EPiC Series in Computing

Volume 57, 2018, Pages 400–417

LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

EPiC Computing

Formal verification of the YubiKey and YubiHSM APIs in Maude-NPA*

Antonio González-Burgueño[1], Damián Aparicio-Sánchez[2], Santiago Escobar[2], Catherine Meadows[3], and José Meseguer[4]

[1] University of Oslo, Oslo, Norway antonigo@ifi.uio.no
[2] DSIC-ELP, Universitat Politècnica de València, Valencia, Spain {daapsnc@dsic.upv.es, sescobar@dsic.upv.es}
[3] Naval Research Laboratory, Washington DC, USA meadows@itd.nrl.navy.mil
[4] University of Illinois at Urbana-Champaign, Urbana IL, USA meseguer@illinois.edu

Abstract

We perform an automated analysis of two devices developed by Yubico: *YubiKey*, designed to authenticate a user to network-based services, and *YubiHSM*, Yubico's hardware security module. Both are analyzed using the Maude-NPA cryptographic protocol analyzer. Although previous work has been done applying formal tools to these devices, there has not been any completely automated analysis. This is not surprising, because both YubiKey and YubiHSM, which make use of cryptographic APIs, involve a number of complex features: (i) discrete time in the form of *Lamport clocks*, (ii) a mutable memory for storing previously seen keys or nonces, (iii) event-based properties that require an analysis of sequences of actions, and (iv) reasoning modulo exclusive-or. Maude-NPA has provided support for exclusive-or for years but has not provided support for the other three features, which we show can also be supported by using constraints on natural numbers, protocol composition and reasoning modulo associativity. In this work, we have been able to automatically prove security properties of YubiKey and find the known attacks on the YubiHSM, in both cases beyond the capabilities of previous work using the Tamarin Prover due to the need of auxiliary user-defined lemmas and limited support for exclusive-or. Tamarin has recently been endowed with exclusive-or and we have rewritten the original specification of YubiHSM in Tamarin to use exclusive-or, confirming that both attacks on YubiHSM can be carried out by this recent version of Tamarin.

1 Introduction

Nowadays there exist several security tokens having the form of a smartcard or an USB device, which are designed for protecting cryptographic values from an intruder, e.g, hosting service,

*Partially supported by the EU (FEDER) and the Spanish MINECO under grant TIN 2015-69175-C4-1-R, by the Generalitat Valenciana under grant PROMETEOII/2015/013, by the US Air Force Office of Scientific Research under award number FA9550-17-1-0286, and by NRL under contract number N00173-17-1-G002.

email, e-commerce, online banks, etc. They are also used to ease authentication for the authorized users of a service, e.g., if you are using a service that verifies your Personal Identification Number (PIN), the same service should not be used for checking your flights, reading your emails, etc. By using an Application Programming Interface (API) to separate the service from the authentication system, such problems can be prevented.

Yubico is a leading company on open authentication standards and has developed two core inventions: the *YubiKey*, a small USB designed to authenticate a user against network-based services, and the *YubiHSM*, Yubico's hardware security module (HSM). The YubiKey allows for the secure authentication of a user against network-based services by considering different methods: one-time password (OTP), public key encryption, public key authentication, and the Universal 2nd Factor (U2F) protocol [5]. YubiKey works by using a secret value (i.e., a running counter) and some random values, all encrypted using a 128 bit Advanced Encryption Standard (AES). An important feature of YubiKey is that it is independent of the operating system and does not require any installation, because it works with the USB system drivers. YubiHSM is intended to operate in conjunction with a host application. It supports several modes of operation, but the key concept is a symmetric scheme where one device at one location can generate a secure data element in a secure environment. Although the main application area is for securing YubiKey's OTP authentication/validation operations, the use of several generic cryptographic primitives allows a wider range of applications. The increasing success of YubiKey and YubiHSM has led to its use by governments, universities and companies like Google, Facebook, Dropbox, CERN, Bank of America etc., including more than 30,000 customers [4].

Cryptographic Application Programmer Interfaces (Crypto APIs) are commonly used to secure interaction between applications and hardware security module (HSMs), and are used in both YubiKey and YubiHSM. However, many crypto APIs have been subjected to intruder manipulation to disclose relevant information, as is the case for YubiHSM. In [19,20], Künnemann and Steel show two kinds of attacks on the first released version of the YubiHSM API: (i) if the intruder had access to the server running YubiKey, where AES keys are generated, then it was able to obtain plaintext in the clear; and (ii) even if the intruder had no access to the server running YubiKey, it could use previous nonces to obtain AES keys. However, there has not been any completely automated analysis of these two attacks to date because both YubiKey and YubiHSM involve a number of complex features: (1) discrete time in the form of *Lamport clocks*, (2) a mutable memory for storing previously seen keys or nonces, (3) event-based properties that require an analysis of sequences of actions, and (4) reasoning modulo exclusive-or. Maude-NPA [1] has provided support for exclusive-or for years but has not provided support for the other three features, which we show can also be supported by using constraints on natural numbers, protocol composition and reasoning modulo associativity.

