Analysing protocols subject to guessing attacks

Gavin Lowe

November 11, 2001

Abstract

In this paper we consider guessing attacks upon security protocols, where an intruder guesses one of the values used (typically a poorly-chosen password) and then seeks to verify that guess. We formalise such attacks, and in particular the way in which the guess is verified. We then describe how to model such attacks within the process algebra CSP, so that they can be detected using the model checker FDR, and illustrate our technique on some examples.

1 Introduction

It is well known that people tend to choose poor passwords [8]. Such poorly-chosen passwords are vulnerable to *guessing attacks*, where an intruder observes some information based upon the password, tries to guess the password, and is able to verify the guess in some way.

For example, consider the following simple protocol, which aims to authenticate a user (or client) a to a server b using a password $passwd(a, b)$:

Message 1. $a \to b \ : \ a, b$
Message 2. $b \to a \ : \ nb$
Message 3. $a \to b \ : \ \{|nb|\}_{passwd(a,b)}$

Here nb is a nonce; we use the notation $\{|m|\}_k$ to represent m encrypted with k. An intruder could overhear this exchange and attempt a guessing attack: if he guesses that the password is "`aardvark`", he could calculate $\{|nb|\}_{\texttt{aardvark}}$ and compare it with the ciphertext from message 3; if they match, this verifies the guess (with high probability); if they don't match, he can guess again. Of course, this guessing can be automated, trying each word in some list in turn. If the password was chosen by a person, there is a fair probability that it will appear in a suitable dictionary, and so this attack is feasible. On the other hand, if it was generated by a good random number generator from a large space of passwords, then guessing will be infeasible. We will be considering only the former, feasible, form of guessing attack in this paper.

Note that this guessing attack can be performed off-line, to avoid detection of incorrect guesses. Contrast this with an on-line attack, where the intruder sends messages based upon guesses, where incorrect guesses will be detected and probably logged, possibly leading to additional defences being mounted, such as blocking further attempts. We will deal only with off-line guessing attacks in this paper.

In this paper we study how protocols subject to guessing attacks can be analysed. There has been much work in recent years on protocol analysis, but this aspect has been largely ignored. In Section 2 we produce a general model for such guessing attacks and define how a guess can be verified. In Section 3 we adapt this general model to the CSP/FDR approach to protocol analysis, and thus show how guessing attacks can be detected automatically. In Section 4 we illustrate our approach by applying it to a couple of well-known protocols that aim to be resistant to guessing attacks. We sum up in Section 5.

2 Formalising guessing attacks

A guessing attack consists of the intruder guessing a value g, and then verifying that guess in some way. The verification will be by the intruder using g to produce a value v, which we call the *verifier*; the verifier will demonstrate that the guess was correct, i.e., an incorrect guess would not have lead to this value. This verification can take a number of different forms: (1) the intruder knew v initially or has seen v during the protocol run; (2) the intruder produced v in two distinct ways from g; or (3) v is an asymmetric key, and the intruder knows v's inverse from somewhere. We consider some examples, below, to illustrate these forms of verification:

Example 1 If the protocol directly discloses v and $\{|v|\}_g$ to the intruder (as in the example in Section 1, with $v = nb$ and $g = passwd(a, b)$), then the guess of g allows the intruder to decrypt the second component to obtain v again, thereby verifying the guess; alternatively, the guess of g allows the intruder to encrypt v with g to obtain $\{|v|\}_g$ again.

Example 2 If the protocol directly discloses $\{|g|\}_g$ to the intruder, then if the intruder guesses g, he can encrypt g with itself to obtain $\{|g|\}_g$ again, thereby verifying the guess; alternatively, the intruder could decrypt $\{|g|\}_g$ with g to obtain g again.

Example 3 If the protocol allows the intruder to deduce $\{|v, v|\}_g$ directly, then the guess of g allows the intruder to obtain (v, v), from which he can obtain v in two different ways, thereby verifying the guess.

Example 4 If the intruder can deduce $PK(a)$ and $\{|SK(a)|\}_g$,[1] then the guess of g allows the intruder to obtain $SK(a)$; he can then verify the guess by seeing that this key is the inverse of $PK(a)$, by encrypting some arbitrary text with one key and decrypting with the other.

[1] These two terms are included in the file `/etc/publickey` under SunOS-4.0.

