Is Your Security Protocol on Time?*

Gizela Jakubowska\textsuperscript{1} and Wojciech Penczek\textsuperscript{2}

\textsuperscript{1} Szczecin University of Technology, Faculty of Computer Science and Information Systems
gjakubowska@wi.ps.pl
\textsuperscript{2} Institute of Computer Science, PAS, and Institute of Informatics, Podlasie Academy
penczek@ipipan.waw.pl

Abstract. In this paper we offer a novel methodology for verifying correctness of (timed) security protocols. The idea consists in computing the time of a correct execution of a session and finding out whether the Intruder can change it to shorter or longer by an active attack. Moreover, we generalize the correspondence property so that attacks can be also discovered when some time constraints are not satisfied. As case studies we verify generalized authentication of KERBEROS, TMN, Neumann Stubblebine Protocol, Andrew Secure Protocol, WMF, and NSPK.

1 Introduction

Security (or authentication) protocols define the rules of exchanging some messages between the parties in order to establish a secure communication channel between them, i.e., they provide the mechanism aimed at guaranteeing that the other party is who they say they are (authentication), that confidential information is not visible to non-authorised parties (secrecy), and that the information exchanged by two parties cannot be altered by an intruder (integrity). There are several approaches to verification of untimed security protocols, see e.g., [1–5]. Quite recently there have also been defined approaches to verification of time dependent protocols [6–9]. Our approach is closer to the work by Corin at al. [10], where security protocols are directly modeled in terms of networks of timed automata extended with integer variables, and verified with UppAal [11]. The authors address timeouts and retransmissions, but do not show how one can model timestamps [12] in such an approach.

There are several methods for finding attacks on security protocols. One can check whether the authentication property is satisfied, but this is not sufficient for discovering several ‘authentication-independent’ types of attacks. So, knowing the type of an attack one check whether it can occur or not. But, how to look for entirely unknown (types of) attacks? Clearly, we can test the knowledge of the Intruder in order to find out whether he possesses some ‘insecure’ information. This requires either to use a special (epistemic) formalism for expressing

* The authors acknowledge partial support from the Ministry of Science and Information Society Technologies under grant number 3 T11C 011 28 and N516 038 31/3853.
properties or to encode the knowledge of the Intruder into the states of the model, which leads to either making verification or the model itself more complicated. Moreover, no feedback for implementators follows from such a method of checking correctness. This paper\(^1\) offers a novel method for finding attacks of any type either known or unknown. Our method consists in computing the time of a correct execution of a session and finding out whether the Intruder can change it to shorter or longer by an active attack. This method can be applied to both the timed as well as untimed protocols. To this aim and in order to make our modelling closer to real applications, the model of a protocol involves delays and timeouts on transitions by setting time constraints on actions to be executed. Timestamps are not necessary for using our method, but we take the opportunity to show how to tackle protocols with timestamps as well.

Our experimental results show that the known replay attacks on KERBEROS and Neumann Stubblebine protocol [14] are found using the classical correspondence property, whereas an attack on TMN [15] and a type flaw attack on Andrew Secure RPC Protocol (ASP) [14] requires verifying our generalised (timed) property. While playing with timing constraints, in addition to finding attacks, we can also identify time dependencies in protocols for which some known attacks can be eliminated. An example of such a protocol is Needham-Schroeder Public-Key (NSPK), where one can find an attack [16] using the correspondence property, but this attack disappears when we introduce timeouts and set them in an appropriate way. Formally, our method consists in translating a Common Language specification of a security protocol, possibly with timestamps, to one of the higher-level language IL\(^2\) [17], and then again translating automatically the specification (obtained) in IL to a timed automaton without integer variables\(^3\).

The rest of the paper is organized as follows. Section 2 introduces security protocols and the Dolev-Yao model of the Intruder. In Section 3 timing aspects of the protocols are discussed. Our timed authentication property is defined in Section 4. The implementation in IL is described in Section 5. The experimental results and conclusions are presented in Section 6 and Section 7.

2 Modelling Security Protocols and the Intruder

In this section we introduce basic syntax for writing security protocols and discuss the Dolev-Yao model of the Intruder assumed in this paper.

We describe the protocols using a standard notation [14], called Common Syntax (CS, for short) developed for cryptographic protocols [15]. Usually, protocols involve two, three or four roles, which we denote with the capital letters \(A, B\) for the principals, and with \(S\) or \(S'\) for the servers.

Let a protocol \(Q\) be represented by a finite sequence of instructions:

\(^1\) Some preliminary results [13] were presented at CS&P'06.
\(^2\) IL is the acronym for the Intermediate Language (ver. 1.0).
\(^3\) Thanks to that in addition to UppAal we can also use model checkers like Kronos [18] or VerICS [19] accepting timed automata (or their networks) without integer variables.
1. \( X_1 \rightarrow Y_1 : M_1 \)
...
\( n. \) \( X_n \rightarrow Y_n : M_n \)

where \( X_i, Y_i \in \{ A, B, S, S' \} \) and \( X_i \neq Y_i \) for \( 1 \leq i \leq n \), \( Y_i = X_{i+1} \) for \( 1 \leq i < n \), and \( M_i \) is called a message variable\(^4\). The informal meaning of the instruction \( A \rightarrow B : M \) is that a principal of role \( A \) sends a message, which is a value of the variable \( M \), to a principal of role \( B \). Each message variable \( M \) is composed of variables ranging over identifiers representing principals \((PV)\), keys \((KV)\), nonces \((NV)\), and possibly timestamps \((TV)\), and their lifetimes \((LV)\).

