Timed Analysis of Security Protocols

R. Corin∗, S. Etalle, P.H. Hartel and A. Mader
University of Twente, The Netherlands

Abstract

We propose a method for engineering security protocols that are aware of timing aspects. We study a simplified version of the well-known Needham Schroeder protocol and the complete Yahalom protocol, where timing information allows the study of different attack scenarios. We model check the protocols using UPPAAL. Further, a taxonomy is obtained by studying and categorising protocols from the well known Clark Jacob library and the Security Protocol Open Repository (SPORE) library. Finally, we present some new challenges and threats that arise when considering time in the analysis, by providing a novel protocol that uses time challenges and exposing a timing attack over an implementation of an existing security protocol.

Keywords: timed automata, security protocols, model checking.

1 Introduction

The communication via a shared medium, like the Internet, is inherently insecure: anyone has access to en route messages and can potentially eavesdrop or even manipulate the ongoing communication. Security protocols are distributed programs specifically designed to achieve secure communication over such media, typically exchanging messages built constructed using cryptographic operations (e.g. message encryption).

Security protocols are difficult to design correctly, hence their analysis is critical. A particularly successful model to analyze security protocols is the Dolev Yao model [16], in which the attacker is assumed to have complete control over the network. Also, the model assumes ideal cryptography, where cryptographic operations are assumed to be perfect. The Dolev Yao model is attractive because it can be easily formalized using languages and tools based on formal methods. Moreover, the model has an appropriate level of abstraction, as many attacks are independent of the underlying details of the cryptographic operations and are based only on combinations of message exchanges plus knowledge gathered by the attacker during the execution.

Typically, Dolev Yao methods for formal verification of security protocols (among the proposals [27, 14, 10]) do not take time into account, and this choice simplifies the analysis. However, security protocols, like distributed programs in general, are sensitive to the passage of time. Recently, consideration of time in the analysis of security protocols has received some attention (see Related Work below), but this attention has been focused mostly on timestamps.

In this paper1 we develop an analysis model for security protocols that explicitly takes into account time flowing during the execution of a protocol. In general, in the design and implementation of a security protocol two aspects of timing must be considered at some stage:

1. Time can influence the flow of messages. For instance, when a message does not arrive in a timely fashion (i.e. timeouts), retransmissions or other actions have to be considered.

2. Time information can be included within protocol messages (e.g. timestamps).

∗Contact author’s email is ricardo.corin@utwente.nl.

1An earlier version appeared in [11].
Consider first (1) above. In general, the influence of time on the flow of messages is not usually considered by current state of the art methods for analysing protocols. However, we believe it to be crucial because (i) Even if the abstract protocol does not decide what action to take at a particular moment of the execution (e.g. in the case of timeouts), the actual implementation will eventually have to consider these issues anyway; (ii) The efficiency and security of the implementation depends critically on these specific decisions; and (iii) The timing of message flows in a protocol can be exploited by an attacker.

Now consider item (2) above. There, we believe that making judicious use of timing information in a protocol has received attention but mostly in the limited setting of using time stamps as opposed to nonces. However, time information can be used to influence message flows as well, as we illustrate in Section 6.

Contributions Our study covers several issues in the study of time in security protocols.

• Firstly, in Section 2 we study which kinds of timing issues, like timeouts and retransmissions, may arise in the study of security protocols. We then proceed in Section 3 to present a method for the design and analysis of security protocols that consider these timing issues. The method is based on modelling security protocols using timed automata 3. In support of the method we use UPPAAL 4 as a tool to simulate, debug and verify security protocols against classical safety goals like secrecy and authentication, in a real time scenario, using reachability properties. As examples, we analyse a simplified version of the Needham Schroeder protocol 26 and the full Yahalom protocol 9 in Section 4.

• Secondly, in Section 5 we categorize all the protocols from the Clark and Jacob library and the SPORE library into different (more abstract) patterns of message flows with timeouts. We then analyse each abstract pattern, independently of the actual protocols, and establish their timing efficiency and security.

• Finally, in Section 6 we illustrate some novel opportunities and difficulties that appear when considering time in the design and analysis of security protocols:
  – In Section 6.1 we give an example protocol that accomplishes authentication by exploiting the timeliness of messages. The protocol uses time in a conceptually new way, by employing time challenges as a replacement for nonces.
  – As a second example of a novel difficulty in Section 6.2 we describe how timing attacks 17 can be applied to security protocols, by describing an attack over Abadi’s private authentication protocol 2. Although these protocols can be modelled as timed automata, thus permitting general verification, we leave the detailed verification as future work since for this we need a model checker that is also probabilistic (like 13 or 24): our nondeterministic intruder of UPPAAL is too powerful, since it can always guess correctly times and values even if the probability of guessing is negligible.

2 Timeouts and Retransmissions

To illustrate how time influences the analysis of security protocols (even when it does not explicitly use timing information), consider the following protocol written in the usual notation.

\[ 1. A \rightarrow B : M_{AB} \]
\[ 2. B \rightarrow A : M_{BA} \]

Here, first A sends message \( M_{AB} \) to B, and later B sends message \( M_{BA} \) to A. This high-level view does not consider timing. To consider time, we first need to assume that both A and B have
Figure 1: Left: Timeouts (i) typical (ii) windowed; Right: timeout actions (iii) chained abort and (iv) retransmission.

timers. In this paper, we do not require timers between parties to be synchronised (see below for a discussion). The next step consists in distinguishing the different operations that occur, with their respective times. In Step 1, it takes some time to create $M_{AB}$. The other operation that takes time is the actual sending of the message, i.e. the time it takes $M_{AB}$ to travel from $A$ to $B$. This transmission time is unbounded, since the message may be lost or intercepted, and therefore $A$ may need to timeout: After $A$ sends $M_{AB}$, she starts a timer that will timeout if $M_{BA}$ (Step 2 of the above protocol) is not received after some waiting, say $t_A$ (Figure 1 (i)). Clearly, $t_A$ should be greater than the time of creating $M_{BA}$, plus the average time of sending both $M_{AB}$ and $M_{BA}$. In general, $A$ does not need to start waiting for a response immediately after sending a message; for instance, $A$ could hibernate (or start doing another task) for some time $s_A$ before beginning to expect the response $M_{BA}$. This results in a windowed timeout (Figure 1 (ii)). Typically, the values for $s_A$ and $t_A$ depend on implementation details. However, an implementation independent quantitative analysis could already give an early indication of what attacks can be mounted for some values that are no longer possible for others (e.g. a smaller $t_A$ and a larger $s_A$).

