Timed Model Checking of Security Protocols

R. Corin[1]*, S. Etalle[1,2], P.H. Hartel[1] and A. Mader[1]
[1] Faculty of Computer Science,University of Twente, The Netherlands
[2] CWI, Center for Mathematics and Computer Science Amsterdam

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. Timing information allows the study of different attack scenarios. We illustrate the attacks by model checking the protocol using Uppaal.

We also present new challenges and threats that arise when considering time.

Keywords: model checking, timed automata, security protocols.

1 Introduction

Typically, methods for formal verification of security protocols (among the proposals [12, 21, 10, 8]) 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.

There are other aspects of *time* that influence the behaviour of a protocol. Consider a simple protocol, written in the usual notation:

1. $A \rightarrow B \quad : \quad M_{AB}$
2. $B \rightarrow A \quad : \quad M_{BA}$

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 *timers*. In this paper, we do not require timers between parties to be synchronized (see below for a discussion).

The next step consists in distinguishing the different operations that occur, with their respective times. In message 1, it takes time to create M_{AB}. The other operation that takes time is the actual sending of the message, ie. the time it takes the message to traverse from A to B. The first complication appears here: the 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} (message 2 of the above protocol) is not received after some waiting, say t_A. 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}. Typically, the value for t_A depends on implementation details. However, a quantitative analysis indicates what attacks can be mounted for some values of t_A that are no longer possible for other (eg. smaller) values.

Another, usually not considered issue is the *action* to be taken when a timeout occurs. Typically, the implicit assumption is that the protocol should abort. This means that the protocol party that reaches the timeout deduces that a fault has happened. However, this

*Contact author email: corin@cs.utwente.nl.

could *not* be the case [19], and protocols could execute by considering messages that *do not* arrive at a certain moment. Moreover, 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. Then the question is whether to retransmit the identical message M_{AB}, or to recompute some parts before resending the message. Here, the tradeoff is 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 synchronization of A and B, which is expensive (see [23] 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 [5]. Nevertheless, much effort has been dedicated to the analysis of security protocols which use timestamps (see Related Work), so in this paper we do not pursue this direction.

As first and main contribution, in Section 2 we present a method for the engineering of security protocols that considers timing issues, like timeouts and retransmissions. The method is based on modelling security protocols using timed automata [2]. Furthermore, we use Uppaal [3] as a tool to simulate, debug and verify security protocols against classical properties like secrecy and authentication, *in a real time scenario*. As examples, we analyse a simplified version of the Needham Schroeder protocol [20] and the full Yahalom protocol [7].

Secondly, in Section 3 we illustrate the opportunities and difficulties that appear when considering time in the analysis of security protocols. First, 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, we describe how timing attacks [13] can be applied to security protocols, by describing an attack over Abadi's private authentication protocol [1]. Although the protocols can be modelled as timed automata, we leave the verification as future work since we need a model checker that is also *probabilistic* (like [9]): our nondeterministic intruder of Uppaal is too powerful, and it can always guess correctly times and values, even if the probability of guessing is negligible.

1.1 Related Work

As we already mentioned, many approaches focus on the study of protocols that use timestamps, see e.g. [11, 14, 21, 4, 18].

Recent work of Delzanno et.al. [11] 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 [14] 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 [21] also analyses protocols with timing information; his work shares with us the model checking method, 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, Locatelli and Martinelli [18], 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. The authors also show how timeouts can be modelled, although retransmissions are not discussed.

Regarding our timing attack upon Abadi's protocol, Focardi et al [16] develop formal

models for Felten and Schneider's *web privacy* timing attack [15]; 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.

2 A Method for Engineering Security Protocols

We use timed automata [2] to model protocol participants, and this has several advantages. First, our method requires the designer to provide a precise and relatively detailed protocol specification, which helps to disambiguate the protocol behaviour. Second, timing values like timeouts need to be set at each state, while retransmissions can be easily specified as transitions to other protocol states.

Once modelled as automata, the protocol can be fed to the real time tool Uppaal, which allows the protocol to be simulated. 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, verification of the protocol is automatic for finite scenarios.

