Analysing a Stream Authentication Protocol using Model Checking

Philippa Broadfoot and Gavin Lowe

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
{philippa.broadfoot, gavin.lowe}@comlab.ox.ac.uk

Abstract. In this paper, we consider how one can analyse a stream authentication protocol using model checking techniques. In particular, we will be focusing on the Timed Efficient Stream Loss-tolerant Authentication Protocol, TESLA. This protocol differs from the standard class of authentication protocols previously analysed using model checking techniques in the following interesting way: an unbounded stream of messages is broadcast by a sender, making use of an unbounded stream of keys; the authentication of the n-th message in the stream is achieved on receipt of the $n+1$-th message. We show that, despite the infinite nature of the protocol, it is possible to build a finite model that correctly captures its behaviour.

1 Introduction

In this paper, we consider how one can capture and analyse a stream authentication protocol using model checking techniques. In particular, we will be focusing on the Timed Efficient Stream Loss-tolerant Authentication Protocol, TESLA, developed by Perrig *et al.* [PCTS00]. This protocol is designed to enable authentication of a continuous stream of packets, broadcast over an unreliable medium to a group of receivers; an example application would be to provide authenticated streamed audio or video over the Internet.

This protocol differs from the standard class of authentication protocols previously analysed using model checking techniques in the following interesting way: a continuous stream of messages is broadcast by a sender; the authentication of the n-th message in the stream is achieved on the receipt of the $n+1$-th message. Thus, receivers use information in later packets to authenticate earlier packets. We give a complete description of the protocol below. A particular challenge when modelling and analysing this protocol is that it uses an unbounded stream of cryptographic keys.

In [Arc02], Myla Archer analyses TESLA using the theorem prover TAME. She states:

> Model checking this kind of protocol is not feasible because an infinite state system is required to represent the inductive relationship between an arbitrary n-th packet and the initial packet.

We took this claim as a challenge, and the current paper is the result. In particular, we construct a finite CSP model [Hoa85,Ros97], which can be analysed using the model checker FDR [For00], that correctly captures the unbounded stream of keys. To simulate the unbounded stream of keys in the system, we make use of the data independence techniques developed by Roscoe [Ros98].

The rest of this paper is organised as follows. In Section 2 we describe the TESLA protocol in more detail. In particular, we describe the first two schemes (out of five) presented in [PCTS00]. In Section 3 we present a natural but infinite state model of the basic protocol (Scheme I) within our CSP/FDR framework, so as to introduce many of the basic techniques. We reduce this system to an equivalent finite version in Section 4 by applying the data independence techniques to the generation of keys; we also present a number of techniques for reducing the size of the model's state space, so as to make the analysis more efficient. In Section 5 we describe how we adapt our model to handle the second version of the protocol (Scheme II), where each key is formed as the hash of the following key. We conclude and discuss future work in Section 6. A brief introduction to CSP is included in Appendix A.

The main contributions of this paper are:

- An application of data independence techniques to the analysis of a stream protocol, demonstrating the feasibility of applying model checking techniques to protocols of this class;
- An extension of existing techniques so as to deal with hash-chaining;
- The presentation of a number of state-space reduction techniques.

2 The TESLA protocol

The TESLA protocol is designed to enable authentication of a continuous stream of packets over an unreliable medium. One important factor is efficiency of throughput. Therefore, apart from the initial message, the protocol does not make use of expensive cryptographic primitives such as public key signatures; instead it makes use of message authentication codes (MACs) and commitments using cryptographic hashes.

Perrig *et al.* [PCTS00] introduce 5 schemes for the TESLA protocol, each addressing additional requirements to the previous one. We will consider only the first two of these schemes in this paper.

Scheme 1 Informally, TESLA works as follows. An initial authentication is achieved using a public key signature; subsequent messages are authenticated using MACs, linked back to the initial signature.

In message $n-1$, the sender S generates a key k_n, and transmits $f(k_n)$, where f is a suitable cryptographic hash function, to the receivers, as a commitment to that key; in message n, S sends a data packet m_n, authenticated using a MAC with key k_n; the key itself is revealed in message $n+1$. Each receiver checks that the received k_n corresponds to the commitment received in message $n-1$, verifies the MAC in message n, and then accepts the data packet m_n as authentic.

Message n also contains a commitment to the next key k_{n+1}, authenticated by the MAC, thus allowing a chain of authentications.

More formally, the initial messages are as follows:[1]

$$\begin{array}{lll} \text{Msg } 0a. & R \to S & : n_R \\ \text{Msg } 0b. & S \to R & : \{f(k_1), n_R\}_{SK(S)} \\ \text{Msg } 1. & S \to R & : D_1,\ MAC(k_1,\ D_1) \text{ where } D_1 = \langle m_1,\ f(k_2)\rangle \\ \text{Msg } 2. & S \to R & : D_2,\ MAC(k_2,\ D_2) \text{ where } D_2 = \langle m_2,\ f(k_3),\ k_1\rangle. \end{array}$$

n_R is a nonce generated by the receiver to ensure freshness. The signature on $f(k_1)$ in message 0b acts as an authenticated commitment to k_1. When R receives k_1 in message 2, he can be sure that it really was sent by S. This allows R to verify the MAC in message 1, and so be assured of the authenticity of the data m_1. Further, R is assured of the authenticity of $f(k_2)$, the commitment to the next key.

For $n > 1$, the n-th message is:[2]

$$\text{Msg } n. \quad S \to R \ :\ D_n,\ MAC(k_n,\ D_n) \text{ where } D_n = \langle m_n,\ f(k_{n+1}),\ k_{n-1}\rangle.$$

An authenticated commitment to k_{n-1} will have been received in message $n-2$; the receipt of this key assures R of the authenticity of the data received in message $n-1$, and also the commitment to k_n in that message. Later, when k_n is received in message $n+1$, R will be assured of the authenticity of the data m_n and the commitment $f(k_{n+1})$ in message n.