Another issue that is not considered either in previous approaches is that the action to be taken when a timeout occurs is sensitive. Typically, the implicit assumption is that the protocol should abort, as it is the case in Figure 1 (i). This means that the protocol party that reaches the timeout deduces that a fault has happened. However, aborting may not consist only of stopping execution altogether. For example, if we consider protocols with several parties, we may wish that when a party timeouts it also communicates its decision to abort to other, still active parties. For instance, consider the following protocol:

1. $A \rightarrow B : M_{AB}$  
   $A$ starts timer expecting $M_{BA}$

2. $A \rightarrow S : M_{AS}$  
   $S$ starts session timer

3. $B \rightarrow A : M_{BA}$

Here, if $A$ times out on Step 2, she could communicate the abort decision to $S$, as shown in in Figure 1 (iii).

Aborting execution is not the only feasible action to perform after a timeout, and in principle protocols could successfully execute when messages do not arrive at certain moments. Even if we do assume that a fault occurred, aborting may not be the best choice: sometimes, message retransmission is a better, more efficient and also more realistic option, as depicted in Figure 1 (iv). In this case, a question which arises is whether to retransmit the original message ($M_{AB}$ for Figure 1 (iv)), or to recompute some parts before resending the message. Here, the tradeoff is, as usual, between efficiency versus security.

Time information can also be included in the contents of $M_{AB}$ and $M_{BA}$. A typical value to include is a timestamp, to prevent replay attacks. However, this requires secure clock synchronisation of $A$ and $B$, which is expensive (see Mills [29] for a security protocol to achieve this). In fact, this is the reason for which Bellovin et al. recommend to switch to nonces in the Kerberos protocol [7]. Recently, the analysis of security protocols using timestamps has received consider-
able attention from the research community (see Related Work in Section 7). Therefore, in this paper we do not pursue this direction.

3 A Method for Analysing Security Protocols

We use timed automata [3] to model protocol participants, and this has several advantages. Firstly, our method requires the designer to provide a precise and relatively detailed protocol specification, which helps to disambiguate the protocol behaviour. Secondly, timing values like timeouts need to be set at each state, while retransmissions can be specified as transitions to other protocol states.

Once modelled as timed automata, the protocol can be fed to the real time model checker UPPAAL, which allows the protocol to be simulated and verified. The simulation provides the designer with a good insight of the inner workings of protocol, and already at this stage specific timing values like timeouts can be tuned. Then the designer can proceed with verification of specific properties. As usual in model checking, the verification of the protocol with UPPAAL is automatic.

The resulting timed automata model is an informative and precise description of the security protocol, and thus, it provides a practical way to strengthen implementations while keeping efficiency in mind.

As a third and final step we propose to transfer timing information back to the high level protocol description. This serves to highlight the role of time in the implementation, but also (as we will demonstrate in Section 6.1), to make timing an integral aspect of the protocol design.

3.1 Timed Automata and UPPAAL

In this paper, the timed automata of Alur and Dill are used for modelling [3]. In general, timed automata models have an infinite state space. The region automaton construction, however, shows that this infinite state space can be mapped to an automaton with a finite number of equivalence classes (regions or zones) as states [3]. Finite-state model checking techniques can then be applied to the reduced, finite region automaton. A number of model checkers for timed automata is available, for instance, Kronos [31] and UPPAAL [4].

Parallel composition of automata is one of the main sources for expressiveness. This operation allows to decompose complex behaviour, thus supporting transparent modelling. When composing automata in parallel, we need also to provide some form of communication. For the timed automata we use in this paper, communication comes in form of hand-shake synchronisation. Two parallel automata can share a synchronisation channel, i.e. both have a transition labelled with a complementing channel name, e.g. synchronise! in the example of Figure 2. These transitions may only be taken together, at the same moment. In Figure 2, we see an example for a transition, labelled by a guard that has to be true when the transition is taken, a synchronisation channel, and a variable update.

Data transmission is typically modelled by a synchronisation, where global variables are updated. These global variables contain the data that are transmitted.

\[
\begin{align*}
\text{var}_v &> \text{const}_c \\
\text{synchronise!} \\
\text{var}_w &:= \text{var}_w + 1
\end{align*}
\]

Figure 2: Example transition with guard, synchronisation and update
Timed automata extend “classical” automata by the use of real-valued clock variables. All clock variables increase with the same speed. For timed automata we make a difference between a state and a location: a state is a location where all clocks have a fixed value. In this sense a location symbolically represents an infinite set of states, for all the different clock valuations. In Figure 3 an elementary fragment of timed automata is shown. When the transition from location I to location II takes place, the clock clock is reset to 0. Location II may only be left at time $D$, where $D$ is a constant. The invariant $\text{clock} \leq D$ at location II enforces that the transition to III has to be taken at time $D$.

Typically, the initial location of an automaton is denoted by a double circle. We also make use of committed locations, which have the property that they have to be left immediately. In most cases committed locations are used to model a sequence of actions with atomic execution.

The properties verified by the model checker of UPPAAL are reachability properties, like “there is a state where property $p$ holds reachable from the initial state”, or the dual “in all reachable states property $p$ holds”. The latter is falsified if the model checker finds a state that does not satisfy $p$. In this case a diagnostic trace from the initial state to the state that does not satisfy $p$ is produced by the model checker; it serves as counterexample.

We use this mechanism to find attacks. If we can characterize for example the fact that some secret is not secret any more as a propositional property, and the model checker finds a state where this property holds, the diagnostic trace describes a sequence of actions that leads to this state, which gives precisely the attack.

Note that in this context verification comes very much in the guise of debugging. Finding an attack requires an adequate problem model. Not finding an attack increases the confidence in the modelled protocol, but does not exclude that attacks could be found in other models for the same protocol.

