Fault-Preserving Simplifying Transformations for Security Protocols
or
Not Just the Needham Schroeder Public Key Protocol

Mei Lin Hui* Gavin Lowe†

December 8, 2000

Abstract

Recent techniques for analyzing security protocols have tended to concentrate upon the small protocols that are typically found in the academic literature. However, there is a huge gulf between these and most large commercial protocols: the latter typically have many more fields, and much higher levels of nested encryption. As a result, existing techniques are difficult to apply directly to these large protocols.

In this paper we develop the notion of *fault-preserving simplifying transformations*: transformations that have the property of preserving insecurities; the effect of such transformations is that if we can verify the transformed protocol, then we will have verified the original protocol. We identify a number of such fault-preserving simplifying transformations, and use them in the analysis of a commercial protocol.

1 Introduction

Many security protocols have been suggested, with various goals, such as establishing a cryptographic key, or achieving authentication (where an agent becomes sure of the identity of the other agent taking part in the protocol). These protocols are supposed to succeed even in the presence of a malicious agent, called an *intruder*; this intruder is assumed to have complete control over the communications network, and so can intercept messages, and introduce new messages into the system using information from messages that have previously been seen. Unfortunately, a large proportion of the protocols that have been suggested do not succeed in their stated goals!

In recent years, a number of techniques have been developed for analysing protocols; the two main categories of such techniques are:

*Communications and Electronic Security Group, GCHQ, PO Box 144, Cheltenham GL52 5UE, UK; e-mail: `mlhui8@hotmail.com`.

†Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; e-mail: `gavin.lowe@comlab.ox.ac.uk`.

Model checking techniques where a model checker is used to search the state space of a small system running the protocol, looking for attacks; see, for example, [13, 5, 12, 6, 10, 8, 14, 11]

Direct proofs where the protocol is proved correct directly (possibly with the help of machine assistance), typically using some form of induction; see, for example, [19, 2, 15, 22].

These techniques have proved successful at analysing the protocols that have appeared in the academic literature. Such protocols are normally small, typically having between two and seven messages, each message having between two and ten fields. However, most commercial protocols are considerably more complicated, typically containing several layers of nested encryptions, and significant amounts of redundancy. For example, the CyberCash Main Sequence Protocol—a protocol for carrying out commercial transactions over the Internet—is given in Appendix A; this protocol contains dozens of fields, and in one place, six levels of nested cryptography.

There are a number of reasons why commercial protocols are more complicated than ones in the academic literature:

- Commercial protocols contain many fields that are included for functionality rather than security; for example, the CyberCash Protocol includes a field representing the customer's postal code.
- Often messages are hashed before being signed; this hashing reduces the amount of text transferred by the protocol, but increases the complexity of the protocol from the point of view of analysis.
- Sometimes protocols are simply over-engineered; common techniques are to include a copy of all previous fields in each message, or to include more encryptions than are necessary.

This extra complexity of such protocols makes analysis much more difficult: for the model checkers, it leads to an explosion in the state space and the message space; for those doing direct proofs, the complexity just makes the protocol harder to understand, let alone verify it.

However, we will often have a strong feeling that much of the complexity of such a protocol could be removed without altering its security: for example, some of the fields and some of the nested encryption might appear to be irrelevant to security. This 'strong feeling' stems from experience in analysing protocols, allowing us to recognize when a simpler amount of encryption provides the security required. If we could prove that this complexity is indeed not needed for the security of the protocol, then it would be enough to analyse the protocol with this complexity removed. This is the question we address in this paper. We identify a number of *simplifying transformations* on protocols that have the property that if there is an attack upon the original protocol (i.e. it is insecure), then there is also an attack upon the simplified protocol; we call transformations with this property "*fault-preserving*". This means that if we can verify the simplified protocol, we will have verified the original protocol.

Further, many security protocols are intended to satisfy more than one security property, with different parts of the protocol geared towards different security properties; for example, it is often the case that one encryption (with a symmetric or public key) aims to ensure secrecy of some data, while another encryption (with a secret key) aims to provide authentication. Our simplifying transformations will help us to consider these separately; when considering whether a protocol really does satisfy a particular property, we will be able—to a certain extent—to ignore those parts of the protocol geared towards other properties.

Note that the property of being fault-preserving is only one way round: it is possible to apply fault-preserving simplifying transformations in such a way as to introduce new attacks, by oversimplifying the protocol. The idea will be, starting from the original protocol, to apply as many fault-preserving simplifying transformations as possible, trying to avoid introducing new attacks, and then to analyse the simplified protocol. If the simplified protocol is secure, then so is the original; if, however, our analysis discloses an attack upon the simplified protocol, we have to consider whether there is a corresponding attack on the original protocol (by effectively undoing all the simplifying transformations); if there *is* such an attack, then clearly the original protocol is flawed; otherwise, we have oversimplified, so we have to try again. Normally our experience and intuition will help us avoid oversimplifying: it is often clear that a particular encryption is necessary to keep a particular field secret, or to authenticate an agent; if we do oversimplify, then our mistakes normally teach us something about the protocol.

Our analyses sometimes produce an additional benefit; sometimes they reveal that a protocol really is over-engineered, in which case our simplifications produce a cleaner, simpler, and more efficient protocol.

We will consider two main categories of transformations:

Renaming transformations where the messages in the original protocol are uniformly renamed according to some function;

Structural transformations where the structure of the protocol is changed, with messages being added or removed, or the senders and/or receivers of messages changing.

Our approach for the renaming transformations is to identify a function f from the messages of the original protocol P_1 to the messages of the simplified protocol P_2, describing how messages are simplified. We then lift this function to traces (i.e. finite sequences of messages), by pointwise application. By contrast, with structural transformations we define the function f directly over traces.

In order for the transformations to be fault-preserving, we require that if tr is a trace representing an attack upon P_1, then $f(tr)$ is an attack upon P_2. Proving this for a particular f will consist of two things: that if tr is a trace of P_1, then $f(tr)$ is a trace of P_2; and that if trace tr constitutes an attack upon P_1, then $f(tr)$ constitutes an attack upon P_2.

In the next section, we present the model we will be using to reason about simplifying transformations: we give a model of a system running a protocol, along with an intruder who can interact with that system, and formalize some security properties. In Section 3 we formalize the notion of a renaming transformation, and show that if such a transformation satisfies two properties, then it is fault-preserving; then in Section 4 we show that a number of useful renaming transformations satisfy these properties. We consider two fault-preserving structural transformations in Section 5. In Section 6 we illustrate our techniques by applying them to the CyberCash Main Sequence Protocol, simplifying the protocol to a stage where we can apply standard analysis techniques. We sum up in Section 7.

2 Modelling security protocols

We will use the process algebra CSP [3, 16], to reason about security protocols, to formalize the simplifying transformations, and to verify that they satisfy the desired condition that if there is an attack on the original protocol, then there is an attack on the simplified protocol. A brief introduction to CSP is included in Appendix B.

We use CSP because it is well suited for reasoning about systems of communicating agents. Further, it has been much used for security protocol analysis, and the translation from descriptions of protocols in "standard form" into CSP is well understood. Also, CSP has a well developed theory, which we may exploit. Finally, the renaming transformations are easy to formalize—they are simply CSP renamings! We should stress, though, that our results may be used to support *any* protocol analysis technique, not just those based around CSP.

In Section 2.1 we define the underlying space of messages; in Section 2.2 we describe how the honest agents are modelled in CSP, and in Section 2.3 we do the same for the intruder; then in Section 2.4 we formalize what we mean by an attack upon the protocol, thus formalizing the security properties that we are interested in.

2.1 Messages

Our model of a system running a protocol is based around a datatype $Message$, representing the messages that are sent or received by agents. We define the message space to be built up from some set $Atom$ of atomic values, by pairing, encryption and hashing. We define $Key \subseteq Atom$ to be those atoms used for encrypting and decrypting messages; we write K^{-1} for the inverse key of K. We let $HashFn$ be the set of cryptographic hash functions. The set $Message$ is defined by:

$$\begin{aligned} Message \ ::= \ & \textsc{Atom}\, Atom \mid \\ & \textsc{Pair}\, Message\ Message \mid \\ & \textsc{Encrypt}\, Message\ Key \mid \\ & \textsc{Hash}\, HashFn\ Message. \end{aligned}$$

We will use the syntactic sugar (M, M') for $\textsc{Pair}\,M\,M'$, $\{M\}_K$ for $\textsc{Encrypt}\,M\,K$, and $g(\!|M|\!)$ for $\textsc{Hash}\,g\,M$.

Note that implicit in the above datatype definition is that two messages are equivalent only if they have identical fields, so, for example, encryption is injective:

$$\{M\}_K = \{M'\}_{K'} \Leftrightarrow M = M' \wedge K = K'.$$

This is the *perfect encryption assumption*.

This datatype definition means that some messages are distinguished that we would like to identify, for example $((\textsc{Atom}\,A, \textsc{Atom}\,B), \textsc{Atom}\,C)$ and $(\textsc{Atom}\,A, (\textsc{Atom}\,B, \textsc{Atom}\,C))$: the definition does not capture the associativity of pairing. We therefore identify a *normal form* for messages, and will ensure that only messages in normal form are passed in the system[1]. The normal form will have all pairing associating to the right, for example $(\textsc{Atom}\,A, (\textsc{Atom}\,B, \textsc{Atom}\,C))$. This means that for any pair in normal form, the first element will not itself be a pair. We write $pair(M)$ if M is a pair (easily defined over the structure of the message space), and define a function to convert a message into normal form as follows:

$$\begin{array}{rcl}
NF & : & Message \rightarrow Message, \\
NF(\textsc{Atom}\,A) & \widehat{=} & \textsc{Atom}\,A, \\
NF((M_1, M_2), M_3) & \widehat{=} & \text{let } (M_1', M_2') = NF(M_1, M_2) \text{ in } (M_1', NF(M_2', M_3)), \\
NF(M_1, M_2) & \widehat{=} & (NF(M_1), NF(M_2)), \text{if } \neg pair(M_1), \\
NF(\{M\}_K) & \widehat{=} & \{NF(M)\}_K, \\
NF(g(\!|M|\!)) & \widehat{=} & g(\!|NF(M)|\!).
\end{array}$$

We define $NFMessage$ to be the set of all messages in normal form.

2.2 Honest agents

We will model each agent taking part in the protocol by a CSP process, using the following events:

- events of the form $send.A.B.M$, representing agent A sending (normal form) message M, with intended recipient B;
- events of the form $receive.A.B.M$, representing agent B receiving message M, apparently from A;
- events of the form $claimSecret.A.B.M$, representing that A thinks that M is a secret shared only with B; if B is not the intruder, then the intruder should not learn M;
- events of the form $running.A.B.Ms$, representing that A thinks he is running the protocol with B; Ms is a sequence of messages, recording some details about the run in question;

[1] Having to define a normal form in this way is very much orthogonal to the main thrust of the paper, but is unfortunately necessary; the reader is advised not to be distracted too much by the details.

- events of the form $finished.A.B.Ms$, representing that A thinks he has finished a run of the protocol with B; Ms is a sequence of messages, recording some details about the run in question.

Events of the first two forms are used for modelling the main communications of the protocol. The latter three forms are signal events, which will be used for writing specifications for the protocol; we discuss this further in Section 2.4.

Each honest agent can be represented by a CSP process, describing the way the agent sends and receives messages. The precise details will depend upon the protocol in question; a number of previous papers (e.g. [6, 8, 10, 18]) have described the translation from a protocol description in "standard form" to a CSP representation. However, it turns out that we don't need to assume anything else about the CSP representation of the honest agents (apart from the fact that such a representation exists)—our results apply to arbitrary CSP processes.