The protocol requires an important time synchronisation assumption, the *security condition*: the receiver will not accept message n if it arrives after the sender might have sent message $n+1$ (otherwise an intruder can capture message $n+1$, and use the key k_n from within it to fake a message n). Thus the agents' clocks need to be loosely synchronised; this is achieved within an initial exchange.

Scheme II Scheme II differs from Scheme I by not generating a stream of fresh keys, but instead generating a single key k_m, and calculating each key k_n, for $n < m$, by hashing k_m $m-n$ times: $k_n = f^{m-n}(k_m)$. Hence each key acts as a commitment to the next: when the receiver obtains k_n he can verify that $f(k_n) = k_{n-1}$. Thus the explicit commitment can be dropped, and the protocol simplified to:

$$\begin{array}{lll} \text{Msg } 0a. & R \to S & : n_R \\ \text{Msg } 0b. & S \to R & : \{k_0, n_R\}_{SK(S)} \\ \text{Msg } 1. & S \to R & : m_1,\ MAC(k_1,\ m_1). \end{array}$$

[1] The message numbering scheme is a bit clumsy, but has the advantage that data packet m_n is received in message n and authenticated using key k_n.

[2] In the original version [PCTS00], the n-th MAC is authenticated using $f'(k_n)$, instead of k_n, where f' is a second hash function; we can omit the use of f' for modelling purposes.

And for $n > 1$:

$$\text{Msg } n. \quad S \rightarrow R \; : \; D_n,\, MAC(k_n,\, D_n) \text{ where } D_n = \langle m_n,\, k_{n-1} \rangle.$$

One aim of this second version is to be able to tolerate an arbitrary number of packet losses, and to drop unauthenticated packets, yet continue to authenticate later packets.

3 Modelling the basic protocol

In this section, we present a natural, but infinite state model of the basic protocol (Scheme I), explaining some of the techniques we use for modelling and analysing security protocols within the CSP/FDR framework. In the next section we reduce this system to an equivalent finite version, by applying data independence techniques.

The basic idea is to build CSP models of the sender and receiver, following the protocol definition. We also model the most general intruder who can interact with the protocol, overhearing, intercepting and faking messages; however, as is standard, we assume strong cryptography, so we do not allow the intruder to encrypt or decrypt messages unless he has the approriate key. These processes synchronise upon appropriate events, capturing the assumption that the intruder has control over the network, and so can decide what the honest agents receive. We then define a specification process that captures the security requirements to be analysed; FDR can be used to discover if the specification is satisfied. See [Low96,**?**,**?**] for more details and examples of the technique.

We will actually consider a slightly simplified version of the protocol: we omit the old key k_{n-1} from the MAC, since it seems to be redundant:

$$\text{Msg } n. \quad S \rightarrow R \; : \; m_n,\, f(k_{n+1}),\, k_{n-1},\, MAC(k_n,\, \langle m_n,\, f(k_{n+1}) \rangle).$$

This simplification is *fault-preserving* in the sense of Hui and Lowe [HL01]: if there is an attack upon the original protocol, there is also an attack upon this simplified protocol; hence if we can verify the simplified protocol, we will have verified the original protocol.

3.1 Sender and receiver processes

The sender and receiver nodes are modelled as standard CSP processes that can perform *send* and *receive* events according to the protocol description. We assume that the intruder has complete control over the communications medium; hence all communication goes through the intruder, as in Figure 1.

As discussed in Section 2, the TESLA protocol relies upon a time synchronisation between the sender and receiver processes (known as the *Security Property* in [PCTS00]). We capture this requirement in our models by introducing the special event *tock* that represents the passage of one time unit. The processes

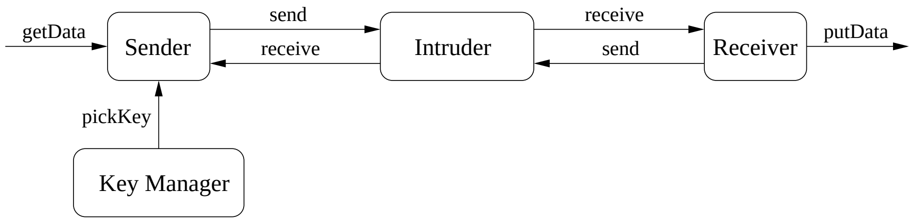

Fig. 1. Overview of the network.

representing the sender S and receiver R synchronise upon *tock*, modelling the fact that the two agents' clocks are loosely synchronised. This allows R to tell whether it has received message n before S might have sent message $n+1$.

We begin by presenting a process to represent the sender. It would be natural to parameterise this process by the infinite sequence of keys that are used to authenticate messages. However, in order to ease the transition to the next model, we instead arrange for the keys to be provided by an external process, *KeyManager*, and for the sender to obtain keys on a (private) channel *pickKey* (see Figure 1).

The sender is initially willing to receive an authentication request (message 0a) containing a nonce n_R; it responds by obtaining a key (k_1) from the key manager, and sending back the initial authentication message (message 0b); it then waits until the next time unit before becoming ready to send the next message; if no initial authentication request is received, it simply stays in the same state:

$$
\begin{aligned}
& \mathit{Sender}_0(S) = \\
& \quad \Box\, R : \mathit{Agent},\ n_R : \mathit{Nonce} \bullet \\
& \qquad \mathit{receive} \,.\, R \,.\, S \,.\, (\mathit{Msg}_{0a},\ \langle n_R \rangle) \rightarrow \mathit{pickKey}?k_1 \rightarrow \\
& \qquad \mathit{send} \,.\, S \,.\, R \,.\, (\mathit{Msg}_{0b},\ \langle \{f(k_1),\ n_R\}_{SK(S)} \rangle) \rightarrow \mathit{tock} \rightarrow \mathit{Sender}_1(S,\ R,\ k_1) \\
& \quad \Box \\
& \quad \mathit{tock} \rightarrow \mathit{Sender}_0(S).
\end{aligned}
$$

