04] are an example of this.

It has to be admitted that [KS92] gives opportunities to misinterpret the result. The abstract starts with:

> "It is shown that, for inputs of length n, the probabilistic (bounded error) communication complexity of *set intersection* is $\Theta(n)$."

With "inputs of length n", the authors tacitly mean 'inputs of length n *if encoded in a specific manner, namely …*'. Moreover, where they write "set intersection", the authors actually mean 'intersection cardinality > 0', which is the dual of *set disjointness* (see Figure 8.2 on page 110).

Therefore, the result of KSR *does not* imply that the lower bound for communication complexity of the secure computation of set intersection is at least proportional in $|n|$.

C.3 Fuzzy Private Matching

In [FNP04], on pages 16 and 17, a problem closely related to knowledge authentication is presented: Private Fuzzy Matching. The problem is not only to find exact matches, but also 'close matches', which they define well. The protocol they present on page [FNP04, page 17] is however incorrect. This problem has been identified by Łukasz Chmielewski. In [CH06], a corrected protocol, and other protocols for this problem are presented.

The T-1 protocol has a prototype implementation as a Java application. It is explained how this prototype is operated. The prototype allows one to stress-test the T-1 protocol.

Appendix D

The Secret Prover

With use of the T-1 protocol, it is possible to prove possession of arbitrary secrets (Chapter 9). The 'Secret Prover' is a prototype implementation of the T-1 protocol. With it, people can prove possession of files, and explore how the protocol works. The Secret Prover is a Java application[1], which can be downloaded from `http://www.teepe.com/phdthesis/demo` . Java 1.4 or newer is required, which is available for Mac OS, Windows and many Unices, including Linux.

In this appendix, we will talk you through the whole application, in such a way that you can run the protocol yourself. No deep understanding of the T-1 protocol is required. Running the protocol hands-on, with some help from the screenshots printed in this chapter, may even help to gain some basic understanding of the T-1 protocol.

The Secret Prover has the following features:

- Execution of all three configurations of the protocol (the verifier initiates, the prover initiates, mutual proof).
- Support for the following hash algorithms: MD5, SHA-1, SHA-256, SHA-384 and SHA-512.[2]
- Support for using a nonce, or encryption. The following encryption algorithms are supported: DES, Triple DES, Blowfish.
- The transport mechanisms used is TCP/IP. (i.e., it works over the internet.)
- There is no authentication of the identity of the principals in the protocol. The authenticity of the communication channel is also not guaranteed.

[1] The current version number is 0.03.

[2] For SHA-256, SHA-384 and SHA-512, Java 1.5 is required. If one side of the protocol tries to use one of these hash algorithms, while it is not supported at the other side of the protocol, this is detected and a warning is given.

- Multiple proof messages can be sent per protocol.[3]
- Protocols can be interleaved.
- It is possible to perform 'fake' actions in the protocol.
- It is possible to trim down ('mutilate') hash values to a few bits, to simulate the effect of hash collisions on the protocol.

If you want to run the prototype yourself and experiment with it, you should read on from here. If on the other hand you only want so look how it works, without running anything yourself, you may skip Section D.1, and if you are very impatient, you might skip Section D.2 as well.

D.1 Starting Up and Connection Control

Once you have downloaded the application file you can start it up. On Unix-like operating systems, this is done by typing

```
# java -jar Prover.jar &
```

at the console. In more graphical environments, like Mac OS X, the file can simply be 'double-clicked'. Once the application has launched, you will see appear the main application window (Figure D.1).

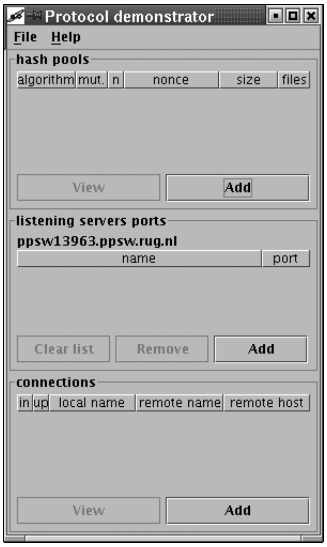


FIGURE D.1:
Main application window.

The main application window consists of four parts, from top to bottom:

A menu bar From here you can stop the application or open the splash screen.

Hash pools This is where pre-computed hash values are stored, it will be described in Section D.2.

Listening server ports Here all connection listeners are managed. When you open a connection listener, someone can contact you by opening a connection to the specified host and port.

Connections Here all connections are managed. You may contact somebody, and the connection will be displayed here. Also, if you have a connection listener open, if somebody contacts you through the listener, the connection will show up here.

[3] This is a slight generalization of the T-1 protocol. In the T-1 protocol, if the prover is also the initiator, he can only send one single h_2 value. In the prototype, an initiating prover may send multiple h_2 values. This generalization has been implemented to ease the process of experimenting with the protocol.

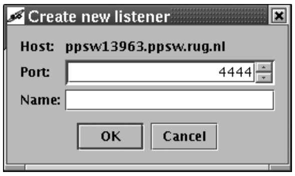


FIGURE D.2: Opening a connection listener.

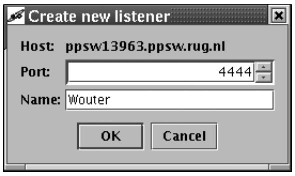


FIGURE D.3: Filling in a name.

D.1.1 Opening a Connection Listener

The TCP/IP protocol works using 'meeting places'. A meeting place is a combination of a computer (a host, an IP address) and a port (a number). The owner of a computer may set up a meeting place by listening at the spot of the number. Anybody else contacting the host at the port will get connected. For any connection, one of the participants has to set up such a 'meeting place'.

By clicking on 'Add' in the 'Listening server ports' pane of the main application window (Figure D.1), you can set up such a meeting place. The host is already set to reflect your computer's IP address, but you do have to provide a port number and a name (Figure D.2). The port number is set to a reasonable default, but you really should fill in the name field. This name is not essential to setting up a connection, but it will be used in the protocols later on.

Throughout the examples I will use the name 'Wouter' (Figure D.3). Later on in the example, a second player will be introduced, and for the second player I will use the name 'Kathy'. In the text I will sometimes refer to Kathy or Wouter. Obviously, you are free to use other names. You should however take note which of the names you use correspond to the roles of Kathy and Wouter, because it may help you to understand the explanation.

If the port for some technical reason or another cannot be used as a meeting place, you will be informed so, and in that case you should use another port number and try again.

Now you may by whatever means you see fit inform somebody else to contact you at the specified host and port. The host is the name or IP address of your computer. For your convenience, this is displayed in the 'Listening server ports' pane of the main application window. The port number is the number you just specified.

Instruct some friend or colleague to run the software as well. The best and most convincing way to see how it works is to run the protocol with somebody else. However, if you lack somebody to play the game with, or don't want to bother anybody, you may connect yourself to the server and the port. That is what we're going to do in the next section.

D.1.2 Making a Connection

By clicking 'Add' in the 'Connections' pane of the main application window (Figure D.1), you get the 'connection' window shown in Figure D.4. Since

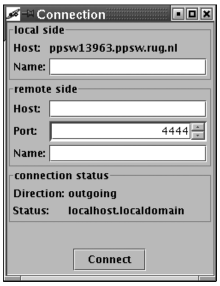

Connection
local side
Host: ppsw13963.ppsw.rug.nl
Name:
remote side
Host:
Port: 4444
Name:
connection status
Direction: outgoing
Status: localhost.localdomain
Connect

FIGURE D.4: Making a connection.

