**SOFTWARE VERIFICATION RESEARCH CENTRE**

**THE UNIVERSITY OF QUEENSLAND**

**Queensland 4072**
**Australia**

**TECHNICAL REPORT**

**No. 01-22**

# A Survey of Verification Techniques for Security Protocols

**C. J. Fidge**

**July 2001**

**Phone: +61 7 3365 1003**
**Fax: +61 7 3365 1533**
**http://svrc.it.uq.edu.au**

**Note:** Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

# A Survey of Verification Techniques for Security Protocols

C. J. Fidge
Software Verification Research Centre
The University of Queensland
Queensland, Australia

## Abstract

Security protocols aim to allow secure electronic communication despite the potential presence of eavesdroppers. Guaranteeing their correctness is vital in many applications. This report briefly surveys the many formal specification and verification techniques proposed for describing and analysing security protocols.

## 1 Introduction

Security protocols aim to allow secure electronic communication despite the presence of eavesdroppers. Guaranteeing their correctness is vital in many applications such as defence or commerce. However, providing such guarantees has proven very difficult. The computing discipline of *formal methods* is devoted to unambiguous specification of system requirements, and mathematically-precise proofs of system properties. Many researchers have therefore attempted to use formal methods to verify the correctness of security protocols.

The results to date, however, have been disappointing. Several protocols which have been 'proven' correct have later been found vulnerable to attack [14]. We can recognise two reasons for this.

Firstly, specifying protocols and their desired properties in existing formalisms has been difficult. Formal methods can express functional behaviour and requirements, but they do not provide for typical security concepts such as 'confidentiality', 'authentication' or 'encryption'. Therefore, either a way must be found of representing security concepts using existing constructs, or the formalism must be extended with new features specifically for security protocol analysis.

Secondly, formal verification of system properties relies on having a complete description of the system under consideration. In the case of a security protocol, this means modelling both the communicating agents and the potential eavesdropper or attacker. However, the very nature of security protocols is that they are vulnerable to forms of attack that were not anticipated by their original designers. Such attacks will also be absent from the corresponding formal model and 'verification' then proves nothing about the protocol's security in such an environment. This is a fundamental hurdle. At best, formal verification can prove only that a protocol is resistant to all *anticipated* attacks—formal methods cannot provide 100% security

guarantees [34, §8].

Given these difficulties, an extraordinarily wide range of formalisms have been applied to verification of security protocols, with varying degrees of success. This report briefly surveys the more prominent approaches. Section 2 reviews some basic security protocol concepts and terminology. Section 3 surveys 'verification by analysis' techniques. These are based on developing a formal model of the protocol and then discovering the properties of that model. Section 4 surveys 'verification by construction' techniques. This complementary approach is based on first defining the desired protocol properties and then deriving (a model of) a security protocol that has such properties.

# 2 Background: Security Protocols

This section briefly reviews some basic security concepts and terminology.

## 2.1 A Protocol Hierarchy

Using cryptographic techniques [40] to achieve information security despite the possible presence of attackers relies on a layered set of activities.

(1) A **cryptographic algorithm** is needed for enciphering and deciphering data, given suitable keys [43][7]. Such an algorithm must have two important properties. *Indistinguishability of encryptions* is the inability of an observer to tell any meaningful difference between two encrypted messages or learn anything about the original plaintext messages [2][7]. *Non-malleability* is the inability of an intruder to respond meaningfully to a unique 'challenge' message [7]. Encryption algorithms serve to preserve confidentiality, guarantee authenticity, bind data, and generate random numbers [1, §5.1]. In the Secure Multipurpose Internet Mail Exchange (S/MIME) protocol, encrypted data is referred to as *enveloped* [22, Ch. 5]. In public key cryptography (see below), keys must be provided in pairs. A message encrypted with one such key must be decryptable using the other. Moreover, to support authentication (see below), both keys must be suitable for encrypting data [32].

(2) A **cryptographic protocol** defines the rules for exchanging information between communicating agents [43, Ch. 7]. It defines the sequence in which messages must be sent and the required contents of each message. The protocol assumes that communicating parties already have access to appropriate keys. In symmetric systems all keys must be kept secret, but asymmetric systems such as *public key cryptography* rely on both public and private keys. The *public key* is freely available and is used to encrypt data that is to be sent to a particular agent. The corresponding *private key* is known to the receiving agent only, and is used to decrypt the message received. The messages themselves may consist of some combination of data, addresses and keys, in either plaintext or encrypted forms. Cryptographic protocols are often defined in terms of *nonces*. This is a unique datum produced by a particular agent for use within a particular protocol instance [30], usually to guard against replay attacks [14] or to challenge the other party.

(3) A **key distribution protocol** determines how keys are maintained and distributed to agents that wish to communicate [43, Ch. 8][25]. (Of-

ten the key distribution and cryptographic protocols are considered together.) Key distribution can be divided into off and on-line algorithms [8, p. 100]. The most widely known example of a key distribution (and cryptographic) protocol is the Needham-Schroeder public-key protocol, which exists in both flawed [4][30][16] and corrected [17][30][34][23] versions. Another flawed protocol is Netscape's Secure Sockets Layer (SSL) protocol [1, p. 9] which was later upgraded to produce the Transport Layer Security (TLS) protocol [28, p. 136]. Another well-known example is the Otway-Rees shared-key protocol [44][34][14]. Some less commonly-cited ones include Yahalom's protocol [16, §9], the Wide-Mouthed Frog protocol [16, §7], and the obsolete Andrew Secure Remote Procedure Call Handshake protocol [16, §8]. There are also some standardised protocols such as CCITT's X.509 [16, §11][1, p. 10] and S/MIME [22, Ch. 5]. Boyd presents several variants of key distribution protocols [10].

(4) An **authentication protocol** is used to verify the identities of communicating agents [44]. (Originally key distribution was considered part of authentication, but it is now thought preferable to distinguish them [12].) Authentication protocols work via the use of *digitial signatures* [32][22, Ch. 5]. The sender of a message creates a signature by hashing the message to be sent with a publicly-known hashing function, and then encrypting the resulting hash value with the sender's private key. The receiver can then check that the signature and encrypted message both came from the same source by (a) decrypting both the signature and message with the sender's public key, (b) hashing the plaintext message, and (c) checking that the resulting hash value is the same as the decrypted signature. Some early examples of authentication protocols include the Needham-Schroeder-inspired Kerberos [45, §4][16, §6][1, p. 10], SPX [45, §4] and SELANE [45, §4] protocols.

(5) **Certification** allows trustworthy use of public keys [32]. Initially it requires a relationship to be established between agents and a *Certification Authority* by some trusted means outside the communications protocol, e.g., by physical exchange of disks containing keys. Digital signatures are then used so that the Certification Authority can sign public keys which are then published and can be used with confidence by agents that trust the Authority. Furthermore, chains of *cross-certificates* can be used to link Certification Authorities so that agents residing under different authorities can communicate [32].

(6) So far the protocols discussed have been aimed at providing protection from external attackers or eavesdroppers. **Non-repudiation** protocols are used when the communicating agents do not trust one another, and wish to be protected from future malicious behaviour by the other party [39], rather than an external entity. This may be the case, for instance, in commerce applications [46]. The protocols aim to provide irrefutable evidence that a particular communication took place. In the S/MIME protocol digital signatures are used to achieve *non-repudiation of origin* so that a sender cannot later deny having sent a message [22, Ch. 5]. Since a signature can be created only with the particular sender's private key, no other agent could have created and sent a signed message. The complementary requirement of *non-*

*repudiation of receipt* is used so that a receiver cannot deny having successfully received a message, and is achieved by having the receiver automatically return a signed acknowledgement [32, §4.4]. Examples include the Fair Exchange protocol [13], and the Zhou-Gollman protocol [39], among others [6].

Finally, we note that Abadi and Needham [1] describe numerous protocols, many of them with known flaws. Anderson and Needham also describe several flawed protocols [5].