The sender obtains the next key (k_2), gets the first piece of data to be sent (m_{curr}) from some channel *getData*, and then sends the data and the commitment to the next key, including a MAC using the current key (message 1); it then waits until the next time unit before becoming ready to send the next message:

$$
\begin{aligned}
& \mathit{Sender}_1(S,\ R,\ k_1) = \\
& \quad \mathit{pickKey}?k_2 \rightarrow \mathit{getData} \,.\, S?m_{curr} \rightarrow \\
& \quad \mathit{send} \,.\, S \,.\, R \,.\, (\mathit{Msg}_1, \langle m_{curr},\ f(k_2),\ \mathit{MAC}(k_1,\ \langle m_{curr},\ f(k_2) \rangle) \rangle) \rightarrow \\
& \quad \mathit{tock} \rightarrow \mathit{Sender}_n(S,\ R,\ k_1,\ k_2).
\end{aligned}
$$

Subsequent behaviour is very similar to the previous step, except the previous key (k_{prev}) is included in each message. This is the last time the sender uses this key, so he can now forget it, modelled by the event *forget* . k_{prev}; we include this forgetting as an explicit event to ease the transition to the next model:

$$\begin{array}{l} Sender_n(S,\ R,\ k_{prev},\ k_{curr}) = \\ \quad pickKey?k_{next} \rightarrow getData\ .\ S?m_{curr} \rightarrow \\ \quad send\ .\ S\ .\ R\ .\ (Msg_n,\ \langle m_{curr},\ f(k_{next}),\ k_{prev},\ MAC(k_{curr},\ \langle m_{curr},\ f(k_{next})\rangle)\rangle) \rightarrow \\ \quad forget\ .\ S\ .\ k_{prev} \rightarrow tock \rightarrow Sender_n(S,\ R,\ k_{curr}, k_{next}). \end{array}$$

It is straightforward to model the process *KeyManager* that issues keys; it is parameterised by the sequence of keys that are issued:

$$KeyManager(xs) = pickKey\ .\ head(xs) \rightarrow KeyManager(tail(xs)).$$

It is precisely this mechanism of generating fresh keys that needs to be adapted in Section 4 in order to reduce the model to an equivalent finite version.

Before defining the receiver process, it is convenient to define the sets *AllCommits* and *AllMacs* of key commitments and MACs that the receiver might receive. The receiver is unable to tell immediately whether it has received a valid commitment or MAC, so should also be willing to receive an arbitrary bit-string; we model such bit-strings using a special value *Garbage*:

$$\begin{array}{rcl} AllCommits & = & \{f(k) \mid k \in Key\} \cup \{Garbage\}, \\ AllMacs & = & \{MAC(k,\ \langle m,\ f\rangle) \mid k \in Key,\ m \in Packet, f \in AllCommits\} \\ & & \cup \{Garbage\}. \end{array}$$

The receiver begins by sending a nonce to the sender as an authentication request (message 0a). It then becomes willing to receive an appropriate initial authentication message (message 0b); it becomes ready to receive the next message in the next time unit. If the initial authentication message is not received before the end of the current time unit, the receiver should abort.

$$\begin{array}{l} Receiver_0(R,\ n_R) = \\ \quad \sqcap\, S : Agent \bullet send\ .\ R\ .\ S\ .\ (Msg_{0a},\ n_R) \rightarrow Receiver_0'(R,\ S,\ n_R), \\ Receiver_0'(R,\ S,\ n_R) = \\ \quad \Box\, f_{next} : AllCommits \bullet \\ \quad\quad receive\ .\ S\ .\ R\ .\ (Msg_{0b},\ \{f_{next},\ n_R\}_{SK(S)}) \rightarrow tock \rightarrow Receiver_1(R,\ S,\ f_{next}) \\ \quad \Box \\ \quad tock \rightarrow Abort(R). \end{array}$$

The receiver is then willing to receive an appropriate message 1; note that it should be willing to accept an arbitrary key commitment and MAC, because it is not yet able to verify either. If the message is not received before the end of the time unit, the receiver should abort.

$$\begin{array}{l} Receiver_1(R,\ S,\ f_{curr}) = \\ \quad \Box\, m_{curr} : Packet,\ f_{next} : AllCommits,\ mac_{curr} : AllMacs \bullet \\ \quad\quad receive\ .\ S\ .\ R\ .\ (Msg_1, \langle m_{curr},\ f_{next},\ mac_{curr}\rangle) \rightarrow \\ \quad\quad tock \rightarrow Receiver_n(R,\ S,\ f_{curr},\ f_{next},\ m_{curr},\ mac_{curr}) \\ \quad \Box \\ \quad tock \rightarrow Abort(R). \end{array}$$

The receiver has now reached the phase where it can receive a message each time unit. For each message received, the authenticity of the previous message is verified by checking that (i) $f(k_{prev})$ is equal to the key commitment in the previous message, f_{prev}; and (ii) the MAC is authentic, i.e. $MAC(k_{prev}, \langle m_{prev}, f_{curr}\rangle)$ is equal to the MAC sent in the previous message, mac_{prev}. If these conditions are satisfied, then the receiver outputs the data on a channel *putData*; he then forgets the old key and moves to the next time unit. If either of these checks fails, or no message arrives, then the protocol run is aborted.

