LABORATOIRE SPÉCIFICATION ET VÉRIFICATION

MASTER THESIS LEVEL I

Formal Analysis of YubiKey

Student:
Loredana VAMANU

Supervisor:
Graham STEEL,
Research Fellow, INRIA

February 20, 2012

1 Introduction

An increasingly common solution for secure systems that are deployed in an insecure environment is the use of a tamper-proof hardware module or of a cryptographic coprocessor. There already exist several security tokens having the form of a smartcard or of an USB device, which are designed for protecting cryptographic values from an intruder. They are used for generating and storage of sensitive data, allowing a secure login for different applications e.g: emails, online banking, door entry, etc or to ease authentication for the authorized users of a service.

There exist tokens that need to be able to communicate with the outside world, for example to be able to permit for the keys stored on the device to be used by cryptographic functions and also for the keys to be updated. The communication with the outside world is made through an API. However,an interface that does not introduce flaws into the security of the token is hard to design, because it also needs to make sure that any malicious software installed can not succeed to gain access to the secrets from the token. There already exist proven vulnerable interfaces. One example is *Cryptoki*, the API described by the most commonly used standard for tokens: the *RSA PKCS#11*. In a recent paper [5], M. Bortolozzo, M. Centenaro, R. Focardi and G. Steel proposed an automated tool *Tookan* for analysing devices that use the *Cryptoki* interface. It uses reverse engineering to find out the functionalities a device implements and then creates a formal model that is provided to a model checker in order to find attacks. They found attacks on a variety of tokens that are used by many people. Mostly the attacks were found on tokens that offer multiple functionalities.

Other tokens are designed such that it is needed a hardware stimulus in order to trigger a new use. This is the case of the YubiKey token, a small USB device which has as purpose the authentication of the user against some service. It has four use cases: to generate standard one time passwords, to store a static password, to be used with third party OATH servers or for use in a challenge-response protocol. The device is equipped with a button that once touched it triggers a new authentication attempt. Though, the YubiKey is used by more than 15.000 users, from what we know, it has not been formally analysed yet. In this paper we try to model the YubiKey protocol for the use case of generating one time passwords and prove that it is resilient against replay attacks. Then we show how adding the use of one time passwords from a YubiKey can secure some previously known insecure protocols.

We tried first to formalize the use of the YubiKey for one time passwords using Scyther [3], a recent symbolic analysis tool that intends to fill the gap between the computational and symbolic approaches in analysing security protocols. The novelty of this tool is the possibility to model an adversary more powerful than a Dolev-Yao adversary type. The difference from the usual model of an intruder is that the adversary in Scyther is able to compromise honest agents during the run of the protocol, being capable of learning the short term secrets(e.g: random numbers, session key) but also the long term secrets (e.g: secret key) of an agent. Modeling the adversary with this different capabilities makes it possible to find attacks that were only detected using the computational approach. Unfortunately, we were not able to model the YubiKey protocol in Scyther; it is more appropriate for small protocols, because in the current version is not possible to model a branching rule or to simulate a loop.

For our second approach we considered to model the protocol in AIF [10], a specification

language that was proposed in 2010 by S.A. M ödersheim. It was introduced to extend the scope of the over-approximation of protocols by a set of Horn clauses such that to be possible to apply it also for protocols that use databases and in which is allowed to revoke some terms, which means that the set of true facts does not grow monotonically with state transition. He proposed a new approach to abstract fresh data by their status and membership in the databases of the participants and permit for the membership to be revocable, while the facts grow monotonically with the transition of the database. It is also provided a translation tool that takes the abstraction in AIF and transforms it into a set of Horn clauses that can be verified using the tools SPASS and ProVerif, which implement resolution techniques for first order Horn clauses. The YubiKey protocol for generating one time passwords seemed to be a perfect candidate to be abstracted in AIF, because the one time password can be represented as a term that is in a state valid before sending it to the server and then it changes into a state used, meaning it can not be validated again. We succeed to obtain an approximative abstraction of the protocol and the result of the verification was that the protocol is resilient against replay attacks.

Organization. The paper is organized as follows. We present the protocol of the YubiKey in Section 2. In Section 3 and Section 4 we briefly present the two tools considered and then discuss about the model we proposed for the YubiKey and also we present the difficulties we encountered. We conclude in Section 5 with a summary about the expressiveness of the two tools.

2 YubiKey

2.1 General description

The *YubiKey* [6] is a small device that is designed to authenticate the user against networks or services, in special web services where the use of a login id and password are not safe enough. It can be connected to a computer through the USB port and it does not need external power supply, receiving power from the computer. Also, it is independent of the operating system and it does not require any additional driver installation because it uses the driver for an USB keyboard, emulating one.

The 2.0 version of the YubiKey has two slots, two separate credentials, such that from the four uses cases it can be used in the same time to store for example a 128 bits long *static password* or to generate *one time passwords*($OTPs$). We analyse the use case for generating OTPs.

The YubiKey has been widely adopted by companies and universities [2]. The most important companies are: Microsoft which uses it to increase security for cloud platform Windows Azure and indirectly Google, which offers to the clients of Google Apps for Business the possibility to login securely using this device [**?**].

Despite its popularity, there is no previous formal analysis of the protocol of the YubiKey as far as we know. Our main goal is to provide a formal model and use a verification tool to prove that the protocol used by the YubiKey is secure against replay attacks, an important security property that a device that is used to generate one time passwords should assure.

First, we give an informal description of the protocol and of the properties the YubiKey is expected to have. Then, we present the models we were able to create.

2.2 Protocol

The authentication protocol of the YubiKey involves three actors: the user, the service and 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 authentication protocol consists of two messages exchanged. The first message is sent from the service to the verification server with its identification number, a nonce and the one time password 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 over these fields using a shared key between the client and server.

The user provides the client C with an OTP_u by filling in a field of a form with it
$C \rightarrow S : id_C, Nonce, OTP_u$
$S \rightarrow C : OTP_u, status, Nonce, hmac(K_{CS}, OTP_u, status, Nonce)$

Figure 1: The YubiKey protocol

In the next sections we detail the two phases, the generation of the password and the verification process.

2.2.1 Generating OTPs using YubiKey

The *one time passwords* or the OTPs are a *32 character passwords* 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:

- **Unique Device ID:** 6 bytes.
- **Session Counter:** 2 bytes. It is a counter initialized to 1 at first and incremented at each plug in of the device if it is used.
- **Timestamp:** 3 bytes. At each plug in it is set to a random value and afterwards it is incremented by a clock in order to track the period of time the device was connected during a session use.
- **Session Use:** 1 byte. It is the second counter and it stores the number of authenticated tokens generated during a session.
- **Pseudo-random:** 2 bytes. Field used to store a random number generated using as seed the touch button sensor USB activity.

- **CRC-16-value:** 2 bytes. Stores the checksum computed over the entire token excluding the CRC field.

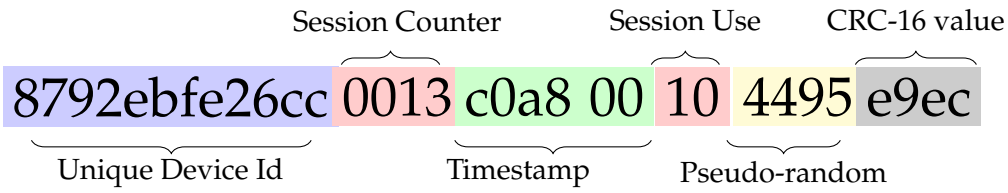

Figure 2: *Structure of the plaintext of OTPs* (session: 19, session use: 16)

The use of the two counters, timer and the pseudo-random field ensure that each plain text is *unique*. After the generation, the plaintext is encrypted following the *AES algorithm* with a *unique secret key* that is stored on the device but also known by the server and the result is encoded in a *Modified Hexadecimal coding, Modhex*. This coding uses only the characters that have the same position in all keyboard layouts in order to make the device independent of the language settings of the operating system. The result is a 32 character long password that can be sent to be verified against the server.

