Model Checking Security Protocols Using FDR 1 *

Joshua D. Guttman

guttman@mitre.org

The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420

July 22, 1996

* Work supported by the National Security Agency under United States Army CECOM contract DAAB 07-96-C-E601.

Acknowledgments

I would like to express my indebtedness to Jonathan Millen of The MITRE Corporation. He has taught me a great deal about security protocols, their vulnerabilities, and how to find them.

I am grateful to Paul Gardiner, Michael Goldsmith, and David Jackson of Formal Systems (Europe), and to Gavin Lowe and Bill Roscoe of Oxford University. They developed FDR and pioneered its use to analyze security protocols. I am also grateful to Peter Ryan of the Defence Research Agency (UK), who conceived this line of work and who guided me into it.

I would like to thank David Opitz of the National Security Agency for his insightful comments on a draft of this paper.

Contents

List of Figures

1 Introduction

1.1 The Problem about Security Protocols

Security protocols are distributed algorithms used to distribute public keys, agree on secret keys, or authenticate the presence of a party over a network. The public literature on security protocols stems largely from a classic 1978 paper by Needham and Schroeder [27]. Two facts about these protocols are striking:

- They are remarkably simple, involving generally no more than half a dozen messages, and rarely requiring conditional branches, let alone loops;
- They are frequently wrong [2, 5, 6, 7, 8, 9, 15, 16, 17, 20, 21, 26, 32, 33].

The errors generally tend to be serious, in that an error will frequently allow a malicious participant to completely subvert the purpose of the protocol.

As an example of a security protocol, we give in Figure 1 the original Needham-Schroeder secret key protocol [27]. In this protocol, S is a key server trusted by both parties A and B; N_a and N_b are nonces, which is to say numbers chosen with the intent that they be new and unguessable; and K_{pq} is a symmetric encryption key shared between the parties P and Q. By $\{M\}_K$ we mean the result of encrypting message M using key K.

1.1.1 Existing Approaches (circa 1994)

In 1981, only a few years after the seminal Needham-Schroeder paper [27], Denning and Sacco [9] published a description of a flaw in the Needham-Schroeder secret key protocol.

Let us suppose that the parties A and B use the session key K_{ab} to exchange a large amount of encrypted data during a period of (let us say arbitrarily) a day. The next day they repeat the protocol to select a new key. Meanwhile, a penetrator has recorded the traffic encrypted with K_{ab}, as well as the message $\{K_{ab}, A\}_{K_{bs}}$. Since this is a substantial amount of data encrypted with the same session key, we must expect that it will be vulnerable to cryptanalysis. Indeed, this is presumably the reason why a key distribution protocol is needed at all. So let us suppose that a week's effort by clever cryptanalysts with computers discloses the value of K_{ab}.

1. $A \longrightarrow S: \quad A, B, N_a$
2. $S \longrightarrow A: \quad \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3. $A \longrightarrow B: \quad \{K_{ab}, A\}_{K_{bs}}$
4. $B \longrightarrow A: \quad \{N_b\}_{K_{ab}}$
5. $A \longrightarrow B: \quad \{N_b + 1\}_{K_{ab}}$

Figure 1: The Needham-Schroeder Secret Key Protocol

The penetrator can now trick B into reusing K_{ab}. At about the time B expects to receive a new key from A, the penetrator sends him the old message 3: $\{K_{ab}, A\}_{K_{bs}}$. When B responds with an encrypted nonce $\{N_b'\}_{K_{ab}}$, the penetrator traps it, uses K_{ab} to recover N_b', and returns a message $\{N_b' + 1\}_{K_{ab}}$. B will then accept the replayed key, and—the penetrator hopes—initiate transmission of confidential data using it.

Clearly, these protocols can have subtle flaws. These flaws might be unobserved until after the protocols had been implemented. Then a clever penetrator might exploit them to subvert elaborate network security mechanisms. Indeed, this idea that a flaw might remain unnoticed was validated in 1995, when Gavin Lowe [17] discovered a flaw in the other main protocol introduced by Needham and Schroeder in 1978 [27]. Although a less crucial flaw had been known for years, Lowe's paper illustrates that a classic, man-in-the-middle attack against a well-known protocol may lie hidden for seventeen years.

Needham and Schroeder had been aware that if their protocols were wrong, they might be very subtly wrong; they had recommended that researchers consider systematic methods for determining whether flaws might exist. Perhaps the first paper to make progress in this direction appeared in 1983, written by Dolev and Yao [10]. By the late 80s, two main lines of approach were established. One, exemplified by MITRE's Interrogator program [23, 24] and the Naval Research Laboratory's Protocol Analyzer [21, 22], views the protocol as a state machine. The goal of the analysis is to discover a path through its state space ending in a compromise of information to a penetrator; the NRL tool also supports proofs that no such path exists. Both of these tools adopt a "backwards search" approach: The analysis starts from a state in which a compromise has occurred, and the analyst tries to guide the tool backward through possible state machine transitions until an initial state has been reached. Both tools use algebraic methods (though different ones), so that states may be represented symbolically and whole classes of similar transitions considered at once.

The second main line of approach built logical systems—modal logics of belief—rather than state machines. This approach was initiated by Burrows, Abadi and Needham in what is still the most impressive paper in the field [3]. The paper develops a logic with operators to express the beliefs of the parties to the protocol. For instance, A believes that his nonce N_a is fresh, and he believes that if S sends him a session key K_{ab} encrypted in message 2, then K_{ab} is a suitable key for A and B to use for a session. This latter belief formalizes the idea that the principal A *trusts* the server S as a key server. A logic of this kind allows A to infer that he may safely use K_{ab}; however, it is impossible for B to prove that *he* may safely use K_{ab} without the additional hypothesis that message 3, $\{K_{ab}, A\}_{K_{bs}}$, is fresh. The fact that the logic requires this hypothesis allows an analyst to consider whether B might instead receive a stale message 3, thus leading to a recognition of the Denning-Sacco attack. The paper [3] uses this logic to analyze several protocols with highly illuminating results.

A succession of papers (such as [11]) have extended this logical approach, allowing more operators. Although more aspects of the trust relations among the parties may be expressed, the rules of inference also become more complex.

1.1.2 Drawbacks of the Existing Approaches

Although much of great interest—both theoretical and practical—has been accomplished by the two approaches described, they are acknowledged to have weaknesses.

One important theoretical weakness with the logical approach is that it requires a step of idealization, in which the person carrying out the formalization determines what each message is "saying," as for instance the second message in the Needham-Schroeder secret key protocol says, among other things, that K_{ab} is a suitable key for a session between A and B. There is the question whether the server really intends this, of course, but there is also a separate question whether there might be some subtle reason why the server's intention would be in error. This point was made by Nessett [28], possibly in too blunt a form [4]. A paper by Mao and Boyd [20] raises it in a form that is harder to ignore. They show how the original Burrows, Abadi, Needham paper recommended a flawed version of a basically sound protocol (Otway-Rees [29]), because it appeared strong enough, when translated into their logic, to prove that a good key was exchanged. The point is reinforced by Lowe's flaw [17] in the Needham-Schroeder public key protocol, which is also "proved" correct in [3]. Since the idealization process appears so fragile, and so impervious to straightforward rules, an element of good judgment—or good luck—remains at the core of this formal method.

In the case of the state-machine approaches, the underlying semantics is far more intuitive, and idealization is not a significant problem. Here the drawback is more prosaic. Namely, these tools require a great deal of human ingenuity to use effectively. Now perhaps this should not be surprising: Cryptanalysis also appears highly capital-intensive so far as ingenuity is concerned. However, the tools appear to require a different kind of ingenuity, and a new range of skills. By requiring an analyst to master a genuinely different range of techniques, we run the risk that the set of people willing and able to master both the skill sets needed for traditional cryptanalysis and for these new protocol-oriented flaws may be vanishingly small.

Moreover, the state-machine approaches have not effectively used the computational power of modern machines. Thus, it is attractive to consider a new approach that will shift more of the burden onto computers, and require less new specialized expertise.

1.1.3 The Model Checking Approach

Presumably for much these reasons, in 1994 the United Kingdom Defence Research Agency (DRA) initiated a research program in security protocol analysis. Participants included, among others, Peter Ryan of DRA as well as Bill Roscoe and Gavin Lowe, both connected with Oxford University and the firm Formal Systems (Europe).

They have applied model checking to security protocol analysis. Suppose that a protocol is represented as a labeled transition system, a simple and convenient model for concurrent processes which is also compatible with state machine models. Relations such as refinement between a pair of labeled transition systems can be tested systematically by stepping through them together, and traversing all accessible arcs, at least when the number of arcs is not enormous. This approach to verifying properties of labeled transition systems is called "model checking." Reasonably efficient algorithms for doing so are available in several software packages, FDR being the one previously developed by the Formal Systems (Europe) researchers. FDR is a tool that accepts process description expressed in CSP, a language for expressing Communicating Sequential Processes [13]. These are then translated into labeled transition systems. The system can check whether one labeled transition system *refines* a second, in the sense that possible behaviors of the former are also possible behaviors of the latter. The notion of behavior used here is one which considers *failures* and *divergences* in a technical sense developed in [13] (see also below, page 11); hence the name "Failures and Divergences Refinement."

The advantage of this approach is that the computer can be used effectively in examining the large number of possible transitions. The expertise of the user is required only to represent the protocol itself and also the possible actions of a penetrator. This requires some skill because it is necessary to ensure that the representation will correspond to a small enough transition system that the model checking software can succeed. Success is measured in different terms for flawed and unflawed protocols: In the case of a flawed protocol, the software need only examine enough arcs to find a flaw. In the case of an unflawed protocol, it must visit all of the accessible arcs to be able to report that no flaw exists.

This line of work has now established itself, having discovered new flaws in the Needham-Schroeder public key protocol [17] and others [19].

1.2 Planned Approach

At the outset of this work, I was no expert in security protocols. As a consequence, in the initial portion of this task, I focused on an attempt to understand the essential features of security protocol attacks. In this section I will briefly describe the view I developed in the fall of 1995.

