Proving security protocols with model checkers by data independence techniques

A.W. Roscoe
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK*

Abstract

Model checkers such as FDR have been extremely effective in checking for, and finding, attacks on cryptographic protocols – see, for example [11, 12, 14] and many of the papers in [3]. Their use in proving protocols has, on the other hand, generally been limited to showing that a given small instance, usually restricted by the finiteness of some set of resources such as keys and nonces, is free of attacks. While for specific protocols there are frequently good reasons for supposing that this will find any attack, it leaves a substantial gap in the method. The purpose of this paper is to show how techniques borrowed from data independence and related fields can be used to achieve the illusion that nodes can call upon an infinite supply of different nonces, keys, etc., even though the actual types used for these things remain finite. It is thus possible to create models of protocols in which nodes do not have to stop after a small number of runs and to claim that, within certain limits, a finite-state run on a model checker has proved that a given protocol is secure from attack. We use a single protocol as a case study, but believe our techniques are much more widely applicable.

1 Introduction

Cryptographic protocols frequently depend on the uniqueness, and often on the unguessability, of some of the data objects they use such as keys and nonces. If we are programming an agent running such a protocol, either for practical use or as part of a program for feeding into a model checker, then each time it creates a new nonce (say) it relies on the fact that it (and usually everyone else) has not used this particular value before. Frequently, of course, there will be no mechanism in place to guarantee this uniqueness, rather the contrary being discounted because of its extreme improbability. On the other hand, the way model checkers work mean that one cannot rely on probability in this way, so if using one you have to include some mechanism for *enforcing* no repeats.

The pragmatics of running model checkers mean, unfortunately, that the sizes of types such as nonces have to be restricted to far smaller sizes than the types they represent in implementations. Usually they have to be kept down to single figures if the combinatorics of how they can create messages of the protocol is not to take other types that have to be considered, such as the overall alphabet size and the set of facts that a potential intruder might learn, beyond the level that can be managed. The models that the author and others created therefore allocated a small finite number of these values to each node that has to "invent" them during a run, so that each time a nonce (say) was required a node took one of those remaining from its initial allocation or, if there were none left, simply stopped. This use of agents with the capacity for only a finite number of runs led to several difficulties.

First and foremost, it has meant that while model checkers are rightly regarded as extremely effective tools for finding attacks on protocols, they could only be used to prove that no attack exists on the assumption that each node only engages in a very finite amount of activity. While there are often good intuitive reasons for believing that the limited check would find any attack, these are generally difficult to formalise into a component of a complete proof. Therefore it has been necessary to look to other varieties of tool, such as theorem provers (see, for example, Paulson's work [16]), for proofs once one's model checker has failed to find an attack.

Secondly, it means that questions of no loss of service (in the presence, for example, of an attacker who makes a finite but unbounded number of interventions) are difficult to address, even though the formalisms (such

*Email: Bill.Roscoe@comlab.ox.ac.uk Published in *Proceedings, 1998 IEEE Computer Security Foundations Workshop*, Rockport Massachusetts, June, 1998.

as CSP) used to model the protocols are usually well adapted to test such issues. For it will only take a small number of actions for an attacker to disrupt the limited number of runs that one of these cut down systems can manage. It would be far better to have a way of modelling agents that, when one run is disrupted by enemy action, can reset and start again as often as is required.

My purpose in this paper is to show how it is frequently possible to achieve the aim of allowing agents to pick a "fresh" nonce (or other object) for each run (without limit on how many they can perform), while only actually having finitely many of them in the type running on the model checker. This is done without curtailing the ability of the intruder to generate attacks, though it may in some circumstances introduce some artificial attacks that are not really possible. Where no attack is found on such a system, it is much easier to argue that the protocol under examination is secure than it was with the earlier class of limited-run model.

This seemingly impossible goal was attained by using methods derived from the subject of *data independence*: where a program is parameterised by a data type in the sense that it passes members of the type around, but does not constrain what the type is and may only have its control-flow affected by members of the type in tightly defined ways. The ways in which the CSP models of protocols use types such as keys, nonces and agent identities fall clearly within the scope of this theory. Timestamps are more marginal because of the way in which they are compared, and for that reason are not considered in this paper. The usual result of a data independence analysis is a *threshold*: a finite size of the type which is sufficient to demonstrate correctness for all finite or infinite sizes. The particular nature of the protocol models (especially the nature of the intruder or spy process, and the fact that all generated values from the types are assumed to be distinct) meant that for some of these types no finite threshold is derivable using standard results. Nevertheless the same sorts of techniques used to prove data independence theorems can be employed to justify transformations being applied to a data independent type at run time. The result of these transformations is that values from type T are continually shifted around, and carefully identified with each other, to create room for another value to be created that the program will treat exactly as though it were fresh.

The rest of this paper is organised as follows. Firstly we summarise the techniques used to model crypto protocols in CSP. Next we describe the basics of the theory of data independence and discuss the extent to which the protocol models fit this pattern. In Section 4 we introduce the transformation techniques which allow unbounded runs within finite types, showing both the theory and the CSP implementation methods that have been employed to date. Finally we discuss the implications of this new method and the extent to which one can reasonably claim to have proved a protocol because a specific check has succeeded.

Throughout we use, as an illustrative example Lowe's revised and corrected version of the Needham-Schroeder Public-Key protocol [11].