2.2.2 Verification of an OTP

The verification server has two databases in which it stores information about the YubiKey users and the services that are registered for verification. For each YubiKey device it has an entry in the $YubiKeys$ table with the shared secret key, the values of the fields of the last received OTP and the value of its own clock when it received the last password. The database of the services is used to store a unique id of the service and a secret shared key that is used for signing the challenged password concatenated with its validity status. The key shared between the service and the verification server is obviously different from the secret key shared by the YubiKey device and the server. The service has no information about the YubiKey's configuration. The user sends the OTP generated by a YubiKey device to the service which forwards it to the verification server.

Each dynamically generated OTP, as above, can be concatenated with an optional static public id of a maximum length of 128 bits which can be used to extract the appropriate AES key from the server's database to decrypt the password.

The verification of an OTP is performed by comparing the received values with the ones stored in the database, after the decryption was made. The following cases are possible:

- **Valid otp**, database is updated with the new values if:
  - the session counter has a bigger value that the last one received
  - the session counter has the same value, but the session use counter is incremented
- **Invalid otp**, if:
  - the unique device id is incorrect
  - the session counter is smaller than the value stored

– the session counter equals the value stored, but the session use counter is smaller

In order to prevent a *phishing attack* of the form in which the intruder asks the user for an OTP which he will use later to get access to the service, then the time elapsed between two OTP generations can be verified. This is possible due to the fact that the server remembers its own clock value at the moment he received a new valid OTP from the user and the database contains the timestamp value of the last valid OTP. Comparing the two differences of time it can be prevented an attack of this form. In our model we ignore this additional verification step and we consider the value of the counters only.

After the verification is performed, the server sends to the service the OTP, the result, the nonce received together with a *HMAC* of these fields generated using the secret shared key with the client. If the OTP is valid, then the client will grant access to the user.

2.3 Security properties

In this section, we summarise the security properties the YubiKey guarantees and the known flaws of the protocol.

2.3.1 Cryptographic security

The plaintext of the OTP is encrypted with a 128 bit AES-key, which means that an attacker has 2^{128} possibilities from which to guess the key. Even though the plaintext is easy to recognise a brute force attack over the space of keys it is not possible with today's technology.

2.3.2 Replay attacks

The protocol used by the YubiKey for authenticating an user is is supposed to be resilient to replay attacks. This is due to the fact that at each use the counters that form the OTP are incremented and passwords with lower values than the last seen are not accepted by the server. The form of the OTP makes the verification protocol efficient being only necessarily to store the last received values for the fields of an OTP and not being required to search in a big database with all the values ever used, which would introduce an unnecessary overload.

2.3.3 Phishing attacks

The YubiKey is known to be insecure against *man-in-the-middle* or *real-time phishing* attacks. If the attacker tricks the user into giving him a couple of OTPs, he is able to get access to the service. One solution to lower the success probability of this form of attack is to use an additional username and a password for authentication besides the OTP. Another important fact that should be consider is that the attacker can mount a successful attack using an OTP that he succeded to extract from the user only until the next successful authentication of the same user to the service. In this case the OTPs that the attacker possess are invalidated because one of the counters values of each OTP has a smaller value than what the server stored as the last counter value for that $YubiKey$ device. Mutual authentication is considered to be

one solution against real-time phishing attacks but the providers of the YubiKey considered that it could introduce other forms of attacks.

2.3.4 Extracting the secret symmetric key or OTPs from the YubiKey

The chances for an intruder to succeed in extracting the secret key or OTPs from a YubiKey device are small. This is due to the fact that there is no software method through which to get the key or OTPs from the device, furthermore the generation of OTPs is triggered by touching the button on the device therefore for the intruder to get some OTPs besides the ones that the user intended, he should have physical access to the device. Chances can be increased if a conspiring manufacter introduces a hardware flaw that can be exploited by the intruder to extract information from the device, but still again the probability for this to happen is extremely small.

2.3.5 Conclusion of YubiKey's security

In addition to the small flaws presented above: *possible real-time phishing attacks* or the attack based on a *hardware flaw*, the YubiKey can not be secure also against *hardware attack*, the case in which the intruder gains physical access to the server where all the keys and values are stored. Against a possible dishonest access to the YubiKey, a password can be added to it and requested in the case of an update of the secret key is made. Instead, the YubiKey is thought to be *secure against replay attacks* due to the form of the OTP, but there is no formal proof of this property. Our main goal is to prove using an automated verification tool that it has this important property and then to use the protocol of YubiKey in order to make other protocols secure against replay attacks.

3 Analysis 1: Scyther

The first tool we considered to use to model and verify the protocol was $Scyther$ [8], a symbolic verification tool for security protocols developed by C. Cremers. We were motivated to use it due to the work of D. Basin and C. Cremers in [3] where they introduce a refined model of adversaries that are able to compromise honest agents during the protocol run, fact that permitted to discover new, unreported attacks on protocols. The tool can verify unbounded session security protocols and it guarantees that when an attack is found, certainly there is an attack in the concrete model because it does not use an approximation method. Another characteristic of this tool is that it always terminates. This is due to the fact that if a result can not be obtained (because of the undecidability of this problem) then the verification is made over a bounded number of sessions. It differentiates itself from the other automated tools being able to model properties such as weak forward secrecy, key impersonation and adversaries that can reveal the local state or can compromise the random numbers generated by the participants in the protocol. It is the first symbolic protocol analysis tool that considers adversaries with different capabilities, a property that was previously only available in the computational model.

3.1 Structure of Scyther

The modelling language of Scyther is named *Security Protocol Description Language* and is represented by a limited grammar. A security protocol is seen as a set of roles, therefore naturally there exists a structure $protocol$ that can be composed from multiple $role$ structures that are used to model what actions one type of agent is able to do.

The security properties that are intended to be verified are expressed in the form of a $claim$; there are some predefined properties: $Secret$, requires a term which secrecy is verified, $Nisynch$ refers to non-injective synchronization, $Niagree$ refers to non-injective agreement, $Reachable$ checks whether the claim is reachable, if there exists a trace such that the claim occurs and the last property is $Empty$ which is intended to be used when $Scyther$ is used together with other verification tools. Assuming that a role in a protocol has the requirement *claim(I, Nisynch)*, then the claim evaluates to true if in its trace *there exist actual send and read events s and r, such that the following three conditions for correct communications hold: the order of the events must be correct, the events are instantiations of the right role events by the runs of honest partners, and the message must be communicated correctly*(definition considered by C. Cremers from [7]). In the case of non-injective agreement the conditions are the same except from the restriction for the order of $send$ and $recv$ events which is dropped. In this case the two agents need to have appropriate coupling events with the correct messages.

3.2 Restrictions in using Scyther

The Scyther tool, though it provides unique properties it is limited in expressiveness. For example, the equational theory is not fully supported and in order to use it the user should use a trick which consists of specifying the desired rules for a function as an exchange of messages from a form of the function to another form. This additional messages exchange form the role of a fake agent and all roles like this that are introduced for the support of the

equational theory are organized in a protocol that can be used by any agent that takes part in the main protocol, therefore it is also accessible to the intruder.

For example, in the Diffie-Hellman key exchange protocol there is the need to specify that multiplication is commutative. In this protocol two agents want to establish a secret shared key by exchanging information over a public channel. To achieve this goal, the initiator sends a prime number p and a primitive root of it g in clear and then the responder generates a random number x that remains secret and sends back g^x mod p. In the next step the initiator generates its secret random number y, sends g^y mod p and he is able to calculate the shared key as $(g^x)^y$ mod p. The responder after receiving the message is also able to deduce the key as $(g^y)^x$ mod p.

In order for the verification to be correct, the two deduced shared keys should be equal which happens only if the rule $(g^y)^x$ *mod* $p = (g^x)^y$ *mod* p is introduced. As mentioned above this property is specified in Scyther as a helping protocol that can be used by anyone and consists of a role in which *the agent* first expects a message of the form $(g^y)^x$ and sends a message with the two exponents commuted $(g^x)^y$. Below it is the code in $SPDL$ extracted from the model that C. Cremers and D. Basin propose in their paper.

```
protocol @exponentiation(RA)
{
        role RA
        {
                var T1,T2: Ticket;
                recv_!1(RA,RA, g2(g1(T1),T2));
                send_!2(RA,RA, g2(g1(T2),T1));
        }
}
```

Figure 3: Simulating $(g^x)^y = (g^y)^x$ in SPDL

The use of Scyther is limited also because the actions of each agent are described as a sequence of events. Therefore, when a protocol is executed the actions described by the roles are made in the exact order meaning the length of the protocol is given by the number of actions that can be made. Moreover, there is no primitive to use for branching or loops [4] that can appear in a protocol. The case of branching is handled implicitly by rejecting the messages that do not match the expected pattern. For certain protocols this solution may not be enough, which is also the case of the YubiKey protocol. In the verification step, the server should be able to take different actions based on the validity of the OTP received: if the counter values are smaller than the last values received, the server sends a response with the status $replayed$ and it does not update the database for that certain device, otherwise it sends the status ok and it updates the new values of the counters.

