Formal Analysis of Time-Dependent Cryptographic Protocols in Real-Time Maude

Peter Csaba Ölveczky and Martin Grimeland
Department of Informatics, University of Oslo, Norway
{peterol, marting}@ifi.uio.no

Abstract

This paper investigates the suitability of applying the general-purpose Real-Time Maude tool to the formal specification and model checking analysis of time-dependent cryptographic protocols. We restrict the intruders so that they become non-Zeno*, and propose a complete analysis method for finding attacks that are reachable from the initial state. Our method has been used on the benchmark* wide-mouthed frog *(WMF) and* Kerberos *protocols, on which we can find all the well known flaws in short time. We use the WMF protocol to illustrate formal specification and the use of our method to analyze timed authentication properties.*

1 Introduction

Formal analysis based on model checking has proved very useful to find subtle errors in cryptographic protocols. Consequently, there are many cryptographic protocol analysis tools based on formal methods (e.g., [9, 4, 10]). Such tools typically do not support modeling and analyzing real-time aspects, and are therefore not well suited to model *time-sensitive* cryptographic protocols such as Kerberos [7], wide-mouthed frog [1], and TESLA [18].

In this paper, we investigate the suitability of using Real-Time Maude [15, 14] for the formal specification and analysis of time-sensitive cryptographic protocols. Real-Time Maude is a high-performance tool that extends the rewriting logic-based Maude system [2] to support the formal specification and analysis of object-based real-time systems. Real-Time Maude emphasizes ease and expressiveness of specification, and provides a spectrum of analysis methods, including simulation through timed rewriting, time-bounded temporal logic model checking, and time-bounded and unbounded explicit-state search for reachability analysis. Real-Time Maude has proved useful for analyzing a wide range of advanced real-time systems, including communication protocols [16] and scheduling [13] and wireless sensor network algorithms [17].

The first problem we address is the extension of the "Dolev-Yao" intruder model to the timed setting. To justify the common and useful abstraction that state transitions take zero time, a real-time system should not allow "Zeno" behaviors in which an infinite number of such transitions takes place in a finite time interval. A "naive" Dolev-Yao intruder, however, can create an infinite number of fake messages in zero time. We therefore restrict the intruder so that it can perform at most N malicious actions per time unit. With such a non-Zeno intruder, any *time-bounded* search and LTL model checking command is guaranteed to terminate. We use this fact to define a mechanizable analysis strategy, where we increase the intruder capability N and/or the time bound in the analysis command after each search and model checking command in which no attack is found, and then execute the command again, until the attack is found (or we run out of memory or patience). The advantages of this method are that:

1. *Completeness* of the analysis is regained, in the sense that any flawed behavior that can take place from the initial state *without* time or intruder limits will eventually be discovered.
2. Time-bounded LTL model checking can be used to analyze properties beyond reachability.
3. Search analysis becomes feasible with this method. Indeed, none of the search analyses in this paper could find the attacks before memory was exhausted when we used a Zeno intruder.
4. The method can be fully automated.

We have tested our method on the benchmark *wide-mouthed frog* (WMF) protocol [1] and on the much larger Kerberos protocol [7], and found all known flaws in typically less time than in other analyses.

In this paper, we illustrate our methodology with the formal specification and analysis of WMF. The

1-4244-0910-1/07/$20.00 ©2007 IEEE

WMF protocol was proposed by Burrows as a simple protocol for ensuring *timed authentication*, and has become something of a benchmark time-sensitive cryptographic protocol. WMF has previously been analyzed by model checkers for crypto-analysis [3, 9]. Without any ingenuity in the definition of initial states, our tool could automatically discover all known faults within minutes. Indeed, for the two faults mentioned in [3], our method found the flaws faster than the backwards symbolic reachability analysis reported in [3].

Another advantage of our approach compared to using domain-specific crypto-analysis tools is that we can include the cryptographic protocol as a part of a larger system, or in combination with other kinds of protocols, and that we can model much larger and complex protocols for which Maude has proved useful in the untimed setting [5]. Furthermore, our tool allows simulation for prototyping purposes, and provides expressive analysis features to allow model checking of a wide range of properties beyond just reachability properties.