### 3.2 Overview of the UPPAAL Model

Let us now describe the general form of our model, in some detail. We model the protocol participants (initiator, responder, etc) and the intruder as timed automata. Additionally, we model cryptography as another automaton, the cryptographic device, which acts as an impartial party that regulates the access to data. In Figure 4 we illustrate a setting consisting of one initiator and one responder. Here, boxes in bold represent our general intruder and the cryptographic device,
while dashed boxes represent the actual initiator and responder. These participants use the cryptographic device to perform operations, but communicate through the intruder (thus the intruder is identified with the network itself, obtaining a Dolev Yao like intruder [10]). Our modelling is modular, and allows us to “plug in” different participants (e.g., in the analysis of the Yahalom we add a server), while the bold boxes, i.e. the intruder and the cryptographic device, are the core model.

While modelling security protocols as timed automata in UPPAAL, we will focus on modelling the times required by the principals to encrypt and decrypt values (and generate nonces), but not on the actual time that takes the sending (transmission times are assumed to be unknown). Therefore, for our results to be useful, we assume that computing times (e.g. cryptographic operations) are not negligible w.r.t. communication times, and thus choices for timeout values depend both on communication and computing times.

3.3 Modelling Cryptography

The automaton for a cryptographic device is presented in Figure 5. This cryptographic device performs nonce generation and public key cryptography. Later we also use a device for symmetric cryptography, which can be obtained from the one in Figure 5 in a straightforward manner. In fact, our method allows different cryptographic devices to be plugged in as needed (e.g. to add hashing). Basically, the device model is a shared table containing pairs of plaintexts and keys. The first service of the cryptographic device is to provide fresh nonces to the protocol participants (and also the intruder). The process of nonce generation is started via synchronization on the gen_nonce channel. To model the new nonce creation, the local variable gennonce is incremented with the constant seed plus the value of param1 that includes the ID of the requesting participant (this ensures that initial generated nonces differ from each other). The number of possible nonces is bounded by ensuring that gennonce is always smaller than a fixed constant MAX. After synchronization, a global result variable is updated with a generated nonce, and the device finishes by synchronizing on the finish_nonce channel.

Encryption and decryption are modelled by two local arrays to the cryptographic device, namely plain and key. When a party wants to encrypt some value d with key k, it synchronises with the device via the channel start_encrypt. If the device has still room in its tables, it stores d in the plain array and k in the key array. As a result, it sets in the global variable result the index in which d and k reside in the arrays. This index is the “ciphertext”. Upon decryption, the ciphertext is provided to the cryptographic device, which then checks that the provided key is correct: Since we model public-key encryption, the private key of a public key k is simply modelled as a function f.s.t. f(k) > MAX, so that private keys do not clash with generated nonces and hence are never known by the attacker simply by guessing. In this simple case we simply let f(k) = 10k, which since only one nonce is needed by the participants in the example protocol of Section 4.1, gives enough room for the attacker to generate nonces.

State constructions Now that we have the cryptographic device, an honest principal can use different state constructions to perform cryptographic operations. In Figure 6 we show the different kinds of state constructions used in our models, which designers should use as building blocks for the representation of protocol participants.

In the upper left of Figure 6 we see the building block for nonce generation. Here, a protocol participant first resets the clock t, assigns its identity to variable param (used by cryptographic device to provide different nonces to different participants) and then fires via the gen_nonce channel. Then the participant enters a state in which it waits until the time of nonce generation happens (time_gennonce), synchronises via the finish_nonce channel and obtains the return value via variable result. Encryption and decryption are analogous, and only differ in that they use two parameters param1 and param2 (for plaintext and key in the former, and ciphertext and key in the latter).
Figure 5: Timed automaton for a Cryptographic Device

Figure 6: State constructions: nonce generation (above), encryption (middle) and decryption (bottom)
3.4 Modelling the Adversary

The intruder, presented in Figure 7, works basically as a Dolev Yao intruder [16]. The intruder models the network itself, by acting as an intermediary of communication between the initiator and responder. This is modelled by letting the intruder synchronise on both channels \textit{init\_msg} and \textit{resp\_msg}. Upon synchronising by receiving a message, the intruder moves to state (SINT1), where it saves the message \textit{msg} in its local variable \textit{data} and resets an index variable \textit{i} which bounds the total number of actions allowed to do before continuing execution. Then, the intruder moves to state (SINT3), where it makes a nondeterministic choice for an action. More precisely, it can decide to:

- Choose an identity in its local variable \textit{pk} (State SINT4)
- Encrypt a value (State SINT5)
- Decrypt a value (State SINT6)
- Generate a nonce (State SINT7)
- Save variable \textit{data} as message \textit{msg}.

The intruder can then continue to perform these actions, choose to send a message or simply block a message and continue the execution. Moreover, the intruder can also delay arbitrarily a message, by waiting in state (SINT2).

Note that the intruder is independent of the actual protocol under study, and hence it is generic to analyze protocols using public key encryption (although this intruder is not able to concatenate messages; we extend it in the next section).

4 Analysing Protocols

We first consider a simple protocol to illustrate our technique. Later, we move on to analyse the more complex Yahalom protocol.

4.1 An Example Protocol

In this section we study and model in UPPAAL a simplified version of the Needham Schroeder protocol, thoroughly studied in the literature (see eg. [26]). Differently from the Needham Schroeder protocol whose goal is to achieve mutual authentication, our simpler protocol aims at authenticat-
ing the initiator $A$ to a responder $B$ only (we do not lose generality here, this is just a simplification to improve presentation). The protocol is as follows:

1. $A \rightarrow B : A$
2. $B \rightarrow A : \{N_B\}_{K_A}$
3. $A \rightarrow B : \{N_B\}_{K_B}$

In the first message, the initiator $A$ sends a message containing its identity to the responder $B$. When $B$ receives this message, it generates a nonce $N_B$, encrypts it with the public key $K_A$ of $A$ and sends it back to $A$. Upon receipt, $A$ decrypts this message with her private key, obtains the nonce $N_B$, reencrypts it with the public key $K_B$ of $B$ and sends it back to $B$.

We can now move on to describe the actual initiator, responder and intruder. Both the initiator and responder have local constants time\_out, which represent their timeout values. Also, the initiator, responder and intruder have local constants time\_gennonce, time\_encrypt and time\_decrypt that represent the time required to generate a nonce, encrypt a value or decrypt a value, respectively for each principal.