Formally, the message variables are generated by the following grammar:

\[
\text{Message} ::= \text{Component} \times \text{Component}^* \\
\text{Component} ::= \text{Cipher} \mid \text{Atom} \\
\text{Cipher} ::= \{ \text{Component}^* \} \hspace{1pt} K \\
\text{Atom} ::= P \mid N \mid K \mid T \mid L,
\]

where \( P \in PV, N \in NV, K \in KV, T \in TV, \) and \( L \in LV \).

For \( X, Y \in \{ A, B, S, S' \} \), \( K_{XY} \) is a key variable of \( X \) and \( Y \), \( N_X \) is a nonce variable of \( X \), \( T_X \) is a timestamp variable of \( X \), and \( L_X \) is a lifetime variable of \( X \). Moreover, by \( R \) we denote a random number variable. A message \( M \) can be encrypted with a key \( K \), denoted by \( \{ M \} K \). For example \( \{ T_X, N_X \} K_{XY} \) is a message variable containing the timestamp variable \( T_X \) and the nonce variable \( N_X \) encrypted with the key variable \( K_{XY} \).

The keys shared between an agent and a server (e.g., \( K_{AS}, K_{BS} \)) or between two agents (e.g., \( K_{AB} \)) are considered. The nonces represent random non-predictable numbers that are declared to be used only once by a concrete agent. The timestamps are unique identifiers whose values are provided by the (local) clock of its issuing entity and determine the time when a given message is generated. A value related to a timestamp is a lifetime defining how long since the timestamp creation it is acceptable to use each of the components of the messages the timestamp relates to. Next, we give an example of the protocol which is considered in the following sections.

**Example 1.** The protocol ASP [20] is the following sequence of four instructions:

1. \( A \rightarrow B : M_1 = A, \{ N_A \} K_{AB} \);
2. \( B \rightarrow A : M_2 = \{ N_A + 1, N_B \} K_{AB} \);
3. \( A \rightarrow B : M_3 = \{ N_B + 1 \} K_{AB} \);
4. \( B \rightarrow A : M_4 = \{ K'_{AB}, N'_B \} K_{AB} \).

In the first message \( A \) (Initiator) sends a nonce \( N_A \), which \( B \) (Responder) increments and returns as the second message together with his nonce \( N_B \). If \( A \) is satisfied with the value returned, then he returns \( B \)'s nonce incremented by 1. Then, \( B \) receives and checks the third message and if it is matched, then he sends a new session key to \( A \) together with a new value \( N'_B \) to be used in a subsequent communication.

\(^4\) We will frequently refer to variables via their names like message, principal, nonce, key, and timestamp if this does not lead to confusion.
A concrete message consists of components which are built of atomic cryptographic primitives that are elements of the following finite sets of identifiers:

\[ P = \{s, a, b, c, \iota, \ldots \} \] — principal, also called agents or participants, \( SK = \{k_a, k_b, k_{ab}, \ldots \} \) — symmetric keys, \( AK = \{k_a, k_b, k_a^{-1}, k_b^{-1}, \ldots \} \) — asymmetric keys, let \( K = SK \cup AK \) be the set of all the keys, \( N = \{n_a, n_b, n'_a, n'_b, \ldots \} \) — nonces, \( T = \{t_a, t_b, t_s, \ldots \} \) — timestamps, \( L = \{l, l', \ldots \} \) — lifetimes, \( R = \{r_a, r_b, \ldots \} \) — random numbers. \(^5\)

By a protocol run we mean a finite sequence of instructions resulting from a fixed number of possibly parallel sessions of the protocol. In a session of the protocol the variables are instantiated by concrete identifiers (names) of the principals and of the message elements. By a component’s type we understand the sequence of the types of its atoms together with the braces. For example the type of the component \( c = \{k, p, l \}k \) is \( \{K, P, L\}K \) and the type of the component \( c = \{\{p, l\}\}k \) is \( \{\{P, L\}\}K \). Notice that for each protocol there is a finite set of the types of components that can be used for composing all the messages in this protocol. These types of components are denoted by \( C_1, C_2, \ldots, C_n \).

### 2.1 Modelling the Intruder

Notice that the set \( P \) contains the identifier \( \iota \) used for denoting the Intruder. The Dolev-Yao model of the Intruder is assumed \(^2\). One of the capabilities (besides exploiting intercepted messages) is that the Intruder can impersonate each agent executing the protocol, so he can play each of the roles of the protocol. Even thought the Intruder has got his own keys, nonces etc., he can also try to use all the information he is receiving in the protocol run as his own (e.g., nonces). When the Intruder impersonates an agent \( x (x \in P\setminus\{\iota\}) \), we denote this by \( \iota(x) \).