Also, the fact that it is not possible to model loops made it impossible to model the YubiKey using this formalism. If we were interested to analyse properties such as secrecy then the protocol could have been modelled as a simple protocol with the exchange of two messages. But we intend to prove that the YubiKey protocol is secure against replay attack, therefore the validation method used by the server in this protocol plays an important role.

One of the problems is that in this case the server and the user should be able to remember the values of the counters which influences the future content of the messages exchanged. We considered that the protocol should have multiple rounds, one round standing for one request and response of the verification of an OTP and from one round to another informations related to the value of the counters is transferred. Due to this property of the protocol, it means that different instances of the same user of the YubiKey can not be independent. They share information about counters, a fact that makes the analysis of this protocol more complicated than for most authentication protocols.

We tried different workarounds making use of the helping protocol, but the limitations of Scyther, some of them were described above, made impossible to model the YubiKey protocol for the OTP use case.

3.3 Key Exchange Protocols analysed in the presence of compromising adversaries

Verifying protocols using the Scyther tool can lead to interesting results, finding attacks that were not possible in the case of an Dolev-Yao adversary. The first article that presented this extended version of the Scyther tool is *Modeling and Analyzing Security in the Presence of Compromising Adversaries* [3] by D. Basin and C. Cremers in which they study key exchange protocols. While testing the models that they propose, for some protocols we found small modelling differences from the actual protocol, but which lead to the existence of certain attacks. Also, in the case of one version of a protocol we were expecting for the tool to find an attack, but due to the form of one of the adversary-compromise rule it could not be found. Before explaining in detail this differences we introduce briefly how Scyther tool works.

3.3.1 Adversary model

The model of adversary in the Scyther tool is studied from three perspectives: the type of data that he can compromise, the type of agents that can become dishonest and the interval when the compromise can be made. An adversary may compromise *long term secrets*, keys but also *short term secrets*, for example random values generated, session keys or he can do in certain conditions state revealing, which means he gains knowledge of the most recent local state of an agent. There are three categories of agents: the agent for whom the security properties are verified named $(actor)$, the agents that are involved in the communication with the $actor$ and the others agents. The adversary can compromise the information of agents before, during or after the actor takes part in the protocol.

Almost all the key exchange protocols considered were not secure against adversaries that are able to do *reveal the state* of honest agents. The adversary is capable of discovering the local state of agents that are different from the $actor$ and from agents that are $partners$ with the actor. The $partners$ are the agents that have matching history with the actor, which means that for each event*(send/receive)* of the actor there exits a corresponding event *(receive/send)* event with the same message sent by the actor and in plus the order of the events is preserved.

3.3.2 Remarks on the models

In this section, we discuss the small differences we consider to exist between the real protocol and the model proposed in the paper and one curious result.

The curious result was obtained for the Diffie-Hellman protocol, that was presented briefly above. They modeled the simplest version of the protocol which consists of two messages exchanged but also another version in which a third message for confirmation is added. An attack was found for the second version but not for the first one which seems to be more vulnerable to attacks. Therefore, the confirmation message actually in this model seems to introduce a flaw to the protocol; a fact that is counter-intuitive.

Diffie-Hellman key exchange protocol	
DH-2msg	DH-3msg
$A \rightarrow B$: $g^x, B, \{g^x, B\}_{sk(A)}$	$A \rightarrow B$: $g^x, B, \{g^x, B\}_{sk(A)}$
$B \rightarrow A$: $g^x, g^y, \mathbf{i}, A, \{g^x, g^y, \mathbf{i}, A\}_{sk(B)}$	$B \rightarrow A$: $g^x, g^y, \mathbf{i}, A, \{g^x, g^y, \mathbf{i}, A\}_{sk(B)}$
	$A \rightarrow B$: $g^x, g^y, \mathbf{i}, B, \{g^x, g^y, \mathbf{i}, B\}_{sk(A)}$

To understand why this result appeared we must describe the adversary that was considered for the verification. The adversary is able to compromise the long term key of the agents that are not communicating with the $actor$ at any time and the ones of the actor and its partner after the protocol is finished. He can also extract the session key and the most recent state of the agents not involved with the $actor$. The adversary can reveal the state of the suitable agents only in the case that the agents have not reach the end of the protocol. This is due to the fact that state revealing means to be able to compromise the most recent information that the agent added in his memory and when the agent finishes the protocol, is believed that it deletes everything from the memory so the information cannot be accessed.

The exchanged key can be discovered by the adversary in the particular case in which the same agent is in the role of the initiator and of the responder and the $actor$ is the responder. The attack can be mounted because in this case the second and the third message are the same in the DH-3msg protocol. Therefore, the adversary can wait for the responder to send the second message and he can replay the message back to him making him to reach end the protocol believing he shared successfully the key. Reaching the end of the protocol, the adversary is able to compromise the long term key of the $actor$ which he can use to impersonate the $actor$ into the initiator run of the protocol. He changes the second message sent by the responder by replacing the index i with a random number Ni and then he can create the signature over the new values, because he knows the $sk(A)$. When the initiator receives this new message, he is able to deduce the same key as the responder because the index i in their model is not part of key. The adversary is allowed to do a state revealing of the initiator, because the initiator even though started the protocol with the $actor$ as a partner, he became part of the agents that can be compromised because the message he received is different from the same message the $actor$ sent in the second step of the protocol. Therefore, the intruder can compromise the shared key.

However, no attacks were found while verifying the DH-2msg protocol with Scyther when considering the same type of adversary even though the same technique seems to be working. The adversary is able to compromise the long term secret key after the responder

finishes its part in the protocol and then to alter the message that the initiator receives in order to disqualify him as a partner of the actor. The problem is that the adversary is able to do a state revealing only in the case that the agent still has actions that he has to do to complete its role in the protocol. Therefore, in the case of the 2-message protocol the adversary is not permitted to reveal the state of the initiator. We argue that even in this case it should be possible to reveal the local state, because of the use of the protocol. The goal is to share a key between two parties such that it can be used later to encrypt the communication between the two parties which means that in most cases the key is kept in memory in order to be available to encrypt the messages. Adding a mock message to the DH-2msg protocol the expected attack is found.

Surprisingly we found differences between some models and the actual protocols. This is the case for the DH-3msg protocol, but also for another variant of Diffie-Hellman key exchange protocol known as DHKE-1. In both cases, the shared key is calculated after a different formula than what is suggested by the protocols. Moreover, in the case of the DHKE-1 there were two articles given as references, but the variant chosen to model did not meet any variant presented in the articles. We modeled the protocol following the articles and in fact these models were secure against the adversary for which the two authors reported that they found a new, unreported attack.

In the case of the DH-3msg the shared key should be obtained as the result of the hash function $h_i(g^{xy})$, instead the key is considered g^{xy}, making the index i without a purpose in the protocol. If it is considered that the last local state of the initiator in the moment he is able to deduce the key(in the second step of the protocol) only consists of the key of the form $h_i(g^{xy})$, then the attack that was described can not be mounted any more.

These differences that we discovered made us get in depth into the cases in which a protocol is vulnerable to a state revealing attack. A flaw that can be exploited is messages containing information that does not play an important role in obtaining the key. Obviously, the method used to obtain the key from the information that the two parties exchange plays an important role to the security of the protocol.

4 Analysis 2: AIF

The second approach that we considered for modelling the YubiKey protocol is AIF, a variant of AVISPA Intermediate Format. It is a convenient specification language that was recently proposed by S.A. Mödersheim [10] for abstraction and over-approximation of protocols and web services that depend on databases and for which revocation of certain terms is allowed. There already existed a popular method for abstracting protocols which is by a set of Horn clauses, but the approaches had limitations. An important one is the impossibility to model protocols for which the set of true facts grows non-monotonically with the execution of the protocol, because in the case of Horn clauses deduction is monotonic. S.A. Mödersheim proposed a solution for this limitation by defining a different abstraction of fresh data but preserving as a central method the Horn clauses approach.