2 Protocol modelling with CSP/FDR

Protocols are traditionally described in the literature as a series of messages between the various legitimate participants. In the case of the revised NSPK protocol (in its abbreviated version without a public key server) these are

1. $A \to B : \{N_A, A\}_{pk(B)}$
2. $B \to A : \{N_B, N_A, B\}_{pk(A)}$
3. $A \to B : \{N_B\}_{pk(B)}$

The revision is the inclusion of B's name in message 2. Without this there is the now well-known attack discovered by Lowe.

Such a description implicitly describes the role of each participant in the protocol and carries the implication that whenever an agent has, *from its point of view* executed all the communications implied by the protocol, then it can take whatever action is enabled by the protocol. The above protocol is intended to *authenticate* A and B to each other and, we will assume, to allow them to enter into a session with each other. It is relatively easy to derive CSP versions of agents running this protocol from such a description: indeed Lowe's tool Casper [13] performs this task essentially automatically. For the protocol above the code for an agent with identity `id` might be:

```
User(id) = Send(id) [] Resp(id)

Send(id) =
|~| b:diff(agents,{id}) @
  let na=???? within
  comm.id.b.pke(pk(b),Sq.<na,id>) ->
  (User(id) []
   ([] nb:nonces @
    comm.b.id.pke(pk(id),Sq.<nb,na,b>) ->
    (User(id) []
     comm.id.b.pke(pk(b),nb) ->
                    Session(id,b,nb,na))))
```

```
Resp(id) =
[] a:diff(agents,{id}) @
  [] na:nonces @
     comm.a.id.pke(pk(id),Sq.<na,a>) ->
     (User(id) []
      (let nb = ???? within
       comm.id.a.pke(pk(a),Sq.<nb,na,id>) ->
       comm.a.id.pke(pk(id),nb) ->
                        Session(id,a,nb,na)))
```

Note the following:

- The messages passed around in this implementation of the protocol are drawn from an abstract, constructed data type that allows us to treat operations such as encryption as symbolic. (More details of this type can be found below.)

- The abilities of nodes to encrypt and decrypt messages are implicit in the way they create messages and understand the messages they receive. The fact that various states will only accept messages of a given form, sometimes only when containing a specific nonce, effectively gives a way of coding in the conditions for progress in the protocol. To all intents and purposes such entries in the protocol represent an equality test (between the nonce expected and that actually received) with progress only being possible when the test succeeds.

- Nodes do not check anything about the nonces they receive but were not originally generated by them (for example, for uniqueness).

- The mechanism for choosing a new nonce has here been left undecided, since that is the essence of this paper. In previous treatments each node was given a (small) finite supply and had to halt when this was exhausted.

- A node is allowed to "back out" of the protocol run via the choices of reverting to the base state that appear at various points. This is necessary as part of our model since nodes will only continue with the protocol if they get appropriate responses within some defined time. Unless we assume that a node, once it has aborted one run of the protocol, will never try to perform it again, it is necessary to look at what consequences the messages that our node has communicated on an aborted run might have on the system's subsequent behaviour.

- We have left the actions of a node once it thinks it is in a session as yet unspecified. We must expect, however, that there is the possibility that the session might close and that it might then run the protocol again.

Having done this it is usual[1] to place as many agents as are necessary for a complete run of the protocol (in this case two; the most frequent difference to this being when some sort of server is required in addition) in parallel in the context of a network/intruder process that can do the following:

- It can act as other agents, which may or may not act in a trustworthy way.

- It can overhear all messages that pass between the trustworthy agents.

- It can prevent a message sent by one agent from reaching the intended recipient.

- It can fake a message to any agent, purporting to be from any other, only subject to what the rules built into the cryptosystem in use allow it to create based on its initial knowledge and subsequent listening.

What one usually then seeks to show is that any session that may occur between the trustworthy nodes is secure, no matter what the actions of the network from the range above. This includes showing that if either of the nodes *thinks* it has completed a run of the protocol with the other, then it really has.

The resulting network in our case is one of only three processes, though the interconnections are quite complex as shown in Figure 1 because of the multiple roles the spy process can play and because of the different fates and sources of communications on the channels apparently between the two trustworthy nodes.

The spy process is conceptually simple: it can overhear or take out any message at all, and can fake any message that it knows enough to generate (which may well be through having heard the same message, even though it cannot understand the message: in other words a *replay*). In a CSP model one calculates the set of messages that can ever pass around the system, and from that the set of relevant facts and deductions. Deductions are pairs (X,f) where, if the spy knows every fact in the set X, it can also generate f. Let us consider, for example, the following data type, which is the one used for the example in this paper.

[1] The exception to this rule would be when you wish to rely on some other number – either more or less – acting properly. For example, you might wish to prove something about the result of a run even when one of the participants is possibly corrupt.

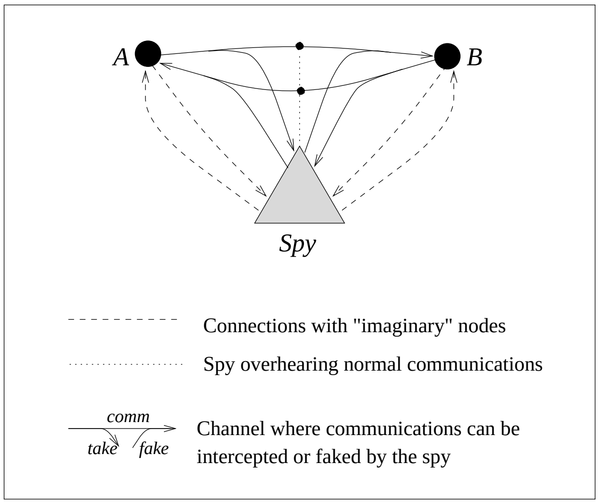

Figure 1. The network to test a simple crypto-protocol

```
datatype fact = Sq. Seq(fact) |
                PK. (fact , fact) |
                Encrypt. (fact, fact) |
                Agent . AGENT |
                Nonce . NONCE |
                pk . AGENT | sk . AGENT |
                AtoB | BtoA | Cmessage
```