**Intruder: Knowledge, Processes and Actions.** Typically, the following Dolev-Yao capabilities of the Intruder are assumed \(^2\):

- He eavesdrops every letter passing through the Net, duplicates it and stores a copy in his local memory \( IK \), called database or Intruder knowledge.
- He can affect intercepted letters by changing their headers (rerouting), replaying a letter, replacing an original letter with a modified or a new one, and finally send it or just delete intercepted letter.
- He can create any brand new letter built upon his knowledge.
- He can derive new facts basing only on his actual knowledge. This means he can decrypt and decompose any message only if he possesses a proper key.
- He can also exploit the malicious agent, who can initialize some sessions of the protocol. The Intruder can use his knowledge.

\(^5\) We provide a variant of our implementation where the sets \( N \) and \( K \) are unified into the set \( R \).
An Optimized Intruder. The crucial point in an efficient modelling of the Intruder is to restrict his capabilities as much as possible, but without eliminating any of the possible attacks. One obvious restriction is that the Intruder generates messages of such a type and pattern, which can be potentially accepted by some agent. This means that the Intruder should be only sending messages, which are correct. So, intercepted messages are only modified by replacing some components with other components of the same type and pattern, which can be generated using the Intruder’s knowledge and then send over to the responder. However, when the Intruder acts as an initiator (for example sending the message $M_1$ or $M_3$ in KERBEROS), generating a completely new message can bring him to success. Our restriction on the Intruder’s behaviour follows the rules of the optimized and the lazy Intruder of [21, 2, 13].

3 Timing Aspects

In this section we discuss all the timing aspects we consider in the implementations of security protocols.

Timeout. After sending a message $M_i$, a principal $p \in P$ playing a role $X$, where $X \in \{A, B, S\}$, in a session $j$ of a protocol is waiting for a response message. The maximal period of time the sender is allowed to wait for it is called a timeout we denote as $t_{\text{out},i}(j)$. Then, the next action of the principal $p$ is executed if a response message had been received before the timeout passed. When the timeout is reached, and no response message has arrived, the principal who sends the message $M_i$ in the session $j$ can execute one of the following alternative actions: resending a message (called a retransmission) or starting the same session again (called reset of a session).

When verifying cryptographic protocols with timestamps, it is desirable to check their safety with respect to the timestamps and analyse their relationship with timeouts fixed in the protocol. It is well known that in some types of protocols it is not possible to set a timeout for each message. In such a case a timestamp indicating the time of creating a message is useful. Then, the receiver can decide whether the message is fresh or not, depending on the value of the timestamp and its lifetime $L$.

Time of creating a message $M_i$. For each principal $p \in P$ we set a time of performing each mathematical operation like encryption - $\tau_{\text{enc}}(p)$, decryption - $\tau_{\text{dec}}(p)$, and generating random values - $\tau_{\text{gen}}(p)$. This way we can compute the time $\tau_{M_i}(p)$ of creating any message $M_i$ by the principal $p$.

$$
\tau_{M_i}(p) = (n_1 \ast \tau_{\text{dec}}(p)) + (n_2 \ast \tau_{\text{enc}}(p)) + (n_3 \ast \tau_{\text{gen}}(p)),
$$

---

6 He can initiate a session either from the beginning or from a middle of the protocol.
7 Note, however, that there are messages without a corresponding response. This depends on the construction of a protocol and there are protocols without any response message (e.g., WMF protocol).
where \( n_2(n_3) \) is the number of the operations of encryption (generating random values, resp.) to be performed in order to generate a message \( M_i \), whereas \( n_1 \) is the number of the operations of decryption after receiving \( M_{i-1} \) (of the previous step of the protocol). The Intruder’s time of composing a message using components he has intercepted in previous steps of the protocol is denoted by \( \tau_{\text{com},M_i}(\iota) \), but when it is assumed to be equal for all the messages it is denoted with \( \tau_{\text{com}}(\iota) \).

**Delay.** The delay \( \tau_d \) represents the time of message transmission from sender to receiver. As we have said before, all the messages are passing through the Intruder’s part of the model, so how long it takes from sending to receiving a message depends on the value of a delay and the actions the Intruder performs. We assume that \( \tau_d \in (\tau_{\text{d,min}}, \tau_{\text{d,i}}) \) and the \( \tau_{\text{d,min}} \) is a minimal delay of the network we have set before the run of the protocol. In order to simplify the formalism we assume that the minimal delay is the same for all the message transmissions \(^8\). The value of \( \tau_{\text{d,i}} \) represents the maximal delay for a step \( i \) of the protocol (transferring a message \( M_i \)) and it is computed with respect to the timeouts (lifetimes) which cover the sequences of actions this step belongs to, the times of composing the messages that are sent in these sequences of actions, and \( \tau_{\text{d,min}} \). The details of computing \( \tau_{\text{d,i}} \) are given later in this section.

**Time of a session.** This is an expected time of performing all the steps of a protocol including the time of a transmission of messages through the net. It is specified as a time interval \( \langle T_{\text{min}}, T_{\text{max}} \rangle \), where \( T_{\text{min}}(T_{\text{max}}) \) is the minimal (maximal, resp.) time of an execution of all the actions allowing for all the possible delays. Below, we give a definition of the graph of the message flow in a protocol \( Q \) of \( n \) instructions and its one session \( R \):

\[
\begin{align*}
1. & \quad X_1 \rightarrow Y_1 : M_1 \\
\vdots \\
n. & \quad X_n \rightarrow Y_n : M_n \\
1. & \quad p_1 \rightarrow q_1 : m_1 \\
\vdots \\
n. & \quad p_n \rightarrow q_n : m_n
\end{align*}
\]

where \( p_i \) (\( q_i \)) is the principal sending (receiving, resp.) \( m_i \), for \( 1 \leq i \leq n \).