The YubiKey protocol seems to be a perfect candidate for which to use the abstraction proposed, due to the counter values that are stored by the server in a database and once an OTP is accepted for an user all the OPTs that have a smaller values should be invalidated. It

needs to be possible to deduce less after one valid OTP is received. We succeeded in making an approximative abstraction of the protocol and to express the property for security against replay attacks in AIF and then using the translator from AIF to Horn clauses, also realized by S.A. Mödersheim, we verified the abstraction using the theorem prover $SPASS$ and the verifier $ProVerif$. We discuss the abstraction in subsection 4.2, but first we describe briefly the approach of S.A. Mödersheim regarding the AIF language.

We mention that we will use throughout the paper as an example the Diffie-Hellman protocol and we will consider that each agent has a database in which he remembers the exponent created by himself and another database for the exponents received from the other agents with whom he shared a key. We motivate that this abstraction is necessary in the case the verification is made for replay attacks, in order to be able to identify if an old message is replayed by the intruder making an honest agent believe it is a fresh one. In order for the intruder to successfully share a key with the agent it is also necessary for him to find the secret exponent from the message. It is believed that this attack is possible in case the protocol keeps a history on the computer [9].

4.1 Using AIF for specification

The general model on which the reasoning is based is one in which the participants maintain a database for storing freshly generated data(e.g: nonces, keys, etc) together with the status or any important information related to it(e.g: the agent with whom it is shared). The abstraction of all created data is defined by the status and the membership in the database of the participants, such that two terms are mapped to the same abstract term if and only if they belong to the same participant and have the same role. This approach is different from the usual because the infinite set of data is mapped to finitely many equivalence classes, but they depend on the state of the database.

For example, in the Diffie-Hellman abstraction described above, each agent a has two databases to save the exponents, let's call them: $created(a)$ and $received(a)$. Each exponent can be in two states, $current$ if it is part of a shared key that is still in use or $used$ if it was part of a key that became invalid. Though, the first intuition is to represent a currently used exponent created by the agent a to share a key with b as a registration $(x, b, current) \in created(a)$, this representation is not in correspondence with the abstraction of fresh data introduced above. S.A. Mödersheim suggested that for each status to be considered a different database such that the representation becomes $x \in created(a, b, current)$; this way each database is considered to be an equivalence class that contains only data of the same type such that it is not possible to distinguish between the data that belongs to the same database. In the abstraction the database is seen as a set.

AIF specifies a set of rules that defines a *state transition system*. A state is represented by a set of $facts$ and so called *positive set conditions* which are conditions of the form $t \in s$ where t is a term which belongs to a ground message term set s. The transition rule from a state to another is of a form of implication where the left-hand side conditions describe the states to which the rule can be applied and the right side of the implication describes the changes that are made to the state after the transition.

Another difference from other approaches is that in AIF the facts are $persistent$ which means that if a fact holds in one state then it remains true in all the successor states. But as

mentioned earlier, for modeling web services or protocols with databases it is necessary to be able to implement a way to invalidate some deductions, this is realized in AIF by being able to remove from a state the membership to certain sets and then make the rules such that are closely related to the membership in the sets. The membership removal from a certain set is made by creating a transition rule in which in the left side of the rule exists a *positive condition*, not repeated in the right-side of the implication.

To illustrate the idea with Diffie-Hellman protocol, we model below the rule to invalidate a shared key, which means to take the exponents from both $created$ and $received$ sets from the $current$ status to $used$.

```
X in created(a,b,current).
Y in created(b,a,current).
X in received(b,a,current).
Y in received(a,b,current)
=>
X in created(a,b,used).
Y in created(b,a,used).
X in received(b,a,used).
Y in received(a,b,used);
```

Figure 4: AIF rule to invalidate the exponents in Diffie-Hellman protocol

The fact that the construction in AIF let the set of $facts$ to grow monotonically over transitions and have set membership revocable made it possible to implement a translation from the state transition system in AIF to a set of Horn clauses that can then be verified with already existing tools. The problem with having an abstraction in which the facts are persistent is that it creates the possibility for the participants in the protocol to make an action any number of times, even if in the real system it is can not be done(for example the messages contain timestamps). Therefore, an analysis of a protocol against replay attacks should be made with special care.

Also, this approach is an over-approximation of the protocol which means that the verification can give false attacks; a correct protocol can be found as faulty. This problem can be solved if the attacks found are checked by hand and in the case they are proven to be false then the abstraction should be refined in order to simulate better the protocol.

4.2 Set-Based Abstraction

In this section we detail the set-based abstraction, abstracting the fresh data according to their membership, and the translation from a state dependent abstraction to a state independent abstraction using Horn clauses.

First, the so called positive set membership rules that have as conditions $t \in s$ where s is a ground message term set are transformed for a state S into encodings of the form $val(b_1, ..., b_N)$ where $b_i = 1$ if and only if there exist a condition $(t \in s_i) \in S$. For example the fact that the exponent X created by a was also received by b: $Xincreated(a, b, current).Xinreceived(b, a, current)$ is abstracted to $val(1, 0, 1, 0, 0, 0, 0, 0)$ for a total order on the sets given by the order of the

conditions in the rule.

Second, to prepare the translation from the standard transition rules that use the real sets to implication rules it was needed to cope with the transformation of facts that appears due to the state transition of the database. It was introduced a new kind of rule named *term implication rule* that has the form $\phi \rightarrow x \twoheadrightarrow x'$, which stands for if the clauses in ϕ are true then all facts defined for x become true also for x'. In other words, if the transformation of x is from a currently in use state to used x' , then all facts that the intruder knew about the current exponent x he also knows about the used exponent x', which means that the facts for x remain but the same facts are added with x replaced by x'. Further, the *term implication rules* of the form $s \twoheadrightarrow t$ have to be encoded into Horn clauses. It appears that the transformation yields an infinite number of Horn clauses, because $s \twoheadrightarrow t$ stands for $C[s] \Rightarrow C[t]$ for any context C, but it is proposed to introduce two binary fact symbols with the help of which it is possible to introduce some implications such that the infinite enumeration is limited to only to the contexts that can be instantiated to a fact that can be derived.

The method proposed to abstract data based on the membership of it to the databases and the encoding of the updating of the databases into *term implication rule* is an elegant solution for extending the use of the over-approximation protocols by a set if Horn clauses. Also, the AIF language is easy to use for specification and to understand because it uses syntactic sugar. For example, it is possible to write a rule having as parameters predefined sets of so called *enumeration variables*, which means that a rule of that type will be introduced for each constant from the set. It is also available the use of universal quantification of the variables in the set in negative set conditions.

4.3 Abstracting the YubiKey protocol with AIF

In this section we present our approaches on abstracting the YubiKey protocols and we discuss the problems we encountered while trying to refine our abstraction.

We made some assumptions regarding the protocol. First is regarding the symmetric key , we assume that it was previously safely shared between the server and the other agents. Secondly, we did not include the updating of the key which is possible for the YubiKey, because it is made using a special software tool that is downloaded on the local machine and then is shared with the server over a secured connection with https. We want to verify the protocol with the assumptions that the symmetric keys are secret.

The simplest model for the YubiKey is the one in which the OTP is abstracted as a fresh nonce encrypted with the secret symmetric key. In this case it is needed for the server to remember for each participant all the nonces received to be able to verify the $OTPs$ for freshness. Not only the verification process is more complicated in this abstraction than in the real system, but it does not model one unique property of the YubiKey related to the case in which the same YubiKey is used to authenticate to several servers. If an intruder intercepts a valid OTP of a user to a server then in the real system the intruder has a small window, until the user authenticates himself, in which he can successfully use the OTP to get access to the services that are granted by any other server that recognizes that YubiKey. But, in the case the window expired if the server receives the intercepted OTP then it sends back the OTP with the status replayed. On the other hand in the abstraction the replayed OTP is interpreted by the server as valid.