We write P_A for the CSP process representing the honest agent with identity A; this process will have the following alphabet:

$$\begin{aligned}
\alpha(P_A) \;\widehat{=}\; & \{send.A.B.M \mid B \in Agent, M \in NFMessage\} \\
& \cup \{receive.B.A.M \mid B \in Agent, M \in NFMessage\} \\
& \cup \{claimSecret.A.M \mid M \in NFMessage\} \\
& \cup \{running.A.B.Ms \mid B \in Agent, M \in NFMessage^*\} \\
& \cup \{finished.A.B.Ms \mid B \in Agent, M \in NFMessage^*\}.
\end{aligned}$$

2.3 The intruder

We now define the most general intruder who can interact with a system running the protocol. Following [18], the definition is based around a relation $\vdash$, which defines the ways in which the intruder can deduce new messages; we write $B \vdash M$ if message M can be deduced from the set of messages B; the relation is defined by the following six rules:

member $M \in B \Rightarrow B \vdash M$;

pairing $B \vdash M \wedge B \vdash M' \wedge \neg pair(M) \Rightarrow B \vdash (M, M')$;

splitting $B \vdash (M, M') \Rightarrow B \vdash M \wedge B \vdash M'$;

encryption $B \vdash M \wedge B \vdash \textsc{Atom}\,K \wedge K \in Key \Rightarrow B \vdash \{M\}_K$;

decryption $B \vdash \{M\}_K \wedge B \vdash \textsc{Atom}\,K^{-1} \Rightarrow B \vdash M$;

hashing $B \vdash M \wedge g \in HashFn \Rightarrow B \vdash g(\!|M|\!)$.

The intruder can create new messages by creating or splitting pairs, by encrypting or decrypting with keys he can produce, and by hashing.

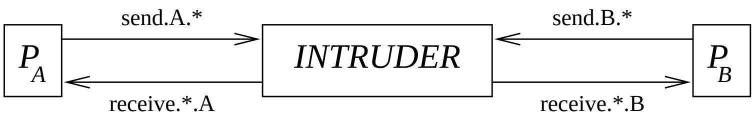
We will need the following two properties of this relation:

Lemma 1

$$\begin{aligned}
B \vdash M \wedge B \subseteq B' \;&\Rightarrow\; B' \vdash M, \\
B \vdash M \wedge B \cup \{M\} \vdash M' \;&\Rightarrow\; B \vdash M'.
\end{aligned}$$

If the intruder can deduce a message M from some set B, then he can certainly deduce it from some larger set B'; and if the intruder can deduce a message M, then he can use that message in subsequent deductions. These properties can be proved by induction over the defining rules for $\vdash$.

Our model of the system will ensure that the intruder receives every message sent by an honest agent, and can then choose what messages are received by honest agents. A system with two honest agents, A and B, can therefore be pictured as below:



In effect, the intruder process also models the effect of the communication medium.

The intruder can be represented by a CSP process parameterized by the set S of messages that he either initially knew, or has subsequently seen. From this knowledge, he will be able to produce any of the messages in the set:

$$deds(S) \mathrel{\widehat{=}} \{M \in Message \mid S \vdash M\}.$$

The intruder with knowledge S can be defined as follows (recall that the *send* channel represents messages that are sent by honest agents and so are received by the intruder, and that the *receive* channel represents messages that are received by honest agents and are sent by the intruder):

$$\begin{array}{l} INTRUDER(S) \mathrel{\widehat{=}} \\ \quad \Box_{M \in Message}\, send?A?B!M \rightarrow INTRUDER(S \cup \{M\}) \\ \quad \Box \\ \quad \Box_{M \in deds(S)}\, receive?A?B!M \rightarrow INTRUDER(S) \\ \quad \Box \\ \quad \Box_{M \in deds(S)}\, leak.M \rightarrow INTRUDER(S). \end{array}$$

The intruder may:

- see messages sent by honest agents, and so learn the contents;
- send messages with contents that can be derived from his current knowledge, such messages being received by an honest agent;
- signal that he knows any message that can be derived from his current knowledge.

We assume that the intruder initially knows some set IIK (*intruder's initial knowledge*) of messages. The complete system is then represented by the CSP process:

$$SYSTEM \mathrel{\widehat{=}} \left(|||_{A \in Honest}\, P_A\right) \parallel INTRUDER(IIK),$$

(where $Honest$ is the set of honest agents' identities) that is, the parallel composition of the honest agents and the intruder, synchronizing on their common events (i.e. the *send* and *receive* events).

2.4 Attacks

In this subsection we consider possible requirements of a protocol. We will produce various *trace specifications* representing different security properties that we would like systems running the protocol to satisfy. A trace specification is a predicate *SPEC(tr)* whose free variable *tr* represents a trace. A process satisfies such a specification if *SPEC(tr)* is true for all its traces:

$$P \ \mathbf{sat} \ \mathit{SPEC} \Leftrightarrow \forall\, tr \in traces(P) \bullet \mathit{SPEC(tr)}.$$

Thus, we will be interested in whether $SYSTEM$ **sat** *SPEC*, for various specifications representing security properties.

First of all, we define a breach of secrecy to occur when an honest agent thinks a certain message (typically an atomic message) is secret, but the intruder learns it. We formalize secrecy by specifying that if an agent A performs a *claimSecret.A.B.M* event (representing that A believes M to be a secret shared only with B), where B is not the intruder and belongs to the set of honest agents, then the intruder should never perform $leak.M$ (i.e. should never learn M):

$$\begin{aligned} \mathit{Secrecy}(tr) \ \widehat{=}\ & \forall\, A \in \mathit{Agent}\,;\, B \in Honest \bullet \\ & \quad \mathit{claimSecret}.A.B.M \text{ in } tr \Rightarrow \neg leak.M \text{ in } tr. \end{aligned}$$

We now define some entity authentication specifications. As explained in [7], we believe that the term "authentication" refers to a collection of related requirements, rather than there being a single generic authentication specification. The following is one of the more common forms of authentication:

> Whenever A completes a run of the protocol, apparently with B, then B has previously been running the protocol, apparently with A, and the two agents agreed on some collection Ms of messages, and each such run of A corresponds to a *unique* run of B.

To formalize this requirement, we specify that the number of $finished$ events performed by A concerning B with some sequence of messages Ms must be no more than the number of corresponding *running* events performed by B. We write this requirement as a trace specification, parameterized by the set *AgreementSet* of sequences of messages upon which the agents should agree (for example, if it is required that the agents agree upon a key and a nonce, then the agreement set will be all sequences of the form $\langle K, N\rangle$ for keys K and nonces N):

$$\begin{aligned} & \mathit{Agreement}_{\mathit{AgreementSet}}(tr) \ \widehat{=} \\ & \quad \forall\, A \in \mathit{Agent}\,;\, B \in Honest\,;\, Ms \in \mathit{AgreementSet} \bullet \\ & \qquad tr \downarrow finished.A.B.Ms \leq tr \downarrow running.B.A.Ms. \end{aligned}$$

If tr is a trace, and e is an event, then $tr \downarrow e$ represents the number of occurrences of e in tr. Of course, in order to ensure that this specification captures the requirements outlined above, we need to ensure that the $finished$ and $running$ events correctly capture the informal descriptions given to them above:

- the *finished* events should be performed by honest agents when they complete a protocol run;
- the *running* events should be performed by an honest agent B at the point that it is required that B reaches when its partner A completes its run; normally this will be just before the last *send* event that has a causal link with a message received by A.

See [7] for more details.

We believe it would be completely straightforward to extend the results of this paper to other authentication specifications, such as the aliveness and non-injective agreement specifications of [7]. It might also be possible to deal with completely different specifications such as non-repudiation or anonymity.

3 Renaming transformations on protocols

In this section we formalize how we represent renaming transformations upon protocols; we then state a result, giving sufficient conditions for a transformation to be safe.

3.1 Formalizing renaming transformations

A renaming transformation will be defined via a function

$$f \ : \ Message \rightarrow Message,$$

which defines how messages in the original protocol are replaced by messages in the simplified protocol. For example, f might be a function that removes all instances of a particular atomic field, or that strips off all instances of a particular encryption. In the following, we will lift f to apply to several different types of arguments.

We lift the function f to events as follows:

$$\begin{aligned} f(\mathit{send}.A.B.M) &\mathrel{\widehat{=}} \mathit{send}.A.B.f(M), \\ f(\mathit{receive}.A.B.M) &\mathrel{\widehat{=}} \mathit{receive}.A.B.f(M), \\ f(\mathit{claimSecret}.A.B.M) &\mathrel{\widehat{=}} \mathit{claimSecret}.A.B.f(M), \\ f(\mathit{running}.A.B.Ms) &\mathrel{\widehat{=}} \mathit{running}.A.B.f(Ms), \\ f(\mathit{finished}.A.B.Ms) &\mathrel{\widehat{=}} \mathit{finished}.A.B.f(Ms), \end{aligned}$$

and lift it to traces by:

$$f(tr) \mathrel{\widehat{=}} \langle f(e) \mid e \leftarrow tr \rangle.$$

We now define the CSP representation of an honest agent A in the transformed system, corresponding to the process P_A in the original system. We need a process that sends or receives the message $f(M)$ whenever the process P_A sends or receives M. We therefore define the new representation to be the process $f(P_A)$ (recall that the CSP process $f(P)$ is the process that acts like P, except where every event e is renamed to $f(e)$). This is certainly the right model for the honest agents when one considers only the sending and receiving of messages. Some protocols, however, involve more than just a straightforward sending and receiving of messages; for example, in many cases agents perform tests upon the messages they receive; in these cases, it's not immediately obvious how the simplifying transformations affect the protocol. This definition gives the answer: the behaviour of the agent in the transformed protocol is described by the process $f(P_A)$.

When we transform the protocol, we similarly have to transform the initial knowledge of the intruder; we will let IIK' be this transformed initial knowledge. Below we will state various results relating IIK and IIK': these results will tell us what initial knowledge (IIK') we have to give the intruder in the transformed system, in order to prove properties of the original system for a particular choice (IIK) of the intruder's initial knowledge there.

The complete transformed system is modelled, therefore, by:

$$\mathit{SYSTEM}' \;\widehat{=}\; \left(|||_{A \in Honest}\, f(P_A)\right) \parallel \mathit{INTRUDER}(IIK').$$

3.2 A general result for fault-preserving renaming transformations

We now present a general result that gives sufficient conditions under which a renaming transformation is fault-preserving. In the next section we give particular examples of such transformations.

In order to show that a particular transformation f is fault-preserving, we need to show that if there is an attack upon SYSTEM', then there is an attack upon SYSTEM. We begin by showing that if $tr \in traces(\mathit{SYSTEM})$ then $f(tr) \in traces(\mathit{SYSTEM}')$; we then go on to consider attacks.

It turns out that we will need two conditions upon the transformation function f and the intruder's initial knowledge in the two systems. The first condition is that if the intruder can deduce a particular message M in the original protocol, then he can deduce the corresponding message $f(M)$ in the corresponding setting in the transformed protocol:

$$\begin{array}{l} \forall\, B \in \mathbb{P}(NFMessage)\,;\, M \in NFMessage \bullet \\ \qquad B \cup IIK \vdash M \Rightarrow f(B) \cup IIK' \vdash f(M). \end{array} \tag{1}$$

We call this property the *distribution* property. (This property is very similar to Roscoe's positive property from [17].)