The resulting 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 3.1), to make time an integral aspect of the protocol itself.

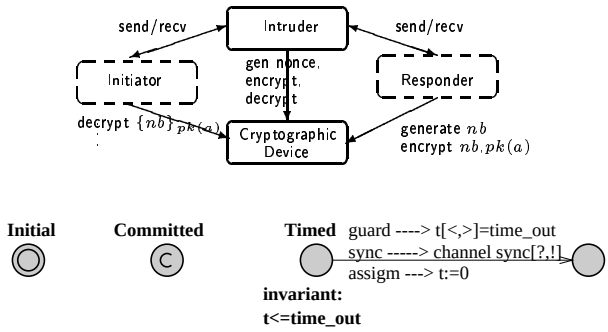

Figure 1: Modelling cryptography in Uppaal (top) State Constructions (bottom)

2.1 The Uppaal Model

Let us now describe the general form of our models, 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 1 (top) we illustrate a scenario 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 [12]). Our modelling is *modular*, and allows us to "plug in" different participants (eg., in the analysis of the Yahalom we add a *server*), while the bold boxes, ie. the intruder and the cryptographic device, are the core model.

In Figure 1 (bottom) we show the different kinds of state constructions used in our models. Designers should use these constructs as building blocks for the representation of protocol participants. Leftmost we see the *ini-*

tial state, shown as a double circled state. In the middle, a state with a "C" denotes a *committed* state, which indicates that the automaton should not wait while being at that state, but should immediately through one enabled transition. Rightmost we see a *timed* state, in which time is allowed to pass while the automaton is at that state. The state can have a time invariant (in our framework, we use the invariants to model timeouts). Transitions may come in and out from the states; the transitions may have *guards* to be enabled (requiring, for example, that the timeout has (not) occurred to proceed), *channel synchronizations* ("?" is an input and "!" an output) and *variable assignments* (eg., resetting a clock).

2.2 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. [20]). Differently from the Needham Schroeder protocol whose goal is to achieve mutual authentication, our simpler protocol aims at authenticating 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:

$$\begin{array}{lcl} 1.\ A \rightarrow B & : & A \\ 2.\ B \rightarrow A & : & \{N_B\}_{K_A} \\ 3.\ A \rightarrow B & : & \{N_B\}_{K_B} \end{array}$$

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.

Modelling While modelling security protocols as timed automata in Uppaal, we will focus on modelling the times required by the agents 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).

The automaton for the cryptographic device is presented in Figure 2. Basically, the device 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. The local variable *gennonce* is incremented with the constant *seed*, modelling the new nonce creation. After synchronization, a global `result` variable is updated with a generated nonce, and after some more time the device finishes by synchronizing on the *finish_nonce* channel.

Encryption and decryption is 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 synchronizes 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 $10k$, since anyway the intruder has not the ability to multiply by 10, hence cannot guess the private keys of agents.

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

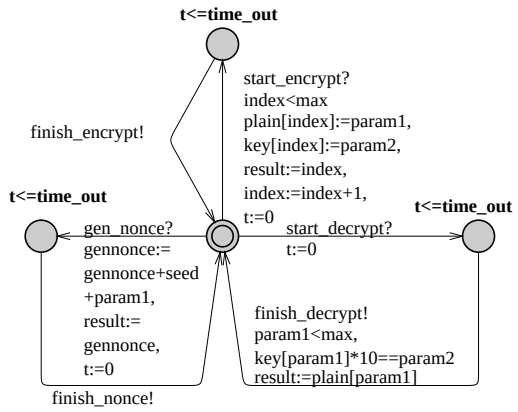

Figure 2: Timed automaton for a Cryptographic Device

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* agent.