4.3.1 First approach: Ordered Nonces

We wanted to refine the abstraction in order to cover the situation described above. For this we maintained the encrypted nonce as a representation of the OTP, but we added an order relation over the nonces making them closer to the counter values that are part of the password. We distinguish between the latest created nonce for the OTP and all the older nonces created but not yet validated by the server. We abstracted this by permitting the server and each user u to share two databases: $nonces(u, lastcreated)$ and $nonces(u, created)$ for the two types of nonces.

When the server receives an OTP with a nonce that is part of the $created$ set then it validates it sending back an message with the OTP and with the ok status, but also it changes the membership of the nonce for that agent to a new private set named $used$. Because of the abstraction in AIF, a nonce can not be seen as a value and rather as an equivalence class which implies that in fact this rule takes all the nonces from the $created$ set and adds them to the $used$ one. Once one OTP that is not the last created reaches the server for verification all the other created $OTPs$ but not yet used become invalidated. The only OTP that can be accepted by the server is the last one created.

In the second case in which the server receives the recently created OTP of an user, he accepts it and then changes the membership to the used database for both the last created nonce and the ones part of the created. It simulates the reality because if the server remembered for an user the last value for the counter for example 9 and then it receives from the same user the value 12, the server can not accept in the future as valid $OTPs$ having values 10 or 11 even though they were not previously received.

The slight problem in this abstraction is that in the case of the verification of an OTP that is part of the $created$ set, the server possibly invalidates more $OTPs$ than in the real system. This fact rises from not being able to distinguish between elements that are in the same equivalence class, but we try to find a solution by refining more the abstraction for the OTP.

As mentioned in the subsection about AIF, because we want to prove that the YubiKey is resilient to replay attacks we need to pay more attention to the rule for specifying the attack. We consider that the proper abstraction for a successful replay attack on the protocol is to request to be reachable a state in which a nonce is in the same time in the $lastcreated$ or $created$ set and in the $used$ set. This is the necessary condition for an invalidated OTP to be verified as valid by the server.

We used the translation tool that can be downloaded together with the AIF library to generate the set of Horn clauses for the approximation of the YubiKey protocol and then we used SPASS to a proof for an attack. The deduction terminates after two minutes with the response that the protocol is resilient against replay attacks. We used a machine with 2 processors Intel(R) Xeon(R) CPU X5650, 6 Cores at a 2.66 GHz Clock Speed.

4.3.2 Second approach: Incrementation Function

We tried to refine the $created$ equivalence class of the nonces by representing the $OTPs$ as a triplet formed from agent's identity, a nonce and a counter value: $\{A, N_a, inc(zero)\}_{k_{AS}}$. The counter is abstracted by a successor function which we named inc, abbreviation from

incrementation, and the value is given by the number of the applications of the inc function to a constant, $zero$. For example $inc(zero)$ encodes 1 and $inc(inc(inc(zero)))$ the value 3.

This representation of the OTP introduced difficulties while trying to abstract the creation of a new valid OTP and the verification step. The problem was the need to remember for each user the last created counter value or the last received valid value. In the last case the server needs to know for each agent the last value he received in order to compare it with the counter value from the OTP he receives next. The solution was to add two facts: $next$ for the agents and exp_next for the server. We detail the use of the exp_next fact, because the fact introduced for agents for remembering the last created value is very similar. The exp_next fact requires four parameters: server's and agent's identity, a nonce and a counter value. As the name suggests, a fact of this type is added each time the server receives a valid OTP from any agent to remember which is the next counter value expected. At first, we only considered the case in which the server accepts an OTP only if the counter value is equal to the last valid counter value incremented by 1. We discuss the general case later.

The use of the exp_next fact is represented in figure 5. The rule applies for the states in which a new password $otp(a, Na, M1)$ is sent to the server to be verified and it is a valid password, fact enforced by the condition $exp\_next(s, a, Nb, M1)$ which means that the server was expecting the value of the counter to be exactly the one from the password. The state changes after the rule is used; the next value that the servers expects to see from the same user is $inc(M1)$, the incrementation of the recently received value of counter and a message is sent with the confirmation that the password is valid.

AIF rule	**Set Membership Abstraction**
Nb in noncesS(s,a,lastvalue).	occurs(val(1,X, ...)).
$exp\_next(s, a, Nb, M1)$.	$exp\_next(u, val(1, X, ...), Y)$.
Na notin noncesS(s,a,lastvalue).	occurs(val(0,0, ...)).
Na notin noncesS(s,a,used).	
iknows(otp(a,Na,M1))	iknows(otp(u,val(0,0, ...),Y)
$\Rightarrow$	$\rightarrow$
Nb in noncesS(s,a,used).	val(1,X, ...) $\twoheadrightarrow$ val(0,1, ...).
Na in noncesS(s,a,lastvalue).	val(0,0, ...) $\twoheadrightarrow$ val(1,0, ...).
$exp\_next(a, Na, inc(M1))$.	$exp\_next(u, val(1, 0, ...), inc(Y))$.
$response\_valid$;	$response\_valid$;

Figure 5: AIF rule for accepting a valid password of the agent a by the server s. The order of sets is

As mentioned earlier, in AIF facts are persistent which means that after the rule above is applied it remains true for the entire run of the protocol $exp\_next(s, a, Nb, M1)$ but also $exp\_next(s, a, Nb, inc(M1))$. Therefore, it may seem that the server could accept an OTP that has the value of the counter smaller than the last seen. This is not the case because we use the revocable membership of the nonces that are part of the facts to simulate an "invalidation" of facts. Also, because of the representation we use for the counter values, the verification rule for a valid OTP is correct. Below, we detail the abstraction we proposed for the YubiKey protocol.

We use the nonces as a mapping to a corresponding counter value, because although

having the value of the counters as part of the sets may look like a straightforward solution and that would actually be closer to the real system this is not possible in AIF. It is because each set(e.g: $noncesS(s, a, lastvalue)$, $noncesS(s, a, used)$) is an equivalence class, which implies that the terms that are part of it must be indistinguishable from each other. Therefore, it is not permitted to introduce in the sets functions because this way it is possible to compare a term from the set to another and it is only possible to have variables and constants as part of sets. This is why our solution was to use nonces as the terms that represent the counter values in the sets.

In the real system, a counter value can be in several states: the last value created or received, an old value or for the users the current value that they are waiting the verification response from the server. Following this classification we introduced several sets but which are formed from nonces, such that for the server the sets have the form: $noncesS(s, u, ServSts)$ and for the agents the form: $nonces(u, UsrSts)$, where $ServSts$ and $UsrSts$ are *enumeration variables* replacing the status that a nonce can have from the perspective of the server and respectively of the user. $ServSts$ stands for $\{lastvalue, used\}$, separating the nonces in two sets: the first for the ones that are part of the last received valid OTP and the second for the rest of the nonces that were received. In the case of the database of users, the enumeration variable $UsrSts$ is replacing three values: $curr, lastvalue, used$; the last two have the same meaning as for the server and the additional $curr$, abbreviation from current, stands for the nonces that are part of $OTPs$ that were sent to the server for verification.

Further we try to explain how we succeed to manipulate the facts with the set membership of nonces. We added an initialization rule in which we specify that the first value of the counter that can be created is 1 and the server expects to receive for the first time from the agents an OTP which has the counter value 1(see figure 6). The condition for the rule to be enabled is the sets are empty, which is true if the run of the protocol just started. The initial state changes into a state in which the facts $next$ and exp_next are added constitued from a dummy nonce $firstNonce$ and the counter value $inc(zero)$. Also, the nonce $firstNonce$ is added to the sets last value seen for both the agent and the server and additionally, for consistency, to the used set of nonces of the agent.

```
firstNonce notin nonces(a,lastvalue).
firstNonce notin nonces(a,used).
firstNonce notin noncesS(s,a,lastvalue).
firstNonce notin noncesS(s,a,used)
=>
firstNonce in nonces(a,lastvalue).
firstNonce in nonces(a,used).
firstNonce in noncesS(s,a,lastvalue).
next(a,firstNonce,inc(zero)).
exp_next(s,a,firstNonce,inc(zero));
```

Figure 6: Initialisation rule in AIF for the server s and agent a

The next step in the protocol is the creation of an OTP by the agent. It is required to find the proper value for a valid OTP, which means there should exist a fact $next(a, Na, cnt)$

having the agent as parameter and the nonce in the fact should be in the set of the last values but also in the used, which means that the verification process has ended for the last password sent. The result of this rule is that the new password is created and sent to the server for verification. As it can be seen from figure 5, for a password to be verified to a valid one, it is necessary first of all to be fresh, which is modelled by asking that the nonce from the OTP is not in the used and neither in the last used set of the server. Then the second requirement is related to the value of the counter which has to be the same with the one that the server expects. The right expected value at a round of the protocol is selected from the old, seen values by requiring that there should exist a nonce that is in the last value set of the server and that for this nonce and agent there exist also a fact exp_next. This is true because of the encoding we did, the nonce that is in the set last used of the agent or the server points to the next value that should be created or received.