The automata for the initiator and responder of our simple protocol presented above are given in Figure 8 (the dashed transitions of the responder correspond to retransmissions, discussed in Section 4.1.2). The initiator $A$ starts her execution when activated via channel start (State SI0). The actual identity of the initiator role is set via the global variable init\_id (this and other role variables are chosen by the Init automaton, described below). The initiator saves init\_id as the first message (see protocol message 1). Then, the initiator starts her protocol execution, by firing via the channel init\_msg. After this, the initiator starts a clock $t$ and waits for a response, or until $t$ reaches time\_out (State SI2). If the timeout occurs, the protocol is aborted (a retransmission at this point would be equivalent to restart the protocol). If a response is received before the timeout via the init\_msg channel, $A$ tries to decrypt the received message $msg$. This takes time time\_decrypt for the initiator. After the decryption, the initiator reencrypts the obtained nonce (stored in result) and finally sends the last message via the init\_msg channel, setting to true its local boolean variable finish.

The responder automaton $B$ works similarly to the initiator. After receiving the start signal, $B$ waits for the message containing the claimed identity of $A$ (State SR1). When received, $B$ saves the first message in the local variable claimed\_id. After this, $B$ generates a nonce by contacting the cryptographic device. When ready (State SR3), $B$ encrypts the nonce with the value received in Message 1 (we identify identities with public keys). After finishing the encryption (State SR4), the message is sent and $B$ starts to wait for a response (State SR5). If an answer comes before the timeout, $B$ decrypts the message and checks that the challenge is indeed the one $B$ sent. If so, the local boolean variable finish is set to true.

### 4.1.1 Verification

We wish to verify that our simple protocol indeed accomplishes authentication of $A$ to $B$. To this end, we will model check one session of the protocol containing one initiator, one responder and one intruder. We use a special Init automaton that instantiates the initiator and responder with identities (like $A$, $B$ and $I$), and then starts the execution run by broadcasting via the start channel. The init automaton is given in Figure 9.

The property we check, AUT, is shown in Table 1. AUT states that if we reach a state in which the responder has finished executing but the claimed id (corresponding to the first message of protocol) does not coincide with the actual identity of the initiator, the protocol is flawed. Indeed, a state in which the initiator can “lie” and still force the responder to finish means that authentication is violated. This is one of the possible forms of authentication failure. It is outside the scope of this paper to illustrate different authentication flaws (see Lowe [27] and Cremers et al. [12] for more on authentication notions).
Figure 8: Schemas of timed automata for the Initiator (top) and the Responder (bottom)

Figure 9: Timed automaton for the Init automaton

\[
AUT = E <> \text{Responder}.\text{finish} \text{ and } \text{Responder}.\text{claimed_id} = \text{resp_party}
\]
\[
AUT_y = E <> \text{Initiator}.\text{finish} \text{ and } \\
\text{Initiator}.\text{ticks} < (\text{Responder}.\text{time_encrypt} + \text{Responder}.\text{time_gennonce} + \text{Server}.\text{time_encrypt} \times 2 + \text{Server}.\text{time_decrypt}) - 1
\]

Table 1: UPPAAL properties
If we use a long timeout for $B$, i.e.

$$B.time_{\text{out}} \geq Intruder.time_{\text{decrypt}} + Intruder.time_{\text{encrypt}} + A.time_{\text{encrypt}} + A.time_{\text{decrypt}}$$

here UPPAAL finds a man-in-the-middle attack, presented on the left hand side of Table 2. This attack is similar to Lowe’s attack [26], in which an attacker fools $B$ into thinking he is communicating with $A$, while in reality $A$ only talks to $I$. Of course, we could patch the protocol as Lowe did. But, in the context of time, it is interesting to model-check the protocol with a tighter timeout, i.e. $B.time_{\text{out}} < Intruder.time_{\text{decrypt}} + Intruder.time_{\text{encrypt}} + A.time_{\text{encrypt}} + A.time_{\text{decrypt}}$. When this constraint is verified, the man-in-the-middle attack vanishes. Of course, we cannot pretend that $B$ knows the intruder’s times of encryption and decryption. Nevertheless, $B$ can set $B.time_{\text{out}} = A.time_{\text{encrypt}} + A.time_{\text{decrypt}}$, leaving no space for any interruption.

A second attack which is independent of timeouts (even if we set $B.time_{\text{out}} = 0$!) was also found by UPPAAL; this time, the vulnerability is much simpler. We report it on the right hand side of Table 2. This attack corresponds to a “reflection” replay attack [30]. This attack occurs when the intruder simply replies $B$’s message. The attacker fools $B$ into thinking its communicating with himself, while it is not true in reality. Interestingly, suppose we change message 3 of the protocol to $3’$. $A \rightarrow B : \{N_B + 1\}K_B$. Now, the above replay attack is prevented, since message 2 is not valid as message 3 anymore. Of course, a patch à la Lowe for both also prevents both problems:

1. $A \rightarrow B : A$
2. $B \rightarrow A : \{B, N_B\}K_A$
3. $A \rightarrow B : \{N_B\}K_B$

Having find confirmation that our framework is capable of finding untimed attacks (and thus confirming known attacks), we proceed to provide a good baseline to study extended security protocols with timing issues, like timeouts and retransmissions.

### 4.1.2 Retransmissions

Consider again the automaton for the responder, given in Figure 8. In state SR4, the responder sends the challenge $\{N_B\}K_A$, and waits for a response in state SR5. If the response does not arrive before the timeout, the responder simply aborts. Now we consider possible retransmissions that allow the protocol to recover and continue its execution. With timed automata, retransmissions are easy to model by adding transition arrows from state SR5 to previous states of the automaton (the dashed lines in Figure 8); These transitions are guarded, allowing to perform the action only when the timeout is reached (i.e., $t >= time_{\text{out}}$). A further refinement not explored here would be to add counters so that the number of retransmissions can be limited before aborting.

We consider two potential target states for the timeout of the Responder in SR5, namely states SR3 and SR2. Choosing the former corresponds to retransmitting the exact same message that was sent before, $\{N_B\}K_A$. On the other hand, linking the retransmission arrow to SR2 corresponds to recomputing the whole message, by creating a new nonce $N_B’$ and sending $\{N_B’\}K_A$.
We implemented both strategies in our UPPAAL model. As can be expected, retransmitting the exact message once has the effect of duplicating the timeout for $B$, and thus the man-in-the-middle attack becomes possible even for tight timeout values. On the other hand, recomputing the whole message preserves the security of the protocol, at a higher computational cost. This evidences that indeed these design decisions are important for both security and efficiency, and a careful analysis can help to choose the best timeouts and retransmissions for a practical implementation.

