A Necessarily Parallel Attack *

Jonathan K. Millen
SRI International
Menlo Park, CA 94025
millen@csl.sri.com

Abstract

An artificial protocol called the "ffgg" protocol is constructed, with a parallel attack exposing a secret data item. It is proved that a parallel attack is necessary, because the protocol is shown to be secure under non-parallel attacks. We use an inductive approach in the PVS verification environment.

1 Introduction

Model checking has proved to be a successful way to find vulnerabilities in cryptographic protocols. See, for example, [3, 4, 7]. If a model checker fails to find an attack, however, it may only mean that there is no attack on the particular finite model of the system that was analyzed. Under certain restrictive assumptions about the protocol, Lowe has shown that it is sufficient to analyze a "small system" with one honest agent in each role, each of whom can run the protocol just once with the other honest agents [1]. Some upper bounds are given by Stoller [8].

The purpose of this paper is to give an example of a flawed protocol with the property that, to find an attack, it is necessary to analyze a system with at least two processes running the same role for the same principal. Furthermore, the two processes must run concurrently; that is, the protocol is secure if the two processes are serialized. An attack requiring this type of role concurrency is called a *parallel* attack. An example of a parallel attack was given for the "Π" protocol in [11]. Parallel attacks are significant because state exploration techniques encounter a combinatorial explosion with concurrent processes that is avoided if they can be serialized.

The existence of such attacks is not in question; the novel result here is the proof that sometimes a parallel attack is *necessary,* i.e., that the protocol is secure against non-parallel attacks. This is not the case for the Π protocol, which has other attacks.

We have constructed an artificial protocol called the "ffgg" protocol that has a parallel attack, and prove that it is secure against non-parallel attacks. The virtue of this artificial protocol is that it is apparent how to generalize the construction in such a way as to produce protocols that require higher-order parallel attacks; that is, attacks with three or more concurrent strands of the same principal in the same role. (The term "strand" comes from [10]; it refers to the activity of a single process in a protocol session.)

To show that a parallel attack is necessary, we use a technique that exemplifies the advantages of inductive proofs and verification environments. The idea is to attempt a proof that the protocol is secure as it stands, and reduce the proof to one remaining case that fails only if role concurrency (necessary for a parallel attack) is allowed. The details of that last case clearly exhibit how the attack is set up.

The proof technique is based primarily on Paulson's work [6], but it borrows the "ideal" concept from [9]. The proof was constructed and checked using the PVS verification environment [5].

2 The ffgg Protocol

In this protocol, A and B are agents (principals possessing a public/private key pair), N_1 and N_2 are nonces, M is a secret of the same field length as a nonce, and PKB is B's public key.

1. $A \rightarrow B : A$
2. $B \rightarrow A : B, N_1, N_2$
   $T = \{N_1, N_2, M\}_{PKB}$
3. $A \rightarrow B : A, T\%\{N_1, X, Y\}_{PKB}$
4. $B \rightarrow A : N_1, X, \{X, Y, N_1\}_{PKB}$

* Supported by DARPA through the Air Force Research Laboratory under Contract F30602-98-C-0258

This protocol description makes use of the Casper %-notation introduced by Lowe [2]. In general, a message field $u\%v$ is constructed by the sender according to the term u but is interpreted by the receiver according to v. This typically happens when one party or the other knows less about the structure of the message field.

In this case, when B receives message 3, B checks N_1. It also extracts the next field, but does not care whether it matches N_2; it just saves it as X.

The use of *PKB* rather than *PKA* in the last message is intentional, though odd-looking. We do not claim that this protocol is suitable for any practical application, only that it lends itself to an interesting theoretical result.

We called this the "ffgg" protocol because the responder B has two state transitions: the first, or f-transition, is to reply to message 1 with message 2, and the second, or g-transition, is to reply to message 3 with message 4. In the attack scenario, there is another B responder doing f' and g' transitions, and these are interleaved concurrently with f and g in the pattern $ff'gg'$.

2.1 The Parallel Attack

A message-modification attack that exposes the secret data field M is presented below.

An agent identifier in parentheses indicates interference by the attacker: if the source is in parentheses, the message has been forged or modified by the attacker. If the destination is in parentheses, the message is intercepted before it reaches the named destination.

