UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

TESIS DE MÁSTER

Mejoras en la interacción con la herramienta Maude-NPA

Author: Damián Aparicio Sánchez

Tutor: Santiago Escobar Román

Date: September, 2017

Departamento de Sistemas Informáticos y Computación
Universitat Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia, Spain

DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Abstract

The security in modern online services is increasingly more and more vulnerable to intruder attacks. Companies like Yubiko works on devices like Yubikey, a USB device that provides strong identification. In this thesis, we have specified and analyzed the cryptographic protocol underneath the Yubikey device using the Maude-NPA tool, a state-of-the-art cryptographic protocol analyzer. During this thesis, we learned how the Yubikey protocol works, we learn how to use the Tamarin proper, another protocol analyzer, and how to specify and analysis this protocol using the Maude-NPA features.

Contents

1. Introduction

Nowadays there exists several security tokens having the form of a smartcard or of an USB device, which are designed for protecting cryptographic values from an intruder e.g: hosting service, emails, e-commerce, online Banks, etc. They are also used to ease authentication for the authorized users at a service. If you are using a service that verifies your Personal Identification Number (PIN), it should not be used for checking your flights, reading your emails, etc. It should in fact only be used when it is necessary to check your PIN. In this way, if we can use an Application Programing Interface (API) to separate the service from the authentificator system, we can say that a certain amount of attacks related to user identification are discarded. Yubikey token is an API designed with a button that once it is touched triggers a new user. The device has as purpose the authentication of the user against some service.

The goal of this master thesis is to formally analyze the Yubikey protocol API using Muade-NPA, a-state-of-art security protocol verification tool. The complexity of this work lies in the time/order of execution process. The purpose of this master is whether Maude-NPA is useful to specify and control this execution process. To prove our work, we use composition to control the global memory and its set of facts. We were able to verifiy the Yubikey protocol.

The master thesis has had two parts. The first part is about how security tokens in special the Yubikey protocol API Works. This was formally discussed in Section 4.1 the PhD thesis of Robert Künnemann [Künnemann14]. We had to learn the syntax and funcionality of the tool Tamarin prover [Tamarin16], another protocol analysis tool, because the Yubikey protocol was specified in Tamarin prover. The second part was about the specification of the Yubikey protocol and the difficulties Maude-NPA has had, as well as errors found during the realization of this master thesis.

The thesis is organized as follows. In Section 2 and Section 3 we briefly present the two tools, Tamarin prover and Maude-NPA, in Section 4 how Yubikey protocol works. We conclude in Section 5 with the specification and verification of the protocol.

2. Tamarin Prover

Tamarin prover [Tamarin16] is a cryptographic protocol verifier which allows the user several things at the same time to specify complex security properties (both trace and equivalence properties), to model cryptographic primitives by means of an equational theory, and to model protocols with state information. The class of equational theories supported by the tool is the class of subterm-convergent equational theories, in addition to built-in theories for Diffie-Hellman exponentiations, bilinear pairings, multisets and equational theories with the finite variant property.

The Tamarin prover is a powerful tool for the symbolic modeling and analysis of security protocols[Simon13]. It takes as input a security protocol model, specifying the actions taken by agents running the protocol in different roles (e.g., the protocol initiator, the responder, and the trusted key server), a specification of the adversary, and a specification of the protocol's desired properties. Tamarin prover can then be used to automatically construct a proof that, even when arbitrarily many instances of the protocol's roles are interleaved in parallel, together with the actions of the adversary, the protocol fulfils its specified properties. In this thesis, we provide an overview of this tool and its use.

2.1. Tamarin Prover Syntax

The syntax of security protocols are delimited by **begin** and **end**. We explain briefly the formal syntax in the following paragrahs, for futher detail see[Referencia].

security_protocol_theory = **theory** name **begin** body **end**

The security protocol has multiset rewriting rules that are
specified in the body.

```
body =(signature_spec | rule | restriction | lemma | formal_comment)+

signature_spec = functions | equations | builtin

builtin  = 'builtins' ':' builtins list

builtins = 'diffie-hellman' | 'hash' | 'symm-encrypt' | 'asymm-encrypt | 'sign'
```

Rules are the most important part to understand how Tamarin prover works. The rules operate on the system’s state, which are expressed as a multiset of facts. Facts can be seen as predicates proving state information. Rules have a premise (left-hand side) and a conclusion (right-hand side), separated by the arrow symbol -->. A rule can be executed when the premise is present in the system’s state, and when the rule is executed, the system’s state is modified according to the conclusion.

```
rule = 'rule' name ':' '[' facts ']' ( '-->'  |  '--[' action facts ']->') '[' facts ']'
```

Some rules contain action facts inside the arrow (e.g., --[action facts]->). They are just like facts, but unlike regular facts the other facts do not appear in the state, only on the trace. The security properties are checked on the traces, this means action facts are used afterwards to determine which agents (roles) are compromised.

In the last part of the specification of the protocol in Tamarin prover we have lemmas to specify security properties. By default, the formula below of the lemma is interpreted as a property that must hold for all traces of the protocol of the security protocol theory.

```
lemma := 'lemma' ident [lemmaattrs] ':'

             [trace_quantifier]

             '"' formula '"'

             proof

lemmaattrs = '['('source' | 'reuse' | 'use_induction'  | 'hide_lemma=' ident)']'

tracequantifier := 'all-traces' | 'exists-trace'

proof  = ... a proof as output by the Tamarin prover ...
```

2.2. Protocol specification in Tamarin Prover

In this section, we provide an informal description about protocol specification model.

Tamarin prover models are specified using three levels, we are going to focus in the first two of them:

1. Rules
2. Facts
3. Terms

2.2.1. Rules

Multiset rewriting is used at the concurrent execution of the protocol and at the adversary. A rewriting rule has three parts, each of them is a sequence of facts (we have already seen the structure in the previous section), the rules of a posible specification model are:

```
rule FirstRule:
[ ]
-->
[ F('1','x'), F('2','y') ]


rule SecondRule:
[ F(u,v) ]
-->
[ H(u), G('3',h(v)) ]
```

In this case the initial state of the comunication is the empty multiset. A rule can be applied to a state if it can be instantiated such that the facts its left hand side are contained in the set of facts of the current state. In this case, the left-hand side facts are removed from the state, and replaced by the instantiated right hand side.

For instant the rule FirstRule can be instantiated repeatedly. And for any instantiation of FirstRule, it will be followed by a state that contains *[F('1','x')] or [F('2','y')]* and in this case SecondRule can be applied to both of them. Each of these instantiations leads to a new successor state.

2.2.2. Facts

Facts follow the simple structure F(a_1,…,an) for a fact symbol F and terms a_1,…,a_n can be seen as predicates storing state information. Tamarin has different fact types, these are used to model interaction with the communications, the most used are:

Name(a1,…,an) = normal facts, produced by rules and also consumed by rules, hence they might appear in one state but not in the next.