It appears, from a review of a large number of protocol failures [2, 5, 6, 7, 8, 9, 15, 16, 17, 20, 21, 26, 33], that there are above all two main causes for protocol failures. One is that messages or fragments may be recorded by an intruder in a legitimate session, and then replayed as part of an attack. This class of attacks is the main subject of the BAN paper [3]. A second source of attacks, not easily handled within the BAN framework, is the fact that protocols may contain *unintended services*. This insight probably derives originally from Dolev and Yao [10]; Ulf Carlsen [5] discusses them under the name "oracle attacks," providing a number of other references. Two simple examples will explain the basic insight.

1. $A \longrightarrow S$: $B, \{K_{as}\}_{K_s^+}$
2. $S \longrightarrow B$: A
3. $B \longrightarrow S$: $\{K_{ab}\}_{K_s^+}$
4. $S \longrightarrow A$: $\{K_{ab}\}_{K_{as}}$
5. $A \longrightarrow B$: $\{\mathtt{data}\}_{K_{ab}}$

Figure 2: TMN Protocol

1.2.1 An Attack on the TMN Protocol

The first is an attack on the Tatebayashi/Matsuzaki/Newman (TMN) [34] protocol. This ill-fated protocol, intended to establish a session key for secure mobile communications, has inadequate authentication of the parties, and thus fails in many ways.

The protocol, shown in a simple, abstracted form in Figure 2, uses both asymmetric public-key encryption and also symmetric secret-key encryption. We will write $\{M\}_{K^+}$ and $\{M\}_{K^-}$ when K^+ and K^- are the public and private keys of a single key-pair for asymmetric encryption. $\{M\}_{K^+}$ and $\{M\}_{K^-}$ are normally regarded as encryption and decryption respectively. We will write $\{M\}_K$ for symmetric encryption with the shared key K. In this protocol, A first declares to the trusted server S his desire to have a session with B, and his desire to receive the session key encrypted (symmetrically) under the key K_{as}. The key is sent encrypted with the public key of the server S. S informs B of A's desire to communicate. B in turn provides a session key, which he encrypts with the public key of S. S passes the session key to A, encrypted under the key K_{as}.

What is wrong with this protocol is that S is providing an *unintended service*. In order to fulfill his role in the protocol, he must in step 4 provide a translation of the new session key K_{ab} from being encrypted in his own public key to being encrypted in a key selected by the initiator of the run. This "key-translation service" for messages can be manipulated by the initiator and respondent, if they are confederates. For instance, suppose that after a legitimate run, which we will call round 1, the third message 1.3 has been recorded by a corrupt party D. Then D has his confederate C initiate another run of the protocol, which we will call round 2, shown in Figure 3. In step 2.4 the server has obligingly

2.1. $C \longrightarrow S$: $D, \{K_{cs}\}_{K_s^+}$
2.2. $S \longrightarrow D$: C
2.3. $D \longrightarrow S$: $\{K_{ab}\}_{K_s^+}$ (replayed, identical with 1.3)
2.4. $S \longrightarrow C$: $\{K_{ab}\}_{K_{cs}}$

Figure 3: A Simple Attack on the TMN Protocol

1. $A \longrightarrow S$: A, B
2. $S \longrightarrow A$: $\{K_{B+}, B\}_{K_s^-}$
3. $A \longrightarrow B$: $\{N_a, A\}_{K_{B+}}$
4. $B \longrightarrow S$: B, A
5. $S \longrightarrow B$: $\{K_{A+}, A\}_{K_s^-}$
6. $B \longrightarrow A$: $\{N_a, N_b\}_{K_{A+}}$
7. $A \longrightarrow B$: $\{N_b\}_{K_{B+}}$

Figure 4: The Needham-Schroeder Public Key Protocol

3. $A \longrightarrow B$: $\{N_a, A\}_{K_{B+}}$
6. $B \longrightarrow A$: $\{N_a, N_b\}_{K_{A+}}$
7. $A \longrightarrow B$: $\{N_b\}_{K_{B+}}$

Figure 5: The Simplified Needham-Schroeder Public Key Protocol

translated the key for A and B's session into a form that C can read. Thus, if C has recorded the session, he can now decrypt it. This particularly simple attack will suffice for our present purpose, namely giving an example the phenomenon of unintended services. Other attacks are also available for more sophisticated variants of TMN.

1.2.2 An Attack on the Needham-Schroeder Public Key Protocol

Let us consider next Gavin Lowe's attack [17] against the Needham-Schroeder public key protocol [27]. The protocol is shown in Figure 4. As Needham and Schroeder point out, there are two different ingredients in this protocol. One is a mini-protocol for retrieving public-key certificates from the server S. It is executed twice, once in messages 1 and 2, in which A queries S about B's key, and then again in messages 4 and 5, in which B queries S about A's key. The remainder consists of an exchange of nonces between A and B; these nonces may be used in various ways to protect an ensuing session.

The BAN paper [3] shows that the mini-protocol for retrieving public-key certificates is flawed: a principal, B for instance, may have his key compromised and then register a new public key with S. An attacker can then entrap A by replaying a copy of B's old certificate in place of the valid, new message 2. The BAN authors identify this flaw, and then go on to prove that the remainder of the protocol is correct: that is, if messages 2 and 5 are *fresh*, then the protocol is correct.

Their conclusion, however, is unsound [17]. Consider the remaining portion of the protocol, in which we assume that A and B have managed somehow to learn each others' correct public key (Figure 5). In this protocol, A is offering a translation service. In order to play his assigned protocol role, A must translate

1.3 $A \longrightarrow P$: $\{N_a, A\}_{K_{P+}}$
2.3 $P \longrightarrow B$: $\{N_a, A\}_{K_{B+}}$
1,2.6 $B \longrightarrow A$: $\{N_a, N_b\}_{K_{A+}}$ A thinks this message came from P
1.7 $A \longrightarrow P$: $\{N_b\}_{K_{P+}}$
2.7 $P \longrightarrow B$: $\{N_b\}_{K_{B+}}$

Figure 6: Lowe's Attack on the Needham-Schroeder Public Key Protocol

the nonce N_b from a form in which it is encrypted with his own public key, into a form in which it is encrypted with the public key of his interlocutor. In doing so, he may not necessarily be translating a message actually created by this interlocutor. There is however a constraint; A need do so only if N_b is preceded by the nonce N_a which A himself generated at the beginning of an as yet unfinished run.

Let us compare this with the challenge that is presented to any penetrator who would attempt to break the protocol by impersonating A in an interchange with B. The penetrator is presented with a message (message 6 as shown in Figure 5) encrypted with A's public key, which contains a nonce N_b chosen by B that must be translated into B's public key. The translation can be achieved only with the participation of A only himself, as only he can determine the value of N_b from $\{X, N_b\}_{K_{A+}}$. The nonce N_b is accompanied by another nonce (N_a in the notation of Figure 5), which may have been selected by the penetrator.

With this analysis, it is fairly clear what the penetrator can succeed. The penetrator must control some principal P; when the intended victim A initiates a session with P, then P will be able to use that session to impersonate A to a principal B of his choice. The attack, shown in Figure 6, was originally discovered by model checking, but has provided an excellent illustration of the intuition that unintended services lead to attacks. It is so simple that it seems astonishing that it was not discovered, at least in the open literature, for seventeen years.

What error led the BAN paper to its unsound proof? From one point of view, it was an error of idealization [20]. The BAN proof formalizes message 6 as conveying the assertion

$$\{\langle A \overset{N_b}{\rightleftharpoons} B\rangle_{N_a}\}_{K_a^+} \tag{1}$$

This is intended to say, in a form understandable by A alone, that N_b is a secret shared between A and B.[1] A should believe it because it is associated with the secret N_a, presumed known only to A and B. However, this assertion explicitly mentions the principals A and B, while the message itself does not. Thus, while

[1]From a somewhat different point of view, the problem with message 6 could be said to be a problem of honesty. B is saying something that he has no right to believe, because B cannot be certain that N_a is a secret shared only with A. BAN logic certainly assumes that the (uncorrupted) parties to a protocol are honest, and do not say what they cannot responsibly believe.

B thinks of himself as making the assertion shown in Formula 1, A thinks of himself as receiving the assertion

$$\{\langle A \overset{N_b}{\rightleftharpoons} P \rangle_{N_a}\}_{K_a^+}.$$

A similar confusion infects message 7. This is perhaps the perfect example for the third "Prudent Engineering Principle" given by Abadi and Needham [1]:

> **Principle 3**: If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message.

One must also be careful to distinguish principals from the roles that they play in a particular run of a protocol. In this attack, there are three principals, namely A, B, and P. A plays the role of initiator in the first run, and B plays the role of respondent in the second run. P plays the role of respondent in the first run; purporting to be A, P also manages the role of initiator in the second run. Thus, it is indeed the case that the initiator can believe Formula 1 where the variable B is instantiated by whoever is the respondent in the run that produced message 6; however, A cannot determine whether that is in fact the principal named B. Perhaps the lack of a reasonable distinction between principals and their roles contributed to the error that the BAN authors fell into (cf. [32]).

In order to avoid that problem, in this paper we will adopt a convention. Henceforth, A and B will refer to *principals*, who may take on different roles in different runs. P will also refer to a principal, assumed to be controlled by the penetrator. By contrast, for the roles of initiator and respondent, we will use the letters I and R. In the case of a trusted server S, the assumption of trust assures that no other principal can adopt the server's role. Hence this distinction need not be applied to S.

Are there other flaws in this protocol? Apparently not [18]. There is another "unintended service," but it seems impossible to exploit. This service is provided by B; B undertakes to translate nonces encrypted in his public key to a form encrypted in the public key of the sender (as given in the second encrypted position in message 3). This cannot be used to *discover* the value of N_a, except by A himself, as the response comes back encrypted in A's public key. Moreover, no party other than A would produce a message $\{N_a, A\}_{K_{B+}}$ in normal operation. Nor is there any point to any principal using this service to discover values of the form $\{N_a, N_b\}_{K_{A+}}$, by feeding them to B, as anyone can learn K_{A+} and therefore compute these values privately.

1.3 Scope of this Paper

In view of the analysis just described, we planned to devote FY 1996 to three main activities:

1. Examining two prominent model checking approaches for their suitability for this work, namely FDR (mentioned above) and Spin (developed at AT&T Research by Gerard Holzmann and his colleagues [14]);

2. Considering how to incorporate the "unintended services" insight in order to reduce the model checking state space;

3. Implementing the approach and using it, probably initially to rediscover known attacks.

In the current paper, we will document our experiences with FDR (Section 2). We will try to explain why we believe that—though they may be effective in the hands of a tool expert like the Oxford researchers—they are unlikely to form the basis for a tool that can be effectively used by those who are "only" expert in cryptography and protocols.