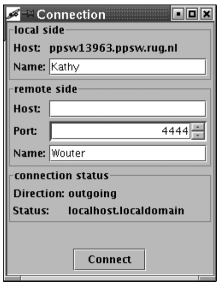

Connection
local side
Host: ppsw13963.ppsw.rug.nl
Name: Kathy
remote side
Host:
Port: 4444
Name: Wouter
connection status
Direction: outgoing
Status: localhost.localdomain
Connect

FIGURE D.5: Filling in connection details.

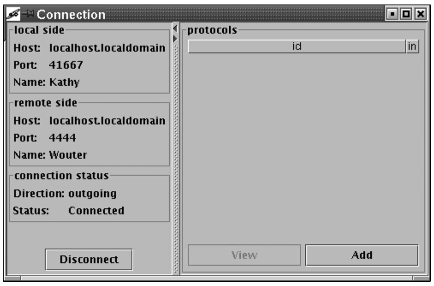

Connection
local side
Host: localhost.localdomain
Port: 41667
Name: Kathy
remote side
Host: localhost.localdomain
Port: 4444
Name: Wouter
connection status
Direction: outgoing
Status: Connected
Disconnect

FIGURE D.6: An initiated connection (outgoing).

we are connecting to ourselves, we should distinguish between our own two 'egos'. A good way to easily distinguish these is to put all windows concerning ego #1 (Wouter) on the left hand side of the screen, and all windows concerning ego #2 (Kathy) of the right hand side of the screen. (Now drag the connection window to the right hand side of your screen.)

In the top of the connection window, you see the 'local side' pane: this is about who you (the connecting side) are. Choose some name for your ego #2 and fill it in. Below that, you see the 'remote side pane': this is what meeting place (host and port) you are connecting to, and who you expect to find there (Figure D.5). If you leave empty the host field, it will connect to the local computer. (And we have just opened a listener on the local computer on port 4444, so this will all work out.) As soon as you click 'Connect', the software will try and make the connection for you. The window will 'grow' and transform into the window shown in Figure D.6.

When you have done this, you will see that some more windows have popped up. Most importantly, a window has popped up that belongs to the listening port (Figure D.7). If you would not have connected to yourself, but to somebody on another host (i.e., another computer), the window would have come up there.

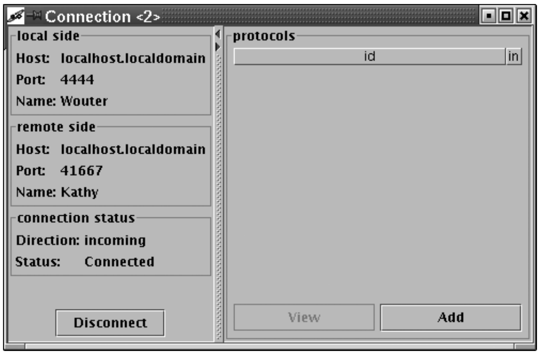
Also, two 'authentication' windows have popped up (Figure D.8), which inform both sides of the connection that some authentication should take place.[4]

You may click on 'OK' to close the authentication windows.

Of course, it may happen that the person responding on the meeting place is not the person you expected to meet there. In that case an 'authentication mismatch' warning is given (Figure D.9).[5]

[4] The T-1 protocol requires an authenticated communication channel. We haven't implemented this authentication, because it would complicate the setup process even more, while the authentication is not essential for demonstrating the protocol. Of course, the authentication is essential in the sense that without it, anybody can claim to be anybody else, which is clearly undesirable. Any production implementation should obviously include authentication of the players and of the communication channel.

[5] Since no real authentication is done in this prototype, this warning is mainly cosmetic.

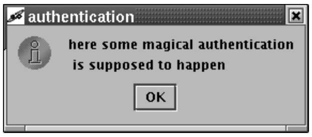


Host: localhost.localdomain
Port: 4444
Name: Wouter

Host: localhost.localdomain
Port: 41667
Name: Kathy

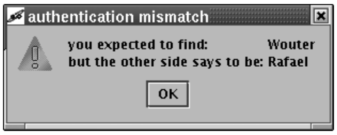
Direction: incoming
Status: Connected

FIGURE D.7:
Receiving a connection (incoming).



here some magical authentication
is supposed to happen

FIGURE D.8:
An authentication warning.



you expected to find: Wouter
but the other side says to be: Rafael

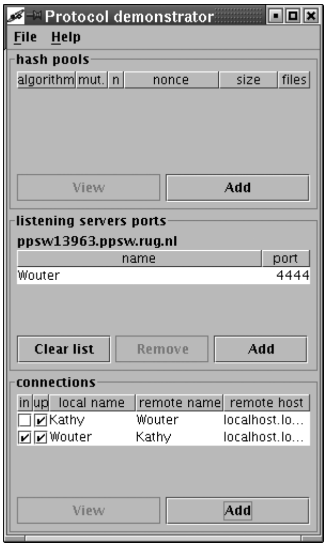
FIGURE D.9:
An authentication mismatch.

If all has gone well, the main application window will look like Figure D.10:

Listening server ports In this pane a line has appeared which informs you the port is listening for new connections. You may select ports and remove/close them using the buttons in the pane.

Connections In this pane, the two sides of the connection are shown. If you would be connecting to somebody on another host, only one line would have shown up. If you click the 'View' button, the window containing the connection details will be shown (either Figure D.6 or D.7).[6]

Congratulations! you have passed the 'boring' part of the prototype! In the next section, we will pre-compute hash values, and store them in *hash pools*.



algorithm	mut.	n	nonce	size	files

ppsw13963.ppsw.rug.nl

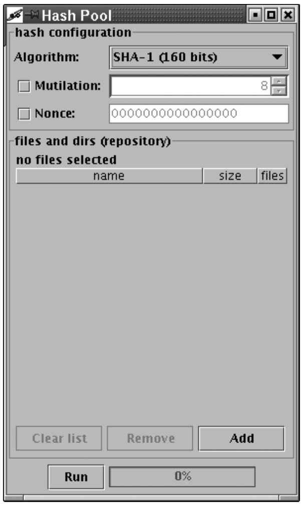
name	port
Wouter	4444

in	up	local name	remote name	remote host
☐	✓	Kathy	Wouter	localhost.lo...
✓	✓	Wouter	Kathy	localhost.lo...

FIGURE D.10:
Main application window, with connections.

[6] The details shown in the connections pane are as follows:

- The columns 'local name', 'remote name' and 'remote host' are pretty self-explanatory.
- The column 'in' shows whether it is an incoming connection (you have set up a listener/meeting place and somebody went there), or whether it is an outgoing connection (you went to a meeting place that somebody else has set up). Checked means it is an incoming connection, unchecked means it is an outgoing connection.
- The column 'up' is checked if the connection is up, that is, it has not been terminated in some way or another.



Hash Pool

hash configuration

Algorithm: SHA-1 (160 bits)

☐ Mutilation: 8

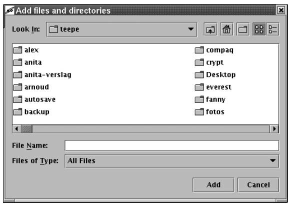
☐ Nonce: 0000000000000000

files and dirs (repository)

no files selected

name	size	files

FIGURE D.11: A new hash pool window.