There are two responder processes running for agent B; the second process is associated with primed symbols B', N_1', N_2'. Note that, because the second responder process is running on behalf of the same agent B, it still uses the same public key *PKB*.

1. $A \rightarrow B : A$
   - $1'.\ (A) \rightarrow B' : A$

$2a.\ B \rightarrow (A) : N_1, N_2$
   - $2'.\ B' \rightarrow (A) : N_1', N_2'$

$2b.\ (B) \rightarrow A : N_1, N_1'$

3. $A \rightarrow B : \{N_1, N_1', M\}_{PKB}$
4. $B \rightarrow (A) : N_1, N_1', \{N_1', M, N_1\}_{PKB}$
   - $3'.\ (A) \rightarrow B' : \{N_1', M, N_1\}_{PKB}$
   - $4'.\ B' \rightarrow (A) : N_1', M, \{M, N_1, N_1'\}_{PKB}$

This attack is illustrated in Figure 1. The secret M is exposed in the last message from the second B strand. Having shown that there is a parallel attack, we must now establish that parallelism is a necessary feature of any successful attack.

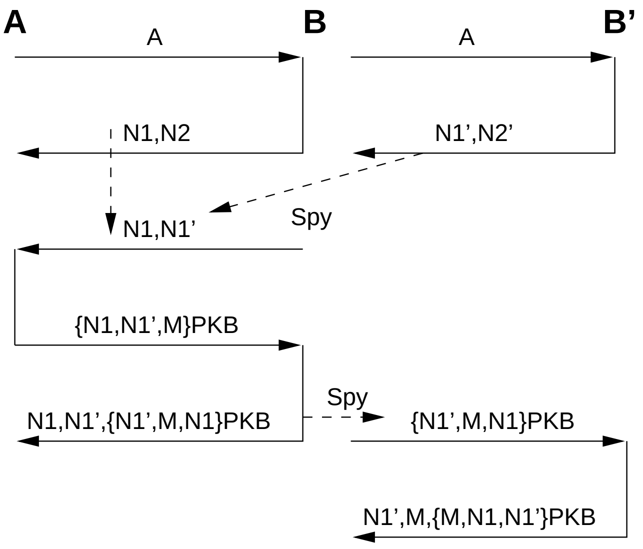

Figure 1. Parallel attack

2.2 The Modeling Approach

We adopt Paulson's representation of the network state as a trace of message events. The protocol is captured by rules that permit message events to be added to a trace. Most of the rules represent state transitions by agents following a specified role in the protocol. Another rule represents the fabrication of message events by the "Spy." In our PVS model, a trace is a LISP-like list, and message events are added on the left by the `cons` operation. (The state of each process is implicit in the message history, so that the trace need only contain messages and not state events, but state events could have been used if we had needed them.)

The modeling task begins by defining abstract data types for message fields and message events. Then we define and prove the properties of the basic operations on sets of message fields: parts, analz and synth. The reader is referred to [6] for a full discussion of these operators.

Briefly, parts(S) is the set of all subfields of fields in the set S, including components of concatenations and the plaintext of encryptions (but not the keys). analz(S) is the subset of parts(S) consisting of only those subfields that are accessible to an attacker. These include components of concatenations, and the plaintext of those encryptions where the inverse key is in analz(S). Finally, synth(S) is the set of fields constructible from S by concatenation and encryption using keys in S. The functions parts and synth are defined recursively. The function analz is defined as an inductive relation, meaning that it is the fixpoint of a monotone sequence of approximations. Inductive relation definitions are automatically validated by PVS.

Primitive data types agent, num, and pkey are used to construct message fields. The num type is for numbers that could be used for nonces or secrets. The pkey type is for public (or corresponding private) keys.

Message fields are constructed by type coercion from agents, nums, and pkeys, and also by concatenation and encryption. The message field constructors are:

> Agent(agent)
> Num(num)
> Pkey(pkey)
> Con(field, field)
> Ped(pkey, field)

A message event is a term of the form `Said`(*Src, Dest, Cont*) where *Src* and *Dest* are agents and the content *Cont* is a field. (The symbols *Src, Dest, Cont* are accessor functions.)