## 2.2 Attacks

There are many forms of attack an intruder may make upon a security protocol. The most common ones are listed below [8][38].

**Eavesdropping** is when the intruder reads messages passing between sender and receiver, without their knowledge.

**Blocking** is when the intruder intercepts a message and prevents it reaching the intended recipient.

**Forging** is when the intruder sends messages to other agents which purport to be from someone else.

The success of such attacks depends on finding flaws in the protocol. Carlsen details the causes of numerous potential protocol flaws [19].

## 2.3 Properties

To resist such attacks, a security protocol must have certain characteristics or properties. A wide variety of desirable properties have been identified for security protocols, often described with confusing or inconstent terminology. In verification terminology, the properties to be proven for security protocols are usually *safety* ones [38], i.e., those that show that nothing 'bad' ever happens. (However, *liveness* properties, i.e., showing that something 'good' will eventually happen, may be a concern for non-repudiation protocols [39].)

The basic services of cryptography are confidentiality, integrity [22, Ch. 5] and authentication [9, p. 695]. **Confidentiality** requires that only the intended recipient can read an encrypted message [38]. To achieve this requires that, following key distribution, only the sender and receiver may know the appropriate key(s) [4, p. 4][36, §2.3]. This also implies that any newly created key must be 'fresh' [36, §2.3], and that an agent can possess a unique datum created by another only if the owner has previously agreed to share it [17, §3].

The **authentication** property requires that two agents that have just communicated using a certain key can be certain of the origin and destination of the message, respectively [4, p.4]. In other words, messages cannot be forged [38]. Whereas confidentiality requires that only a receiving agent possessing the right (public) key may read (unlock) a message, authentication requires that only a sending agent possessing the right (private) key can write (lock) a message [9, p. 695].

The **integrity** property requires that the content of a message is not changed by an attacker between the sender and recipient [22, Ch. 5]. It can be achieved using digital signatures, assuming that the hashing function is collision free [32].

Different properties are of concern for non-repudiation protocols [6, §2.1]. The issues here are non-repudiation of origin and receipt, and fairness [39].

**Non-repudiation of origin** is a safety property concerning the ability of the receiver of a message to convince a third party that it was indeed sent by its supposed originator [39, §4.2] [22, Ch. 5]. **Non-repudiation of receipt** is a liveness property concerning the ability of a sender to convince a third party that the message was received [32], or *will be* received, by its intended recipient [39, §4.2]. **Fairness** is a liveness property concerning the free availablity of messages and evidence between communicating parties [39, §4.3].

A particular protocol may provide only a particular subset of these properties. For instance, message integrity, origin authentication and non-repudiation of origin may all be provided as part of an *authentication service* [32, p. 291].

### 2.4 A Standard Notation

Traditionally, security protocols are described using a semi-formal 'standard notation' [20][4][1]. This consists of a list of communications. For example, the following is a standard notation description of a simple (but very insecure) cryptographic protocol [43, p. 159].

1. $A \rightarrow B : \{N_A\}_{K_{AB}}, A$
2. $B \rightarrow A : \{N_A\}_{K_{BA}}, B$

Each step describes a directed interaction between two named agents, and the message communicated. For instance, step 1 above says that agent $A$ sends a message to agent $B$. (A wildcard destination '$*$' can be used for broadcast messages [36].) The message itself contains two parts. The first is a unique nonce $N_A$ created by agent $A$. However, this is encrypted using a key $K_{AB}$ specifically intended for communication from $A$ to $B$. Data $D$ encrypted with key $K$ is denoted '$\{D\}_K$'. The second part of the message is $A$'s address or identity (in plaintext form)—the comma ',' denotes concatenation of message components. Encryption '$\{D\}_K$' and concatenation ',' can be combined in arbitrary ways to represent, for example, a message comprising several data items all encrypted at once with the same key, or a message comprising the concatenation of several items encrypted with different keys, or multiply encrypted data items, and so on.

Although widely used in the security community, a serious problem for verifying security protocols is that standard notation is not a sufficient starting point for formal analysis [4, §5]. In particular, standard notation: lacks strong typing; makes unstated assumptions; describes actions rather than goals; can express external communication actions only; cannot describe unexpected actions by intruders; and is oriented to describing message syntax rather than semantics [20]. Since it ignores actions internal to an agent, it is also impossible to fully describe the implementation of a protocol in standard notation [36, §2.2].

## 3 Proof by Analysis

The aim of applying verification techniques to the security domain is to prove that a particular protocol (Section 2.1) is resistant to a particular form of attack (Section 2.2). The challenges are that the way security protocols are typically defined (Section 2.4) differs markedly from traditional ways of specifying computations, and the properties that are intended to make protocols secure (Section 2.3) are diffi-

cult to state formally.

In this section we survey a number of attempts to adapt traditional verification techniques to the security domain. All such techniques begin by expressing the protocol in a formal notation, and then proving that the resulting model has desirable security properties. Such proofs can be performed in either of two complementary ways, logical proof (Section 3.1) or model checking (Section 3.2).

### 3.1 Logics

Mathematical proof offers the strongest possible guarantee that a (model of a) system behaves correctly. Unfortunately, such proofs are complex and intellectually challenging. Automated theorem provers can relieve much of the tedium of performing non-trivial proofs, but are difficult to use.

Logic based proofs for security protocols build on traditional mathematical reasoning, using models extended for representing security concepts. Logical notations typically offer a wide range of data structures and operators which makes them capable of accurately representing the messages transmitted in security protocols. Unfortunately, a particular weakness of logic-based models is that they have no in-built notions of communication concepts such as message sequencing and these must be explicitly expressed in the model.

The first step is to take a security protocol definition, usually given in terms of the standard notation (Section 2.4), and express it in mathematical logic. Given the well-established structure of standard notation descriptions, it is tempting to assume that this can be done automatically. Indeed, Carlsen [20] describes a tool for converting standard notation descriptions into the *CKT5* modal logic. It produces separate descriptions of the protocol from each of the agent's viewpoints—message transmission and reception are treated as separate events.

In this logic a protocol is described as a set of axioms, each representing the knowledge known to an agent, or an action performed by an agent, at a particular step in the protocol. Sequencing of protocol steps is achieved by explicit ordering of timestamps associated with each axiom. Predefined language constructs and axioms are provided for definition of a message ($msg$), sending ($S$) and reception ($R$) of messages, creation of nonces ($nonce$), and encryption ($e$). For example, consider the following fragment of Yahalom's protocol [20, p. 138].

1. $A \rightarrow B : A, N_A$
2. $B \rightarrow S : B, \{A, N_A, N_B\}_{K_{BS}}$

Firstly, agent $A$ sends a message to agent $B$ containing $A$'s address and a nonce created by $A$. Agent $B$ then sends a message to server agent $S$ containing its own address and an encrypted part consisting of the data sent by $A$, plus a new nonce created by $B$. The tool will translate this to a logical expression for each of the three agents. For instance, agent $B$'s role would be represented as follows [20, p. 140].

$$\begin{aligned}&\forall A, B, K_{BC}, N_A, N_B, t_1, t_2, t_3 \bullet \\ &\quad R_{B,t_1} msg(A.N_A) \wedge \\ &\quad nonce(N_B, t_2, B) \wedge \\ &\quad S_{B,t_3} msg(B.e(K_{BS}, A.N_A.N_B)) \wedge \\ &\quad t_1 < t_2 < t_3\end{aligned}$$

The first conjunct denotes agent $B$ receiving the first message. Full stops '.' are used to represent concatenation

NS1 $(t_1 \in tr \wedge A \neq B \wedge \mathbf{nonce}(N_A) \notin \mathbf{used}(t_1)) \Rightarrow$
$(\mathbf{says}(A, B, \mathbf{crypt}(\mathbf{pubk}(B), \langle \mathbf{nonce}(N_A), \mathbf{agent}(A) \rangle)) \# t_1) \in tr$

NS2 $(t_2 \in tr \wedge A \neq B \wedge \mathbf{nonce}(N_B) \notin \mathbf{used}(t_2) \wedge$
$\mathbf{says}(X, B, \mathbf{crypt}(\mathbf{pubk}(B), \langle \mathbf{nonce}(N_A), \mathbf{agent}(A) \rangle)) \in \mathbf{set}(t_2)) \Rightarrow$
$(\mathbf{says}(B, A, \mathbf{crypt}(\mathbf{pubk}(A), \langle \mathbf{nonce}(N_A), \mathbf{nonce}(N_B) \rangle)) \# t_2) \in tr$

Figure 1: Isabelle/HOL encoding of two protocol steps [34, Fig. 2].

of message components. The second conjunct represents the internal action (implicit in the standard notation version) of agent $B$ creating new nonce $N_B$. The third conjunct denotes $B$ sending the second message, and the fourth defines the order among the timestamped events. Similar expressions will be generated for agents $A$ and $S$. To support reasoning, the formal description must contain significantly more detail than the equivalent standard notation description. Formal proofs of properties then proceed in terms of the underlying modal logic and predefined axioms and operators.

