YubiSecure? Formal Security Analysis Results for the Yubikey and YubiHSM

Robert Künnemann[1,2] and Graham Steel[2]

[1] LSV & INRIA Saclay – Île-de-France
[2] INRIA Project ProSecCo, Paris, France

Abstract. The Yubikey is a small hardware device designed to authenticate a user against network-based services. Despite its widespread adoption (over a million devices have been shipped by Yubico to more than 20 000 customers including Google and Microsoft), the Yubikey protocols have received relatively little security analysis in the academic literature. In the first part of this paper, we give a formal model for the operation of the Yubikey one-time password (OTP) protocol. We prove security properties of the protocol for an unbounded number of fresh OTPs using a protocol analysis tool, tamarin.
In the second part of the paper, we analyze the security of the protocol with respect to an adversary that has temporary access to the authentication server. To address this scenario, Yubico offers a small Hardware Security Module (HSM) called the YubiHSM, intended to protect keys even in the event of server compromise. We show if the same YubiHSM configuration is used both to set up Yubikeys and run the authentication protocol, then there is inevitably an attack that leaks all of the keys to the attacker. Our discovery of this attack lead to a Yubico security advisory in February 2012. For the case where separate servers are used for the two tasks, we give a configuration for which we can show using the same verification tool that if an adversary that can compromise the server running the Yubikey-protocol, but not the server used to set up new Yubikeys, then he cannot obtain the keys used to produce one-time passwords.

Keywords: Key management, Security APIs, Yubikey

1 Introduction

The problem of user authentication is central to computer security and of increasing importance as the cloud computing paradigm becomes more prevalent. Many efforts have been made to replace or supplement user passwords with stronger authentication mechanisms [1]. The Yubikey is one such effort. Manufactured by the Swedish company Yubico, the Yubikey itself is a low cost ($25), thumb-sized USB device. In its typical configuration, it generates one-time passwords (OTPs) based on encryptions of a secret value, a running counter and some random values using a unique AES-128 key contained in the device. A Yubikey authentication server verifies an OTP only if it decrypts under the correct AES key to give a

valid secret value with a counter larger than the last one accepted. The counter is therefore used as a means to prevent replay attacks. The system is used by a range of governments, universities and enterprises, e.g. Google, Microsoft, Agfa and Symantec [2].

Despite its widespread deployment, the Yubikey protocol has received little independent security analysis. Yubico themselves present some security arguments on their website [3]. A first independent analysis was given by blogger Fredrik Björck in 2009 [4], raising issues that Yubico responded to for a subsequent post [5]. The only formal analysis we are aware of was carried out by Vamanu [6], who succeeded in showing security for an abstract version of the Yubikey OTP protocol for a bounded number of fresh OTPs. In this paper, we use a new protocol analysis tool tamarin [7], not available at the time of Vamanu's analysis. We are able to prove the protocol secure in an abstract model for an unbounded number of fresh OTPs.

The aforementioned results assume that the authentication server remains secure. Unfortunately, such servers are sometimes breached, as in the case of the RSA SecurID system where attackers were able to compromise the secret seed values stored on the server and so fake logins for sensitive organisations such as Lockheed Martin [8]. RSA now use a Hardware Security Module (HSM) to protect seeds in the event of server compromise. Yubico also offer (and use themselves) an application specific HSM, the YubiHSM to protect the Yubikey AES keys in the event of an authentication server compromise by encrypting them under a master key stored inside the HSM. In the second part of our paper, we analyse the security of the YubiHSM API. First we show that due to an apparent oversight in the cryptographic design, an attacker with access to the server where Yubikey AES keys are generated is able to decrypt the encrypted keys and obtain them in clear. We informed Yubico of this problem in February 2012 and they issued a security advisory [9]. We then prove secrecy of keys in various configurations of YubiHSMs and servers, and suggest design changes that would allow a single server to be used securely.

All our analysis and proofs are in an abstract model of cryptography in the Dolev-Yao style, and make various assumptions (that we will make explicit) about the behaviour of the Yubikey and YubiHSM. At the end of the paper we will discuss how we could refine our models in future work.

The rest of the paper proceeds as follows. In section 2, we described the Yubikey and its OTP protocol. We model and analyse the security of this protocol in section 3. We then describe the YubiHSM in section 4, and the attacks we found in section 5. We model the HSM API and prove secrecy of the sensitive keys for various configurations in section 6. Finally we evaluate our results (7) and conclude (8).

2 The Yubikey Authentication Protocol

In the following, we will cover the authentication protocol as it pertains to version 2.0 of the Yubikey device [10].

The Yubikey is connected to the computer via the USB port. It identifies itself as a standard USB keyboard in order to be usable *out-of-the-box* in most environments using the operating system's native drivers. Since USB keyboards send "scan codes" rather than actual characters, a modified hexadecimal encoding, called *modhex* is employed, which uses characters that have the same position on many different keyboard layouts, including the German QUERTZ, the French AZERTY and the US QWERTY layout. Each keystroke carries 4 bits of information [10, Section 6.2].

The Yubikey can be configured to work in any of the following modes [10, Section 2.1]:

- *Yubikey OTP*, which is the method that is typically employed
- *OATH-HOTP*, where the OTP is generated according to the standard RFC 4226 HOTP algorithm,
- *Challenge-response mode*, where a client-side API is used to retrieve the OTP, instead of the keyboard emulation, and
- *Static mode*, where a (static) password is output instead of an OTP.