Note that the guess and attempted verification can take place at any point during the history of a protocol: when we search for successful guessing attacks, we will consider all possible ways the intruder could interact with the protocol, intercepting and sending messages, before the guess itself.

Let *Fact* contain all submessages of messages in the protocol[2]. Our definition of guessing attacks is largely independent of the form of the set *Fact*, but in the examples of this section we take *Fact* to be built up from some set of atoms using pairing and encryption.

We assume that the intruder can directly deduce new facts from facts he already knows (without guessing). We model these deductions by a one-step deduction relation:

$$\vdash \ :: \ \mathbb{P}(\mathit{Fact}) \times \mathit{Label} \times \mathit{Fact}$$

The idea is that $S \vdash_l f$ represents that f can be deduced from the set S in a single step; the label l indicates how the deduction is done, and is necessary to capture the concept of deriving a fact in two different ways, mentioned above.

Again, our definition of guessing attacks is largely independent of the details of this relation, but for the examples of this section we will use a fairly standard derivation relation, defined as follows, where f, f' range over facts, and k ranges over keys:

$$\{f, f'\} \vdash_{\mathsf{pair}} (f, f'), \qquad \{(f, f')\} \vdash_{\mathsf{fst}} f, \qquad \{(f, f')\} \vdash_{\mathsf{snd}} f',$$
$$\{f, k\} \vdash_{\mathsf{enc}} \{\!|f|\!\}_k, \qquad \{\{\!|f|\!\}_k, k^{-1}\} \vdash_{\mathsf{dec}} f.$$

We write $IK \models_{tr} IK'$ if the intruder can deduce the set of facts IK' from the set of facts IK, using zero or more deduction steps ("IK" stands for "intruder's knowledge"); the sequence tr records the trace of deductions the intruder performs. We define $\models$ by the following rules:

$$IK \models_{\langle\rangle} IK,$$
$$S \subseteq IK \wedge S \vdash_l f \wedge IK \cup \{f\} \models_{tr} IK' \ \Rightarrow \ IK \models_{\langle S \vdash_l f\rangle ^\frown tr} IK'.$$

If the intruder can deduce f from some subset S of his knowledge, then he can add f to his knowledge, and use it to deduce more facts. We drop the subscript where it is unimportant.

We now define what it means for the intruder to verify a guess g using verifier v from knowledge IK. Firstly, from this knowledge and the guess, the intruder must be able to reach some knowledge IK':

$$IK \cup \{g\} \models_{tr} IK'. \tag{1}$$

One of those deductions must produce the verifier v:

$$S \vdash_l v \ \mathsf{in} \ tr. \tag{2}$$

[2] We use the word "fact" for these, preferring to reserve the word "message" for an actual message of the protocol.

However, this deduction must really depend upon the knowledge of g: it must be impossible to obtain the information S necessary for the deduction without knowing g:

$$\nexists\, IK'' \bullet (IK \models IK'' \supseteq S). \tag{3}$$

Further, either the intruder can deduce v in a second way (first disjunct below) (the conjunct $(S, l) \neq (S', l')$ ensures this really is a *different* way of deducing v); or the intruder already knew v in the initial state, or guessed it (second disjunct); or v is an asymmetric key whose inverse the intruder knows (third disjunct):

$$\begin{aligned} & S' \vdash_{l'} v \text{ in } tr \wedge (S, l) \neq (S', l') \\ \vee\ & v \in IK \cup \{g\} \\ \vee\ & v \in ASYMMETRIC\_KEYS \wedge v^{-1} \in IK'. \end{aligned} \tag{4}$$

We will require a fifth condition, designed to prevent certain false attacks, which we will describe below. First, we illustrate how this definition applies to the above examples. Our descriptions concentrate on conditions (1), (2) and (4), which capture the deductions the intruder makes; we leave it to the reader to check condition (3) in each case.

Example 1 Suppose $IK = \{v, \{\!|v|\!\}_g\}$; then $IK \cup \{g\} \models_{\langle \{\{\!|v|\!\}_g, g\} \vdash_{\mathsf{dec}} v \rangle} IK \cup \{g\}$ satisfies (1); the deduction $\{\{\!|v|\!\}_g, g\} \vdash_{\mathsf{dec}} v$ satisfies equation (2); and $v \in IK \cup \{g\}$ so equation (4) is satisfied.