2 Real-Time Maude

A Real-Time Maude *module* specifies a *real-time rewrite theory* (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [11] theory with Σ a signature[1] and E a set of conditional equations. The theory (Σ, E) specifies the system's state space as an algebraic data type, and must contain a specification of a sort `Time` modeling the time domain.

- IR is a set of *labeled conditional instantaneous rewrite rules* specifying the system's *instantaneous* local transitions, each of which is written `crl [`l`] :` t `=>` t' `if` *cond*, where l is a *label*. Such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t', *provided* the condition holds. The rules are applied *modulo* the equations E.[2]

- TR is a set of *tick (rewrite) rules*, having syntax

  `crl [`l`] : {`t`} => {`t'`} in time` τ `if` *cond* `.`

  that model time elapse. `{_}` is a built-in constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

[1] That is, Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols* (or *operators*).

[2] E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

The initial states must be ground terms of sort `GlobalSystem` and must be reducible to terms of the form `{`t`}` using the equations in the specifications.

A *class* declaration

`class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n `.`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term $\texttt{<}\ O : C \mid att_1 : val_1, ..., att_n : val_n\ \texttt{>}$ where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the values of the attributes att_1 to att_n. In a concurrent object-oriented system, the state, which is usually called a *configuration*, is a term of the built-in sort `Configuration`. It has typically the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude. The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

```
rl [l] : m(O,w)
         < O : C | a1 : x, a2 : O', a3 : z >
        =>
         < O : C | a1 : x + w, a2 : O', a3 : z >
         dly(m'(O'),x) .
```

defines a family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions have the effect of altering the attribute `a1` of the object `O` and of sending a new message `m'(O')` *with (least) delay* `x` (see [15]). "Irrelevant" attributes (such as `a3`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule.

Timed modules are *executable* under reasonable assumptions, and Real-Time Maude provides a spectrum of analysis capabilities. Real-Time Maude's *timed "fair" rewrite* command simulates *one* behavior of the system *up to a certain duration*. Real-Time Maude's timed *search* command uses a breadth-first strategy to search for states that are reachable from an initial state t within time τ, match a *search pattern*, and satisfy a *search condition*. The command that searches for *one* such state is written

`(tsearch [1]` t `=>*` *pattern* `such that` *cond* `in time <=` τ `.)`

For analyzing more advanced properties, Real-Time Maude extends Maude's *linear temporal logic model checker* [2] to check whether each behavior "up to a certain time" satisfies a temporal logic formula [15]. When the specification is non-Zeno, only a finite number of states can be reached from an initial state within a given duration, ensuring that time-bounded model checking terminates.

3 Analysis with Non-Zeno Intruders

A cryptographic protocol is usually analyzed in the presence of "Dolev-Yao" intruders. Such an intruder can participate in ordinary protocol runs, but can also intercept, overhear, and redirect any message in transit, and can create fake messages with the knowledge it has acquired. An intruder cannot decrypt encrypted messages for which it does not know the decryption key, but can use encrypted parts to create fake messages.

Model checking a cryptographic protocol typically consists of characterizing the states in which the desired property has been broken, and then exhaustively searching for the reachability of such a bad state from an initial state that contains an intruder.

3.1 Non-Zeno Intruders

To conveniently model and analyze real-time systems, most state transitions are assumed to be *instantaneous* (i.e., to take zero time). For this abstraction to make sense, systems should be *non-Zeno*, in that only a finite number of such transitions can be performed in a finite time interval. However, a naive lifting of a Dolev-Yao intruder to the timed setting can lead to Zeno behaviors, as an intruder may repeatedly create the same fake message at the same time.

We therefore propose a general method in which an intruder is restricted so it can perform *at most* `maxActions` number of transitions per time unit. In that way, Real-Time Maude *time-bounded* search and LTL temporal logic model checking commands are guaranteed to terminate.