Here, AGENT and NONCE are whatever types of agent identities and nonces we are using[2], PK and Encrypt are respectively constructors representing public-key and symmetric-key encryption of the second thing by the first and pk and sk are constructors creating the public and secret key of each agent. You would then expect *at least* to have the following inferences in the spy relevant to any set Z of facts:

```
deductions1(Z) = {({Sq . m}, nth(j,m)) ,
     ({nth(i,m) | i <-{0..#m-1}}, Sq . m) |
                    Sq.m <- Z, j<-{0..#m-1}}

deductions2(Z) = {({m, k}, encrypt(k,m) ) ,
                  ({encrypt(k,m), k}, m) |
                    Encrypt.(k,m) <- Z}

deductions3(Z) = {({m, k}, pke(k,m) ) ,
                  ({pke(k,m), dual(k)}, m) |
                    PK.(k,m) <- Z}
```

[2]This type is somewhat more structured than that given in [21], being parameterised by these two sets, to make the data independence analysis we will see later more transparent.

In other words, it can construct and destruct sequences, and can make up and decrypt both public-key and symmetric-key encryptions subject to possessing the right information (note the difference between these last two cases). There could be more, depending on the properties of the particular encryptions in use. These could be, for example

```
deductions4(Z) = {({encrypt(k,m), m}, k) |
                    Encrypt.(k,m) <- Z}
```

"from a message and its encryption, deduce the key", or the following, which is a consequence of a class of known attacks on low exponent RSA under which knowing a pair or encryptions that are in a known linear (as in this case) or more general polynomial relationship with each other can lead to a spy being able to decrypt (see [1], and [22] for a more complete CSP encoding of the resulting deductions)

```
deductions5(Z) =
{({a,b,pke(k,Sq.<a,f>),pke(k,Sq.<b,f>)},f) |
PK.(k,Sq.<a,f>) <- Z, PK.(k',Sq.<b,f'>) <- Z,
                          f==f', k==k', a!=b}
```

This last one says that if we know two *different* values that the fact f is paired with under encryptions using the same public key, then we can deduce f. The second of these is, for reasons we will see later, particularly interesting in the context of the theory we are developing.

We will, however, in our example, use only the basic set (1–3) above: in other words, we will assume that both the public- and symmetric-key algorithms are, free of all but the most basic of deductions.

The spy process one wants is then created by applying a suitable renaming to one equivalent to the following, initialised so that it has whatever information set we deem appropriate (presumably including all "public" information and whatever information is necessary to allow it to function as the other nodes it can play the role of, and some nonces distinct from those that the trustworthy nodes may invent for themselves).

```
Spy1(X) = learn?x -> Spy1(Close(union(X,{x})))
          [] say?x:X -> Spy1(X)
```

where Close(X) is a function that produces all facts derivable from the set X under the chosen deductive system.

In order that this spy process can even potentially be run on FDR, it is necessary that the sets of facts over which it ranges and deductions that it uses are finite. Thus the various data types making up the messages such as agent identities, keys and nonces need to range over finite sets. In practice these have to be kept very small: no more than a few values each, if the sizes of the sets of facts and deductions are not going to grow beyond the range up to (say) 2–3,000 that can be handled on FDR reasonably. In fact, this definition of the spy cannot be run on present versions of FDR because the compilation regime it uses would insist on computing all its approximately 2^N states, where N is the number of facts that the parameter-sets are builts from. As discussed in [21, 22], the above definition is replaced in actual runs by a more efficient representation, namely a parallel composition of one two-state process for each learnable fact with deductions being carried out by communication over a special channel. What is important to us, however, is that the resulting process is equivalent to the simple process above.

Having set up a model of the protocol like this we have to decide what constitutes an attack, or, equivalently, what specification the system is hoped to satisfy. A tremendous amount of literature has been devoted to this point, for example [2, 12, 20]. For various reasons, in this paper, I will concentrate on the extensional specifications used in [21] (so called in [20] because it examines the results that can occur from running the protocol, rather than at the protocol messages themselves). In this, nodes enter a session after running a protocol and use one of the nonces generated during the protocol run as a symmetric key, and we test (i) that any message Alice or Bob thinks is from the other (within a session) really is and (ii) that messages between Alice and Bob stay secret from the spy. This is done by using special symbols `AtoB` and `BtoA` for the messages Alice and Bob send each other when they *think* they are connected in a session. We can tell something has gone wrong with *authentication* (i) if either receives anything else than what it ought when in a session with the other, and something has gone wrong with *secrecy* (ii) if the spy ever learns one of these symbols. Both of these errors are raised via the occurrence of special error events within the system, and so we can test for these conditions by hiding all events other than the relevant error signal:

```
assert STOP [T= System\diff(Sigma,{|error|})
assert STOP [T= System\diff(Sigma,{|spyknows|})
```

The very simple form of these two checks will be a considerable help later on. In particular, because these are trace checks, and because our theory is rather easier to develop for that type of check, we will concentrate on that sort of check from here on.