The result of applying the valid verification step consists of the move of the nonce from the last value set to the used set and the addition of the nonce from the verified password to the last value set. Also, a new fact for remembering the next valid value of the counter is added : $exp\_next(a, Na, inc(M1))$, having the new nonce pointing to the new expected next value. As said earlier we "invalidate" the fact $exp\_next(a, Na, M1)$, by changing the set membership of the nonce Na. This fact is more visible in the right side of figure 5, where the rule in AIF is written in an intermediate format that makes easy to transform it into a set of Horn clauses. The fact $exp\_next(u, val(1, X, ...), Y)$ changes due to the changing of the state membership of the nonce into $exp\_next(u, val(0, 1, ...), Y)$. Therefore, after the rule is applied the same OTP with the counter value $M1$ does not validate the conditions of this rule anymore. On the other hand, having the facts persistent the rule can be applied again but the meaning is different. If it is applied again it means that in fact a new fresh OTP is received because it contains a nonce that was not registered before by the server.

We considered that a replay attack can be mounted in the case that a password that was already verified is sent back to the server and is verified as a valid one. The rule in AIF has as conditions the requirement that a nonce is in the same time into the last value and into used set of the server and that it exists a fact exp_next having as parameter this nonce. The rule is the same as in the first approach we proposed and depends approximatively by the same terms or the same conditions. Therefore, the verification of this approximation of the protocol can not bring much more than the one for the first approach. On another hand, this approach is an important step into finding a more refined approximation of the YubiKey protocol.

We used SPASS for verifying our approximation of the protocol, but it did not terminate. The problem is with the representation of the counter values using a function. From one round of the protocol to another facts are added using as parameter the result of the inc function applied to a value that is already part of a fact. This way, it can not conclude that the same set of information is deduced from one round to another. The termination was forced by bounding the counter value to the maximum 2. In less than an hour it terminated. The code in AIF can be found in Appendix B.

4.3.3 Extending the model

We intended to extend the abstraction above by only limit the counter value to be bigger than the last received in the case of a valid password and not the next succesor. Following the abstraction above a bounded case can be created. It is possible to repeat the rule that we presented for the the valid verification step having as a requirement an OTP with the counter value of the form: $inc(inc(M1))$, $inc(inc(inc(M1)$ and so on, where $M1$ is the incremented value of the last valid OTP.

Another approach is to create a set of rules to simulate a comparing function between two counter values, for example to be able to check if a value is bigger than another value. One method to abstract the greater than function is to decrement the value that it is expected to be bigger until either it equals the second value(in this case a fact gt having the two value is added) or $zero$ is reached (a fact lt with the two values as parameters is added). This way in the verification step it must be checked if the counter value just received is greater than all the others received from that agent or greater than the last value received. Unfortunately, because in AIF facts are persistent and also it is impossible to use an universal quantifier over facts or any other quantifier the method described above can not be applied.

The second solution is to add a fact $known$, having as parameters an agent and a counter value which is added for each value of the counter which is invalidated for the agent. When the server receives a new valid OTP it adds a fact $known$ for that value, but also for all the smaller values. For this, the method described above can be used, such that the value is decremented until $zero$ and at each step a fact $known$ is added. This way the rule for verifying an OTP that is not valid only needs to have as condition that the counter value received is part of a $known$ fact for that agent. The hardest part remains to create a correct verification rule for a valid password. The intuitive condition is to add just a condition to have the received value not known. Unfortunately, we did not find a way to express it, encountering difficulties because a negation of a fact is not possible and neither to compare it with all the received values. If a method to abstract this is found then this approach seems elegant and the closest to the real system.

4.4 Protocols protected by YubiKey

The YubiKey protocol being resilient to replay attacks can be used with success to correct other protocols that are not secure against this type of attacks.

We take as an example the DH-2msg protocol that was discussed in the previous sections. It can be insecure in the case the agents save the history of their roles in the protocol on their local machine. An intruder can enter into the possession of the file using a malicious software that is downloaded accidentally by the user on its machine and if the intruder monitored the network then he can impersonate successfully the agent to other honest agents. The solution we propose is very simple; it is enough to add the use of the YubiKey into the protocol. The initiator of the protocol should include an OTP into the message, in clear but also into the signature. This way the responder can send the OTP to the server and then if the server sends back that the password is valid the responder can be sure that the message is fresh.

DH-2msg	DH-2msg with YubiKey
$A \to B$: $g^x, B, \{g^x, B\}_{sk(A)}$	$A \to B$: $g^x, OTP_A, B, \{g^x, OTP_A, B\}_{sk(A)}$
$B \to A$: $g^x, g^y, A, \{g^x, g^y, A\}_{sk(B)}$	$B \to A$: $g^x, g^y, A, \{g^x, g^y, A\}_{sk(B)}$

Figure 7: Diffie-Hellman protected by YubiKey

Even though the intruder is able to learn exponents from past sessions of the protocol, he is not able anymore to exchange a key with a honest agent. A replay of an old message to which the intruder knows the exponent is not enough to successfully impersonate the agent in the new model, because even though it is possible to intercept an OTP that would be validated by the server but is not able to create the signature.

We implemented the abstraction of this model in AIF and then tried to verify it with SPASS, but unfortunately it took too long to find a result.

We think that the YubiKey can be used successfully to correct protocols that are not resilient to replay attacks. We presented a simple example, but the same method can be applied to other protocols.

5 Conclusion

We tried two tools in order to verify a protocol and we encountered difficulties in both cases, which shows that still more complicated protocols can not be verified automatic.

The Scyther tool introduces adversaries that can compromise agents during the run of the protocol learning their keys or the random numbers generated. It can verify a protocol against adversaries that range from the usual model in symbolic analysis, the $Dolev - Yao$ to much more powerful adversaries, allowing to find attacks for protocols that were previously online discovered by the computational approach. It can verify protocols with an unbounded number of sessions and nonces and is very in general and for a small protocol like $Needham - Schroeder$ it takes a few seconds for the verification to complete. Unfortunately, it just permits to model small protocols. Some of the limitations are the interpretation of equality between terms as syntactic equality and the roles being specified as a straight-line sequence of events, which does not permit control flow primitives such as branching and looping.

The language AIF permits a larger range of protocols to be abstracted. It was thought as an extension for the use case of the method of approximating protocols by a set of Horn clauses, by permitting also the abstraction of protocols and web-services that use a database. One of the difficulties with this approach is that the facts are persistent which leads to the situation in which a participant can react to a message any number of times, even though in the real system is not possible. There is a solution that can be used to simulate non-persistent facts, which we used but unfortunately it was not enough to obtained a more refined model. The problem is introduced by the counter values that can not be used as revocable terms.

A solution to create the refined model for the YubiKey where the counter values are abstracted is to introduce them into the state. This way facts are invalidated, but a problem still exists regarding the termination of the verification process.

As a future work, it would be interesting to model the YubiKey with an approach such

that the verification against replay attacks resumes to verify the injective agreement property. We are not sure it can be done. There exists another property that was introduced into the hierarchy of authentication by C. Cremers [7], named synchronization which is stronger than the agreement notion and is obtained from the non-injective synchronization and a loop property. If a protocol verifies this property then it also verify the agreement property which means the protocol is secure against replay attacks. Unfortunately, it is required for threads of the protocol to be independent, which means that a round of the protocol can not share information with other rounds, which means it can not be used with the YubiKey protocol.

Also, it would be interesting to find more protocols that can be protected by the YubiKey.

References

[1] Google apps bussiness login using yubikey information available at:. http://wiki.yubico.com/wiki/index.php/Applications:Google_Apps.

[2] Yubikey's customers list available at:. http://www.yubico.com/references.