We will focus only on the Yubikey OTP mode, which we will explain in detail. Depending on the authentication module used on the server, there are four basic authentication modes [11, Section 3.4.1]:

- User Name + Password + YubiKey OTP
- User Name or YubiKey OTP + Password
- YubiKey OTP only
- User Name + Password

As the security provided by a user-chosen password is an orthogonal topic and the OTP is the main feature of the Yubikey, we will only focus on the third authentication mode.

The string emitted by the Yubikey is a 44-character string (i. e., 22 bytes of information in modhex encoding) and consists of the unique public ID (6 bytes) and the OTP (16 bytes) [12]. The length of the OTP is exactly the block-length of AES. It contains the following information in that order [10, Section 6.1].

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

See Figure 1 for an example.

Yubico assigns an AES key and a public and secret ID to the Yubikey before shipment, but they can be overwritten. The Yubikey is *write-only* in this regard, thus it is not possible to retrieve secret ID nor the AES key. The session counter is incremented whenever the Yubikey is plugged in. Once it reaches its limit of $2^{16} = 65536$, it cannot be used anymore. The timecode is incremented by an 8Hz

internal clock. When it reaches its limit, the session is terminated, i. e., no more OTPs can be generated. This happens after approximately 24 days. The token counter is incremented whenever an OTP is generated. When it reaches its limit of 256, it restarts at 1 instead of terminating the session. The pseudo-random value of length two bytes is supposed to add entropy to the plain-text, while the CRC is supposed to detect transmission errors. It does not provide cryptographic integrity.

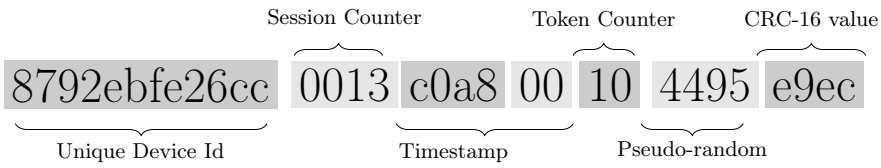

Fig. 1. Structure of the OTP (session: 19, token: 16)

A Yubikey stores public and secret ID *pid* and *sid*, and the AES key k, and is used in the following authentication protocol: The user provides a client C with the Yubikey's output *pid*, *otp* by filling it in a form. ("||" denotes concatenation.)

$$C \to S : pid \parallel otp \parallel nonce \parallel$$
$$S \to C : otp \parallel nonce \parallel hmac \parallel status$$

where *nonce* is a randomly chosen value between 8 and 20 bytes, *hmac* is a 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, the value of the internal timestamp, session counter and token counter when the key was pressed and more [13].

The server S accepts the token if and only 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 if the timestamp is in a certain window with respect to the previous timestamp received, however, our model does not include the timing of messages, therefore we ignore this (optional) check.

3 Formal Analysis in the Case of an Uncompromised Server

We performed the analysis using the tamarin protocol prover [7], a new tool for the symbolic analysis of security protocols. It supports both falsification and verification of security goals which can be expressed as first-order formulas. The

secrecy problem with unbounded nonces can be expressed in tamarin. Since this problem is undecidable [14], and due to the fact that the tool is sound and complete, it is not guaranteed to terminate. In order to achieve termination, some intervention is necessary: lemmas need to be used to cut branches in the proof attempt.

We are using tamarin for the analysis because it supports the modelling of explicit state, for example the last counter received for some Yubikey saved on the server. The popular protocol verification tool ProVerif [15], to take an example, represents protocol actions as Horn clauses, with the consequence that the set of predicates is monotonic: it is possible to derive new facts, but not to change or "forget" them. The Yubikey protocol, however, relies on the fact that once an OTP with a counter value has been accepted, the last counter value is updated. Certain OTP values that would have been accepted before will be rejected from this moment on. The resolution algorithm employed by ProVerif does not capture this (directly). There have been experiments with several abstractions that aim at incorporating state into Horn clauses [16,17], as well as the protocol analyser Scyther, but they have not been adequate for proving the absence of replay attacks for an unbounded number of session [6]. There is an extension that incorporates state synchronisation into the Strand Space Model [18], but as yet no tool support.

In tamarin, protocols are modelled as rewriting rules operating on a multiset of facts representing the protocol (Input/Output behaviour, long-term keys, short-term keys, session etc.). A *Fact* $F(t_1, \ldots, t_k)$ consists of a fact symbol F or arity k and terms $t_1, \ldots, t_i$. We will denote the set of ground facts $\mathcal{G}$. The state of the system is a finite multiset of such facts, written $\mathcal{G}^{\#}$. There is a set of "special" fact symbols used to encode the adversary's knowledge ($!\mathsf{K}$), freshness information (Fr) and messages on the network (In and Out). Other facts are used to represent the protocol state. The set of facts is partitioned into *linear* and *persistent* fact symbols. Linear facts can be consumed only once, persistent facts can be consumed arbitrarily often and are marked with an exclamation mark. Multiset rewriting rules are labelled by so-called *actions*. They consist of *premises* l, actions a and *conclusions* r, and are denoted $l -[a]\!\!\rightarrow r$. For example,