The second condition is that if the intruder initially knows a particular message in the original system, then he knows the corresponding message in the transformed system:

$$f(IIK) \subseteq IIK'. \tag{2}$$

This property tells us how to choose the initial knowledge for the intruder in the transformed system, given his initial knowledge in the original system. (It turns out that in some cases we will want to take IIK' to be a proper superset of $f(IIK)$; see Section 4.3, for example.) The following lemma relates the intruders in the two systems:

Lemma 2 If the transformation function f satisfies equations (1) and (2), then

$$INTRUDER(IIK') \sqsubseteq f(INTRUDER(IIK)).$$

Proof: We will prove:

$$\forall S \in \mathbb{P}(Message) \cdot INTRUDER(f(S) \cup IIK') \sqsubseteq f(INTRUDER(S)),$$

which implies the result by taking $S = IIK$, and making use of the inclusion $IIK' \supseteq f(IIK)$. For convenience, we define $f^+(S) \mathrel{\widehat{=}} f(S) \cup IIK'$, and calculate as follows:

$$\begin{array}{cl}
& INTRUDER(f^+(S)) \\
= & \langle \text{definition} \rangle \\
& \Box_{M' \in Message}\, send?A?B!M' \to INTRUDER(f^+(S) \cup \{M'\}) \\
& \Box \\
& \Box_{M' \in deds(f^+(S))}\, receive?A?B!M' \to INTRUDER(f^+(S)) \\
& \Box \\
& \Box_{M' \in deds(f^+(S))}\, leak.M' \to INTRUDER(f^+(S)) \\
\sqsubseteq & \left\langle \begin{array}{l} \{f(M) \mid M \in Message\} \subseteq Message; \\ \{f(M) \mid M \in Message \wedge f^+(S) \vdash f(M)\} \subseteq deds(f^+(S)) \end{array} \right\rangle \\
& \Box_{M \in Message}\, send?A?B!f(M) \to INTRUDER(f^+(S) \cup \{f(M)\}) \\
& \Box \\
& \Box_{M \in \{M \in Message \mid f^+(S) \vdash f(M)\}}\, receive?A?B!f(M) \to INTRUDER(f^+(S)) \\
& \Box \\
& \Box_{M \in \{M \in Message \mid f^+(S) \vdash f(M)\}}\, leak.f(M) \to INTRUDER(f^+(S)) \\
\sqsubseteq & \langle f^+(S) \cup \{f(M)\} = f^+(S \cup \{M\}); \text{distribution property (1)} \rangle \\
& \Box_{M \in Message}\, send?A?B!f(M) \to INTRUDER(f^+(S \cup \{M\})) \\
& \Box \\
& \Box_{M \in deds(S)}\, receive?A?B!f(M) \to INTRUDER(f^+(S)) \\
& \Box \\
& \Box_{M \in deds(S)}\, leak.f(M) \to INTRUDER(f^+(S)).
\end{array}$$

Also,

$$\begin{array}{cl}
& f(INTRUDER(S)) \\
= & \langle \text{definition; laws of CSP} \rangle \\
& \Box_{M \in Message}\, send?A?B!f(M) \to f(INTRUDER(S \cup \{M\})) \\
& \Box \\
& \Box_{M \in deds(S)}\, receive?A?B!f(M) \to f(INTRUDER(S)) \\
& \Box \\
& \Box_{M \in deds(S)}\, leak.f(M) \to f(INTRUDER(S)).
\end{array}$$

Note the similarity of the right hand sides of the above equations. Hence we have shown that each $INTRUDER(f^+(S))$ process is refined by the result of applying a defining equation for the vector of processes

$$\langle f(\mathit{INTRUDER}(S')) \mid S' \in \mathbb{P}(Message) \rangle$$

to the vector of processes

$$\langle \mathit{INTRUDER}(f^{+}(S')) \mid S' \in \mathbb{P}(Message) \rangle.$$

We may thus use fixed-point induction [16, page 122] to deduce that

$$\mathit{INTRUDER}(f^{+}(S)) \sqsubseteq f(\mathit{INTRUDER}(S)).$$

□

We now prove our main theorem, which relates the traces of the two systems.

Theorem 3 If the transformation function f satisfies equations (1) and (2), then

$$\forall\, tr \in traces(\mathit{SYSTEM}) \cdot f(tr) \in traces(\mathit{SYSTEM}').$$

Proof: The result of the theorem can be rewritten as:

$$\mathit{SYSTEM}' \sqsubseteq f(\mathit{SYSTEM}),$$

which we prove as follows:

$$\begin{array}{ll}
 & f(\mathit{SYSTEM}) \\
= & \langle \text{definition} \rangle \\
 & f\left(\left(|||_{A \in Agent}\, P_A\right) \parallel \mathit{INTRUDER}(IIK)\right) \\
\sqsupseteq & \langle \text{CSP laws 1 and 2 from Appendix B} \rangle \\
 & \left(|||_{A \in Agent}\, f(P_A)\right) \parallel f(\mathit{INTRUDER}(IIK)) \\
\sqsupseteq & \langle \text{from Lemma 2} \rangle \\
 & \left(|||_{A \in Agent}\, f(P_A)\right) \parallel \mathit{INTRUDER}(IIK') \\
= & \langle \text{definition} \rangle \\
 & \mathit{SYSTEM}'.
\end{array}$$

□

The above theorem shows that every trace of the original system is "matched" by one in the new system. We want to show that every *attack* on the original system is "matched" by one on the new system. We produce sufficient conditions such that if tr constitutes an attack upon the original protocol, then $f(tr)$ constitutes an attack on the transformed protocol. We begin with the secrecy specification:

Lemma 4 If a particular trace tr constitutes a breach of secrecy on the original protocol, then $f(tr)$ is a breach of secrecy on the simplified protocol:

$$\neg Secrecy(tr) \;\Rightarrow\; \neg Secrecy(f(tr))\ .$$

Proof: Suppose tr constitutes a breach of secrecy on the original protocol, i.e., $\neg Secrecy(tr)$. Then for some $A \in Agent$, $B \in Honest$, $M \in Message$, we have $claimSecret.A.B.M$ in tr and $leak.M$ in tr. Hence $claimSecret.A.B.f(M)$ in $f(tr)$ and $leak.f(M)$ in $f(tr)$. But this constitutes a breach of secrecy on the simplified protocol: $\neg Secrecy(f(tr))$. □

This result shows that such transformations are fault-preserving for the secrecy property:

Corollary 5 If equations (1) and (2) are satisfied, then:

$$SYSTEM' \ \mathbf{sat} \ Secrecy \ \Rightarrow \ SYSTEM \ \mathbf{sat} \ Secrecy.$$

Proof: Suppose $SYSTEM'$ **sat** $Secrecy$, and suppose for a contradiction that $\neg(SYSTEM \ \mathbf{sat} \ Secrecy)$. Then there is some trace $tr \in traces(SYSTEM)$ such that $\neg Secrecy(tr)$. Then by Theorem 3, $f(tr) \in traces(SYSTEM')$, and by Lemma 4, $\neg Secrecy(f(tr))$. But then $f(tr)$ constitutes a breach of secrecy on $SYSTEM'$, which contradicts the assumption that $SYSTEM'$ **sat** $Secrecy$. □

One noteworthy point about the above result concerns which messages should be secret in the transformed protocol: from the proof of Lemma 4 it is easy to see that a breach of secrecy of message M in the original protocol corresponds to a breach of secrecy of $f(M)$ in the transformed protocol; hence, in order to test whether M is kept secret in the original protocol, we need to test whether $f(M)$ is kept secret in the transformed protocol.

We now consider the agreement authentication specification. If the messages upon which agreement is intended in the original protocol is $AgreementSet$, then we should take the messages upon which agreement is intended in the transformed protocol to be $f(AgreementSet)$. Further, we require the following condition:

$$\begin{aligned} &\forall Ms \in AgreementSet \,;\, Ms' \in Message^* \bullet \\ &\quad Ms \neq Ms' \Rightarrow f(Ms) \neq f(Ms'). \end{aligned} \tag{3}$$

That is, f does not map a different sequence Ms' of messages to the same value as a member Ms of $AgreementSet$.

Lemma 6 If equation (3) is satisfied, then if a particular trace tr constitutes a failure of agreement authentication on the original protocol, then $f(tr)$ constitutes a failure of agreement authentication on the simplified protocol:

$$\neg Agreement_{AgreementSet}(tr) \ \Rightarrow \ \neg Agreement_{f(AgreementSet)}(f(tr)).$$

Proof: Suppose tr constitutes a failure of agreement authentication on the original protocol: $\neg Agreement_{AgreementSet}(tr)$. Then for some $A \in Agent$, $B \in Honest$, $Ms \in AgreementSet$, we have:

$$tr \downarrow finished.A.B.Ms \ > \ tr \downarrow running.B.A.Ms.$$

By the condition (3) on f:

$$\begin{aligned} f(tr) \downarrow finished.A.B.f(Ms) \ &= \ tr \downarrow finished.A.B.Ms, \\ f(tr) \downarrow running.B.A.f(Ms) \ &= \ tr \downarrow running.B.A.Ms, \end{aligned}$$

and so

$$f(tr) \downarrow finished.A.B.f(Ms) \;>\; f(tr) \downarrow running.B.A.f(Ms).$$

That is, $\neg Agreement_{f(AgreementSet)}(f(tr))$. □

The reason for condition (3) about f can be explained by the following example: suppose the protocol aims to achieve agreement upon a key, so $AgreementSet$ is the set of all singleton sequences of keys; and suppose the function f maps two different keys K_1 and K_2 to the same value, so $f(\langle K_1\rangle) = f(\langle K_2\rangle)$ and condition (3) is not satisfied; then there might be some attack on the original protocol where the two honest agents end up accepting these different keys K_1 and K_2, but this would not be recognized as an attack in the simplified protocol, because the two agents would use the same key $f(K_1) = f(K_2)$.

The following corollary shows that such transformations are fault-preserving with respect to the agreement property; it is proved analogously to Corollary 5:

Corollary 7 If equations (1), (2) and (3) are satisfied, then:

$$\begin{aligned}&SYSTEM' \;\mathbf{sat}\; Agreement_{f(AgreementSet)} \Rightarrow\\ &\qquad SYSTEM \;\mathbf{sat}\; Agreement_{AgreementSet}\,.\end{aligned}$$

4 Examples of fault-preserving renaming transformations

In this section we present a number of renaming transformations that satisfy the conditions from the previous section, and are hence fault-preserving.

4.1 Removing encryptions

We begin by defining a class of transformations that removes some of the encryptions from the protocol. We identify a set $Encs$ of encrypted messages such that all encrypted components $\{M\}_K$ from $Encs$ that appear in the protocol should be replaced with the body M. We define the transformation function f as follows:

$$\begin{aligned}
f(\textsc{Atom}\,A) &\mathrel{\widehat{=}} \textsc{Atom}\,A,\\
f(M, M') &\mathrel{\widehat{=}} NF(f(M), f(M')),\\
f(\{M\}_K) &\mathrel{\widehat{=}} \begin{cases} f(M), & \text{if } \{M\}_K \in Encs,\\ \{f(M)\}_K, & \text{otherwise,}\end{cases}\\
f(g(\!|M|\!)) &\mathrel{\widehat{=}} g(\!|f(M)|\!).
\end{aligned}$$

The interesting case is the case for $\{M\}_K$ with $\{M\}_K \in Encs$, where the outermost encryption is stripped off; most of the other cases simply apply the function recursively; the "NF" in the definition for pairs is necessary to ensure that the messages returned by the function are in normal form.

We define the intruder's initial knowledge in the transformed protocol as the result of applying the simplifying function to his knowledge in the original protocol:

$$IIK' \mathrel{\widehat{=}} f(IIK).$$

This clearly satisfies condition (2). In [4] we show that this transformation also satisfies the distribution property, condition (1). Hence, we can use Corollary 5 to deduce that this transformation is a fault-preserving transformation with respect to secrecy:

$$\mathit{SYSTEM}' \,\mathbf{sat}\, \mathit{Secrecy} \;\Rightarrow\; \mathit{SYSTEM} \,\mathbf{sat}\, \mathit{Secrecy}.$$

Further, f is clearly injective on atoms, so we can use Corollary 7 to deduce that this transformation is a fault-preserving transformation with respect to agreement on atoms:

If *AgreementSet* is a set of sequences of atoms, then

$$\begin{aligned} &\mathit{SYSTEM}' \,\mathbf{sat}\, \mathit{Agreement}_{f(\mathit{AgreementSet})} \Rightarrow \\ &\qquad \mathit{SYSTEM} \,\mathbf{sat}\, \mathit{Agreement}_{\mathit{AgreementSet}}. \end{aligned}$$

The way this simplifying transformation will be used in practice is to identify a particular encryption in the protocol description that we think is unnecessary, and to take f to be the function that removes this encryption; that is, we take *Encs* to be the set of all instances of the encrypted component in question. Note though, that if there is a second encrypted component in the protocol description whose instances are of the same type as the instances of the original encryption, then the instances of the second encrypted component are also in *Encs*, and so we must also strip off the encryptions on these instances.