Let \( \text{tstamp}(M_i) \) be the timestamp variable (if exists) of the message \( M_i \), and let \( l_i \) be the value of the lifetime granted for \( \text{tstamp}(M_i) \) in \( m_i \). For example, if \( M_i = \{X, L_X\}_{K_{Xy}}, \{T_X, N_X\}_{K_{Xy}} \), then \( \text{tstamp}(M_i) = T_X \) and if \( m_i = \{x, l_x\}_{K_{xy}}, \{t_x, n_x\}_{K_{xy}} \), then \( l_i = l_x \).

In the graph defined below (Def. 1), the vertices represent control points of the message flow in the protocol such that the vertex \( 2i-1 \) corresponds to sending the message \( i \), whereas the vertex \( 2i \) to receiving this message, for \( i \in \{1, \ldots, n\} \).

There are four types of edges in the graph denoted with \( E_j \) for \( j \in \{1, 2, 3, 4\} \). Each edge \( (2i-1, 2i) \) of \( E_1 \) represents a transfer of the message \( i \) through the

---

\(^8\) We usually assume that \( \tau_{\text{d,min}} \) is close to 0.
net. This edge is labelled with the minimal and maximal time of this transfer. Each edge \((2i, 2i + 1)\) of \(E_2\) represents a composition of the message \(i\) and it is labelled with the time of this composition. The edges of \(E_3\) (\(E_4\)) correspond to timeouts (lifetimes, resp.). The intuition behind the edge \((i, j)\) of \(E_3\) or \(E_4\) is that it covers all the edges of \(E_1\) and \(E_2\) between the vertices from the set \(\{i, \ldots, j\}\).

**Definition 1.** The four-tuple \(G = (V, l_V, E, l_E)\) is a weighted labelled acyclic graph of the message flow in the session \(R\), where

- \(V = \{1, 2, \ldots, 2n\}\) is the set of the vertices,
- \(l_V : V \rightarrow \{+,-\} \times P\) is a vertex labelling function such that
  \[l_V(2i - 1) = -p_i\ (p_i\ is the principal sending the message \(i\))\) and\[l_V(2i) = +p_i\ (q_i\ is the principal receiving the message \(i\))\) for \(1 \leq i \leq n\),
- \(E \subseteq V \times V\) is a set of the directed labelled edges, where \(E = E_1 \cup E_2 \cup E_3 \cup E_4\), \(E_2 \cap (E_3 \cup E_4) = \emptyset\), \(E_1 \cap E_3 = \emptyset\), and
  - \(E_1 = \{(2i - 1, 2i)\mid 1 \leq i \leq n\}\); \(E_1\) contains the edges representing the minimal and the maximal times of transferring messages through the net.
  - \(E_2 = \{(2i - 2, 2i - 1)\mid 1 \leq i \leq n\}\); \(E_2\) contains the edges representing the times of composing the messages.
  - \(E_3 = \{(2i - 1, 2j)\mid l_V(2j) = +p_i, 1 \leq i < j \leq n, \text{ where } j \text{ is the smallest such a number for } i\}\); \(E_3\) contains the edges representing the timeouts between sending the message \(m_i\) and receiving the response message \(m_j\) by \(p_i\).
  - \(E_4 = \{(2i - 1, 2j)\mid tstamp(M_i) = tstamp(M_j), \ 1 \leq i < j \leq n\}\); \(E_4\) contains the edges representing the lifetime \(l_i\) of the timestamp sent in the message \(m_i\) and then received in each message \(m_j\).
- \(l_E : E \rightarrow N \cup (N \times N) \cup (N \times N \times N) \cup (\{\tau_M\} \times N \times P)\) is an edge labelling function s.t.
  an edge \((2i - 1, 2i)\in E_1\ \setminus\ E_4\) is labelled with \(\langle \tau_{d,\text{min}}, \tau_{d,i}\rangle\) for \(1 \leq i \leq n\), \(\tau_{d,\text{min}}\) is fixed, whereas the value of \(\tau_{d,i}\) is computed if there is a lifetime or a timeout covering\(^9\) the edge \((2i - 1, 2i)\), otherwise is equal to \(\infty\), an edge \((2i - 1, 2i)\in E_1\ \cap\ E_4\) is labelled with \(\langle \tau_{d,\text{min}}, \tau_{d,i}\rangle\), \(l_i\) for \(1 \leq i \leq n\), an edge \((2i - 2, 2i - 1)\in E_2\) is labelled with \(\tau_M(p_i)\) for \(1 < i \leq n\), an edge \((2i - 1, 2j)\in E_3\ \cap\ E_4\) is labelled with \(\min\{t_{\text{out},i}, l_i\}\), an edge \((2i - 1, 2j)\in E_3\ \setminus\ E_4\) is labelled with \(t_{\text{out},i}\), and an edge \((2i - 1, 2j)\in E_4\ \setminus\ E_3\) is labelled with \(l_i\), for \(1 \leq i < j \leq n\), where \(\tau_{d,i}\) is calculated as follows.
  For each \(i \in \{1, \ldots, n\}\) define:
  - a set of the timeout- or lifetime-edges covering the edge \((2i - 1, 2i)\) of transferring the message \(m_i\),
    \[\mathcal{LT}_i = \{(k, l) \in E_3 \cup E_4 \mid k \leq 2i - 1 \land l \geq 2i\}\] (1)
  \[\text{This means that there is an edge } (k, l) \in E_3 \cup E_4 \text{ s.t. } k \leq 2i - 1 \text{ and } 2i \leq l.\]
the minimal time of executing all the operations covered by the timeout (lifetime) $l_E((k, l))$, where $p_i$ is the principal composing the message $M_i$.