Example 2 Suppose $IK = \{\{\!|g|\!\}_g\}$, and take $v = g$; then $IK \cup \{g\} \models_{\langle \{\{\!|g|\!\}_g, g\} \vdash_{\mathsf{dec}} g \rangle} IK \cup \{g\}$ satisfies (1); the deduction $\{\{\!|g|\!\}_g, g\} \vdash_{\mathsf{dec}} g$ satisfies equation (2); and $v \in IK \cup \{g\}$ so equation (4) is satisfied.

Example 3 Suppose $IK = \{\{\!|v, v|\!\}_g\}$; then

$$IK \cup \{g\} \models_{\langle \{\{\!|v,v|\!\}_g, g\} \vdash_{\mathsf{dec}} (v,v),\ \{(v,v)\} \vdash_{\mathsf{fst}} v,\ \{(v,v)\} \vdash_{\mathsf{snd}} v \rangle} IK \cup \{g, (v, v), v\}$$

satisfies (1); the deduction $\{(v, v)\} \vdash_{\mathsf{fst}} v$ satisfies (2); and the deduction $\{(v, v)\} \vdash_{\mathsf{snd}} v$ satisfies (4), noting that this deduction is different from the one for (2).

Example 4 Suppose $IK = \{PK(a), \{\!|SK(a)|\!\}_g\}$, and take $v = SK(a)$; then $IK \cup \{g\} \models_{\langle \{\{\!|SK(a)|\!\}_g, g\} \vdash_{\mathsf{dec}} SK(a) \rangle} IK \cup \{g, SK(a)\}$ satisfies (1); the deduction $\{\{\!|SK(a)|\!\}_g, g\} \vdash_{\mathsf{dec}} SK(a)$ satisfies (2); and the fact that $(SK(a))^{-1} = PK(a) \in IK$ satisfies the third disjunct of (4).

As mentioned above, we need another condition in order to prevent certain false attacks. Here's an example to show why. Let $IK = \{\{\!|v|\!\}_g\}$; then

$$IK \cup \{g\} \models_{\langle \{\{\!|v|\!\}_g, g\} \vdash_{\mathsf{dec}} v,\ \{v\} \vdash_{\mathsf{pair}} (v,v),\ \{(v,v)\} \vdash_{\mathsf{fst}} v \rangle} IK \cup \{g, v, (v, v)\}$$

satisfies (1); the deduction $\{\{\!|v|\!\}_g, g\} \vdash_{\mathsf{dec}} v$ satisfies (2); and the deduction $\{(v, v)\} \vdash_{\mathsf{fst}} v$ satisfies (4). However, this is not a valid guessing attack, because the intruder obtains no verification of the guess: the fact that he derived v from (v, v) does not give him that verification, because he built (v, v) himself.

Our extra condition will prohibit deduction steps that simply reverse previous steps, in the way the step $\{(v, v)\} \vdash_{\mathsf{fst}} v$ reverses $\{v\} \vdash_{\mathsf{pair}} (v, v)$. We define a relation undoes that describes when a deduction step undoes another. The relation is defined by the following rules and their symmetric opposites:

$$
\begin{array}{rcl}
\{f, f'\} \vdash_{\mathsf{pair}} (f, f') & \mathsf{undoes} & \{(f, f')\} \vdash_{\mathsf{fst}} f, \\
\{f, f'\} \vdash_{\mathsf{pair}} (f, f') & \mathsf{undoes} & \{(f, f')\} \vdash_{\mathsf{snd}} f, \\
\{(f, f')\} \vdash_{\mathsf{fst}} f & \mathsf{undoes} & \{f, f'\} \vdash_{\mathsf{pair}} (f, f'), \\
\{(f, f')\} \vdash_{\mathsf{snd}} f & \mathsf{undoes} & \{f, f'\} \vdash_{\mathsf{pair}} (f, f'), \\
\{f, k\} \vdash_{\mathsf{enc}} \{\!|f|\!\}_k & \mathsf{undoes} & \{\{\!|f|\!\}_k, k^{-1}\} \vdash_{\mathsf{dec}} f, \\
\{\{\!|f|\!\}_k, k^{-1}\} \vdash_{\mathsf{dec}} f & \mathsf{undoes} & \{f, k\} \vdash_{\mathsf{enc}} \{\!|f|\!\}_k.
\end{array}
$$

Our fifth condition, then, says that, neither of the deductions from conditions (2) or (4) undoes one of the deductions in condition (1):

$$
\begin{array}{l}
\forall (S'' \vdash_{l''} v'') \text{ in } tr \bullet \neg(S \vdash_l v \text{ undoes } S'' \vdash_{l''} v'') \wedge \\
\qquad\qquad\qquad\qquad\qquad \neg(S' \vdash_{l'} v' \text{ undoes } S'' \vdash_{l''} v'').
\end{array} \tag{5}
$$

We leave it to the reader to check that this condition is satisfied by each of the examples above.