$$\mathsf{Out}(x) -[]\!\!\rightarrow !\mathsf{K}(x)$$

formalizes the adversary's capacity to eavesdrop all public communication,

$$!\mathsf{K}(x) -[\mathsf{K}(x)]\!\!\rightarrow \mathsf{In}(x)$$

his capacity to write on the network,i. e., the Dolev-Yao model. More formally, the labeled transition relation $\rightarrow_{\mathcal{M}} \subset \mathcal{G}^{\#} \times \mathcal{P}(\mathcal{G}) \times \mathcal{G}^{\#}$ and a set of ground instantiations of multiset rules $\mathcal{M}$ is defined by the following transition rule:

$$\frac{l -[a]\!\!\rightarrow r \in \mathcal{M} \quad \mathit{lfacts}(l) \subset^{\#} S \quad \mathit{pfacts}(l) \subset \mathit{set}(S)}{S \overset{\mathit{set}(a)}{\longrightarrow}_{\mathcal{M}} ((S \setminus^{\#} \mathit{lfacts}(l)) \cup^{\#} \mathit{mset}(r))}$$

where *lfact* and *pfacts* denote the linear, respectively the permanent facts from a set, *set* and *mset* transform a multiset into a set and vice versa, and

$\subset^\#, \setminus^\#, \cup^\#$ are the multiset equivalents of the subset relation, set difference and set union. The executions are then modelled by a set of traces defined as:

$$\{(A_1, \ldots, A_n) \quad | \; \exists S_1, \ldots, S_n \in \mathcal{G}^\#.\; \emptyset \xrightarrow{A_1}_{\mathcal{M}} \ldots \xrightarrow{A_n}_{\mathcal{M}} S_n \wedge \forall i \neq j.\; \forall x. \\ (S_{i+1} \setminus^\# S_i) = \{\mathsf{Fr}(x)\} \Rightarrow S_{j+1} \setminus^\# S_j) = \{\mathsf{Fr}(x)\}$$

The second condition makes sure that each fresh name is indeed different in the trace.

An important part of the modelling of the Yubikey protocol's counter value was to determine whether one value is smaller than another. We did this using four rules, two defining the relation "Is Successor of", and two defining the relation "Is smaller than". We had to simplify the modelling of the session and token counter: instead of having two counters, we just model a single counter. Since the Yubikey either increases the session counter and resets the token counter, or increases the token counter, it implements a complete lexicographical order on the pair (session counter, token counter).

$$\begin{aligned}
[\mathsf{In}(0), \mathsf{In}(S(0))] &-[\; \mathsf{IsSucc}(0, S(0)), \mathsf{IsZero}(0)]\to \quad [!\mathsf{Succ}(0, S(0))] \\
[\mathsf{In}(y), \mathsf{In}(S(y)), !\mathsf{Succ}(x, y)] &-[\; \mathsf{IsSucc}(y, S(y))]\to \quad [!\mathsf{Succ}(y, S(y))] \\
[!\mathsf{Succ}(x, y)] &-[\; \mathsf{IsSmaller}(x, y)]\to \quad [!\mathsf{Smaller}(x, y)] \\
[!\mathsf{Smaller}(x, y), !\mathsf{Succ}(y, z)] &-[\; \mathsf{IsSmaller}(x, z)]\to \quad [!\mathsf{Smaller}(x, z)]
\end{aligned}$$

Note that the adversary has to input all x and y that are in the relation. We can only express properties about the set of traces in tamarin, e. g., the terms the adversary constructs in a given trace, but not the terms he *could* construct in this trace. By requiring the adversary to produce each number in one of those relation as an input, we can ensure that they are in !K, i. e., the adversary's knowledge.

The following rule models the initialisation of a Yubikey. A fresh public id, secret ID and Yubikey are drawn, and saved on the Server and the Yubikey.

$$\begin{aligned}
&[\mathsf{Fr}(k), \mathsf{Fr}(pid), \mathsf{Fr}(sid)] \\
&-[\mathsf{Init}(pid, k), \mathsf{SecretId}(pid, sid)]\to \\
&\quad [\mathsf{Y}(pid, sid, zero), \mathsf{Server}(pid, sid, zero), !\mathsf{SharedKey}(pid, k), \mathsf{Out}(pid)]
\end{aligned}$$

The next rule models how the counter is increased when a Yubikey is plugged in. As mentioned before, we model both the session and the token counter as a single counter. We justify this by the fact that the Yubikey implements a lexicographical order. We over-approximate in the case that the Yubikey increases the session token by allowing the adversary to instantiate the rule for any counter value that is higher than the previous one, using the !Smaller relation:

$$\begin{aligned}
[!\mathsf{Y}(pid, sid), &\mathsf{Y_counter}(pid, sc), !\mathsf{Smaller}(sc, Ssc)] \\
&-[\mathsf{Yubi}(pid, Ssc)]\to [\mathsf{Y_counter}(pid, Ssc)]
\end{aligned}$$

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

$$[\mathsf{Y}(pid, sid, tc), \mathsf{Fr}(npr), !\mathsf{SharedKey}(pid, k), !\mathsf{Succ}(tc, Stc), \mathsf{Fr}(nonce)]$$
$$-[\mathsf{YubiPress}(pid, tc), \mathsf{YubiOTP}(pid, senc(< sid, tc, npr >, k))]\rightarrow$$
$$[\mathsf{Y}(pid, sid, Stc), \mathsf{Out}(< pid, senc(< sid, tc, npr >, k), nonce >)]$$