Paulson [34] encodes cryptographic protocols in the Isabelle/HOL theorem prover by inductively defining possible traces exhibited by the communicating agents. A number of primitive datatypes are introduced for defining agents (**agent**), nonces (**nonce**), public keys (**pubk**), etc. Also a number of operators on traces are used to turn a trace into a set (**set**), and to tell whether a particular item has ever been used in a trace (**used**). Encryption is represented as an operator with arguments consisting of a key and a sequence of data items (**crypt**). Most significantly, a trace is represented as a sequence of communications events of the form '$\mathbf{says}(\alpha, \beta, \mu)$' which states that agent $\alpha$ sent message $\mu$ to agent $\beta$.

For instance, consider the following fragment of the Needham-Schroeder protocol [34, p. 25].

1. $A \rightarrow B : \{N_A, A\}_{K_{AB}}$
2. $B \rightarrow A : \{N_A, N_B\}_{K_{BA}}$

These two steps would be encoded in Paulson's logic as the two rules shown in Figure 1. (For readability we paraphrase the machine-readable concrete syntax used in Paulson's report.) Here $\langle e, f, \ldots \rangle$ denotes the sequence consisting of elements $e$, $f$, etc, and $e \# S$ is sequence $S$ with element $e$ prepended. Also, let $tr$ denote the set of traces defined by the protocol.

For instance, the antecedent in Rule NS1 identifies $t_1$ as a trace of the current protocol, notes that agents $A$ and $B$ are distinct, and requires that $N_A$ is a fresh nonce. The consequent then defines an extension to trace $t_1$ consisting of the event of agent $A$ sending the first encrypted message to agent $B$. Rule NS2 similarly defines the second protocol step—its antecedent requires that the first event already appears in trace $t_2$. However, the sender's name is shown as '$X$' because agent $B$ does not know who really sent the first message. Properties of this protocol can then be proven by using these rules to inductively reason about the contents of event traces.

A particularly powerful feature of Paulson's approach is that it aims to

predict unforeseen attacks by providing a rule which represents all possible behaviours of an intruder [34, §2.3]. As shown below, this is done by synthesising all possible fraudulent messages that could be generated using the observable information in the current trace [34, Fig. 2].

$$\begin{aligned}\text{Fake}\ \ & (t \in tr \wedge Y \neq I \wedge \\ & \quad X \in \mathbf{synth}(\mathbf{analz}(\mathbf{spies}(t)))) \\ & \Rightarrow \\ & (\mathbf{says}(I, Y, X) \# t) \in tr\end{aligned}$$

Function **spies** returns all the messages an intruder can see in a trace [34, §3.5]. Function **analz** returns all the plaintext data that can be extracted from a set of messages, including plaintext components, and the contents of encrypted components for which a corresponding key can be found in the set [34, §3.2]. Function **synth** then constructs all the messages an intruder could build from a set of message components. Thus the Fake rule states that if there is a message $X$ that intruder $I$ can compose from the observable data in trace $t$, then the intruder will send such a message to some other agent $Y$. A number of *tactics* (mechanical proof procedures) are provided to help explore possible attacks.

Both of the above approaches introduced an explicit encoding of event ordering, via timestamps and trace (history) variables, respectively. Indeed, how event ordering is represented is one of the most important characteristics of a logic for security, especially when reasoning about properties that are not stable [26] (a *stable* property is one that holds forever once it becomes true). Since the events of creating, sending and receiving information are separated in time, logics extended for security applications are usually expressed using 'past' operators [41] or trace variables.

A totally different approach is 'belief' logic, the most well-known example being the *authentication logic* (also called *BAN* logic after its originators—Burrows, Abadi and Needham) [16][14]. This is an entirely new modal logic which introduces a wide range of security objects, formulæ for representing security concepts, and new reasoning rules. In particular, belief logic avoids any explicit notion of time by reasoning with 'stable' properties only [16].

Belief logic introduces a large number of new constructs. For instance: statement $\alpha \models \phi$ says that agent $\alpha$ 'believes' formula $\phi$; $\alpha \lhd \phi$ says that $\alpha$ 'sees' $\phi$, i.e., has been sent this formula by another agent; and $\alpha \mid\sim \phi$ says that $\alpha$ 'once said' $\phi$, i.e., sent this formula in the past. Also: propositional formula $\alpha \overset{\kappa}{\longleftrightarrow} \beta$ says that agents $\alpha$ and $\beta$ may use shared key $\kappa$ to communicate; $\{\phi\}_\kappa$ denotes formula $\phi$ encrypted under key $\phi$; and conjunction of formulæ is denoted by a comma ',' [16].

There are also numerous special-purpose inference rules for reasoning about security models. For instance, the following rule (indirectly) defines an effect of unlocking an encrypted message [16].

**Rule 1**

$$\frac{\alpha \models \alpha \overset{\kappa}{\longleftrightarrow} \beta, \quad \alpha \lhd \{\phi\}_\kappa}{\alpha \models \beta \mid\sim \phi}$$

The antecedent says that agent $\alpha$ believes that $\kappa$ is a key shared with agent $\beta$ and that $\alpha$ has received formula $\phi$ encrypted with key $\kappa$. The consequent then allows us to conclude that $\alpha$ believes that agent $\beta$ has previously disclosed the (plaintext) formula $\phi$. Note that the logic assumes

that formulæ, rather than messages, are communicated between agents.

For example, consider the following two steps from the Otway-Rees authentication protocol [16, §4].

1. $A \rightarrow B : N_X, A, B, \{N_A, N_X, A, B\}_{K_{AS}}$
2. $B \rightarrow S : N_X, A, B, \{N_A, N_X, A, B\}_{K_{AS}}, \{N_B, N_X, A, B\}_{K_{BS}}$

In the first step agent $A$ sends agent $B$ a message containing a nonce $N_X$, the source and destination addresses, and an encrypted component containing the same information concatenated with a nonce $N_A$. The encrypted component was created using key $K_{AS}$, reserved for communication between agent $A$ and authentication server agent $S$. Agent $B$ cannot decrypt this part, but in the second message $B$ forwards it to server $S$ along with a copy of the plaintext part of $A$'s message and another encrypted component, this time containing a new nonce $N_B$.

Given the inadequacies of the standard security notation, reasoning in the authentication logic begins by first reexpressing the protocol in an 'idealised' form [16, §4].

1. $A \rightarrow B : \{N_A, N_C\}_{K_{AS}}$
2. $B \rightarrow S : \{N_A, N_C\}_{K_{AS}}, \{N_B, N_C\}_{K_{BS}}$

All plaintext parts of the messages are omitted because they can be easily forged [16, p. 10]. Also, the repeated, compound datum '$N_X, A, B$' has been replaced with a single new formula $N_C$, to simplify the model. (Although not obvious from the above syntax, it must be remembered that nonces represent *formulæ* in these expressions, with ',' as conjunction.)