3.2 A Complete Analysis Method for Non-Zeno Intruders

We use a *non-Zeno* intruder and a *time-bounded* search or LTL model checking command to search for attacks from a given initial state. Such analysis is *incomplete* for the given initial configuration, since it cannot discover attacks that take longer than the time bound and/or can only happen with an intruder that can perform more than `maxActions` number of actions per time unit. To regain completeness, we must repeat the search or model checking command until the attack is found, increasing the value of `maxActions` and/or the time bound after each unsuccessful command. To cover all such (`maxActions`, *duration*)-pairs, we suggest in a first step a simple "diagonalization" strategy, where we first analyze the system for, say, `maxActions` 2 and duration 3. If the attack is not found, then the search or model checking command is repeated for the `maxActions`/duration-pair $(2, 4)$. And so the process continues with the pairs $(3, 3)$, $(2, 5)$, $(3, 4)$, $(4, 3)$, $(2, 6)$, ..., until the attack is found. Theoretically, this process covers all possible attacks—relative to a given initial system state—while each single model checking/search command terminates. On the pragmatic level, our time-bounded search commands *with Zeno* intruders did not find the attacks below before memory was exhausted, and time-bounded temporal logic checking cannot be performed at all with such an infinitary intruder. Using our strategy, Real-Time Maude search could find all known flaws, as described below. In addition, our analysis method can be fully automated, as each time-bounded command terminates.

4 Overview of the WMF Protocol

The goal of the WMF protocol [1] is to allow two principals A and B to exchange a secret key via a trusted server S. The protocol should ensure *timed authentication* on the part of the responder: the responder should know that the protocol was *recently* initiated by the claimed initiator. The protocol is described as follows in the usual protocol notation:

$$\begin{array}{ll} A \longrightarrow S: & A\,.\,\{T_a, B, K\}_{K_{A,S}} \\ S \longrightarrow B: & \{T_s, A, K\}_{K_{B,S}} \end{array}$$

In the first step, the initiator A sends to the server S the message $A\,.\,\{T_a, B, K\}_{K_{A,S}}$, containing his identity together with a message, encrypted with the key $K_{A,S}$ shared between A and S, that contains the time the message was created (T_a), the identity of the desired responder (B), and the secret key K that A wants to establish with B. In the second step, the trusted server forwards to B the message containing the *current* time T_s, the identity of the initiator, and the suggested key K, all encrypted with the shared key $K_{B,S}$. In both steps, the receiver of a message must check the timestamp to ensure that the message was recently sent.

There are two well-known modifications of this protocol. To avoid mirror attacks due to the similarities of the messages, it has been suggested that the *rank* (`init` or `forward`) is added to the messages, so the two messages become, respectively, $A\,.\,\{T_a, B, K, \texttt{init}\}_{K_{A,S}}$ and $\{T_s, A, K, \texttt{forward}\}_{K_{B,S}}$. In WMF, the responder does not authenticate himself, and it has been suggested that a "hand-shake" is added at the end of the protocol to have *mutual* authentication [8].

4.1 Desired Properties

WMF is usually analyzed w.r.t. the following, increasingly weaker, properties:

Agreement: The responder B in a protocol run can trust the identity of the apparent initiator A, that A initiated a protocol run with B with the desire to send the received key K, and that the run A initiated corresponds to only one connection at B.

Non-injective agreement: As agreement, except that *one* run initiated by A with B can result in B being a responder to that run more than once.

Aliveness: When B is responder, apparently with A, then A must previously have been running the protocol (but not necessarily with B!).

In addition, there is a *recentness* requirement to each of the above requirements, which says that the two principals' runs must be *recent* to each other.

4.2 Some Underlying Assumptions

The clocks in the system are assumed to be synchronized. The assumptions about communication are less clear. We use a very general model, and only assume that a message takes *at least* time `minDelay` to travel from source to destination. There is no upper bound on the time it may take a message to arrive at the destination. Therefore, we also capture the possibility of message losses. A constant `delayLimit` gives the limit for recentness of a message. We assume that the intruder can also intercept messages which have traveled for *less* time than `minDelay`, but that any message sent from the intruder has a delay of at least `minDelay`.

5 Modeling WMF in Real-Time Maude

This section presents our specification of both the original WMF protocol and the protocol extended with ranks. The parts that are omitted in the rank-less version are placed within *thin* square brackets '[' and ']'. The entire specification, together with analysis commands, is available at `http://www.ifi.uio.no/RealTimeMaude/Crypto`.

We give an object-oriented specification of WMF. Principals, who can play both the initiator and "responder" (or "receiver") roles, are modeled as objects of a class `Agent`. Honest principals belong to a subclass `HonestAgent` of `Agent`, while dishonest intruders are objects of a subclass `Intruder` defined in Section 7. The trusted server is an object of the class `Server`.