$$
\begin{array}{l}
Receiver_n(R,\ S,\ f_{prev}, f_{curr}, m_{prev}, mac_{prev}) = \\
\quad \Box\ m_{curr} : Packet, f_{next} : AllCommits, k_{prev} : Key,\ mac_{curr} : AllMacs \bullet \\
\qquad receive\,.\,S\,.\,R\,.\,(Msg_n, \langle m_{curr},\ f_{next},\ k_{prev},\ mac_{curr}\rangle) \rightarrow \\
\qquad if\ f(k_{prev}) = f_{prev} \wedge MAC(k_{prev}, \langle m_{prev}, f_{curr}\rangle) = mac_{prev}\ \ then \\
\qquad\quad putData\,.\,R\,.\,S\,.\,m_{prev} \rightarrow forget\,.\,R\,.\,k_{prev} \rightarrow \\
\qquad\quad tock \rightarrow Receiver_n(R,\ S,\ f_{curr},\ f_{next},\ m_{curr},\ mac_{curr}) \\
\qquad else\ Abort(R) \\
\quad \Box \\
\quad tock \rightarrow Abort(R).
\end{array}
$$

If the receiver aborts a protocol run, represented by the process *Abort*, it simply allows time units to pass.

$$Abort(R) = tock \rightarrow Abort(R).$$

3.2 Intruder process

The intruder model follows the standard Dolev-Yao [DY83] model. He is able to act as other agents, which might or might not behave in a trustworthy way; overhear all network messages; prevent messages from reaching their intended recipients; and finally, fake messages to any agent, purporting to be from any other.

The formal model is built around a set *Deductions* representing the ways in which the intruder can deduce new messages from messages he already knows, for example by encrypting and decrypting with known keys: each element (m, S) of *Deductions* represents that from the set of messages S, he can deduce the message m. The intruder process is parameterised by the set IK of messages that he currently knows. He can intercept messages (on the channel *send*) and add them to his knowledge; send messages that he knows to honest agents (on the channel *receive*); and deduce new messages from messages he already knows.

$$
\begin{array}{l}
Intruder(IK) = \\
\quad send?A\,.\,B\,.\,m \rightarrow Intruder(IK \cup \{m\}) \\
\quad \sqcap \\
\quad \bigsqcap\ m : IK,\ A, B : Agent \bullet receive\,.\,A\,.\,B\,.\,m \rightarrow Intruder(IK) \\
\quad \sqcap \\
\quad \bigsqcap (m, S) : Deductions,\ S \subseteq IK \bullet infer\,.\,(m, S) \rightarrow Intruder(IK \cup \{m\}).
\end{array}
$$

For reasons of efficiency, the actual intruder implementation used is built as a highly parallel composition of many two-state processes, one process for each fact that the intruder could learn. This model was developed by Roscoe and Goldsmith [RG97] and is equivalent to the process above.

The intruder's knowledge is initialised to a set *IIK* containing values that the intruder could be expected to know, including all agents' identities, all public keys, all data values, his own secret key, a nonce and key different from those used by the honest agents, and the value *Garbage* that represents an arbitrary bit-string.

3.3 Overall system

The above processes are combined together in parallel, as illustrated in Figure 1, and all events except for the *getData*, *putData* and *tock* events are hidden (made internal):

$$\begin{aligned}
Agents &= Sender_0(S) \underset{\{\mid pickKey \mid\}}{\parallel} KeyManager(Ks)) \mid\mid\mid Receiver_0(R, N_1) \\
System_0 &= Agents \underset{\{\mid send, receive \mid\}}{\parallel} Intruder(IIK) \\
System &= System_0 \setminus \{\mid send, receive, pickKey, forget \mid\}
\end{aligned}$$

3.4 Specification processes

We now verify that the system meets a suitable specification. According to Perrig *et al.* [PCTS00], TESLA guarantees that the receiver does not accept as authentic any message m_n unless m_n was sent by the sender; in fact, m_n must have been sent in the previous time interval.

We capture this by the following CSP specification. The specification captures our security requirement by defining the order and timing by which messages can be sent by the sender and get accepted by the receiver as authentic. We can use FDR to verify that all traces (i.e. sequences of external events) of the system are traces allowed by the specification.

In the initial state, the specification allows the sender S to input a packet m_{curr}, before evolving after one time unit into the state $Spec'(S, R, m_{curr})$. Alternatively, time is allowed to pass without changing the state of the specification.

$$\begin{aligned}
&Spec(S, R) = \\
&\quad getData\,.\,S?m_{curr} \rightarrow tock \rightarrow Spec'(S, R, m_{curr}) \\
&\quad \sqcap \\
&\quad tock \rightarrow Spec(S, R).
\end{aligned}$$

Subsequently, the receiver can, but might not, output on *putData* the value m_{prev} received in the previous time unit. The sender can input a new value m_{curr}. These events can happen in either order, giving the following specification.

$$
\begin{array}{l}
Spec'(S, R, m_{prev}) = \\
\quad getData \,.\, S?m_{curr} \rightarrow tock \rightarrow Spec'(S, R, m_{curr}) \\
\quad \sqcap \\
\quad getData \,.\, S?m_{curr} \rightarrow putData \,.\, R \,.\, S \,.\, m_{prev} \rightarrow tock \rightarrow Spec'(S, R, m_{curr}) \\
\quad \sqcap \\
\quad putData \,.\, R \,.\, S \,.\, m_{prev} \rightarrow getData \,.\, S?m_{curr} \rightarrow tock \rightarrow Spec'(S, R, m_{curr}).
\end{array}
$$

FDR can be used to verify that the system meets this specification when the key manager is initialised with a finite sequence of keys. The following section describes how to adapt the model to deal with an infinite sequence of keys.

4 A finite model of an infinite system

In this section we show how to transform the model of TESLA from the previous section to an equivalent finite model, which can be analysed using FDR. We make use of the data independence techniques developed by Roscoe [Ros98].

The data independence techniques allow us to simulate a system where agents can call upon an unbounded supply of fresh keys even though the actual type remains finite. In turn this enables us to construct models of protocols where agents can perform unbounded sequential runs and verify security properties for them within a finite check.