\[ tca(k, l) = \sum_{i=(k+1)/2+1}^{l/2} (\tau_{M_i}(p_i) + \tau_{d,min}) + \tau_{d,min} \] (2)

For each $(k, l) \in L_T$:

\[ \text{lt}_{(k,l)} = \max\{\tau_{d,min}, l_E((k, l)) - tca(k, l) + \tau_{d,min}\} \] (3)

For each $i \in \{1, \ldots, n\}$:

\[ \tau_{d,i} = \min\{\text{lt}_{(k,l)} | (k, l) \in L_T\} \] (4)

Notice that we do not consider the time of composing the first and decomposing the last message. It is sufficient to measure the time between sending the first message and receiving the last one. Moreover, if a timeout or a lifetime is incorrectly set, i.e., it does not allow for sending a message, say $M_i$, after the time delay $\tau_{d,min}$, then we set $\tau_{d,i}$ to be equal to $\tau_{d,min}$ as well. But, later when computing the minimal and maximal possible time of executing one session ($T_{min}$ and $T_{max}$), we use this information for setting both the times to 0.

Notice that $\tau_{d,i}$ (in the formula (4)) is computed as the minimum over all the times allowed by the timeout- and lifetimes-edges covering the transfer of the message $m_i$. This is clearly correct as $\tau_{d,i}$ can never exceed the minimal value of a timeout (lifetime)-edge covering the transfer of $m_i$.

In [10] the four typical message flow schemas for cryptographic protocols are presented. Their taxonomy is focused on timeout dependencies, so it does not deal with timestamps in the message flow schemas. As our goal is to apply the method to all the types of protocols without any restriction on their specifications or contents of the messages we introduce lifetime parameters that were not specified in [10]. Next, for each protocol to be verified, we build the corresponding graph according to Definition 1. Then, $T_{min}$ and $T_{max}$ are computed. $T_{min}$ is taken as the weight of the minimal path in this graph, which corresponds to a sequential execution of all the actions within their minimal transmission time. However, if $l_E((k, l)) - tca(k, l) < 0$ for some $i \in \{1, \ldots, n\}$ and $(k, l) \in L_T$, then we set $T_{min} := 0$.

$T_{max}$ is the maximal possible time of executing one session of the protocol taking into account all the timeouts and lifetimes. The idea behind computing $T_{max}$ is as follows. For each transition $(2j - 1, 2j)$ of $E_1$ we find its maximal possible delay, denoted by $\tau_{max,j}$, provided all the preceding transfers of the messages from $m_1$ to $m_{j-1}$ have taken their maximal possible delays. The value of $\tau_{max,j}$ is computed similarly to $\tau_{d,j}$, but for all the transitions $(2i - 1, 2i)$ of $E_1$ prior to $(2j - 1, 2j)$, we take the values of $\tau_{max,i}$ instead of $\tau_{d,min}$. This requires to slightly modify the formulas $L_T((k,l))$ into $L_T'(k,l)$. Then, $T_{max}$ is defined as the sum of $\tau_{max,1}$ and all $\tau_{max,j}$ and $\tau_{M_i}(p_i)$ for $j \in \{2, \ldots, n\}$. But, if $\tau_{max,j} < \tau_{d,min}$ for some $j \in \{1, \ldots, n\}$, then we set $T_{max} := 0$. Below, we formalize the above algorithm in the following inductive definition.
We start with setting $\tau_{\text{max},1} := \tau_{d,1}$.
Next, for $j \in \{2, \ldots, n\}$ we will use the following definitions to calculate $\tau_{\text{max},j}$.
For $(k, l) \in \mathbf{LT}_j$:

$$l_{(k,l)} = l_{E((k,l))} - \sum_{\substack{i=(k+1)/2+1}}^{i=k/2} (\tau_{M_i}(p_i) + \tau_{i-1})$$

(5)

where $\tau_i = \begin{cases} 
\tau_{\text{max},i}, & \text{for } i \leq j, \\
\tau_{d,\min}, & \text{otherwise}
\end{cases}$

Finally:

$$\tau_{\text{max},j} = \min\{l_{(k,l)} \mid (k, l) \in \mathbf{LT}_j\},$$

(6)

$$T_{\text{min}} := \begin{cases} 
\tau_{d,\min} + \sum_{j=2}^{n} (\tau_{M_j}(p_j) + \tau_{d,\min}), & (\forall 1 \leq j \leq n) \tau_{\text{max},j} \geq \tau_{d,\min} \\
0, & \text{otherwise}
\end{cases}$$