[3] David Basin and Cas Cremers. Modeling and analyzing security in the presence of compromising adversaries. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 340–356, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] David Basin, Cas Cremers, and Catherine Meadows. *Model Checking and Security Protocols*, chapter 24. Springer, 2011. To appear.

[5] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing pkcs#11 security tokens. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 260–269, New York, NY, USA, 2010. ACM.

[6] Yubico company. Yubikey official page available at:. http://www.yubico.com/yubikey.

[7] C. J. F. Cremers, S. Mauw, and E. P. de Vink. Injective synchronisation: an extension of the authentication hierarchy. *Theor. Comput. Sci.*, 367:139–161, November 2006.

[8] Cas Cremers. Scyther tool with compromising adversaries extension. available online at. http://people.inf.ethz.ch/cremersc/scyther/compromise/index.html.

[9] Prateek Gupta and Vitaly Shmatikov. Key confirmation and adaptive corruptions in the protocol security logic. Technical report, 2006.

[10] Sebastian Alexander M ödersheim. Abstraction by set-membership: verifying security protocols and web services with databases. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 351–360, New York, NY, USA, 2010. ACM.

A Appendix A

The AIF implementation of the approximation of Yubikey using only nonces.

```
Problem: NoncesYubikey;

Types:
Agent   : {a,i,s};
U       : {a,i};
S       : {s};
D       : {i};
H       : {a};
NonSts: {lastcreated, created, curr, used};
M1,M2 : untyped;
Na,Nb,zero,ok,replayed    : value;


Sets:
nonces(H, NonSts);

Functions:
public  otp/2, pair/2, hmac/3;
private symm_key/1; % we actually don't need it..

Facts:
iknows/1, attack/0, grantacces/2, replayed_otp/1;

Rules:
\Agent. => iknows(Agent);

iknows(pair(M1,M2))=>iknows(M1).iknows(M2);
iknows(M1).iknows(M2)=>iknows(pair(M1,M2));

\D. iknows(M1) => iknows(otp(D, M1));

=[Na]=> iknows(Na);

%init rule
\H.
zero notin nonces(H, created).
zero notin nonces(H, lastcreated).
zero notin nonces(H, used)
=>
zero in nonces(H, lastcreated);

%% create new OTPs
\H.
```

```
Na in nonces(H, lastcreated).
=[Nb]=>
Na in nonces(H, created).
Nb in nonces(H, lastcreated).
Nb in nonces(H, curr).
iknows( otp(H, Nb));

%% verification steps
\H.
iknows( otp(H, Na)).
Na in nonces(H, created)
=>
Na in nonces(H, used).
iknows(pair(otp(H, Na), pair(ok , hmac(H, otp(H, Na), ok))));

%% invalidate smaller values for the counter : all from created go to used
\H.
iknows( otp(H, Na)).
Na in nonces(H, lastcreated).
Nb in nonces(H, created)
=>
Nb in nonces(H, used).
Na in nonces(H, used).
iknows(pair(otp(H, Na), pair(ok , hmac(H, otp(H, Na), ok))));

%% verification -> replayed otp
\H.
iknows( otp(H, Na)).
Na in nonces(H, used)
=>
Na in nonces(H, used).
iknows(pair(otp(H, Na), pair(replayed , hmac(H, otp(H, Na), replayed))));

%client receives the validation answer from server

\H,S.
iknows(pair(otp(H, Na), pair(replayed , hmac(H, otp(H, Na), replayed)))).
Na in nonces(H, curr).
=>
replayed_otp(H);

\H,S.
iknows(pair(otp(H, Na), pair(ok , hmac(H, otp(H, Na), ok)))).
Na in nonces(H, curr)
=>
grantacces(H, Na);
```

```
%% attack rules

% rule to verify correctness of the model
\H.
Na in nonces(H, lastcreated).
Na in nonces(H, created)
=>
attack;

%% replay attack
\H.
Na in nonces(H, created).
Na in nonces(H, used)
=> attack;

\H.
Na in nonces(H, lastcreated).
Na in nonces(H, used)
=> attack;
```

B Appendix B

The AIF implementation of the second approach to abstract the Yubikey using only a nonce and the counter values for the OTP.

```
Problem: Yubikey2;

Types:
Agent   : {a,i,s};
U       : {a,i};
S       : {s};
D       : {i};
H       : {a};
UsrSts  : {curr, lastvalue, used};
ServSts: {lastvalue, used};
M1,M2 : untyped;
Na,Nb,zero,ok,replayed    : value;


Sets:
nonces(U, UsrSts),
noncesS(S, U, ServSts);

Functions:
public inc/1, hmac/3, otp/3, pair/2;
private symm_key/1;
```

```
Facts:
iknows/1, attack/0, next/3, exp_next/4, grantacces/2, replayed_otp/1;

Rules:
\Agent. => iknows(Agent);

iknows(pair(M1,M2))=>iknows(M1).iknows(M2);
iknows(M1).iknows(M2)=>iknows(pair(M1,M2));

\D. iknows(M1).iknows(M2) => iknows(otp(D, M1, M2));

\D. iknows(M1).iknows(M2) => iknows(hmac(D, M1, M2));

=[Na]=> iknows(Na);


%% initialization step
\H,S.
zero notin nonces(H, lastvalue).
zero notin nonces(H, used).
zero notin noncesS(S,H, lastvalue).
zero notin noncesS(S, H, used)
=>
zero in nonces(H, lastvalue).
zero in nonces(H, used).
zero in noncesS(S, H, lastvalue).
next(H, zero, inc(zero)).
exp_next(S, H, zero, inc(zero));

%% the incrementation of the counter
%%\H. knows(H, Na, M1)
%% => next(H, Na, inc(M1));

%% sending an otp to be verified
\H.
Na in nonces(H, lastvalue).
Na in nonces(H, used).
next(H, Na, M2)
=[Nb]=>
Na in nonces(H, used).
Nb in nonces(H, lastvalue).
Nb in nonces(H, curr).
iknows(otp(H, Nb, M2));


%% verification -> valid OTP
```

```
\H,S.
Nb in noncesS(S, H, lastvalue).
%Nb notin nonces(H, lastvalue).
Na notin noncesS(S, H, lastvalue).
Na notin noncesS(S, H, used).
exp_next(S, H, Nb, M1).
iknows(otp(H, Na, M1))
=>
Nb in noncesS(S, H, used).
Na in noncesS(S, H, lastvalue).
exp_next(S, H, Na, inc(M1)).
iknows(pair(otp(H, Na, M1), pair(ok , hmac(H, otp(H, Na, M1), ok))));

%% verification -> replayed OTP
\H,S.
iknows(otp(H,  Na, M1)).
Na notin noncesS(S, H, lastvalue).
Na notin noncesS(S, H, used).
exp_next(S, H, Nb, M1).
Nb in noncesS(S, H, used)
=>
Nb in noncesS(S, H, used).
iknows(pair(otp(H, Na, M1), pair(replayed , hmac(H, otp(H, Na, M1), replayed))));


%% clients receives response from server
\H,S.
iknows(pair(otp(H, Na, M1), pair(replayed, hmac(H, otp(H, Na, M1), replayed)))).
Na in nonces(H, curr)
=>
Na in nonces(H, used).
next(H, Na, inc(M1));
%replayed_otp(H);

\H,S.
iknows(pair(otp(H, Na, M1), pair(ok , hmac(H, otp(H, Na, M1), ok)))).
Na in nonces(H, curr)
=>
Na in nonces(H, used).
next(H, Na, inc(M1));
%grantacces(H, Na);

%% rule to verify if it is correct
%\S, H.
%Na in nonces(H, used).
%Na in nonces(H, curr)
%=>
```

```
%attack;

%% replay attack

\S,H.
Na in noncesS(S, H, lastvalue).
Na in noncesS(S, H, used).
exp_next(S, H, Na, inc(M1)).
=>
attack;
```