The automata for the initiator and responder of our simple protocol presented above are given in Figure 3 (the dashed transitions of the responder correspond to retransmissions, discussed in Section 2.2). The initiator A starts her execution when activated via channel *start* (State SI0). The actual agent playing 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, we try to decrypt the received message `msg` (SI3)). This takes time `time_decrypt` for the initiator. After this decryption (State (SI5)), the initiator reencrypts the obtained nonce (stored in `result`) (State (SI7)) and finally sends the last message via the *init_msg* channel, and 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 a response (State (SR1)). When received, B saves the first message in the local variable `claimed_id` (State (SR2)). After this, B generates a nonce by contacting the cryptographic device. When ready (State (SR4)), it encrypts the nonce with the value received in message 1 (we identify identities with public keys). After finishing the encryption (State (SR5)), the message is sent and B starts to wait for a response (State (SR6)). If an answer comes before the timeout (State (SR7)), B decrypts the message and checks that the challenge is indeed the one he sent. If so, its local boolean variable `finish` is set to `true`.

The intruder, presented in Figure 4, works basically as a Dolev-Yao intruder [12]. 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 synchronize on both channels *init_msg* and *resp_msg*. Upon synchronizing by receiving a message, the intruder moves to state (SINT1), where it saves the message `msg` in its local variable `data` and resets an index variable i which bounds the total number of actions allowed to do at that round. Then, the intruder moves to state (SINT3), where it makes a nondeterministic choice of an action. More precisely, it can decide to:
– Choose an identity in its local variable `pk` (State (SINT4))
– Encrypt a value (State (SINT5))
– Decrypt a value (State (SINT6))
– Generate a nonce (State (SINT7))
– Save variable `data` as message `msg`.
The intruder can then continue doing these ac-

tions, 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).

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 real agents (like A, B and I), and then starts the execution run by broadcasting via the *start* channel.

The property we check, AUT, is shown in Table 1. AUT states that if we reach a state in which the responder has finished his execution but the claimed id (corresponding to the first message of protocol) does *not* coincide with the actual identity of the initiator, then 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 [22] for a classification of different authentication notions).

If we use a long timeout for B, ie. $B.\texttt{time\_out} \geq \mathit{Intruder}.\texttt{time\_decrypt} + \mathit{Intruder}.\texttt{time\_encrypt} + A.\texttt{time\_encrypt} + A.\texttt{time\_decrypt}$, then Uppaal finds a man-in-the-middle attack, presented in the left side of Table 2. This attack is similar to Lowe's attack [20]. 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, ie. $B.\texttt{time\_out} < \mathit{Intruder}.\texttt{time\_decrypt} + \mathit{Intruder}.\texttt{time\_encrypt} + A.\texttt{time\_encrypt} + A.\texttt{time\_decrypt}$. 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 $B.\texttt{time\_out} = A.\texttt{time\_encrypt} + A.\texttt{time\_decrypt}$, leaving *no* space for any interruption.

A second attack which is independent of timeouts (even if we set $B.\texttt{time\_out} = 0$!) was surprisingly found by Uppaal; this time, the vulnerability is much simpler. We report it in the middle part of Table 2. This attack corresponds to a "reflection" replay attack [24], that occurs when the intruder simply replies B's message. 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 (this protocol is presented in the right side of Table 2). However, even though our framework is capable of finding *untimed* attacks (and thus confirming known attacks), we aim at providing a good baseline to study extended security protocols with timing issues, like timeouts and retransmissions.

Retransmissions Consider again the automaton for the responder, given in Figure 3. In state (SR5), the responder sends the challenge $\{N_B\}_{K_A}$, and waits for a response in state (SR6). 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 (SR6) to previous states of the automaton (the dashed lines in Figure 3); These transitions are guarded, allowing to perform the action only when the timeout is reached (ie., $\texttt{t} >= \texttt{time\_out}$). (Furthermore, the transitions could have a counter for allowing only a finite number of retransmis-

