The Perfect "Spy" for Model-Checking Cryptoprotocols

A.W. Roscoe[*] [†] M.H. Goldsmith[†]

August 5, 1997

Abstract

This paper describes the modeling of a fully potent attacker against cryptoprotocols, including its inference system, in the process algebra CSP. Techniques for keeping the state space within practical bounds for the model checker **FDR 2** are explained.

1 Introduction

Over the past four years, we have been engaged in research and development into specification and model-checking techniques for cryptoprotocols. These techniques have discovered flaws in a number of published key-exchange and authentication protocols, as well as more academic studies establishing the necessity of certain message components in more robust ones. Protocol descriptions are interpreted in Hoare's language of Communicating Sequential Processes (CSP), as are specifications such as confidentiality. The model checking exploits (and has in turn inspired) new features of Formal Systems' second generation tool, **FDR 2**.

Not only does this methodology give a elegant and convenient approach, as it generally requires no prior knowledge of potential flaws in the protocol; but it is also able to reason about issues such as liveness (non-denial of service) which are not even expressible in other formalisms. The range of protocol mechanisms which have been addressed shows that there are few, if any, fundamental limitations to the applicability of these techniques; and progress is continually being made in solving the problems of scale involved in extending the work to more complex examples.

[*] Oxford University Computing Laboratory, Parks Road, Oxford OX1 3QD, UK
[†] Formal Systems (Europe) Ltd, 3 Alfred Street, Oxford OX1 4EH, UK
Email: `Bill.Roscoe@comlab.ox.ac.uk`, `Michael.Goldsmith@fsel.com`

2 Modeling Issues

One of the perennial problems with model-checking approaches, especially those using primarily explicit state-exploration algorithms, is state-space growth. Earlier work sought to keep this within bounds by limiting the attacker's "memory" to only a few data items, but even very tight limits typically left this as a limiting factor on the complexity of problem which could practically be addressed. Simple experiments verified the intuition that significant performance benefits could be gained by exploring only those possible behaviors of an intruder which are reachable given the specific history of values observed in a sequence of protocol runs, rather than compiling the whole of the intruder's possible behavior. Indeed, exploiting such a "lazy spy" implemented as an extension to the **FDR 2** system allowed (and positively benefited from) relaxing the limitations on the spy's memory.

The spy comprises two major functions: information gathering, by overhearing or destructively capturing messages; and disinformation, faking messages from data in its possession, subject only to not being able to manipulate encrypted or otherwise protected message components without access to the appropriate keys. These are connected by an information repository, storing data items which have been learnt directly, or are deducible by analysis or synthesis. It is this component which makes the demand on state-space.

An intellectually attractive decomposition would provide a two-state process for each possible "fact", essentially representing the boolean value whether it is available to the spy or not, with some mechanism superposed to implement the inferences. Our initial intuition was that this was perhaps beyond the point of sensible decomposition, but it has turned out to be the case that in most of the classes of example we are considering, it is not only practical but highly desirable to decompose the system in this manner.

First, we can observe that there is no advantage to keeping track of all possible data items. Messages and their larger subcomponents which are constructed from simpler pieces of information essentially by catenation are known if and only if all the subcomponents are. This purely structural deduction can be encoded by making the communication of the compound message equivalent to the communication of all its atomic components (that is, plain text atoms and all encrypted subcomponents). This generally reduces the number of facts which must be tracked to be the sum, rather than the product, of the size of the atomic types involved. This makes practical the following construction.

Given a set *MESSAGES* of possibly interesting messages (essentially, those with the form of messages that are sent in the protocol, but not necessarily respecting any internal or external invariants) and a function *components* mapping the elements of this set to their immediately accessible subcomponents, we can form the converse function, yielding all messages involving a fact f:

```
messages(f) = { m | m <- MESSAGES, member(f,components(m)) }
```

Similarly, given a set *DEDUCTIONS* of (antecedents,conclusion) pairs that axiomatize the inference system, we can identify those yielding or requiring a given fact:

```
inferences(f)   = { (a,c) | (a,c) <- DEDUCTIONS, f == c }
implications(f) = { (a,c) | (a,c) <- DEDUCTIONS, member(f,a) }
```