Analysis of the protocol then proceeds by translating the idealised protocol steps, and any implicit assumptions about the protocol, into logical formulæ. For instance, one of the assumptions in this protocol is that $S$ initially knows that key $K_{AS}$ is already shared by agents $A$ and $S$.

$$S \models A \xleftrightarrow{K_{AS}} S$$

Also, when the second message above is sent, we know that agent $S$ 'sees' the encrypted message components from $B$.

$$S \triangleleft \{N_A, N_C\}_{K_{AS}}, \{N_B, N_C\}_{K_{BS}}$$

We can then use these two predicates as the antecedents in Rule 1 to conclude that agent $S$ believes that agent $A$ sent formulæ $N_A$ and $N_C$ sometime in the past [16, pp. 16–17].

$$S \models A \mid\sim (N_A, N_C)$$

As an entirely new logic specifically dedicated to security protocol proofs, the authentication logic has received a considerable amount of attention, much of it unfavourable. For instance, the translation of protocols into an idealised form has led to erroneous proofs that overlook the role of plaintext information in ensuring security [14]. Consequently, there have also been many attempts to improve on or extend belief logics [42].

Finally, we note that a number of other predicate logic-based formalisms have been used for specifying security protocols including VDM-SL [29], Z [9], and predicate logic with a special 'past' operator [41], although none of these have been used extensively. However, Boyd and Kearney recently showed how an animation tool could be used to find an attack on a non-repudiation protocol written in the

Z specification notation [13]. They expressed the protocol as a simple, highly-abstract state machine. Then a desired invariant property was expressed as a predicate on the system state. The animator was used to "methodically search the tree of possible operation" sequences, and check at each step that the invariant still held. This manually-guided search was successful at finding a particular sequence of operations which caused the protocol to fail. However, for non-trivial examples, there is clearly still a need to "automate the search for insecure states" [13].

### 3.2 Model Checking

Model checking offers a cheaper alternative to formal proof. A model checker is an automatic tool which explores the state space of a model in an attempt to find illegal states. For models with a small state space this search can be exhaustive and the results are then equivalent to a formal proof. More commonly, though, implementation limitations mean that the model cannot be analysed completely, and the results may then be inconclusive. Nevertheless, even an incomplete search may succeed in finding a 'bad' state, and can often do so more efficiently than via formal proof.

Notations suitable for model checking, such as process algebras and Petri Nets, typically have in-built features for event ordering and communication, and have already been used extensively for analysing communications protocols. Unfortunately, though, achieving security also relies on the contents of the messages transmitted. Model checking languages do not have extensive data structure support and ways must therefore be found to represent the complex messages used in security protocols in model checking notations.

Schneider [37] observes that security (safety) properties could be expressed in the Communicating Sequential Processes (CSP) process algebra, including confidentiality and authentication, thanks to the language's trace-based semantics. Since interprocess communication is synchronous in CSP, an explicit model of the communications network is introduced so that send and receive actions can be separated in time. His models abstract away from any internal details of the way agents behave; properties are expressed on the interactions of agents with the communications network. Given two network nodes $i$ and $j$, two unidirectional channels are assumed to link them, via the network. Compound action name '$trans.i.j.m$' denotes the action of node $i$ transmitting a message $m$ to node $j$, while action '$rec.i.j.m$' denotes reception of this message by $j$.

However, defining the contents of message $m$ raises a problem. A particular weakness of process algebras is that they offer little support for complex data structures such as those typically used in cryptographic protocols. To overcome this, Schneider proposes adding a recursive data structure to the language so that messages with arbitrary numbers of components and levels of encryption can be represented. He also suggests adding an 'information system' to the algebra so that formal reasoning can be undertaken about the data that can be extracted from messages. In this way, a generalised 'attacker' process could be defined as a way of searching for unanticipated attacks [37]. Schneider also explains how a non-repudiation protocol can be modelled and analysed in CSP,

$$
\begin{aligned}
&AGENT = \\
&\quad I\_running.A.B \rightarrow \\
&\quad comm!Msg1.A.B.Encrypt.key(B).N_A.A \rightarrow \\
&\quad comm.Msg2.B.A.Encrypt.key(A)?N'_A.N_B \rightarrow \\
&\qquad \vdots \\
&\quad I\_commit.A.B \rightarrow \\
&\qquad \vdots
\end{aligned}
$$

Figure 2: Two protocol steps performed by an agent in CSP [30, p. 150].

to ensure liveness properties such as non-repudiation of origin and receipt, and fairness [39]. However, more work needs to be done to complete Schneider's proposed approach and, in particular, to determine how it can be represented in the Failures Divergences Refinement (FDR) model checker used for CSP.

Lowe [30] similarly seeks to adapt CSP to modelling security protocols. He defines processes for each agent, including an intruder, and analyses their combination using the FDR model checker. Again the problem of how to represent security data structures arises. Lowe approaches this with an extension of Schneider's encoding. Not only are source and destination address fields represented in the action names, but also the contents of a message. For example, consider the following fragment of the Needham-Schroeder protocol [30, p. 149]. (It differs slightly from the one in Section 3.1 because the source and destination address fields are explicit.)

1. $A \rightarrow B : A, B, \{N_A, A\}_{K_{AB}}$
2. $B \rightarrow A : B, A, \{N_A, N_B\}_{K_{BA}}$

The behaviour of agent $A$ is represented in Lowe's approach by a CSP process that performs a series of atomic actions, as shown in Figure 2. In CSP, $x \rightarrow P$ is a process that performs action $x$ and then behaves like process $P$. Thus, process $AGENT$ in Figure 2 performs several atomic actions, one after the other, where the actions' names have been meaningfully divided into a number of fields separated by punctuation marks. By convention, CSP uses full stops '.' to break up compound names, and question '?' and exclamation marks '!' to denote input and output, respectively, although these notations are merely syntactic shorthands [27]. (The role of actions $I\_running$ and $I\_commit$ is explained below.) Here *comm* represents communication taking place over a 'standard' channel, distinct from those used by an attacker. Fields $Msg1$ and $Msg2$ act as counters to identify which step in the protocol the agent is currently performing. The remaining fields denote the data carried in the message, with the word '*Encrypt*' used to mark the start of encrypted fields. A similar model would be given for agent $B$.

The need to 'hardwire' all the fields of each message into the action name is awkward and seems inflexible, particularly when attempting to devise an intruder process with a wide range of possible behaviours. This is solved simply by duplicating each type of action for each combination of operands.

$$
\begin{aligned}
INTRUDER = & \\
& \vdots \\
& intercept.Msg1?A.B.Encrypt.K.N.A' \rightarrow \ldots \\
\Box \quad & intercept.Msg2?B.A.Encrypt.K.N.N' \rightarrow \ldots \\
\Box \quad & intercept.Msg3?A.B.Encrypt.K.N \rightarrow \ldots \\
& \vdots
\end{aligned}
$$

Figure 3: Part of an intruder process in CSP [30, p. 152].

The result is verbose but "reasonably uniform" [30, p. 152]. For instance, part of an intruder process that can intercept any message in this protocol using covert channel *intercept* is shown in Figure 3, where '$\Box$' is CSP's nondeterministic choice operator.

Model checking of the whole system is performed automatically by comparing the CSP specification with another specification that captures desirable properties only. This is the purpose of actions *I_running* and *I_commit* in Figure 2. They serve to announce what point in the overall protocol the 'initiator' agent has reached. Similar actions are inserted in the 'responder' agent, and the whole system is then compared with the following recursive CSP process which considers only the desirable behaviour that the initiator running is followed by the responder successfully committing to the completed secure communication [30, p. 153].

$$
\begin{aligned}
AI = & \, I\_running.A.B \rightarrow \\
& R\_commit.A.B \rightarrow AI
\end{aligned}
$$

Roscoe describes this same example in CSP as well, using the same protocol encoding, except that he illustrates the machine-readable concrete syntax used by the FDR model checker [35, §15.3].