Add files and directories

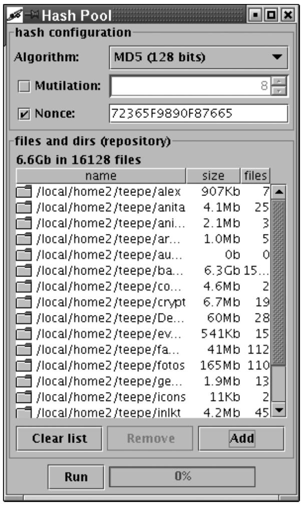
Look In: teepe

alex	compaq
anita	crypt
anita-verslag	Desktop
arnoud	everest
autosave	fanny
backup	fotos

File Name:

Files of Type: All Files

FIGURE D.12: Adding files to a hash pool.



Hash Pool

hash configuration

Algorithm: MD5 (128 bits)

☐ Mutilation: 8

✓ Nonce: 72365F9890F87665

files and dirs (repository)

6.6Gb in 16128 files

name	size	files
/local/home2/teepe/alex	907Kb	7
/local/home2/teepe/anita	4.1Mb	25
/local/home2/teepe/ani...	2.1Mb	3
/local/home2/teepe/ar...	1.0Mb	5
/local/home2/teepe/au...	0b	0
/local/home2/teepe/ba...	6.3Gb	15...
/local/home2/teepe/co...	4.6Mb	2
/local/home2/teepe/crypt	6.7Mb	19
/local/home2/teepe/De...	60Mb	28
/local/home2/teepe/ev...	541Kb	15
/local/home2/teepe/fa...	41Mb	112
/local/home2/teepe/fotos	165Mb	110
/local/home2/teepe/ge...	1.9Mb	13
/local/home2/teepe/icons	11Kb	2
/local/home2/teepe/inlkt	4.2Mb	45

FIGURE D.13: A hash pool with files added.

D.2 Managing Hash Pools

The protocol uses a hash value (h_1) to refer to the file of which knowledge is to be proven. When you initiate the protocol, you can compute the hash value as easy as that. If, however, you are on the responding side of the protocol, you have to find out what file the hash value h_1 refers to. That is what hash pools are for. A hash pool is a collection of precomputed mappings from hash value to file and vice versa.

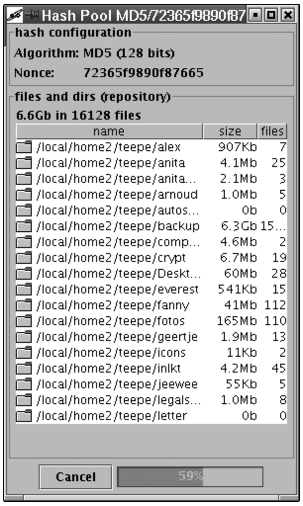
A hash pool has some settings, which determine the exact way the hashes are computed from the files. These settings are:

Algorithm The hash algorithm used to compute the hash. Currently MD5, SHA-1, SHA-256, SHA-384 and SHA-512 are available.

Mutilation All hash algorithms generate a value of a certain number of bits. If we want to stress-test the protocol for what happens if we have (for example) collisions, we can easily enforce collisions by trimming down the hash values to a limited number of bits.

Obviously, in any production environment, you don't want to mutilate your hash values.

Nonce If the protocol uses a nonce (and thus no encryption), a nonce must be used in the process of computing the hash values. In the prototype, nonces are 64 bits long, but they could be of any sufficiently large size.



hash configuration

Algorithm: MD5 (128 bits)

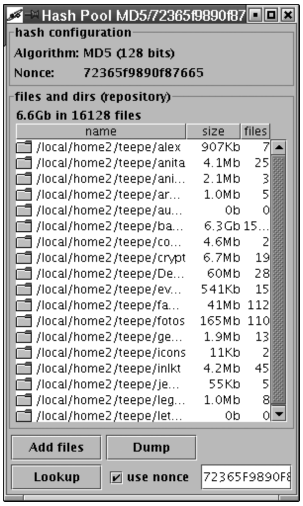
Nonce: 72365f9890f87665

files and dirs (repository)

6.6Gb in 16128 files

name	size	files
/local/home2/teepe/alex	907Kb	7
/local/home2/teepe/anita	4.1Mb	25
/local/home2/teepe/anita...	2.1Mb	3
/local/home2/teepe/arnoud	1.0Mb	5
/local/home2/teepe/autos...	0b	0
/local/home2/teepe/backup	6.3Gb	15...
/local/home2/teepe/comp...	4.6Mb	2
/local/home2/teepe/crypt	6.7Mb	19
/local/home2/teepe/Deskt...	60Mb	28
/local/home2/teepe/everest	541Kb	15
/local/home2/teepe/fanny	41Mb	112
/local/home2/teepe/fotos	165Mb	110
/local/home2/teepe/geertje	1.9Mb	13
/local/home2/teepe/icons	11Kb	2
/local/home2/teepe/inlkt	4.2Mb	45
/local/home2/teepe/jeewee	55Kb	5
/local/home2/teepe/legals...	1.0Mb	8
/local/home2/teepe/letter	0b	0

FIGURE D.14: Computation of hash values.



hash configuration

Algorithm: MD5 (128 bits)

Nonce: 72365f9890f87665

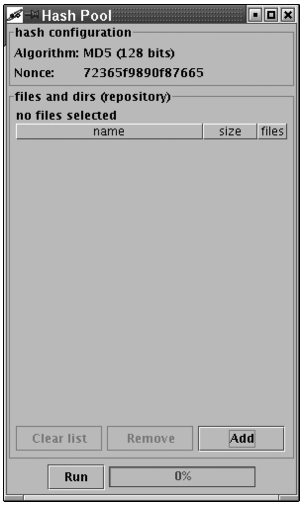
files and dirs (repository)

6.6Gb in 16128 files

name	size	files
/local/home2/teepe/alex	907Kb	7
/local/home2/teepe/anita	4.1Mb	25
/local/home2/teepe/ani...	2.1Mb	3
/local/home2/teepe/ar...	1.0Mb	5
/local/home2/teepe/au...	0b	0
/local/home2/teepe/ba...	6.3Gb	15...
/local/home2/teepe/co...	4.6Mb	2
/local/home2/teepe/crypt	6.7Mb	19
/local/home2/teepe/De...	60Mb	28
/local/home2/teepe/ev...	541Kb	15
/local/home2/teepe/fa...	41Mb	112
/local/home2/teepe/fotos	165Mb	110
/local/home2/teepe/ge...	1.9Mb	13
/local/home2/teepe/icons	11Kb	2
/local/home2/teepe/inlkt	4.2Mb	45
/local/home2/teepe/je...	55Kb	5
/local/home2/teepe/leg...	1.0Mb	8
/local/home2/teepe/let...	0b	0

☑ use nonce 72365F9890F8

FIGURE D.15: A ready hash pool.



hash configuration

Algorithm: MD5 (128 bits)

Nonce: 72365f9890f87665

files and dirs (repository)

no files selected

name	size	files

FIGURE D.16: Adding files to an existing hash pool.