Essentially, the intruder's knowledge within a given domain (of, say, N facts) is represented by N two-state processes each of which represents a given fact which is known or unknown. Transitions from unknown to known are possible by one of two events for each fact f. One possibility is that the fact is a component in "clear" of a message which can be overheard; the other that it is the consequent of an inference from other facts known to the spy:

```
IGNORANT(f) =
  hear?_:messages(f) -> KNOWS(f)
  []
  infer?_:inferences(f) -> KNOWS(f)
```

Once a fact f is known, the process will permit further events representing any inferences which use f as an antecedent, as well as being able to allow messages containing f to be synthesized. In addition, if f has been said to be a secret, its disclosure can be signaled:

```
KNOWS(f) =
  hear?_:messages(f) -> KNOWS(f)
  []
  infer?_:implications(f) -> KNOWS(f)
  []
  say?_:messages(f) -> KNOWS(f)
  []
  member(f,SECRETS) & leak.f -> KNOWS(f)
```

The activity of an intruder performing deductions is thus represented by the occurrence of these *infer* actions, and no additional process is required. The deductions thus make no additional contribution to the state space of the attacker. Synchronizing parallel composition is used to combine these two-state processes in such a way that an inference event can only occur when all of its antecedents are known and its conclusion is not already known,[1] and that hearing and saying compound messages involves the participation of all their components:

[1] The inference events are naturally concealed from the rest of the system; if they could be repeated, this would lead to the possibility of infinite chatter – in contrast, the *hear* events must not be inhibited, as further messages involving f can quite legitimately form part of the protocol. The requirement for non-repetition of *infer* events can be met by blocking those deductions which involve the conclusion among the antecedents of the axiom.

```
SPY =
  ( || f : ATOMIC_FACTS @
      [ Union {
          { hear.m, say.m | m <- messages(f) },
          { infer.d | d <- diff(inferences(f),implications(f)) },
          { infer.d | d <- diff(implications(f),inferences(f)) },
          { leak.f | member(f,SECRETS) }
        }
      ]
        if member(f,SPY_INITIAL_KNOWLEDGE)
          then KNOWS(f)
          else IGNORANT(f)
  ) \ {|infer|}
```

2.1 Managing the deduction system

Although this structure of intruder model does have significant advantages, it does have a crucial practical drawback if implemented directly as described. Because of the way the CSP semantic models treat internal actions, in order to establish the normal refinement properties of a protocol composed with an intruder it is necessary to consider all possible combinations of reachable states. For example if two deductions may occur which do not depend on one another, there are four configurations of the intruder's memory which need to be tested, even though in our application the exact order of deductions will make no difference to the final outcome. This combinatorial explosion is clearly undesirable, and is made worse if the genuine protocol entities can engage in some events without the co-operation of the intruder: each such event further increases the number of interleaved paths by which the intruder can complete the deductive process.

In the analysis of cryptoprotocols, however, we may make use of the specific properties of intruders of the type described above. Since the deduction system is, in semantic fact, deterministic despite the internal actions, we can use partial-order techniques to optimize the exploration. Each state of the intruder has a unique final tau successor; our approach to simplifying the exploration of systems containing an intruder is thus to consider not the parallel process described in the previous section, but the state machine which results from replacing any intruder state by its ultimate tau successor, and to eliminate the internal actions of the intruder from our representation of the process altogether. In effect we evaluate the effect of internal actions of the intruder before considering the intruder's interaction with the environment. This eager evaluation of transitions out of a single state does not, of course, prevent our exploring the actual state space itself in a lazy fashion.

The **FDR 2** system provides a highly flexible interface for adding transformations on state machines, and the tau-removal scheme described above has been implemented using this facility. The resulting transformation is available as an external function *chase* in the **FDR 2** input language.

2.2 Interfacing with the legitimate agents

The legitimate agents of the system are coded so that all of their interactions with one another are to be carried over a channel *comm*:

```
channel comm : Agent.Agent.MESSAGES
```

The first index represents the purported sender of the message, and the second the intended receiver.