Rather than trying to adapt an existing process algebra to the task, Abadi and Gordon define the *spi calculus*, a process algebra specifically designed for modelling cryptographic protocols [2][3]. It is based on the $\pi$-calculus, a particularly terse, but powerful process algebra. Unlike other process algebras, the $\pi$-calculus allows communications channels to be created dynamically and passed as data between processes. The spi-calculus seeks to capitalise on this by using the visible scope of channels as a way of modelling secure communications (although surprisingly little use is made of this feature in examples). The spi-calculus also extends the $\pi$-calculus with key-encryption primitives, and its specifications have a straightforward correspondence to standard notation specifications.

Operators already available for constructing communicating processes in the $\pi$-calculus include: $(\nu c)P$ which declares a fresh channel $c$ whose scope is process $P$; '$\overline{c}\langle d\rangle \,.\, P$' which sends datum $d$ on channel $c$ and subsequently behaves like process $P$ ($P$ can be omitted if there is no subsequent action [2, §2.2]); and '$c(v) \,.\, P$' which reads data from channel $c$ into variable $v$ and then behaves like $P$.

In the spi-calculus the $\nu$ operator is also used for creating fresh

keys. Several additional primitives are also introduced for symmetric [2, §3.1] and asymmetric [2, §5] key cryptography. These include the term '$\{D\}_K$' which denotes (shared-key) encryption of data $D$ with key $K$, as usual. Also operator 'case $L$ of $\{D\}_K$ in $P$' represents an attempt to decrypt term $L$ with key $K$, to extract data $D$. If the attempt is successful the operator behaves like process $P$, otherwise it deadlocks. (This draws attention to failed decryption attempts.)

Consider the following simplified version of the Wide-Mouthed Frog key distribution protocol [2, §3.2.2].

1. $A \rightarrow S : \{K_{AB}\}_{K_{AS}}$
2. $S \rightarrow B : \{K_{AB}\}_{K_{SB}}$
3. $A \rightarrow B : \{N_A\}_{K_{AB}}$

Here agent $A$ wants to send a secure message $N_A$ to agent $B$. To do so, $A$ first creates a new key $K_{AB}$ which it sends to trusted server $S$ who forwards it to agent $B$.

The behaviour of agent $A$ is expressed in the spi-calculus as follows [2, §3.2.2].

$$A = (\nu K_{AB})(\overline{C_{AS}}\langle\{K_{AB}\}_{K_{AS}}\rangle \,.\, \overline{C_{AB}}\langle\{N_A\}_{K_{AB}}\rangle)$$

Since communication in the $\pi$-calculus (and other process algebras) occurs through named channels, rather than naming the target process, it is assumed that channel $C_{AS}$ is used for communication from agent $A$ to agent $S$, $C_{SB}$ for communication from $S$ to $B$, and so on. The $\pi$-calculus' $\nu$ operator is used above to declare a new key $K_{AB}$ which is then sent in encrypted form on channel $C_{AS}$ (and thus out of the scope of the declaration!). This is followed by transmission of the message to agent $B$, encrypted with the new key.

Server agent $S$ is represented as follows [2, §3.2.2].

$$S = C_{AS}(x) \,.\, (\text{case } x \text{ of } \{y\}_{K_{AS}} \text{ in } \overline{C_{SB}}\langle\{y\}_{K_{SB}}\rangle)$$

After receiving data $x$ on channel $C_{AS}$, the process attempts to extract data $y$ using the appropriate key $K_{AS}$. If successful it then forwards $y$, encrypted with key $K_{SB}$, on channel $C_{SB}$. The system as a whole then consists of agents $A$, $B$ and $S$ combined within the scope of declarations for keys $K_{AS}$ and $K_{SB}$, which are used for communication between the standard agents and the server.

Proofs of protocol properties can then be performed by using the language semantics to show the equivalence between such a protocol description and another spi-calculus model that embodies the desired property. Thanks to its executable semantics, the spi-calculus appears well suited to automatic model checking, but no tool support exists for it yet [3, §8].

Just as the spi-calculus extended the previous $\pi$-calculus with cryptographic primitives, the Cryptographic Security Process Algebra (CryptoSPA) [24] similarly builds on the CCS process algebra. Its initial version, called the Security Process Algebra (SPA), extended CCS with CSP's hiding operator, and partitioned the visible actions into 'high and low level' ones [23]. Actions were compound objects containing all the fields of protocol messages. CryptoSPA then further extends this by the addition of message encryption and decryption operators [24].

CCS is a predecessor of the $\pi$-calculus and has many operators in common. In particular, input '$c(v) \,.\, P$' and output '$\overline{c}(d) \,.\, P$' actions are represented in the same way. Also, $P[x/y]$

$$A = \overline{C_{AS}}(A, \{(B, K_{AB})\}_{K_{AS}}) \,.\, \overline{C_{AB}}(\{N_A\}_{K_{AB}})$$
$$S = C_{AS}(u) \,.\, ([u \vdash_{snd} x]([(x, K_{AS}) \vdash_{dec} y]([y \vdash_{snd} z](\overline{C_{SB}}(\{(A, z)\}_{K_{SB}}) \,.\, S))))$$

Figure 4: Two protocol agents defined in CryptoSPA [24, §6].

denotes process $P$ with free occurrences of '$x$' replaced by '$y$' [31, §2.4].

To overcome the inherent weakness of process algebras for manipulating data, CryptoSPA adds data structures for expressing message components, and a logical 'inference system' for manipulating such data structures. For example, the rule

$$\{D\}_K,\ K^{-1} \quad \vdash_{dec} \quad D$$

models message decryption by saying that plaintext data $D$ can be derived from the combination of encrypted message $\{D\}_K$ and symmetric key $K^{-1}$ [24, Fig. 1]. Even simpler, rule

$$(p, q) \quad \vdash_{snd} \quad q$$

extracts the second element $q$ from a pair [24, Fig. 1].

A new operator then makes use of these rules to define processes whose behaviour is conditional on messages received. The process '$[(m_1, \ldots, m_n) \vdash_r x]P; Q$' behaves like either of two processes depending on whether inference rule $r$ can be applied to tuple $(m_1, \ldots, m_n)$ to extract some datum $d$. If the rule is successfully applied then the process behaves like process $P[d/x]$. Otherwise, it behaves like process $Q$. (In examples, $Q$ is frequently omitted—we *assume* this means that the overall process deadlocks if rule application fails.)

Consider the following variant of the Wide-Mouthed Frog protocol [24, §6].

1. $A \to S : A, \{B, K_{AB}\}_{K_{AS}}$
2. $S \to B : \{A, K_{AB}\}_{K_{SB}}$
3. $A \to B : \{N_A\}_{K_{AB}}$

The behaviour of agent $A$ is represented straightforwardly in CryptoSPA as two consecutive output actions as shown in Figure 4. Here $C_{AS}$ and $C_{AB}$ are implicitly assumed to be the channels reserved for communicating between the subscripted pairs of agents in the obvious way.

The behaviour of server agent $S$ in Figure 4 makes use of the inference system to extract data from a message [24, §6]. After receiving some message $u$ from agent $A$ on channel $C_{AS}$, the process performs three operations on it. The first extracts the second component, naming it $x$. The second attempts to decrypt $x$, using key $K_{AS}$, naming the result $y$. The third then extracts the second component from $y$, naming it $z$. Assuming the message received from $A$ has the anticipated format, component $z$ will be the key $K_{AB}$. Agent $S$ then forwards this to agent $B$, via channel $C_{SB}$, encrypting it together with $A$'s address using key $K_{SB}$.

There is also some tool support available for the original SPA formalism. A 'compiler' has been developed which allows protocols expressed in an abstract notation to be translated automatically to the SPA language [21]. Properties of the resulting protocol description can then be model-checked using a separate analysis tool. As in the CSP example shown in Figure 2, such properties are expressed

in terms of auxiliary actions such as '*R_running*' added to the model to make significant points in the computation externally visible [21, §6.4].