To create a hash pool from scratch, press the 'Add' button in the 'hash pools' pane of the main application window (Figures D.1 and D.10). It will display a window in which the described settings can be set (Figure D.11).

Next to some settings, a hash pool obviously has a file list. Using the 'Add' button, files can be selected for inclusion in the hash pool (Figure D.12). You can also add directories. In that case, the program will recursively add all files in the directory (Figure D.13). Only after pressing the 'Run' button, the computation of the hash values starts (Figure D.14). Depending on the total volume of the files, this may take some time. A progress bar will give you a precise approximation of the progress.

After having done the precomputations, the hash pool window (Figure D.15) serves two main purposes:

1. Displaying the list of added files and directories;

2. Allowing you to add more files and directories to the hash pool.

The first purpose is obvious, the second almost as much. After pressing 'Add files' in a hash pool window which has finished computing, it will pop up a new window in which all settings are adopted from the existing hash pool, and you can add new files (Figure D.16). After pressing 'Run', the additional precomputations will be performed and the files will be added to the hash pool. Any protocols bound to this hash pool will be updated automatically.

If you run the protocol, you may indeed use the 'Add files' function quite a lot. That will be described in the Section D.3.3.

D.3 Running the Protocol

There are three configurations of the T-1 protocol, and for each of the configurations, a protocol with and a protocol without encryption exists. As an initiator of the protocol, one can choose to use encryption or use a nonce (two mutually exclusive options), and one can choose whether the initiator only proves, both proves and verifies, or only verifies (three mutually exclusive options). The combination of these choices gives a total of six possible protocols. All six blends of the protocol can be performed with the prototype.

When running a protocol, one has to keep track of several properties of the protocol, such as the file the protocol is about, what hash function is used, what nonce is used, and so on. All protocol windows display this information. When one initiates a protocol, one can also manipulate these attributes. Therefore, before explaining all protocol actions, we will briefly describe all protocol properties, and how they are shown in the protocol window.

In a protocol window, one can see from top to bottom the following things:

chat session The full communication history of the protocol. This displays exactly what bytes are sent along the TCP/IP connection regarding the current protocol run. Messages sent by you are prepended with 'SEND:', and messages sent by the remote side (the person you're communicating with) are prepended with 'RECEIVE:', for clarity.

Protocol configuration Here the security settings of the protocol are set or displayed. It is a superset of the options of a hash pool.

If you initiate the protocol, this is where you choose to either use a nonce, or use encryption. If you choose to use a nonce, you must give one, and if you choose to use encryption, you must choose an encryption algorithm and a key.

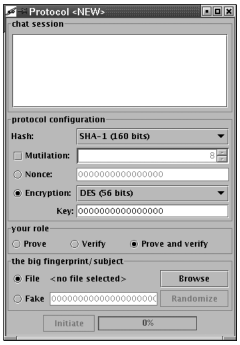
If you are not the initiator, this is where you must fill in either the nonce or the key, depending on the protocol type the initiator has chosen.

Your role Whether you are to prove, you are to verify or to prove and verify. As an initiator, you can set this parameter.

The big fingerprint/subject Here the subject of the protocol is identified. That is: the initiator chooses a file, h_1 will be computed. The initiator will see both the file name and h_1, the responder will only see h_1.

Here is also an option to 'fake', this will be explained later on. For now, you may ignore this option.

A button A multi purpose button. Its purpose depends on the state the protocol is in. While the protocol is being set up, the initiator of the protocol can fire up the protocol by pressing 'Initiate', upon which the communication will commence.


Protocol <NEW>

chat session

protocol configuration

Hash: SHA-1 (160 bits)

☐ Mutilation: 8

Nonce: 0000000000000000

[✓ Encryption:] DES (56 bits)

Key: 0000000000000000

your role

Prove Verify [✓ Prove and verify]

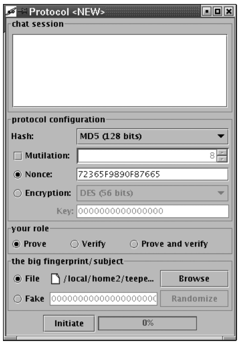
the big fingerprint/subject

[✓ File] <no file selected> Browse

Fake 00000000000000000000 Randomize

Initiate 0%

FIGURE D.17: A new protocol window for the initiator.


Protocol <NEW>

chat session

protocol configuration

Hash: MD5 (128 bits)

☐ Mutilation: 8

[✓ Nonce:] 72365F9890F87665

Encryption: DES (56 bits)

Key: 0000000000000000

your role

[✓ Prove] Verify Prove and verify

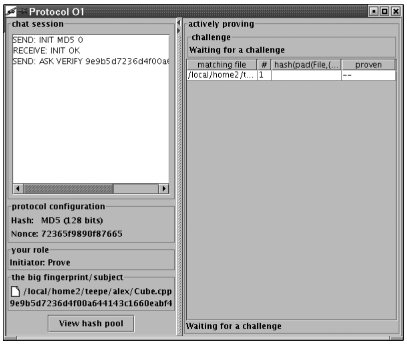
the big fingerprint/subject

[✓ File] /local/home2/teepe... Browse

Fake 00000000000000000000 Randomize

Initiate 0%

FIGURE D.18: A protocol window, configured by the initiator.


Protocol 01

chat session

```
SEND: INIT MD5 0
RECEIVE: INIT OK
SEND: ASK VERIFY 9e9b5d7236d4f00a
```

protocol configuration

Hash: MD5 (128 bits)

Nonce: 72365f9890f87665

your role

Initiator: Prove

the big fingerprint/subject

/local/home2/teepe/alex/Cube.cpp

9e9b5d7236d4f00a644143c1660eabf4

View hash pool

actively proving

challenge

Waiting for a challenge

matching file	#	hash(pad(File,{...	proven
/local/home2/t...	1		--

Waiting for a challenge

FIGURE D.19: A protocol window of the initiator for a protocol that has started.

The responder on the other hand, will see 'Commit ...' when the protocol has just started. Pressing the button, the responder can confirm the nonce or key he has entered.

After this, both sides will see here 'View hash pool'. Each protocol run has an associated hash pool, which can be inspected and extended using this button. It will display the corresponding hash pool window.

The association between hash pool and protocol run is not exclusive: one hash pool may be bound to more than one protocol run.

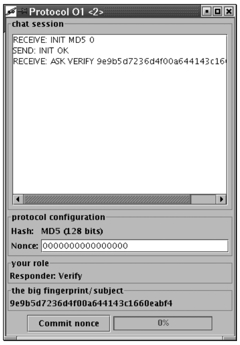
All these options may seem somewhat dazzling, but that will wear away quickly. In the next section, hopefully most questions will vanish.

D.3.1 Initiating a Protocol

Remember, we had set up a connection from Kathy to Wouter. We also had made a hash pool with the MD5 algorithm and the nonce 72365F9890F87665. Kathy (e.g. you) may now press the 'Add' button in the connection window (Figure D.7), and in the window that pops up (Figure D.17) select the hash algorithm, state that she wants to use a nonce, fill in the nonce, say she wants to prove (only), and use the 'Browse' button to select a file (it will look like Figure D.18).

Finally she presses 'Initiate' and the protocol is fired up. Along firing up the protocol, the window is enlarged (Figure D.19). Don't worry, we'll explain what you'll see in the added part later on, in step 3.3.