Rather than simply wire the *comm* channel point-to-point between the agents, the parallel composition of the system needs to allow for the potential actions of the spy. This is mediated by two additional channels of the same type, *take* and *fake*. Renaming is used to present a choice of external action when an agent engages in a *comm*, as shown in Figure 1.

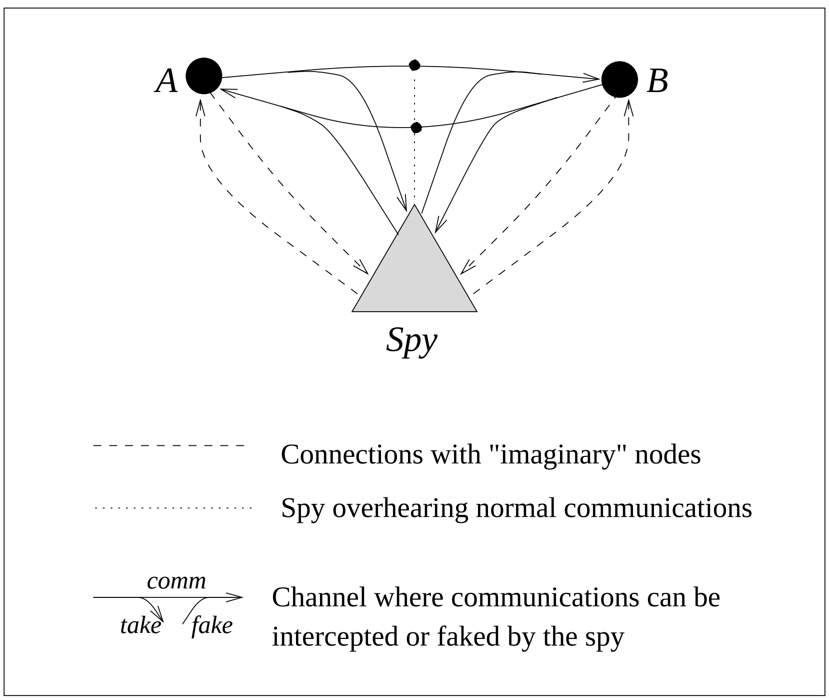

Figure 1: Wiring agents A and B to the spy

At the attacker's end, *hear* events are renamed to give a choice between the *comm* between two agents (modeling simple overhearing) and the corresponding *take* event (modeling complete capture).[2] The *say* events could be renamed to *fake* between any two agents, but in practice, there can be nothing to be gained in faking messages to himself:

```
SYSTEM =
  chase(SPY) [[ hear<-comm.l.a, hear<-take.l.a, say<-fake.a.l |
                l<-Legitimate, a<-Agent ]]
  [| {| comm, take, fake |} |]
  || id : Legitimate
    [ {| comm.id, take.id, comm.a.id, fake.a.id | a <- agent |} ]
      AGENT(id) [[ comm.id<-comm.id, comm.id<-take.id,
                   comm.a.id<-comm.a.id, comm.a.id<-fake.a.id |
                   a <- agent ]]
```

The renaming of one event to several means that which happens is at the choice of the environment, while the process within the renaming has no way of telling which way this has been resolved. If we now hide the *comm*, *take* and *fake* channels, the choice becomes nondeterministic. All the dastardly cunning possible to the spy is captured by the simple expedient of exploring the effect of every random sequence of communications available to him by which he might try to inject a spanner into the works!

2.3 Optimizations

The presentation above has aimed at presenting the ideas with as little clutter as possible. There are a number of simple optimizations which significantly improve the performance, particularly of compiling the low-level machines. For example, the sets of messages and deductions associated with each fact can be pre-computed.

The initial knowledge of the spy can also be treated more efficiently: it is logically closed under deduction (and this can be calculated, if the natural presentation is not), and its effect on the inference system can also be taken into account before we start. We can strike out initially known facts from the antecedents of any axiom, and completely discard any whose conclusion is among them. In this way, rather than initially start some "fact" processes in the *KNOWS* state, we can separate off the initially known space entirely and represent it by a (one-state) process which takes no part in the inferences.