We have also had some experience with Spin; however, our views there are less conclusive. Thus, although we are skeptical that Spin is appropriate, we are not yet ready to give a detailed argument for that view.

We will conclude (Section 2.5) by very briefly describing a more special-purpose strategy that we think may well lead to a more intuitively controllable tool. The approach described in Section 2.5 supersedes items 2 and 3 in the list above.

2 Experiences with FDR 1

2.1 What are CSP and FDR?

2.1.1 CSP

CSP—"Communicating Sequential Processes" [13]—is a process algebra developed by C.A.R. Hoare in the early 1980s. Essentially, it is a simple notation for describing the behavior of processes, where these processes may be specified as a combination of other processes, for instance as their parallel composition. The CSP process algebra should not be confused with a programming language also named CSP, proposed by Hoare some years earlier [12]; although the two are related in some ways, the earlier design is more operational, with a less explicit semantics, and lacks many of the combining operators that make the process algebra such a powerful notation. It is the earlier programming language that furnishes the basis for Promela, the specification language used in Spin [14].

We do not propose to give a summary of CSP; the CSP book [13] is both readable and definitive. We will only describe some of the elements that make CSP appealing for our purposes. In Section 2.1.3 we will introduce only enough of the CSP notation, in the parseable form used by the FDR tool, to make our examples comprehensible.

The CSP process algebra has two main characteristics that distinguish it from other process algebras, for instance CCS [25]. First, CSP and its formal semantics are designed around a *refinement* relation for processes: Q refines P, written $P \sqsubseteq Q$, means that every possible behavior of Q is a possible behavior of P.[2] In this case, we can regard P as a specification and Q as an implementation of it, or at least a step toward implementation.

Second, CSP is distinguished from CCS and similar process algebras by the fact that it separates synchronization from hiding. When two parallel processes P and Q synchronize on an event e, e still remains a visible event that other processes may use to synchronize with the composite $P \parallel Q$. By contrast, if parallel processes in CCS synchronize, then one of them engages in an input event e and the other engages in a corresponding output event $\bar{e}$; to the outside world this composite event is an internal or silent event τ. That is, the event pairs on which processes synchronize in CCS are automatically hidden. However, because in CSP synchronization does not entail hiding, and because of its use of refinement, parallel composition can express *conjunction* of specifications. A process R satisfies the specifications P and Q just in case it satisfies the

[2] There are in fact several different refinement orders that may be used, corresponding to several different notions of possible behavior. In particular, *traces* refinement, written $P \sqsubseteq_T Q$, means that whenever Q can engage in a sequence of events, then P can engage in that sequence of events. *Failures* refinement, written $P \sqsubseteq_F Q$, means that whenever Q can engage in a sequence of events t and thereafter refuse to perform all of the events offered in a set X, then Q can also perform t and thereafter refuse to perform X. *Failures/divergences* refinement, written $P \sqsubseteq_{FD} Q$, means that $P \sqsubseteq_F Q$ holds, and moreover if Q can engage in t and then diverge by performing an infinite sequence of internal actions without accepting any external events, then so can P. For our purposes, trace refinement appears to be good enough.

specification $P \parallel Q$:

$$P \sqsubseteq R \,\&\, Q \sqsubseteq R \quad \text{iff} \quad (P \parallel Q) \sqsubseteq R$$

This property makes it easy to build up complex process specifications by placing simple ones in parallel composition.

CSP also has aspects which make it easy to give compact and lucid specifications of authentication protocols (and their attackers). First, the legitimate parties to a protocol may be specified without reference to the potential actions of a penetrator; a CSP *renaming* operator allows us to hook the legitimate parties together with the penetrator without change to their specifications, as illustrated in Figure 20 (page 21).

Second, CSP allows events to be structured as a sequence of components, $c.v_1.v_2$ for instance, where c is regarded as a channel name and v_1 and v_2 are regarded as a pair of values being transmitted as a single message over that channel. This mechanism is particularly useful because CSP does not make a fundamental distinction between input and output events. In specifying a process, some components of an event may be specified, while other components are offered as free choices to other processes. The former are indicated with an exclamation point; the latter, with a question mark. Thus, for instance, $c!(a+b)?x$ signifies that the first position must contain the value $a+b$, while any value may occur in the second position, and x may be used to refer to that value later. Thus, when one has an intuitive distinction between outputs and inputs, the former may be indicated with an exclamation point, while the latter are indicated with a question mark. However, the CSP notation may also be used to express odd intermediate cases that do not fit the input/output picture. For our purposes, the CSP convention is attractive, because we can use it for a kind of pattern matching, which requires a later message to contain values predicted by a protocol party, in a way that is quite compact and avoids elaborately nested conditional statements.

2.1.2 FDR

FDR is a tool to decide the truth or falsehood of some statements of the form $P \sqsubseteq Q$. No algorithm can decide all questions of this form, as refinement is an undecidable relation. However, for important classes of process, the question can be answered, and FDR implements algorithms that do so, for instance when P and Q may enter only a (modest) finite number of different states.

FDR 1 has been available for several years now, and it furnished the vehicle for the first round of work of the DRA project. FDR 2 has only just become available to users outside Formal Systems (Europe). It is a great deal more flexible than FDR 1, especially because it integrates a functional programming language with its process notation. Future work will examine its potentialities; at the time of the present writing, we are still arranging to install it at MITRE.

2.1.3 FDR's Notation

The notation used in FDR 1 is the result of a decade's tinkering in how to represent CSP in an ASCII format that can be easily parsed. Data in FDR 1 is derived from numbers, atoms, booleans and the like. Atoms are used to represent values symbolically, and for enumerated types. They are also used to represent channels over which data can be communicated. These objects may be paired together using a dot, as in `channel.value` or `channel.v1.v2`.

Sets and sequences are well-represented, with operations such as `head` and `tail` to destructure sequences. The forms `{a,b}` and `<a,b>` are used to introduce enumerated sets and sequences respectively.

The locution { t | x_0 `<-` s_0, ..., x_n `<-` s_n} is a set defined by applying the axiom of replacement: It refers to the set of all values that t may take on, as the variables $x_0 \ldots x_n$ contained in it range freely over the sets $s_0 \ldots s_n$. Since the cardinality of the resulting set is $\prod_{i \leq n} |s_i|$, and since FDR generally finds occasion to build all such objects, this appealing construct must be used sparingly.

When a channel `c` is declared, the set s of values that may be transmitted over `c` is also specified. If a is one of those values (possibly a dotted pair), then we say that `c.`a is one of the events that may occur on `c`. FDR uses the convenient notation `{| c |}` to refer to the set of all events that may occur on the channel `c`. Similarly, `{| c,d |}` refers to the union of the set of all events occurring on `c` with the set of all events occurring on `d`.

If `P` denotes a process, then `a -> P` denotes the process initially willing to engage in event `a` only, and thereafter behaving like `P`. In case the event is complex, and constructed using the pairing operator `.`, the dot may be replaced by either an exclamation point or a question mark, as in `a!b?c -> P`. These are sometimes explained as indicating output and input respectively, and they can indeed be used in this way. However, CSP's treatment of synchronization is somewhat subtle. The components preceded by exclamation points are components that the process will insist on. If another process executing in parallel cannot accept that value, then the parallel composition will deadlock. Components preceded by question marks are dual: The process will accept any value (of the right type). The question mark is in fact a variable-binding operator. The variable following it may be used throughout the remainder of the process description to refer to the value actually received. When several components are present in the same event, dots may be intermixed with exclamation points and question marks. A dot has the same meaning as the previous operator, and has the meaning of an exclamation point if it is the leftmost pairing operator in the term.

The external choice operator `[] x : s @ e -> P` is similar to the question mark. The expression `e` is required to be a dotted term in which `x` occurs as a component. The resulting process is willing to accept any communication matching `e` such that the component matching `x` is in the set `s`. The value of the component matching `x` is bound to the variable `x` throughout the remainder of `P`. If `s` is the set of all values of some type `T`, then the meaning of

`[] x : s @ a.x.b -> P` is identical to `a?x!b -> P`.

Internal choice, written `|~| x : s @ e -> P`, is similar in meaning, except that the environment cannot determine the value to be selected. Instead, it will be determined within the process, in a way not made explicit.

The process `STOP` is a process that refuses to engage in any action.

If `P` and `Q` are processes, and `A` is a set of events, then `P [| A |] Q` is the process which runs `P` and `Q` in parallel, synchronizing on the events in `A`. This means that an event in `A` may occur only if both processes are willing to engage in it; events not contained in `A` may occur freely with the participation of just one of the processes. If `e` is not in `A`, then even if both processes are willing to engage in it, only one "makes progress" when `e` occurs. The set `A` is frequently specified to be the set of events occurring on a number of channels, such as `c` and `d`, so that it is common to see a process specification of the form `P [| {| c,d |} |] Q`.

The special case where `A` is the empty set is important enough to have its own compact notation. `P ||| Q` means that the behaviors of the two processes are interleaved. A sequence of events t is a trace of `P ||| Q` if it may be divided disjointly into two subsequences t_0 and t_1 such that t_0 is a trace of `P` and t_1 is a trace of `Q`.

If `P` is a process, then `P [[ ... ]]` is a renaming of `P`. The behaviors of `P [[ a <- x, a <- y, b <- z ]]`, for instance, are those that result from behaviors of `P` when we freely replace each occurrence of `a` by either `x` or `y`, and replace all occurrences of `b` by `z`. If `P` is a process and `s` is a set of events, then `identify{s,x,P}` is shorthand for `P [[ a <- x, b <- x, c <- x, ... ]]` where `a`, `b`, `c`, ... are the elements of `s`.

2.2 Representing Security Protocols in CSP

The prescribed behavior of the principals in a security protocol—at least, of the types we have been considering—is normally easy to represent as a finite, labeled transition system, and therefore as a CSP process of the kind that FDR 1 can interpret. We found that these process specifications could be presented in a quite natural form, in spite of the FDR 1 system's poor, uninformative parser, which makes it remarkably time-consuming to bring a specification to a correct syntactic form. Nevertheless, the specifications, once written in the correct form, are intuitive in themselves and certainly quite easy to read, assuming some familiarity with the CSP notation.