Our ultimate goal is to prove that our system meets these specifications however many other nodes there are and however large (including infinite) the type of nonces is. Let us denote the system in which the set of agents (including the special names `Alice` and `Bob`) is A, the set of all nonces is N, and the set of nonces initially known to the spy (i.e., that the spy might "invent") is NS by

$$System(A, N, NS)$$

3 Data independence methods

A program P can be said to be *data independent* in the type T if it places no constraints on what T is: the latter can be regarded as a *parameter* of P. Broadly speaking, it can input and output members of T, hold them as parameters, compare them for equality, and apply *polymorphic* operations to them such as tupling, list forming and their inverses. It may not apply other operations either to or creating members of T, including comparison operators such as $<$ or do things like compute the size of T. Data independence has been applied to a variety of notations, originating in [24]. A brief introduction to data independence in CSP can be found in [21]. For further discussion and precise definitions, see [7, 8, 9]. An application to another topic in computer security can be found in [10].

The main objective of data independence analysis is usually to discover a finite *threshold* for a verification problem parameterised by the type T: a size of T beyond which the answer (positive or negative) to the problem will not vary. This can be done successfully for a wide range of problems, as is shown for example in [21, 10].

Several of the types used by crypto protocol models have many of the characteristics of data independence. This is typically true of the types of nonces, keys that are not bound to a specific user, and may also be true of agent identities. The main reason for this is that the abstract data type constructions used in the programs are polymorphic: there is no real difference in building a construction such as

```
PK . (pk . B, Sq . <Nonce . N, Agent . A>)
```

(the representation in our data type of a typical message 1) over the objects `A`, `B` and `N` from building a list or tuple.

There are, however, several features of the protocol descriptions that mean the general purpose results for computing thresholds do not give useful results. Firstly, the assumption that each nonce or key generated is distinct means that there can be no hope of finite thresholds from standard results as our program must at least implicitly carry an unbounded number of values in its state so it knows what to avoid next – and the starting point for threshold calculations in the context of equality tests is the maximum number of values a process ever has to remember. Secondly, the nature of the spy process causes difficulties because it also clearly has the ability to remember an unbounded number of values if they are available.

Since the general-purpose data independence results of the earlier papers had proved to be inapplicable, my approach was to apply some of the methods underlying the proofs of these results directly to the sort of CSP model that a protocol analysis generates. The aim was to take a "full-sized" model of a protocol (one with an unbounded number of other agents, and infinite sets of nonces, etc) and to use these methods to reduce the problem of proving the correctness of $\mathit{System}(A, N, \mathit{NS})$ for all parameter values to a finite check.

One of the most important methods for proving data independence theorems is setting up relations or mappings between the behaviours that two different versions of a parameterised system can display. This is an application of the theory of *logical relations* [15, 17, 18, 23]. Specifically, we can ask the question of when, if T and T' are two values for a type parameter, and $\phi : T \rightarrow T'$ is a function, does the function ϕ "lift" to map each behaviour of the parameterised process $P(T)$ to one of $P(T')$.

Given that we are concentrating on traces, this can be done with scarcely any restriction if ϕ is an injective function (i.e., does not map two distinct values to the same place): we always have

$$\mathit{traces}(P(\phi(T))) = \{\phi(t) \mid t \in \mathit{traces}(P(T))\}$$

for a data independent program P and function ϕ (where, on the left-hand side, ϕ has also been applied to the values of any constants of appropriate type within P, and on the right-hand side, the application of ϕ to a trace means its application to each member of T that occurs within t [3]). As soon, however, as we have a non-injective function the situation becomes more difficult, since it can change the results of whatever equality tests occur within the program. In other words it might map an execution in which two distinct values are input and then compared for equality into one where two equal values are input and compared for equality. There may be no relationship at all between the subsequent behaviours in these two cases, and the above equivalence becomes the possibly strict inequality

$$\mathit{traces}(P(\phi(T))) \subseteq \{\phi(t) \mid t \in \mathit{traces}(P(T))\}$$

[3] We will later feel free to "lift" these functions over objects like traces and sets without comment.

and even this may not hold if P contains distinct constants of type T – for this reason we will exclude the possibility of constants occurring in P until we develop a condition below that handles them properly. What this says, in essence, is that with a bigger type T we may be able to exercise more of P's behaviour.

There is, however, an important exception to this rule, namely the case where the result of the conditional when the equality test fails is the process `STOP`: in other words, when we can guarantee that the result of an equality test proving false will never result in the process performing a trace that it could not have performed (subject to appropriate replacements of values of type T) were the test to prove true. This is a form of the the condition **PosConjEqT** *(Positive Conjunctions)* of data independence. (It is one of a number of technical conditions derived from Lazić's work: see [7], for example. The form we quote here is slightly simplified but is valid in the contexts in which we use it, specifically on the right-hand sides of trace refinement checks.) In such programs the equivalence above is regained, and we can regard it as a "collapsing" mechanism since it allows us to compute a great deal about how a program $P(T)$ treats a large T in terms of how it treats a small one.

We noted in the last section that some aspects of the protocol examples, such as the ways in which agent processes handle keys, nonces and identities, fit into this **PosConjEqT** framework: this is something that will be useful to us later. One exception to this is that we noticed that agent processes may well perform inequality checks with a few constants, such as their own names. Additionally, the presence of a parallel process whose alphabet and communications involve a given constant like this will generally introduce such inequality checks implicitly, as will the special-casing of constants belonging to particular nodes in the spy's initial knowledge. And finally, a process involving constants can legitimately depend on their mutual inequality even when it does satisfy **PosConjEqT**, and it would then be equally inappropriate to apply a function ϕ that identifies these values.