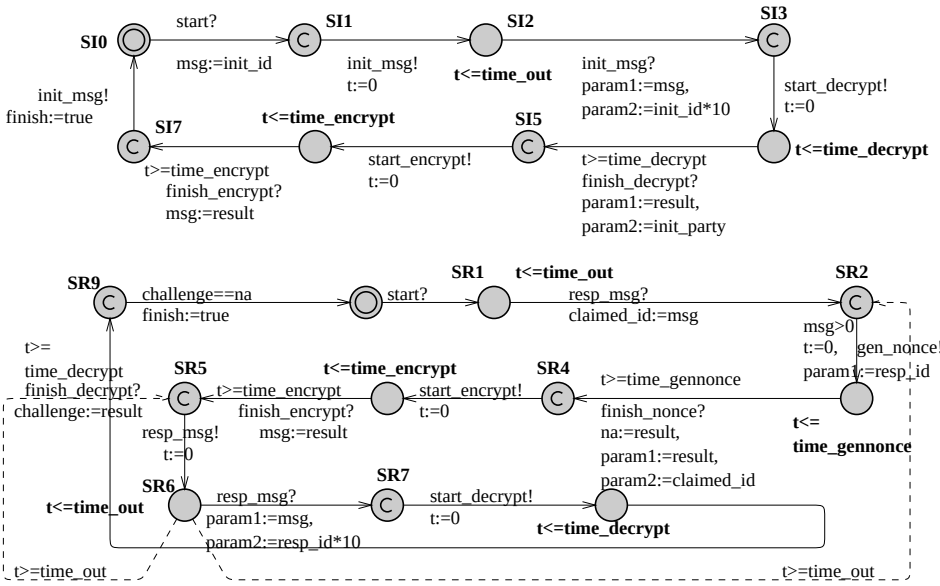

Figure 3: Timed automaton for the Initiator (top) and the Responder (bottom)

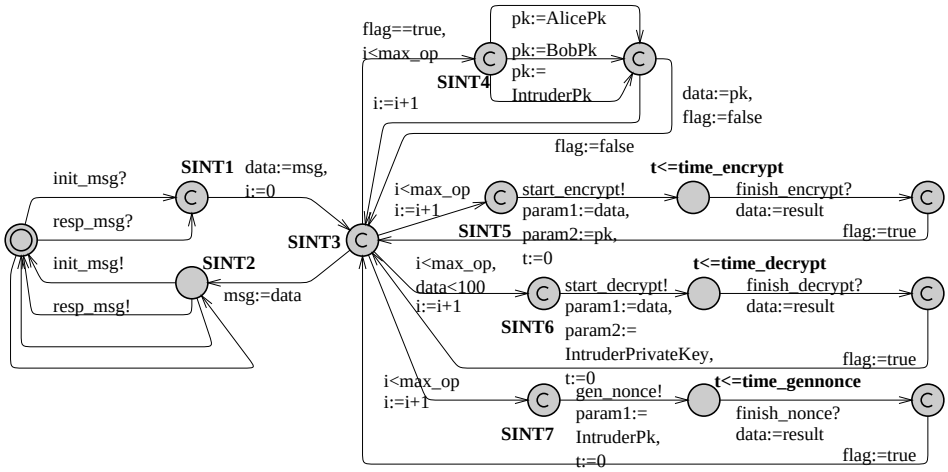

Figure 4: Timed automaton for the Intruder

AUT	=	E <> Responder.finish and Responder.claimed_id! = resp_party
AUT_y	=	E <> Initiator.finish and Initiator.ticks < (Responder.time_encrypt +Responder.time_gennonce + Server.time_encrypt * 2 + Server.time_decrypt) − 1

Table 1: Left: Uppaal properties.

$\alpha.1$	$A \to I$	$:A$	1. $I(B) \to B$	:	B	1. $A \to B$	:	A
$\beta.1$	$I(A) \to B$	$:A$	2. $B \to I(B)$	:	$\{N_B\}_{K_B}$	2. $B \to A$	:	$\{B, N_B\}_{K_A}$
$\beta.2$	$B \to I(A)$	$: \{N_B\}_{K_A}$	3. $I(B) \to B$	:	$\{N_B\}_{K_B}$	3. $A \to B$	:	$\{N_B\}_{K_B}$
$\alpha.2$	$I \to A$	$: \{N_B\}_{K_A}$						
$\alpha.3$	$A \to I$	$: \{N_B\}_{K_I}$						
$\beta.3$	$I(A) \to B$	$: \{N_B\}_{K_B}$						

Table 2: Left: A man-in-the-middle attack. Middle: A replay attack. Right: Patched protocol.

sions before aborting.)