4.2 A Real Protocol

Having illustrated our approach with a simple example we now study a more realistic protocol, the Yahalom protocol [9]. This protocol aims at authentication of $A$ and $B$ as well as key distribution of $A$ and $B$ using a shared server $S$ with whom both $A$ and $B$ share secret keys $K_{AS}$ and $K_{BS}$.

Our choice is based on the fact that Yahalom is a complex and strong protocol, with no known attacks over it (However, a modification proposed by Abadi et al. [8] has a known type-flaw attack). Our aim is to study the protocol in more detail (and thus closer to an implementation) with timing information. The protocol is as follows:

1. $A \rightarrow B$: $A, N_A$
2. $B \rightarrow S$: $B, \{N_B, A, N_A\}K_{BS}$
3. $S \rightarrow A$: $\{B, K_{AB}, N_A, N_B\}K_{AS}, \{A, K_{AB}, N_B\}K_{BS}$
4. $A \rightarrow B$: $\{A, K_{AB}, N_B\}K_{BS}, \{N_B\}K_{AB}$

Here we use symmetric encryption, and key $K_{XY}$ is shared between $X$ and $Y$.

To model concatenation in an efficient way, we gathered several message components into a 16 bit field, thus keeping the state space as small as possible. In our case, we assume that nonces have 4 bits, principal id’s 2 bits and keys 4 bits. To access these values, we use bit-wise and with appropriate masks, and (left,right) bit shifts. Our intruder has also the capability to do the shifts and mask, and we also removed the “public key” choice from the intruder of Figure 7. We have modelled the protocol in UPPAAL (the initiator, responder, server and intruder are shown in Figures 10 and 11).

As we did with the previous protocol, first we check whether authentication of $A$ to $B$ could be falsified, using property $AUT$ from Table 1. This property is not satisfied, confirming that Yahalom is secure. Now we move to study time sensitive issues.

There are two places in which timeouts and retransmissions can occur in this protocol. The first one is in Message 1: After $A$ sends her message, she starts a timer waiting for message 3. Now, suppose that a timeout occurs, and $A$ wants to retransmit her message. We can be confident that resending the same nonce $N_A$ will not affect security, since in any case it was already sent in the clear in the first time. However, an interesting timing issue arises here. An answer that is received too early by $A$ could be suspicious, because some time must pass while $B$ talks to $S$. If $A$ knows $B$’s and $S$’s encryption and decryption times, $A$ could even deliberately “hibernate” (eg. to save energy) until the response is likely to arrive (this models a windowed timeout, see Figure 11(ii)). We model checked this property by measuring the time after $A$ sends her message, and a response arrive (we count ticks, the dashed loop transition of the initiator in Figure 11). The specified property is $AUT_y$, shown in Table 1. This property is not satisfied, confirming that there is no way that the initiator can receive a valid answer before the time required by the responder and server to process $A$’s request. In an implementation, it is reasonable for $A$ to set a timeout like above, since it is realistic to assume that $A$ can know the responder and server’s times of encryption and decryption.

The second timeout is set by $B$ after sending his message at step 2. If a timeout occurs, the retransmission decision is more delicate: It is not clear whether $B$ should resend the original message, should recomput $N_B$ or whether $B$ should abort, since clearly $N_A$ cannot be recomputed. Intuitively, $N_B$ could be reused. We modelled in UPPAAL the retransmission of the exact message.
(as the dashed transition of the responder in Figure 10). When we model check again property $AUT_y$, we obtain that it is still unreachable, confirming that in that case an efficient retransmission of the same message 2 by $B$ is secure.

However, by observing the messages flow, we know that if $B$ timeouts then it is very likely that $A$ has also reached its timeout and aborted (see Figure 1(iv)). This mainly happens because since $A$ is unsure whether $B$ is alive or not, and thus $A$’s timeout needs to be tight. If $A$ knew that $B$ is alive and waiting for an answer from $S$, then $A$ could extend its timeout. We then sketch a more efficient implementation in Figure 12 in which at Message 2 $B$ also sends a special $subm$ message notifying $A$ of the submission to $S$. Then $A$ can extend its timeout with more confidence (the second dashed line in Figure 12). In the case $subm$ is never received by $A$, she can send an $abort$ message to $B$. Of course, in this simple model the attacker can also send this messages, thus performing denial of service attacks; in any case, our attacker is powerful enough to stop communication altogether.

In summary, for the Yahalom protocol we obtain that retransmitting for the responder is secure, and also that the initiator can be implemented to efficiently “hibernate” a safe amount of time before receiving a response.

5 Taxonomy of Message Flows in Security Protocols

The flow of messages of many protocols follow a small set of specific patterns. By exploring the well known Clark Jacob library [9] of authentication protocols and the Security Protocols Open Repository (SPORE) [1], we were able to categorize the protocols in four categories, as shown in Figure 13. To each pattern, we add the corresponding timeouts, and analyze their impacts on security and efficiency. For the original references of the protocols, the interested reader may consult the Clark Jacob library [9] and the SPORE library [1].
Figure 11: Timed automata schemas for the Server (top) and Intruder (bottom)

Figure 12: An more detailed implementation of Yahalom
Not shown in this categorization are non-interactive protocols which do not wait for messages and thus do not require timeouts. In this category fall the Wide Mouthed Frog protocol, the CCITT X.509 simple pass protocol and the CAM protocol for mobile IP.

First, we discuss the simplest pattern in Figure 13 (i). This is a three-message exchange with two participants. This pattern is the simplest and also the most secure one from timing point of view, since timeouts can be set tight, due to the ping-pong nature of the exchanges. To this pattern correspond both the example protocol of Section 4.1 and the one in Section 6.1, and also the protocols CCITT.509 three pass, the Shamir Rivest and Adleman Three-Pass protocol, the ISO XXX Key Three-Pass (and their repeated protocols), the SmartRight view-only protocol (from SPORE) and the Diffie-Hellman key exchange protocol. With a fourth message from B to A in the same fashion we find the Andrew Secure RPC protocol. Adding a third participant S, but still doing ping-pong exchanges, we can add the Needham Schroeder symmetric key protocol and the Denning Sacco protocol.