# 4 Proof by Construction

Rather than modelling an existing protocol and attempting to prove that it has a desirable property, proof by construction begins with a description of the property and attempts to derive a protocol that obeys it. Although semi-formal approaches to designing secure protocols have been suggested [15][11], our interest here is with formal development laws which guarantee that necessary properties are preserved at each step.

In the formal methods community such a process is known as *refinement* and is embodied in a number of refinement calculi. To date, however, only one of these, the B method, has been trialled extensively on security protocols. B is a relatively low level refinement model in which systems are described as state machines. However, it has been widely used in industry thanks to its extensive tool support.

System models in B are represented as *abstract machines*, consisting of state variables, invariants and operations on those variables. Operations are defined using programming language-like statements including assignment (**:=**), composition of simultaneous assignments (||), and conditional statements (**if**). Refinement in B is the process of translating an 'abstract' machine into a 'concrete' one, which may include adding new state variables and 'internal' operations. To prove the correctness of such a refinement it is necessary to devise a suitable *abstraction invariant* that links the abstract and concrete variables. It must then be shown that each operation in the concrete machine is a refinement of the corresponding operation in the abstract one. Typically this means that the concrete operation can exhibit fewer potential behaviours than the abstract one, and that it can be invoked in at least as many states as the abstract operation.

For example, Abrial [4] proposed an approach to developing security protocols using B by working backwards through each step of the protocol and incrementally adding detail to the formal model. Consider the following (flawed) version of the Needham-Schroeder protocol [4, §3].

1. $A \rightarrow S : A, B, N_A$
2. $S \rightarrow A : \{K_{AB}, B, N_A, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$
3. $A \rightarrow B : \{K_{AB}, A\}_{K_{BS}}$

The aim is to send key $K_{AB}$ from agent $A$ to agent $B$. However, to ensure that the key is fresh, it is actually generated by server agent $S$.

To begin the refinement, the desired behaviour of the protocol is first specified as an abstract B machine. Types $KEY$ and $AGENT$ are assumed to contain all possible keys and agents, respectively. Five variables are declared to represent the state of the machine.

$$
\begin{array}{l}
key_1 \subseteq KEY \\
knowA_1 \in key_1 \rightarrow AGENT \\
knowB_1 \in key_1 \rightarrow AGENT \\
believeA_1 \in key_1 \rightarrow AGENT \\
believeB_1 \in key_1 \rightarrow AGENT
\end{array}
$$

Here, the subscripted numbers are used to distinguish the phase of the refinement. Variable *key* represents the set of keys that have been used

previously, if any. The remaining variables are total functions from the keys used to agents. Agent $knowA(k)$ is the agent that created key $k$ (or, more accurately, the agent that invoked the server to create the key on its behalf). Agent $knowB(k)$ is the agent that received key $k$. Agent $believeA(k)$ is the agent that the creator of key $k$ intended to receive it. Agent $believeB(k)$ is the agent that the recipient of key $k$ believes sent it. If the protocol is successful at distributing a new key $k$, then $knowA(k)$ will equal $believeB(k)$ and $knowB(k)$ will equal $believeA(k)$.

To achieve this goal, the machine contains four operations, $Step\_1$ to $Step\_4$. The first three will represent the three steps in the protocol and the fourth defines the overall effect of the protocol. At this stage of the refinement, operations $Step\_1$ to $Step\_3$ are all null operations. All the work is done by operation $Step\_4$ whose body consists of the following parallel (simultaneous) assignments, provided that the key is fresh.

$$
\begin{array}{l}
Step\_4_1 = \\
\quad \vdots \\
\quad \textbf{if } K \notin key_1 \textbf{ then} \\
\quad\quad key_1 := key_1 \cup \{K\} \;||\; \\
\quad\quad knowA_1(K) := A \;||\; \\
\quad\quad knowB_1(K) := B \;||\; \\
\quad\quad believeA_1(K) := B \;||\; \\
\quad\quad believeB_1(K) := A \\
\quad \textbf{end}
\end{array}
$$

Here $K$ represents the particular key $K_{AB}$. Clearly, performing this operation will result in the desired authentication relationship among the variables described above.

The refinement is to then instantiate null operations $Step\_1$ to $Step\_3$ in such a way that they have the same effect as $Step\_4$, but do so by modelling the three steps of the protocol. There are three major data refinement steps, each of which changes both the system state and the four operations. The first refinement instantiates $Step\_3$, to introduce the last step in the protocol. Two of the existing variables are changed slightly [4, §6.1].

$$
\begin{array}{l}
knowB_2 \in key_2 \pfun AGENT \\
believeB_2 \in key_2 \pfun AGENT
\end{array}
$$

Now $knowB$ and $believeB$ are partial, rather than total, functions because information is no longer transferred atomically from agent $A$ to $B$. In the state between agent $A$ sending the third protocol message and agent $B$ receiving it, key $K_{AB}$ will belong to set $key$, but is not yet part of agent $B$'s local knowledge.

Additional variables are also introduced to the state, for modelling messages in transit, assuming given type $MSG$.

$$
\begin{array}{l}
msg_2 \subseteq MSG \\
menc_2 \in msg_2 \fun AGENT \\
mkey_2 \in msg_2 \psurj key_2 \\
magt_2 \in msg_2 \fun AGENT
\end{array}
$$

Set $msg$ contains those messages in transit. The remaining variables serve to represent the contents of the third message in the protocol, which has the form '$\{K_{AB}, A\}_{K_{BS}}$'. Agent $menc(m)$ denotes the agent playing the role of agent $B$ in message $m$. Key $mkey(m)$ denotes the encrypted key $K_{AB}$. Agent $magt(m)$ denotes agent $A$.

Null operation $Step\_3_1$ is then replaced with one that models sending

of the third protocol message.

$$
\begin{array}{l}
Step\_3_2 = \\
\quad \vdots \\
\quad \textbf{if } K \notin key_2 \wedge M \notin msg_2 \textbf{ then} \\
\quad\quad key_2 := key_2 \cup \{K\} \;||\; \\
\quad\quad knowA_2(K) := A \;||\; \\
\quad\quad believeA_2(K) := B \;||\; \\
\quad\quad msg_2 := msg_2 \cup \{M\} \;||\; \\
\quad\quad menc_2(M) := B \;||\; \\
\quad\quad mkey_2(M) := K \;||\; \\
\quad\quad magt_2(M) := A \\
\quad \textbf{end}
\end{array}
$$

Here $M$ represents the message $\{K_{AB}, A\}_{K_{BS}}$.

Comparing operation $Step\_3_2$ to operation $Step\_4_1$ above, we can see than variables $key$, $knowA$ and $believeA$ are updated as before, and new variables $msg$, $menc$, $mkey$ and $magt$ hold details of the message in transit. Therefore, operation $Step\_4_2$ merely needs to update variables $knowB$ and $believeB$ in the refined machine, using information from the 'message' variables.

$$
\begin{array}{l}
Step\_4_2 = \\
\quad \vdots \\
\quad knowB_2(mkey_2(M)) := \\
\quad\quad menc_2(M) \;||\; \\
\quad believeB_2(mkey_2(M)) := \\
\quad\quad magt_2(M)
\end{array}
$$

It then remains to formally prove that the combination of operations $Step\_3_2$ and $Step\_4_2$ achieves the same effect as operation $Step\_4_1$. Although this is reasonably obvious by inspection, the formal proof sketched by Abrial requires 11 pages [4]. The remaining two data refinements and their proofs are even larger since more operations are involved.

Bieber and Boulahia-Cuppens [8] explain how the B method can be used to develop security protocols by refinement in a more general way. (Sadly, their presentation tends to obscure, rather than highlight, their technical achievements.) They first model the state of a communications channel and the knowledge available to each agent [8, §2.2.1], and then the effect of simple unconstrained communication events [8, §2.2.2]. They then specify a perfect communications system as a set of *send* and *receive* operations [8, §3.1] and develop increasingly detailed refinements of it for sealed envelopes [8, §3.2], encryption with shared keys [8, §3.3], on-line key distribution [8, §4.2], and secure key distribution [8, §4.3].