It is worth pointing out that the third disjunct of condition (4) is necessary, in particular for Example 4. The reader may think that, having obtained $PK(a)$ and $SK(a)$, the intruder can verify the guess by picking an arbitrary v, encrypting it with $PK(a)$, and then decrypting with $SK(a)$. However, these two deductions, $\{v, PK(a)\} \vdash_{\mathsf{enc}} \{\!|v|\!\}_{PK(a)}$ and $\{\{\!|v|\!\}_{PK(a)}, SK(a)\} \vdash_{\mathsf{dec}} v$, do not satisfy condition (5).

Above we have assumed that the intruder guesses only a single value. It is straightforward to adapt the above conditions to the case where the intruder can simultaneously guess multiple values: simply replace the $\{g\}$ above by some set G of guessed values. Of course, the complexity of any such guessing attack is exponential in the number of values that need to be correctly guessed, so it is reasonable to place some small bound upon the size of G.

3 Modelling guessing attacks in CSP

In this section, we describe how guessing attacks can be modelled using the process algebra CSP and discovered using the model checker FDR, by adapting the standard intruder model from [10].

We take a slightly different structure for the set *Fact* of facts from that considered in Section 2: we use sequences of facts rather than pairs, for efficiency; and we include a constructor corresponding to Vernam encryption (bit-wise exclusive-or), to extend the scope of our techniques (in particular, the protocol we consider in Section 4.2 uses Vernam encryption).

The model of the intruder is built around a set *Deductions* representing the deductions the intruder can make. Each element $(f, l, S) \in \mathit{Deductions}$ corresponds to a deduction of the form $S \vdash_l f$: if the intruder knows the set of facts S, then he can deduce the fact f using rule l. In fact, for reasons of efficiency, we use slightly different deductions to those in Section 2, combining decryption and splitting into a single rule, and combining sequencing and encryption into a single rule; we also include deductions corresponding to Vernam encryption and decryption. It is worth giving the definition of the undoes relation for these latter deductions:

$$\begin{aligned}
&\{f, f'\} \vdash_{\mathsf{vernEnc}} f \oplus f' \ \ \mathsf{undoes}\ \ \{f \oplus f', f\} \vdash_{\mathsf{vernDec}} f' \\
&\{f \oplus f', f\} \vdash_{\mathsf{vernDec}} f' \ \ \mathsf{undoes}\ \ \{f, f'\} \vdash_{\mathsf{vernEnc}} f \oplus f' \\
&\{f \oplus f', f\} \vdash_{\mathsf{vernDec}} f' \ \ \mathsf{undoes}\ \ \{f \oplus f', f'\} \vdash_{\mathsf{vernDec}} f
\end{aligned}$$

The third term is perhaps not obvious.

The intruder process has two principle states, *INTRUDER* and *INTRUDER'*, corresponding to whether the intruder is not or is currently in a guessing phase. If the intruder is able to use the guess to deduce a fact that he already knows—as captured by the earlier conditions—then that will verify the guess.

Each process is parameterised by the intruder's knowledge, *IK*; this knowledge will change as the intruder sees or deduces new facts. We need to control the deductions performed: we need to prevent the intruder from repeating deductions, to ensure that the two ways in which a fact is deduced are indeed different, as required by condition (4); and we need to prevent the intruder from undoing deductions, as required by condition (5). Therefore the intruder processes will also be parameterised by a set *deds* of deductions, representing those deductions that can still be performed.