To illustrate this point, we will present a specification we wrote to describe the Needham-Schroeder secret key protocol. It exhibits many attractive aspects of the FDR 1 approach to protocol analysis. Many aspects of this specification were suggested by examples developed by Formal Systems (Europe), for instance the files implementing the attacks discussed in [17, 18].

Figure 7 presents the protocol itself again, for convenient reference, but with the roles of initiator and respondent indicated by the letters I and R respectively as proposed in Section 1.2.2. We also replace the final exchange of the encrypted values N_b and $N_b + 1$. Since we will represent nonces by atoms

1. $I \longrightarrow S:$ $\quad I, R, N_i$
2. $S \longrightarrow I:$ $\quad \{N_i, R, K_{ir}, \{K_{ir}, I\}_{K_{rs}}\}_{K_{is}}$
3. $I \longrightarrow R:$ $\quad \{K_{ir}, I\}_{K_{rs}}$
4. $R \longrightarrow I:$ $\quad \{N_r, \text{from\_resp}\}_{K_{ir}}$
5. $I \longrightarrow R:$ $\quad \{N_r, \text{from\_init}\}_{K_{ir}}$

Figure 7: The Needham-Schroeder Secret Key Protocol

```
-- The Principals

Initiator = {A,P}
Responder = {B,P}

-- Nonces and session keys

InitNonceSeq  =  < in_1, in_2, in_3 >
RespNonceSeq  =  < rn_1, rn_2, rn_3 >
SessionKeySeq =  < K_1, K_2 >
InitNonce     =  set(InitNonceSeq)
RespNonce     =  set(RespNonceSeq)
SessionKey    =  set(SessionKeySeq)

Direction     =  { from_resp, from_init }
Data          =  { 0, 1 }
```

Figure 8: CSP: Declaring Entities for NS

(symbols) rather than explicitly as numbers, it would be nonsensical to add one to the nonce as we represent it. Therefore, we will instead use an exchange of the encrypted messages N_r, from_resp and N_r, from_init, with the understanding that the additional atoms might in fact be encoded in an implementation in some straightforward way such as by direction bits, or by adding one.

We first declare the basic entities to be used in the model checking. We stipulate that we will consider runs in which either A or P plays the role of initiator, and either B or P plays the role of respondent. We also allocate six nonces, three to be introduced by initiators in message 1, and three to be introduced by respondents in message 4. Two session keys are allocated. We will allow the initiator and respondent to engage in a session after the key exchange. In this session, the respondent is permitted to send the initiator one bit of data. We include this action so that we will later be able to model the action in which a penetrator breaks an old session key. These declarations take the form shown in Figure 8. The characters "--" start comments, which stretch to the next linefeed. We next declare the data types, which in this case amounts to the sets of objects (message values) that may be transmitted over the various channels of the system (Figure 9).

```
Ticket  =  { Enc.key(r).k1.i | r <- Responder, k1 <- SessionKey,
                               i <- Initiator }
SReply  =  { Enc.key(i).in.r.k1.t | i <- Initiator, in <- InitNonce,
                                    r <- Responder, k1 <- SessionKey,
                                    t <- Ticket}
ENonce  =  { Enc.k1.rn.d | k1 <- SessionKey, rn <- RespNonce,
                           d <- Direction }
EData   =  { Enc.k1.data | k1 <- SessionKey, data <- Data }
```

Figure 9: NS Message Types

```
pragma channel initNonce : InitNonce
pragma channel respNonce : RespNonce

pragma channel start,end : Initiator.Responder
pragma channel s_req     : Initiator.Responder.InitNonce
pragma channel s_rep     : SReply
pragma channel ticket    : Ticket
pragma channel challenge : ENonce
pragma channel response  : ENonce
pragma channel session   : EData
```

Figure 10: Legitimate NS Channels

The locution `Enc.`k`.`d means that the data d are sent encrypted under the key k. This, however, is just a convention. Nothing enforces this interpretation. Thus, the analyst must take care in his coding of the penetrator to ensure that the penetrator learns everything that is not protected by unknown keys, and nothing that is.

Tickets are the objects transmitted from I to R in message 3; SReplies are the replies sent by the server in message 2. An SReply contains an embedded ticket. ENonces and EData contain encrypted nonces and data respectively.

With these message types defined, we next declare the channels that they may be transmitted over (Figure 10). These channels represent only the legitimate behavior of the protocol. The behavior of the penetrator may be grafted on later in quite a slick way using the CSP renaming operator.

Two processes are introduced to dole out the initiators' nonces and respondents' nonces respectively (Figure 11). Their functionality ensures that nonces are available only once.

We next present the specification of the initiator (Figure 12). It is possible to express the initiator as "straight-line code," without explicit conditionals, because of CSP's somewhat subtle treatment of synchronization. The message components preceded by exclamation points (for instance, **`i`**, **`r`**, and **`in`** in **`s_req!i.r.in`**) are values that the initiator `i` will insist on. In some cases (e.g., message 1) the initiator is emitting a message as output. However, in other cases (e.g., message 2), the initiator is receiving a message as input.

```
GiveInitNonce (ins) =
  if (null(ins)) then STOP
  else initNonce!(head(ins)) -> GiveInitNonce(tail(ins))

GiveRespNonce (rns) =
  if (null(rns)) then STOP
  else respNonce!(head(rns)) -> GiveRespNonce(tail(rns))
```

Figure 11: Nonce Management

```
Init (i) =
     start!i?r                         -- Which resp?
  -> initNonce?in                      -- Get new nonce
  -> s_req!i.r.in                      -- Msg 1
  -> s_rep!Enc.key(i).in.r?k1.t        -- Msg 2
  -> ticket!t                          -- Msg 3
  -> challenge!Enc.k1?rn!from_resp     -- Msg 4
  -> response!Enc.k1.rn.from_init      -- Msg 4
  -> session.Enc!k1?data               -- Session
  -> Init(i)                           -- Recur
```

Figure 12: Initiator's Behavior

Nevertheless, the initiator insists on receiving the right values (in the case of message 2, the right shared long-term key `key(i)`, the right nonce value `in`, and the right respondent name `r`), and will deadlock the run of the protocol if these values are not offered. This semantics promotes particularly clear, appealing specifications.

The "dual" of the exclamation point is the question mark: It is in fact a variable-binding operator, and (for instance, in the nonce message `initNonce?in`) it causes the identifiers to be bound, throughout the remainder of their scope, to the values communicated in this message.

The behavior of the server process is very simple (Figure 13). Its state consists of the sequence of not-yet-used session keys. When these are exhausted it can only stop. Otherwise, it receives requests from initiators and responds by embedding the first (`head`) session key into a response, encrypted once for the

```
Server (sks) =
  if (null(sks)) then STOP
  else s_req?i.r.in                    -- Msg 1
  -> s_rep!Enc.key(i).in.r.(head(sks)).Enc.key(r).(head(sks)).i
                                       -- Msg 2
  -> Server(tail(sks))                 -- Recur
```

Figure 13: Server's Behavior

```
Resp (r) =
  ticket!Enc.key(r)?k1.i                -- Msg 3
  -> respNonce?rn                       -- Get a new nonce
  -> challenge!Enc.k1.rn.from_resp      -- Msg 4
  -> response!Enc.k1.rn.from_init       -- Msg 5
  -> session!Enc.k1?data                -- Engage in session
  -> end.i.r                            -- Confirm completion
  -> Resp(r)                            -- Relax
```

Figure 14: Respondent Behavior

```
InitProcs = Init(I) ||| Init(A)
RespProcs = Resp(I) ||| Resp(B)

Untrusted =
        (InitProcs [| {| initNonce |} |] GiveInitNonce(InitNonceSeq))
    [| {| ticket, challenge, response, session |} |]
        (RespProcs [| {| respNonce |} |] GiveRespNonce(RespNonceSeq))

System =
        Server(SessionKeySeq)
    [| {| s_req, s_rep |} |]
        Untrusted

LivenessSpec = [] x : diff(Sigma, {| end |}) @ x -> LivenessSpec
```

Figure 15: NS System without Penetrator

initiator and once (in a Ticket) for the responder. It calls itself recursively with the remainder (`tail`) of the session keys.

The behavior of the responder is likewise straightforward (Figure 14).

We can also easily piece these definitions together to form a representation of the total protocol behavior without the presence of a penetrator. We can use this system-level description to test our specifications, for instance to ensure that the expected protocol run is in fact possible. Various silly errors in previous versions were disclosed in this way.

To do so, we must put the initiators in parallel with the process `GiveInitNonce`, synchronizing (cooperating) on the event `initNonce`. Similarly, the respondents are put in parallel with `GiveRespNonce`, synchronizing on the event `respNonce`. Then the resulting compound processes are put in parallel with each other, synchronizing on messages 3–5, followed by the session itself. This compound represents the behavior of the untrusted principals. We may put it in parallel with the trusted key server, synchronizing on messages 1 and 2. As a test of the liveness of the resulting system, we pretend that we do not want the respondents ever to engage in the event `end`, signifying successful completion of a session (Figure 15). The FDR 1 system now promptly responds

```
<start.A.B,
 initNonce.in_1,
 s_req.A.B.in_1,
 s_rep.Enc.K_a.in_1.B.K_1.Enc.K_b.K_1.A,
 ticket.Enc.K_b.K_1.A,
 respNonce.rn_1,
 challenge.Enc.K_1.rn_1.from_resp,
 response.Enc.K_1.rn_1.from_init,
 session.Enc.K_1.0>
```

Figure 16: NS System without Penetrator: A Desired Behavior

with a counterexample to this pessimistic specification, namely the desired behavior shown in Figure 16.

We would summarize this example by concluding that the normal behavior of this sort of protocol can be compactly, understandably, even elegantly represented in CSP. Other examples, such as for instance the Otway-Rees protocol, were also found to fit this mold quite nicely. There are protocols that would be difficult to formualte in this way within FDR 1, for instance the selective broadcast scheme of Simmons [31]. The obstacle is that the protocol treats expressions uniformly despite having different levels of nesting for encryption operators (and other operators). It is perfectly reasonable for a protocol to treat expressions of different depths in the same way, because in some kinds of implementation the expressions are simply different bitstrings of the same length. However, the approach we have used, following researchers at FS(E), would represent the expressions as having different types. This is a fact specifically about FDR 1, and FDR 2 has a more flexible approach to data types for which it is no longer true.