4.2 Removing hash functions

We now define a class of transformations that removes some of the hash functions from the protocol. The definition is very similar to that for removing encryptions: we identify a set *Encs* of applications of hash functions such that all components $g(\!|M|\!)$ from *Encs* should be replaced with the body M. We define the transformation function f as follows:

$$\begin{aligned} f(\textsc{Atom}\,A) &\mathrel{\widehat{=}} \textsc{Atom}\,A, \\ f(M, M') &\mathrel{\widehat{=}} NF(f(M), f(M')), \\ f(\{M\}_K) &\mathrel{\widehat{=}} \{f(M)\}_K, \\ f(g(\!|M|\!)) &\mathrel{\widehat{=}} \begin{cases} f(M), & \text{if } g(\!|M|\!) \in \mathit{Encs}, \\ g(\!|f(M)|\!), & \text{otherwise.} \end{cases} \end{aligned}$$

We again define the intruder's initial knowledge in the transformed protocol by:

$$IIK' \mathrel{\widehat{=}} f(IIK),$$

which satisfies condition (2). In [4], we prove that the transformation satisfies the distribution property, condition (1). Hence, as with the previous transformation, we can deduce that this is a fault-preserving transformation with respect to secrecy, and with respect to agreement on atoms.

4.3 Removing some atomic or hashed fields

We next define a class of transformations that completely remove some fields from the protocol. It turns out that we must restrict these fields to being either atoms that are not keys, or applications of hash functions. Formally, we let

$$Ms \subseteq Message$$

be the set of such messages, and assume:

$$\begin{aligned}\forall M \in Ms \bullet\ & (\exists A \in Atom \bullet M = \textsc{Atom}\, A \wedge A \notin Key) \\ & \vee (\exists g \in HashFn\,;\, M' \in Msg \bullet M = g(\!| M' |\!)).\end{aligned}$$

The reason we allow only atoms and applications of hash functions to be removed is that the intruder would be unable to use such messages to deduce any other messages other than those containing this message itself; for example, removing a message of the form $\{M\}_K$ would remove a possible means of the intruder learning M. We do not allow keys to be removed, for clearly this too would reduce the intruder's abilities.

In order to define this transformation, we need to specify how it acts upon a message that is to be removed, leaving an "empty message". The $Message$ datatype has no concept of an "empty message", so we model such "empty messages" by a distinguished atomic value $\mathsf{nil} \in Atom$.

We define the transformation as follows:

$$\begin{aligned}
f(M) &\mathrel{\widehat{=}} \textsc{Atom}\,\mathsf{nil}, && \text{if } M \in Ms,\\
f(\textsc{Atom}\,A) &\mathrel{\widehat{=}} \textsc{Atom}\,A, &&\\
f(M, M') &\mathrel{\widehat{=}} \begin{cases} f(M'), & \text{if } f(M) = \textsc{Atom}\,\mathsf{nil},\\ f(M), & \text{if } f(M') = \textsc{Atom}\,\mathsf{nil},\\ (f(M), f(M')), & \text{otherwise,}\end{cases}\\
f(\{M\}_K) &\mathrel{\widehat{=}} \begin{cases} \textsc{Atom}\,\mathsf{nil}, & \text{if } f(M) = \textsc{Atom}\,\mathsf{nil},\\ \{f(M)\}_K, & \text{otherwise,}\end{cases}\\
f(g(\!| M |\!)) &\mathrel{\widehat{=}} \begin{cases} \textsc{Atom}\,\mathsf{nil}, & \text{if } f(M) = \textsc{Atom}\,\mathsf{nil},\\ g(\!| f(M) |\!), & \text{otherwise.}\end{cases}
\end{aligned}$$

We adopt the convention from functional programming languages such as Haskell that the patterns in the definition are matched from top to bottom, so that a definition further down the page will be applied only if no earlier definition applies; this means that the line with the pattern "$\textsc{Atom}\,A$" applies only to atoms that are not matched by the first line, i.e. only to atoms not in Ms.

For composite messages, the definition gets applied recursively to the subcomponents; if either member of a pair gets removed in this way, then we are left with the other member of the pair; if all of the body of an encryption or a hash function application gets removed, then the encryption or hash function is removed.

We define the intruder's initial knowledge in the transformed protocol by:

$$IIK' \mathrel{\widehat{=}} f(IIK) \cup \{\textsc{Atom}\,\mathsf{nil}\}.$$

Note that we assume the intruder can produce $\textsc{Atom}\,\mathsf{nil}$, i.e. an empty message.

Our definitions satisfy conditions (1) and (2). We hence deduce that this is a fault-preserving transformation with respect to secrecy.

Note that the transformation is fault-preserving even if we remove a field S that we are trying to prove is secret. Recall that in order to test whether S is kept secret in the original protocol, we need to test whether $f(S)$ is kept secret in the transformed protocol. But in this case, $f(S) = \textsc{Atom}\,\mathsf{nil}$, which is in the intruder's initial knowledge, and so this transformation will introduce a new attack. Such a transformation is fault-preserving but rather pointless.

If all the values upon which agreement is intended are atoms, and none are removed, i.e.:

$$\begin{array}{l} \forall\, Ms' \in \mathit{AgreementSet} \cdot \forall\, M \text{ in } Ms' \cdot \\ \quad \exists\, A \in \mathit{Atom} \cdot M = \textsc{Atom}\, A \wedge M \notin Ms, \end{array}$$

then the transformation satisfies condition (3), and so we deduce that this is a fault-preserving transformation with respect to agreement on such values. The above condition simply states that if you want to be sure that the protocol guarantees agreement on a particular field, then your simplification shouldn't remove that field!

In practice, we will use this transformation to remove *all* fields of a particular type: to remove one field we would have to define Ms to be all values with which that field could be instantiated, i.e. all values of that type; this would have the effect of removing all fields of the type.

4.4 Renaming atoms

We now consider a class of transformations that uniformly rename atoms in protocol messages. Let f be a renaming function on atoms:

$$f \;:\; \mathit{Atom} \rightarrow \mathit{Atom},$$

which respects inverses of keys:

$$\forall\, K \in \mathit{Key} \cdot f(K)^{-1} = f(K^{-1}).$$

We can lift f to operate over complete messages, in the obvious way:

$$\begin{array}{rcl} f(\textsc{Atom}\, A) & \mathrel{\widehat=} & \textsc{Atom}\, f(A), \\ f(M, M') & \mathrel{\widehat=} & (f(M), f(M')), \\ f(\{M\}_K) & \mathrel{\widehat=} & \{f(M)\}_{f(K)}, \\ f(g(\!|M|\!)) & \mathrel{\widehat=} & g(\!|f(M)|\!). \end{array}$$

And we can define the intruder's initial knowledge in the transformed protocol in the normal way:

$$IIK' \mathrel{\widehat=} f(IIK).$$

This definition clearly satisfies condition (2). The proof of condition (1) is in [4]. Hence we deduce that this is a fault-preserving transformation with respect to secrecy. It is also a fault-preserving transformation with respect to the agreement property provided f is injective over the messages upon which agreement is intended.

4.5 Swapping pairs

We now define a class of transformations that swaps particular pairs of messages when they appear next to one another. Let $Pairs$ be a set of pairs of messages:

$$Pairs : \mathbb{P}(Message \times Message).$$

We define a transformation that, for every pair $(M, M') \in Pairs$, swaps M and M' whenever they occur adjacently within a message. Within compound messages, these swaps are applied from left to right, and from the outside in.

$$\begin{aligned}
f(\textsc{Atom}\, A) &\mathrel{\widehat{=}} \textsc{Atom}\, A, \\
f(M, M') &\mathrel{\widehat{=}} \text{if } (M, M') \in Pairs \text{ then } (f(M'), f(M)) \\
&\qquad \text{else } (f(M), f(M')), \\
&\qquad\quad \text{for } \neg pair(M'), \\
f(M, (M', M'')) &\mathrel{\widehat{=}} \text{if } (M, M') \in Pairs \text{ then } (f(M'), f(M, M'')) \\
&\qquad \text{else } (f(M), f(M', M'')), \\
f(\{M\}_K) &\mathrel{\widehat{=}} \{f(M)\}_K, \\
f(g(\!|M|\!)) &\mathrel{\widehat{=}} g(\!|f(M)|\!).
\end{aligned}$$

We can define the intruder's initial knowledge in the transformed protocol in the obvious way:

$$IIK' \mathrel{\widehat{=}} f(IIK).$$

Clearly this definition satisfies condition (2). It can be shown that condition (1) is also satisfied. Hence we deduce that this is a fault-preserving transformation with respect to secrecy. Further, f is injective on atoms, so this is a fault-preserving transformation with respect to agreement on atoms.

4.6 Coalescing messages

Our next class of transformation coalesces certain pairs of messages, replacing them by the first. We define the transformation with respect to a set of pairs of messages:

$$Pairs : \mathbb{P}(Message \times Message),$$

the idea being that for each $(M, M') \in Pairs$, any occurrence of the pair (M, M') in the protocol is replaced by M.

We formalize the transformation as follows:

$$\begin{aligned}
f(\textsc{Atom}\, A) &\mathrel{\widehat{=}} \textsc{Atom}\, A, \\
f(M, M') &\mathrel{\widehat{=}} \text{if } (M, M') \in Pairs \text{ then } f(M) \text{ else } (f(M), f(M')), \\
&\qquad\quad \text{for } \neg pair(M'), \\
f(M, (M', M'')) &\mathrel{\widehat{=}} \text{if } (M, M') \in Pairs \text{ then } (f(M), f(M'')) \\
&\qquad \text{else } (f(M), f(M', M'')), \\
f(\{M\}_K) &\mathrel{\widehat{=}} \{f(M)\}_K, \\
f(g(\!|M|\!)) &\mathrel{\widehat{=}} g(\!|f(M)|\!).
\end{aligned}$$

We define the intruder's initial knowledge in the transformed system by:

$$IIK' \mathrel{\widehat{=}} f(IIK).$$

It turns out that in order for this transformation to be fault-preserving, we need the following additional condition upon $Pairs$:

$$\forall (M, M') \in Pairs \bullet IIK' \cup \{f(M)\} \vdash f(M'),$$

that is, the message removed can be deduced from the message remaining and the intruder's initial knowledge. If we did not have this condition, then removing M' would remove a possible source of information for the intruder. We will therefore tend to use this transformation in two situations: where the message removed is immediately producible from the intruder's initial knowledge, or where the message removed is equal to the one remaining.

Our definitions satisfy conditions (1) and (2). Hence we deduce that this is a fault-preserving transformation with respect to secrecy. It is also a fault-preserving transformation with respect to the agreement property upon atoms provided the atoms in question are not coalesced away.

4.7 Removing fields from the bodies of encryptions

We now define a class of transformation that allows some fields to be pulled out of the body of an encryption; for example a message of the form $\{(M_1, M_2)\}_K$ could be simplified to $(M_1, \{M_2\}_K)$. We define the transformation with respect to a set of messages $Encs$:

$$Encs \subseteq Message,$$

each member of which is an encryption of a pair:

$$\forall\, M \in Encs \bullet \exists\, M_1, M_2 \in Message \,;\, K \in Key \bullet M = \{(M_1, M_2)\}_K.$$

The transformation is then defined as follows:

$$\begin{aligned}
f(\textsc{Atom}\, A) &\mathrel{\widehat{=}} \textsc{Atom}\, A,\\
f(M, M') &\mathrel{\widehat{=}} NF(f(M), f(M')),\\
f(\{M\}_K) &\mathrel{\widehat{=}} \begin{cases} NF(f(M_1), \{f(M_2)\}_K), & \\ & \text{if } \{M\}_K \in Encs \text{ and } M = (M_1, M_2),\\ \{f(M)\}_K, & \text{otherwise,} \end{cases}\\
f(g(\!|M|\!)) &\mathrel{\widehat{=}} g(\!|f(M)|\!).
\end{aligned}$$

We define the intruder's initial knowledge in the transformed protocol in the normal way:

$$IIK' \mathrel{\widehat{=}} f(IIK).$$

This definition satisfies conditions (1) and (2). Hence we deduce that this is a fault-preserving transformation with respect to secrecy, and as f is injective on atoms, this is a fault-preserving transformation with respect to agreement on atoms.