There are functions `pub` and `prv` on agent into pkey and a symmetric relation `invkey` such that `invkey(pub(`A`))` `= prv(`A`)`. The use of keys to decrypt terms encrypted with their inverses is implicit in the definition of analz.

With this setup, a protocol is defined recursively as a predicate accepting the legal traces of a network containing protocol sessions. As a recursive predicate, it decides when a new message event may be added to an existing legal trace. The top level of the definition is the function:

```
ffgg(Q: trace): RECURSIVE bool
= CASES Q OF
  cons(E, H): ffgg(H) AND
    (Fake(E, H, (initial)) OR
     a1(E, H) OR
     b2(E, H) OR
     a3(E, H) OR
     b4(E, H)),
  null: TRUE
ENDCASES MEASURE length
```

A `MEASURE` is a termination function required in PVS for any recursive definition in PVS to ensure that it is well-defined. The functions `a1, b2, a3, b4` are the individual state transition rules. They are shown in Appendix A.

The first case, `Fake`, is the rule by which the attacker generates messages constructed from data gleaned from the message history. It is actually protocol-independent. Given a set I of fields known initially by the attacker, a `Fake` message must be of the form `Said(Spy, A, X)` where *Spy* is a particular agent, A is any agent, and X must be an element of the set $\mathsf{synth}(\mathsf{analz}(\underline{H} \cup I))$ where H is the current history, and $\underline{H}$ is the set of message contents in the trace H.

The items I known initially by the attacker are the Spy's private key, all public keys, and all agents.

3 The Secrecy Policy

To define the secrecy policy for the protocol ffgg, we begin by identifying the set S_0 of *basic secrets*, namely, the secret field M and all private keys except that of the dishonest agent Spy. The secret M is modeled, for this exercise, as a constant that is not known to the Spy. (When the Spy acts in the A role, it uses a different field in place of M.)

The secrecy policy is that if $A \rightarrow B : X$ occurs in some trace, then $X \notin S_0$. (This is what we want, because if the Spy ever obtains a secret field X, it can transmit it as a message.)

The invariant that we will actually prove is that $X \notin I_k[S_0]$, where $I_k[S_0]$ is a Thayer-Herzog-Guttman *k-ideal*, a set of fields that includes S_0 and which is closed under concatenation with any fields and under encryption with keys in k [9].

For our purposes, we need k to be the set of keys whose corresponding inverse keys are not in S_0. It is necessary to protect this whole ideal because compromising any element of the ideal effectively compromises some element of S_0. It turns out that protecting this ideal is also sufficient.

It is convenient to name the complement of the ideal, which we call a coideal:

$$\mathsf{coideal}(S) = \{X | X \notin \mathsf{ideal}(S)\}$$

Applied to S_0, this defines the set of fields that are *public* with respect to the basic secrets S_0, i.e., fields whose release would not compromise anything in S_0.

The security requirement, now, is:

$$\mathtt{ffgg}(H) \Rightarrow \underline{H} \subseteq \mathsf{coideal}(S_0).$$

To make the induction work, we actually have to prove an apparently stronger statement, which we abbreviate as $\mathtt{safe}(S_0, I)(H)$:

$$\mathtt{ffgg}(H) \Rightarrow \mathsf{analz}(\underline{H} \cup I) \subseteq \mathsf{coideal}(S_0).$$

But this is really an equivalent statement, because any field in $\mathsf{analz}(\underline{H} \cup I)$ can be transmitted by the attacker as the content of the next message.

The proof depends in part on the following *analz-closure* lemma, which holds for any set S of fields:

$$\mathsf{analz}(\mathsf{coideal}(S)) = \mathsf{coideal}(S).$$

This says that the coideal is closed under attacker analysis, implying that protection of the k-ideal is sufficient (we remarked earlier that it was necessary). It was not hard to prove, but it should be generally useful, and it seems to be a new result; it was not given in [9].

Another useful result for the coideal is:

$$\mathsf{synth}(\mathsf{coideal}(S_0)) = \mathsf{coideal}(S_0).$$

This one depends on the primitiveness of elements of S_0.