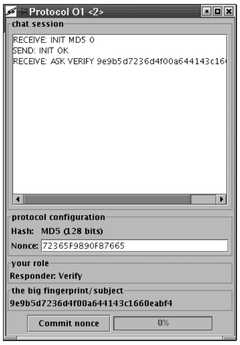
In case the responding side does not know or support the chosen hash and encryption algorithms, you will be informed so and may choose to try again using other algorithms. (Java 1.4 only supports MD5 and SHA-1, Java 1.5 also supports SHA-256, SHA-384 and SHA-512).



```
RECEIVE: INIT MD5 0
SEND: INIT OK
RECEIVE: ASK VERIFY 9e9b5d7236d4f00a644143c16
```

Hash: MD5 (128 bits)
Nonce: 0000000000000000
Responder: Verify
9e9b5d7236d4f00a644143c1660eabf4

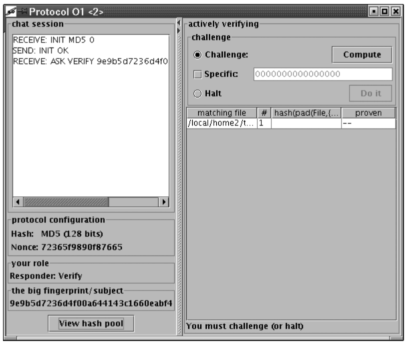
FIGURE D.20: A new protocol window for the responder.



```
RECEIVE: INIT MD5 0
SEND: INIT OK
RECEIVE: ASK VERIFY 9e9b5d7236d4f00a644143c16
```

Hash: MD5 (128 bits)
Nonce: 72365F9890F87665
Responder: Verify
9e9b5d7236d4f00a644143c1660eabf4

FIGURE D.21: The responder has filled in the nonce.



```
RECEIVE: INIT MD5 0
SEND: INIT OK
RECEIVE: ASK VERIFY 9e9b5d7236d4f0
```

Hash: MD5 (128 bits)
Nonce: 72365f9890f87665
Responder: Verify
9e9b5d7236d4f00a644143c1660eabf4

Specific: 0000000000000000

matching file	#	hash(pad(File,{...	proven
/local/home2/t...	1		--

You must challenge (or halt)

FIGURE D.22: The responder has committed the nonce.

D.3.2 Responding to a Protocol

Okay, Kathy has just initiated a protocol to Wouter. This has the result that at Wouter's side, a window has popped (Figure D.20). All protocol attributes are shown, except a crucial part. This crucial part is the nonce, if a nonce is used, and if encryption is used, the key. The responder should provide this information.

When a nonce is used, the nonce influences which hash pool is used. Filling in the wrong nonce will result in not finding the file corresponding to the received hash value.

When encryption is used, filling in the wrong key will result in not being able to decrypt the received hash value. This will be detected, and you will be prompted to try another key instead.

For now, Wouter should somehow know the nonce 72365F9890F87665 and fill it in (Figure D.21), and commit it. After committing, the window is enlarged (Figure D.22), similarly to the enlargement on the initiator's side.

D.3.3 A Side Note on Hash Pools

After initiating a protocol or committing the key or nonce to a protocol, the prototype automatically finds out what hash pool settings should be used for the current protocol. If such a hash pool already exists, it will take it. If such a hash pool does not yet exist, it will create it on the fly, with no files added yet.

Then, for the initiator, the hash value of the chosen file is computed and added to the hash pool, or looked up if it already was in the hash pool.

Strictly spoken, the initiator does not need a hash pool, only the responder does. There may however be situations in which it is desirable for the initiator to maintain a hash pool as well. (For example, if the initiator will be a responder in another protocol run, keeping a hash pool will save him computations.)

The responder will use the hash pool to try to figure out what file the initiator is talking about. If the responder already has done precomputations (i.e., created the hash pool and filled it with files), this will be inferred automagically. The responder may however always, during execution of the protocol, add files to the hash pool. This means that if the initiator has not done any precomputations, he is free to do these computations right on the spot, while the protocol has already started.

D.3.4 Challenging

There is another side note to make before we get to making a challenge. The screenshots and explanations you see and read here investigate the case where the initiator is the proving side, and the responder is the verifying side. If the initiator has chosen another option in the 'role' field (Figure D.17), this would all be different. From here on we talk about provers and verifiers. It is just a mere coincidence that the prover also is the initiator, and nothing more.

Now, Wouter should verify Kathy's possession of the file, so he should go and do this. First step would be to look in the hash pool for a matching file, and if no file was found, Wouter could try and add more files to the hash pool to see whether he finds the matching file. In case of this example, we have already done precomputations, and so Wouter has an educated guess which file it is.

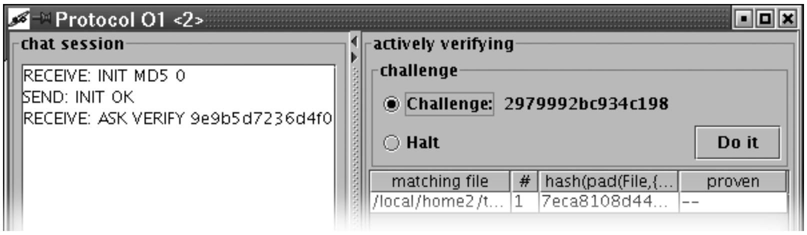
To verify Kathy's possession of the file, Wouter has to challenge Kathy. By pressing 'Compute' (Figure D.22), a challenge will be generated such that it discriminates within $I\star$. In fact, this is just making up some random challenge and test that it does indeed discriminate within $I\star$, if it doesn't, it just tries again with another random challenge.[7]

You may also choose the challenge yourself. In that case check the 'Specific' box, and fill in a challenge (Figure D.22). Then press 'Compute'. This option should not be used in production environments. It allows to test and verify that nobody can perform a man-in-the-middle attack.

Currently, all challenges are 64 bits long. This is not due to any limitation in the protocol, but just because we wanted to limit the number of options to the user. (There are enough options to choose from already.) The size of the challenge should be big enough to make it infeasible for an adversary to try all possible challenges.

You may remember from Chapter 9, that if the verifier is also the responder, he may also halt the protocol. This should typically be done when the responder has no matching file or does not wish to prove possession of the matching file. Therefore, you see that there actually is the choice to either challenge, or to halt. To confirm the choice, press 'Do it' (Figure D.23). After this confirmation the other side will be informed accordingly. For demonstration purposes, let Wouter actually challenge Kathy, and press 'Do it' (it will look like Figure D.24).

[7] Little exercise: Imagine what would happen if we would have thousands of files in our hash pool and would be trimming our hash values to only 3 bits.

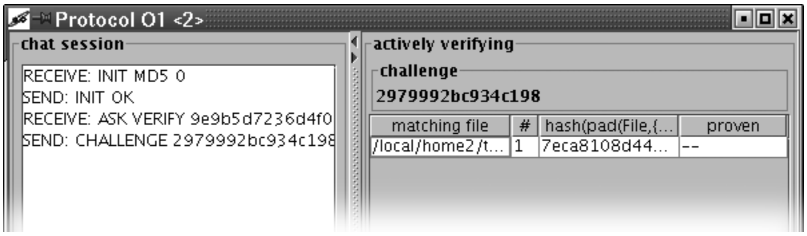


```
RECEIVE: INIT MD5 0
SEND: INIT OK
RECEIVE: ASK VERIFY 9e9b5d7236d4f0
```

(●) Challenge: 2979992bc934c198
( ) Halt

matching file	#	hash(pad(File,{…	proven
/local/home2/t…	1	7eca8108d44…	--

FIGURE D.23: The verifier chooses whether he will halt the protocol. For the chosen challenge, the required hash value h_2 has been computed.