5.1 Some Data Types

Keys. The ith key suggested by an agent A is modeled abstractly by a term `key`(A, i). The key $K_{A,S}$ shared by an agent A and the unique server is denoted `sharedKey`$(A,$ `server`$)$. Since `Oid` is the built-in sort for object identifiers, we declare keys as follows:

```
sort Key .
op key : Oid Nat -> Key [ctor] .
op sharedKey : Oid Oid -> Key [ctor comm] .
```

Connections. For convenient analysis, a node stores the connections it has established, as well as the time each connection was established. A connection where key K was suggested, where A is the apparent initiator and B the apparent responder, and which was initiated/received at time τ, is modeled by the term A `<->` B `with` K `atTime` τ of sort `Conx`:[3]

```
sort Conx .
op _<->_with_atTime_ : Oid Oid Key Time -> Conx .
```

A multiset of `Conx`s can be defined in the usual way:

```
sort ConxSet .  subsort Conx < ConxSet .
op empty : -> ConxSet .
op _&_ : ConxSet ConxSet -> ConxSet
                            [assoc comm id: empty] .
```

Messages. An initiator message $A.\{T_a, B, K,$ `init`$\}_{K_{A,S}}$ is modeled by a term

```
msg (A ; encrypt {Ta,B,K,init}
         with sharedKey(A, server)) from A to server
```

and the forward message $\{T_s, A, K,$ `forward`$\}_{K_{B,S}}$ by

```
msg (encrypt {Ts,A,K,forward} with sharedKey(B,server))
  from server to B
```

for `server` the name of the lone server. These messages are declared as follows:

```
sorts [Rank] MsgContent EncrMsg .
[ops init forward : -> Rank .]
op '{_','_'[',_]'} : Time Oid Key [Rank] -> MsgContent .
op encrypt_with_ : MsgContent Key -> EncrMsg .
op msg_;_from_to_ : Oid EncrMsg Oid Oid -> Msg .
op msg_from_to_ : EncrMsg Oid Oid -> Msg .
```

5.2 The Class Declarations

Each object has a `clock` attribute denoting the current time. An `Agent` also has a multiset `wantedConxs` of the names of the principals with whom it wants to initiate a run of the protocol, and a set `establishedConxs` of the connections (with keys) that the agent considers to be established:

```
class Agent | wantedConxs : OidSet,
              establishedConxs : ConxSet,
              keyCtr : Nat,
              clock : Time .

class HonestAgent .     subclass HonestAgent < Agent .

class Server | clock : Time .
op server : -> Oid .     --- name of key server
```

[3] To save space, '`Conx`' is used for '`Connection`', and the `ctor` attribute is henceforth omitted from constructor declarations.

5.3 The Instantaneous Dynamic Behavior

The instantaneous dynamic behavior of WMF is modeled by three rules: (i) sending an initiator message, (ii) receiving the initiator message and forwarding the message, and (iii) receiving the forwarded messages. Notice that the `establishedConxs` attribute is *not* a part of the *protocol* itself, and does not impact the behavior of the protocol.

The following rule models the start of a run of the protocol. An agent `INIT` has an object name `RESP` in its `wantedConxs` set, and starts a run of the protocol by sending an initiate message (to `server`) with *least* delay `minDelay`:

```
vars O INIT RESP A B I : Oid .  vars T T' : Time .
var OS : OidSet .  vars CS CS' : ConxSet .
vars KEY K : Key .  var N : Nat .  var M : Msg .
var C : Configuration .  var EncrMSG : EncrMsg .
var ENCRMSGS : EncrMsgSet .

rl [initiate_run] :
   < INIT : Agent | wantedConxs : RESP ; OS, keyCtr : N,
                    establishedConxs : CS, clock : T >
  =>
   < INIT : Agent |  wantedConxs : OS, keyCtr : s N,
                     establishedConxs : CS &
                       (INIT <-> RESP
                          with key(INIT, N) atTime T) >
   dly(msg INIT ; encrypt {T, RESP, key(INIT, N)[, init]}
                  with sharedKey(INIT, server)
         from INIT to server, minDelay) .
```