2.3 Representing the Penetrator in CSP

The penetrator is quite a different sort of process from the normal protocol participants. The normal protocol participants are highly constrained in their possible behaviors. Although input values such as nonces help to determine the exact behavior in which they will engage, the behaviors triggered by the different values tend to differ only in choice of names. Resource exhaustion (no more nonces or session keys, in our current example) cause deadlock rather than interesting failure recovery.

By contrast, the penetrator must be designed to have many different possible runs, to explore different possible penetration attacks. The trick is to balance off the size of the state space, which, if too large, limits the proportion of possible scenarios that can in fact be examined by the model checking software, and which, if too small, limits the ability of the penetrator to learn from the data disclosed by the legitimate participants.

In the case of the Denning-Sacco attack on the Needham-Scroeder secret

```
pragma channel take_ticket,     fake_ticket     : Ticket
pragma channel take_challenge                   : ENonce
pragma channel                  fake_response   : ENonce
pragma channel take_session                     : EData
pragma channel break_session_key                : SessionKey
pragma channel jubilate                         : Data
```

Figure 17: Events Controlled by the Penetrator

key protocol, there is no difficulty in creating a successful attacker. We must introduce some additional channels. If `event` is a legitimate event, we will use `fake_event` to represent the penetrator creating a false communication of kind `event`, and `take_event` to represent the penetrator trapping a communication of kind `event`. In the latter case, the intended recipient never sees the event, while in the former case, the apparent sender never sees the event. The penetrator also participates in two other events, also shown in Figure 17: `break_session_key`, which models the possibility that cryptanalysis will break an old session key, and `jubilate`, which signals a successful protocol attack.

The penetrator itself requires some care to program. The simplest approach would be to represent it as a single process, in which case its state would consist of a set of tickets that might be reused, and a set of session keys that have been broken and are available to decode the data of the legitimate parties. This has the drawback that the set of states for the process is then the product of the number of sets of tickets with the number of the sets of session keys. FDR 1's compile time depends on the size of this state space, and its execution time and space requirements do also. Moreover, tickets and session keys are used independently: A ticket is only used for replay, and a session key is only used to fake a response or decode data transmitted in a session. Thus, we may reduce the state space of the penetrator by isolating these two state ingredients into separate processes.

The first penetrator process (Figure 18) keeps as its state a set of tickets that have been observed in the past. Its behavior is to add a ticket to its state whenever a `take_ticket` event occurs, and periodically to try replaying its tickets in `fake_ticket` events. When the state contains too many tickets—that is, more than N of them—then the penetrator forgets one nondeterministically. It is quite sufficient to let N be 1.

The other penetrator process keeps as its state a set of session keys that have been learnt from `break_session_key` events (Figure 19). A key may be used to fake a response to a challenge issued in a genuine message 4, or it may be used to decode a session between legitimate users, if the penetrator ever tricks them into reusing a session key. As before, if the set comes to be too big, then the process forgets one nondeterministically.

We must now use CSP renaming to modify the initiator and respondent process descriptions to account for the possibility of interception or faking. The possibility of using renaming in this way is quite an attractive aspect of using

```
Penetrator_t(tkts) =
  if (N < card(tkts))
  then |~| t : tkts @ Penetrator_t(diff(tkts,{t}))
  else
        take_ticket?t -> Penetrator_t(union({t}, tkts))
    []
        [] t : tkts @ fake_ticket.t -> Penetrator_t(tkts)
```

Figure 18: Penetrator's Treatment of Tickets

```
Penetrator_k(sks) = if (N < card(sks))
  then |~| s : sks @ Penetrator_k(diff(sks,{s}))
  else ([]  s : sks @ take_challenge!Enc.s?rn!from_resp
        ->  fake_response!Enc.s.rn.from_init
        ->  Penetrator_k(sks))
    [] ([]  s : sks @ take_session!Enc.s?data
        ->  jubilate!data
        ->  Penetrator_k(sks))
    [] (session!Enc?s.data
        -> break_session_key!s
        -> Penetrator_k(union({s},sks)))
```

Figure 19: Penetrator's Treatment of Session Keys

CSP to specify these protocols. The "renamed" process definitions are shown in Figure 20. For each outgoing message, either the message may proceed normally to the recipient, or else the message may be trapped by the penetrator. Thus, there are the two possibilities `event <- event` and `event <- take_event`. Dually, for each incoming message, the message may either be a genuine one sent by the apparent sender, or else it may be a fake one created by the penetrator. Hence, the two possibilities `event <- event` and `event <- fake_event`. In the case of the protocol at hand, there is no point in messing around with the first two messages, namely the initiator's request to the server and the server's reply. This reflects the cryptographic assumption that the initiator's long-term

```
Init_A = Init(A)
        [[ ticket <- ticket,            ticket <- take_ticket ]]

Resp_B = Resp(B)
        [[ ticket <- ticket,            ticket <- fake_ticket,
           challenge <- challenge,      challenge <- take_challenge,
           response <- response,        response <- fake_response,
           session <- session,          session <- take_session ]]
```

Figure 20: Renamings to Account for Penetrator Actions

```
InitProcs = Init_P ||| Init_A
RespProcs = Resp_P ||| Resp_B

Untrusted =
        (InitProcs [| {| initNonce |} |] GiveInitNonce(InitNonceSeq))
    [| {| ticket, challenge, response, session |} |]
        (RespProcs [| {| respNonce |} |] GiveRespNonce(RespNonceSeq))

Untrusted_v =
        (Penetrator_t({})
    [| {| take_ticket, fake_ticket |} |]
         Untrusted)
    [| {| take_challenge, fake_response, take_session, session |} |]
         Penetrator_k({})

System =
        Server(SessionKeySeq)
    [| {| s_req, s_rep |} |]
        Untrusted_v

SafetySpec = [] x : diff(Sigma, {| jubilate |}) @ x -> SafetySpec
```

Figure 21: NS System with Penetrator

master key (shared with the server) cannot be compromised. Hence we do not rename those events.

The system as a whole with the possibility of penetration is shown in Figure 21. The safety specification stipulates that anything may happen, so long as the penetrator never engages in the event `jubilate`. This is in fact a "hand-coded" specification of the security goal of the system. It depends on an understanding of what the protocol is trying to accomplish, and of what the penetrator might plausibly want to achieve. Our second example has a different security goal (Section 2.4.3). For a recent discussion of the question, how to formulate security goals in a well-motivated form, see [30].

However, FDR 1, when asked to check the security specification, disposes of it in short order as a piece of wishful thinking, giving the expected Denning-Sacco attack [9] shown in Figure 22.

2.4 A More Demanding Example

The Denning-Sacco attack is particularly easy to find. There are really only two kinds of item that the penetrator is interested in using, namely tickets and session keys. Moreover, there's no question about what actions to perform on these items. The only thing to do with a ticket is to replay it; the penetrator will never be able to decode it because he can never learn a long-term master key. Nor can the penetrator ever construct a phony ticket. Session keys will be

```
<start.A.B,
 initNonce.in_1,
 s_req.A.B.in_1,
 s_rep.Enc.K_a.in_1.B.K_1.Enc.K_b.K_1.A,
 take_ticket.Enc.K_b.K_1.A,
 fake_ticket.Enc.K_b.K_1.A,
 respNonce.rn_1,
 challenge.Enc.K_1.rn_1.from_resp,
 response.Enc.K_1.rn_1.from_init,
 session.Enc.K_1.1,
 end.A.B,
 fake_ticket.Enc.K_b.K_1.A,
 respNonce.rn_2,
 break_session_key.K_1,
 take_challenge.Enc.K_1.rn_2.from_resp,
 fake_response.Enc.K_1.rn_2.from_init,
 take_session.Enc.K_1.1>
```

Figure 22: Attack against the Needham-Schroeder Protocol

1. $I \longrightarrow R: \quad M, I, R, \{N_i, M, I, R\}_{K_i}$
2. $R \longrightarrow S: \quad M, I, R, \{N_i, M, I, R\}_{K_i}, \{N_r, M, I, R\}_{K_r}$
3. $S \longrightarrow R: \quad M, \{N_i, K\}_{K_i}, \{N_r, K\}_{K_r}$
4. $R \longrightarrow I: \quad M, \{N_i, K\}_{K_i}$

Figure 23: The Otway-Rees Protocol

found only by the `break_session_key` event; they can be used only to decode a session.

2.4.1 The Otway-Rees Protocol

Many other protocols are less simple. The Otway-Rees protocol is an (apparently sound) example, shown in Figure 23. If the penetrator controls a particular principal P, then there may be attacks in which P generates messages encrypted under his long-term master key, as is suggested by the existence of an attack on a variant of the Otway-Rees protocol [20]. In [3], the BAN authors analyze the Otway-Rees Protocol and determine that it can be simplified and improved somewhat, while achieving the same trust objectives (Figure 24). This version succumbs to an attack if the penetrator's principal P has a message fragment $\{M, P, B\}_{K_b}$ left over from a previous exchange which P has initiated with B. The component M is a round identifier; the participants do not rely on its being fresh or unpredictable, nor would an implementation check whether a round identifier is being reused.

With this message fragment in hand, P may embark on an attack due to

1. $I \longrightarrow R:$ $M, I, R, \{N_i, M, I, R\}_{K_i}$
2. $R \longrightarrow S:$ $M, I, R, \{N_i, M, I, R\}_{K_i}, N_r, \{M, I, R\}_{K_r}$
3. $S \longrightarrow R:$ $M, \{N_i, K\}_{K_i}, \{N_r, K\}_{K_r}$
4. $R \longrightarrow I:$ $M, \{N_i, K\}_{K_i}$

Figure 24: "Improved" Otway-Rees Protocol

1\. $P \longrightarrow B:$ $M, A, B, \{N_p, M, P, B\}_{K_p}$

2.1 $B \longrightarrow P:$ $M, A, B, \{N_p, M, P, B\}_{K_p}, N_b, \{M, A, B\}_{K_b}$
(meant for S)

2.2 $P \longrightarrow S:$ $M, P, B, \{N_p, M, P, B\}_{K_p}, N_b, \{M, P, B\}_{K_b}$
(apparently from B)

3.1 $S \longrightarrow P:$ $M, \{N_p, K\}_{K_p}, \{N_b, K\}_{K_b}$
(meant for B)

3.2 $P \longrightarrow B:$ $M, \{N_p, K\}_{K_p}, \{N_b, K\}_{K_b}$
(apparently from S)

4\. $B \longrightarrow P:$ $M, \{N_p, K\}_{K_p}$
(meant for A)

Figure 25: Attack on "Improved" Otway-Rees Protocol

Mao and Boyd [20]. It is shown in Figure 25.[3] This attack causes B to believe that the key K is shared with A, whereas it is in fact known to P instead. This attack requires P to construct the encrypted $\{N_p, M, P, B\}_{K_p}$ using the value M known from a previous run.