In() = This fact is used to receive a message from the communication network, that will be controlled by adversary, and can only occur on the left-hand side of a rewrite rule, as normal facts, it is consumed by rules.

Out() = This fact is used to send a message to the communication network that will be controlled by an adversary, and can only occur on the right-hand side of a rewrite rule, as normal facts, it is consumed by rules.

Fr() = This fact must be used when generating fresh (random) values, and can only occur on the left-hand side of a rewrite rule, where its argument is the fresh term, as normal facts, it is consumed by rules.

!Name(a1,…,an) = Special fact will never be removed from state once they are introduced. There are called 'persistent facts'

--[actionFacts]-> = Action facts, used to indicate that the message is supposed to be secret. We used when agents' keys are compromised.

2.3. First Example

To understand better the specification of a protocol we are going to use a example [Simon16] of a communication protocol that consist of two messages, using Alice and Bob notation.

```
C -> S: aenc(k, pkS)
S -> C: h(k)
```

A client C sends to the server S a new fresh key k, encrypted with the public key *pkS* of a server S (*aenc* means asymmetric encryption) generated previously before shipping. The communication follow-up with server S confirms the key's receipt by sending to the client C the hash h of the key.

Tamarin prover uses multiset rewriting rules to operate on the system's state, which is expressed as a multiset (i.e., a bag) of facts. We use facts to store information about the state given by their arguments. The rules have a premise and a conclusion, separated by the arrow symbol -->. Facts in the conclusion will be added to the state, while the premises are removed, unless the fact is preceded by symbol !.

We start with the modeling of the infrasctructure of protocol.

```
rule Register_pk:
[ Fr(~ltk) ]
-->
[ !Ltk($C, ~ltk), !Pk($C, pk(~ltk)) ]
```

This rule has only a premise which is an instance of the *Fr* fact, a built-in fact that denotes a freshly generated name.

First, a fresh name *~ltk* (of sort fresh) is generated, which is the new private key, and non-deterministically choose a public name C, for the role for whom we are generating the key-pair. Then, generates the fact !*Ltk($C, ~ltk)* (the mark ! denotes that the fact is persistent), which denotes the association between role C and its private key *~ltk*. Finally generates the permanent fact *!Pk($C, pk(~ltk))*, which associates agent C and its public key *pk(~ltk)*.

```
rule Get_pk:
[ !Pk(C, pubkey) ]
-->
[ Out(pubkey) ]
```

The second rule reads a public-key database entry and sends the public key to the channel using the built-in fact *Out*,

```
rule Reveal_ltk:
[ !Ltk(C, ltk) ]
--[ LtkReveal(C) ]->
[ Out(ltk) ]
```

This rule is to reveal when the private key has been compromised. This rule reads a private-key database entry and sends it to the channel (knowledge adversary). Besides, the rule has an *actionfact* ‘LtkReveal’ (it is used to determine whether role C was compromised).

Recall the Alice-and-Bob notation of the protocol we want to model:

```
  C -> S: aenc(k, pkS)
  C <- S: h(k)
```

The first step is writing the rules for the client C

```
rule Client_1:
[ Fr(~k), !Pk($S, pkS) ]
// choose fresh key
// lookup public-key of server
-->
[ Client_1( $S, ~k ), Out( aenc(~k, pkS) )]
// Store server and key for next step of thread
// Send the encrypted session key to the server

rule Client_2:
[ Client_1(S, k), In( h(k) )
// Retrieve server and session key from previous step
// Receive hashed session key from network
--[ SessKeyC( S, k ) ]->
// State that the session key 'k'
[]
// was setup with server 'S'
```

Next, server S aswers in one-step to a session-key setup request from the client C.

```
rule Serv_1:
[ !Ltk($S, ~ltkS), In( request ) ]
// lookup the private-key
// receive a request
--[ AnswerRequest($S, adec(request, ~ltkS)) ]->
// Compromise role S
// adec = asymmetric decryption algorithm
[ Out( h(adec(request, ~ltkS)) ) ]
// Return the hash of the decrypted request.
```

In addition, first and second rule models the client C sending its message, and the second rule models the receiving of a response. The third rule models the server S, both receiving the message and requesting in the same rule.

2.3.1. Security properties

In Tamarin prover to evaluate security properties[Meier13] we must use *lemmas* and have to defined over tracers of the actions facts of the protocol execution.

As an example, to verify a lemma saying that it cannot be possible that a client has set up a session key k with a server S and the adversary learned that k unless the adversary performed a long-term key reveal on the server S. We will use the following syntax:

- Ex = existential quantification
- # = temporal variables
- & = conjunction
- f @ i = action constraints, restricions to control the temporal variables, 'i' is optional
- not = negation

Lemma proves that it cannot be posible that a client has set up a session key(SessKeyC) 'key' with a server 'Server' and the adversary knows 'key' without having performed a long-term key reveal on 'Server'.

The mathematical representation of the security property is expressed using action constraints in Tamarin as follows:

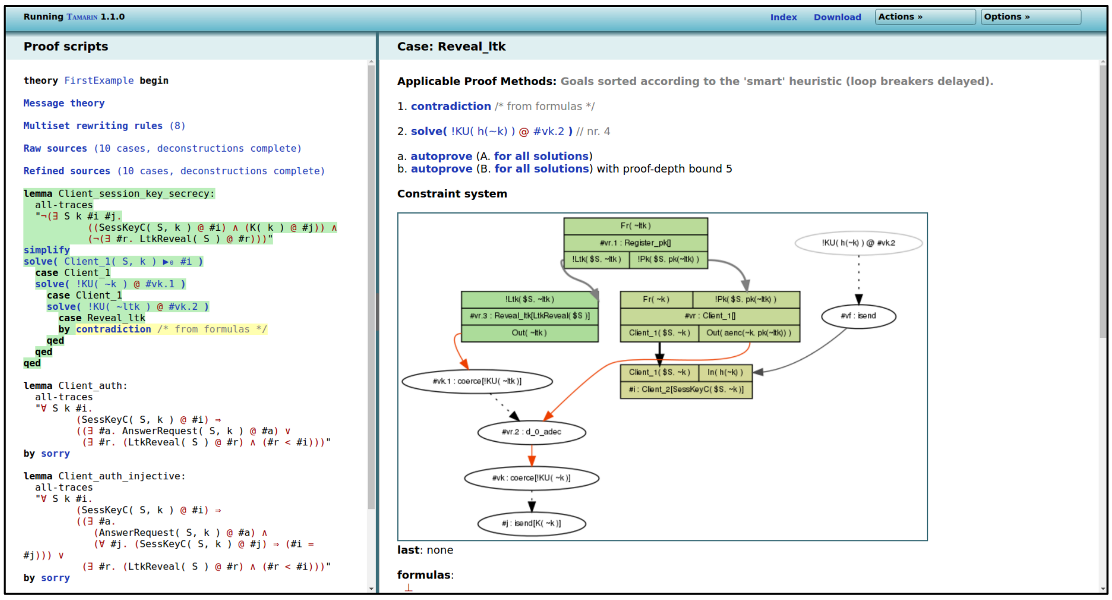
$$\neg(\exists t_1, t_2, Server, key.$$
$$\text{SesskeyC}(Server, key) \ @ \ t_1 \wedge$$
$$\text{K}(key) \ @ \ t_2 \wedge$$
$$\neg(\exists t_3.\text{LtkReveal}(Server) \ @ \ t_3)$$

Note that t_1,t_2 and t_3 are timepoints and therefore **SesskeyC** @ t_1, **K** @ t_2, **LtkRevel** @ t_3 events mean that the trace contains a rule instantiation that produces the action **SesskeyC, K, LtkRevel** at different timepoint t_1, t_2, t_3. Note that they share the Server and key parameters.