(7)

$$T_{\text{max}} := \begin{cases} 
\tau_{\text{max},1} + \sum_{j=2}^{n} (\tau_{\text{max},j} + \tau_{M_j}(p_j)), & (\forall 1 \leq j \leq n) \tau_{\text{max},j} \geq \tau_{d,\min} \\
0, & \text{otherwise}
\end{cases}$$

(8)

**Example 2.** The message flow graph for ASP is shown in Fig. 1. The graph contains 8 vertices as there are 4 instructions in the protocol. There are no edges corresponding to lifetimes as timestamps are not used in the protocol. Notice that the timeout $t_{\text{out},1}$, labelling the edge (1, 4), covers the three edges: (1, 2) - representing the sending of $M_1$ from $a$ to $b$, (2, 3) - composing $M_2$ by $b$, and (3, 4) - the sending of $M_2$ from $b$ to $a$. The timeout $t_{\text{out},2}$, labelling the edge (3, 6), covers the three edges: (3, 4) - representing the sending of $M_2$, (4, 5) - composing $M_3$ by $a$, and (5, 6) - the sending of $M_3$ from $a$ to $b$. Finally, the timeout $t_{\text{out},3}$, labelling the edge (5, 8), covers the three edges: (5, 6) - representing the sending of $M_3$, (6, 7) - composing $M_4$ by $b$, and (7, 8) - the sending of $M_4$ from $b$ to $a$.

![Fig. 1. The message flow graph for ASP](image)

Assuming proper timeouts settings, the simplified formulas for $T_{\text{min}}$ and $T_{\text{max}}$ derived from (3)-(8) for the graph presented in Fig. 1 are as follows: $T_{\text{min}} = \tau_{M_2}(b) + \tau_{M_3}(a) + \tau_{M_4}(b) + 4 * \tau_{d,\min}$. $T_{\text{max}} = t_{\text{out},1} + \tau_{M_3}(a) + t_{\text{out},3}$. If at least one of the timeouts is too short, then $T_{\text{min}}$ and $T_{\text{max}}$ are set to 0.
4 Authentication Property

We start with giving a definition of an attack [22] on an authentication protocol. We do not consider a passive attack [22] which is only based on monitoring the communication channel. Then, we formulate an extension of the authentication property for which we show that it implies an attack on a protocol.

Definition 2. An active attack is one where the adversary attempts to delete, add, or in some other way alter the transmission on the channel. The actions of the Intruder leading to an active attack are called active actions.

Typically security protocols are verified against the (entity) authentication property [23, 24], which can be formulated as the following correspondence property:

If a principal $x$ has finished $N$ sessions with a principal $y$ in a protocol run, then the principal $x$ must have started at least $N$ sessions with the principal $y$. When the above relation is symmetric we capture the mutual entity authentication. In this paper we suggest the timed authentication property, which consists of the above property and the following extension:

If a session of a protocol run between two principals $x$ and $y$ started at the time $T_{\text{start}}$, then it can be finished only within the time interval $(T_{\text{start}} + T_{\text{min}}, T_{\text{start}} + T_{\text{max}})$.

Notice that if a protocol ends before the specified $T_{\text{min}}$, then this may be a result of omitting at least one of the instructions or/and performing at least one of the instructions faster than it was expected. But, if a protocol ends after the specified $T_{\text{max}}$, then this may be caused by some additional actions, a modification of the timestamp, or an execution of at least one of the instructions slower than it was assumed. As the honest participants are not able to change the way they execute the protocol, the only possibility for an unexpected speed up or slow down in the execution of the protocol is due to an active action of the Intruder. So, if a session ends before the minimal time or after the maximal time specified causing that the timed authentication property does not hold, then there is an active attack in the meaning of Definition 2.

It is worth mentioning that the timed authentication property allows for detecting attacks even when the standard correspondence property is satisfied. To this aim observe that the correspondence property holds for a protocol whose session has started but has not terminated, but it fails only if a session has been terminated but it had not been started. Notice that there are attacks in which the Intruder impersonating the Responder leads to a successful end of a session, so that the correspondence property holds. By playing with the times of executing Intruder’s actions we can show that there is a session which is finished before its minimal time $T_{\text{min}}$, which means that the timed authentication property fails.

5 Implementation

The roles $A$ and $B$ of the protocol $Q$ are translated to disjoint parametrized IL processes $\text{Proc}_{X(p)}(j)$, where $p \in \mathcal{P}$ is the agent playing the role $X \in \{A, B\}$,
and \( j \in N \) is the number of a session. Notice that it is possible to run more than one\(^{10}\) instance of each role by each agent of \( \mathcal{P} \). The role \( S (S') \) is translated to the process \( \text{Proc}_S (\text{Proc}_{S'}, \text{resp.}) \), which is not parametrized as it is possible to specify only one instance of each of them for all the possible sessions of the protocol. Each process \( \text{Proc}_X(p)(j) \) and \( \text{Proc}_S \) is specified by a set of local variables \( V_p = \{ N, K, L, T \} \), which are called protocol variables, a set of locations \( Q_p \) of the internal states of \( p \) which represents the local stage of the protocol execution, the initial location \( q_0^p \) in which all the necessary initial values of the protocol variables are set, and a set of (timed) transitions \( T_p(j) \) which correspond to the steps executed by the principal \( p \) playing his role in the session \( j \).