We consider two potential target states for our Responder of Figure 3, namely states (SR5) 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 move 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.

2.3 A real protocol

We study the Yahalom protocol [7]. 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. [6] 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:

$$
\begin{array}{lcl}
1.\ A \to B & : & A, N_A \\
2.\ B \to S & : & B, \{N_B, A, N_A\}_{K_{BS}} \\
3.\ S \to A & : & \{B, K_{AB}, N_A, N_B\}_{K_{AS}}, \\
 & & \quad \{A, K_{AB}, N_B\}_{K_{BS}} \\
4.\ A \to B & : & \{A, K_{AB}, N_B\}_{K_{BS}}, \{N_B\}_{K_{AB}}
\end{array}
$$

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, agent 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 4. We have modelled the protocol in Uppaal (the initiator, responder and server are shown in the Appendix, Figures 5 and 6).

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 happens 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. 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 5). 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 its message intact, should recompute N_B or it should abort, since clearly N_A cannot be recomputed. Intuitively, N_B could be reused. However, if B knows that A will timeout and abort, then B could simply abort as well. We modelled in Uppaal the retransmission of the exact message (as the dashed transition of the responder in Figure 6). 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.

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. We believe that these are useful and practical hints that help to develop more secure and efficient implementations, in this case of the Yahalom protocol.

3 New issues considering time

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

3.1 Using time as information: Timed Challenges

We now show that sometimes other time information than timestamps can still be useful to include in security protocol messages, *even* if the clocks are not synchronized. Consider the following protocol, obtained by omitting the encryption of the last message of the (patched) protocol of Section 2.2:

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

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 (eg. 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 fast, but at a high cost wrt the amount of sent bits. This is realistic; network operators charge

according to quality of service. In this setting, sending N_B in message 3 could be expensive and not desirable. We propose a solution based exclusively on using *time* as information. Let δ_{AB} be the average time it takes a message to be sent from A to B, and analogously δ_{BA}. Then consider the "timed" variant of the above protocol, demonstrating how timing information is brought back to the (abstract) protocol level (ie. Step 3 of Section 2):

1. $A \to B$: A
2. $B \to A$: $\{B, t_B\}_{K_A}$
3. $A \to B$: "*ack*" at time $t_B - \delta_{AB} - \delta_{BA}$

In message 2, 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, he stops the timer t and checks that t is *sufficiently* close to t_B; if so, A is authenticated to B. Of course, the amount of noise in the links influence 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; this would give an intruder the chance to guess t_B, and answer the "*ack*" at the appropriate time. However, we can strengthen the protocol as follows:

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

$\vdots$

$n+3$. $A \to B$: "*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.

3.2 Timing Leaks and the Private Authentication Protocol

We now present a threat over security protocols with branching: the so called timing attack. We illustrate this by showing an attack over Abadi's Private Authentication protocol. This protocol is presented in [1] (their *second* protocol), and proved correct by Abadi and Fournet [17]. We assume that each agent X has a set of communication parties S_X, listing the agents with whom X can communicate. The aim of the protocol is to allow an agent A to communicate *privately* with another agent B. Here "privately" means that no third party should be able to infer the identities of the parties taking part of the communication (i.e. A and B) (Goal 1 of [1]). Moreover, if A wants to communicate with B but $A \notin S_B$, then the protocol should also conceal B's identity (and presence) to A (Goal 2 of [1]). 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 $M = \{\text{"}hello\text{"}, N_A, K_A\}_{K_B}$, and broadcasts $(\text{"}hello\text{"}, M)$.
2. When an agent C receives message $(\text{"}hello\text{"}, M)$, it performs the steps: First,

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 $\{N\}_K$ (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.

Second, B checks that $A \in S_B$. If this fails, i.e. $A \notin S_B$, then B creates a "decoy" message $\{N\}_K$, broadcasts ("*ack*", $\{N\}_K$) and finish its execution. Otherwise B continues to the next step.

Finally, B generates a fresh nonce N_B, and broadcasts message ("*ack*", {"*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 $M = \{$"*hello*", $N_C, K_A\}_{K_B}$, impersonating agent 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.