We thus define a condition $\mathbf{PosConjEqT}'_C$ for C a set of constants: the program satisfies **PosConjEqT** except that it may have non-`STOP` results for equality tests involving at least one member of the set C of constants. What we find here is that the collapsing result above holds provided ϕ is faithful to the values of the constants in C: for $x \in T$ and $c \in C$, $\phi(x) = \phi(c)$ if and only if $x = c$.

3.1 Positive deductive systems

The agent processes in a protocol model are fairly standard-style processes and easy to check for properties like $\mathbf{PosConjEqT}'_C$. The spy process is, however, of unusual construction, depending as it does on sets and a deductive system. As discussed briefly above, the role of constants in the initial knowledge of the spy (such as knowing all secret keys except those of Alice and Bob) has an effect that is easy to see, but more interesting is the role of the deductive system. What turns out to be crucial here is the nature of the preconditions of deductions.

It is frequently the case that a deduction (X, f) will only be able to occur when two objects of a type T (possibly proper subcomponents of members of X) are equal, for example the deduction

```
({Encrypt . (k, m), k}, m)
```

carries an implicit equality check between the key in the encryption and the free-standing key. This is entirely within the spirit of **PosConjEqT** and has the crucial property that the identification of two keys by a function will never disable a deduction. It is easy to imagine deductive systems that do not have this property: for example if knowing 3 *distinct* encryptions of agents' names under key k allowed us to deduce k; and the low exponent deduction on RSA described earlier falls into this category since it relies on the distinctness of the objects that are in a linear relationship.

Formally, we will define a deductive system to be *positive* relative to some type parameter T if, for any function between types $\phi : T_1 \to T_2$, whenever (X, f) is an inference the system generates for type T_1, then $(\phi(X), \phi(f))$ is one generated for T_2. What this essentially requires is that the generation of the deductions is symmetric in T (i.e., treats all members of T equivalently) and never has an inequality requirement over members of T between places they appear on the left-hand side of a deduction.

The standard deductive system described earlier has this property, as do many variants. However, as the above counter-examples show, it is something that one has to be careful of.

The important property we can now state is this: if we build a spy over a positive deductive system, and its initial knowledge set contains no inequality tests (explicit or implicit) with members of our type T other than the constants C, then the resulting process (and hence, subject to obvious conditions on the other parts of the network, the entire protocol model) satisfies $\mathbf{PosConjEqT}'_C$.

3.2 A simple application: agent names

Almost all protocol models for model checkers that one sees make the simplification of allowing the spy only a single identity. In other words, the spy's role in acting as all the other agents in the system than our trusted Alice and Bob is reduced to giving it a single name to play with in this regard. This is usually justified by the claim that any attack in which the intruder used multiple identities would work equally well if all the other identities were reduced to a single one. What we can now do is prove this assertion as a consequence of the theory we have been developing. To make this claim true we need the following:

- All processes are data independent in the type of agent names.
- The two agent processes representing Alice and Bob each satisfies $\mathbf{PosConjEqT}'_C$, where $C = \{Alice, Bob\}$[4]
- The spy process satisfies $\mathbf{PosConjEqT}'_C$ in the sense discussed in the last section (through having a positive deductive system).
- The specification we are trying to prove relates only to nodes' communications involving the names in C. In other words it can – as in our example – be decided by looking at the traces of $System(A, N, NS) \setminus X$ where X includes every event[5] mentioning an agent name outside C.

For, under these circumstances, we know that for any A with at least three members, we can define a collapsing function ϕ that maps every name other than those in C to a fixed name (say *Cameron*) and that this would simply produce the system

$$System(\{Alice, Bob, Cameron\}, N, NS)$$

and, since the application of ϕ to any trace t produced by hiding the above X is just t (for ϕ leaves the only remaining names in t, if any, alone) we get

$$traces(System(A, N, NS) \setminus X) = \\ traces(System(\{Alice, Bob, Cameron\}, N, NS) \setminus \phi(X))$$

[4] In real life it is likely that agent processes only treat their own name in a special way, but the ones used in the context of our extensional specification use each others' names in a specification-related way.

[5] This condition can be relaxed a little in the more general context of data independence arguments, but the extra complications seem unnecessary here.

so plainly the big system satisfies our specification if and only if the small one does.[6]

What we have done here is to derive criteria under which a standard informal argument can be made rigorous.

Note that the particular specifications we are using in our example do not mention any members of A or any other data independent types: error conditions are caught within the process itself and flagged via fixed events. This simplifies proofs using collapsing functions enormously, for if $P(T)$ is any process in which all events involving members of T are hidden, then, if the collapsing identity holds for ϕ,

$$\mathit{traces}(P(T)) = \mathit{traces}(P(\phi(T)))$$

which means that they obviously satisfy the same specifications. As soon as one starts to include behaviour involving T explicitly in the specification, life becomes slightly more complex. In particular one requires a fairly careful classification of, and restrictions on, the specification that are not necessary when the specification does not constrain T: see [21] for basic details and [7] for detailed analysis. This is the main reason why, in this paper, we are concentrating on the specifications we are; it would doubtless be possible to include more general styles of specification, but at the cost of extra complexity we do not need in this initial examination of the subject.