This input 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:

$$[\mathsf{Server}(pid, sid, otc), !\mathsf{SharedKey}(pid, k), !\mathsf{Smaller}(otc, tc),$$
$$\mathsf{In}(< pid, nonce, senc(< sid, tc, pr >, k) >)]$$
$$-[\mathsf{Login}(pid, sid, tc, senc(< sid, tc, pr >, k)), \mathsf{LoginCounter}(pid, otc, tc)]\rightarrow$$
$$[\mathsf{Server}(pid, sid, tc)]$$

for $otp = senc(< sid, tc, pr >, k)$. Tamarin is able to prove the absence of replay attacks, as formalized in the following lemma:

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

Note that i and j are timepoints and $\mathsf{Event}@i$ means that the trace contains a rule instantiation that produces the action Event at timepoint i. The source files and proofs are available at the following website: `http://www.lsv.ens-cachan.fr/~kunneman/yubikey/analysis/yk.tar.gz`.

We introduce two axioms to capture essential properties of the Smaller relation:

$$\forall t_1, t_2, a, b, c.\mathsf{IsSmaller}(a, b)@t_1 \wedge \mathsf{IsSmaller}(b, c)@t_2 \Rightarrow \exists t_3.\mathsf{IsSmaller}(a, c)@t_3 \quad \text{(Transitivity)}$$

$$\neg(\exists a, t.\mathsf{IsSmaller}(a, a)@t) \quad \text{(Antisymmetrie)}$$

It is necessary to axiomatize in particular transitivity, since the adversary is not obliged to produce a !Smaller fact for every value that is described by the !Succ relation. We stress that those lemmas are the only ones we consider axioms, all remaining lemmas can be proven using tamarin.

The absence of replay attacks is proven by showing the following, stronger property to hold true, using an auxiliary lemma that we will describe afterwards:

$$(\forall pid, otc1, tc1, otc2, tc2, t_1, t_2.$$
$$\mathsf{LoginCounter}(pid, otc1, tc1)@t_1 \wedge \mathsf{LoginCounter}(pid, otc2, tc2)@t_2 \wedge t_1 < t_2$$
$$\Rightarrow \exists t_3.\mathsf{IsSmaller}(tc1, tc2)@t_3$$

Intuitively, this means that counter values are increasing in time. The following auxiliary lemma is necessary for tamarin to derive the proof automatically.

This lemma formalizes the idea that for every successful login with counter value x, the Yubikey's button must have been pressed before while its own counter was set to x, too.

$$\forall pid, sid, x, otp, t_2.$$
$$\mathsf{Login}(pid, sid, x, otp)@t_2 \Rightarrow \exists t_1.\mathsf{YubiPress}(pid, x)@t_1 \wedge t_1 < t_2$$

Note that in this model, the adversary has no direct access to the server, he can only control the network. A stronger attack model is discussed in the next section.

4 The YubiHSM

The YubiHSM is also a USB device about 5 times thicker than a Yubikey. According to the Yubico literature it "provides a low-cost [$500] way to move out sensitive information and cryptographic operations away from a vulnerable computer environment without having to invest in expensive dedicated Hardware Security Modules (HSMs)" [19]. The YubiHSM stores a very limited number of AES keys in a way that the server can use them to perform cryptographic operations without the key values ever appearing in the server's memory. These 'master keys' are generated during configuration time and can neither be modified nor read at runtime. The master keys are used to encrypt working keys which can then stored safely on the server's hard disk. The working keys are encrypted inside so-called AEADs (blocks produced by authenticated encryption with associated data). In order to produce or decrypt an AEAD, an AES key and a piece of associated data is required. The YubiHSM uses CCM mode to obtain an AEAD algorithm from the AES block cipher [20].

In the case of the Yubikey protocol, AEADs are used to store the keys the server shares with the Yubikeys, and the associated data is the public ID and the key-handle used to reference the AES key. The idea here is that since the master keys of the YubiHSM cannot be extracted, the attacker never learns the value of any Yubikey AES keys, even if he successfully attacks the server. While he is in control of the server, he is (of course) able to grant or deny authentication to any client at will. However, if the attack is detected and the attacker loses access to the server, it should not be necessary to replace or rewrite the Yubikeys that are in circulation.

5 Two Attacks on the Implementation of Authenticated Encryption

The YubiHSM provides access to about 22 commands that can be activated or de-activated globally, or per key, during configuration. We first examined the YubiHSM API in its default configuration, discovering the following two attacks which led to a security advisory given issued by Yubikey in February 2012 [9].

The attacks use the following two commands: AES_ECB_BLOCK_ENCRYPT takes a handle to an AES key and a plaintext of length of one AES block (16 Bytes) and applies the raw block cipher. YSM_AEAD_GENERATE takes a nonce, a handle to an AES key and some data and outputs an AEAD. More precisely, but still simplified for our purposes, it computes:

$$AEAD(nonce, kh, data) = \left(\left(\Big\| _{i=0}^{\lceil \frac{|data|}{blocksize} \rceil} AES(k, counter_i) \right) \bigoplus data \right) || mac$$

where k is the key referenced by the key-handle kh, $counter_i$ is a counter that is completely determined by $kh, nonce, i$ and the length of $data$ and $blocksize$ is 16 bytes. For the precise definition of mac and $counter$, we refer to RFC 3610 [20]. Figure 2 depicts the counter mode of operation, the method used to derive the AEAD up to the MAC.