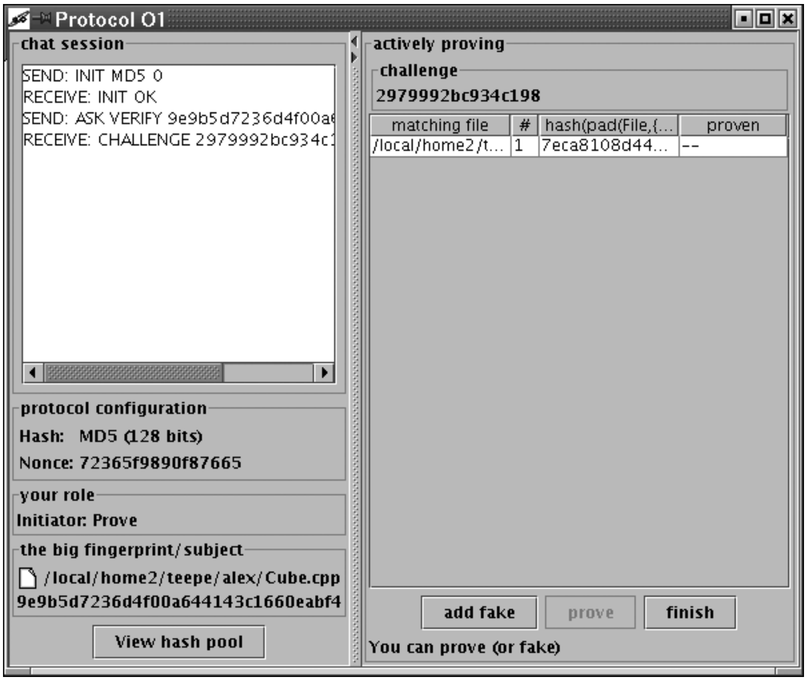


```
RECEIVE: INIT MD5 0
SEND: INIT OK
RECEIVE: ASK VERIFY 9e9b5d7236d4f0
SEND: CHALLENGE 2979992bc934c198
```

2979992bc934c198

matching file	#	hash(pad(File,{…	proven
/local/home2/t…	1	7eca8108d44…	--

FIGURE D.24: The verifier has challenged the prover.



```
SEND: INIT MD5 0
RECEIVE: INIT OK
SEND: ASK VERIFY 9e9b5d7236d4f00a
RECEIVE: CHALLENGE 2979992bc934c
```

2979992bc934c198

matching file	#	hash(pad(File,{…	proven
/local/home2/t…	1	7eca8108d44…	--

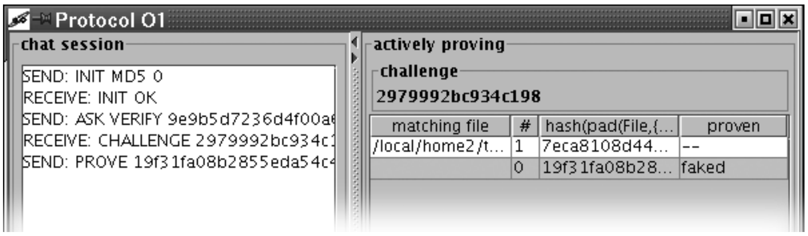
protocol configuration
Hash: MD5 (128 bits)
Nonce: 72365f9890f87665

your role
Initiator: Prove

the big fingerprint/subject
/local/home2/teepe/alex/Cube.cpp
9e9b5d7236d4f00a644143c1660eabf4

You can prove (or fake)

FIGURE D.25: The prover has received a challenge. With this challenge, the required hash value h_2 has been computed.


Protocol 01

chat session

```
SEND: INIT MD5 0
RECEIVE: INIT OK
SEND: ASK VERIFY 9e9b5d7236d4f00a
RECEIVE: CHALLENGE 2979992bc934c1
SEND: PROVE 19f31fa08b2855eda54c4
```

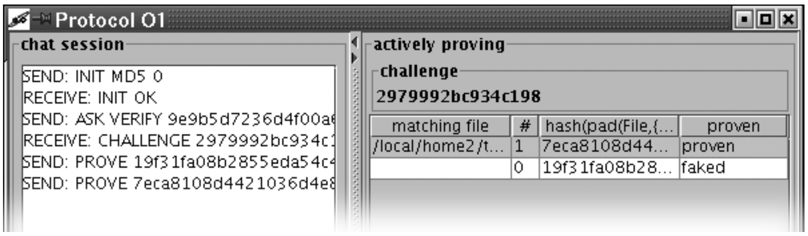
actively proving

challenge

2979992bc934c198

matching file	#	hash(pad(File,{...	proven
/local/home2/t...	1	7eca8108d44...	--
	0	19f31fa08b28...	faked

FIGURE D.26: The prover sends some fake hash value h_2.


Protocol 01

chat session

```
SEND: INIT MD5 0
RECEIVE: INIT OK
SEND: ASK VERIFY 9e9b5d7236d4f00a
RECEIVE: CHALLENGE 2979992bc934c1
SEND: PROVE 19f31fa08b2855eda54c4
SEND: PROVE 7eca8108d4421036d4e8
```

actively proving

challenge

2979992bc934c198

matching file	#	hash(pad(File,{...	proven
/local/home2/t...	1	7eca8108d44...	proven
	0	19f31fa08b28...	faked

FIGURE D.27: The prover sends a genuine hash value h_2.

D.3.5 Proving

When the prover has received a challenge, the main protocol window (which looked like Figure D.19), will have the challenge filled in, and the value h_2 computed (Figure D.25).

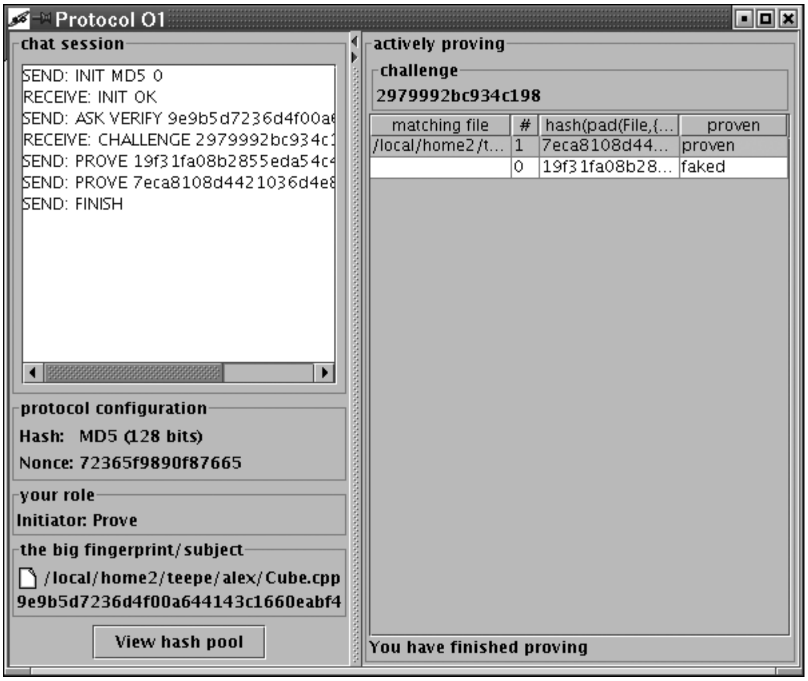
The prover may send any number of h_2 hash values, and should then inform the verifier that no more hash values will be sent.