In the next rule, the server checks whether an `init`-message is recent (`T' monus T < delayLimit`, where x `monus` $y = \max(x - y, 0)$) and, if so, forwards the corresponding message (with appropriate least delay) to the intended responder. If the message is not recent, it is just discarded:

```
rl [forward] :
   (msg INIT ; encrypt {T, RESP, KEY[, init]}
               with sharedKey(INIT, server)
      from INIT to server)
   < server : Server | clock : T' >
  =>
   if (T' monus T) < delayLimit then   --- recent message
     < server : Server | >
     dly(msg (encrypt {T', INIT, KEY[, forward]}
              with sharedKey(RESP, server))
         from server to RESP, minDelay)
   else    --- not recent; just discard message:
     < server : Server | >
   fi .
```

Finally, in the third rule, the intended "responder" receives the `forward` message and, if it is recent, stores the new connection as established:

```
rl [receive_final_msg] :
   (msg (encrypt {T, INIT, KEY[, forward]}
         with sharedKey(RESP, server))
    from server to RESP)
  < RESP : HonestAgent | clock : T',
                         establishedConxs : CS >
=>
 if (T' monus T < delayLimit) then  --- new connection:
   < RESP : HonestAgent | establishedConxs : CS &
                          (INIT <-> RESP
                           with KEY atTime T') >
 else  < RESP : HonestAgent | >  fi .
```

5.4 Behavior in Time

The transmission time of a message can be *any* time greater than or equal to `minDelay`. This implies that there is no urgency in our model. Apart from this aspect, we follow the techniques for object-based specification in [15], and define the tick rule as follows:

```
rl [tick] : {C} => {delta(C, T)} in time T .
```

The tick rule is nondeterministic in that time may advance by *any* amount. Before executing the specification, we must define a *time sampling strategy* to guide the execution of the tick rule [15].

The function `delta` defines the effect of time elapse in a system. It distributes over the objects and messages in a configuration [15], and applies to single objects by just increasing their clocks according to the elapsed time, and to delayed messages by decreasing the remaining delay:

```
eq delta(< O : HonestAgent | clock : T >, T') =
         < O : HonestAgent | clock : T + T' > .
eq delta(< O : Server | clock : T >, T') =
         < O : Server | clock : T + T' > .
eq delta(dly(M, T), T') = dly(M, T monus T') .
```

Our specification is parametric in the time domain, which may be dense or discrete. We import the built-in module `NAT-TIME-DOMAIN` to define the sort `Time` to be the natural numbers.

6 Analyzing WMF Without Intruders

This section illustrates how WMF can be simulated and model checked in a non-hostile environment. To capture all possible behaviors, we select the time sampling strategy that advances time by one time unit in each tick rule application by giving the command `(set tick def 1 .)`. We define the constants `minDelay` and `delayLimit` to be, respectively, `1` and `2`, and define the following initial states `init1` and `init3`:[4]

```
subsort String < Oid .      --- agent names are strings
ops init1 init3 : -> GlobalSystem .
```

[4] Parts of specification and commands will be replaced by '...'

```
eq init1 =
   {< "Alice" : HonestAgent |
                wantedConxs : "Bob", keyCtr : 1,
                establishedConxs : empty, clock : 0 >
    < "Bob" : HonestAgent |
              wantedConxs : "Alice", keyCtr : 1,
              establishedConxs : empty, clock : 0 >
    < server : Server | clock : 0 >} .

eq init3 =
   {< "Alice" : HonestAgent |
                wantedConxs : "Bob" ; "Charlie",
                establishedConxs : empty,
                clock : 0, keyCtr : 1 >
    < "Bob" : HonestAgent |
              wantedConxs : "Alice" ; "Charlie", ... >
    < "Charlie" : HonestAgent |
                  wantedConxs : "Alice", ... >
    < server : Server | clock : 0 >}.
```

In `init1`, two agents want to initiate contact with each other. `init3` has three agents, two of which want to initiate two connections each. A node may initiate a run at *any* time. Furthermore, `wantedConx` is a multiset, so that `"Alice"` can initiate contact with `"Charlie"` either *before* or *after* contacting `"Bob"`.