2.4.2 Representing Otway-Rees in FDR 1

The Otway-Rees protocol (and its variant) may be very easily coded into CSP, using much the same style as before. The basic entities are shown in Figure 26.

The messages used by the protocol may be broken into a number of convenient pieces. We will regard a triple $\langle m, i, r\rangle$ as a query identifier. We will also regard an encrypted unit of the form $\{N, M, I, R\}_K$ as a package containing a nonce or "nonce pack." In the protocol, a nonce pack always originates either with the initiator, in which case $K = K_i$ or with the respondent, in which case $K = K_r$. It is convenient, in order to restrict the number of entities, to distinguish initiator nonce packs from respondent nonce packs as different message

[3] It is always a question how to represent attacks, in which the actual sender and recipient of a message frequently differ from the apparent sender and intended recipient. In Figure 25 we adopt the convention that $A \longrightarrow B$ means that A actually originated the message and that B actually received it. Apparent sender and intended recipient are indicated by a verbal comment, where necessary.

```
Initiator = {A,P}
Responder = {A,P}
Principal = union(Initiator,Responder)

-- Nonces and session keys

InitNonceSeq    =  < in_1, in_2 >
RespNonceSeq    =  < rn_1, rn_2 >
RoundIdSeq      =  < m_1,  m_2 >
SessionKeySeq   =  < K_1, K_2 >
InitNonce       =  set(InitNonceSeq)
RespNonce       =  set(RespNonceSeq)
Nonce           =  union(InitNonce,RespNonce)
RoundId         =  set(RoundIdSeq)
SessionKey      =  set(SessionKeySeq)
```

Figure 26: Basic Entities for Otway-Rees

types.[4] Similarly, we regard an encrypted unit of the form $\{N, K\}_{K'}$ as a key pack, and we distinguish key packs according to whether they are intended for an initiator or a respondent. The resulting message types are defined in Figure 27. The channels may now be introduced in Figure 28. Channel names begin with two letters from the set $\{\mathtt{i}, \mathtt{r}, \mathtt{s}\}$. The first letter indicates whether the channel carries messages from the initiator, respondent or server, respectively, and the second letter indicates which role the message is directed to.

A few simple processes are needed to manage the nonces and round identifiers (Figure 29). The initiator process (Figure 30) begins by accepting the name of a respondent, after which it retrieves an initiator's nonce and a round identifier. It then sends the respondent a query identifier and an initiator's nonce pack. The respondent eventually returns the same round identifier and an initiator's key pack with the new session key. The initiator acknowledges its success and then returns to the top of its loop.

The responder is more complex. We show it (Figure 31) divided into a sequence of three subprocesses so as to make explicit the state required at different moments.

The behavior of the server, as usual, is straightforward (Figure 32). With this we have completed a compact and, I believe, attractive specification of the Otway-Rees protocol. Its system-level structure, without the penetrator,

[4] Convenient, but perhaps dangerous. An analysis of the protocol makes it clear that a nonce pack is always unambiguously of one kind or the other, and in the normal execution of the protocol there is no loss in distinguishing them syntactically. Moreover, in the FDR context, there is a significant advantage: The parser took prohibitively long to translate a naïve version of the specification lacking this distinction, because the set of nonce packs was too large. However, an attack might be overlooked as a result, because it depends on the fact that an attacker can substitute a responder's nonce pack in place of an initiator's nonce pack in some circumstances. The (relatively weak) attack given below in Figure 44, page 35, is of just this kind.

```
QueryId         = { rid.i.r | rid <- RoundId,
                              i <-  Initiator, r <- Responder }
INoncePack      = { Enc.key(i).in.rid.i.r |
                              in <-  InitNonce, rid <- RoundId,
                              i <-  Initiator, r <- Responder }
RNoncePack      = { Enc.key(r).rn.rid.i.r |
                              rn <-  RespNonce, rid <- RoundId,
                              i <-  Initiator, r <- Responder }
IKeyPack        = { Enc.key(i).in.k |
                              i <-  Initiator, in <-  InitNonce,
                              k <- SessionKey }
RKeyPack        = { Enc.key(r).rn.k |
                              r <- Responder, rn <-  RespNonce,
                              k <- SessionKey }
```

Figure 27: Message Types

```
pragma channel initNonce                : InitNonce
pragma channel respNonce                : RespNonce
pragma channel roundId                  : RoundId
pragma channel start,confirm,end        : Initiator.Responder

pragma channel irQueryId                : QueryId
pragma channel irINoncePack             : INoncePack

pragma channel rsQueryId                : QueryId
pragma channel rsINoncePack             : INoncePack
pragma channel rsRNoncePack             : RNoncePack

pragma channel srRoundId                : RoundId
pragma channel srIKeyPack               : IKeyPack
pragma channel srRKeyPack               : RKeyPack

pragma channel riRoundId                : RoundId
pragma channel riIKeyPack               : IKeyPack
```

Figure 28: Channels

```
GiveInitNonce (ins) =
  if (null(ins)) then STOP
  else initNonce!(head(ins)) -> GiveInitNonce(tail(ins))

GiveRespNonce (rns) =
  if (null(rns)) then STOP
  else respNonce!(head(rns)) -> GiveRespNonce(tail(rns))


GiveRoundId (rids) =
  if (null(rids)) then STOP
  else roundId!(head(rids)) -> GiveRoundId(tail(rids))
```

Figure 29: Nonce and Round Identifier Management

```
Init(i) =
        start.i?r
    ->  initNonce?in
    ->  roundId?m
    ->  irQueryId!m.i.r
    ->  irINoncePack!Enc.key(i).in.m.i.r
    ->  riRoundId!m
    ->  riIKeyPack!Enc.key(i).in?k
    ->  end!i.r -> Init(i)
```

Figure 30: Otway-Rees Initiator

```
Resp(r) =
        irQueryId?m.i.r
    ->  irINoncePack?q
    ->  respNonce?rn
    ->  rsQueryId!m.i.r
    ->  rsINoncePack!q
    ->  Resp1(r,i,m,rn)

Resp1(r,i,m,rn) =
        rsRNoncePack!Enc.key(r).rn.m.i.r
    ->  srRoundId!m
    ->  srIKeyPack?kp
    ->  srRKeyPack!Enc.key(r).rn?k
    ->  Resp2(r,i,m,kp)

Resp2(r,i,m,kp) =
        confirm!i.r
    ->  riRoundId!m
    ->  riIKeyPack!kp
    ->  Resp(r)
```

Figure 31: Otway-Rees Respondent

```
Server(sks) = if (null(sks)) then STOP else
        rsQueryId?m.i.r
    ->  rsINoncePack!Enc.(key(i))?in!m.i.r
    ->  rsRNoncePack!Enc.(key(r))?rn!m.i.r
    ->  srRoundId!m
    ->  srIKeyPack!Enc.(key(i)).in.(head(sks))
    ->  srRKeyPack!Enc.(key(r)).rn.(head(sks))
    ->  Server(tail(sks))
```

Figure 32: Otway-Rees Server

```
InitProcs = Init(P) ||| Init(A)
RespProcs = Resp(P) ||| Resp(B)

Untrusted =
        ((InitProcs [| {| initNonce |} |] GiveInitNonce(InitNonceSeq))
                   [| {| roundId |} |]   GiveRoundId(RoundIdSeq))
    [| {| irQueryId, irINoncePack, riRoundId, riIKeyPack |} |]
        (RespProcs [| {| respNonce |} |] GiveRespNonce(RespNonceSeq))

System =
        Server(SessionKeySeq)
    [| {| rsQueryId, rsINoncePack, rsRNoncePack,
          srRoundId, srIKeyPack, srRKeyPack |} |]
        Untrusted

LivenessSpec = [] x : diff(Sigma, {| end |}) @ x -> LivenessSpec
```

Figure 33: Otway-Rees System without Penetrator

may be specified in a way analogous to our Needham-Schroeder example (see Figure 15); we show it in Figure 33.

2.4.3 An Otway-Rees Penetrator

Specifying the penetrator in a satisfactory way is, however, more difficult. The reason is the fact mentioned earlier (page 23), namely that there are more operations on the data that may possibly be of use to the penetrator. As a consequence, the penetrator must maintain more state, and that also means that it must be constructed as a more unified process. When we constructed the Needham-Schroeder penetrator as a pair of processes, one concerned with tickets and one concerned with session keys, we were relying on the fact that no interesting information can be discovered by applying a session key to a ticket (which is instead encrypted using a long-term master key). Nor can any interesting data objects be faked by applying a session key to a ticket. If these combinations could be fruitful, a penetrator who separated these kinds of information into distinct processes that interleave without communicating might miss some crucial trick.

Indeed, when we look at the Mao-Boyd attack on the modified Otway-Rees protocol (Figure 25), we see that it combines the penetrator's master key $\mathtt{key}(P)$ with the values contained in the saved respondent's nonce pack $\{M, P, B\}_{K_b}$. The penetrator must also retain the round identifier M for use later in message 2.2.