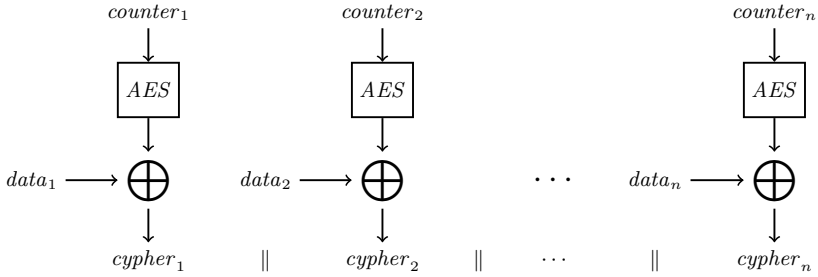

Fig. 2. AES in counter mode (simplified)

The AEADs used to store keys for decrypting OTPs in the Yubikey protocol are special cases: the plaintext is a concatenation of the respective Yubikey's AES key and secret device ID (22 bytes in total), and *nonce* consists of the Yubikey's public id.

An attacker with access to the command AES_ECB_BLOCK_ENCRYPT is able to decrypt an AEAD by recreating the blocks of the key-stream, i. e., $AES(k, counter_i)$. He xors the result with the AEAD truncated by 8 bytes (the length of *mac*) and yields *data*. When the attacker is able to compromise the server, he learns the AEAD and the key-handle used to produce it. Since the nonce is the public ID of the Yubikey, he can compute $counter_i$ and, using AES_ECB_BLOCK_ENCRYPT the key-stream. It is in the nature of the counter-mode that encryption and decryption are the same operation. According to the reference manual[19, Section 4.3], "the YubiHSM intentionally does not provide any functions that decrypts an AEAD and returns it in clear text, either fully or partial.". We therefore consider the protection of the AEAD's contents a security goal of the YubiHSM, which is violated by this attack. The attack can

be prevented by disabling the AES_ECB_BLOCK_ENCRYPT command on the relevant key handles at configuration time.

The second attack uses only YSM_AEAD_GENERATE: if the attacker produces $AEAD(nonce, kh, 0^l)$ for the same handle kh and $nonce$ a previously generated AEAD of length l was created with (he can discard mac). Again he directly recovers the key-stream. Once again, it is possible to decrypt AEADs. This attack is worse than the first one, because YSM_AEAD_GENERATE is necessary for the set-up of Yubikeys. Note that the attack applies also to the commands YSM_RANDOM_AEAD_GENERATE and YSM_BUFFER_-AEAD_GENERATE [19, p. 28-29].

This second attack is harder to prevent, since in order to set up Yubikeys with their AES keys, the YSM_AEAD_GENERATE command must at some point be enabled. The security advisory suggests that the threat can be "mitigated by observing that a YubiHSM used to generate AEADs is guarded closely to not permit maliciously crafted input." In the next section, we try to interpret this advice into a concrete configuration for which can prove security of the sensitive keys. Then, in section 7, we describe practical ways in which this configuration could be used.

6 Analysis in the Case of Server Compromise

In the following, we will assume the following corruption scenario: in addition to the capacities described in Section 3, the attacker can read the AEADs stored on the server and he can access the HSM. Every AEAD is created using the same key on the HSM, the handle to this key is made public. The public ID is given to the adversary when the Yubikey is set up. Counter values are guessable, so there is no need to give the adversary explicit access to this data. The adversary is still not able to directly read the data stored on the Yubikey or YubiHSM. Note that is this situation, the attacker can trivially approve or deny authorisation requests to the server, hence we cannot expect to show absence of replay attacks. We are rather interested in whether the attacker can recover the secret keys used to create OTPs, which would allow him to continue to obtain authorisation even once he is denied access to the server.

We model the xor operator in a very simplified manner. The equational theory we employ allows to recover the two attacks described in Section 5, but it does not capture all attacks that the xor-operator might permit in this context. For this reason, the positive security results in this section have to be taken with caution. We model xor using the function symbols xor, $dexor1$ and $dexor2$ and the equations $dexor1(xor(a,b),a) = b$ and $dexor2(xor(a,b),b) = a$. Using this equational theory, we are able to rediscover the attacks described in the previous section. The current version of tamarin (0.6.1.0) does not have built-in support yet, but it is planned for future releases.

The adversary is able to produce !Succ and !Smaller facts, in the same way as described in Section 3. We initialise the YubiHSM with exactly one key-handle:

$$[\mathsf{Fr}(k), \mathsf{Fr}(kh)] -[\mathsf{MasterKey}(k), \mathsf{OneTime}()]\rightarrow [!\mathsf{HSM}(kh, k), \mathsf{Out}(kh), !\mathsf{YSM\_AEAD\_YUBIKEY\_OTP\_DECODE}(kh)]$$

We make sure that this rule is only instantiated once by adding a corresponding condition to the lemmas we prove.