The specification of this lemma using Tamarin syntax is as follows:

```
lemma Client_session_key_secrecy:
"not(Ex S k #i #j.
SessKeyC(S, k) @ #i & K(k) @ #j & not(Ex #r. LtkReveal(S) @ r))"
```

In the Figure 1 we show how Tamarin has verified the lemma, using green color to highlight different parts, that mean it was successfully proven. If we had found a counterexample, it would be colored in red.



Proof scripts

```
theory FirstExample begin

Message theory

Multiset rewriting rules (8)

Raw sources (10 cases, deconstructions complete)

Refined sources (10 cases, deconstructions complete)

lemma Client_session_key_secrecy:
  all-traces
  "¬(∃ S k #i #j.
           ((SessKeyC( S, k ) @ #i) ∧ (K( k ) @ #j)) ∧
           (¬(∃ #r. LtkReveal( S ) @ #r)))"
simplify
solve( Client_1( S, k ) ▶₀ #i )
  case Client_1
  solve( !KU( ~k ) @ #vk.1 )
    case Client_1
    solve( !KU( ~ltk ) @ #vk.2 )
      case Reveal_ltk
      by contradiction /* from formulas */
    qed
  qed
qed

lemma Client_auth:
  all-traces
  "∀ S k #i.
         (SessKeyC( S, k ) @ #i) ⇒
         ((∃ #a. AnswerRequest( S, k ) @ #a) ∨
          (∃ #r. (LtkReveal( S ) @ #r) ∧ (#r < #i)))"
by sorry

lemma Client_auth_injective:
  all-traces
  "∀ S k #i.
         (SessKeyC( S, k ) @ #i) ⇒
         ((∃ #a.
            (AnswerRequest( S, k ) @ #a) ∧
            (∀ #j. (SessKeyC( S, k ) @ #j) ⇒ (#i =
#j))) ∨
          (∃ #r. (LtkReveal( S ) @ #r) ∧ (#r < #i)))"
by sorry
```

Case: Reveal_ltk

Applicable Proof Methods: Goals sorted according to the 'smart' heuristic (loop breakers delayed).

1. **contradiction** /* from formulas */

2. **solve(** !KU(h(~k)) @ #vk.2 **)** // nr. 4

a. **autoprove** (A. **for all solutions**)
b. **autoprove** (B. **for all solutions**) with proof-depth bound 5

Constraint system

last: none

formulas:
⊥

Figure 1 : Image from Tamarin prover tool

3. Maude-NPA

Maude-NPA[MNPA17] is an analysis tool used for verifying the security of cryptographic protocols, that takes the ability to check algebraic properties. Besides the tool is specialized to prove proofs of security as well as to search for attacks, we have focused in this part to verify the security guarantee of Yubikey protocol. Maude-NPA is a backwards search tool, i.e., it searches backwards from a final state to determine whether or not it is reachable from an initial state, in this way we can prove that final state is unreachable.

3.1. Maude-NPA structure

Maude-NPA tool consists in three sections called modules, having a fixed format and fixed module names, The first module is the *syntax of the protocol* that is focused in the statement of sorts and operators. The second module specifies the *algebraic properties* of the operators. And lastly the third module using a strand-theoretic notation or a process algebra. This module includes the intruder capabilities (using formal model Dolev-yao), regular strands or processes (describing the behavior of roles). It also contains analysis strands called attacks, which describing behavior that we want to verificate.

3.2. Specifying Syntax

This module is used to specify protocol and all it is relevant items in the current version of the Maude-NPA, the syntax is specified in the module PROTOCOL-EXAMPLE-SYMBOLS.

3.2.1. Sorts, Subsorts and Operators

In general, sorts are used to specify different types of data, that are used for different purposes. We have a special sort called Msg that represents what messages are going to look like in our protocol. If a protocol makes no additional sort distinctions, i.e., if it is an unsorted protocol, there will be no extra sorts, and every symbol will be of sort Msg.

```
sorts Name Nonce .
subsort Name Nonce < Msg .
subsort Name < Public .
```

Most sorts are user-defined, however, there are several special sorts that are automatically imported by any Maude-NPA protocol definition. These are:

Msg : Sorts defined by the user must be subsorts of Msg, and no sort defined by the user can be a supersort of Msg. This sort cannot be empty, i.e., it is necessary to define at least one symbol of sort Msg or of a subsort of Msg.

Fresh : The sort Fresh is used to identify terms that must be unique. This sort is typically used as an argument of some data that must be unique, such as a nonce, or a session key, e.g., "n(A,r)" or "k(A,B,r)" where r is a variable of sort Fresh. It is not necessary to define symbols of sort Fresh, i.e., the sort Fresh can be empty.

Public : The sort Public is used to identify terms that are publically available, and therefore assumed known by the intruders. This sort cannot be empty.

The next step is to specify the operations, Maude-NPA is very flexible and allows several operator declarations. We have the standard prefix sysntax and the mix-fix syntaxs (e.g., _;_) these operator allow us to have the posibility to determine what equational attributes we want, such as associativity, commutativity, and identity.

prefix syntax

```
op Fr : Fresh -> FrNonce .
op Init : FrNonce FrNonce -> Init .
op Server : FrNonce FrNonce Counter -> Server .
```

mix-fix syntax

```
op _+_ : Counter Counter -> Counter [assoc comm] .
op _;_ : StdMsg StdMsg -> StdMsg [gather (e E)] .
op _++_ : EventList EventList -> EventList [assoc id: nil prec 31] .
op _+++_ : EventList End -> EndEventList [prec 33] .
```

Maude-NPA has the capacity to work with special unification algorithms, like associative *assoc*, commutative *comm* and identity *id: keyword*. Other characteristic that Maude-NPA can control using the operator attribute
gather (e E) associativity to the left; whereas *gather (E e)* indicates association to the right. Precedence[Maude16] *prec number* is given to natural number, where a lower value indicates a higher priority in this parser.

3.3. Algebraic Properties

Maude-NPA verification is a powerfull symbolic reachability analysis modulo the equational theory of the protocol.