We show an attack in which I can find whether $A \in S_B$. First, suppose that $I \notin S_B$ (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:

First, 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.

Second, I sends a message ("*hello*", {"*hello*", $N_I, K_I\}_{K_B}$). Again, this generates decoy responses from the other parties which I can time. However, if B is present, then *one* response will have longer time, reflecting the successful decryption and check that $I \notin S_B$ performed by B (Recall we assume that $I \notin S_B$). Up to this point, I has information that allows him to infer B's presence; Thus, this attack already violates goal 2 of Abadi's requirements [1]: B should protect its presence if a party X is willing to communicate with B but $X \notin S_B$.

Thirdly, 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. If this response takes the same time as the above item, then I can deduce that $A \notin S_B$. 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 for the time in which B is not present; t_1 corresponds for the time in which B is present but its communicating party $X \notin S_B$. Finally, t_2 corresponds for 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.

4 Conclusions

We model security protocols using timed automata. Using Uppaal allows us to simulate, debug and verify security protocols in a real time setting. The intruder model can be naturally encoded in Uppaal, showing that using a general tool is feasible. We modelled a general Dolev-Yao style intruder, although its modelling as timed automata extends its power implicitly, to take into account the time sensitivity. The consideration of time allows

us to specify security protocols in detail, with timeouts and retransmissions, which is closer to an implementation. Analysis of security under these detailed specifications provides further confidence of the security of a protocol implementation.

The mere act of sending a message at a specific moment in time, and not another, carries information. We propose a 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.

Threats specifically involving timing should also be considered, and one example is 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 Abadi's private authentication protocol [1]. Solutions to timing attacks are expensive (including noise injection or branch equalisation), and it is not our purpose to investigate such possibilities; we merely lift the known problem of timing attacks (typically mounted to obtain secrets keys) to security protocols in general, where the information leakage can be, in principle, anything.

As we already mentioned, one possible direction as future work is to consider a timed and probabilistic model checker (like [9]), that would allow us to study the protocols of Section 3.

Moreover, using a probabilistic setting would also 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 would be to implement a compiler from a meta notation (similar to the standard notation, plus timing information) supporting symbolic terms, to Uppaal automata (with only integer values as datatype).

Ultimately, these directions of future work would contribute to a method of secure systems engineering.

References

[1] M. Abadi. Private authentication. In R. Dingledine and P. F. Syverson, editors, *2nd. Int. Workshop on Privacy Enhancing Technologies (PET)*, volume LNCS 2482, pages 27–40, San Francisco, California, Apr 2002. Springer-Verlag, Berlin.

[2] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, (138):183–335, 1994.

[3] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Möller, P. Pettersson, C. Weise, and W. Yi. UPPAAL - now, next, and future. In F. Cassez, C. Jard, B. Rozoy, and M. D. Ryand, editors, *4th Summer School on Modeling and Verification of Parallel Processes (MOVEP)*, volume LNCS 2067, pages 99–124, Nantes, France, Jun 2000. Springer-Verlag, Berlin.

[4] G. Bella and L. C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In J.-J. Quisquater, editor, *Proc. 5th European Symposium on Research in Computer Security*, volume 1485 of *LNCS*, pages 361–375, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag.

[5] S. M. Bellovin and M. Merritt. Limitations of the kerberos authentication system. *SIGCOMM Comput. Commun. Rev.*, 20(5):119–132, 1990.

[6] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[7] J. Clark and J. Jacob. *A survey of authentication protocol literature.* Univ. of York, 2000.

[8] R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In M. V. Hermenegildo and G. Puebla, editors, *9th Int. Static Analysis Symp. (SAS)*, volume LNCS 2477, pages 326–341, Madrid, Spain, Sep 2002. Springer-Verlag, Berlin.

[9] P. R. D'Argenio, H. Hermanns, J.-P. Katoen, and J. Klaren. MODEST: A modelling language for stochastic timed systems. In L. de Alfaro and S. Gilmore, editors, *Joint Int. Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV)*, volume LNCS 2165, pages 87–104, Aachen, Germany, Sep 2001. Springer-Verlag, Berlin.