The h_2 hash values the prover sends would in an optimal case always be truthful. However, we want to demonstrate that sending untruthful h_2 hash values will not result in the verifier becoming convinced. Therefore, the prover has two options for sending a h_2 hash value:

1. Just make up some untruthful h_2 hash value, and send it. To do this, the prover presses the 'add fake' button, which will add a fake file to $I\star$ with a random h_2 hash value. This 'hash value' may be edited. After this, select the fake file and press 'prove' (Figure D.26).

2. Choose a matching file from $I\star$, and sent the corresponding hash value h_2 (is has already been computed). To do this, the prover selects one or more files from the list, and presses the 'prove' button (Figure D.27).

When the prover has performed actions as he sees fit, he can inform the verifier that he is finished. This is done by pressing the 'finish' button (it will look like Figure D.28).



```
SEND: INIT MD5 0
RECEIVE: INIT OK
SEND: ASK VERIFY 9e9b5d7236d4f00a
RECEIVE: CHALLENGE 2979992bc934c1
SEND: PROVE 19f31fa08b2855eda54c
SEND: PROVE 7eca8108d4421036d4e
SEND: FINISH
```

Hash: MD5 (128 bits)
Nonce: 72365f9890f87665

Initiator: Prove

/local/home2/teepe/alex/Cube.cpp
9e9b5d7236d4f00a644143c1660eabf4

2979992bc934c198

matching file	#	hash(pad(File,{...	proven
/local/home2/t...	1	7eca8108d44...	proven
	0	19f31fa08b28...	faked

You have finished proving

FIGURE D.28: The prover hash halted the protocol (no more proofs will follow).

D.3.6 Verifying

After the verifier has challenged the prover, he is left with the task of the actual verification of the h_2 messages sent by the prover.

The prover can send a number of h_2 hash values, and then inform that no more h_2 hash values will follow. Each incoming h_2 hash value is looked up in the set $I\star$. If there is no matching file, there are two possible interpretations:

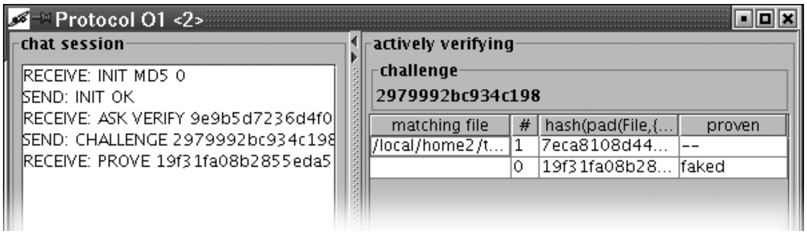
1. The prover has a file which gives the same h_1 hash value as the file the verifier has. This is increasingly unlikely as the hash size gets bigger. With a hash size of 128 bits, this is negligibly unlikely. (Assumed that the used hash function is a sufficiently good one.)

2. The prover has just made up some h_2 hash value.

Since the first option is so unlikely, if a hash value h_2 is received with no matching file in $I\star$, it will be marked as ‘faked’ (Figure D.29).

If a matching file is found, this file will be marked as ‘proven’, and this will be shown to the verifier (Figure D.30).

Finally, when the prover informs the verifier that he is finished, the verifier can finalize his interpretation of the received hashes. If the file was marked as ‘proven’, the prover both possessed the file and wanted to prove it to the

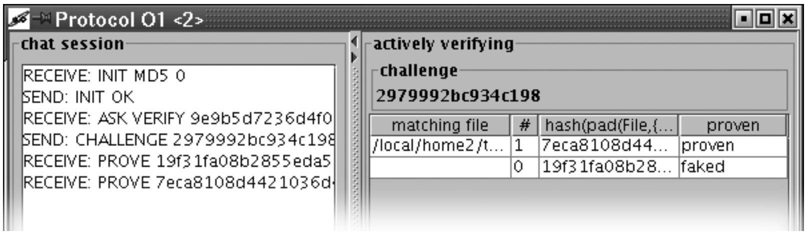


```
RECEIVE: INIT MD5 0
SEND: INIT OK
RECEIVE: ASK VERIFY 9e9b5d7236d4f0
SEND: CHALLENGE 2979992bc934c198
RECEIVE: PROVE 19f31fa08b2855eda5
```

challenge: 2979992bc934c198

matching file	#	hash(pad(File,{...	proven
/local/home2/t...	1	7eca8108d44...	--
	0	19f31fa08b28...	faked

FIGURE D.29: The verifier receives an unexpected value of h_2. It is marked as 'fake'.

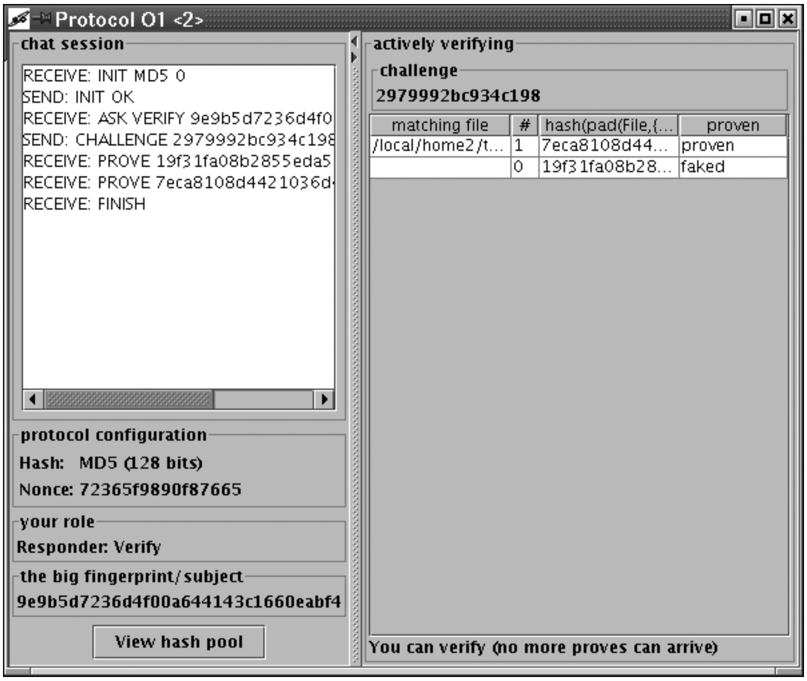


```
RECEIVE: INIT MD5 0
SEND: INIT OK
RECEIVE: ASK VERIFY 9e9b5d7236d4f0
SEND: CHALLENGE 2979992bc934c198
RECEIVE: PROVE 19f31fa08b2855eda5
RECEIVE: PROVE 7eca8108d4421036d
```

challenge: 2979992bc934c198

matching file	#	hash(pad(File,{...	proven
/local/home2/t...	1	7eca8108d44...	proven
	0	19f31fa08b28...	faked