As a consequence, a penetrator in the style of the Needham-Schroeder penetrator is going to be inadequate. It does not remember past round identifiers, and hence cannot be expected to find the attack. Nevertheless, an implementation of the same general approach, adapted to the Otway-Rees protocol, is

```
pragma channel take_irQueryId, fake_irQueryId          : QueryId
pragma channel take_irINoncePack, fake_irINoncePack    : INoncePack

pragma channel take_rsQueryId, fake_rsQueryId          : QueryId
pragma channel take_rsINoncePack, fake_rsINoncePack    : INoncePack
pragma channel take_rsRNoncePack, fake_rsRNoncePack    : RNoncePack

pragma channel fake_srRoundId                          : RoundId
pragma channel take_srIKeyPack, fake_srIKeyPack        : IKeyPack
pragma channel take_srRKeyPack, fake_srRKeyPack        : RKeyPack

pragma channel fake_riRoundId                          : RoundId
pragma channel fake_riIKeyPack                         : IKeyPack
```

Figure 34: Otway-Rees Penetrator: Channels

```
PQueryId(ms, qids) =
  if (N+1 < card(ms))
  then |~| m : ms @ PQueryId(diff(ms,{m}), qids)
  else if (N < card(qids))
  then |~| q : qids @ PQueryId(ms, diff(qids,{q}))
  else take_irQueryId?m.i.r
          -> PQueryId(union({m},ms),union({m.i.r}, qids))
    []
       ([] q : qids @ fake_rsQueryId!q -> PQueryId(ms, qids))
    []
       ([] m : ms @ fake_srRoundId!m -> PQueryId(ms, qids))
    []
       ([] m : ms @ fake_riRoundId!m -> PQueryId(ms, qids))
```

Figure 35: Otway-Rees Penetrator: Round Ids and Query Ids

shown in Figures 34–39.

In order to check this specification, we must again introduce the rewritings of the principals in order to take account of the fact that messages may be faked or taken by the penetrator. Figure 40 shows these renamings in the case of one initiator and one respondent; in the case of the principal P, who is controlled by the penetrator and must be able to initiate sessions and respond to them, the renamings are similar. In this figure, we also present renamings to the server to make the server vulnerable, since messages to the server may be phony, and messages from the server may be trapped before reaching their intended recipient. We then construct the system from these components as usual, taking the vulnerable versions of the initiators, responders, and server (Figure 41). Finally, we add in the penetrator, who synchronizes with the system using the "`fake`" and "`take`" events (Figure 42). A liveness specification for the system ensures that the FDR 1 representation is reasonable (Figure 43).

```
PINoncePack(inps) =
  if (N < card(inps))
  then |~| inp : inps @ PINoncePack(diff(inps,{inp}))
  else
        take_irINoncePack?inp -> PINoncePack(union(inps,{inp}))
    []
        take_rsINoncePack?inp -> PINoncePack(union(inps,{inp}))
    []
        ([] inp : inps @ fake_irINoncePack!inp -> PINoncePack(inps))
    []
        ([] inp : inps @ fake_rsINoncePack!inp -> PINoncePack(inps))
```

Figure 36: Otway-Rees Penetrator: Initiator Nonce Packs

```
PRNoncePack(rnps) =
  if (N < card(rnps))
  then |~| rnp : rnps @ PRNoncePack(diff(rnps,{rnp}))
  else
        take_rsRNoncePack?rnp -> PRNoncePack(union(rnps,{rnp}))
    []
        ([] rnp : rnps @ fake_rsRNoncePack!rnp -> PRNoncePack(rnps))
```

Figure 37: Otway-Rees Penetrator: Respondent Nonce Packs

```
PRKeyPack(rkps)=
  if (N < card(rkps))
  then |~| rkp : rkps @ PRKeyPack(diff(rkps,{rkp}))
  else
        take_srRKeyPack?rkp -> PRKeyPack(union(rkps,{rkp}))
    []
        ([] rkp : rkps @ fake_srRKeyPack?rkp -> PRKeyPack(rkps))
```

Figure 38: Otway-Rees Penetrator: Respondent Key Packs

```
PIKeyPack(ikps)=
  if (N < card(ikps))
  then |~| ikp : ikps @ PIKeyPack(diff(ikps,{ikp}))
  else
        take_srIKeyPack?ikp -> PIKeyPack(union(ikps,{ikp}))
    []
        ([] ikp : ikps @ fake_srIKeyPack?ikp -> PIKeyPack(ikps))
    []
        ([] ikp : ikps @ fake_riIKeyPack?ikp -> PIKeyPack(ikps))
```

Figure 39: Otway-Rees Penetrator: Initiator Key Packs

```
Init_A =
 Init(A)
    [[ irQueryId <- irQueryId,          irQueryId <- take_irQueryId,
       irINoncePack <- irINoncePack,    irINoncePack <- take_irINoncePack,
       riRoundId <- riRoundId,          riRoundId <- fake_riRoundId,
       riIKeyPack <- riIKeyPack,        riIKeyPack <- fake_riIKeyPack
       ]]

Resp_B =
 Resp(B)
    [[ irQueryId <- irQueryId,          irQueryId <- fake_irQueryId,
       irINoncePack <- irINoncePack,    irINoncePack <- fake_irINoncePack,
       rsQueryId <- rsQueryId,          rsQueryId <- take_rsQueryId,
       rsINoncePack <- rsINoncePack,    rsINoncePack <- take_rsINoncePack,
       rsRNoncePack <- rsRNoncePack,    rsRNoncePack <- take_rsRNoncePack,
       srRoundId <- srRoundId,          srRoundId <- fake_srRoundId,
       srIKeyPack <- srIKeyPack,        srIKeyPack <- fake_srIKeyPack,
       srRKeyPack <- srRKeyPack,        srRKeyPack <- fake_srRKeyPack
       ]]

Server_v =
 Server(SessionKeySeq)
    [[ rsQueryId <- rsQueryId,          rsQueryId <- fake_rsQueryId,
       rsINoncePack <- rsINoncePack,    rsINoncePack <- fake_rsINoncePack,
       rsRNoncePack <- rsRNoncePack,    rsRNoncePack <- fake_rsRNoncePack,
       srIKeyPack <- srIKeyPack,        srIKeyPack <- take_srIKeyPack,
       srRKeyPack <- srRKeyPack,        srRKeyPack <- take_srRKeyPack
       ]]
```

Figure 40: Otway-Rees Renamings

```
InitProcs = Init_A ||| Init_P
RespProcs = Resp_A ||| Resp_P

Untrusted =
        ((InitProcs [| {| initNonce |} |] GiveInitNonce(InitNonceSeq))
                    [| {| roundId |} |]   GiveRoundId(RoundIdSeq))
    [| {| irQueryId, irINoncePack, riRoundId, riIKeyPack |} |]
        (RespProcs [| {| respNonce |} |] GiveRespNonce(RespNonceSeq))

System =
        Server_v
    [| {| rsQueryId, rsINoncePack, rsRNoncePack,
          srRoundId, srIKeyPack, srRKeyPack |} |]
        Untrusted
```

Figure 41: Otway-Rees System Specification, with Vulnerable Components

```
System_v =
        System
    [| {| take_irQueryId, fake_irQueryId,
          take_irINoncePack, fake_irINoncePack,

          take_rsQueryId, fake_rsQueryId,
          take_rsINoncePack, fake_rsINoncePack,
          take_rsRNoncePack, fake_rsRNoncePack,

          fake_srRoundId,
          take_srIKeyPack, fake_srIKeyPack,
          take_srRKeyPack, fake_srRKeyPack,

          fake_riRoundId,
          fake_riIKeyPack  |} |]
        Penetrator
```

Figure 42: Otway-Rees System with Penetrator

In addition, a safety specification demands that whenever the respondent offers a `confirm` event to indicate that it has committed to a session, then the corresponding initiator has executed a `start` event previously, indicating that it has genuinely requested a session. This specification, expressed in the process `SafetySpec` in Figure 43, is a hand-coded condition, although a different hand-coded condition from the one we used for the Needham-Schroeder Secret Key protocol. It expresses the goal of ensuring that in each session that runs to completion in a `confirm` event, the responder has ascertained that a genuine request has occurred, and that request was made on behalf of the expected initiator. `SafetySpec` expresses an authentication goal, namely that the responder should successfully authenticate the apparent initiator as having requested the session.

Unfortunately even this simplified penetrator, which keeps too little state information to find the attack on the modified Otway-Rees protocol, has too large a state space. On a fast Sun compute-server, FDR 1 runs for an hour and a quarter, visiting some 5 million states, before crashing with an out-of-memory error. The process occupies approximately 250 MB of virtual memory when it terminates, most of that storage physically resident on this compute-server. It thus appears that substantial ingenuity is needed to build a penetrator that will explore a comprehensive set of possible behaviors, without demanding intractable amounts of memory.

2.4.4 Apparent Limitations of FDR 1

Our experience has identified four main limitations on the effective use of FDR 1:

- Considerable ingenuity is required to program an effective penetrator;

```
LivenessSpec = [] x : diff(Sigma, {| end |}) @ x -> LivenessSpec

Neutral = diff(Sigma, {| start, confirm |})
pragma channel a
System_vh = identify(Neutral, a, System_v)

SafetyRec(starts) =
        (a -> SafetyRec(starts))
    []
        ([] ir : starts @ confirm.ir -> SafetyRec(diff(starts,{ir})))
    []
        ([] ir : { i.r | i <- Initiator, r <- Responder} @
                start.ir -> SafetyRec(union(starts,{ir})))

SafetySpec = SafetyRec({})
```

Figure 43: Otway-Rees Correctness Specifications

- FDR 1's user interface is remarkably frail and uninformative (despite its attractive graphics);
- FDR 1—by contrast to FDR 2—offers little that would allow one to encapsulate abstract data;
- FDR 1—and apparently also FDR 2—does not allow one to select a search strategy to ensure that interesting sequences of actions will be explored first.

Programming the Penetrator We have already illustrated this point in connection with the Otway-Rees protocol. In this respect FDR 2 may have advantages. On 6 June 1996, after most of the work reported here, we visited Formal Systems (Europe); we were given an introduction to FDR 2 and to very recent work of A.W. Roscoe on "Building the Perfect Spy."

Roscoe's perfect spy exploits two main characteristics of FDR 2. First, FDR 2 incorporates a functional (i.e. purely applicative) programming language. This enables one to construct the spy for various protocols in a much more explicit, uniform way. Second, FDR 2 has a number of techniques to "compress" processes or reduce the size of their state spaces. In addition, FDR 2 does not construct the whole state space statically; rather, states are constructed dynamically, as the model-checker finds it necessary to visit them. As a consequence, inaccessible states need never be constructed at all.

It remains to be seen whether FDR 2's perfect spy will really be good enough in the sense of providing a tractable way of finding attacks on flawed protocols. However, during the 6 June meeting we proposed Otway-Rees as an example, with interesting results. Two minor, but apparently previously unknown, attacks were found against the (unmodified) Otway-Rees protocol.