The basic idea is as follows: once a key is no longer held by any honest participant (i.e., each agent who held the key has performed a corresponding *forget* event), we *recycle* the key: that is, we allow the same key to be subsequently re-issued to the sender; hence, because there is a finite bound on the number of keys held by honest participants, we can make do with a finite number of keys.

However, at the point where we recycle a key k, the intruder's memory will include messages using that key, because the intruder overhears all messages that pass on the network; in order to prevent FDR from discovering bogus attacks, based upon reusing the same key, we need to transform the intruder's memory appropriately. The transformation replaces k with a special key K_b, known as the *background key*; more precisely, every message containing k in the intruder's memory is replaced by a corresponding message containing K_b. The effect of this is that anything the intruder could have done using k before the recycling, he can now do using K_b. Thus we collapse all old keys down to the single background key.

Note that this key recycling is a feature of the model rather than of the protocol. We argue below that it is safe in the sense that it does not lose attacks. Further, it turns out not to introduce any false attacks.

The technique is implemented within the CSP protocol model by adapting the key manager process (which issues keys to the sender), so that it keeps track of which honest agents currently hold which keys. It synchronises on the messages where agents acquire new keys (*pickKey* events in the case of the sender, and receipt of all messages in the case of the receiver) and release keys (*forget* events); when a key has been released by all agents, the manager triggers the recycling mechanism, remapping that key within the intruder's memory, as described above. (We need to adapt the model of the receiver slightly so that it releases all keys when it aborts.) That key can then be re-issued.

We write $System'$ for the resulting system, and $System'_0$ for the system without the top level of hiding (analagous to $System_0$). Since $System'$ is a finite model, we can check that it refines $Spec$ automatically using the model checker FDR.

In [RB99], Roscoe proves that this transformation is sound; we outline the argument here. The critical property that justifies this transformation is

$$System' \sqsubseteq System. \tag{1}$$

If we then use FDR to verify that $Spec \sqsubseteq System'$, we can deduce that $Spec \sqsubseteq System$, by the transitivity of refinement.

Let ϕ be the function over traces that replaces each key from the infinite model wtih the corresponding key in the finite model, in particular replacing keys that have been forgotten by all honest agents with the background key K_b. Equation (1) holds because of the following relationship between $System_0$ and $System'_0$, noting that all the keys are hidden at the top level:

$$traces(System'_0) \supseteq \{\phi(tr) \mid tr \in traces(System_0)\} \tag{2}$$

In turn, property (2) holds because the corresponding property holds for each individual agent in the system[3]:

$$\begin{aligned} traces(Agents') &= \{\phi(tr) \mid tr \in traces(Agents)\} \\ traces(Intruder'(IIK)) &\supseteq \{\phi(tr) \mid tr \in traces(Intruder(IIK))\} \end{aligned}$$

The first property holds by construction of the new key manager, and because the sender and receiver processes are data independent in the type of keys. The latter holds because of the following property of the deduction system:

$$(m, S) \in Deductions \Rightarrow (\phi(m), \phi(S)) \in Deductions$$

which says that for any deduction the intruder can make in the original system, he can make the corresponding deduction in the reduced system.

Modelling the protocol directly as above is fine in theory, but in practice leads to an infeasibly large state space. Below we discuss a number of implementation strategies that help overcome this problem. For ease of presentation, we discuss the optimisations independently, although in practice we combined them.

Number of data packets One interesting question concerns the number of distinct data packets that we need to include in our model. It turns out that it is enough to include just two distinct packets M_0 and M_1: if the protocol fails in a setting

[3] We write "$Agents'$" and "$Intruder'$" for the new models of the honest agents and intruder, respectively.

where more than two packets are used, then it will also fail in the model with just two packets. To see why this is the case informally, suppose there were an error in the former case represented by trace tr; this error must be because the receiver had received a particular packet $\widehat{M}$ incorrectly; consider the effect of replacing in tr all occurrences of $\widehat{M}$ by M_1, and all occurrences of other packets by M_0; it is easy to see that this would be a trace of the model with just two packets, and would be an error because the receiver would accept M_1 incorrectly. This argument can be formalised, making use of a general theorem by Lazić [Laz99] [Ros97, Theorem 15.2.2].

Splitting protocol messages Large protocol messages comprising many fields are expensive to analyse using FDR, because they lead to a large message space and a large degree of branching in the corresponding processes. Splitting such messages into several consecutive ones is a simple, yet effective reduction technique that we frequently use to handle this situation. In the case of TESLA, we split the main message:

$$\text{Msg } n. \quad S \rightarrow R \;:\; m_n,\, f(k_{n+1}),\, k_{n-1},\, MAC(k_n,\, \langle m_n,\, f(k_{n+1})\rangle).$$

into the following two consecutive smaller messages:

$$\begin{array}{ll} \text{Msg } n_a. & S \rightarrow R \;:\; m_n,\, f(k_{n+1}),\, k_{n-1} \\ \text{Msg } n_b. & S \rightarrow R \;:\; MAC(k_n,\, \langle m_n,\, f(k_{n+1})\rangle). \end{array}$$

This strategy speeds up checking by an order of a magnitude. Hui and Lowe prove in [HL01, Theorem 11] that this transformation is sound in the sense that a subsequent verification of the transformed protocol implies the correctness of the original protocol.