The time interval specified for the transition indicates the period of time in which the transition is allowed to be fired since its enabling. The processes running in parallel are synchronized through a set of buffers \( \mathcal{B} \) and global variables \( V_G \). Communication between the participants is realized via two sets of buffers, called \( \mathcal{B}^{\text{IN}} \) and \( \mathcal{B}^{\text{OUT}} \). A sender puts the message \( i \) into the buffer of \( \mathcal{B}_i^{\text{OUT}} \) which is denoted as \( \text{Send}(\mathcal{B}_i^{\text{OUT}}, m) \), whereas a receiver accepts the message \( i \) from the buffer of \( \mathcal{B}_i^{\text{IN}} \) denoted as \( \text{Accept}(\mathcal{B}_i^{\text{IN}}, m) \). The Intruder transfers messages from \( \mathcal{B}^{\text{OUT}} \) to \( \mathcal{B}^{\text{IN}} \), so that he can easily control the flow and contents of the messages.

All the processes of Intruder are operating on a special set \( \mathcal{I} \) of buffers, called Intruder's knowledge. The buffers are used for storing all the messages passing through the net according to the types of components. But, if components corresponding to some cryptographic primitives have not been defined for a verified protocol, then the buffers for them are created additionally. Such an organization allows the Intruder for an easy access to desired data types in order to compose new correct messages. In case of ASP we have the following buffers: \( \mathcal{B}_{C_1}, \mathcal{B}_{C_2}, \) and \( \mathcal{B}_{C_3} \) corresponding to the three types of components, whereas \( \mathcal{B}_K \) (for keys) and \( \mathcal{B}_N \) (for nonces) would be added as there are no components corresponding to the primitives of \( K \) and \( N \). The automated procedures of the Intruder’s behaviour are described in [25]. The process of composing messages is based on two main procedures: \text{ComposeHeader} and \text{ComposeComponent} that use the components stored in \( \mathcal{I} \) in order to compose all the possible correct messages. The decomposing process consists of \text{DecomposeMessage} and \text{DecomposeComponent} procedures. The first one splits messages into components which are then analysed by the second procedure. The plain and encrypted components are stored in \( \mathcal{I} \). If the Intruder happens to be in the possession of the proper key to decrypt a component, then this component is decomposed into subcomponents that are saved in \( \mathcal{I} \).

The timing aspects discussed are modelled by timed transitions. To implement a timeout, for each session \( j \) we introduce the variable \( T_{\text{mr}}(j) \) that indicates the state of a process \( \text{Proc}_X(p)(j) \) (\( p \in \mathcal{P} \) and \( j \in N \)) after the sending of message \( M_i \).\(^{11}\) Moreover, we implement the process \( \text{TmOut}(j) \) which is responsible for managing the value of this variable. When the process \( \text{Proc}_X(p)(j) \) sends

\[10\] We assume that the number of sessions is bounded.

\[11\] Note that only one role in a session \( j \) can send a message \( M_i \).
a message and starts a timer of the timeout by setting $T_{mr_i}(j)$ to 1, the transition $t_1$ in the process $TmOut_i(j)$ gets enabled and fired after the time reaches $t_{out,i}(j)$. This transition changes the value of $T_{mr_i}(j)$ back to 0. The process $Proc_{X_{(p_j)}}(j)$ is able to receive an expected message as long as $T_{mr_i}(j)$ equals to 1. When this condition fails to hold, the process returns to the state Init and generates new initial values like keys, nonces, and other random variables.

To implement a timestamp, the global variable $T_i$ is introduced and initially set to 0. This variable represents the state of the timestamp $i$ which may be either ‘valid’ or ‘not valid’. The process $Timestamp_i$ is responsible for managing the value of $T_i$. When timestamp $i$ is generated, $T_i$ is set to $i$ and the transition from Init to NotValid gets enabled (see Fig. 2). Then, after exactly $L_i$ units of time the transition from Init to NotValid is forced, and it sets back $T_i$ to 0. When the agent receives a message, he tests the corresponding timestamp by reading its identifier from the buffer and then checking the value of the variable $T_i$ of the timestamp. If the value is equal to $i$ and the message is found as matched, then it is accepted. Otherwise, the message is not accepted. An outdated timestamp cannot be validated anymore.

![Fig. 2. The automata models of the processes TmOut_i and Timestamp_i](image)

A set of processes $Property_{xy}(j)$ in IL, one for each session $j$, is used to model the timed authentication property. The process $Property$ for one session is shown in Fig. 3. There are three possible locations (in addition to Init) each process $Property$ can be in: AUTH, ERR, and CERR. The state AUTH is reached when the last message in a protocol is accepted within the correct time interval. The transition to the state ERR is caused when either the protocol ends too early or too late, whereas the state CERR is reached when the correspondence property is not satisfied. The pair of global variables start and end is used to synchronize the participants’ processes with the process $Property$. When the protocol starts, the variable start is set to 1, the transition from Init to AUTH, and both the transitions from Init to ERR become enabled. Each of them is fireable within the time interval specified, but it is fired only when the variable end has changed its value to 1 (during this time interval).