1 $P \longrightarrow B$: M, B, B, X

2.1 $B \longrightarrow P$: $M, B, B, X, \{N_b, M, B, B\}_{K_b}$
(meant for S)

2.2 $P \longrightarrow S$: $M, B, B, \{N_b, M, B, B\}_{K_b}, \{N_b, M, B, B\}_{K_b}$
(apparently from B)

3 $S \longrightarrow B$: $M, \{N_b, K\}_{K_b}, \{N_b, K\}_{K_b}$

4 $B \longrightarrow P$: $M, \{N_b, K\}_{K_b}$
(meant for B itself, trapped by P)

Figure 44: A New Otway-Rees Attack

The first attack—shown in Figure 44—causes a party to open a session with itself, despite the fact that no such session has been requested. In this attack, X is a unrecognizable bitstring of the length expected for the initiator's nonce-pack. As a consequence, B is tricked into thinking that he's talking to himself when he really isn't. The attack depends on a number of assumptions:

- The server will not check that the nonces are fresh (in particular, that the second is distinct from the first);
- The server will not check that the initiator and responder are distinct;
- A message from B to itself may either be trapped by the penetrator, or else will be discarded if unexpected;
- Committing to unwanted sessions with oneself causes some harm, perhaps via denial of service where the total number of concurrent sessions may be limited.

Under these assumptions the attack could damage a system using the Otway-Rees protocol.

The second attack is a replay attack. In this attack the penetrator replays message 1 from a legitimate run of the protocol between A and B. As a consequence, B comes to believe that there are two sessions with A, whereas only one exists. It also assumes that the server will not check for freshness. Again, there may be some damage done if unwanted sessions cause a denial of service.

These attacks are minor: Otway-Rees could not possibly, in itself, allow B to conclude that A knows the new session key K [3]. Thus, it is unlikely that these attacks undermine any conclusion to which a BAN-style analysis would lead.

Controlling FDR 1 FDR 1 is remarkably demanding on its users, at least in the case of a nonexpert user. There are four main sources of the trouble:

- The parser is remarkably uninformative, almost willfully uninformative, in its error messages.

```
parse error (near line 0 of stdin)
(recently read ' \012              [| ')
```

Figure 45: FDR 1 Parser Error Message

- Compiling processes to the labelled transition system (lts) form is time-consuming, and the length depends on the size of the state and the size of the sets of events that may be communicated.
- There is no static type-checking, so type errors are invisible until running the refinement checker (after compilation).
- Liveness checks as shown above in Figures 15 and 43 had to be invented (or more likely reinvented) by the author; it was initially very difficult to garner any operational feedback on the behavior of a specification.

As an example of the behavior of the parser, consider the error message displayed in Figure 45. By searching the source file for a linefeed (ASCII code 10, octal 012) followed by fourteen spaces and the two characters "`[|`", we found that the parse error was on line 80 in the input file. Parentheses should have been used for grouping the operands to the parallel operator in a term of the form `P [| s1 |] Q [| s2 |] R`.[5] Surely a better error message could be imagined.

To take a different example, it transpires that parsing a file which stipulates that another file be `include`d has different results than parsing the file (without the `include` directive) and then issuing an `AddFile` command to add the contents of the second file. This took hours to discover.

In my experience, the FDR X window interface is pretty, but of relatively little practical value. A parser with informative error messages and a command line interface would be clearly preferable. Another great improvement would be incremental compilation: when a specification is updated, only the affected portions would be recompiled. Incremental compilation would save a great deal of time, but it may be technically infeasible.

Lack of Data Abstraction From one point of view, one of the great strengths of FDR is that CSP includes a number of mechanisms that may be used to achieve different kinds of abstraction. They include hiding, renaming, and parametrization.

However, from a different point of view, FDR 1 is lacking in an important abstraction mechanism. This is data abstraction. It would be highly convenient, in programming penetrators, to be able to encapsulate the data type of encrypted messages. As a consequence, the penetrator could be programmed to take any actions that did not break the abstraction barrier around the encrypted message data type. The data type abstraction mechanism would then

[5] The parallel operator in Hoare's original treatment of CSP is in fact associative, but the form used in FDR is not, because of its explicit synchronization set.

ensure that the penetrator cannot decrypt messages (or create fake encrypted messages) without possessing the key.

Without this sort of data abstraction, we must take care—throughout our specification—that all actions adhere to the convention that `Enc.`k`.`m behaves like an encryption of m under key k. This is tedious, and it impedes free-wheeling approaches to programming the penetrator. Depending on whether the programming language embedded in FDR 2 has a module system, this criticism of FDR 1 may not apply to FDR 2.

Inflexibility of Search Strategy One final problem with FDR, which applies to both versions, is the fact that the search strategy is embedded deep within the FDR engine. It is not available for control by the analyst. As a consequence, although we may have heuristic information about what paths through the state space are likely to be of interest, we cannot convey our information to FDR. This is in fact the case with the "unintended services" we described in Section 1.2. We would like to be able to concentrate our model checking on paths that frequently exercise the unintended services of the protocol at issue. We would then be likely to find attacks, if they exist, even when the overall number of paths is very large indeed. However, this approach seems impossible to achieve within FDR.

2.5 Plans for Future Work

Future work will encompass two main elements. First, we will continue to examine independently developed model checkers, especially FDR 2, to appraise their suitability for security protocol analysis.

Second, we intend to develop a very special purpose workbench for systematic state space exploration of security protocols. It will avoid many of the sophisticated algorithms involved in general purpose model checkers, but should instead offer tailored facilities to allow an analyst to explore attacks, and especially to explore the types of behavior most likely to generate significant attacks. In particular, the search strategy for the penetrator will be highly programmable, with the consequence that it will be possible to implement the "unintended services" insight described in Section 1.2.

The input language for the tool will be CAPSL, a language intended for protocol specification and analysis under development by J.K. Millen of MITRE. CAPSL has been the subject of substantial discussion with a wide range of leaders in the cryptographic protocol field, and is intended to serve as the input form for a variety of different approaches. By adopting CAPSL as the specification format, we hope to leverage on much outside work. In particular, we hope to be able to demonstrate the ability to compile a significant number of CAPSL specifications before the end of the fiscal year, carrying out simple search strategies to locate a number of attacks. Implementing the algebra needed for exploiting "unintended services" will be a goal for the next fiscal year.

References

[1] Martin Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. In *Proceedings, 1994 IEEE Symposium on Research in Security and Privacy*, pages 122–136. IEEE, IEEE Computer Society Press, 1994.

[2] Colin Boyd. Hidden assumptions in cryptographic protocols. *Proceedings of the IEE*, 137 Pt E(6):433–436, November 1990.

[3] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. Technical Report 39, Digital Systems Research Center, February 1989.

[4] Michael Burrows, Martin Abadi, and Roger M. Needham. Rejoinder to Nessett. *ACM Operating Systems Review*, 24(2):39–40, April 1990.

[5] Ulf Carlsen. Cryptographic protocol flaws. In *Proceedings 7th IEEE Computer Security Foundations Workshop*, pages 192–200. IEEE Computer Society, 1994.

[6] John Clark and Jeremy Jacob. A Library of Security Protocols with Attacks. Technical report, Dept. of Computer Science, University of York, June 1995.

[7] John Clark and Jeremy Jacob. On the Security of Hwang and Chen's SPLICE/AS Augmented Protocol. University of York, Department of Computer Science, June 1995.

[8] John Clark and Jeremy Jacob. On the security of recent protocols. *Information Processing Letters*, 56(3):151–155, November 1995.

[9] Dorothy Denning and G. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8), August 1981.

[10] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[11] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning about belief in cryptographic protocols. In Deborah Cooper and Teresa Lunt, editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, May 1990.

[12] C. A. R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–77, 1978.

[13] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[14] Gerard Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[15] Tzonelih Hwang and Yung-Hsiang Chen. On the security of SPLICE/AS: The authentication system in WIDE Internet. *Information Processing Letters*, 53:97–101, 1995.

[16] Tzonelih Hwang, Narn-Yoh Lee, Chuang-Ming Li, Ming-Yung Ko, and Yung-Hsiang Chen. Two attacks on Neuman-Stubblebine authentication protocols. *Information Processing Letters*, 53:103–107, 1995.

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

[18] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceeedings of* TACAS, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996.

[19] Gavin Lowe. Some new attacks upon security protocols. In *Proceedings of the Computer Security Foundations Workshop IX*. IEEE Computer Society Press, 1996.

[20] Wenbo Mao and Colin Boyd. Towards the formal analysis of security protocols. In *Proceedings of the Computer Security Foundations Workshop VI*, pages 147–158. IEEE Computer Society Press, 1993.

[21] Catherine Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–35, 1992.

[22] Catherine Meadows. A model of computation for the NRL protocol analyzer. In *Proceedings of the Computer Security Foundations Workshop VII*, pages 84–89. IEEE, IEEE Computer Society Press, 1994.

[23] J. K. Millen, S. C. Clark, and S. B. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, 13(2):274–288, February 1987.

[24] Jonathan K. Millen. The Interrogator model. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 251–60, 1995.

[25] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.

[26] Judy H. Moore. Protocol failures in cryptosystems. *Proceedings of the IEEE*, 76(5), May 1988.

[27] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), December 1978.

[28] Daniel M. Nessett. A critique of the Burrows, Abadi and Needham logic. *ACM Operating Systems Review*, 24(2):35–38, April 1990.

[29] D. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, January 1987.

[30] A. W. Roscoe. Intensional specifications of security protocols. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pages 28–38, 1996.

[31] Gustavus J. Simmons. How to (selectively) broadcast a secret. In *1985 IEEE Symposium on Security and Privacy*, pages 108–113, 1985.

[32] E. Snekkenes. Roles in cryptographic protocols. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1992.

[33] Paul Syverson. A taxonomy of replay attacks. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, pages 131–136. IEEE Computer Society Press, 1994.

[34] M. Tatebayashi, N. Matsuzaki, and D. Newman. Key distribution protocol for digital mobile communication systems. In G. Brassard, editor, *Advances in Cryptology:* CRYPTO *'89*, volume 435 of *Lecture Notes in Computer Science*, pages 324–331. Springer Verlag, 1990.