4 Towards a general proof

In our example we have shown it is sufficient to deal with the case of two-plus-one agent names, but there is still the problem of the infinite type of nonces that is required. All parts of our program apart from the mechanism that generates nonces for Alice and Bob (whom we are assuming always use entirely fresh ones) satisfy **PosConjEqT** in the type of nonces, but that mechanism assuredly does not. It is not hard to argue, however, using similar methods to the last section, that we can assume that the spy has initially only one nonce in its knowledge, and that nonce is never generated for the two trustworthy users by our nonce-creation function.

Noticing that, each time a node generates a nonce, it immediately communicates it to another user, it is possible to re-cast the original descriptions of reliable users so that the choice of which nonce they create is delegated to an external nonce manager process NM: you can think of this as an artifact of our modelling technique, in much the same way that the spy is. Every time a process wants to create a nonce with an output communication, we turn the communication into an input of the nonce and force the communication to synchronise with NM (in addition to all the other processes it is synchronising with).

If we then program NM so that it remembers all the nonces it has given out and never issues the same one twice or the one belonging to the spy at all, we have implemented our assumptions about how nonces are selected. Aside from this process, our example system satisfies **PosConjEqT** on the type N of nonces, but plainly the complete system does not.

As our system runs, the trustworthy nodes hold a small finite number of nonces at any one time (in our case[7], at most 4), but the other two processes both hold an unbounded number. NM remembers all the ones seen so far so that it doesn't repeat itself, and the spy remembers everything it has seen (including objects containing an arbitrary number of nonces) so that this can potentially be exploited in future attacks. Notice that what really matters about NM is that it always hands out a nonce unknown to any of the other processes.

It may seem like a strange thing to do, but there is no reason why we should not apply a transformation function ϕ to a data independent program in the middle of a run. Suppose P is such a program, satisfying **PosConjEqT**. Any state P' that P may have reached during its execution (i.e., P' is a state reached in its operational semantics, namely a program that represents how P behaves after some sequence of actions) will still be a data independent program, though it could well have acquired some values from T that it holds in its identifiers which were not present in its initial state. For a given assignment to these identifiers we will always have

$$\mathit{traces}(\phi(P')) \supseteq \{\phi(t) \mid t \in \mathit{traces}(P')\}$$

because of the **PosConjEqT** property, but the inequality may be strict because some values held in identifiers in P' that are distinct may get mapped to the same place by ϕ, resulting in an equality test in $\phi(P')$ giving the answer *true* where it gave *false* in P'.

What we are going to do is to apply transformations that identify some of the old nonces, no longer known to Alice and Bob, but remembered by the spy. This

[6] What we have actually done here is to show that a data-independent process satisfying $\mathbf{PosConjEqT}'_C$, with at least $\mid C \mid +1$ things in T and all members of C distinct meets a trace specification that is independent of all members of T other than C if and only if the specification is satisfied for T having exactly $\mid C \mid +1$ members. This is thus a threshold calculation.

[7] For this bound we are assuming that neither node is active in more than one protocol run at a time: this issue is discussed a little more in the Conclusions.

does not directly affect the states of the trustworthy node processes, but can potentially increase the traces of the spy because of the above inequality and might therefore lead to the resulting process failing to satisfy a trace specification though it did before. We will see an example of this later. What it cannot do, however, is lead to it satisfying a trace specification not mentioning T that it did before. Hence if the transformed P' meets such a specification, then so also did the untransformed one.

These transformations can be carried out whenever we wish during the execution of our program, and in practice have to be programmed into the spy (whose memory gets transformed each time) and NM, which can itself now re-use the values that the spy has "forgotten" through the transformation. If we make sure that the number of distinct values remembered by the spy is bounded, by applying transformations whenever it gets too large, then we can get away with having only a finite set of nonces. We have found a way of using the same ones as "fresh" over and over again!

On the assumptions (i) that the deductive system in use is positive, (ii) the trustworthy processes satisfy **PosConjEqT** in N and (iii) the specification we are using does not explicitly constrain N (for otherwise we would have to be careful that it was not itself perturbed by the transformations) we now have a "fail-safe" method for attempting to prove trace properties of our protocols using finite types. Fail-safe here means that it will guarantee to find an attack if there is one, but may still fail even when there is none because a transformation has enabled the spy to perform an inference it could not have performed without. Whether or not such false attacks are thrown up is heavily affected by the strategy that is used for generating the transformations applied during execution: that is what we now discuss.

From an implementation and clarity point of view I have found it useful to divide the type N into two parts which we might term *foreground* F and *background* B. The foreground values are the ones supplied to the trustworthy nodes when they request a fresh value, and the background values contain all those initially known to the spy and those to which redundant foreground values are mapped. The background values will be accepted by Alice and Bob as valid nonces, just will never be generated by them as their own.

Provided the manager process is kept accurately up to date with which foreground values are presently known to Alice and Bob, the size of F needs to be exactly the largest number of nonces that (a) were generated by one of these two nodes and (b) have continuously been known to at least one of them ever since. In our example protocol this is evidently no more than 4, and is in fact[8] 3. There is no benefit from using more foreground values than are needed.

The more background values there are, the more flexibility there will be in deciding where to map each redundant member of F. In designing the strategy for where to map each one, one should be conscious of the desirability of enabling as few spurious deductions as possible. The following principles seem to the author to be desirable, the first of them indispensible.

(a) If the type under consideration ever gets used as keys, one must never identify one that the spy presently does know with one that it does not. For that would give it the immediate "ability" to decrypt all the messages it may hold under the unknown key.

(b) It might cause problems if a redundant member of F is mapped to a member of B that is currently meaningful to a trustworthy node, for this might give the spy extra messages it can use in its dealings with that node.