4 The Proof

As remarked earlier, the proof that the ffgg protocol is secure except under a parallel attack begins as a proof that the protocol is secure, but the case splitting ends with a single case which cannot be proved. The hypotheses of that case exhibit a parallel attack. We know that this case cannot be disposed of because an instance of this case, a successful parallel attack, has already been exhibited.

To prove $\mathtt{safe}(S_0, I)(H)$ by induction, we check first that $\mathsf{analz}(I) \subseteq \mathsf{coideal}(S_0)$, by choice of I, then show that the safety property for H implies the safety property for a protocol trace of the form `cons(E, H)`, where E is a permissible message event.

It is sufficient, given the induction hypothesis on H, to show that $\underline{E} \in \mathsf{coideal}(S_0)$, since, if we write $E.H$ for `cons(E, H)`, we would have

$$\begin{aligned}\underline{E.H} \cup I &= \{\underline{E}\} \cup (\underline{H} \cup I)\\ &\subseteq \{\underline{E}\} \cup \mathsf{analz}(\underline{H} \cup I)\\ &\subseteq \mathsf{coideal}(S_0)\end{aligned}$$

and therefore, by monotonicity of analz and the analz-closure lemma mentioned above,

$$\mathsf{analz}(\underline{E.H} \cup I) \subseteq \mathsf{coideal}(S_0).$$

The proof that $\underline{E} \in \mathsf{coideal}(S_0)$ divides into cases according to the rule used to justify E: one of the protocol rules or the Fake rule. In some cases, the determination of whether the content of E is in the coideal depends on the prior message that was received and used to construct E, generating a further case split on the prior message rule, and so on.

4.1 The Last Case

The last remaining case is displayed in Appendix B. What is shown is a PVS sequent: it is a proof goal stating that the conjunction of the hypotheses, the formulas above the long turnstile `|-------`, implies the disjunction of the consequents, the formulas below it. If the case fails, as it must because the protocol is insecure, it is because there is an instance where the hypotheses are true and the consequents are false. Note the induction hypothesis in line [-10], where `(secrets)` is S_0 and `(initial)` is I.

The consequent `B = Spy` is essentially a hypothesis in this case that B is not the Spy; the case where B is the spy has been disposed of elsewhere.

The sequent shows messages from two B-strands, one identified with nonce Z and one identified with nonce Y. Message 2 and message 4 belong to the same strand (and the same process) if they have the same first nonce, because of the causal relationship in rule b4. We can show that these two strands are not serializable by examining the ordering of the message events.

An event E_1 precedes E_2 in a trace `cons(E2, H)` if E_1 is a member of H. We make use of a predicate `extends?` saying that one trace is an extension of another.

The membership and `extends?` hypotheses are pictured in Figure 2, and the implied message ordering in the two B strands are shown in Figure 3. Because message 4 in the Y strand is between messages 2 and 4 of the Z strand, we can see that these two B strands are not serializable. Consequently, any trace that disproves the theorem (i.e., any attack) must be parallel.

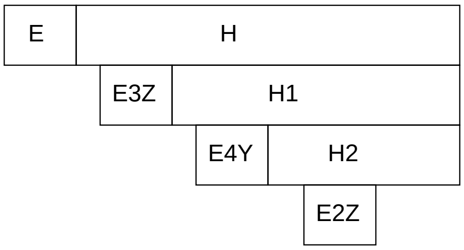

Figure 2. Trace extensions

4.2 Effort

The final proof (to the last case) was replayed by the proof checker in about nine minutes on a Pentium Pro. It consisted of a large number of mostly primitive steps, with some applications of separately-proved lemmas and a few invocations of general-purpose built-in strategies (e.g., "grind"). Many sequences of steps could have been repackaged into fewer strategies or avoided by applying grind-like steps more often. Construction of higher-level strategies to simplify future proofs is in progress.