3 Example Deduction System

The datatype of which *MESSAGES* is a finite subset is generally recursive. It contains as branches agent names, nonces, keys of all sorts, and structured

[2] In both cases, we must take care that a legitimate agent is involved as the sender; otherwise the spy could learn facts from overhearing itself!

messages – both simple catenation, and encryption:

```
datatype DATA =
  Alice | Bob | Sam | Xavier | ... |
  Na1 | Na2 | Nb1 | Nb2 | Nx1 | Nx2 | ... |
  Kas | Kbs | Kxs | ... |
  SKa | SKb | SKx | ... |
  PKa | PKb | PKx | ... |
  Sq.Seq(DATA) |
  Symmetric.DATA.Seq(DATA) |
  PublicKey.DATA.Seq(DATA) |
  ...
```

The particular extent of the type will depend on the protocol in question; Lowe's CAPSER synthesizes the datatype from the types specified in the input script.

MESSAGES is the subset of this type which includes all the bodies of messages of the forms used in the protocol which are type-correct (have nonces in the right place, use keys as the key in encryptions, and so on). We can decompose these using the *components* function discussed above, to give the set *FACTS* (of which the set *ATOMIC_FACTS* over which we replicated the spy's knowledge cells is a subset).

There are standard axioms concerning encryption which will apply whenever the relevant type of encryption is part of the protocol. For symmetric encrytion:

```
SymmetricDeductions =
   Union (
     {
       { ({Symmetric.k.xs, k}, x) | x <- set(xs) },
       { ({k, x | x <- set(xs)}, Symmetric.k.xs) }
     | Symmetric.k.xs <- FACTS
     } )
```

For public-key encryption, we require the function *dual* which maps each key to its inverse (public to secret, and vice versa); then we have

```
PublicKeyDeductions =
   Union (
     {
       { ({PublicKey.k.xs, dual(k)}, x) | x <- set(xs) },
       { ({k, x | x <- set(xs)}, PublicKey.k.xs) }
     | PublicKey.k.xs <- FACTS
     } )
```

For many systems these will be all the deductions which are necessary to model.

CSP is not an appropriate vehicle either for describing encryption algorithms or for devising methods of deciphering coded messages. That involves a lot of sophisticated mathematics in number theory, algebra, etc. It is often the case that a use of encryption fails not because of vulnerability of the cipher in use, but because of the way it is used, which is the scenario we have been addressing so far. All too frequently it is possible to defeat protocols using and supporting encryption even under the assumption that the encryption method used is unbreakable. In other cases, however, the combination of weaknesses in the precise encryption method and the shape of messages in the protocol allow additional attacks; if these weaknesses are made known as axioms in the inference system, then **FDR 2** can search out the attacks.

Examples of the kind of weakness which it is straightforward to model include schemes such as block ciphers where (subject to alignment of the data items) knowing the encryption of a sequence of data items is tantamount to knowing their encryptions under the same key individually, without needing to know the key! Cipher block chaining exhibits a similar if less fatal property, in that the encryption of prefixes of a sequence can be inferred from the encryption of the whole.

Algebraic attacks on low-exponent RSA have been exhibited by Franklin, Reiter and others [e.g., CRYPTO '95 Rump Session, August 1995]. If this is the form of public-key encryption used, then we can add deductions to reflect the additional fragility:

```
LowRSAdeductions =
  Union (
    {
      { ({PublicKey.k.<a,x>,PublicKey.k.<x>,a},x)
      | PublicKey.k.<a,x> <- FACTS,
        member(PublicKey.k.<x>,FACTS) },
      { ({PublicKey.k.<x,a>,PublicKey.k.<x>,a},x)
      | PublicKey.k.<a,x> <- FACTS,
        member(PublicKey.k.<x>,FACTS) },
      { ({PublicKey.k.<a,x>,PublicKey.k.<b,x>,a,b},x)
      | PublicKey.k.<a,x> <- FACTS, b <- FACTS, b != a
        member(PublicKey.k.<b,x>,FACTS) },
      { ({PublicKey.k.<a,x>,PublicKey.k.<x,b>,a,b},x)
      | PublicKey.k.<a,x> <- FACTS, b <- FACTS,
        member(PublicKey.k.<x,b>,FACTS) },
      { ({PublicKey.k.<x,a>,PublicKey.k.<b,x>,a,b},x)
      | PublicKey.k.<x,a> <- FACTS, b <- FACTS,
        member(PublicKey.k.<b,x>,FACTS) },
      { ({PublicKey.k.<x,a>,PublicKey.k.<x,b>,a,b},x)
      | PublicKey.k.<x,a> <- FACTS, b <- FACTS, b != a
        member(PublicKey.k.<x,b>,FACTS) }
    } )
```