The following rules model the fact that the adversary can communicate with the YubiHSM and read the list of AEADs stored on the authentication server.

$$\begin{aligned}
[\mathsf{OutHSM}(x)] -[& \quad \mathsf{HSMRead}(x)]\rightarrow & [\mathsf{Out}(x)] \\
[\mathsf{In}(x)] -[& \quad \mathsf{HSMWrite}(x)]\rightarrow & [\mathsf{InHSM}(x)] \\
[!\mathsf{S_AEAD}(pid, aead)] -[& \quad \mathsf{AEADRead}(aead), \mathsf{HSMRead}(aead)]\rightarrow & [\mathsf{Out}(aead)]
\end{aligned}$$

The next rules aim at modelling the HSM. We modelled a set of 4 rules in total, but only YSM_AEAD_YUBIKEY_OTP_DECODE is used. (YSM_AEAD_GENERATE is directly incorporated into the rule BuyANewYubikey, see below.)

$$\begin{aligned}
&[\mathsf{InHSM}(< did, kh, aead, otp >), !\mathsf{HSM}(kh, k), \\
&\quad !\mathsf{YSM_AEAD_YUBIKEY_OTP_DECODE}(kh)] \\
&-[\mathsf{OtpDecode}(k2, k, < did, sc, rand >, sc, xor(senc(ks, k), < k2, did >), mac) \\
&\quad \mathsf{OtpDecodeMaster}(k2, k)]\rightarrow [\mathsf{OutHSM}(sc)]
\end{aligned}$$

where $ks = \mathsf{keystream}(kh, N)$, $mac = \mathsf{mac}(< k2, did >, k)$, $aead =< \mathsf{xor}(senc(ks, k), < k2, did >), mac >$ and $otp = \mathsf{senc}(< did, sc, rand >, k2)$.

The rules for emitting the OTP and the login are modelled in a way very similar to Section 3, but of course we model the encryption used inside the AEAD in more detail. Here, the server-side rule for the login.

$$\begin{aligned}
&[\mathsf{In}(< pid, nonce, senc(< sid, tc, pr >, k2) >), !\mathsf{HSM}(kh, k), !\mathsf{S_sid}(pid, sid), \\
&!\mathsf{S_AEAD}(pid, aead), \quad \mathsf{S_Counter}(pid, otc), !\mathsf{Smaller}(otc, tc)] \\
&\quad -[\mathsf{Login}(pid, tc, senc(< sid, tc, pr >, k2))]\rightarrow [\mathsf{S_Counter}(pid, tc)]
\end{aligned}$$

where ks, mac and $aead$ are defined as before.

Tamarin is able to prove that, within our limited model of xor, the adversary never learns a Yubikey AES key or a YubiHSM master key - in other words, AEADs, as well as the key used to produce them, stay confidential. The proof does not need human intervention, however, some additional typing invariants are needed in order to reach termination. For instance, the following invariant is used to proof that a key k_2 shared between the authentication server and the Yubikey can only be learned when, either the key used to encrypt the AEADs is leaked, of when the one-time condition is violated.

$$\begin{aligned}
&\forall t_1, t_2, pid, k2.\mathsf{Init}(pid, k2)@t_1 \wedge \mathsf{K}(k2)@t_2 \\
&\Rightarrow (\exists t_3, t_4, k.\mathsf{K}(k)@t_3 \wedge \mathsf{MasterKey}(k)@t_4 \wedge t_3 < t_2) \\
&\quad \vee (\exists t_3, t_4 . OneTime()@t_3 \wedge \mathsf{OneTime}()@t_4 \wedge \neg(t_3 = t_4))
\end{aligned}$$

7 Evaluation

The positive and negative results in this paper provide formal criteria to evaluate the security of the Yubikey protocol in different scenarios.

Positive Results: Under the assumption that the adversary can control the network, but is not able to compromise the client or the authentication server, we have shown he cannot mount replay attack. Furthermore, if a YubiHSM is used configured such that YSM_AEAD_YUBIKEY_OTP_DECODE is the only available command, then even in case the adversary is able to compromise the server, the Yubikey AES keys remain secure. All these results are subject to our abstract modelling of cryptography and the algebraic properties of XOR of course.

Since the Yubikeys need to be provisioned with their AES keys and secret identities must be stored in the AEADs, we propose two set-ups that can be used to obtain the configuration used in the analysis:

1. *One Server, One YubiHSM:* There is a set-up phase which serves the purpose of producing AEADs (using any of the YSM_AEAD_GENERATE commands) and writing the key and secret/public identity on the Yubikey. This phase should take place in a secure environment. Afterwards, the YubiHSM is returned to configuration mode and all commands disabled except YSM_AEAD_YUBIKEY_OTP_DECODE. In this set-up, only one YubiHSM is needed, but it is not possible to add new Yubikeys once the second phase has begun without taking the server off-line and returning the YubiHSM to configuration mode. Note that switching the YubiHSM into configuration mode requires physical access to the device, hence would not be possible for an attacker who has remotely compromised the server.
2. *Two Servers, Two YubiHSMs:* There is one server that handles the authentication protocol, and one that handles the set-up of the Yubikeys. The latter is isolated from the network and only used for this very purpose, so we consider it a secure environment. We configure two YubiHSMs such that they store the same master-key (the key used to produce AEADs). The first is used for the authentication server and has only YSM_AEAD_YUBIKEY_OTP_DECODE set to true, the second is used in the set-up server and has only YSM_AEAD_GENERATE set to true. The set-up server produces the list of public ids and corresponding AEADs, which is transferred to the authentication server in a secure way, for example in fixed intervals (every night) using fresh USB keys. The transfer does not necessarily have to provide integrity or secrecy (as the adversary can block the authentication via the network, anyway), but it should only be allowed in one direction.

Reading between the lines (since no YubiHSM configuration details are given) it seems that Yubico themselves use this set-up to provision Yubikeys [21].