[10] G. Delzanno and S. Etalle. Proof theory, transformations, and logic programming for debugging security protocols. In A. Pettorossi, editor, *11th Int. Logic Based Program Synthesis and Transformation (LOPSTR)*, volume LNCS 2372, pages 76–90, Paphos, Greece, Nov 2001. Springer-Verlag, Berlin.

[11] G. Delzanno and P. Ganty. Automatic verification of time sensitive cryptographic protocols. pages 342 – 356, March 2004.

[12] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[13] E. English and S. Hamilton. Network security under siege: the timing attack. *IEEE Computer*, 29(3):95–97, Mar 1996.

[14] N. Evans and S. Schneider. Analysing time dependent security properties in CSP using PVS. In *ESORICS*, pages 222–237, 2000.

[15] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *7th ACM conference on Computer and communications security*, pages 25–32, Athens, Greece, 2000. ACM Press, New York.

[16] R. Focardi, R. Gorrieri, R. Lanotte, A. Maggiolo-Schettini, F. Martinelli, S. Tini, and E. Tronci. Formal models of timing attacks on web privacy. In Marina Lenisa and Marino Miculan, editors, *Electronic Notes in Theoretical Computer Science*, volume 62. Elsevier, 2002.

[17] C. Fournet and M. Abadi. Hiding names: Private authentication in the applied pi calculus. In *Symp. on Software Security*, volume LNCS 2609, pages 317–338, Tokyo, Japan, Jan 2003. Springer-Verlag Heidelberg.

[18] R. Gorrieri, E. Locatelli, and F. Martinelli. A simple language for real-time cryptographic protocol analysis. In Pierpaolo Degano, editor, *12th European Symposium on Programming, ESOP 2003*, volume 2618 of *Lecture Notes in Computer Science*, pages 114–128, Heidelberg, 2003. Springer-Verlag.

[19] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2), April 1984.

[20] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, November 1995.

[21] G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW '97)*, pages 18–30. IEEE, 1997.

[22] G. Lowe. Casper: A compiler for the analysis of security protocols. In *10th Computer Security Foundations Workshop (CSFW)*, pages 18–30, Rockport, Massachusetts, Jun 1997. IEEE Computer Society Press, Los Alamitos, California.

[23] D. Mills. Cryptographic authentication for real-time network protocols. 45:135–144, 1999.

[24] P. Syverson. A taxonomy of replay attacks. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, 1994.

A Yahalom Automata

We show the Initiator in Figure 5, Responder and Server in Figure 6.