These deductions capture the simplest linear cases of the identified weaknesses; further axioms could be added to deal with multivariate polynomial relationships between the bodies of messages encrypted with the same key, where this gives rise to feasible attacks.

3.1 Algebraic equivalences

As well as the construction/destruction style of inference we have considered so far, there are sometimes equivalences between terms so that the semantic value that they represent is in fact identical. Examples include the commutativity and cancellation properties of exclusive-or, and the commutativity of many forms of public-key encryption.

Not only may these give rise to attacks (signing-after-encryption problems, for example), but they may be required for the correct operation of the protocol: Diffie-Helman style key exchange, for example, relies on commutativity between exponentiations and of the function used to combine the two half-keys.

One way to achieve this would be to code the equivalence as deductions which can take place even within opaque encrypted terms, and then rely upon the spy to *take* one agent's view of the value and *fake* the other's. This is somewhat bizarre, and certainly would not extend to establishing any liveness properties.

A superior approach is to identify which *MESSAGES* are equivalent (by computing the transitive closure of that kind of deduction system), and then use renaming to identify the "external" view of any member of a given equivalence class with a canonical representative. Where the spy can gain access to additional values by moving outside the normal space of terms used in the protocol – as for instance, exploiting

$$(a \oplus b) \oplus (b \oplus c) = a \oplus c$$

when the protocol never xor's xor's together – then there are two equally possible solutions. Either such additional equivalences can be coded in as deductions; or the spy can be given license to use a suitable larger language, and the renaming will then take care of it once more.

This technique of modeling algebraic equivalences can also be used to weaken the type system, so that an agent may be fooled into thinking a key is a nonce, for example, and perhaps be persuaded to decrypt it.

4 Conclusion

Overall, this approach has a number of significant advantages over previous attempts at such modeling in **FDR**:

- Because all possible knowledge (within the admittedly fixed domain) is known, we place no arbitrary restrictions on the size of a spy's knowledge, strengthening the value of our positive results (i.e. those where no attack is found).

- Because the intruder no longer discards information, the intruder's state after all pending deductions have been completed is a direct function of values previously communicated in the protocol, and so the intruder does not introduce as much additional state.
- Because the overall behavior of the intruder in now explored in parallel with the evolution of the protocol rather than in advance, we effectively implement a lazy exploration strategy, and examine only those intruder states which are reachable by actual protocol behavior.
- The individual sequential processes are small, and thus well-suited to the **FDR** compilers. The parallel composition and synchronization is complex, but it is efficiently handled by **FDR 2**'s supercompilation approach.

These techniques are now incorporated into Gavin Lowe's CASPER system which compiles high level protocol descriptions into CSP scripts for checking on **FDR 2**. It is possible to combine the lazy spy techniques naturally with methods for allowing the spy to exploit any algebraic relationships which may exist between encrypted and similar objects.

4.1 Acknowledgments

The work described in this paper formed part of a collaborative Strategic Research Project funded by the UK Defence Research Agency, led by Peter Ryan. The original ideas have undoubtedly benefited from comments and polish from many members of the project team, and particularly from Lowe's experiences in incorporating the scheme into CASPER.

References to (and in some cases downloadable copies of) papers arising from and around this project can be found at URL:

`http://www.comlab.ox.ac.uk/oucl/groups/security/`