The following timed search command searches for a state, reachable from `init3`, in which all the desired connections have been established:[5]

```
(tsearch [1] init3 =>*
 {< "Alice" : HonestAgent |
     establishedConxs :
      ("Alice" <-> "Bob" with K1:Key atTime T1:Time) &
      ("Alice" <-> "Charlie" with K2:Key atTime T2:Time) &
      ("Bob" <-> "Alice" with K3:Key atTime T3:Time) &
      ("Charlie" <-> "Alice" with K4:Key atTime T4:Time) >
  < "Bob" : HonestAgent |
     establishedConxs :
      ("Bob" <-> "Alice" with K3:Key atTime T5:Time) &
      ("Bob" <-> "Charlie" with K6:Key atTime T6:Time) &
      ("Alice" <-> "Bob" with K1:Key atTime T7:Time) >
  < "Charlie" : HonestAgent | ... >
  < server : Server | clock : T11:Time >} in time < 4 .)
```

Real-Time Maude returned such a desired reachable state (in 52 seconds).

We next analyze our specification w.r.t. the strongest desired property: *recent agreement*. That is, for each connection established by a responder, there must have been a *distinct*, *recent* connection initiated by the corresponding initiator, and with the same key values. Note that an agent may want to initiate more than one connection with some other agent. We analyze recent agreement by searching for a state in which there exists a pair of agents A and B for which the recent agreement property does *not* hold, namely, where B's multiset of connections initiated by A cannot be matched to the initiated connections of A. The following function `recentAgrOK` is defined so that `recentAgrOK`$(A, B, conxs_A, conxs_B)$ is `true` if each connection from A to B in $conxs_B$ can be matched by the corresponding recent connection in $conxs_A$:[6]

```
ceq recentAgrOK(INIT, RESP,
                (INIT <-> RESP with K atTime T) & CS,
                (INIT <-> RESP with K atTime T') & CS') =
    recentAgrOK(INIT, RESP, CS, CS') if recent(T, T') .

--- no corresponding connection at the initiator:
eq recentAgrOK(INIT, RESP, CS, CS') =
      not (INIT <-> RESP in CS') [owise] .

op _<->_in_ : Oid Oid ConxSet -> Bool .
eq A <-> B in ((A <-> B with K atTime T) & CS) = true .
eq A <-> B in CS = false [owise] .

op recent : Time Time -> Bool .
eq recent(T, T') =
      (T' monus T) <= (2 * (delayLimit monus 1)) .
```

The following command checks whether it is possible to reach a state, in time < 5 from `init3`, where there is a pair of agents for which the recent agreement property does not hold:

```
(tsearch [1] init3 =>*
  {C:Configuration
   < O:Oid : HonestAgent | establishedConxs : CS:ConxSet >
   < O':Oid : HonestAgent | establishedConxs :
                            CS':ConxSet >}
     such that not recentAgrOK(O:Oid, O':Oid,
                               CS:ConxSet, CS':ConxSet)
    in time < 5 .)
```

No such state was found (the search took 850 seconds; checking state `init1` within duration `50` took 100 seconds). WMF looks good in a non-hostile environment.

7 Specification of Non-Zeno Intruders

This section specifies the non-Zeno intruder for WMF. Since an intruder can also participate in ordinary protocol runs, the `Intruder` class is a subclass of `Agent`. In addition, an intruder has attributes for storing the set of agent names it knows (`agentsSeen`) and the set of encrypted message contents that it has seen but could not decrypt (`msgsSeen`). Finally, an intruder can only perform a malicious action when its `actionsLeft` value is greater than zero:

```
class Intruder | msgsSeen : EncrMsgSet,
                 agentsSeen : OidSet, actionsLeft : Nat .
subclass Intruder < Agent .
```

[5] Terms *name*:*sort* are variables declared on-the-fly.

[6] The attribute `[owise]` (for "otherwise") in the second and fourth equations means that the corresponding equation can only be applied when the first and third equations cannot be applied.

The intruder class has rules for receiving a forward message in an ordinary protocol run (where it also learns of a new agent name), for intercepting messages of the two kinds, and for creating fake messages of both kinds using the information it knows. Other actions, such as overhearing a message, can be performed as a combination of interception and message creation.