FIGURE D.30: The verifier receives the h_2 he expected. The corresponding file is marked as 'proven'.



```
RECEIVE: INIT MD5 0
SEND: INIT OK
RECEIVE: ASK VERIFY 9e9b5d7236d4f0
SEND: CHALLENGE 2979992bc934c198
RECEIVE: PROVE 19f31fa08b2855eda5
RECEIVE: PROVE 7eca8108d4421036d
RECEIVE: FINISH
```

protocol configuration
Hash: MD5 (128 bits)
Nonce: 72365f9890f87665

your role
Responder: Verify

the big fingerprint/subject
9e9b5d7236d4f00a644143c1660eabf4

challenge: 2979992bc934c198

matching file	#	hash(pad(File,{...	proven
/local/home2/t...	1	7eca8108d44...	proven
	0	19f31fa08b28...	faked

You can verify (no more proves can arrive)

FIGURE D.31: The verifier has been informed that the prover has halted the protocol. (No more proofs will follow)

verifier. If no such mark has been made, the prover either did not possess the file, or did possess the file but did now want to prove this to the verifier.

D.3.7 Faking

The prototype can demonstrate that the T-1 protocol is secure in the malicious adversary model. The prototype facilitates all kinds of malicious behavior:

h_1 The initial hash value h_1 can be set manually. In the initialization of the protocol (Section D.3.1), instead of selecting a specific file to run the protocol with, one can choose 'Fake', and fill in the h_1 value directly. The button 'Randomize' is at ones disposal to help you make up some random value for h_1.[8]

C The challenge the verifier sends to the prover, should be made up at random, which is what the prototype can do. However, a malicious player may want to challenge in a specific way. While challenging (Section D.3.4), one can check the 'Specific' box, in which case the malicious player can fill in the challenge he wants to use.[9]

h_2 The response h_2 hash value the prover sends, can also be faked. This has been demonstrated in Sections D.3.5 and D.3.6.

Using these features, a malicious adversary can perform actions which are syntactically correct, but not intended. This kind of actions will never result in a non-malicious player being misled. That is, a non-malicious player will never be sneaked into believing that a malicious player possesses a file while he does not possess the file.

D.4 Closing

After having played around, a moment in time might arrive in which a user of the prototype decides to temporarily stop playing around with the software. Specifically for those users, the 'Exit' action is implemented. It is found in the 'File' menu of the main application window.

Exiting the program will result in all information in the hash pools being lost. If at a later moment one wants to rerun the protocols, one will need to rebuild the hash pools from scratch.

In case one only wants to stop communicating with some other player, one can disconnect the other player by pressing the 'Disconnect' button in the connection window (Figure D.7). All non-terminated protocols using the connection will obviously stop.

[8] Using this feature, one can try to claim possession of a file which one does not have. The protocol will however never let you successfully prove you do indeed possess the claimed file.

[9] Using this function, the malicious player might try to perform a man-in-the-middle attack. He will not succeed, because he will need to present a hash value h_2 with the given challenge and also a name, and such a hash will never be computed by another trustworthy user.

Appendix E

Notation

E.1 Symbols

$\mid\!\sim$	once conveyed (GNY logic, BAN logic)
$\vdash$	derivable within a logic
$\models$	observable within a model
$\mid\!\equiv$	believes (GNY logic, BAN logic)
$\overset{\cdot}{\mapsto}$	public key (GNY logic, BAN logic)
$\overset{\cdot}{\leftrightarrow}$	shared secret (GNY logic, BAN logic)
$\sharp(\cdot)$	fresh (GNY logic, BAN logic)
$\phi(\cdot)$	recognizable (GNY logic)
$*$	not-originated-here (GNY logic)
$\lhd$	is told (GNY logic, BAN logic)
$\ni$	possesses (GNY logic)
$\odot$	combining function
	(used in the randomize-then-combine paradigm)
$\oplus$	bitwise exclusive or
$\mid \cdot \mid$	the cardinality (number of elements) of a set
$\{\cdot\}$	a set
$\{\cdot\}_{\cdot}$	encryption;
	$\{M\}_K$ denotes the message M encrypted under symmetric key K;
	$\{M\}_{+K}$ denotes the message M encrypted under public key $+K$;
	$\{M\}_{-K}$ denotes the message M signed under private key $-K$

E.2 Letters

Ω	the domain of all possible secrets
Φ	the compressed domain of all possible secrets (Section 8.6)
A	the principal Alice
$\mathscr{B}$	the set of beliefs ($\models$) of a principal (BAN logic)
B	the principal Bob
c	a constant factor in a formula
C	either: • the principal Cecil (trustworthy) or Charlie (untrustworthy) • a challenge (a message or bit string constructed for the sake of being unpredictable)
D	a designator
E	the principal Eve (the evil eavesdropper)
ϵ	the empty string
ε	an error margin
$f(\cdot)$	a compression function (used in the Merkle-Damgård paradigm)
$g(\cdot)$	a randomizing function (used in the randomize-then-combine paradigm)
h	a hash value; in the T-1 and T-2 protocols: h_1 is the hash value that 'points at' a secret h_2 is the hash value that 'proves possession of' a secret
$H(\cdot)$	a non-keyed cryptographic hash function
I	a specific secret; I_Q is a secret of principal Q ($I_Q \in KB_Q$)
$I\star$	a set of specific secrets; $I_Q\star = \{I_Q \in KB_Q \| H(I_Q, N) = h_1\}$; or $I_Q\star = \{I_Q \in KB_Q \| H(I_Q) = h_1\}$
k	a security parameter (a measure of the strength of a cryptographic function)
K	a symmetric key
$-K$	a private key (the corresponding public key is $+K$); $-K_A$ denotes the private key of A (etc.)
$+K$	a public key (the corresponding private key is $-K$); $+K_A$ denotes the public key of A (etc.)
KB	a set of IB's possessed by an principal; KB_A denotes the set of IB's of A (etc.)
l	the length of a bit string
ln	the logarithm (with base 2)
M	a message
$\mathscr{M}$	the set of messages seen ($\triangleleft$) by a principal (BAN logic)
$MAC(\cdot,\cdot)$	a keyed cryptographic hash function (also called message authentication code, or MAC); $MAC(K, M)$ denotes message M hashed under key K
N	a nonce (a message constructed for the sake of being fresh)
p	a prefix of a hash value

	p_1 is the prefix of h_1
	p_2 is the prefix of h_2
P	the principal Peggy (the prover)
$\mathcal{P}(\cdot)$	the *power set*, the set of all possible subsets;
	$\overset{<\infty}{\mathcal{P}}$ is the set of all possible *finite* subsets;
	$\overset{<\infty}{\mathcal{P}}(\{0,1\}^*)$ is the set of all possible finite sets of finite bit strings
Q	a principal which may be either Alice, Bob, Cecil, Eve, Peggy or Victor
s	a global state in a protocol run; s_Q is a local state of principal Q (BAN logic)
S	either: • a secret (either private or shared) (GNY logic) • a set of binary strings (used in the T-2 protocol)
t	a timestamp or time interval (see Section 7.4)
V	the principal Victor (the verifier)