4.8 Simplifying signatures

Finally, we define a class of transformations that is not applicable to the CyberCash Protocol, but which we found useful when simplifying the SET protocol [20]. The SET protocol contains many messages of the form $M, \{h(M)\}_K$, where h is a hash function, and K is a signature key; the term $\{h(M)\}_K$ acts as a signature for M. The "removing hash functions" transformation from Section 4.2 allows us to remove the h. We define here a transformation that allows us to simplify such a message further, to $\{M\}_K$.

We define the transformations with respect to a set *Encs*. We assume all the messages in *Encs* are of the form $\{M\}_K$ where $\textsc{Atom}\, K^{-1} \in IIK$. We assume further that all the bodies of these encryptions have the same *nesting factor*, that is the same amount of nested pairing. Formally, we define the nesting factor of a message as follows:

$$\begin{aligned} \mathsf{nesting\text{-}factor}(M, M') &\;\widehat{=}\; 1 + \mathsf{nesting\text{-}factor}(M'), \\ \mathsf{nesting\text{-}factor}(M) &\;\widehat{=}\; 0 \quad \text{for } \neg pair(M). \end{aligned}$$

Our assumption about *Encs* is then:

$$\begin{aligned} &\exists\, n : \mathbb{N} \bullet \forall\, M \in Encs \bullet \exists\, M', K \bullet \\ &\quad M = \{M'\}_K \wedge \textsc{Atom}\, K^{-1} \in IIK \wedge \mathsf{nesting\text{-}factor}(M') = n. \end{aligned}$$

The condition about the encryptions all having the same nesting factor will make our definition of the transformation clearer. In practice, it is not a significant restriction, because if one wanted to perform a transformation corresponding to several different nesting factors, then one would simply have to perform several transformations, one for each nesting factor.

Let *Encs* be as above, and suppose the nesting factor of the bodies of the messages in *Encs* is n; this means that all members of *Encs* will be of the form $\{(M_0, (M_1, (\ldots, M_n)) \ldots)\}_K$, allowing us to pattern match. We can define the transformation as follows:

$$\begin{aligned} &f(\textsc{Atom}\, A) \,\widehat{=}\, \textsc{Atom}\, A, \\ &f(M_0, (M_1, (\ldots, (M_n, \{(M_0, (M_1, (\ldots, M_n)) \ldots)\}_K)) \ldots) \,\widehat{=} \\ &\quad f(\{(M_0, (M_1, (\ldots, M_n)) \ldots)\}_K) \\ &\qquad\quad \text{for } \{(M_0, (M_1, (\ldots, M_n)) \ldots)\}_K \in Encs, \\ &f(M_0, (M_1, (\ldots, (M_n, (\{(M_0, (M_1, (\ldots, M_n)) \ldots)\}_K, M)) \ldots) \,\widehat{=} \\ &\quad f(\{(M_0, (M_1, (\ldots, M_n)) \ldots)\}_K, M) \\ &\qquad\quad \text{for } \{(M_0, (M_1, (\ldots, M_n)) \ldots)\}_K \in Encs, \\ &f(M, M') \,\widehat{=}\, (f(M), f(M')), \\ &f(\{M\}_K) \,\widehat{=}\, \{f(M)\}_K, \\ &f(g(\!|M|\!)) \,\widehat{=}\, g(\!|f(M)|\!). \end{aligned}$$

The second and third cases are the interesting ones, where the transformation has a direct effect, removing the plaintext fields $M_0, \ldots, M_n$, respectively in the cases where the encryption is not or is followed by further fields. We again use the convention that definitions are pattern matched from top to bottom, so the fourth case will deal with all pairs not matched by the interesting cases.

We define the intruder's initial knowledge in the transformed protocol in the normal way:

$$IIK' \mathrel{\widehat{=}} f(IIK).$$

This definition satisfies conditions (1) and (2). Hence we deduce that this is a fault-preserving transformation with respect to secrecy, and as f is injective on atoms, this is a fault-preserving transformation with respect to agreement on atoms.

5 Structural transformations

We now consider two classes of transformations that are not formed by simple renaming of messages within the protocol, but instead change the structure of the protocol; we consider a transformation that splits a single message into two, and another that replaces two messages with a single message, redirecting a message that is sent via a third party so that it is sent direct.

5.1 Splitting messages

We begin by describing message splitting transformations. The idea is that if we have a message (M_1, M_2), then we replace this message with a pair of messages, M_1 and M_2 respectively.

Such a transformation can be advantageous when model checking, because it reduces the size of the message space (the number of distinct messages in the model); however, there is a trade-off, because it increases the size of the state space; our experiences using the model checker FDR suggests that it is a worthwhile transformation for moderately large messages. It is perhaps more useful when used in combination with a redirecting transformation, defined in the next section.

We define the transformation with respect to a set $Pairs$ of pairs of messages, the idea being that $Pairs$ represents those messages that should be split. The definition of the transformation is as a function f over traces:

$$\begin{aligned}
&f(\langle\rangle) \mathrel{\widehat{=}} \langle\rangle,\\
&f(\langle send.A.B.(M_1, M_2)\rangle^\frown tr') \mathrel{\widehat{=}}\\
&\qquad \langle send.A.B.M_1, send.A.B.M_2\rangle^\frown f(tr'), \quad \text{for } (M_1, M_2) \in Pairs,\\
&f(\langle receive.A.B.(M_1, M_2)\rangle^\frown tr') \mathrel{\widehat{=}}\\
&\qquad \langle receive.A.B.M_1, receive.A.B.M_2\rangle^\frown f(tr'), \quad \text{for } (M_1, M_2) \in Pairs,\\
&f(\langle e\rangle^\frown tr') \mathrel{\widehat{=}} \langle e\rangle^\frown f(tr').
\end{aligned}$$

The final case deals with all cases other than those messages that we split, including sending and receiving of messages, and those signal events used for specifying security properties.

We aim to show that if tr is an attack upon the original protocol, then $f(tr)$ is an attack upon the transformed protocol.

We now define the behaviours of an honest agent A in the transformed system; we define the process P'_A representing this agent, corresponding to the process P_A in the original system, to be a process such that:

$$\forall\, tr \in traces(P_A) \bullet f(tr) \in traces(P'_A).$$

What this means is that whenever the process P_A sends or receives a message (M, M') from $Pairs$, the process P'_A will send or receive the pair of messages M and M'.

The following lemma shows that the collection of honest agents, as a whole, satisfies a similar property; it can be proved directly from the above definition and the semantic definition of interleaving:

Lemma 8 $\forall\, tr \in traces(|||_{A \in Honest}\, P_A) \bullet f(tr) \in traces(|||_{A \in Honest}\, P'_A)$.

We now aim to prove a similar result about the traces of the intruder. We will need the following lemma, showing that the intruder can do everything from having seen a pair of messages that he can do from the corresponding paired message:

Lemma 9

$$\begin{array}{l} \forall\, tr \in traces(INTRUDER(S \cup \{(M_1, M_2)\})) \bullet \\ \qquad tr \in traces(INTRUDER(S \cup \{M_1, M_2\})). \end{array}$$

Proof: This lemma can be proved using a simple induction over traces, making use of the following property of the derivation relation:

$$S \cup \{M_1, M_2\} \vdash M \;\Leftrightarrow\; S \cup \{(M_1, M_2)\} \vdash M.$$

□

We make use of this lemma to prove the following result about the traces of the intruder.

Lemma 10

$$\forall\, tr \in traces(INTRUDER(S)) \bullet f(tr) \in traces(INTRUDER(S)).$$

Proof: By induction on tr. The interesting cases are where we have a message $(M_1, M_2) \in Pairs$.

- Case $tr = \langle send.A.B.(M_1, M_2)\rangle ^\frown tr'$ and $(M_1, M_2) \in Pairs$. Then

$$\begin{array}{ll} & tr \in traces(INTRUDER(S)) \\ \Rightarrow & \langle \text{intruder definition} \rangle \\ & tr' \in traces(INTRUDER(S \cup \{(M_1, M_2)\})) \\ \Rightarrow & \langle \text{by the inductive hypothesis} \rangle \\ & f(tr') \in traces(INTRUDER(S \cup \{(M_1, M_2)\})) \\ \Rightarrow & \langle \text{Lemma 9} \rangle \\ & f(tr') \in traces(INTRUDER(S \cup \{M_1, M_2\})) \\ \Rightarrow & \langle \text{intruder definition} \rangle \\ & \langle send.A.B.M_1, send.A.B.M_2\rangle ^\frown f(tr') \in traces(INTRUDER(S)) \\ \Rightarrow & f(tr) \in traces(INTRUDER(S)). \end{array}$$

- Case $tr = \langle receive.A.B.(M_1, M_2)\rangle \frown tr'$ and $(M_1, M_2) \in Pairs$.

$$
\begin{aligned}
& tr \in traces(INTRUDER(S)) \\
\Rightarrow\ & \big\langle \text{intruder definition} \big\rangle \\
& S \vdash (M_1, M_2) \wedge tr' \in traces(INTRUDER(S)) \\
\Rightarrow\ & \big\langle \textbf{splitting} \text{ rule; inductive hypothesis} \big\rangle \\
& S \vdash M_1 \wedge S \vdash M_2 \wedge f(tr') \in traces(INTRUDER(S)) \\
\Rightarrow\ & \big\langle \text{intruder definition} \big\rangle \\
& \langle receive.A.B.M_1, receive.A.B.M_2\rangle \frown f(tr') \in \\
& \quad traces(INTRUDER(S)) \\
\Rightarrow\ & f(tr) \in traces(INTRUDER(S)).
\end{aligned}
$$

□

We can now consider the system as a whole. Let $SYSTEM$ be formed from the parallel composition of the intruder and the honest agents represented by the P_A processes, as in Section 2, and let $SYSTEM'$ be similarly formed from the intruder and the P'_A processes. The following theorem relates the traces of the two systems.

Theorem 11 $\forall\, tr \in traces(SYSTEM) \bullet f(tr) \in traces(SYSTEM')$.

Proof: This follows from Lemmas 8 and 10, and the definition of parallel composition. □

We now want to show that every attack on the original system is "matched" by one on the new system.

Firstly, the transformation preserves attacks on the secrecy specification:

Lemma 12 $\neg Secrecy(tr) \Rightarrow \neg Secrecy(f(tr))$.

Proof: If we have a breach of secrecy on the original protocol, $\neg Secrecy(tr)$, then we have a $claimSecret.A.B.M$ and $leak.M$ event occurring in tr. Hence as the transformation preserves the signal events, we have the same $claimSecretA.B.M$ and $leak.M$ in $f(tr)$, which is a breach of secrecy on the simplified protocol, $\neg Secrecy(f(tr))$. □

Hence we can show this transformation is fault-preserving for the secrecy property.

Corollary 13 $SYSTEM'$ **sat** $Secrecy \Rightarrow SYSTEM$ **sat** $Secrecy$.

We now consider the agreement authentication specification. The signal events are preserved by the transformation, so the proof takes a very similar form to that for secrecy:

Lemma 14 $\neg Agreement_{AgreementSet}(tr) \Rightarrow \neg Agreement_{AgreementSet}(f(tr))$.

Corollary 15 $SYSTEM'$ **sat** $Agreement_{AgreementSet} \Rightarrow SYSTEM$ **sat** $Agreement_{AgreementSet}$.

One thing to note is that the placement of the signal events in the transformed protocol should correspond to their placement in the original protocol. However, this might not be the natural placement for the signals in the transformed protocol. For instance, suppose we are considering whether A is authenticated to B, and the final message of the protocol is from A to B and is one of those that is split. As described in Section 2.4, the normal placement for A's *running* event would be just before this message; hence in the transformed protocol the *running* event should occur before the corresponding pair of messages; but this does not correspond to the natural placement of this event, which would be between these two messages.