Maude-NPA supports three algebraic properties types: (I)equational axioms, such as commutativity, or associativity-commutativity, called axioms. (II)Equational rules, called variant equations, and (III)equational rules for dedicated unification algorithms, called dedicated equations. Variant and dedicated equations are specified in the PROTOCOL-EXAMPLE-ALGEBRAIC.

```
var Z : Msg . var A : Name .
```

Encryption/Decryption Cancellation

```
eq pk(A,sk(A,Z)) = Z [nonexec] .
eq sk(A,pk(A,Z)) = Z [nonexec] .
```

An equation is oriented into a rewrite rule in which the lefthand side of the equation is reduced to the righthand side.

For the declaration of Yubikey protocol has not been needed, and do not need to know more about algebraic properties for this protocol.

3.4. Specifying the strands

Maude-NPA used strands for protocol specification and intruder knowladge, there are specificate in the module PROTOCOL-SPECIFICATION.

3.4.1. Dolev-Yao strands

The intruder knowledge uses the formal Dolev-Yao model[Dolev83] to prove properties, and the symbol & as the union operator for sets of strands. The intruder strands consists of a sequence of negative node, followed by possitive nodes. Also can consider that variables are not shared between strands, and thus will appropriately rename them when necessary.

For example, on the concatenation and deconcatenation if the intruder knows *X:StdMsg* and *Y:StdMsg*, he can obtain *X:StdMsg ; Y:StdMsg* . If he knows *X:StdMsg ; Y:StdMsg* he can obtain *X:StdMsg* and *Y:StdMsg*. Since each intruder strand can have at most one positive node, we need to use three strands to specify these actions:

```
eq STRANDS-DOLEVYAO =
:: nil :: [ nil | -(X:StdMsg), -(Y:StdMsg), +(X:StdMsg ; Y:StdMsg), nil ] &
:: nil :: [ nil | -(X:StdMsg ; Y:StdMsg), +(X:StdMsg), nil ] &
:: nil :: [ nil | -(X:StdMsg ; Y:StdMsg), +(Y:StdMsg), nil ]
….
...

[nonexec] .
```

Note, the notation *:StdMsg* is not necesary if we have declarated the variable X before.

In order to give to the intruder the ability to generate his own sort Counter, we would represent this using the following rule:

```
:: nil :: [ nil | +(counter1), nil ]
```

We need to provide the intruder strands for all the operation that are defined, unless one is explicitly making the assumption that the intruder can not perform the given operation.

3.4.2. Protocol Strands

In this part of a specification we define the messages that are sent and received by each of roles/participants.

We specify one strand per role, however, since Maude-NPA supports an arbitrary number of sessions, each strand (i.e., each role) can be instantiated an arbitrary number of times. We recall the informal specification of FirstExample used in Tamarin prover, as follows:

```
C -> S: aenc(k, pkS)
S -> C: h(k)
```

At the specification of the protocol strands it is important to remark do it from the point of view of the principal that controls the role. For example, in FirstExample the client C starts out by generating a fresh symmetric key k, encrypts it with the public key pkS of a server S (aenc stands for asymmetric encryption), and sends it to S. She get back confirming the key's receipt by sending the hash of the key back to the client.

In order to represent the initiator's strand, we model the construction of C's nonce explicitly as n(C,k), where r is a variable of sort Fresh belonging to C's strand.

```
eq STRANDS-PROTOCOL =

:: r ::
[ nil | +(aenc(k,pkS(C,S ; n(C,r)), - (h(k)), nil ]
&
:: ::
[ nil | -(aenc(k,pks(C,S ; N)), +(h(k)), nil ]
[nonexec] .
```

3.4.3. Protocol Analysis

The attacks patterns are defined in the last part of the module PROTOCOL-SPECIFICATION and for Maude-NPA performs a backwards reachability. In Maude-NPA, each state found during the backwards analysis (i.e., a backwards search) is represented in five different sections separated by the symbol ||. We will focus in two of them, first one: state Id, and second one: set of current protocol and intruder strands. In the state id will start with the number 0. A sort example of a type of attack pattern representing the intruder learning the nonce generated by client:

```
Eq ATTACK-STATE(0) =
:: r ::
[ nil | +(aenc(k,pkS(c,S ; n(c,r)), - (h(k)), nil ]
|| n(c,r) inI
|| nil
|| nil
|| nil
[nonexec] .
```

The intruder knowledge represents what messages the intruder knows (*symbol inI*) or does not yet know (*symbol !inI*) at each state of a protocol execution.

4. Yubikey Protocol

4.1. General information

The YubiKey is a small USB device manufactured by Yubico. The goal of this device is to authenticate a user against network-based services. It supports one-time passwords (OTPs) using an encryption of a secret value, that is, a running counter and some random values representing information encrypted using a AES-128 encryption. Also, it is indepent of the operating system and does not require any installation because it works with USB system drivers.

Yubikey's incremental success has led to its use in goverments, universities and companies like Google, Facebook, Dropbox, CERN, etc., including more than 30,000 customers. [Yubico07]

Due to its success, the Yubikey protocol has received little independent security analysis. And it is interesting to prove security properties of the Yubikey API for an unbounded number of fresh OTPs using the Maude-NPA tool.

All our analysis follow the Dolev-Yao model of cryptography, when we refer to the Yubikey or Yubikey protocol, we analysed the version 2.3 of the device*[Yubikey16]*.

4.2. Yubikey Authentification Protocol

The Yubikey authentication server accepts an OTP only if it decrypts under the correct AES key into a valid secret value containing a counter larger than the last accepted counter. The counter is, thus, used as a means to prevent replay attacks.

The Yubikey is connected to the computer via USB port, using the operating system's native drivers. We will focus on the Yubikey OTP mode, that has exactly one button. If the button is pressed, it emits a string that can be verified only once against a server in order to receive the permission to access a service. A request for a new authentication token is triggered by touching a button that is on the YubiKey device. As a result, some counters that are stored on the device are incremented and some random values are generated in order to create a fresh 16-byte plaintext having the following concatenated fields

- the unique secret ID (6 bytes)
- a session counter (2 byte)
- a timestamp (3 byte)
- a token counter (1 byte)
- a pseudo-random value (2 bytes)
- CRC-16 value (2 bytes)

The figure 2 [Vamanu12] represent the set of field explained before:

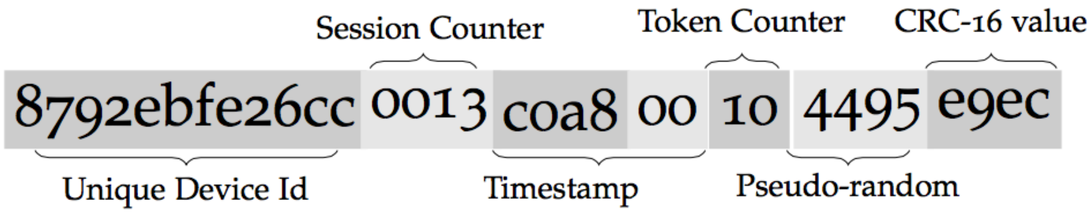


Figure 2 : Structure of the plantext of OTPs

The authentication protocol of the YubiKey involves three roles: (I)the user, (II)the service and (III)the verification server. The user can have access to the service if it provides its own valid one time password generated by the YubiKey; the validity is verified by the verification server. The figure 3 [Merkel09] is a simple example of the Yubikey protocol with three roles.