In the case of our example, precaution (b) is not necessary: no new attacks are created by ignoring it, though the author expects that there will be protocols where it is necessary. Precaution (a) is not necessary for the authentication specification, which still succeeds if it is ignored. It is, however, necessary for the secrecy specification, because of the way we have used the nonces generated during runs as keys: if Alice runs a session with 'Cameron' (the spy) and then one with Bob, she will actually use separate nonces for these runs. But if our mappings send them both to the same value when they become redundant then the spy will become 'able' to decrypt the messages that passed between Alice and Bob on their session because it legitimately knew the nonces that appeared during its own session with Alice.

There is one potential problem with precaution (a), namely that the spy might learn, through inferences, all of the values in B that are set aside as targets for mapping unknown redundant values. It is wise, therefore, to put an error-trap into one's model to detect this situation, in the same way that we guarded above against running out of members of F. In our example, however, this does not occur and it is possible to get both specifications to run successfully with the three members of F already mentioned, and two members of B: one initially known to the spy (and the target

[8] If you do attempt to use a smaller-than-obvious number like this it is important that the model you use generates an error flag if the supply does not, in the event, prove adequate. This is done in my implementations.

of all redundant values that are) and also one that is not (and fortunately never becomes known during the run). This proves, therefore, when combined with the preceding analysis, that $System(A, N, NS)$ satisfies both these specifications however large the three types become.

The author suspects that, provided the two precautions above are followed in the nonce reduction strategy, it will be very rare to find false attacks, but there is no sense in which they are claimed to be complete. He conjectures that in any case where the only factor making the type of facts relevant to the spy infinite is the type N we are manipulating, then it is possible, for large enough finite B, to create a strategy that is guaranteed not to introduce false attacks. Such a result would be interesting theoretically in the sense that it would imply *decidability*, but the size of B generated would probably be so large as to make the resultant check impractical on FDR.

5 Implementation considerations

When implementing the very active nonce management regime discussed in the last section it is necessary to include enough communications from Alice and Bob to NM so that it knows what nonces are meaningful to them. Such messages are, of course, an artifice of the model with no analogue in the real world. The process NM then issues 'fresh' nonces to Alice and Bob as discussed above, and issues commands to the spy to map redundant nonces in F to nonces in B. To implement precaution (a) above it is necessary that NM can enquire of the spy whether it already knows a particular nonce, so that it can map it to an appropriate place.

So the state of the process NM has to contain information on how many times each nonce is presently relevant to (each of) Alice and Bob (bearing in mind that they may be persuaded to give a nonce more than one role in a protocol). As necessary it generates mappings of nonces in F that are not relevant to Alice or Bob to whatever member of B its strategy dictates.

The spy process has to be modified so that it can tell NM whether it knows a particular value in B, and more significantly has to become able to implement the mappings on its memory requested by NM. This is trivial in the case of the absract `Spy1(X)` process described earlier, but requires a little more ingenuity in the case of the 'lazy' spy built out of many parallel components. What happens is that, to map nonce $n_1 \in F$ to $n_2 \in B$, components of the lazy spy that know something involving n_1 are commanded to transfer their knowledge to the corresponding component involving n_2, before forgetting what they knew.

If NM has several values it can hand out when asked for a fresh nonce, then clearly it does not matter which it gives, because all are treated completely symmetrically by the rest of the network. Therefore (i) there is no point in investigating the effects of handing out different options and (ii) there may be scope for choosing *which* value to hand out with a view to cutting down the overall range of states visited. The concept of symmetry reductions over states spaces is well known to be related to data independence and is discussed, for example, in [4, 5, 6].

Readers interested in discovering the details of the CSP coding of the processes discussed here can obtain several alternative implementations via URL:

`http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/examples/security`

The result of all this effort is a model which has roughly the same number of nonces as the protocol models we were accustomed to running previously, though the additional complexity of the spy, the extra process NM, and the fact that the nonces get used in more permutations than had previously been the case, mean that it has both somewhat more and larger (and so slower to run) states than earlier models. FDR 2.24 takes several hours to get through the approximately 200,000 states discovered: bear in mind that this is after the effects of the partial order compression method `chase` described in [21, 22].

6 Conclusions and prospects

We have developed a method by which it is possible to prove far more complete results on model checkers than hitherto. It seems likely to the author that the same techniques will apply to the majority of protocols in which timestamps are not used, and where the relevant deductive system is positive. There is no reason why it should not be applied to several types at once (such as keys and nonces).

One should always be careful to state the limits of what one has proved in a case like this. Evidently the use of an abstract data type and a specified set of deductive rules over it implies that we are assuming, in our proof, that there are no other subtle properties of the encryption system that an intruder can exploit and, unless it has been built into the deductive system in some way, no way that the intruder can decrypt messages without the possession of the appropriate key. Secondly, we are assuming that a node will impose whatever discipline on accepting messages is implied by the protocol, in particular that it will not interpret a message in one shape as a message in another. Thus we have not allowed in this treatment for attacks based on type confusion, under which, for example, some advantage might be gained if an agent could be persuaded to accept another agents name as a nonce.

An important limitation on the result one has proved with a check of the form we have described is that it does not allow for either of Alice or Bob running more than one session at a time. If it is realistic that they might, then you should really include an appropriate number of copies of each in the network, which would in turn increase the number of foreground values in our types. This would increase the number of states in our example to a prohibitive level even with two copies of each agent, and of course we would like to prove appropriate results for an arbitrary number of copies of each – in other words, factoring a further parameter out of the system. This must remain a topic for future research.