The person-time required for the proof is difficult to mea-

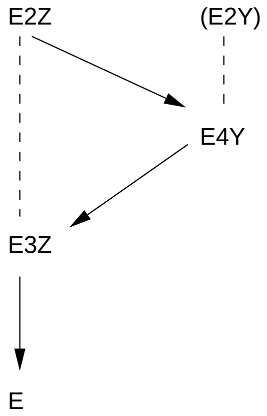

Figure 3. Event ordering

sure because it was a combination of specification activity, revisions of the protocol and proof approach, creation and proof of generally applicable lemmas and underlying theories, and the proof itself. The total level of effort was on the order of weeks, but the proof steps could be entered manually with full understanding of the thought behind the proof in less than a day.

5 Conclusions

We have given an example protocol, the ffgg protocol, for which a parallel attack exists, and for which we have proved that a parallel attack is necessary to disclose a secrecy compromise. Specifically, any attack must have two non-serializable strands of the same agent playing the same role.

We needed an inductive approach to accomplish this proof because belief logics do not address secrecy, and state search methods are limited by finite bounds. The only exceptions are for protocols satisfying restrictive structural conditions that are not satisfied by our example.

I believe that the use of a verification environment was necessary to achieve the highest possible assurance in the result, as well as precision in stating it. The proof is complex enough so that I trust the machine-checked result much more than I would have trusted my own hand proof.

It is apparent from the structure of the messages that any degree of parallelism could be forced by inserting more nonces. There is a family of protocols $\mathrm{f}^n\mathrm{g}^n$ for which we conjecture (but have not yet proved) that n non-serializable B strands of the same agent are required for an attack. The nth protocol is:

1. $A \rightarrow B : A$
2. $B \rightarrow A : B, N_1, N_2, ..., N_n$
   $T = \{N_1, N_2, ..., N_n, M\}_{PKB}$
3. $A \rightarrow B : A, T\%\{N_1, X_2, ..., X_n, Y\}_{PKB}$
4. $B \rightarrow A : N_1, X_2, \{X_2, ..., X_n, Y, N_1\}_{PKB}$

Acknowledgement The motivation for this example and improvements in its presentation arose from helpful discussions with Grit Denker.

References

[1] G. Lowe, "Towards a completeness result for model checking of security protocols," *1998 Computer Security Foundations Workshop,* IEEE Computer Society, 1998.

[2] G. Lowe, "Casper: a compiler for the analysis of security protocols," *J. Computer Security* 6(1), 1998, pp. 53-84.

[3] W. Marrero, E. Clarke, and S. Jha, "Model checking for security protocols," Carnegie Mellon University, CMU-CS-97-139, 1997.

[4] J. C. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using Murphi," *IEEE Symposium on Security and Privacy,* IEEE Computer Society, 1997, pp. 141-151.

[5] N. Shankar, S. Owre, J. Rushby, D. Stringer-Calvert, "PVS Prover Guide," Version 2.2, SRI International, September 1998.

[6] L. Paulson, "The inductive approach to verifying cryptographic protocols," *J. Computer Security* 6(1), 1998, pp. 85-128.

[7] A. W. Roscoe, "Modelling and verifying key exchange protocols using FDR," *1995 Computer Security Foundations Workshop,* IEEE Computer Society, 1995, pp. 98-107.

[8] S. D. Stoller, "A reduction for automated verification of authentication protocols," Technical Report 520, Computer Science Department, Indiana University, 1998.

[9] F. J. Thayer, J. Herzog, and J. Guttman, "Honest ideals on strand spaces," *1998 Computer Security Foundations Workshop,* IEEE Computer Society, 1998, pp. 66-77.

[10] F. J. Thayer, J. Herzog, and J. Guttman, "Strand spaces: why is a security protocol correct?" *1998 IEEE Symposium on Security and Privacy,* IEEE Computer Society, 1998, pp. 160-171.

[11] T. Woo and S. Lam, "A lesson on authentication protocol design," *ACM Operating Systems Review* 28(3), July 1994, pp. 24-37.

A Appendix: ffgg Protocol Specification

The protocol ffgg is a recursive boolean function on traces, which are lists of message events. It accepts traces whose chronologically last (left-hand) event is added according to one of the rules a1, b2, a3, b4, or Fake. The Fake rule is explained in the text; it is protocol-independent.

The specified version of the protocol includes the message numbers as the first field of each message. Each rule says that a message E of a given format may be added provided that certain conditions on the prior trace H are satisfied, usually including the existence of a prior message.