The rule for intercepting an `init`-message is as follows, and can only be performed when the `actionsLeft` value is greater than zero:[7]

```
crl [intercept_msg_1] :
    dly(msg INIT ; EncrMSG from INIT to server, T)
    < I : Intruder | agentsSeen : OS, actionsLeft : s N,
                     msgsSeen : ENCRMSGS >
  =>
    < I : Intruder | agentsSeen : add(INIT, OS),
                     actionsLeft : N,
                     msgsSeen : ENCRMSGS ; EncrMSG >
  if I =/= INIT .
```

In this rule, the intruder has learnt a name (`INIT`) and an encrypted message content (`EncrMSG`). In our flexible communication model, an intruder can also intercept messages that are not yet "ripe" (`dly(...,T)`).

The rule for creating a fake forward message takes *any* name `A` from the set `A ; OS` of known names and *any* encrypted message content `EncrMSG` (without knowing its rank) from the set of known contents, and sends a fake message to `A`, pretending to be from `server`:

```
rl [fake_msg_2] :
  < I : Intruder | actionsLeft : s N,
                   agentsSeen : A ; OS,
                   msgsSeen : EncrMSG ; ENCRMSGS >
 =>
  < I : Intruder | actionsLeft : N >
  dly(msg EncrMSG from server to A, minDelay) .
```

Finally, we define how the passage of time affects an intruder, namely, by increasing its clock and recharging its `actionsLeft` attribute:

```
eq delta(< I : Intruder | actionsLeft : N,
                          clock : T >, T') =
         < I : Intruder | actionsLeft : maxActions * T',
                          clock : T + T' > .
```

8 Analyzing WMF with Intruders

We analyze WMF in a hostile environment using the method given in Section 3. To the previous initial state `init1` we add a non-Zeno intruder object:

```
eq init1+Intruder =
    {< "Alice" : HonestAgent |  ... >
     < "Bob" : HonestAgent | ... >
     < server : Server | ... >       --- as before
     < "Eve" : Intruder | wantedConxs : none, clock : 0,
                          establishedConxs : empty,
                          msgsSeen : noEncrMsg,
                          actionsLeft : 2, keyCtr : 1,
                          agentsSeen : none >} .
```

[7] The term `s N` will match any number which is a *successor* (`s`) of a natural number.

8.1 Analyzing WMF Without Ranks

We analyze the *original* version of WMF with respect to the weakest property, *aliveness*: if B has been a responder in a protocol run allegedly initiated by A, then A must previously have run the protocol. The following search command checks whether, from a state `init1+Intruder` *with* `"Bob"` *removed from the state*, it is possible to reach a state where `"Alice"` has established a connection initiated by the non-existing `"Bob"`:

```
(tsearch [1]
  {< "Alice" : HonestAgent | wantedConxs : "Bob", ... >
   < server : Server | clock : 0 >
   < "Eve" : Intruder | ...  >} --- as in init1+Intruder
  =>*
  {REST:Configuration
   < "Alice" : HonestAgent | establishedConxs :
          ("Bob" <-> "Alice" with K:Key atTime T:Time)
           & CS:ConxSet >}  in time < 4 .)
```

The search (the first in our method of repeated searches) found such a solution in 30 milliseconds. We can then use Real-Time Maude to exhibit the path leading to the state matching the search pattern [12]. It is actually a trivial mirror attack, where the intruder intercepts `"Alice"`'s initiate message, strips away the first part, and sends back the rest to `"Alice"`:

$$\begin{array}{ll} \texttt{Alice} \rightarrow \texttt{Eve(Bob)} : & \texttt{Alice}.\{0, \texttt{Bob}, K_{\texttt{Alice},1}\}_{K_{\texttt{Alice},\texttt{server}}} \\ \texttt{Eve(Bob)} \rightarrow \texttt{Alice} : & \{0, \texttt{Bob}, K_{\texttt{Alice},1}\}_{K_{\texttt{Alice},\texttt{server}}} \end{array}$$

8.2 Analyzing WMF with Ranks

We now analyze the modified version of WMF, and use the same search command (with `init3` replaced by `init1+Intruder`) as in our analysis of the *recent agreement* property in the intruder-less setting in Section 6. The third search in our strategy (i.e., for `maxActions` 3 and time bound 3) found a state violating recent agreement in 1.5 seconds. (The corresponding path shows a trivial replay attack in which the intruder intercepts a forward message, and at the same time forwards two copies of it.) The first two searches took 65 and 447 seconds on a Pentium Xeon 3.6 GHz. In [3] it took 3093 seconds to find this violation.