Negative Results: In case either of the permissions AES_ECB_BLOCK_ENCRYPT or YSM_AEAD_GENERATE are activated on a master key handle (which by default they both are), the YubiHSM does protect the keys used to produce one-time passwords encrypted under that master key. Since YSM_AEAD_GENERATE (or YSM_BUFFER_AEAD_GENERATE) are needed in order to set a Yubikey up, this means that separate setup and authorisation configurations have to be used in order to benefit from the use of the YubiHSM, i. e., have a higher level of security than in the case where the keys are stored unencrypted on the hard disk. Unfortunately, open source code available on the web in e.g. the yhsmpam project [22], designed to use the YubiHSM to protect passwords from server compromise, uses the insecure configuration, i.e. one YubiHSM with both YSM_AEAD_GENERATE and (in this case) YSM_AEAD_DECRYPT_CMP enabled, and hence would not provide the security intended.

Possible changes to the YubiHSM: We will now discuss two possible countermeasures against this kind of attack that could be incorporated into future versions of the YubiHSM to allow a single device to be used securely, the first which may be seen as a kind of stop-gap measure, the second which is a more satisfactory solution using more suitable crypto:

1. *AEAD_GENERATE with a randomly drawn nonce:* All three YubiHSM commands to generate AEADs (YSM_AEAD_GENERATE, YSM_BUFFER_AEAD_GENERATE and YSM_RANDOM_AEAD_GENERATE) allow the user to supply the nonce that is used. This would not be possible if they were replaced by a command similar to YSM_AEAD_GENERATE that chooses the nonce randomly and outputs it at the end, so it is possible to use the nonce as the public ID of the Yubikey. However, even in this case there is an online guessing attack on the HSM: an AEAD can be decrypted if the right nonce is guessed. We can assume that the adversary has gathered a set of honestly generated OTPs, so he is able to recognize the correct nonce. Since the nonce space is rather small (2^{48}) in comparison to the key-space of AES-128, the adversary could perform a brute-force search. We measured the amount of time it takes to perform 10 000 YSM_AEAD_GENERATE operations on the YubiHSM. The average value is 0.2178 ms, so it would take approximately 1900 years to traverse the nonce space. Even so this is not completely reassuring.
2. *SIV-mode:* The SIV mode of operation [23] is designed to be resistant to repeated IVs. It is an authenticated encryption mode that works by deriving the IV from the MAC that will be used for authentication. As such it is deterministic - two identical plaintexts will have an identical ciphertext - but the encryption function cannot be inverted in the same way as CCM mode by giving the same IV (the encryption function does not take an IV as input, the only way to force the same IV to be used is to give the same plaintext). This would seem to suit the requirements of the YubiHSM very well, since it is only keys that will be encrypted hence the chances of repeating a plaintext

are negligible. Even if the same key is put on two different Yubikeys, they will have different private IDs yielding different AEADs. In our view this is the most satisfactory solution.

Methodology: Since the tamarin prover could not derive the results in this paper without user intervention, we think it is valuable to give some information about the way we derived those results.

We first modelled the protocol using multiset rewriting rules, which was a straight-forward task. Then, we stated a sanity lemma saying "There does not exist a trace that corresponds to a successful protocol run" to verify that our model is sane. Tamarin should be able to derive a counter-example, which is a proof that a correct protocol run is possible.

We stated the security property we wanted to prove, e. g., the absence of replay-attacks. Since tamarin did not produce a proof on its own, we investigated the proof derivation in interactive mode. We deduced lemmas that seemed necessary to cut branches in the proof that are looping, so-called typing invariants. An example is YSM_AEAD_YUBIKEY_OTP_DECODE: it outputs a subterm of its input, namely the counter value. Whenever tamarin tries to derive a term t, it uses backward induction to find combination of AEAD and OTP that allows to conclude that the result of this operation is t. Meier, Cremers and Basin propose a technique they call *decryption-chain reasoning* in [24, Section 3b] that we used to formulate our typing invariant. Once the invariant is stated, it needs to be proven. Sometimes the invariant depends on another invariant, that needs to be found manually. We used trial and error to find a set of invariants that lead to a successful verification of the security property, and then minimised this set of lemmas by experimentally deleting them to find out which ones were strictly necessary.

All in all, it took about 1 month to do the analysis presented in this work. The modelling of the protocol took no more than half a day. Finding a modelling of the natural numbers and the "Is smaller than" relation took a week. The lion's share of the time was spent in searching the right invariants for termination. The running time of tamarin is acceptable: proving the absence of replay-attacks in case of an uncompromised server takes around 35 seconds, proving confidentiality of keys in the case of a compromised server takes around 50 seconds, both on a 2.4GHz Intel Core 2 Duo with 4GB RAM.

8 Conclusions

We were able to show the absence of replay-attacks of the Yubikey protocol in the formal model. This has been attempted before, using a variety of protocol analysis tools, but it was previously only possible for a fixed number of nonces. This shows, that verification based on multiset-rewriting rules is a promising approach for realistic protocols, as long as the right invariants can be found. Perhaps surprisingly, the modelling of the "is smaller than" relation proves to be a surmountable problem.

To evaluate the YubiHSM, a small device that can potentially perform cryptographic operations without revealing the key to the computer it is connected to, we considered a more challenging scenario. Here, the attacker can access the server and communicate with this device. Two attacks show that in the default configuration, the encryption operations are implemented in a way such that they do not provide confidentiality of plaintexts. The user must either set up the authentication server and the YubiHSM such that provisioning and authorisation commands are never available at the same time on the same server. We proposed a change to the cryptographic design that would relax this restriction.