For example, one of their simplest refinement steps introduces the following trivial protocol for sending message $M$ from agent $A$ to $B$ [8, §3.3.2].

1. $A \rightarrow B : \{M\}_{K_{AB}}$

This protocol is secure *assuming* that key $K_{AB}$ can be known only by agents $A$ and $B$. (This unrealistic assumption is removed in later refinement steps that introduce key distribution.)

To specify the desired confidentiality property of this protocol, Bieber and Boulahia-Cuppens define a B machine which models the transmission and reception of messages in *envelopes* that can be opened only by the sender and intended recipient [8, §3.2]. Their given types include the set of plaintext messages, $MSG$, the set of envelopes, $ENV$, and the set of agents, $AGENT$. The machine includes the following state variables.

$$
\begin{array}{l}
sentBy \in ENV \leftrightarrow AGENT \\
sentTo \in ENV \leftrightarrow AGENT \\
channel \in ENV \leftrightarrow (AGENT \times \\
\qquad\qquad\qquad AGENT) \\
content \in ENV \leftrightarrow MSG \\
opener \in ENV \leftrightarrow AGENT
\end{array}
$$

Relations *sentBy* and *sentTo* associate an envelope with its sender and address, respectively. Relation *channel* models the state of a communications channel which may contain envelopes with source and destination agent pairs [8, §2.2.1]. Relation *content* associates envelopes with the plaintext messages they contain. Relation *opener* associates envelopes with the agent, or agents, who are capable of opening the envelope [8, §3.2.1]. (Bieber and Boulahia-Cuppens misleadingly call this variable '*source*'.)

The machine includes operations which denote message transmission, reception, and potential behaviours of attackers. For instance, the *send* operation has the following behaviour for a message $M$, that is sent by agent $A$ to agent $B$, inside envelope $E$ [8, §3.2.2]. Let $f[S]$ denote the image of set $S$ through function $f$. The B statement '**any** $x$ **where** $P$ **then** $S$ **end**' nondeterministically chooses variables $x$ that satisfy predicate $P$ and then performs statement $S$.

**any**
  $E$
**where**
  $E \in ENV \wedge$
  $(E \mapsto M) \in content \wedge$
  $opener[\{E\}] = \{A, B\}$
**then**
  $sentBy := sentBy \cup \{E \mapsto A\} \;||$
  $sentTo := sentTo \cup \{E \mapsto B\} \;||$
  $channel := channel \cup$
    $\{E \mapsto (A, B)\}$
**end**

Thus envelope $E$ is associated with message $M$; the only agents allowed to open the envelope are $A$ and $B$; and the state is updated to record the sender and intended recipient of the envelope, and the fact that the envelope has been sent into the communications channel. The other operations are similar.

The aim of the refinement is to translate this specification into a model that uses a shared key to protect the envelope's contents. To do this, a given type $KEY$ is assumed and some additional state variables are introduced [8, §3.3.1].

$keyComp \in ENV \rightarrow KEY$
$keyAuth \in KEY \leftrightarrow AGENT$

Function *keyComp* associates envelopes with the key used to lock them. Relation *keyAuth* associates keys with the agents who have copies of the key. The various operations are then modified to make use of the new variables. The *send* operation described above is refined as follows.

**any**
  $E, K$
**where**
  $E \in ENV \wedge$
  $(E \mapsto M) \in content \wedge$
  $K \in KEY \wedge$
  $keyAuth[\{K\}] = \{A, B\}$
**then**
  $sentBy := sentBy \cup \{E \mapsto A\} \;||$
  $sentTo := sentTo \cup \{E \mapsto B\} \;||$
  $channel := channel \cup$
    $\{E \mapsto (A, B)\}$
**end**

Here $K$ represents key $K_{AB}$ in the protocol. The authority to open the envelope is now indirectly associated with access to the key, rather than directly naming the agents who may do so. The corresponding *receive* operation allows agents in the set $keyAuth[\{K\}]$ only to open the envelope.

As usual, verifying that this refinement step is correct involves some significant proofs. Bieber and Boulahia-Cuppens note that they have used the

$Msg2 =$
**any**
$N_A, N_B$
**where**
$(A, B, m1(N_A, A, K_{AB})) \in history \wedge$
$N_B \in shared(A, B) \wedge$
$N_B \notin nonces(history)$
**then**
$history := history \frown \langle (B, A, m2(N_A, N_B, B, K_{BA})) \rangle$
**end**

Figure 5: A communication action modelled in B [17, §6].

B toolkit's interactive theorem prover to discharge the proof obligations associated with such refinement steps [8, §2.2.3].

Indeed, the complexity of the proof obligations remains a major hurdle to the use of refinement for verifying security protocols. Butler has explored this issue in considerable depth [17][18]. He specifies protocols in B using a history variable to capture the sequence of communications.

Consider the first two steps of the (corrected) Needham-Schroeder protocol [17, Fig. 3].

1. $A \rightarrow B : \{N_A, A\}_{K_{AB}}$
2. $B \rightarrow A : \{N_A, N_B, B\}_{K_{BA}}$

To model this, Butler declares the following variables.

$$history : \text{seq}(AGENT \times AGENT \times MSG)$$
$$share : (AGENT \times AGENT) \rightarrow \mathbb{P}\, NONCE$$

The *history* variable is a sequence consisting of the sender, receiver and message for each communication to date [17, p. 8]. Function *shared* associates pairs of agents with the set of nonces shared between them [17, p. 7].

Then the B operation representing the second communications event in the protocol is shown in Figure 5. Here function $nonces(h)$ extracts the set of nonces appearing in history $h$ [17, p. 8]. Also $\langle e, f, \ldots \rangle$ denotes the sequence consisting of elements $e$, $f$, etc, and $S \frown T$ is sequence $S$ concatenated with sequence $T$. Constructors $m1$ and $m2$ in Figure 5 are tags in the disjoint-union type $MSG$, and serve to compose messages of the particular forms used by this protocol [17, p. 8].

The operation requires that the first protocol message $m1$ already appears in the history. It then composes and appends the second message $m2$ to the history. Once this is done, a separate operation in the model updates the set of nonces that have been seen to date by agent $B$ [17, p. 9].

Butler explains how data refinements of such machines can be performed. B operations are *guarded* by a predicate that defines the conditions under which they may be invoked. This can be calculated from the operation's body [17, §4]. Unusually, Butler allows such guards to be strengthened [18, §3.2][17, §5]. This reduces the number of states in which the operation may occur—a refinement step may therefore result in the concrete

model deadlocking in states where the abstract one could proceed. Normally this would not be a valid data refinement step, but is justified for security protocols on the basis that the security properties of interest are safety properties only, not liveness. A protocol that deadlocks is considered 'secure' since it cannot disclose any information.

In performing data refinements using this model, however, Butler experienced considerable difficulty in devising suitable abstraction invariants. A single refinement step was found to require an invariant involving 12 complex conjuncts [17, §8]. Proofs of side conditions involving such an invariant were extremely demanding. Although the B toolkit generated all of the necessary conditions, its theorem prover discharged only 10% of these automatically [17, §9]. Even worse, any errors or omissions in the choice of abstraction invariant led to considerable wasted effort. In recent work, therefore, Butler has concentrated on ways of systematically deriving the required invariant [18, §3.3]. Although promising, further work seems to be required in this area [18, §9].

Finally, a totally different approach to protocol refinement was proposed by Jürjens [28]. He uses a communications model based on dataflow traces, and writes system models using operators on traces. He defines secrecy as the inability of a stream to disclose a secret and then suggests that refinement *could* be used to preserve this notion. Refinement in this dataflow-based approach has three forms: trace subsetting (reducing nondeterminism), refining interfaces (cf. data refinement) and 'conditional' refinement (strengthening an effect by taking advantage of knowledge in the assumption).