To analyze the weaker *recent non-injective agreement* property, i.e., that for each established responder-connection, the supposed initiator did indeed initiate a run with the responder and with the correct key, we search for a state where this property does not hold:

```
(tsearch [1] init1+Intruder =>*
  {REST:Configuration
   < B:Oid : X:Agent | establishedConxs :
              (A:Oid <-> B:Oid with K:Key atTime T:Time)
              & CS:ConxSet >
   < A:Oid : Y:Agent | establishedConxs : CS':ConxSet >}
        such that
         not ((A:Oid <-> B:Oid with K:Key atTime T:Time)
              in CS':ConxSet)
   in time < 5 .)
```

which uses the auxiliary function

```
op _in_ : Conx ConxSet -> Bool .
ceq (A <-> B with K atTime T) in
    ((A <-> B with K atTime T') & CS)
        = true if recent(T', T) .
eq A <-> B with K atTime T in CS = false [owise] .
```

The first two such searches (in 56 and 390 seconds) did not find any state violating the recent non-injective agreement property. Neither did the third search until it ran out of memory.

9 Concluding Remarks

In this paper we have investigated the suitability of the general-purpose Real-Time Maude tool for the formal modeling and analysis of time-dependent cryptographic protocols. We illustrate such specification and analysis with the benchmark WMF protocol. We restrict the intruder to make it non-Zeno, and define a simple mechanizable analysis method for gradually increasing the time bound and intruder capabilities until an attack is found. This method yields a complete semi-decision procedure for the reachability problem from a given initial configuration.

We have tested our method on the WMF and Kerberos protocols (see [6] for details on the latter). The results are promising. First of all, we obtain what we believe are fairly intuitive object-oriented specifications, in part because we do not need an "ad hoc" treatment of time. Second, we could easily find all known flaws in the protocols in typically shorter time than in other analysis efforts.[8]

Given the promising initial results, we should refine and implement our method, as well as developing useful state space reduction and symbolic techniques to be able to analyze larger time-dependent security protocols. Finally, the expressiveness of Real-Time Maude, also illustrated by, e.g., analysis of advanced sensor network algorithms [17], should make it a promising candidate for analyzing more advanced protocols in novel settings, such as security protocols for wireless sensor networks (e.g., [19]).

[8] The analysis in the paper [9] only treats the original version of WMF, so the results in [3] provide the only comparable figures for the modified version of WMF.

References

[1] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.2)*, 2005. `http://maude.cs.uiuc.edu`.

[3] G. Delzanno and P. Ganty. Automatic verification of time sensitive cryptographic protocols. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *LNCS*, pages 342–356. Springer, 2004.

[4] Formal Systems (Europe) Ltd. *FDR2 user manual*, 2003.

[5] A. Goodloe, C. A. Gunter, and M.-O. Stehr. Formal prototyping in early stages of protocol design. In *WITS'05*. ACM Press, 2005.

[6] M. Grimeland. Modeling and analysis of time-dependent security protocols in Real-Time Maude. Master's thesis, Dept. Informatics, Univ. Oslo, 2006. `http://www.ifi.uio.no/RealTimeMaude/Crypto/`.

[7] Kerberos: The network authentication protocol. `http://web.mit.edu/Kerberos/`.

[8] G. Lowe. A family of attacks upon authentication protocols. Technical report, Dept. of Mathematics and Computer Science, University of Leicester, 1997.

[9] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[10] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2), 1996.

[11] J. Meseguer. Membership algebra as a logical framework for equational specification. In *Proc. WADT'97*, volume 1376 of *LNCS*. Springer, 1998.

[12] P. C. Ölveczky. *Real-Time Maude 2.2 Manual*, 2006. `http://www.ifi.uio.no/RealTimeMaude/`.

[13] P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In *FASE'06*, volume 3922 of *LNCS*. Springer, 2006.

[14] P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In *FASE 2004*, volume 2984 of *LNCS*. Springer, 2004.

[15] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 2007. To appear.

[16] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29:253–293, 2006.

[17] P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In *IPDPS 2006*. IEEE, 2006.

[18] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. The TESLA broadcast authentication protocol. *RSA CryptoBytes*, 5(Summer), 2002.

[19] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tyga. SPINS: Security protocols for sensor networks. *Wireless Networks Journal*, 8(5):521–534, 2002.