In the non-guessing state the intruder can: (1) hear a message on the network, and add the components of that message to his knowledge (a message is modelled as a sequence of facts; $components(m)$ gives the corresponding set of facts); (2) say a message (i.e. send it on the network) provided he knows all the components; (3) perform a deduction (f, l, S), provided he knows all of S; the fact f is added to his knowledge; that deduction and all its undoing deductions are removed from the set *deds* of deductions that are still allowed; (4) signal, via the channel *leak*, if he has learnt a secret; (5) guess some guessable value g, and enter the guessing state.

$$\begin{array}{l}
INTRUDER(IK, deds) = \\
\quad hear?m \rightarrow INTRUDER(IK \cup components(m), deds) \\
\quad \Box \\
\quad (\Box\, m : Message, components(m) \subseteq IK \bullet \\
\qquad say.m \rightarrow INTRUDER(IK, deds)) \\
\quad \Box \\
\quad (\Box (f, l, S) : deds, S \subseteq IK \bullet \\
\qquad infer.(f, l, S) \rightarrow INTRUDER(IK \cup \{f\}, deds \ominus (f, l, S))) \\
\quad \Box \\
\quad (\Box\, f : SECRETS \cap IK \bullet leak.f \rightarrow INTRUDER(IK, deds)) \\
\quad \Box \\
\quad guess?g \rightarrow INTRUDER'(IK \cup \{g\}, deds, g),
\end{array}$$

where the operator $\ominus$ performs the removal referred to in (3) above:

$$
\begin{aligned}
&deds \ominus (f, l, S) \mathrel{\widehat{=}} \\
&\quad deds - (\{(f, l, S)\} \cup \{(f', l', S') \mid (f', l', S') \;\mathsf{undoes}\; (f, l, S)\}).
\end{aligned}
$$

Recall that we are interested in *off-line* guessing attacks; we will therefore not allow any network events during the guessing period. During the guessing period, the intruder can continue to perform deductions. If one of these deductions produces a fact that he already knows, or the inverse of a key he already knows, then this verifies his earlier guess:

$$
\begin{aligned}
&INTRUDER'(IK, deds, g) = \\
&\quad \Box (f, S) : deds, S \subseteq IK \bullet \\
&\qquad infer.(f, l, S) \rightarrow \\
&\qquad \text{if } f \in IK \vee f \in ASYMMETRIC\_KEYS \wedge f^{-1} \in IK \\
&\qquad \text{then } verify.(f, g) \rightarrow INTRUDER(IK, deds \ominus (f, l, S)) \\
&\qquad \text{else } INTRUDER'(IK \cup \{f\}, deds \ominus (f, l, S), g).
\end{aligned}
$$

We instantiate the intruder's knowledge with some suitable initial value IIK, initialise *deds* with the set of all deductions, and hide the deductions. At many intermediate states, there will be several different deductions available; the intruder does not lose any abilities by performing as many deductions as possible, so it makes sense to force him to perform them all; further, the order of the deductions makes no difference, so picking one particular order will reduce the state space. The function *chase* does this: it forces internal τ events to be performed, picking an arbitrary order for them:

$$
chase\,(INTRUDER(IIK, Deductions) \setminus infer)\,.
$$

The use of *chase* has a fortunate and necessary consequence: by forcing deductions to occur, the intruder will perform a guess only when no more deductions are available; this means that any subsequent deduction will really depend upon the guess, as required by condition (3).

We can incorporate the above intruder model into the standard CSP model of a system running the protocol under consideration. In order to find out if a guessing attack is possible, we can use FDR to find out if any *verify* event can occur. Of course, a successful guessing attack can also lead to breaches of secrecy (either of the password or other values disclosed in the attack) or failures of authentication; these can be tested for using the normal secrecy and authentication specifications.

It is straightforward to adapt the above intruder model to allow the intruder to guess multiple values, by allowing further *guess* events from the $INTRUDER'$ state (possibly subject to some maximum number).

Casper [5] is a compiler that produces CSP models of security protocols, suitable for checking using FDR, from more concise descriptions. We have extended Casper to model guessing attacks. We have tested it on the earlier examples, and used it in the examples of the following section, obtaining the expected results.

4 Examples

In this section we consider some examples, illustrating the techniques described earlier.