Associating incorrect MACs The model of the receiver allows him to receive an arbitrary MAC in each message:

$$\begin{array}{l} Receiver_n(R,\, S,\, f_{prev}, f_{curr}, m_{prev}, mac_{prev}) = \\ \quad \square \ldots mac_{curr} : AllMacs \bullet \\ \quad\quad receive\,.\,S\,.\,R\,.\,(Msg_n, \langle m_{curr},\, f_{next},\, k_{prev},\, mac_{curr}\rangle) \rightarrow \ldots, \end{array}$$

where *AllMacs* contains all valid MACs and also the special value *Garbage* that models a bit-string not representing a valid MAC. However, if the MAC does not correspond to m_{curr} and f_{next} then it will be rejected at the next step. Therefore the above model allows the receiver to receive many different incorrect MACs, all of which are treated in the same way. We can reduce the state space by an order of magnitude by collapsing all of these incorrect MACs to a single value, namely *Garbage*, noting that there is no essential difference between a behaviour using this value and a behaviour using a different incorrect MAC. Thus we rewrite the receiver to:

$$\begin{array}{l} Receiver_n(R,\, S,\, f_{prev}, f_{curr}, m_{prev}, mac_{prev}) = \\ \quad \square \ldots mac_{curr} : \{MAC(k, \langle m_{curr}, f_{next}\rangle) \mid k \in Key\} \cup \{Garbage\} \bullet \\ \quad\quad receive\,.\,S\,.\,R\,.\,(Msg_n, \langle m_{curr},\, f_{next},\, k_{prev},\, mac_{curr}\rangle) \rightarrow \ldots. \end{array}$$

This transformation can be justified in a similar way to the transformation that reduced the key stream to a finite stream. Equation (2) holds when we use the reduction function ϕ that maps all incorrect MACs onto *Garbage*. Note that in particular (using obvious names for the old and new versions of the processes):

$$\mathit{traces}(\mathit{Receiver}') \;=\; \{\phi(tr) \mid tr \in \mathit{traces}(\mathit{Receiver})\},$$

by the above argument; and

$$\mathit{traces}(\mathit{Intruder}'(\mathit{IIK})) \;\supseteq\; \{\phi(tr) \mid tr \in \mathit{traces}(\mathit{Intruder}(\mathit{IIK}))\},$$

because the intruder can always send the value *Garbage* in place of an incorrect MAC.

Placement of tests One can also obtain a large speed up by careful placement of the test that verifies the previous MAC. In the earlier model, the receiver was willing to receive any key in message n, and then checked the previous MAC. It is more efficient to simply refuse to input any key that does not allow the MAC to be verified, as follows:

$$\begin{array}{l}
\mathit{Receiver}_n(R,\, S,\, f_{prev}, f_{curr}, m_{prev}, mac_{prev}) = \\
\quad \Box\, k_{prev} : \mathit{Key},\, f(k_{prev}) = f_{prev} \wedge MAC(k_{prev}, \langle m_{prev}, f_{curr}\rangle) = mac_{prev} \bullet \\
\quad \Box\, m_{curr} : \mathit{Packet},\, f_{next} : \mathit{AllCommits},\, mac_{curr} : \mathit{AllMacs} \bullet \\
\qquad \mathit{receive} \,.\, S \,.\, R \,.\, (\mathit{Msg}_n, \langle m_{curr},\, f_{next},\, k_{prev},\, mac_{curr}\rangle) \to \ldots .
\end{array}$$

This reduction can be justified in a similar way to previous reductions.

Placement of forget actions Recall that the sender and receiver perform *forget* events to indicate that they have finished using a key. It turns out that placing this *forget* event as early as possible—i.e. immediately after the last event using that key—leads to a large reduction in the size of the state space.

Combining events In our previous model, we included different events within the sender for obtaining the key from the key manager and for sending the message using that key. It is more efficient to combine these events into a single event, arranging for the sender to be able to use an arbitrary key, and arranging for the key manager to allow an arbitrary send event using the key it wants to supply next, and synchronising these two processes appropriately. It is also possible to combine the *getData* event with the *send* event, meaning that the sender's environment chooses the value sent.

$$\begin{array}{l}
\mathit{Sender}_1(S,\, R,\, k_{curr}) = \\
\quad \Box\, k_{next} : \mathit{Key},\, m_{curr} : \mathit{Packet} \bullet \\
\quad \mathit{send} \,.\, S \,.\, R \,.\, (\mathit{Msg}_1, \langle m_{curr},\, f(k_{next}),\, MAC(k_{curr},\, \langle m_{curr},\, f(k_{next})\rangle)\rangle) \to \ldots .
\end{array}$$

Once the whole system has been combined, a renaming can be applied for compatibility with the specification. It is also possible to combine the *forget* events with appropriate messages, although we have not done this.

Combining these state space reduction techniques with the key recycling mechanism, we have been able to construct a reduced model of TESLA that simulates the key chain mechanism. The agent processes are able to perform an unbounded number of runs; the use of the recycling mechanism gives the appearance of an infinite supply of fresh keys within a finite state model.

5 Modelling key chaining and re-authentication

In this section we show how to adapt the protocol so as to deal with Scheme II, where explicit key commitments are omitted, but instead each key is formed as the hash of the following key. Most of the adaptation is straightforward, except for the key chaining, which we describe below. We also extend the model to allow new initial authentications following packet loss or failed checks by the receiver.

5.1 Modelling key chaining

Recall that the n-th key k_n is calculated by hashing some fixed key k_m $m - n$ times. The receiver should accept a key k_{curr} only after checking that it hashes to the previously accepted key k_{prev}. Clearly we cannot model this hash-chaining through explicit modelling of the hashes, for there is no fixed bound on the number of hashes, and so the state space would be infinite.

Our model of the hash-chaining instead makes use of the following observation: k_{curr} hashes to k_{prev} precisely when k_{prev} and k_{curr} are keys that were issued consecutively by the key manager. We therefore introduce a *key checker* process that keeps track of the order in which keys were issued, and allows the event $check \, . \, k_1 \, . \, k_2$ if k_1 and k_2 were consecutively issued; we then model the checking of hashes by having the receiver process attempt the event $check \, . \, k_{prev} \, . \, k_{curr}$; synchronising the two processes upon *check* events ensures that only appropriate checks succeed.