We learned that it is possible to obtain formal results on the YubiKey and YubiHSM for an unbounded model using tamarin, which gives us hope for the unbounded formal analysis of other Security APIs, for example PKCS#11 and the TPM. However, there are plenty of improvements to be made: we currently treat the session and token counter on the Yubikey as a single value, and simplify the algebraic theory of xor considerably. The treatment of xor is expected to be included in future versions of the tamarin tool. Our own work will concentrate on developing a methodology for deriving the lemmas tamarin needs to obtain a proof automatically, thus permitting more automation and more expressive models.

References

1. Bonneau, J., Herley, C., Oorschot, P.C.v., Stajano, F.: The quest to replace passwords: a framework for comparative evaluation of Web authentication schemes. Technical Report UCAM-CL-TR-817, University of Cambridge, Computer Laboratory (March 2012) To appear in Proceedings of IEEE Symposium on Security and Privacy 2012.
2. AB, Y.: Yubico customer list. available at: `http://www.yubico.com/references`
3. AB, Y.: Yubikey security evaluation: Discussion of security properties and best practices. available at `http://static.yubico.com/var/uploads/pdfs/Security_Evaluation_2009-09-09.pdf` (September 2009) v2.0.
4. Björck, F.: Yubikey security weaknesses. On *Security DJ* Blog, `http://web.archive.org/web/20100203110742/http://security.dj/?p=4` (February 2009)
5. Björck, F.: Increased security for yubikey. On *Security DJ* Blog, `http://web.archive.org/web/20100725005817/http://security.dj/?p=154` (August 2009)
6. Vamanu, L.: Formal analysis of Yubikey, available online at `http://n.ethz.ch/~lvamanu/download/YubiKeyAnalysis.pdf`. Master's thesis, École normale supérieure de Cachan (August 2011)
7. Schmidt, B., Meier, S., Cremers, C., Basin., D.: Automated analysis of Diffie-Hellman protocols and advanced security properties. In: CSF (to appear), IEEE Computer Society (2012)
8. Kaminsky, D.: On the RSA SecurID compromise. Available at `http://dankaminsky.com/2011/06/09/securid/`. (June 2011)
9. Yubico Inc.: Yubihsm 1.0 security advisory 2012-01 (published online: `http://static.yubico.com/var/uploads/pdfs/SecurityAdvisory%202012-02-13.pdf`) (February 2012)

10. Yubico AB Kungsgatan 37, 111 56 Stockholm Sweden: The YubiKey Manual - Usage, configuration and introduction of basic concepts (Version 2.2), available at: http://www.yubico.com/documentation. (June 2010)
11. Yubico AB Kungsgatan 37, 111 56 Stockholm Sweden: YubiKey Authentication Module Design Guide and Best Practices (Version 1.0), available at: http://www.yubico.com/documentation
12. Kamikaze28, et al.: Specification of the Yubikey operation in the Yubico wiki. available at: http://wiki.yubico.com/wiki/index.php/Yubikey (June 2012)
13. The yubikey-val-server-php project: Validation protocol version 2.0. available at: http://code.google.com/p/yubikey-val-server-php/wiki/ValidationProtocolV20 (October 2011)
14. Durgin, N., Lincoln, P., Mitchell, J., Scedrov, A.: Undecidability of bounded security protocols. In Heintze, N., Clarke, E., eds.: Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP, Trento, Italy. (July 1999) Electronic proceedings available at http://www.cs.bell-labs.com/who/nch/fmsp99/program.html.
15. Blanchet, B.: Automatic verification of correspondences for security protocols. Journal of Computer Security **17**(4) (July 2009) 363–434
16. Arapinis, M., Ritter, E., Ryan, M.D.: Statverif: Verification of stateful processes. In: CSF, IEEE Computer Society (2011) 33–47
17. Mödersheim, S.: Abstraction by set-membership: verifying security protocols and web services with databases. [25] 351–360
18. Guttman, J.D.: State and progress in strand spaces: Proving fair exchange. J. Autom. Reasoning **48**(2) (2012) 159–195
19. Yubico AB Kungsgatan 37, 111 56 Stockholm Sweden: Yubico YubiHSM - Cryptographic Hardware Security Module (Version 1.0), available at: http://www.yubico.com/documentation. (September 2011)
20. Whiting, D., Housley, R., Ferguson, N.: Counter with CBC-MAC (CCM). RFC 3610 (Informational) (September 2003)
21. Yubico AB Kungsgatan 37, 111 56 Stockholm Sweden: Yubicloud Validation Service - (Version 1.1), available at: http://www.yubico.com/documentation. (May 2012)
22. Habets, T.: Yubihsm login helper program. available at: http://code.google.com/p/yhsmpam/
23. Rogaway, P., Shrimpton, T.: Deterministic authenticated encryption: A provable-security treatment of the keywrap problem (2006)
24. Meier, S., Cremers, C.J.F., Basin, D.A.: Strong invariants for the efficient construction of machine-checked protocol security proofs. [25] 231–245
25. Al-Shaer, E., Keromytis, A.D., Shmatikov, V., eds.: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010. In Al-Shaer, E., Keromytis, A.D., Shmatikov, V., eds.: ACM Conference on Computer and Communications Security, ACM (2010)