# 5 Conclusion

A wide range of formal methods have been applied to verification of security protocols over many years. That no one technique has yet emerged victorious reflects the challenging nature of the field. Proof by analysis, as embodied in traditional verification and model checking techniques, has received the most attention to date. In this area Paulson's logic [34] and the spi-calculus [3] appear to be the most elegant products to date. Proof by construction, as embodied by the refinement approach, appears to have attracted relatively little attention, and is therefore worthy of further exploration. Butler has made significant inroads into the problem [18], but major simplifications are needed before the approach is practical.

It is clear that security protocols have posed a significant challenge to formal methods. For instance, merely devising suitable data structures to represent the contents of messages has proven to be a stumbling block. We have seen, for instance, message contents awkwardly encoded as syntactic parts of action names in CSP, and as a set of separate variables for each message component in B. Similarly, the simple concept of message sequencing needed to be explicitly represented using counters or history variables in the logics and B machine models. Consequently, security protocol specifications, proofs and refinements have all been discouragingly complex to date.

Also notably absent in the papers surveyed is the notion of probability. Given the recent emergence of mature formalisms for reasoning about probabilistic computations [33], the time may now be ripe to explore probabilistic proofs of protocol security.

**Acknowledgements**

I wish to thank Antonio Cerone for reviewing a draft of this report, and Peter Kearney for supplying the references. This research was funded by the Information Technology Division of the Defence Science and Technology Organisation.

# References

[1] A. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi Calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*. ACM Press, 1997.

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[4] J.-R. Abrial. Cyptographic protocol specification and design. Draft, March 1995.

[5] R. Anderson and R. Needham. Programming Satan's computer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 426–440. Springer-Verlag, 1995.

[6] N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. Technical Report RZ 2976 (#93022), IBM Zurich Research Laboratory, November 1997.

[7] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In H. Krawczyk, editor, *Advances in Cryptology (CRYPTO'98)*, volume 1462 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998. Extended abstract.

[8] P. Bieber and N. Boulahia-Cuppens. Formal development of authentication protocols. In D. Till, editor, *Sixth Refinement Workshop*, pages 80–102. Springer-Verlag, 1994.

[9] C. Boyd. Security architectures using formal methods. *IEEE Journal on Selected Areas in Communications*, 11(5):694–701, June 1993.

[10] C. Boyd. A class of flexible and efficient key management protocols. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pages 2–8. IEEE Computer Society, 1996.

[11] C. Boyd. A framework for design of key establishment protocols. In J. Pieprzyk, editor, *Information Security and Privacy (ACISP'96)*, volume 1172 of *Lecture Notes in Computer Science*, pages 146–157. Springer-Verlag, 1996.

[12] C. Boyd. Towards extensional goals in authentication protocols. Information Security Re-

search Centre, Queensland University of Technology, 1997.

[13] C. Boyd and P. Kearney. Exploring fair exchange protocols using specification animation. In *Proceedings of the Information Security Workshop (ISW 2000)*, volume 1975 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag, 2000.

[14] C. Boyd and W. Mao. On a limitation of BAN logic. In T. Helleseth, editor, *Advances in Cryptology (EUROCRYPT'93)*, volume 1055 of *Lecture Notes in Computer Science*, pages 240–246. Springer-Verlag, 1993.

[15] C. Boyd and W. Mao. Designing secure key exchange protocols. In D. Gollman, editor, *Computer Security (ESORICS'94)*, volume 875 of *Lecture Notes in Computer Science*, pages 93–105. Springer-Verlag, 1994.

[16] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report TR 39, Digital Equipment Corporation, February 1989.

[17] M. Butler. Using refinement to analyse the safety of an authentication protocol. Technical Report DSSE-TR-98-8, Declarative Systems and Software Engineering group, Department of Electronics and Computer Science, University of Southampton, October 1998.

[18] M. Butler. On the use of data refinement in the development of secure communications systems. Technical Report DSSE-TR-2001-1, Declarative Systems and Software Engineering group, Department of Electronics and Computer Science, University of Southampton, January 2001.

[19] U. Carlsen. Cryptographic protocol flaws—know your enemy. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop (CSFW-7)*, June 1994.

[20] U. Carlsen. Generating formal cryptographic protocol specifications. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 137–146. IEEE Computer Society Press, 1994.

[21] A. Durante, R. Focardi, and R. Gorrieri. CVS: A compiler for the analysis of cryptographic protocols, June 1999.

[22] J. Feghhi, P. Williams, and J. Feghhi. *Digital Certificates: Applied Internet Security*. Addison Wesley, 1998.

[23] R. Focardi, A. Ghelli, and R. Gorrieri. Using non interference for the analysis of security protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*. Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University, September 1997.

[24] R. Focardi, R. Gorrieri, and F. Martinelli. Message authentication through non interference. In T. Rus, editor, *Algebraic Methodology And Software Technology (AMAST 2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 258–272. Springer-Verlag, 2000.

[25] W. Fumy and M. Munzert. A modular approach to key distribution. In A. J. Menezes and S. A. Vanstone, editors, *Advances in Cryptology—CRYPTO'90*, volume 537 of *Lecture Notes in Computer Science*, pages 274–283. Springer-Verlag, 1990.

[26] N. Heintze and J. D. Tygar. A model for secure protocols and their compositions. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–13. IEEE Computer Society Press, May 1994. Extended abstract.

[27] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[28] J. Jürjens. Secrecy-preserving refinement. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 135–152. Springer-Verlag, 2001.

[29] P. A. Lindsay. Specification and validation of a network security policy model. In J. C. Bicarregui, editor, *Proof in VDM: Case Studies*, chapter 3, pages 65–93. Springer-Verlag, 1998.

[30] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[31] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[32] C. Mitchell, D. Rush, and M. Walker. A secure messaging architecture implementing the X.400-1988 security features. *The Computer Journal*, 33(4):290–295, 1990.

[33] C. Morgan and A. McIver. pGCL: Formal reasoning for random algorithms. *South African Computer Journal*, (22):14–27, March 1999. Special issue on the 1998 Winter School on Formal and Applied Computer Science.

[34] L. Paulson. The inductive approach to verifying cryptographic protocols. University of Cambridge Computer Laboratory, December 1998.

[35] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[36] C. Rudolph. A formal model for systematic design of key establishment protocols. In C. Boyd and E. Dawson, editors, *Information Security and Privacy (ACISP'98)*, volume 1438 of *Lecture Notes in Computer Science*, pages 332–343. Springer-Verlag, 1998.

[37] S. Schneider. Security properties and CSP. In *Proceedings of the 1996 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society Press, 1996.

[38] S. Schneider. Specification and verification in Timed CSP. In M. Joseph, editor, *Real-Time Systems—Specification, Verification and Analysis*, chapter 6,

pages 147–181. Springer-Verlag, 1996.

[39] S. Schneider. Formal analysis of a non-repudiation protocol. In *Proceedings of the 1998 IEEE Computer Security Foundations Workshop*, 1998.

[40] B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, second edition, 1996.

[41] P. Syverson and C. Meadows. A logical language for specifying cryptographic protocol requirements. In *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 165–177. IEEE Computer Society Press, 1993.

[42] P. F. Syverson and P. C. van Oorschot. On unifying some cryptographic protocol logics. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 14–28. IEEE Computer Society Press, May 1994.

[43] J. C. A. van der Lubbe. *Basic Methods of Cryptography*. Cambridge University Press, 1998.

[44] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194. IEEE Computer Society Press, 1993.

[45] R. Yahalom, B. Klein, and Th. Beth. Trust relationships in secure systems—a distributed authentication perspective. In *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 150–164. IEEE Computer Society Press, 1993.

[46] J. Zhou, R. Deng, and F. Bao. Some remarks on a fair exchange protocol. In H. Imai and Y. Zhong, editors, *Public Key Cryptography 2000*, volume 1751 of *Lecture Notes in Computer Science*, pages 46–57, 2000.