In more detail:

- The key checker allows an event $check \, . \, k_1 \, . \, k_2$ for fresh keys k_1 and k_2 precisely when those keys were consecutively issued.

Hence the receiver will accept a foreground key precisely when it was issued immediately after the previously accepted key, which corresponds to the case that it hashes to the previous key. The key checker should allow additional *check* events in the situation where the intruder has managed to get the receiver to accept a key that has already been recycled:

- The key checker allows the event $check \, . \, K_b \, . \, k$ for fresh or background keys k.

The key checker should at least allow $check \, . \, K_b \, . \, k$ if k is the key issued immediately after a foreground key that could correspond to this K_b (because a corresponding *check* event would have been possible if we were not using the recycling mechanism). We allow more such *check* events, which is sound, because it only gives the intruder more scope to launch attacks; this turns out not to introduce any false attacks, essentially because if the intruder had managed to get the receiver to accept K_b, then he would have already broken the protocol anyway.

One might have expected the key checker to also allow events of the form $check \,.\, k_1 \,.\, K_b$ for fresh keys k_1 when the key issued after k_1 has been recycled. However, it turns out that allowing such checks introduces false attacks into the model: if the key following k_1 has been recycled, the intruder can fake a MAC using the background key, even though there is no corresponding behaviour in the infinite state model. We avoid this problem by enforcing that keys are recycled in the same order in which they are issued, so as to prevent this circumstance from arising.

We can then model the receiver process as follows:

$$
\begin{array}{l}
Receiver_n(R,\ S,\ k_{prev}, m_{prev}, mac_{prev}) = \\
\quad \square\ k_{curr} : Key,\ MAC(k_{curr}, m_{prev}) = mac_{prev} \bullet \\
\qquad check.k_{prev}.k_{curr} \rightarrow forget.R.k_{prev} \rightarrow \\
\qquad \square\ m_{curr} : Packet,\ mac_{curr} : AllMacs \bullet \\
\qquad\quad receive \,.\, S \,.\, R \,.\, (Msg_n, \langle m_{curr},\ k_{curr},\ mac_{curr}\rangle) \rightarrow \ldots .
\end{array}
$$

Note that the *check* and *forget* events are performed as early as possible, to reduce the size of the state space. Further state space reductions techniques from Section 4 can be applied.

The reduction, from a model that uses explicit hashes to form the keys to the finite model described above, can be justified as before. Equation (2) holds for the function ϕ over traces that replaces keys and inserts *check* events appropriately. In particular

$$
traces(Receiver' \underset{\{\!|\, check \,|\!\}}{\parallel} KeyChecker \setminus \{\!|\ getKey\ |\!\}) \ \supseteq\ \{\phi(tr) \mid tr \in traces(Receiver)\}.
$$

5.2 Modelling new initial authentications

We can also adapt the model of the protocol to allow the receiver to attempt new initial re-authentications. We can adapt the *Abort* state of the receiver to allow him to re-start after a nondeterministic amount of time:

$$
Abort(R) = tock \rightarrow Abort(R) \sqcap RECEIVER_0(R).
$$

However, the receiver should use a different nonce each time it attempts a re-authentication, and so we need to model an infinite supply of nonces. Doing so is simple: we simply recycle nonces in the same way that we recycled keys, using a nonce manager process that issues fresh nonces to the receiver, and recycles nonces that are no longer stored by any agent.

We can then adapt the model of the sender to accept such re-authentication attempts, by adding the following branch to the $Sender_n$ process:

$$
\begin{array}{l}
Sender_n(S, R, k_{prev}) = \\
\quad \textit{“as before”} \\
\quad \Box \\
\quad \Box\, R : Agent,\ n_R : Nonce \bullet \\
\qquad receive\,.\,R\,.\,S\,.\,(Msg_{0a},\ \langle n_R \rangle) \rightarrow pickKey?k_{next} \rightarrow \\
\qquad send\,.\,S\,.\,R\,.\,(Msg_{0b},\ \langle \{k_{prev},\ n_R\}_{SK(S)} \rangle) \rightarrow Sender_n(S,\ R,\ k_{prev}).
\end{array}
$$

6 Conclusions

In this paper, we described how to model and analyse the Timed Efficient Stream Loss-tolerant Authentication Protocol presented in [PCTS00], using model checking techniques within the CSP/FDR framework. This was initially motivated by a challenging statement by Archer [Arc02] claiming that this class of protocols is infeasible to analyse using model checking techniques.

We started by presenting a CSP model for Scheme I of the protocol using an unbounded supply of fresh cryptographic keys. We then showed how one can apply the data independence techniques presented in [Ros98,**?**] to reduce this system to an equivalent finite version. This was achieved by implementing the recycling mechanism upon a finite set of fresh keys, creating the necessary illusion of having an unbounded stream of them. A number of reduction strategies were developed and incorporated into the model to keep the state space within a feasible range.

This protocol model was then extended to capture Scheme II. In particular, we had to develop a technique for modelling unbounded hash-chains. We also had to adapt the model, to allow recycling of nonces, so as to allow an unbounded number of repeat authentications.

Our analysis showed that the protocol is correct, at least when restricted to a system comprising a single sender and a single receiver. It is interesting to ask whether we can extend this result to larger systems. We can argue informally as follows.

- It is easy to extend this result to multiple receivers: the intruder would not learn anything from interactions with additional receivers except more nonces, and these would not help him, because he could only use them in the same way that he could use his own nonces; further, any failure of the protocol when executed by multiple receivers would correspond to a failure for one of those receivers, which we would have discovered in the above analysis.
- It is also easy to extend the result to multiple senders with different identities: if the intruder were able to replay messages from a server S_2 and have them accepted as coming from another server S_1, then in the model of this paper he would have been able to use his own messages in the same way as the

messages of S_2 so as to have them accepted as coming from S_1, which again we would have discovered in the above analysis.
- Finally, it is not possible to extend the result to the case where a single server runs the protocol multiple times simultaneously, issuing different data streams, but using the same key for the initial authentications; indeed in this circumstance it is possible for the intruder to cause confusion between these data streams.