5.2 Redirecting messages

The final class of transformations we consider is what we call *redirecting transformations*. In many protocols, some information is to be sent from an agent A to another agent B; however, rather than sending this message direct, the message is sent via some third party C; C does nothing with this message except forward it:

$$\begin{array}{ll} \mathit{ac}. & A \rightarrow C \ : \ M \\ \mathit{cb}. & C \rightarrow B \ : \ M \end{array}$$

(where *ac* and *cb* are message numbers; the names are intended as mnemonics for the sender and receiver). Our simplification will adapt the protocol so that this message is sent direct:

$$\mathit{ab}. \quad A \rightarrow B \ : \ M \ .$$

Such a transformation produces very significant gains when model checking, because in the original protocol, it is possible to interleave C's events with A's and B's in many different ways; further, C should be willing to accept any message for M, because it performs no checks on this message, which markedly increases the state space. An experiment applying this transformation to the Otway Rees protocol reduced the time FDR took to analyze a system from 7 minutes to 7 seconds.

In order to define this transformation, we adapt our model of protocols slightly, so as to include message numbers within the events representing those messages; for example, we write *send.ac.A.C.M* to represent A sending the message *ac*, above.

We define the transformation with respect to particular values of *ac*, *cb*, and *ab*. We assume that all traces *tr* of the original protocol conform to the above description, so if an agent C sends the message *cb*, then he must previously have received the message *ac*:

$$\begin{array}{l} \forall\, tr', C, B, M \bullet \\ \qquad tr' \frown \langle \mathit{send.cb.C.B.M} \rangle \leq tr \Rightarrow \exists\, A \bullet \mathit{receive.ac.A.C.M} \;\mathrm{in}\; tr'. \end{array} \tag{4}$$

We define the transformation as a function over traces. The send event for message *ac* is renamed to the send event for message *ab*; the receive event for message cb is renamed to the receive event for message ab; the receive and send events of agent C are removed.

$$
\begin{aligned}
f(\langle\rangle) &\mathrel{\widehat{=}} \langle\rangle, \\
f(tr' {}^\frown \langle send.ac.A.C.M\rangle) &\mathrel{\widehat{=}} f(tr') {}^\frown \langle send.ab.A.B.M\rangle, \\
f(tr' {}^\frown \langle receive.ac.A.C.M\rangle) &\mathrel{\widehat{=}} f(tr'), \\
f(tr' {}^\frown \langle send.cb.C.B.M\rangle) &\mathrel{\widehat{=}} f(tr'), \\
f(tr' {}^\frown \langle receive.cb.C.B.M\rangle) &\mathrel{\widehat{=}} f(tr') {}^\frown \langle receive.ab.A.B.M\rangle, \\
f(tr' {}^\frown \langle e\rangle) &\mathrel{\widehat{=}} f(tr') {}^\frown \langle e\rangle.
\end{aligned}
$$

We define f working from the end of the trace, rather than the beginning, to simplify proofs (particularly Lemma 17) about the intruder, below.

As with the previous transformation, we take the above function to define the behaviour of the honest agents in the transformed system. Hence, if P'_A is the process representing an agent A in the transformed system, corresponding to process P_A in the original system, then

$$\forall\, tr \in traces(P_A) \bullet f(tr) \in traces(P'_A).$$

The following result lifts this property to the collection of all the honest agents:

Lemma 16 $\forall\, tr \in traces(|||_{A \in Honest} P_A) \bullet f(tr) \in traces(|||_{A \in Honest} P'_A)$.

Proof: This is an induction over the trace tr, using the semantic definition of interleaving, and performing a case analysis based on the definition of f. □

We now seek to prove a similar result for the intruder. It is convenient to use a slightly different characterization of the intruder. We define $\mathsf{data}(tr)$ to be the set of messages sent by honest agents during trace tr:

$$\mathsf{data}(tr) \mathrel{\widehat{=}} \{M \mid send.n.A.B.M \text{ in } tr\}.$$

We can then characterize the traces of the intruder by:

$$
\begin{aligned}
& tr \in traces(INTRUDER(IIK)) \Leftrightarrow \\
& \quad \forall\, tr', n, A, B, M \bullet \\
& \qquad \begin{pmatrix} tr' {}^\frown \langle receive.n.A.B.M\rangle \le tr \\ \vee\ tr' {}^\frown \langle leak.M\rangle \le tr \end{pmatrix} \Rightarrow IIK \cup \mathsf{data}(tr') \vdash M \\
& \quad \wedge\ \alpha(tr) \subseteq \{|send, receive, leak|\},
\end{aligned}
\tag{5}
$$

that is, if the intruder can perform a *receive* or *leak* event, then the contents of that event must be deducible from the preceding trace and the intruder's initial knowledge. This characterization can be deduced from the earlier model via a straightforward induction over tr.

The following lemma shows that the intruder can produce the same messages in corresponding situations in the two systems, i.e. after tr or $f(tr)$.

Lemma 17 If trace tr satisfies Equation 4, then

$$\forall\, S, M \bullet S \cup \mathsf{data}(tr) \vdash M \Rightarrow S \cup \mathsf{data}(f(tr)) \vdash M.$$

Proof: By induction over tr. The interesting case is when the last event of tr is a *send* event that the transformation removes, i.e. $tr = tr' \frown \langle send.cb.C.B.M' \rangle$, so $f(tr) = f(tr')$. Note that Equation 4 implies that $receive.ac.A.C.M'$ in tr', for some A, and so $S \cup \mathsf{data}(tr') \vdash M'$.

$$
\begin{array}{ll}
 & S \cup \mathsf{data}(tr) \vdash M \\
\Rightarrow & \big\langle \text{from the above; } \mathsf{data}(tr) = \{M'\} \cup \mathsf{data}(tr') \big\rangle \\
 & S \cup \mathsf{data}(tr') \vdash M' \wedge S \cup \{M'\} \cup \mathsf{data}(tr') \vdash M \\
\Rightarrow & \big\langle \text{by the inductive hypothesis} \big\rangle \\
 & S \cup \mathsf{data}(f(tr')) \vdash M' \wedge S \cup \{M'\} \cup \mathsf{data}(f(tr')) \vdash M \\
\Rightarrow & \big\langle \text{Lemma 1; } f(tr') = f(tr) \big\rangle \\
 & S \cup \mathsf{data}(f(tr)) \vdash M.
\end{array}
$$

□

We can now prove that the traces of the intruder satisfy the required inclusion:

Lemma 18 If trace tr satisfies Equation 4, and $tr \in traces(INTRUDER(S))$, then $f(tr) \in traces(INTRUDER(S))$.

Proof: This is an induction on the length of the trace tr using the previous lemma and Equation 5. □

Hence we can prove the corresponding result for the system as a whole. Let $SYSTEM$ and $SYSTEM'$ be as before.

Theorem 19 $\forall\, tr \in traces(SYSTEM) \cdot f(tr) \in traces(SYSTEM')$.

Proof: Note that if $tr \in traces(SYSTEM)$ then the restriction of tr to the intruder's alphabet satisfies Equation 4; hence the conditions of Lemma 18 are satisfied. We can then make use of this lemma and Lemma 16, and prove the result using the definition of parallel composition. □

Finally, we consider attacks. Note that the events used to specify security properties are preserved by the transformation. Hence we can use the same argument as in Lemmas 12 and 14, above, to prove that if trace tr is an attack against either secrecy or agreement, then so is $f(tr)$. Hence this transformation is fault-preserving with respect to these security properties.

As with the splitting transformation, the placement of the signal events in the transformed protocol should correspond to their placement in the original protocol, which might not be their natural placement. For instance, consider the case of testing whether C is authenticated to B in the example protocol, in the case that message cb is the final message of the protocol. The normal placement for C's *running* event would be just before message cb; hence the *running* event should occur at the end of C's run in the transformed protocol; but this does not correspond to the natural placement of this event, which would be earlier.

6 Case study: The CyberCash Main Sequence Protocol

We illustrate our simplifying transformations by sketching how they can be used to simplify the CyberCash Main Sequence Protocol. A complete description of the protocol is included in Appendix A, taken from [1]; the following discussion should be read in conjunction with this protocol description.

The protocol is designed to allow credit card purchases to take place securely between a customer and merchant, with the aid of CyberCash who acts as a trusted third party. The customer and merchant each generate a session key, which they encrypt with CyberCash's public key; they then use this session key to encrypt data sent to CyberCash, for secrecy. They each sign some data with a secret key, for authentication. CyberCash verifies the transaction, and sends acknowledgements to each, encrypted with the appropriate session key.

The goals of the protocol are not well documented, but it is reasonable to assume that the credit card is intended to remain secret, that the customer and merchant should agree on the details of the transaction and the amount payable, and that the customer and merchant should be authenticated to CyberCash.

In order to make the simplifying transformations easier to apply (particularly the swapping and coalescing pairs transformations), we have produced tool support as an extension to Casper [8]. The tool takes in a description of the complete protocol and a specification of a sequence of transformations, and applies them in order.

6.1 Tool Support

The syntax for the protocol description is precisely as in Casper; we describe the syntax for defining the transformations below.

The transformations in section 4 were defined using values; however, protocols are normally described in terms of variables, which can be instantiated with different values (of the appropriate types) in different runs. We want to define transformations that are applied uniformly, independent of the values with which variables are instantiated. Therefore, the tool supports transformations defined in terms of types, and applies these transformations to all variables and values of the appropriate types.

Here are some examples of simplifying transformations as they should appear in the input file:

```
#Simplifications

RemoveFields [Nonce, TimeStamp]
RemoveEncryption {Nonce, Agent}{PublicKey}
RemoveHash f(Nonce, Nonce, TimeStamp)
Coalesce (Agent, Agent)
SwapPairs (Nonce, Agent)
```

The effect of these transformations is as follows:

`RemoveFields [Nonce, TimeStamp]` This removes every variable of type `Nonce` or `TimeStamp` from the protocol description; it corresponds to the transformation of Section 4.3.

`RemoveEncryption {Nonce, Agent}{PublicKey}` This transformation removes a level of encryption from the protocol description. The notation "`{Nonce, Agent}{PublicKey}`" is intended to represent the set of all messages of the form $\{N, A\}_K$ for N in `Nonce`, A in `Agent` and K in `PublicKey`. This is the transformation of Section 4.1 where *Encs* is taken to be this set.

`RemoveHash f(Nonce, Nonce, TimeStamp)` This transformation removes some hash functions from the protocol description. This is the transformation of Section 4.2 where *Encs* is taken to be the set of all messages of the form $\mathtt{f}(N, N', T)$ for N, N' in `Nonce`, T in `TimeStamp`.

`Coalesce (Agent,Agent)` This transformation coalesces adjacent pairs of agent identities. This is the transformation described in Section 4.6, where *Pairs* is taken to be the set of all pairs of agent identities. At present, the tool only coalesces atoms that are the same type.

`SwapPairs (Nonce, Agent)` This transformation swaps adjacent pairs of nonces and atoms. It is the transformation of Section 4.5 where *Pairs* is taken to be the the set of all (`Nonce`, `Agent`) pairs.

6.2 Simplifying the protocol

The complete protocol contains dozens of fields, but very few seem directly relevant to the security of the protocol. Many fields are included solely for reasons of functionality (e.g. *CardCityOptional*). Our first simplification is therefore a "remove fields" simplification, to remove all atomic fields other than those that we believe are necessary: the fields that are retained are the price of the transaction according to each agent (*Amount* and *MerchantAmount*), the customer's credit card number (*CardNumber*), agents' identities, keys, and transaction numbers (which act as a kind of run identifier). (If we were testing whether the protocol guarantees timeliness, we would want to retain the timestamps.)

The protocol description contains many occurrences of the fields *PrSignedHash_* (also known as *MerchantSignedHash_*) and *PrHash_*. These seem to duplicate security already present, and so appear unnecessary. Several hashes of public keys are passed around by the protocol; these similarly seem to be unnecessary for the security of the protocol. Our second transformation is to remove all of these fields, using the transformation of Section 4.3.

The next simplification is to remove all applications of the hash function $MD5$, using the transformation of Section 4.2; this hash function is simply used to reduce the size of messages, for efficiency.