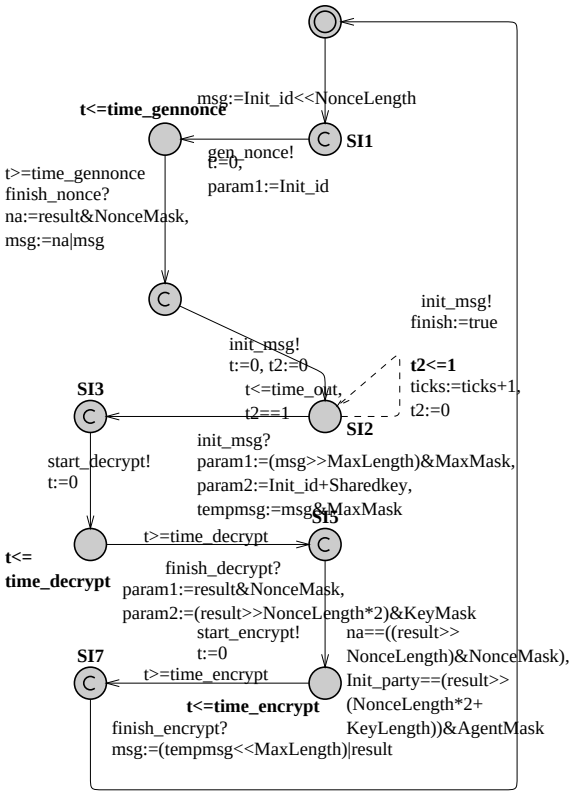

Figure 5: Timed automaton for the Yahalom Initiator

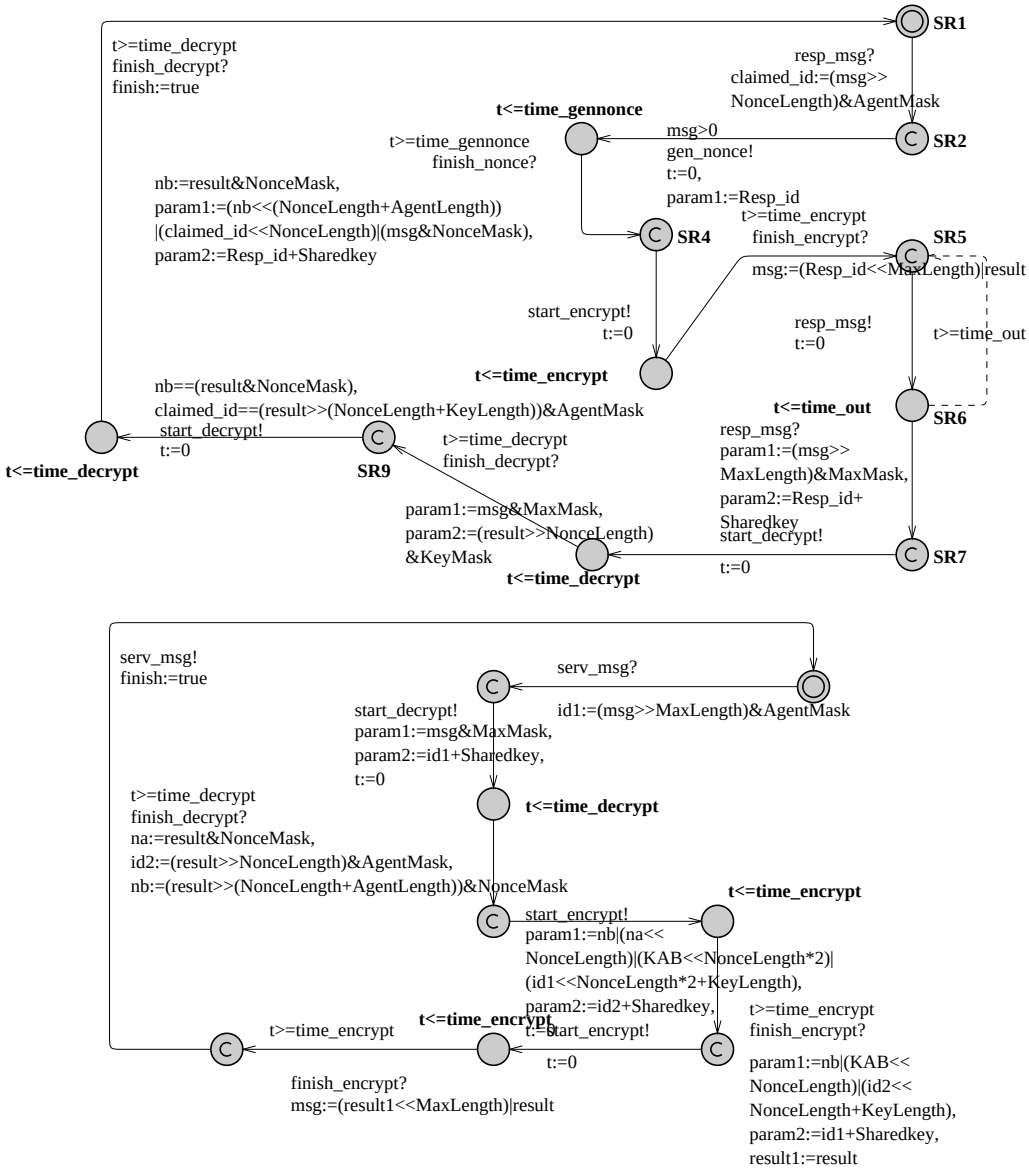

Figure 6: Timed automaton for the Yahalom Responder (top) and Server (bottom)