To implement the (basic) correspondence property, for each pair $(x, y)$ of principals, where $x, y \in P \setminus \{server\}$, we use the global variable $V_{AUTH}(x, y)$ whose value is equal to 0 at the beginning of a protocol run. Then, at the beginning of each new session for $x$ and $y$ the variable $V_{AUTH}(x, y)$ is incremented by 1, and at the end of each session the variable $V_{AUTH}(x, y)$ is decremented by 1. As long as the value of $V_{AUTH}(x, y)$ is non-negative the (basic) correspondence property holds.
6 Experimental Results

Our experiments are performed in order to show how values of time constraints influence safety of protocols as well as how they can be used to find or eliminate flaws in these protocols. We used the following two tools: Verics [19] and Kronos [18]. Both of them have been fed with product automata generated by Verics. For the protocols ASP, KERBEROS, NSP, and TMN the results are displayed in Fig. 4, where reachability of the states AUTH, ERR, and C_ERR depends on values of the timed parameters.

In order to find an attack we set the time for each of the Intruder's actions to belong to the time interval \([0, T_{max})\). We test each protocol against the property shown in Fig. 3. When the state ERR (or C_ERR) of the property is reached, we look through the path leading to this state in the product automaton to get
the time intervals for time parameters for which the attack has been discovered. This may result in finding an execution of the protocol, which is finished before the specified minimal time of a session. This way flaws are found in ASP [14] and TMN [15] as for both the protocols the state ERR is reachable (see Fig. 4 (a),(b)). We have also checked the relationships between the lifetime of a timestamp and \( \tau_{\text{com}}(t) \)\(^{12}\). In Fig. 4 (a) the relationship between \( \tau_{\text{com}}(t) \) and the time of finishing ASP is presented. If composing a new message by the Intruder takes less time than composing a message of type \( M_3 \) by the principal \( b \), then the protocol is finished before \( T_{\text{min}} \).

For both the protocols NSP and KERBEROS if \( \tau_{\text{com}}(t) < l \), then the correspondence property does not hold (see Fig. 4 (c),(d)). Moreover, the results for NSP show that when \( l \) does not allow for finishing the protocol (it is too short) but the time \( \tau_{\text{com}}(t) \) is shorter than \( l \), the protocol is finished before the time specified and the state ERR becomes reachable. The minimal value of the lifetime of a timestamp allowing to finish the protocol is denoted as \( l_{\text{min}} \).

For KERBEROS the state ERR becomes reachable as well if additionally \( \tau_{\text{com}}(t) < \tau_{\text{com}}^1(t) \) or \( \tau_{\text{com}}(t) > \tau_{\text{com}}^2(t) \), where \( \tau_{\text{com}}^1(t) = T_{\text{min}} - \tau_{M_4}(b) \) and \( \tau_{\text{com}}^2(t) = T_{\text{max}} - \tau_{M_4}(b) \).

The protocol WMF\(^{13}\) is another example for which we show that the timed property can be unsatisfied while the correspondence property still holds. This is the case when the time of a session is exceeded and is not limited even by a timestamp. So, the state ERR of the property is reached, for the following time constraint \( \tau_{\text{com}}(t) + \tau_{d,1} + \tau_{d,2} < l_1 \), where \( l_1 \) is the lifetime set for \( m_1 \).

While playing with timing constraints, in addition to finding attacks, we can also identify time dependencies for which some known attacks in protocols can be eliminated. Consider the protocol NSPK, where one can find an attack [16] using the correspondence property. This attack can be eliminated when the timeouts are set in an appropriate way. The condition is as follows: \( (\tau_{d,i} - \tau_{d,\min}) < \tau(M_i)(t) \) for \( i = \{1,3\} \). Recall that \( \tau_{d,i} \) depends on the corresponding timeouts.

7 Conclusions

In this paper we offered a novel methodology for verifying correctness of (timed) security protocols whose actions are parametrized with time. Our main contribution consists in generalizing the correspondence property so that several attacks can be discovered when some time constraints are not satisfied. The verified model of a protocol is obtained via a translation from CL to the high level specification language IL, and then again to timed automata. We have introduced several time parameters into the model and discussed their meaning for the protocols considered. As some of these parameters depend on principal’s abilities (e.g., the times of cryptographic operations) we based our methodology on one session of a protocol run rather than on a protocol itself.

\(^{12}\) This is the Intruder’s time of composing a message using components he has intercepted in previous steps of the protocol.

\(^{13}\) 1. \( A \rightarrow S : A, \{T_A, B, K_{AB}\}_{K_{AS}} \) 2. \( S \rightarrow B : \{T_S, A, K_{AB}\}_{K_{BS}} \).
Our timed correspondence property is specified as a timed IL process where the insecure states are constrained by time intervals. We showed how to build the message flow graph for a protocol session with respect to the timed parameters discussed in order to calculate the essential values used in the property. The definition of such a graph is applicable to most of the protocols [14, 15].

The verification based on checking the reachability of the error states for the timed correspondence property leads us to find attacks that cannot be discovered using the standard correspondence property. The experimental results for the protocols ASP and TMN confirm that the timed property is not satisfied as both the protocols can be finished before the specified minimal time of a session.

Using this property it is also possible to find an insecure relationship between timed parameters, which may show weak points of the protocol. We showed a subtle relationship between lifetimes of the timestamps and timeouts in NSP and KERBEROS, while for KERBEROS these parameters can be tuned carefully to avoid an unexpected finish of the protocol.

Moreover, time constrains are used to find time dependencies for which some known attacks in protocols can be eliminated.

References