C Appendix C

The Diffie-Hellman key exchange protocol version with two messages exchanged abstracted in AIF.

```
%% The model can be corrected adding the Yubikey use
%% The first message should contain a signature over the g1(X) and an OTP.
%% In case of an replay the server should reject the OTP and the agent not to sha
%% In the second message there is no need to add an OTP as a response, it is enou
%% over both of them

Problem: DH;

Types:
Agent   : {a,b,i};
D       : {i};
H       : {a,b};
H1      : {a};
H2      : {b};
ValSts  : {curr, used};
X, Y    : value;
M1,M2 : untyped;
Sets:
created(H,H,ValSts),
received(H, H, ValSts);

Functions:
public  g1/1, g2/2, sign/4, pair/2;
private symm_key/1;

Facts:
iknows/1, shared_key/3, attack/0;

Rules:
```

```
\Agent. => iknows(Agent);

iknows(pair(M1,M2))=>iknows(M1).iknows(M2);
iknows(M1).iknows(M2)=>iknows(pair(M1,M2));

iknows(X)=> iknows(g1(X));

iknows(Y).iknows(g1(X)) => iknows(g2(X, g1(Y)));
iknows(g2(X, g1(Y))) => iknows(g2(g1(X), Y));
iknows(g2(g1(X), Y)) => iknows(g2(X, g1(Y)));

\D,Agent.
iknows(M1).iknows(M2)
=>
iknows(sign(D, M1, M2, Agent));

%\Agent,H.
%iknows(sign(Agent, M1, M2, H))
%=>
%iknows(M1).iknows(M2);


=[X]=> iknows(X);

\H1,H2.
=[X]=>
X in created(H1,H2,curr).
iknows(pair(H1,g1(X)));

\H2,H1.
iknows(pair(H1,g1(X))).
=[Y]=>
Y in created(H2, H1, curr).
X in received(H2, H1, curr).
shared_key(H2, H1, g2(Y, g1(X))).
iknows(pair(H2, pair(g1(Y), sign(H2,g1(Y),g1(X),H1))));

\H1,H2.
X in created(H1,H2,curr).
iknows(pair(H2, pair(g1(Y), sign(H2,M1,g1(X),H1))))
=>
X in created(H1,H2,curr).
Y in received(H1,H2,curr).
shared_key(H2, H1, g2(M1,X));

%% consider that after the key expired the exponents are leaked
```

```
\H1,H2.
X in created(H1,H2, curr).
Y in created(H2,H1, curr).
X in received(H2,H1, curr).
Y in received(H1,H2, curr)
=>
iknows(X).
X in created(H1,H2, used).
Y in created(H2,H1, used).
X in received(H2,H1, used).
Y in received(H1,H2, used);
```

```
\H1,H2.
X in created(H1, H2, used).
X in received(H2, H1, curr).
Y in created(H2, H1, curr).
shared_key(H2,H1,g2(Y, g1(X)))
=> attack;
```

D Appendix D

The protected Diffie-Hellman key exchange protocol by Yubikey abstracted in AIF.

```
%% The model can be corrected adding the Yubikey use
%% The first message should contain a signature over the g1(X) and an OTP.
%% In case of an replay the server should reject the OTP and the agent not to sha
%% In the second message there is no need to add an OTP as a response, it is enou
%% over both of them

Problem: DH;

Types:
Agent   : {a,b,s,i};
U       : {a,i};
S       : {s};
D       : {i};
H       : {a,b};
H1      : {a};
H2      : {b};
Con     : {ok, replayed};
ValSts  : {curr, used};
NonSts: {lastcreated, created, curr, used};
M1,M2,M3,M4 : untyped;
```

```
X,Y,Na,Nb,zero : value;

Sets:
personal(H,H,ValSts),
received(H,H, ValSts),
nonces(H, NonSts);

Functions:
public  g1/1, g2/2, sign/4, otp/2, pair/2, hmac/3, msg1/4;
private sk/1;

Facts:
iknows/1, shared_key/3, attack/0, grantacces/2, replayed_otp/1;

Rules:

%% intruder rules %%
\Agent. => iknows(Agent);

iknows(pair(M1,M2))
=>
iknows(M1).
iknows(M2);

iknows(M1).
iknows(M2)
=>
iknows(pair(M1,M2));

iknows(msg1(M1, M2, M3, M4))
=>
iknows(M1).
iknows(M2).
iknows(M3).
iknows(M4);

\Agent.
iknows(M2).
iknows(M3).
iknows(M4)
=>
iknows(msg1(Agent, M2, M3, M4));

iknows(X)
=>
iknows(g1(X));
```

```
iknows(Y).iknows(g1(X)) => iknows(g2(X, g1(Y)));
iknows(g2(X, g1(Y))) => iknows(g2(g1(X), Y));
iknows(g2(g1(X), Y)) => iknows(g2(X, g1(Y)));

\D,Agent.
iknows(M1).iknows(M2)
=>
iknows(sign(D, M1, M2, Agent));

%\Agent,H.
%iknows(sign(Agent, M1, M2, H))
%=>
%iknows(M1).iknows(M2);

=[X]=> iknows(X);

\D. iknows(M1) => iknows(otp(D, M1));

%% agents rules %%

\H.
zero notin nonces(H, created).
zero notin nonces(H, lastcreated).
zero notin nonces(H, used)
=>
zero in nonces(H, lastcreated);

\H1,H2.
Na in nonces(H1, lastcreated).
=[X]=>
Na in nonces(H1, created).
X in nonces(H1, lastcreated).
X in nonces(H1, curr).
X in personal(H1,H2,curr).
iknows(msg1(H2,g1(X), otp(H1, X), sign(H1, H2, g1(X), otp(H1, X))));

\H2,H1.
iknows(msg1(H2,g1(X), otp(H1, Nb), sign(H1, H2, g1(X), otp(H1, Nb))))
=>
iknows( pair(H2, otp(H1, Nb)));

%% verification -> valid + counter not last value
\H2, H1.
iknows( pair(H2, otp(H1, Na))).
Na in nonces(H1, created)
=>
Na in nonces(H1, used).
```

```
iknows(pair(otp(H1, Na), pair(ok , hmac(H2, otp(H1, Na), ok))));

%% verification -> valid + invalidate smaller values for the counter : all from c
\H2, H1.
iknows( pair(H2, otp(H1, Na))).
Na in nonces(H1, lastcreated).
Nb in nonces(H1, created)
=>
Nb in nonces(H1, used).
Na in nonces(H1, used).
iknows(pair(otp(H1, Na), pair(ok , hmac(H2, otp(H1, Na), ok))));

%% verification -> replayed otp
\H2,H1.
iknows(pair(H2, otp(H1, Na))).
Na in nonces(H1, used)
=>
Na in nonces(H1, used).
iknows(pair(otp(H1, Na), pair(replayed , hmac(H2, otp(H1, Na), replayed))));

%% the responder sends its part of the key after the verification was correct
\H2,H1.
iknows(msg1(H2, g1(X), otp(H1, Nb), sign(H1, H2, g1(X), otp(H1, Nb)))).
iknows(pair(otp(H1, Na), pair(ok , hmac(H2, otp(H1, Na), ok)))).
=[Y]=>
Y in personal(H2, H1, curr).
X in received(H2, H1, curr).
shared_key(H2, H1, g2(Y, g1(X))).
iknows(pair(H2, pair(g1(Y), sign(H2,g1(Y),g1(X),H1))));

\H1,H2.
X in personal(H1,H2,curr).
iknows(pair(H2, pair(g1(Y), sign(H2,M1,g1(X),H1))))
=>
X in personal(H1,H2,curr).
Y in received(H1,H2,curr).
shared_key(H2, H1, g2(M1,X));

%% consider that after the key expired the exponents are leaked

\H1,H2.
X in personal(H1,H2, curr).
Y in personal(H2,H1, curr).
X in received(H2,H1, curr).
Y in received(H1,H2, curr)
=>
iknows(X).
```

```
X in personal(H1,H2, used).
Y in personal(H2,H1, used).
X in received(H2,H1, used).
Y in received(H1,H2, used);

\H1,H2.
X in personal(H1, H2, used).
X in received(H2, H1, curr).
X in nonces(H1, created).
X in nonces(H1, used).
Y in personal(H2, H1, curr).
shared_key(H2,H1,g2(Y, g1(X)))
=> attack;

\H1,H2.
X in personal(H1, H2, used).
X in received(H2, H1, curr).
X in nonces(H1, lastcreated).
X in nonces(H1, used).
Y in personal(H2, H1, curr).
shared_key(H2,H1,g2(Y, g1(X)))
=> attack;
```