In most of the rules, we have followed Paulson's convention of permitting messages to be repeated. This does not affect secrecy. For b4, however, there is a condition that no prior message 4 has been sent in the same strand.

The full specification also contains type declarations and definitions of `initial` and `secrets`. These have been omitted here since they were discussed in the text. Recall that `Num` converts numbers to messages, so that it is used for 1, 2, 3, 4 and M as well as nonces. The freshness of N_1 and N_2 is guaranteed by two hypotheses in the b2 rule. There is also an assumption (not shown) that M is not one of 1, 2, 3, 4.

The predicate `parts_tr` is the obvious extension of `parts` to traces: `parts_tr(H)(X)` means that X is part of the content of some message event in H.

```
a1(E, H): bool =             % A -> B: 1 A
  EXISTS (A, B):
  A /= B AND
  E = Said(A, B, Con(Num(1), Agent(A)))

b2(E, H): bool =             % B -> A: 2 B N1 N2
      EXISTS (A, B, C, N1, N2):
  E = Said(B, A, Con(Num(2), Con(Agent(B),
               Con(Num(N1), Num(N2)))))
  AND member(Said(C, B, Con(Num(1), Agent(A))), H)
  AND NOT parts_tr(H)(Num(N1))
  AND NOT parts_tr(H)(Num(N2))
  AND N1 /= M AND N2 /= M

a3(E, H): bool =             % A -> B: 3 A {N1 N2 M}PB
  EXISTS (A, B, C, N1, N2, Ma):
  E = Said(A, B, Con(Num(3), Con(Agent(A), Ped(pub(B),
      Con(Num(N1), Con(Num(N2), Num(Ma)))))))
  AND (B /= Spy OR Ma /= M)
  AND NOT parts_tr(H)(Num(Ma))
  AND member(Said(C, A, Con(Num(2),
      Con(Agent(B), Con(Num(N1), Num(N2))))), H)

b4(E, H): bool =             % B -> A: 4 N1 N2 {N2 M N1}PB
  EXISTS (A, B, C, X, Y, N1, N2):
  E = Said(B, A, Con(Num(4), Con(Num(N1),
      Con(X, Ped(pub(B), Con(X, Con(Y, Num(N1))))))))
  AND member(Said(C, B, Con(Num(3), Con(Agent(A),
      Ped(pub(B), Con(Num(N1), Con(X, Y)))))), H)
  AND member(Said(B, A, (Con(Num(2), Con(Agent(B),
      Con(Num(N1), Num(N2)))))), H) % B checks A and N1
  AND NOT EXISTS Z:          % B has not sent this before
      member(Said(B, A, Con(Num(4), Con(Num(N1), Z))), H)
```

```
ffgg(Q: trace): RECURSIVE bool =
  CASES Q OF
  cons(E, H) : ffgg(H) AND
      (Fake(E, H, initial)
      OR a1(E, H)
      OR b2(E, H)
      OR a3(E, H)
      OR b4(E, H)),
  null : TRUE
  ENDCASES MEASURE length
```

B Appendix: The Last Case

```
Main_lemma.5.3.1.3.2.1.1.1.5.2.2.2.1.5.1.3.1.3 :

[-1]    E =
        Said(B, A,
             Con(Num(4),
                 Con(Num(Z),
                     Con(X, Ped(pub(B),
                         Con(X, Con(Y, Num(Z)))))))
[-2]    ffgg(cons(E, H))
[-3]    member(E2Z, H2)
[-4]    extends?(H, cons(E4Y, H2))
[-5]    E2Z =
        Said(B, A, Con(Num(2),
             Con(Agent(B), Con(Num(Z), Num(N2)))))
[-6]    E4Y =
        Said(B, A2,
             Con(Num(4),
                 Con(Num(N1),
                     Con(Num(Z),
                         Ped(pub(B), Con(Num(Z),
                             Con(X, Num(N1)))))))
[-7]    extends?(H1, cons(E4Y, H2))
[-8]    extends?(H, cons(E3Z, H1))
[-9]    ideal((secrets))(X)
[-10]    safe((secrets), (initial))(H)
  |-------
[1]   B = Spy

Rule?
```