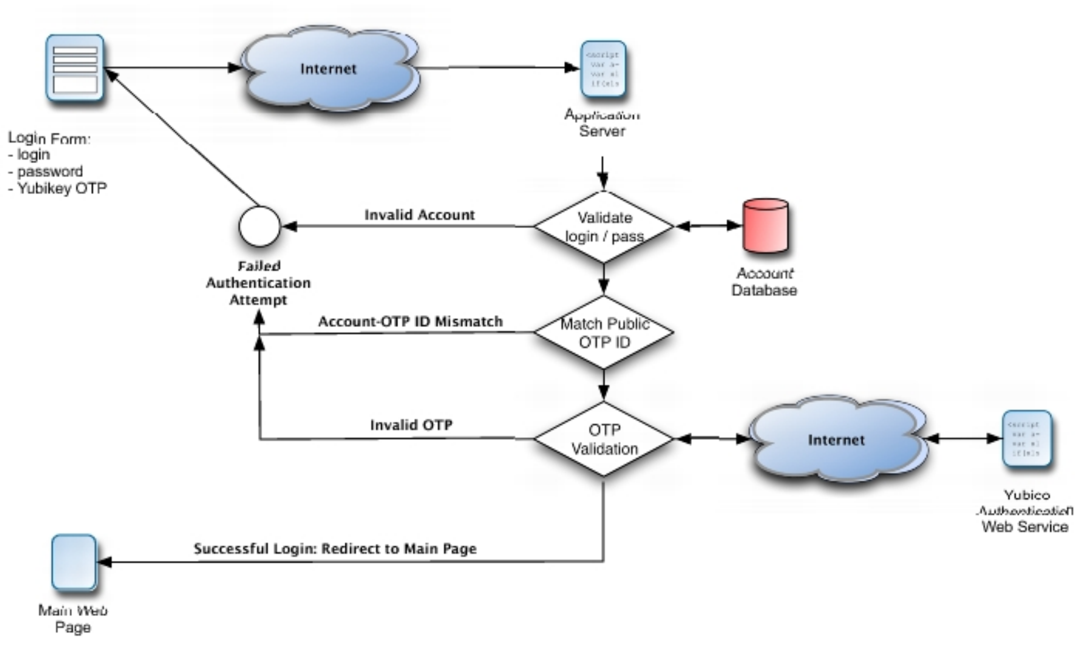


Figure 3 : Yubikey OTP Validation Flow

The authentication protocol consists of two exchanged messages. The first message is sent from the service to the verification server with its identification number *pid*, a *nonce* and the one time password (*OTP)* received from a user.

The second message is the response from the server after the verification of the password is made and consists of the password, the validity status, the nonce sent by the client and a HMAC (keyed-hash Message Authentication Code) over these fields using a shared key between the client and server.

A Yubikey stores a pid*(public id)*, *sid(secret id)*, and the AES k(key), and is used in the following authentication protocol:

The user provides a client C with the Yubikey's output *pid*, *otp*,e.g., by performing it in a web form.

C -> S : pid,otp,nonce

S -> C : otp,nonce,hmac(key$_{SC}$,otp,status,Nonce),status

Here *nonce* is a randomly chosen value between 8 and 20 bytes and *hmac* is a Message Authentication Code (MAC) over the parameters using a key present on the server and the client. By *status*, we denote additional status information given in the response, containing an error code that indicates either success or where the verification of the *otp* failed, plus (in case of success) the value of the internal timestamp, session counter and token counter when the key was pressed.

The server S accepts the token if either the session counter is bigger than the last one received, or the session counter has the same value but the token counter is incremented. It is possible to verify whether the timestamp is in a certain window with respect to the previous timestamp received, however, the model that we have specified in Maude-NPA does not include the timing of messages, therefore we ignore this (optional) check.

4.3. Formal Analysis

The only formal specification in the literature is given in the PhD thesis of Robert Künnemann[Künnemann14, Künnemann12]. In that paper, a specification of Yubikey in Tamarin is given. In the following, we describe such a specification.

The first rule is the initialisation of Yubikey, a fresh prublic ID(pid), secret ID(sid) and Yubikey-key(k) are saved on the server and the Yubikey.

Fr(k), Fr(pid), Fr(sid)
−[Protocol(), Init(pid, k), ExtendedInit(pid, sid, k)]→
!Y(pid,sid),Y_counter(pid, '1'),Server(pid,sid, '1'), !SharedKey(pid, k)

The Fr facts guarantee that each instantiation of this rule replaces k, pid and sid by different fresh names. Y_counter is the current counter value stored on the Yubikey.

The next rule models how the counter is increased when a Yubikey is plugged in

Y_counter(pid, otc), In(tc)
−[Yubi(pid, tc),Smaller(otc, tc)]→
Y_counter(pid, tc)

Note that the intruder has to input tc. By requiring the intruder to produce all counter values, we can ensure that they are in !K, i. e., the adversary's knowledge.

When the button is pressed, an encryption is output in addition to increasing the counter:

!Y(pid, sid), Y_counter(pid, tc), !SharedKey(pid, k), In(tc), Fr(npr), Fr(nonce)
−[YubiPress(pid, tc)]→
Y_counter(pid, tc +' 1'), Out(⟨pid, nonce, senc(⟨sid, tc, npr⟩, k)⟩)

The output can be used to authenticate with the server, in case that the counter inside the encryption is larger than the last counter stored on the server:

Server(pid, sid, otc), In(⟨pid, nonce, otp⟩), !SharedKey(pid, k), In(otc)
−[Login(pid, sid, tc, otp), LoginCounter(pid, otc, tc), Smaller(otc, tc)]→
Server(pid, sid, tc)

otp = senc(⟨sid,tc,pr⟩,k)

5. Yubikey protocol specified in Maude-NPA

The purpose of this master thesis is to prove that Yubikey can be specified in the Maude-NPA protocol analyzer and that the most interesting security properties can be verified. In this section we describe our efforts to specify the protocol in Maude-NPA and to analyze five interesting properties.