This produces the following simplified version of the protocol:

$$
\begin{array}{lrl}
1. & M \to C & : MerchantAmount, MerchantCcId \\
2. & C \to M & : Id, MerchantCcID, Transaction, \\
 & & MerchantAmount, MerchantCcId, \\
 & & \{kcs\}_{PKCyberKey}, OpaqueCH1\_ \\
3. & M \to CB & : MerchantCcId, MerchantTransaction, \\
 & & \{kcs\}_{PKCyberKey}, OpaqueCH1\_, \\
 & & \{kms\}_{PKMerchantCyberKey}, MerchantOpaqueCM1\_ \\
4. & CB \to M & : MerchantCcId, MerchantTransaction, \\
 & & \{CardNumber, MerchantAmount, \\
 & & \quad MerchantCcId, Id, Transaction\}_{kms}, \\
 & & \{CardNumber, Amount\}_{kcs} \\
5. & M \to C & : MerchantCcId, Id, \\
 & & Transaction, MerchantAmount, \\
 & & \{CardNumber, Amount\}_{kcs}
\end{array}
$$

where

$$
\begin{array}{l}
OpaqueCH1\_ = \\
\quad \{\ Amount, CardNumber, \\
\quad\quad \{\ Amount, CardNumber, Id, MerchantCcId, \\
\quad\quad\quad MerchantAmount, MerchantCcId, Transaction \\
\quad\quad \}_{SK(C)} \\
\quad \}_{kcs}, \\
MerchantOpaqueCM1\_ = \\
\quad \{\ Id, MerchantAmount, MerchantAmount, \\
\quad\quad MerchantCcId, Transaction, \\
\quad\quad \{\ Id, MerchantAmount, MerchantCcId, \\
\quad\quad\quad MerchantAmount, MerchantCcId, Transaction \\
\quad\quad \}_{SK(M)} \\
\quad \}_{kms}.
\end{array}
$$

There is still nested encryption within $OpaqueCH1\_$ and $MerchantOpaqueCM1\_$: an encryption with a secret, asymmetric key, and an encryption with a symmetric shared key. The encryption with the asymmetric key acts as a signature to provide authentication, whereas the encryption with the symmetric key is to ensure the secrecy of fields such as $CardNumber$. Both these encryptions are essential to the security of the protocol. However, if we consider the two security properties separately, we can perform different simplifications for each, in each case removing one encryption while retaining the other.

Simplifying for secrecy analysis

In order to simplify the protocol for secrecy analysis, we begin by removing the signatures. In the resulting protocol, several fields are repeated within some encryptions. By swapping pairs of atoms around we can bring together like atoms, which can then be coalesced to a single atom. Doing this, produces the following protocol:

$$
\begin{array}{lrcl}
1. & M \to C & : & MerchantCcId, MerchantAmount \\
2. & C \to M & : & Id, MerchantAmount, MerchantCcId, \\
 & & & Transaction, \{kcs\}_{PKCyberKey}, \\
 & & & \{Amount, CardNumber, MerchantCcId, Id, \\
 & & & \quad MerchantAmount, Transaction\}_{kcs} \\
3. & M \to CB & : & MerchantCcId, MerchantTransaction, \{kcs\}_{PKCyberKey}, \\
 & & & \{Amount, CardNumber, MerchantCcId, Id, \\
 & & & \quad MerchantAmount, Transaction\}_{kcs}, \\
 & & & \{kms\}_{PKMerchantCyberKey}, \\
 & & & \{Id, MerchantAmount, MerchantCcId, \\
 & & & \quad MerchantTransaction, Transaction\}_{kms} \\
4. & CB \to M & : & MerchantCcId, MerchantTransaction, \\
 & & & \{CardNumber, MerchantAmount, Id, \\
 & & & \quad MerchantCcId, Transaction\}_{kms}, \\
 & & & \{CardNumber, Amount\}_{kcs} \\
5. & M \to C & : & MerchantCcId, MerchantAmount, Transaction, Id, \\
 & & & \{CardNumber, Amount\}_{kcs} .
\end{array}
$$

Note that the term

$$
\begin{array}{l}
\{kcs\}_{PKCyberKey}, \\
\{Amount, CardNumber, MerchantCcId, Id, \\
\quad MerchantAmount, Transaction\}_{kcs}
\end{array}
$$

appears in both messages 2 and 3. We can use the swapping pairs and splitting transformations to transform these messages to:

$$
\begin{array}{lrcl}
2a. & C \to M & : & Id, MerchantAmount, MerchantCcId, Transaction \\
2b. & C \to M & : & \{kcs\}_{PKCyberKey}, \\
 & & & \{Amount, CardNumber, MerchantCcId, Id, \\
 & & & \quad MerchantAmount, Transaction\}_{kcs} \\
3a. & M \to CB & : & \{kcs\}_{PKCyberKey}, \\
 & & & \{Amount, CardNumber, MerchantCcId, Id, \\
 & & & \quad MerchantAmount, Transaction\}_{kcs} \\
3b. & M \to CB & : & MerchantCcId, MerchantTransaction, \\
 & & & \{kms\}_{PKMerchantCyberKey}, \\
 & & & \{Id, MerchantAmount, MerchantCcId, \\
 & & & \quad MerchantTransaction, Transaction\}_{kms} .
\end{array}
$$

We can then use a redirecting transformation to replace messages 2b and 3a by a single message from C to CB.

Similarly, we can use a combination of swapping, splitting and redirecting so that the term $\{CardNumber, Amount\}_{kcs}$ is sent directly from CB to C.

This gives the following fully simplified version of the protocol:

$$
\begin{array}{llll}
1. & M \to C & : & MerchantCcId, MerchantAmount \\
2a. & C \to M & : & Id, MerchantAmount, \\
 & & & MerchantCcId, Transaction \\
2b. & C \to CB & : & \{kcs\}_{PKCyberKey}, \\
 & & & \{Amount, CardNumber, MerchantCcId, Id, \\
 & & & \quad MerchantAmount, Transaction\}_{kcs} \\
3. & M \to CB & : & MerchantCcId, MerchantTransaction, \\
 & & & \{kms\}_{PKMerchantCyberKey}, \\
 & & & \{Id, MerchantAmount, MerchantCcId, \\
 & & & \quad MerchantTransaction, Transaction\}_{kms} \\
4a. & CB \to M & : & MerchantCcId, MerchantTransaction, \\
 & & & \{CardNumber, MerchantAmount, Id, \\
 & & & \quad MerchantCcId, Transaction\}_{kms} \\
4b. & CB \to C & : & \{CardNumber, Amount\}_{kcs} \\
5. & M \to C & : & MerchantCcId, MerchantAmount, Transaction, Id\ .
\end{array}
$$

Simplifying for authentication analysis

In order to simplify the protocol for authentication analysis, we remove the encryptions using the symmetric keys kcs and kms. We can continue to simplify the protocol as above, using the swapping pairs and coalescing pairs transformations to remove duplicated atoms, and then using the splitting and redirecting transformations to send certain terms direct. This produces the following version of the protocol:

$$
\begin{array}{llll}
1. & M \to C & : & MerchantCcId, MerchantAmount \\
2a. & C \to M & : & Id, MerchantAmount, MerchantCcId, \\
 & & & Transaction, Amount, CardNumber \\
2b. & C \to CB & : & \{kcs\}_{PKCyberKey}, \\
 & & & \{Amount, CardNumber, MerchantCcId, Id, \\
 & & & \quad MerchantAmount, Transaction\}_{SK(C)} \\
3. & M \to CB & : & MerchantCcId, MerchantTransaction, \\
 & & & \{kms\}_{PKMerchantCyberKey}, Id, \\
 & & & MerchantAmount, Transaction, \\
 & & & \{Id, MerchantAmount, MerchantCcId, \\
 & & & \quad MerchantTransaction, Transaction\}_{SK(M)} \\
4a. & CB \to M & : & MerchantCcId, MerchantTransaction, \\
 & & & \{CardNumber, MerchantAmount, Id, \\
 & & & \quad MerchantCcId, Transaction\}_{kms} \\
4b. & CB \to C & : & \{CardNumber, Amount\}_{kcs} \\
5. & M \to C & : & MerchantCcId, MerchantAmount, Transaction, Id\ .
\end{array}
$$

6.3 Analysis of the simplified protocol

We analyzed the simplified version of the Main Sequence Protocol using Casper and FDR [8], testing whether the protocol kept the customer's credit card number and the session keys secret, and testing whether the two agents were authenticated to CyberCash and to one another with agreement on the amount of the transaction.

We found a rather weak authentication attack upon the simplified protocol, where the intruder watches a run of the protocol up until message 4b, intercepting message 4a, and then fakes the message 5 from what he has seen in previous messages. This can be traced back to an attack on the full protocol, where the intruder intercepts message 4 and fakes message 5. The customer thinks he has completed the protocol run successfully, but the merchant thinks the run was broken off after message 3. It is arguable whether this should be classified as an attack; however, it is certainly a behaviour that we should be aware of. This is an attack according to our definition of authentication [7], which is that when one agent completes a run of the protocol, the other agent should at least have progressed as far as the last message with an apparent causal link to a message received by the first agent. When we adapted our authentication specification appropriately, we were able to verify that whenever the customer completes a protocol run, the merchant has progressed at least as far as message 3.

We found another attack, because we oversimplified the protocol. We removed the field $MerchantCcId$, because we didn't realize that this was effectively the merchant's identity; unsurprisingly, this introduced an authentication attack. The simplifications were easily undone to realize that there was no corresponding attack on the original protocol, and hence that we had oversimplified.

Our analysis has revealed some hidden assumptions about the operation of the protocol; without these assumptions, the protocol is insecure.

1. The identity numbers—the fields Id and $MerchantCdId$—are known to all agents, and so when an agent receives another's identity number, he can tell whether it is the correct number. If certificates were used, these could be used to link the identities to the identity numbers. Without this assumption, there are attacks where the intruder replaces one identity number with another. In one attack, the merchant sells to the intruder, and the intruder begins another run impersonating the merchant and selling to the customer; the result is that CyberCash believes the transaction is between the merchant and customer, and so debits the customer's account but the intruder receives the goods. In the other attack, the customer tries to buy from the intruder, and the intruder impersonates the customer in a run with the real merchant; CyberCash believes the transaction is between the real merchant and customer, and the merchant will deliver the goods to the customer; it is hard to see what the intruder gains from this attack, but it is a behaviour we should be aware of.

2. The keys $PKMerchantCyberKey$ and $PKCyberKey$ are not public keys, known by everyone, but are in fact secret, shared, asymmetric keys: CyberCash shares a $PKMerchantCyberKey$ with each merchant, and a

$PKCyberKey$ with each customer. This is important, as these keys are used as part of the authentication mechanism.

Failure to find an attack using a model checker does not (in the absence of arguments like those in [9, 17]) necessarily imply that the protocol is secure: there might be attacks upon systems running the protocol that are larger than those analysed. However, the model checking results certainly improve our confidence in the security of the protocol. Steve Schneider has analysed the simplified versions of the protocol using rank functions [19], and found them to be secure[2]; this implies that the original protocol is also secure.

7 Conclusions

In this paper we have introduced the notion of *fault-preserving simplifying transformations*: transformations that have the property of preserving attacks. We have produced sufficient conditions for a renaming transformation to be fault-preserving, and used this result to show that a number of transformations are indeed fault-preserving. We have also proved that two structural transformations are fault-preserving. We have illustrated these techniques by applying them to a large commercial protocol, simplifying this down to a size that can be modelled and analysed. Attacks that are found can be traced back to the original protocol to see if they are feasible.

We close by giving brief pointers to current and future work. We have applied these techniques to the SET protocol [20]. Again this is a complicated protocol, which is further complicated by having many options. We intend to consider further commercial protocols, for example SSL [21].

Most complications in commercial protocols lie in the large number of fields and nested encryption within each message; we are able to simplify these with our current techniques; it is likely that further case studies will reveal further useful fault-preserving transformations.