Secondly, we identify three-party protocols, in which a server S also takes part in the communication (Figure 13 (ii)) but ping-pong exchanges are not anymore used. This pattern is potentially unsafe and inefficient for A, since she has to wait until a long timeout as elapsed after the first message before receiving an answer from S. This is due to the fact that three messages have to be exchanged after A’s initial message. By consulting again the Clark and Jacob library and the SPORE repository, we see that the Otway Rees protocol, the Gong mutual authentication protocol, the Woo-Lam mutual authentication protocol, the Carlsen protocol, and finally the Kehne Schoenwalder Langendorfer (KSL) protocol all fall in this category. Adding ping-pong exchanges before and after the exchanges of Figure 13 (ii) we find the Needham Schroeder Public key protocol. Adding a ping-pong exchange before Figure 13 (ii), and removing the last exchange gives us the SPLICE/AS protocol.

Thirdly, we see a pattern to which only the Kao Chow protocol belongs in Figure 13 (iii). This pattern is however better than (ii), since shorter and fewer timeouts are used: A needs to wait for the timeout corresponding to only two messages (instead of three as in (ii)), and B has to wait for only one timeout (comparing to two timeouts in (ii)).

Finally, in Figure 13 (iv) we see the last pattern. This pattern is worse than (iii) since B needs to wait longer (for two messages instead of one in (iii)). However, it is unclear whether it is better than (ii), which uses two timeouts of one message each: the actual efficiency and security depends on the actual timeout values used in each case. This category is inhabited by the Yahalom and Neuman Stubblebine protocols.

This taxonomy shows how authentication protocols can be categorized into a handful of patterns. The efficiency and security that an implementation of a protocol will have depends on which pattern the protocol follows, and thus it is useful to consider the patterns in isolation from the actual protocols. In this paper we do not pursue this further, although as future work it would be interesting to further analyze these abstract timing patterns induced from security protocols.
6 Beyond Model Checking: Novel Issues Considering Time

So far our method has been used for analysis purposes, i.e. to model, classify and debug security protocols as a source of hints for the improvement of the protocol implementations. We now explore some ideas to improve the protocols themselves, and also present the threat of a more subtle attack, based only on timing.

6.1 Using Time as Information: Timed Challenges

Sometimes it can be useful to include other timed information than timestamps, even if the clocks are not synchronised. Consider the following protocol, obtained by omitting the encryption of the last message of the (patched) protocol of Section 4.1:

1. A → B : A
2. B → A : \{B, N_B\}_K_A
3. A → B : N_B

Even though N_B is now sent in the clear, this protocol still achieves authentication of A to B, although now the nonce obviously cannot be regarded as a shared secret. Still, the intruder can prevent a successful run of the protocol (e.g. by intercepting message 3), hence the protocol is as strong as it was before in this respect.

Imagine now a situation in which there is a link from A to B in which data can be sent very fast, but at a high cost per bit sent. For example, think that the high cost of sending information comes from the fact that we have devices with a very limited amount of energy, like wireless sensor networks for instance. Alternatively, in some networks, operators charge according to quality of service, and many networks have asymmetric links (e.g. Cable modem and ADSL).

Assume therefore that, sending N_B in message 3 is expensive and not desirable. We propose a solution based exclusively on using time as information. Let \delta_{AB} be the average time it takes for a message to be sent from A to B, and analogously \delta_{BA}. Then consider the “timed” variant of the above protocol, demonstrating how timing information is brought back to the (abstract) protocol level (i.e. Step 3 of Section 4.1):

1. A → B : A
2. B → A : \{B, t_B\}_K_A
3. A → B : “ack” at time \(t_B - \delta_{AB} - \delta_{BA}\)

In Message 2, instead of a nonce, B generates some random time value \(t_B > \delta_{AB} + \delta_{BA}\), concatenates it with B’s identity and encrypts the message with A’s public key. Then, B starts a timer \(t\) and sends the message. Upon reception, A extracts \(t_B\), waits time \(t_B - \delta_{AB} - \delta_{BA}\), and replies the single bit message “ack”. When B receives this message, the timer is stopped and B checks that \(t\) is sufficiently close to \(t_B\); if so, A is authenticated to B. Of course, the amount of noise in the time measurements influences what we mean by “sufficiently close” above. Also, to be realistic, the length in bits of \(t_B\) should be small enough, otherwise B would be waiting too long (on average); this would give an intruder the chance to guess \(t_B\), and answer the “ack” at the appropriate time. Moreover, if encryption \{·\} is deterministic, an attacker can record \{B, t_B\}_K_A and the answers \(t_B\) in a table. As soon as A chooses \(t_B\) again (and this is likely since \(t_B\) is small) and sends out \{B, t_B\}_K_A, the attacker can notice the repeated message in its table since encryption is deterministic, and hence the attacker can intercept the message (so it never arrives to B) and still authenticate to A, since the attacker knows precisely when to send \(t_B\). Probabilistic encryption solves this issue, although even then the attacker can simply guess \(t_B\) and violate authentication.
with non-negligible probability. However, we can strengthen the protocol as follows:

1. $A \rightarrow B : A$
2. $B \rightarrow A : \{B, t_{B_1}, \ldots, t_{B_n}\}_{K_A}$
3. $A \rightarrow B : \text{"ack" at time } t_{B_1} - \delta_{AB} - \delta_{BA}$

$\vdots$

$n + 3. A \rightarrow B : \text{"ack" at time } t_{B_n} - \delta_{AB} - \delta_{BA}$

For example, if $t_{B_i}$ is of length 4 bits, for $i \in [1..n]$, then the total answer is $n$ bits, in comparison with an answer of $4n$ bits required in the nonce protocol.

Of course, sending several short messages can be worse than sending one long message, in which case our protocol would not be so useful. In general, the value of $n$ must be chosen as small as possible, depending on the desired security and network latency. A fast network allows us to reduce $n$ and at the same time increment the length of $t_{B_i}$, for $i \in [1..n]$.

Intuitively, the sent times of the “ack”’s represent information, and the above protocols exploit that. To the best of our knowledge, this is a novel usage of time in security protocols.