5.1. Protocol sysmbols

In the following we show how to specify the sorts, operations for Yubikey protocol in the Maude-NPA's syntax.

```
sorts Event End EventList StdMsg GlobalMsg Fr FrNonce EndEventList .
sorts Y Ycounter Server SharedKey Senc Init ExtendedInit Yubi Smaller YubiPress Login
        LoginCounter Counter .

subsorts EventList StdMsg GlobalMsg EndEventList < Msg .
subsort Event < EventList .
subsort FrNonce Y SharedKey Senc Counter < StdMsg .
subsorts Ycounter Server < GlobalMsg .
subsorts Init ExtendedInit Yubi Smaller YubiPress Login LoginCounter < Event .
subsort Counter < Public .

op Fr : Fresh -> FrNonce .
op Init : FrNonce FrNonce -> Init .
op ExtendedInit : FrNonce FrNonce FrNonce -> ExtendedInit .
op Y : FrNonce FrNonce -> Y .
op Ycounter : FrNonce Counter -> Ycounter .
op Server : FrNonce FrNonce Counter -> Server.
op Yubi : FrNonce Counter -> Yubi .
op Smaller : Counter Counter -> Smaller .
op 1 : -> Counter .
op _+_ : Counter Counter -> Counter [assoc comm] .
op SharedKey : FrNonce FrNonce -> SharedKey .
op _;_ : StdMsg StdMsg -> StdMsg [gather (e E)] .
op empty : -> GlobalMsg .
op _@_ : GlobalMsg GlobalMsg -> GlobalMsg [assoc comm id: empty] .
op senc : StdMsg FrNonce -> Senc .
op YubiPress : FrNonce Counter -> YubiPress .
op Login : FrNonce FrNonce Counter StdMsg -> Login .
op LoginCounter : FrNonce Counter Counter -> LoginCounter .
op yubikey : -> Role .
op nil : -> EventList .
op _++_ : EventList EventList -> EventList [assoc id: nil prec 31] .
op _+++_ : EventList End -> EndEventList [prec 33] .
op end : -> End .
```

We are going to focus on explaining few important operations, two of the most important for to make Yubikey protocol works, is the operator _++_ and _+++_ used for control temporal variables (These are the @ the action constraints in Tamarin). We decided to create a sort *End* to avoid a finite ejecution of the first strand and also, control the end of the Yubikey protocol.

The Yubikey protocol uses the time/cycle of execution to control the execution process. Maude-NPA does not have the ability to control them. For that we developed the way to simulate with composition a set of facts to control global memory. Thus, we made the operator _@_ to represent this global memory.

An example of the global memory operator:

```
{yubikey -> yubikey ;; 1-1 ;; Ycounter(pid:FrNonce,counter2:Counter) @
        mem:GlobalMsg},nil ]
```

We have three parts in this synchronization message, (I) *yubikey -> yubikey* is for represent how role send and how role recibe this strand, (II) to determine is a one to one message, (III) The *Event/s* has occurred before.

5.2. Strand specification

We follow the active Dolev-Yao intruder model[Dolev83], where the intruder has full control of the communicacion channel and can intercept messages, generate new messages, fake previously seen messages. In this model, all the intruder capabilities are specified using two kind of strands: composing (of the form -(X_1),…,-(X_n), +(F(X_1,...,X_n)) and decomposing (of the form -(F(X_1,...,X_n)), +(X_i))

```
eq STRANDS-DOLEVYAO =

:: nil :: [ nil | -(pid), -(sid), +(Y(pid,sid)), nil ] &
:: nil :: [ nil | -(pid), -(counter1), +(Ycounter(pid,counter1)), nil ] &
:: nil :: [ nil | +(counter1), nil ] &
:: nil :: [ nil | -(pid), -(sid), -(counter1), +(Server(pid,sid,counter1)), nil ] &
:: nil :: [ nil | -(key), -(pid), +(SharedKey(pid,key)), nil ] &
:: r  :: [ nil | +(Fr(r)), nil ]  &
:: nil :: [ nil | -(X:StdMsg), -(K:FrNonce), +(senc(X:StdMsg,K:FrNonce)), nil ] &
:: nil :: [ nil | -(K:FrNonce), -(senc(X:StdMsg,K:FrNonce)), +(X:StdMsg), nil ] &
:: nil :: [ nil | -(X:StdMsg), -(Y:StdMsg), +(X:StdMsg ; Y:StdMsg), nil ] &
:: nil :: [ nil | -(X:StdMsg ; Y:StdMsg), +(X:StdMsg), nil ] &
:: nil :: [ nil | -(X:StdMsg ; Y:StdMsg), +(Y:StdMsg), nil ]
[nonexec] .
```

The strands of this protocol are specified as follows:

```
eq STRANDS-PROTOCOL =

:: rk,rpid,rsid ::
[nil |
+(Init(Fr(rpid:Fresh),Fr(rk:Fresh)) ++
        ExtendedInit(Fr(rpid:Fresh),Fr(rsid:Fresh),Fr(rk:Fresh))),
+(Y(Fr(rpid:Fresh),Fr(rsid:Fresh))),

+(SharedKey(Fr(rpid:Fresh),Fr(rk:Fresh))),
{yubikey -> yubikey ;; 1-1 ;; Ycounter(Fr(rpid:Fresh), 1) @
Server(Fr(rpid:Fresh),Fr(rsid:Fresh),1)},nil ]
&
```

This first strand is the initialisation of a Yubikey. A set of three fresh, identifiers are used: public ID(rpid), secret ID (rsid) and Yubikey-key (rk). They are "created" and assigned by the server and the Yubikey. For each instantiation of this strand we have different Fresh names.

The Yubikey is identified by its public ID (rpid). The message 'Y' is sent to the comunication channel and it stores the corresponding secret ID (rsid). The message 'SharedKey' is also sent to the communication channel storing the corresponding key 'rk', which is shared with the Server. Both sent messages are used to associate the secret ID to both the Server and the Yubikey for further authentication. Ycounter is initialiazed in the global memory with the current counter value stored on the Yubikey.

```
:: nil ::
[nil | {yubikey -> yubikey ;; 1-1 ;;  Ycounter(pid:FrNonce,counter1:Counter) @
mem:GlobalMsg},
-(counter2:Counter),
-(EL:EventList),
+(EL:EventList ++ Yubi(pid:FrNonce,counter2:Counter) ++
        Smaller(counter1:Counter,counter2:Counter)),
{yubikey -> yubikey ;; 1-1 ;; Ycounter(pid:FrNonce,counter2:Counter) @
mem:GlobalMsg},nil ]
&
```

The second strand represents another Yubikey begins plugged. This strand checks that the new associated counter is smaller than any previous one by using the predicate *Smaller*. The protogol manages that this new counter is smaller the previous one, with the event *Smaller*. This is used to know whether Yubikey appeared before in the communication channel. Note that the adversary has to input *counter2*, we will explain this in the protocol analysis section.

```
:: rnpr,rnonce ::
[nil | {yubikey -> yubikey ;; 1-1 ;;  Ycounter(pid:FrNonce,counter2:Counter) @
mem:GlobalMsg},
-(Y(pid:FrNonce,sid:FrNonce)),
-(SharedKey(pid:FrNonce,key:FrNonce)),
-(counter2:Counter),
-(EL:EventList),
+(EL:EventList ++ YubiPress(pid:FrNonce,counter2:Counter)),
+(pid:FrNonce ; Fr(rnonce:Fresh) ; senc(sid:FrNonce ; counter2:Counter ;
        Fr(rnpr:Fresh),key:FrNonce)),
{yubikey -> yubikey ;; 1-1 ;; Ycounter(pid:FrNonce,counter2:Counter + 1) @
mem:GlobalMsg}, nil]
&
```