4.1 Encrypted key exchange

In [1], Bellovin and Merritt introduce the following key-establishment protocol, using a poorly-chosen password:

$$\begin{array}{lll}
\text{Message 1.} & a \to b & : \ a, \{\!|pk|\!\}_{passwd(a,b)} \\
\text{Message 2.} & b \to a & : \ \{\!|\{\!|k|\!\}_{pk}|\!\}_{passwd(a,b)} \\
\text{Message 3.} & a \to b & : \ \{\!|na|\!\}_k \\
\text{Message 4.} & b \to a & : \ \{\!|na, nb|\!\}_k \\
\text{Message 5.} & a \to b & : \ \{\!|nb|\!\}_k \ .
\end{array}$$

Here pk is an *asymmetric* key, and k is a symmetric session key. We have analysed this protocol using Casper and FDR, and found no attacks.

It is interesting to ask whether pk needs to be an asymmetric key. We changed it to a symmetric key in the model, and discovered an attack. The intruder can observe a normal run of the protocol between A and B, guess $passwd(A, B)$, and then verify the guess using A's nonce Na. FDR produces a long trace containing all the deductions the intruder can make as a result of the guess. However, the relevant events are:

$$\begin{array}{l}
guess.passwd(A, B) \\
infer.(\{\!|K|\!\}_{PK},\ Dec.1,\ \{\{\!|\{\!|K|\!\}_{PK}|\!\}_{passwd(A,B)}, passwd(A, B)\}) \\
infer.(PK,\ Dec.1,\ \{\{\!|PK|\!\}_{passwd(A,B)}, passwd(A, B)\}) \\
infer.(K,\ Dec.1,\ \{\{\!|K|\!\}_{PK}, PK\}) \\
infer.(Na,\ Dec.1,\ \{\{\!|Na, Nb|\!\}_K, K\}) \\
infer.(Na,\ Dec.1,\ \{\{\!|Na|\!\}_K, K\}) \\
verify.(Na,\ passwd(A, B)).
\end{array}$$

The intruder: decrypts the second message with $passwd(A, B)$ to obtain $\{\!|K|\!\}_{PK}$; decrypts the first message with $passwd(A, B)$ to obtain PK; decrypts $\{\!|K|\!\}_{PK}$ with PK to obtain K (this step is not possible if PK is an asymmetric key); decrypts the third message to obtain Na; and decrypts the fourth message to obtain Na again; the fact that he obtains the same value Na from both these decryptions verifies the guess. It is also possible to verify the guess by decrypting the fourth and fifth messages to obtain Nb in two different ways.

4.2 Lomas et al.'s protocol

In [4], Lomas, Gong, Saltzer and Needham introduced the following protocol:

$$
\begin{array}{lll}
\text{Message 1.} & a \rightarrow s & : \{\!|a, b, na1, na2, ca, \{\!|ta|\!\}_{passwd(a)}|\!\}_{pks} \\
\text{Message 2.} & s \rightarrow b & : a, b \\
\text{Message 3.} & b \rightarrow s & : \{\!|b, a, nb1, nb2, cb, \{\!|tb|\!\}_{passwd(b)}|\!\}_{pks} \\
\text{Message 4.} & s \rightarrow a & : \{\!|na1, k \oplus na2|\!\}_{passwd(a)} \\
\text{Message 5.} & s \rightarrow b & : \{\!|nb1, k \oplus nb2|\!\}_{passwd(b)} \\
\text{Message 6.} & b \rightarrow a & : \{\!|rb|\!\}_k \\
\text{Message 7.} & a \rightarrow b & : \{\!|f(rb), ra|\!\}_k \\
\text{Message 8.} & b \rightarrow a & : \{\!|f(ra)|\!\}_k \ .
\end{array}
$$

Here pks is the public key of server s; $na1$, $na2$, $nb1$, $nb2$, ra, rb, ca and cb are all random numbers; ta and tb are timestamps; f is an arbitrary function (we model it as a hash function); and $\oplus$ represents Vernam encryption.

When we modelled this protocol and analysed it using Casper/FDR, we found the following attack.