Our model of Scheme II allows the receiver to recover from the loss or corruption of packets only by performing a repeat authentication. TESLA itself allows the receiver to recover from d successive lost or corrupted packets, by checking that the next key received, when hashed $d + 1$ times, gives the previously accepted key. It would be interesting to extend our model to consider this case, at least for a limited range of values of d. However, we believe that it is not possible to consider this extension for arbitrary values of d, because the system seems inherently infinite state. Further, even for small values of d, one would run into problems with a state space explosion; however, a parallel version of FDR, designed for execution on a network of computers, is now available, which could be employed on this problem.

In [PCTS00], Perrig *et al.* present three further versions of the TESLA protocol.

- Scheme III looks at how one can achieve faster transfer rates. One of the restrictions in the previous schemes is that the receiver must receive packet M_n before the corresponding key is disclosed in packet M_{n+1}. Under Scheme III, the key for packet M_n is disclosed in some later packet M_{n+d}, where d is some suitably chosen parameter.
- Scheme IV extends the protocol to allow for dynamic packet rates.
- Scheme V extends the protocol further to accommodate a broader spectrum of receivers. This is achieved by introducing multiple authentication chains with different disclosure rates, thereby allowing the receivers to choose those that are most efficient to their setup.

In principle, one could extend our models to consider these extensions, although to a limited extent: for example one could consider a limited range of values for d in Scheme III, and a limited range of receivers in scheme V. However, similar limitations apply to those discussed above concerning recovery from packet loss.

We would also like to investigate an alternative method for recycling the keys: use a fixed set of keys, $K_1, \ldots, K_n$ for a suitable value of n (in the setting of Section 5, we need $n = 5$) and arrange that at each stage the new key issued is K_1; in order for this to work, we would need to arrange a re-mapping of keys on each *tock* event, both within the states of the honest agents and the intruder: we replace K_1 with K_2, replace K_2 with K_3, ..., and replace K_n with the background key K_b. Thus each key K_i represents the key introduced $i - 1$ time units ago. The effect of this is that each message sent by the sender would introduce K_1, have the MAC authenticated using K_2, and disclose K_3. This increased regularity should cause a decrease in the size of the state space.

TESLA is based upon the Guy Fawkes Protocol [ABC$^+$98]; it would seem straightforward to adapt the techniques of this paper to that protocol. A further protocol of interest is the second one presented in [PCTS00], the Efficient Multi-chained Stream Signature Protocol (EMSS). TESLA does not provide non-repudiation; EMSS is designed to fulfill this requirement. Non-repudiation is a property that is easily captured within the CSP/FDR framework; see [Hui01].

Acknowledgements

This work was partially funded by the UK Engineering and Physical Sciences Research Committee as part of the e-science program.

References

[ABC$^+$98] Ross Anderson, Francesco Bergadano, Bruno Crispo, Jong-Hyeon Lee, Charalampos Manifavas, and Roger Needham. A new family of authentication protocols. *Operating Systems Review*, 32(4):9–20, 1998.

[Arc02] Myla Archer. Proving correctness of the basic TESLA multicast stream authentication protocol with TAME. In *Workshop on Issues in the Theory of Security*, 2002.

[DY83] D. Dolev and A. C. Yao. On the security of public-key protocols. *Communications of the ACM*, 29(8):198–208, August 1983.

[For00] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement—FDR2 User Manual*, 2000. Available via `http://www.fsel.com/fdr2_manual.html`.

[HL01] Mei Lin Hui and Gavin Lowe. Fault-preserving simplifying transformations for security protocols. *Journal of Computer Science*, 9(1, 2):3–46, 2001.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hui01] Mei Lin Hui. *A CSP Approach to the Analysis of Security Protocols*. PhD thesis, University of Leicester, 2001.

[Laz99] Ranko Lazić. *Theorems for mechanical verification of data-independent CSP*. D.Phil, Oxford University, 1999.

[Low96] Gavin 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.

[PCTS00] Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Xiaodong Song. Efficient authentication and signing of multicast streams over lossy channels. In *IEEE Symposium on Security and Privacy*, pages 56–73, May 2000.

[RB99] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(2, 3):147–190, 1999.

[RG97] A. W. Roscoe and M. H. Goldsmith. The perfect 'spy' for model-checking crypto-protocols. In *Proceedings of DIMACS workshop on the design and formal verification of cryptographic protocols*, 1997.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[Ros98] A. W. Roscoe. Proving security protocols with model checkers by data independence techniques. In *11th IEEE Computer Security Foundations Workshop*, pages 84–95, 1998.

A CSP notation

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, $c.5$ is an event of channel c. The event *tock* represents the passage of one unit of time.

The process $a \to P$ can perform the event a, and then act like P. The process $c?x \to P_x$ inputs a value x from channel c and then acts like P_x.

The process $P \;\Box\; Q$ represents an external choice between P and Q; the initial events of both processes are offered to the environment; when an event is performed, that resolves the choice. $P \sqcap Q$ represents an internal or nondeterministic choice between P and Q; the process can act like either P or Q, with the choice being made according to some criteria that we do not model.

The external and internal choice operators can also be distributed over a set of processes. $\Box\, i : I \bullet P_i$ and $\bigsqcap i : I \bullet P_i$ represent replicated external and nondeterministic choices, indexed over set I. The range of indexing can be restricted using a predicate; for example, $\Box\, i : I, p(i) \bullet P_i$ restricts the indexing to those values i such that $p(i)$ is true.