This third strand is follow-up of the first one. When the Yubikey button is pressed an encryption is sent (senc event). The whole sent message can be used to authenticate with the server. In addition to increasing the value counter:

```
:: nil ::
[nil | {yubikey -> yubikey ;; 1-1 ;;  Server(pid:FrNonce,sid:FrNonce,counter1:Counter) @
mem:GlobalMsg},
-(SharedKey(pid:FrNonce,key:FrNonce)),
-(pid:FrNonce ; nonce:FrNonce ; senc(sid:FrNonce ; counter2:Counter ; npr:FrNonce,
key:FrNonce)),
-(counter1:Counter),
-(EL:EventList),
+(EL:EventList ++ Login(pid:FrNonce,sid:FrNonce,counter2:Counter,senc(sid:FrNonce ;
        counter2:Counter ; npr:FrNonce, key:FrNonce)) ++
        LoginCounter(pid:FrNonce,counter1:Counter,counter2:Counter)  ++
        Smaller(counter1:Counter,counter2:Counter) +++ end), nil]
[nonexec] .
```

In the last part of the specification this strand is for when the server receives a petition of login and is acepted if the condition that the counter inside the encryption is larger than the last counter stored on the server.

5.3. Protocol Analysis

In this section we describe how we have analyzed the Yubikey protocol with Maude-NPA to prove the following verification properties.

The following four properties escribe in the PhD thesis of Robert Künnemann [Künnemann14] were proved. And the last attack verification is to obtain a regular execution of the Yubikey protocol.

5.3.1. The absence of replay attacks

$\neg(\exists i, j, pid, sid, x, otp1, otp2.$
$Login(pid, sid, x, otp1) @ i \wedge Login(pid, sid, x, otp2) @ j \wedge \neg(i = j)).$

The first property is about the absence of replay attacks, there are no two distinct logins that accept the same counter value.

```
eq ATTACK-STATE(1)
= empty
||
((EL0:EventList ++ Login(pid,sid,c6:Counter,X:StdMsg) ++ EL1:EventList ++
        Login(pid,sid,c6:Counter,Y:StdMsg) ++ EL2:EventList)
        inI, X:StdMsg != Y:StdMsg)
|| nil
|| nil
|| nil
 [nonexec] .
```

Note, the condition *X:StdMsg != Y:StdMsg* would not be necesary to validate the property, the condition helped to reduce the search space.

Execution result:

```
reduce in MAUDE-NPA : summary(1,0) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(1,1) .
result Summary: States>> 0 Solutions>> 0
```

The result means that cannot reach an initial state and has a finite search space, proving it secure.

5.3.2. Injective correspondence

Second property is an injective correspondence between pressing the button on a Yubikey and a successful login:

$$\begin{aligned}&\forall\, pid, sid, x, otp, t2.\\ &\quad Login(pid,sid,x,otp)@t2 \Rightarrow\\ &\qquad \exists t1.YubiPress(pid, x)@t1 \wedge t1 < t2\\ &\qquad \wedge \forall otp2, t3.Login(pid, sid, x, otp2)@t3 \Rightarrow t3 = t2\end{aligned}$$

A successful login must have been preceded by a button press for the same counter value. Furthermore, there is not second, distinct login for this counter value.

```
eq ATTACK-STATE(2)
= empty
||
((EL:EventList ++ YubiPress(pid:FrNonce,c4:Counter) ++ EL1:EventList ++
Login(pid:FrNonce,sid:FrNonce,c4:Counter,X:StdMsg)
         ++ EL2:EventList ++
        Login(pid:FrNonce,sid:FrNonce,c5:Counter,Y:StdMsg))
        inI, c4:Counter != c5:Counter)
|| nil
|| nil
|| nil
[nonexec] .
```

Execution result:

```
reduce in MAUDE-NPA : summary(1,0) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(1,1) .
result Summary: States>> 0 Solutions>> 0
```

The result means that cannot reach an initial state and has a finite search space, proving it secure.

5.3.3. Increasing the counter

For the third fact, the counter values associated to logins are increasing over the time, which implies that all successful logins have different counter value.

∀ pid, otc1, tc1, otc2, tc2, t1, t2, t3. Smaller(tc1, tc2)@t3
∧ LoginCounter(pid, otc1, tc1)@t1
∧ LoginCounter(pid, otc2, tc2)@t2
⇒ t1 < t2

```
eq ATTACK-STATE(3)
 = empty
||
((EL:EventList ++ LoginCounter(pid:FrNonce,c1:Counter,c2:Counter) ++
Smaller(c1:Counter,c2:Counter) ++ EL1:EventList ++
LoginCounter(pid:FrNonce,c3:Counter,c4:Counter) ++
Smaller(c3:Counter,c4:Counter) +++ end) inI)
|| nil
|| nil
|| nil
[nonexec] .
```

Execution result:

```
reduce in MAUDE-NPA : summary(3,0) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(3,1) .
result Summary: States>> 0 Solutions>> 0
```

The result means that cannot reach an initial state and has a finite search space, proving it secure.

5.3.4. Counter value control

Another property stronger than the previous property is controling the counter values are different over the time. Note, that Tamarin can use invariant and Maude-NPA not.

```
∀ pid, otc1, tc1, otc2, tc2, t1, t2. LoginCounter(pid, otc1, tc1)@t1
        ∧ LoginCounter(pid, otc2, tc2)@t2 ∧ t1 < t2
        ⇒ ∃z.tc2 = z+tc1
```

```
eq ATTACK-STATE(4)
= empty
||
((EL0:EventList ++ Login(pid,sid,c6:Counter,X:StdMsg) ++ EL1:EventList ++
Login(pid,sid,c6:Counter,Y:StdMsg) ++ EL2:EventList)
        inI, X:StdMsg != Y:StdMsg),
((EL5:EventList ++ YubiPress(pid,c5:Counter) ++ EL6:EventList ++
        Login(pid,sid,c5:Counter,X:StdMsg) ++ EL7:EventList) inI)
|| nil
|| nil
|| nil
[nonexec] .
```

Note, that the condition X:StdMsg != Y:StdMsg would not be necessary for the property.

Execution result:

```
reduce in MAUDE-NPA : summary(4,0) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(4,1) .
result Summary: States>> 14 Solutions>> 0

reduce in MAUDE-NPA : summary(4,2) .
result Summary: States>> 21 Solutions>> 0

reduce in MAUDE-NPA : summary(4,3) .
result Summary: States>> 28 Solutions>> 0

reduce in MAUDE-NPA : summary(4,4) .
result Summary: States>> 21 Solutions>> 0
```

```
reduce in MAUDE-NPA : summary(4,5) .
result Summary: States>> 7 Solutions>> 0

reduce in MAUDE-NPA : summary(4,6) .
result Summary: States>> 0 Solutions>> 0
```

The result means that cannot reach an initial state and has a finite search space, proving it secure.