$$
\begin{array}{lll}
\text{Message } \alpha.1. & A \rightarrow S & : \{\!|A, I, Na1, Na2, Ca, \{\!|0|\!\}_{passwd(A)}|\!\}_{PKs} \\
\text{Message } \alpha.2. & S \rightarrow I & : A, I \\
\text{Message } \alpha.3. & I \rightarrow S & : \{\!|I, A, Ni, Ni, Ci, \{\!|0|\!\}_{passwd(I)}|\!\}_{PKs} \\
\text{Message } \alpha.4. & S \rightarrow A & : \{\!|Na1, K1 \oplus Na2|\!\}_{passwd(A)} \\
\text{Message } \beta.1. & I_A \rightarrow S & : \{\!|A, I, Na1, Na2, Ca, \{\!|0|\!\}_{passwd(A)}|\!\}_{PKs} \\
\text{Message } \beta.2. & S \rightarrow I & : A, I \\
\text{Message } \beta.3. & I \rightarrow S & : \{\!|I, A, Ni, Ni, Ci, \{\!|0|\!\}_{passwd(I)}|\!\}_{PKs} \\
\text{Message } \beta.4. & S \rightarrow I_A & : \{\!|Na1, K2 \oplus Na2|\!\}_{passwd(A)} \ .
\end{array}
$$

The timestamp 0 is arbitrary. The intruder replays A's messages from the (partial) run α in run β, so as to get the intruder to issue another message 4 using the same nonces. The intruder guesses the value $passwd(A)$, and verifies the guess using verifier $Na1$; the relevant events of the verification are:

$$
\begin{array}{l}
guess.passwd(A) \\
infer.(Na1,\ Dec.1,\ \{\{\!|Na1, K1 \oplus Na2|\!\}_{passwd(A)}, passwd(A)\}) \\
infer.(Na1,\ Dec.1,\ \{\{\!|Na1, K2 \oplus Na2|\!\}_{passwd(A)}, passwd(A)\}) \\
verify.(Na1,\ passwd(A))
\end{array}
$$

The intruder obtains $Na1$ by decrypting both $\{\!|Na1, K1 \oplus Na2|\!\}_{passwd(A)}$ and $\{\!|Na1, K2 \oplus Na2|\!\}_{passwd(A)}$.

There are a couple of variations on this attack: the nonce $Na2$ could be used as the verifier; or the intruder could replay the responder's messages from a normal run so as to obtain two instances of message 5 using the same nonces, guess the responder's password, and verify it using either of the responder's nonces.

It turns out that this attack is not new. It is noted in a later paper by the same authors, [3], which also notes that it can be prevented by the server detecting replays of messages.

The latter paper introduced another protocol, replacing the timestamps with a nonce generated by the server. We have analysed this protocol using Casper and FDR, and found no attacks.

It is worth asking what is the point of the fields ca and cb in messages 1 and 3. If we change the model to remove these fields, we find the following attack.

$$\begin{array}{lrcl}
\text{Message 1.} & A \rightarrow S & : & \{\!|A, I, Na1, Na2, \{\!|0|\!\}_{passwd(A)}|\!\}_{PKs} \\
\text{Message 2.} & S \rightarrow A & : & A, I \\
\text{Message 3.} & I \rightarrow S & : & \{\!|I, A, Ni, Ni, \{\!|0|\!\}_{passwd(I)}|\!\}_{PKs} \\
\text{Message 4.} & S \rightarrow A & : & \{\!|Na1, K \oplus Na2|\!\}_{passwd(A)} \\
\text{Message 5.} & S \rightarrow I & : & \{\!|Ni, K \oplus Ni|\!\}_{passwd(I)}\ .
\end{array}$$

The intruder runs the protocol, using his own identity with some initiator A. He then guesses A's password, and verifies it as follows:

$$\begin{array}{l}
\mathit{infer}.(K \oplus Ni,\ Dec.2,\ \{\{\!|Ni, K \oplus Ni|\!\}_{passwd(I)}, passwd(I)\}) \\
\mathit{infer}.(K,\ VernDec,\ \{K \oplus Ni, Ni\}) \\
\mathit{guess}.passwd(A) \\
\mathit{infer}.(\{\!|0|\!\}_{passwd(A)},\ Enc,\ \{0, passwd(A)\}) \\
\mathit{infer}.(K \oplus Na2,\ Dec.2,\ \{\{\!|Na1, K \oplus Na2|\!\}_{passwd(A)}, passwd(A)\}) \\
\mathit{infer}.(Na1,\ Dec.1,\ \{\{\!|Na1, K \oplus Na2|\!\}_{passwd(A)}, passwd(A)\}) \\
\mathit{infer}.(Na2,\ VernDec,\ \{K \oplus Na2, K\}) \\
\mathit{infer}.(\{\!|A, I, Na1, Na2, \{\!|0|\!\}_{passwd(A)}|\!\}_{PKs}, \\
\quad Enc,\ \{A, I, Na1, Na2, \{\!|0|\!\}_{passwd(A)}, PKs\}) \\
\mathit{verify}.(\{\!|A, I, Na1, Na2, \{\!|0|\!\}_{passwd(A)}|\!\}_{PKs},\ passwd(A)).
\end{array}$$

The intruder obtains the session key in the normal way; he obtains A's nonces from message 4; he then uses the knowledge he has accumulated to create a message of the form of message 1, compares it with the ciphertext he observed for the actual message 1, and so verifies the guess.