In this paper we concentrated on specifications that did not directly involve the types being manipulated, since this simplified various arguments. It seems very probable that most of the trace specifications people have used for protocols could be brought within the scope of this work, as have trace specifications that constrain communications in T in other areas of data independence. But this is a topic for further research.

Our analysis has been based on trace specifications, both because the protocol models we are considering do not attempt to satisfy any stronger variety – to do so requires many further implementation details and assumptions about the spy – and because it simplifies the data independence arguments. It is, nevertheless, possible to apply data independence arguments to stronger styles of specifications, as shown by [21, 7], for example. And the fact that we do not now have to restrict how many runs an agent can perform suggests that our new modelling techniques are likely to be extremely useful in liveness and no-loss-of service analyses.

There seems no reason in principle why these techniques should not be applicable over a wider range of notations and model checkers than CSP and FDR: modelling protocols has become a popular pastime for users of a wide range of tools in the last year or two. The formality of arguments based on other notation would, however, be limited if they did not have a comparable notion of data independence. Also there seems to be no reason why the coding of the CSP scripts necessary to run these checks should not become as systematic and automated (via Casper) as have other protocols, thus relieving users of the difficulties of devising the new spy and NM processes for themselves.

Acknowledgements

This work would have been impossible without the pioneering efforts of Ranko Lazić in developing the theory of data independence upon which the arguments here are based. Equally it would have been impossible without the continued work of the staff of Formal Systems on the FDR tool. The author is also grateful for the helpful remarks of the anonymous referees.

The work reported in this paper was supported by DERA Malvern and the U.S. Office of Naval Research.

References

[1] D. Coppersmith, M. Franklin, J. Patarin, and M. Reiter, *Low-exponent RSA with related messages*, In Advances in Cryptology – EUROCRYPT '96 (LNCS 1070), 1996.

[2] W. Diffie, P.C. van Oorschot and M.J. Weiner. *Authentication and key exchanges.* Design, Codes and Cryptography **2**, pp107-125 (1992).

[3] Proceedings of DIMACS workshop on the design and formal verification of cryptographic protocols, 1997. Published on the world-wide web at URL: `http://dimacs.rutgers.edu/Workshops/Security/program2/program.html`

[4] E.A. Emerson and A.P. Sistla, *Utilizing symmetry when model checking under fairness assumptions: an automata-theoretic approach*, 309–324, Proceedings of the 7$^{\text{th}}$ CAV, Springer LNCS 939, 1995.

[5] E.A. Emerson and A.P. Sistla, *Symmetry and model checking*, Formal Methods in System Design, **9**, 105–131, 1996.

[6] C.N. Ip and D.L. Dill, *Better verification through symmetry*, Formal Methods in System Design, **9**, 41–75, 1996.

[7] R.S. Lazić, *A semantic study of data-independence with applications to the mechanical verification of concurrent systems*, Oxford University D.Phil thesis, to appear in 1997.

[8] R.S. Lazić and A.W. Roscoe, *Using logical relations for automated verification of data-independent CSP*, Proceedings of the Workshop on Automated Formal Methods (Oxford, U.K.), Electronic Notes in Theoretical Computer Science 5, 1997.

[9] R.S. Lazić and A.W. Roscoe, *A semantic study of data-independence with applications to model-checking*, Submitted for publication, 1998.

[10] R.S. Lazić and A.W. Roscoe, *Verifying determinism of data-independent systems with labellings, arrays and constants*, Submitted for publication, 1998.

[11] G. Lowe, *Breaking and fixing the Needham-Schroeder public-key protocol using FDR*, Proceedings of TACAS '97, Springer LNCS 1055, 1996.

[12] G. Lowe, *Some new attacks upon security protocols*, Proceedings of 1996 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1996.

[13] G. Lowe, *Casper: a compiler for the analysis of security protocols*, Proceedings of 1997 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1997.

[14] G. Lowe and A.W. Roscoe, *Using CSP to detect errors in the TMN protocol*, IEEE transactions on Software Engineering **23**, 10 659–669 (1997).

[15] J.J. Mitchell, *Type systems for programming languages*, in 'Handbook of theoretical computer science' (van Leeuwen, *ed*), Elsevier, 1990.

[16] L.C. Paulson, *Mechanized Proofs of Security Protocols: Needham-Schroeder with Public Keys*, Report 413, Cambridge University Computer Lab (1997).

[17] G.D. Plotkin, *Lambda-definability in the full type hierarchy*, in 'To H.B. Curry: essays on combinatory logic, lambda calculus and formalism' (Seldin and Hindley, *eds*), Academic Press, 1980.

[18] J.C. Reynolds, *Types, abstraction and parametric polymorphism*, Information Processing 83, 513–523, North-Holland, 1983.

[19] A.W. Roscoe, *Modelling and verifying key-exchange protocols using CSP and FDR*, Proceedings of 1995 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1995.

[20] A.W. Roscoe, *Intensional specifications of security protocols*, Proceedings of 1996 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1996.

[21] A.W. Roscoe, *The theory and practice of concurrency*, Prentice Hall, 1998.

[22] A.W. Roscoe and M.H. Goldsmith, *The perfect 'spy' for model-checking crypto-protocols*, in [3]

[23] P.L. Wadler, *Theorems for free!*, 347–359, Proceedings of the 4^{th} ACM FPLCA, 1989.

[24] P. Wolper, *Expressing interesting properties of programs in propositional temporal logic*, 184–193, Proceedings of the 13^{th} ACM POPL, 1986.