5.3.5. Regular execution

Fort he last validate property we decided to make a regular execution of Yubikey. To determine if since the beginning you can reach the end.

```
eq ATTACK-STATE(5)
=  :: rk,rpid,rsid ::
[nil ,
+(Init(Fr(rpid),Fr(rk)) ++ ExtendedInit(Fr(rpid),Fr(rsid),Fr(rk))),
+(Y(Fr(rpid),Fr(rsid))),
+(SharedKey(Fr(rpid),Fr(rk))),
 {yubikey -> yubikey ;; 1-1 ;; Ycounter(Fr(rpid),1) @ Server(Fr(rpid),Fr(rsid),1)}| nil ]
&

:: rnpr,rnonce ::
[nil , {yubikey -> yubikey ;; 1-1 ;; Ycounter(Fr(rpid),1) @ Server(Fr(rpid),Fr(rsid),1)},
-(Y(Fr(rpid),Fr(rsid))),
-(SharedKey(Fr(rpid),Fr(rk))),
-(1),
-(Init(Fr(rpid),Fr(rk)) ++ ExtendedInit(Fr(rpid),Fr(rsid),Fr(rk))),
+(Init(Fr(rpid),Fr(rk)) ++ ExtendedInit(Fr(rpid),Fr(rsid),Fr(rk)) ++ YubiPress(Fr(rpid),1)),
+(Fr(rpid) ; Fr(rnonce) ; senc(Fr(rsid) ; 1 ; Fr(rnpr),Fr(rk))),
{yubikey -> yubikey ;; 1-1 ;; Ycounter(Fr(rpid),1+ 1) @ Server(Fr(rpid),Fr(rsid),1)} | nil]
&

:: nil ::
[nil , {yubikey -> yubikey ;; 1-1 ;; Server(Fr(rpid),Fr(rsid),1) @ Ycounter(Fr(rpid),1 +1)},
-(SharedKey(Fr(rpid),Fr(rk))),
-(Fr(rpid) ; Fr(rnonce) ; senc(Fr(rsid) ; 1 ; Fr(rnpr), Fr(rk))),
-(1),
-(Init(Fr(rpid),Fr(rk)) ++ ExtendedInit(Fr(rpid),Fr(rsid),Fr(rk)) ++ YubiPress(Fr(rpid),1)),
+(Init(Fr(rpid),Fr(rk)) ++ ExtendedInit(Fr(rpid),Fr(rsid),Fr(rk)) ++ YubiPress(Fr(rpid),1)
         ++ Login(Fr(rpid),Fr(rsid),1,senc(Fr(rsid) ; 1 ; Fr(rnpr), Fr(rk)))
++ LoginCounter(Fr(rpid),1,1) ++ Smaller(1,1) +++ end) | nil
|| empty
|| nill
|| nil
|| nil
[nonexec] .
```

Execution results:

```
reduce in MAUDE-NPA : summary(5,0) .
result Summary: States>> 1 Solutions>> 0

reduce in MAUDE-NPA : summary(5,1) .
result Summary: States>> 3 Solutions>> 0

reduce in MAUDE-NPA : summary(5,2) .
result Summary: States>> 9 Solutions>> 0

reduce in MAUDE-NPA : summary(5,3) .
result Summary: States>> 21 Solutions>> 0

reduce in MAUDE-NPA : summary(5,4) .
result Summary: States>> 46 Solutions>> 0

reduce in MAUDE-NPA : summary(5,5) .
result Summary: States>> 17 Solutions>> 0

reduce in MAUDE-NPA : summary(5,5) .
result Summary: States>> 3 Solutions>> 0

reduce in MAUDE-NPA : summary(5,5) .
result Summary: States>> 0 Solutions>> 1
```

For this attack pattern Maude-NPA finds an initial state, proving the regular execution of Yubikey protocol.

6. Conclusions and future work

The conclusions that we have drawn from this master thesis is that the Yubikey protocol can be specified and analyzed in Maude-NPA. The whole process has been challenging for us in many ways, specially specifying a high level protocol and controlling the order of execution of its facts thanks to strand composition.

During the process of specification and analysis we have found errors of strand composition and associativity within Maude-NPA, not related to the Yubikey protocol. However, one of our objectives was to test Maude-NPA and finding bugs was a good sign.

For future work, an interesting possibility will be to specify the YubiHSM protocol that uses exclusive-or, since Maude-NPA is one of the few tools able to work with exclusive-or. YubiHSM is also a USB device focused in server security. If the secrets stored on these servers are compromised, it can result in the compromise of all cryptographic keys and passwords resident on the server. This type of high security protocols would be a great challenge to be specified and analyzed in Maude-NPA.

7. Bibliography

[Dolev83] D. Dolev & A. Yao. On the security of public key protocols. IEEE Transaction on Information Theory, vol. 29, no. 2.1983

[Künnemann14] Künnnemann, Robert. *Foundations for analyzing security APIs in the symbolic and computa- tional model*. Cryptography and Security. École normale supérieure de Cachan - ENS Cachan, 2014.

[Künnemann12] Künnemann, Robert and Graham, Steel. *YubiSecure? Formal security analysis results for the Yubikey and YubiHSM*. International Workshop on Security and Trust Management. Springer, Berlin, Heidelberg, 2012.

[Maude16] Clavel, M., Durán, F., Eker, S. Escobar, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., & Talcott, C. (2016). *Maude manual* (version 2.7.1). SRI International, Menlo Park.

[MNPA17] Santiago Escobar, Catherine Meadows, José Meseguer. *Muade-NPA*. Version 3.0. 2017.

[Simon13] Meier, Simon, et al. *The TAMARIN prover for the symbolic analysis of security protocols*. International Conference on Computer Aided Verification. Springer, Berlin, Heidelberg, 2013.

[Simon16] Meier, Simon, Benedikt Schmidt. *Initial Example for the Tamarin Manual*. 2016. url: https://goo.gl/9rSrj7.

[Meier13] Meier, Simon. "*Advancing automated security protocol verification.*" *PhD diss., 2013.*

[Merkel09] Merkel, Dirk. *Yubikey*. Linux Journal 2009, no. 177 (2009): 1.

[Tamarin16] *Tamarin Prover Manual*, 2016. url: https://goo.gl/vu3UaL.

[Vamanu12] *L. Vamanu. Formal Analysis of Yubikey. Master's thesis, INRIA, 2012.*

[Yubico07] *Yubico Inc. Homepage,* 2007. url: https://www.yubico.com.

[Yubikey16] *The YubiKey Manual - Yubico Authenticator User's Guide.* YubicoAB. 2016. url: http://www.yubico.com/documentation.