This attack depends upon the intruder being able to estimate—or guess—the timestamp A uses in message 1; if the intruder cannot do this, then the attack is prevented. In the original protocol, the field ca has the same effect: it prevents the intruder from building the message 1.

We can find a very similar attack against the responder of the protocol, exploiting the absence of cb.

5 Conclusions

In this paper we have considered guessing attacks upon security protocols, produced a formal model explaining how an intruder can verify a guess, shown how to use CSP and FDR to discover such attacks, and illustrated our techniques on a number of examples.

In this final section we consider a few possible extensions and some related work.

5.1 Extensions

One extension to the work in this paper concerns protocols that contain fields that can be used for verification, but are unguessable. For example, suppose we have a protocol that contains some English sentence encrypted with the guessable value. Then a correct guess will reveal the English sentence, verifying the guess; but it seems unreasonable to suppose that the intruder can guess what the English sentence is in advance.

Within our framework of Section 2, we could model this scenario by assuming that such fields are not initially known to the intruder, but allow them to be used for verification by adding a disjunct $v \in V$ to condition (4), where V is the set of all such verifying values. We could model them within the CSP model of Section 3 in essentially the same way, adding a disjunct to the test for verification within the $INTRUDER'$ process.

One interesting question is whether we have modelled all ways in which the intruder can verify his guess. Our formalism captures all guessing attacks that we have considered, or that we have found in the literature, but it is hard to be sure that there are no others.

This paper developed a general definition of guessing attacks, and specialised it to one particular protocol analysis technique. It would be interesting to specialise it to other protocol analysis techniques such as strand spaces [12], rank functions [11, 10], theorem provers [9, 2], or other model checkers, e.g. [6, 7]. There are two aspects of the search for a guessing attack: the search for a suitable set of knowledge for the intruder, and the search for a verifiable guess within that knowledge. Model checkers can combine these two aspects into a single automated search. It remains to be seen whether other approaches can combine them so easily.

5.2 Related work

In [3], Gong et al. give a definition of guessing attacks. However, this seems less general than our definition. For a start, they assume that the set of facts known to the intruder (IK in our definition) is given to the analyser, whereas we allow IK to range over all obtainable knowledge sets. Further, they only consider the case where the verifying fact is initially known to the intruder, corresponding to the case $v \in IK$ in our formalisation: thus

- they do not consider the case that v is produced in two different ways, corresponding to the first disjunct of our condition (4);
- they do not consider the case where the verifying fact is the same as the

guessed fact, corresponding to part of the second disjunct of our condition (4); this possibility is needed for Example 2.

(They have an extra deduction, producing some known value from a key and its inverse, so they do include the case captured by the third disjunct of our condition (4).)

References

[1] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Computer Security Conference on Research in Security and Privacy*, pages 72–84, 1992.

[2] Ernie Cohen. Taps: A first-order verifier for cryptographic protocols. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 144–158, 2000.

[3] Li Gong, T. Mark A. Lomas, Roger M. Needham, and Jerome H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, 1993.

[4] T. Mark A. Lomas, Li Gong, Jerome H. Saltzer, and Roger M. Needham. Reducing risks from poorly chosen keys. *Operating Systems Review*, 23(5):14–18, 1989. Proceedings of the Twelfth ACM Symposium on Operating Systems Principles.

[5] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.

[6] Will Marrero, Edmund Clarke, and Somesh Jha. A model checker for authentication protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. Available via URL `http://dimacs.rutgers.edu/Workshops/Security/program2/program.html`.

[7] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murϕ. In *IEEE Symposium on Security and Privacy*, 1997.

[8] Robert Morris and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

[9] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[10] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *Modelling and Analysis of Security Protocols*. Pearson Education, 2000.

[11] Steve Schneider. Verifying authentication protocols with CSP. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, 1997.

[12] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2, 3):191–230, 1999.