**Application** This protocol can be used to authenticate a whole chain of network packets, as follows. Suppose $A$ has a large sequence of $n$ packets which must be streamed to $B$ over a network. For instance, these packets can represent an audio stream in the Internet. We want to authenticate the audio stream, but we do not wish to spend lots of resources on doing this. Let $t_{B_i} \in \{\delta_{AB} + \delta_{BA}, \delta_{AB} + \delta_{BA} + C\}$ for some constant $C$ and $p_i$ denote packet $i$, for $1 \leq i \leq n$. Then the protocol becomes:

1. $A \rightarrow B : A, n$
2. $B \rightarrow A : \{B, t_{B_1}, \ldots, t_{B_n}\}_{K_A}$
3. $A \rightarrow B : p_1 \text{ at time } t_{B_1} - \delta_{AB} - \delta_{BA}$

$\vdots$

$n + 3. A \rightarrow B : p_n \text{ at time } t_{B_n} - \delta_{AB} - \delta_{BA}$

When $t_{B_i}$ is $\delta_{AB} + \delta_{BA}$, it is the delay introduced by $A$ is zero, ie. $p_i$ is sent right away. However, when $t_{B_i}$ is $\delta_{AB} + \delta_{BA} + C$, the delay is $C$. To be as efficient as possible, $C$ should be chosen to be the minimum amount of time that allows $B$ to distinguish the delay modulated by $A$.

In this protocol, only one bit is authenticated per packet. However, the larger the $n$ is, the more confidence we can obtain of $A$’s authentication.

**Discussion** In this protocol, we are in reality exploiting a well-known feature of channel coding: a so-called timing covert channel. In such a channel, the transmitting party modulates packets so that information can be passed even if its not allowed by the environment. Our usage differs in three ways:

- Firstly, we use a mixed approach, in which some information is sent in the standard channel, and other is sent in the timing channel.

- A second difference is more fundamental than the previous one. Our usage of the timing channel is purposely public, and there is no environment trying to stop the unauthorized information flow. Timing is used only because of its practical advantages, namely low-bandwidth.

- Finally, in our protocol both communicating parties do not initially trust on the other’s identity, in principle: Indeed, ours is an authentication protocol.
6.2 Timing Leaks in an Implementation of the Private Authentication Protocol

We now present a threat against an implementation of security protocols with branching: the so-called timing attack. We illustrate this by showing an attack over a careless implementation of Abadi’s Private Authentication (PAP) protocol [2] (The second protocol). It is worthwhile to mention that the protocol has been proved correct by Abadi and Fournet [21] in a setting without time. We assume that each principal \( X \) has a set of communication parties \( S_X \), listing the principals with whom \( X \) can communicate. The aim of the protocol is to allow a principal \( A \) to communicate privately with another principal \( B \). Here “privately” means that no third party should be able to infer the identities of the parties taking part in the communication (i.e. \( A \) and \( B \)) (PAP Goal 1). Moreover, if \( A \) wants to communicate with \( B \) but \( A \not\in S_B \), the protocol should also conceal \( B \)’s identity (and presence) to \( A \) (PAP Goal 2). A run of the protocol in which \( A \) wants to communicate with \( B \) proceeds as follows:

1. \( A \) generates a nonce \( N_A \). Then, \( A \) prepares a message \( M = \{\text{“hello”}, N_A, K_A\}_{K_B} \), and broadcasts (“hello”, \( M \)).

2. When a principal \( C \) receives message (“hello”, \( M \)), it performs the following three steps:

   (a) \( C \) tries to decrypt \( M \) with its own private key. If the decryption fails, (which implies that \( C \neq B \)), then \( C \) creates a “decoy” message \( \{N\}_K \) (creating a random \( K \), and keeping \( K^{-1} \) secret), broadcasts (“ack”, \( \{N\}_K \)) and finish its execution. If decryption succeeds, then \( C = B \) (and so from now on we will refer to \( C \) as \( B \)). \( B \) then continues to the next step.

   (b) \( B \) checks that \( A \in S_B \). If this fails, i.e. \( A \not\in S_B \), then \( B \) creates a “decoy” message \( \{N\}_K \), broadcasts (“ack”, \( \{N\}_K \)) and finishes execution. Otherwise \( B \) continues to the next step.

   (c) Finally, \( B \) generates a fresh nonce \( N_B \), and broadcasts the message (“ack”, \( \{\text{“ack”}, N_A, N_B, K_B\}_{K_A} \)).

It is interesting to see the use of “decoy” messages, to prevent attacks in which an intruder \( I \) prepares a message \( M = \{\text{“hello”}, N_C, K_A\}_{K_B} \), impersonating principal \( A \). If decoy messages were not present, then \( I \) would send (“hello”, \( M \)), and deduce whether \( A \in S_B \) by noticing a response from \( B \). However, using decoys only helps to confuse an attacker doing traffic analysis, and breaks down when considering a “timed” intruder, as we will show in the next section.

An Attack Over an Implementation of the Private Authentication Protocol

We show an attack in which \( I \) can find whether \( A \in S_B \). The attack is illustrated in Figure 14, where \( I \) is trying to attack \( A \), \( B \) and \( C \), which since \( I \) does not know their identities are called \( X \), \( Y \) and \( Z \). First, suppose that \( I \not\in S_B \) (the attack for the case in which \( I \in S_B \) is analogous). First \( I \) needs to know how long, on average, it takes to \( B \) to compute each step of the protocol as described above. To discover this \( I \) could prepare various messages:

1. Firstly, \( I \) sends a message (“hello”, \( \{N\}_K \)), where \( K \) is not the public key of any other participant. This would generate a number of decoy responses from the other participants, which \( I \) can time (Step 1 in Figure 14 for times \( x \), \( y \) and \( z \)).

2. Secondly, \( I \) sends a message (“hello”, \( \{\text{“hello”}, N_I, K_I\}_{K_B} \)). Again, this generates decoy responses from the other parties which \( I \) can time (Step 2 in the figure). However, if \( B \) is present, then one response will have longer time (i.e. we get times \( x \), \( z \) and \( y' \) with \( y' \) longer than \( y \)), reflecting the successful decryption and check that \( I \not\in S_B \) performed by \( B \) (Recall we assume that \( I \not\in S_B \)). Up to this point, \( I \) has information that allows him to
Figure 14: The attack over the PAP protocol, where X, Y and Z are unknown identities by the intruder I. A, B and C are real identities, with corresponding $S_A$, $S_B$ and $S_C$ sets; $x$, $y$, $y'$, $y''$ and $z$ are timing values (Dashed circles indicate the intruder’s knowledge of inner values)
infer B’s presence (hence the dashed circle in the figure); Thus, this attack already violates goal 2 of Abadi’s requirements \[2\]: B should protect its presence if a party X is willing to communicate with B but \(X \not\in S_B\) (Step 2).