In practice, these techniques are straightforward to apply. The hardest part of the analysis of commercial protocols lies in simply understanding the protocol (from what is often poor documentation). We believe that—once we have understood it—it would take us about one day to perform the simplifications and subsequent analysis on such a protocol.

We would like to extend this work to deal with other forms of authentication specification, and also other security properties such as non-repudiation.

The overall effect of these simplifications is to reduce a protocol to a version which can be easily understood. The main aspects of the protocol are retained, to ensure it still fulfills its original function, yet it becomes simple enough to apply standard analysis techniques.

Acknowledgements

We would like to thank Steve Schneider for useful discussions about the CyberCash Main Sequence Protocol. We would also like to thank the anonymous

[2]Personal communication.

referees, whose useful comments have greatly improved the presentation of this paper.

This research was carried out while the authors were members of the Department of Mathematics and Computer Science at the University of Leicester. It was supported by grants from the UK Engineering and Physical Sciences Research Council, the US Office of Naval Research and the UK Defence Evaluation and Research Agency.

References

[1] S. Brackin. Automatic formal analysis of two large commercial 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`.

[2] J. W. Gray, III and J. McLean. Using temporal logic to specify and verify cryptographic protocols (progress report). In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, 1995.

[3] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[4] M. L. Hui and G. Lowe. Safe simplifying transformations for security protocols. Technical Report 1999/1, University of Leicester, 1999. Available via URL `http://www.mcs.le.ac.uk/~glowe/Security/Papers/simplify.ps`.

[5] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2), 1994.

[6] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996. Also in *Software—Concepts and Tools*, 17:93–102, 1996.

[7] G. Lowe. A hierarchy of authentication specifications. In *Proceedings of 10th IEEE Computer Security Foundations Workshop*, 1997.

[8] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998. World Wide Web home page at URL `http://www.mcs.le.ac.uk/~glowe/Security/Casper/index.html`.

[9] G. Lowe. Towards a completeness result for model checking of security protocols (extended abstract). In *11th Computer Security Foundations Workshop*, 1998.

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

[11] W. Marrero, E. Clarke, and S. 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`.

[12] C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[13] J. K. Millen, S. C. Clark, and S. B. Freedman. The interrogator: Protocol security analysis. *IEEE Transactions on software Engineering*, 13(2), 1987.

[14] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *IEEE Symposium on Security and Privacy*, 1997.

[15] L. C. Paulson. Proving properties of security protocols by induction. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, 1997.

[16] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[17] A. W. Roscoe. Proving security protocols with model checkers by data independence techniques. In *11th Computer Security Foundations Workshop*, pages 84–95, 1998.

[18] S. Schneider. Security properties and CSP. In *IEEE Computer Society Symposium on Security and Privacy*, Oakland, 1996.
[19] S. Schneider. Verifying authentication protocols with CSP. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, 1997.
[20] Secure Electronic Transaction (SET) specification. Available from `http://www.setco.org/download.html`, 1997.
[21] The SSL protocol. Available via `http://home.netscape.com/newsref/std/SSL.html`, 1996.
[22] F. J. Thayer Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *1998 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 1998.

A The CyberCash Main Sequence Protocol

We include here a description of the CyberCash Main Sequence Protocol, based upon Steve Brackin's description from [1].

$$\begin{array}{lll}
1. & M \rightarrow C & : Accepts, MerchantAmount, \\
 & & \quad MerchantAmount2Optional, MerchantCcId, \\
 & & \quad MerchantOrderId, MerchantDate, \\
 & & \quad MerchantSwVersion, Note, Payload, PayloadNote, \\
 & & \quad Type, UrlCancel, UrlFail, UrlPayTo, UrlSuccess, \\
 & & \quad MD5(Payload), MerchantSignedHashKey_, \\
 & & \quad MerchantSignedHash_ \\
2. & C \rightarrow M & : CyberKey, Date, Id, MerchantCcId, MerchantDate, \\
 & & \quad MerchantSignedHashKey_, OrderId, ServiceCategory, \\
 & & \quad Transaction, Type, PrHash_, PrSignedHash_, \\
 & & \quad OpaqPrefixCH1_, OpaqueCH1_ \\
3. & M \rightarrow CB & : CyberKey, MerchantCcId, MerchantCyberKey, \\
 & & \quad MerchantDate, MerchantTransaction, ServiceCategory, \\
 & & \quad OpaqPrefixCH1_, OpaqueCH1_, \\
 & & \quad MerchantOpaqPrefixCM1_, MerchantOpaqueCM1_ \\
4. & CB \rightarrow M & : MerchantCcId, MerchantTransaction, MerchantDate, \\
 & & \quad ServiceCategory, MerchantOpaqueCM6_, OpaqueCM6_ \\
5. & M \rightarrow C & : Date, MerchantCcId, MerchantDate, \\
 & & \quad MerchantMessage, MerchantResponseCode, \\
 & & \quad MerchantSwVersion, Id, ServiceCategory, \\
 & & \quad Transaction, Type, PrHash_, \\
 & & \quad PrSignedHash_, OpaqueCM6_ \; .
\end{array}$$

Those fields whose names finish with an underscore are abbreviations, which are defined below:

$$MerchantSignedHashKey\_ \mathrel{\widehat{=}} MD5(PK(M)),$$

$$\begin{array}{l}
MerchantSignedHash_ \mathrel{\widehat{=}} \\
\quad \{MD5(Accepts, MerchantDate, MerchantAmount, MerchantCcId, \\
\qquad\qquad MerchantOrderId, MerchantSignedHashKey_, Note, Type, \\
\qquad\qquad UrlCancel, UrlFail, UrlPayto, UrlSuccess)\}_{SK(M)},
\end{array}$$

$PrHash\_ \mathrel{\widehat{=}}$
$MD5(Accepts, Date, MerchantAmount, MerchantCcId,$
$MerchantOrderId, MerchantSignedHashKey\_, Note, Type,$
$UrlCancel, UrlFail, UrlPayTo, UrlSuccess),$

$PrSignedHash\_ \mathrel{\widehat{=}} MerchantSignedHash\_,$

$OpaqPrefixCH1\_ \mathrel{\widehat{=}} \{kcs\}_{PKCyberKey},$

$OpaqueCH1\_ \mathrel{\widehat{=}}$
$\{Amount, CardCIdOptional, CardCityOptional, CardCountryOptional,$
$CardExpirationDate, CardName, CardNumber,$
$CardOtherFieldsOptional, CardPostalCodeOptional,$
$CardPrefixOptional, CardSalt, CardStateOptional,$
$CardStreetOptional, CardType, SwVersion,$
$MD5(PK(C)), SignatureCH1\_\}_{kcs},$

$SignatureCH1\_ \mathrel{\widehat{=}}$
$\{MD5(Amount, CardCIdOptional, CardCityOptional,$
$CardCountryOptional, CardExpirationDate, CardName,$
$CardNumber, CardOtherFieldsOptional,$
$CardPostalCodeOptional, CardPrefixOptional, CardSalt,$
$CardStateOptional, CardStreetOptional, CardType,$
$CyberKey, Date, Id, MerchantCcId,$
$MerchantSignedHashKey\_, OrderId, PrHash\_,$
$PrSignedHash\_, SwVersion, Transaction, Type)\}_{SK(C)},$

$MerchantOpaqPrefixCM1\_ \mathrel{\widehat{=}} \{kms\}_{PKMerchantCyberKey},$

$MerchantOpaqueCM1\_ \mathrel{\widehat{=}}$
$\{Date, DescriptionListOptional, Id, MerchantAmount, MerchantDba,$
$MerchantLocationOptional, MerchantMessage,$
$MerchantSignedHashKey\_, MerchantSwMessageOptional,$
$MerchantSwServerOptional, MerchantSwVersion,$
$MerchantUrlOptional, OrderId, PrHash\_, PrSignedHash\_,$
$RetrievalReferenceNumberOptional, ServerDateMerchantOptional,$
$TerminalIdFuture, Transaction, TransactionDescriptionOptional,$
$Type, MD5(PKMerchantCyberKey), MerchantSignatureCM1\_\}_{kms},$

$MerchantSignatureCM1\_ \mathrel{\widehat{=}}$
$\{MD5(CyberKey, Date, Id, MerchantAmount, MerchantCcId,$
$MerchantCyberKey, MerchantDate, MerchantTransaction,$
$OrderId, PrHash\_, PrSignedHash\_,$
$ServerDateMerchantOptional, Transaction, Type)\}_{SK(M)},$

$$
\begin{aligned}
& MerchantOpaqueCM6\_ \mathrel{\widehat{=}} \\
& \quad \{AcquirerRefDataOptional, ActionCode, \\
& \quad\; Addn1ResponseDataOptional, AuthorizationCode, \\
& \quad\; AvsInfoOptional, CardCIdOptional, CardCityOptional, \\
& \quad\; CardCountryOptional, CardExpirationDate, CardName, \\
& \quad\; CardNumber, CardPostalCodeOptional, CardPrefixOptional, \\
& \quad\; CardStateOptional, CardStreetOptional, CardType, Date, \\
& \quad\; DebuggingInfoOptional, Id, MerchantMessage, \\
& \quad\; MerchantSignedHashKey\_, MerchantSwMessageOptional, \\
& \quad\; MerchantSwSeverityOptional, OrderId, \\
& \quad\; ProcessorErrorCodeFuture, PrHash\_, PrSignedHash\_, \\
& \quad\; MerchantResponseCode, ResponseDetailCodeFuture, \\
& \quad\; RetrievalReferenceNumberOptional, ServerDate, TerminalIdFuture, \\
& \quad\; Transaction, Type\}_{kms}, \\
& OpaqueCM6\_ \mathrel{\widehat{=}} \\
& \quad \{Amount, AuthorizationCode, CardCIdOptional, \\
& \quad\; CardCityOptional, CardCountryOptional, CardExpirationDate, \\
& \quad\; CardName, CardNumber, CardOtherFieldsOptional, \\
& \quad\; CardPostalCodeOptional, CardPrefixOptional, CardSalt, \\
& \quad\; CardStateOptional, CardStreetOptional, CardTypeOptional, \\
& \quad\; MerchantDba, MerchantLocationOptional, MerchantUrlOptional, \\
& \quad\; Message, OrderId, ResponseCode, ServerDate, \\
& \quad\; SwMessageOptional, SwSeverityOptional, \\
& \quad\; TransactionDescriptionOptional\}_{kcs}.
\end{aligned}
$$

B A brief overview of CSP

In this section we give a brief overview of CSP. More details can be obtained from [3, 16].

An event represents an atomic communication; this might either be between two processes or between a process and the environment. Channels carry sets of events; for example, $send.A.B.\{NA, Nb\}_{Kb}$ is an event of channel *send*. Σ represents the set of all events. The notation $\{\mid a, b \mid\}$ represents the set of all events over channels a and b. We write αP for the alphabet of P (i.e. the set of events P can perform).

In this paper we use processes defined using the following syntax:

$a \rightarrow P$	process that can perform the event a, and then act like P.
$P \mathbin{\Box} Q$	external choice; the process can act like either P or Q; the choice is made by the environment.
$\Box_{i \in I} P_i$	external choice between the processes P_i, where i ranges over I.
$f(P)$	process that acts like P, except every event a is renamed to $f(a)$.
$P \parallel Q$	parallel composition of P and Q, synchronizing on events from the intersection of their alphabets.
$\vert\vert\vert_{i \in I} P_i$	interleaving of the processes P_i, where i ranges over I; i.e. parallel composition with no synchronization.

The traces model of CSP represents a process by the set of traces it can perform, where a trace is a finite sequence of events. We write $traces(P)$ for the set of traces of P. We say that process P is (trace) refined by process Q if the traces of P are a superset of the traces of Q:

$$P \sqsubseteq Q \;\Leftrightarrow\; traces(P) \supseteq traces(Q).$$

We use the following laws concerning refinement in this paper:

1. $f(P \parallel Q) \sqsupseteq f(P) \parallel f(Q)$ provided $f(\alpha P \cap \alpha Q) = f(\alpha P) \cap f(\alpha Q)$.
2. $f(|||_{i \in I} P_i) \sqsupseteq |||_{i \in I} f(P_i)$.