3. Finally, if B is present, then I sends message (“hello”, \{“hello”, \(N_I, K_A\}\)\(K_B\)). This would generate again the same decoy responses, except one that takes longer (Step 3 in the figure, for \(x, z\) and \(y''\) with \(y''\) longer than \(y'\)). If this response takes the same time as the above item, then I can deduce that \(A \not\in S_B\). Otherwise, if the response takes longer (reflecting the nonce generation \(N_B\) and encryption performed by B) then I can deduce that \(A \in S_B\).

If \(I \in S_B\), then the second step above returns the longest time, and the third message would take either less time or equal.

After recording this information, I has three time values \(t_0, t_1\) and \(t_2\). \(t_0\) corresponds to the time in which B is not present; \(t_1\) corresponds to the time in which B is present but its communicating party \(X \not\in S_B\). Finally, \(t_2\) corresponds to the case in which B is present and its communicating party \(X \in S_B\). With these values at hand, now an attacker can check if \(A \in S_B\) for an arbitrary A.

Timing in networks is often accurate but if the accuracy is too low, the intruder can repeat the timings (i), (ii) and (iii) and perform statistical analysis to increase the probability of the inferences to be correct \[23\]. We propose this as future work, when we have a probabilistic, timed model checker at disposal.

7 Related Work

Many approaches focus on the study of protocols that use timestamps \[15, 18, 27, 6, 22\]. Recent work of Delzanno et.al. \[15\] presents an automatic procedure to verify protocols that use timestamps, like the Wide Mouthed Frog protocol. In their work, differently from ours, a global clock is assumed, and timeouts and retransmissions are not discussed. Evans and Schneider \[18\] present a framework for timed analysis. Differently from our (UPPAAL) model checking, it is based on a semi-decision procedure with discrete time. In that work, the usage of retransmissions is hinted at as future work, but not (yet) addressed. Lowe \[27\] also analyses protocols with timing information; his work shares with us the model checking approach, although Lowe’s approach is based on a discrete time model. A global clock is also assumed, and timeouts nor retransmissions are addressed. Closer to ours is the work of Gorrieri et al. \[22\], in which a real-time process algebra is presented for the analysis of time-dependent properties. They focus on compositionality results, and no model checking is presented. Gorrieri et al. also show how timeouts can be modelled, although retransmissions are not discussed.

Regarding our timing attack upon Abadi’s protocol, Focardi et al \[20\] develop formal models for Felten and Schneider’s web privacy timing attack \[19\]; their modelling activity shares with our work the idea of using timed automata for analysis, although our attack illustrates a timing attack over a “pure” security protocol.

8 Conclusions

Security protocol analysis is a mature field of research that has produced numerous results and valuable tools for the correct engineering of security protocols. Despite a large body of literature on the subject, most analysis methods do not take time into consideration (with the exception of a few papers, considering mainly the use of timestamps). We argue that this is not realistic as all distributed protocols need to implement timeouts (possibly followed by retransmissions).

In this paper we address some of the issues that need to be considered when including time-related parameters in the engineering process of a security protocol.
Our first contribution is a method for the design and analysis of security protocols that consider timing issues. We model security protocols using timed automata, and use UPPAAL to simulate, debug and verify security protocols in a real time setting. To this end, we employ a general Dolev Yao style intruder (naturally encoded in UPPAAL), and we remark that modelling the intruder as a timed automata implicitly extends its power to take into account the time sensitivity. Our method allows us to specify security protocols in detail, with timeouts and retransmissions. This increases the confidence in the analysis, since the modelled protocols are closer to their implementations than the classical analysis (e.g. CASPER [27] or the constraint-based methods of [28, 10]).

Secondly, by analyzing the protocols in the Clark and Jacob library and the SPORE library, we see that most protocols schemas (w.r.t. timeouts) fit into a small number of common patterns. We analyse the efficiency and security of each of the patterns. Still, as possible future work we would like to perform a full UPPAAL analysis of each of these literature protocols (just as we do for Yahalom in Section 4.2).

Other novel and more real-life protocols which are sensitive to timing issues (e.g. besides timeouts, use for instance puzzles) may benefit from our analysis, e.g. the Host-Identity-Protocol (HIP), initially analysed by Aura et al. [5].

Our third contribution is an illustration of the implicit information carried by timing. The mere act of sending a message at a specific moment in time, and not another, carries information. We propose a novel security protocol that exploits this fact to achieve authentication. The protocol replaces the standard nonces with timed challenges, which must be replied at specific moments in time to be successful. Although it is a preliminary idea, it exposes clearly the fact that security protocols can use and take advantage of time.

Finally we address threats specifically involving timing should also be considered; specifically, timing attacks. We illustrate these attacks in the context of security protocols, where branching allows an intruder to deduce information that is intended to be kept secret. Specifically, we mount an attack over an imperfect implementation of Abadi’s private authentication protocol [2]. Solutions to avoid timing attacks in the implementations are usually expensive (e.g. noise injection or branch equalization), and it is not our purpose to investigate them. Here we merely lift the known problem of timing attacks, typically mounted against the cryptosystem to obtain secrets, to security protocols in general where the information leakage can be, in principle, anything.

One possible direction of future work is to consider a timed and probabilistic model checker (in the lines of [13] or [24]), that would allow us to study the protocols of Section 6. Moreover, a probabilistic setting would allow us to model, more realistically, the network latency. This, in turn, would provide us with a finer method to tune sensitive timing values. Another possible direction for future research would be to implement a compiler from a meta notation (similar to the standard notation, plus timing information) supporting symbolic terms, to UPPAAL automata. Ultimately, these directions of future work would contribute to a method of secure systems engineering.

Acknowledgements Gabriele Lenzini, Conrado Daws, Jeroen Doumen, Jerry den Hartog, Ari Saptawijaya and the anonymous reviewers of FMSE2004 provided useful comments.

References