This paper is the third in a series using Maude-NPA to analyze cryptographic APIs; earlier work appeared in [17,18]. We find this problem area one of particular interest for two reasons. First, these APIs often use exclusive-or and this gives us the opportunity to explore how well Maude-NPA can be applied to protocols that use exclusive-or. Secondly, cryptographic APIs offer a number of other challenging features and this allows us to explore how Maude-NPA can handle them. Our analysis was carried out on generation 2 of YubiKey and version 0.9.8 beta of the YubiHSM, as was the analysis of [19]. In order to facilitate comparison with earlier work, our formal specifications of YubiKey and YubiHSM follow those of [19] as closely as possible.

Contributions

1. We automatically prove the secrecy and authentication properties of YubiKey and find

both attacks on YubiHSM, beyond the capabilities of Tamarin [2] in the earlier analysis in [19, 20], which was only able to find one attack due to limited support for exclusive-or. Tamarin has recently been endowed with exclusive-or in [11]. In Section 6.1 we have rewritten the original specification of YubiHSM in Tamarin to use exclusive-or and checked that both attacks on YubiHSM can now be carried out by Tamarin.

2. Our analysis was completely automatic and either found an attack or terminated with a finite search graph, showing that no attack of that kind exists. That is, Maude-NPA did not need any human guiding or auxiliary lemmas. In contrast, both the earlier analysis in [19, 20] and our own analysis using the latest version of Tamarin in Section 6.1 involved some auxiliary user-defined lemmas in order to prove properties of YubiKey and YubiHSM. Mutable state was considered a very difficult problem for a very long time until Tamarin came along [22], and often enough requires manual intervention in the form of auxiliary lemmas, i.e., those that are proven by Tamarin but are not security properties themselves, and have to be specified by the savvy user. A push-button verifier, such as Maude-NPA, has usually a much broader appeal for the general audience. In Maude-NPA, system specification is described by state transitions manipulating strands; without any possibility of incorporating properties, such as auxiliary lemmas, beyond the actual equational properties of the protocol. The analysis of security properties in Maude-NPA relies on various sound and complete state space reduction techniques that help to identify unreachable and redundant states [13].

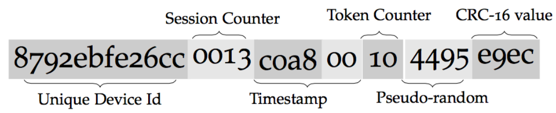
3. We implemented Lamport clocks, mutable memory, and event-based properties for the first time in Maude-NPA, even though the tool does not support these natively, by using constraints on natural numbers, protocol composition and reasoning modulo associativity. These techniques should be applicable to protocols with similar properties.

Plan of the paper. In Sections 2 and 3 we give an overview of the YubiKey and YubiHSM, respectively. In Section 4 we explain how Lamport clocks, mutable memory, and event-based properties are implemented in Maude-NPA. In Section 5 we describe how we specified YubiKey and YubiHSM in Maude-NPA. In Section 6 we describe our experiments. Finally, in Section 7 we discuss related work, and we conclude in Section 8.

2 The YubiKey Device

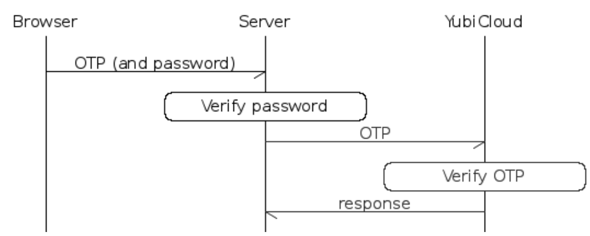
The YubiKey USB device [28] is an authentication device capable of generating One Time Passwords (OTPs). The YubiKey connects to a USB port and identifies itself as a standard USB device such as a keyboard, which allows it to be used in most computing environments using the system's native drivers.

We will focus on the YubiKey OTP mode, a mode that uses a button physically located on the YubiKey. When this button is pressed, it emits a string that can be verified only once against a server in order to receive the permission to access a service. Furthermore, a request for a new authentication token is triggered also by touching the YubiKey button. As a result of this request, 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. An OTP has the following concatenated fields [27]:



Each OTP sent by the YubiKey is encrypted using an AES key. Thus, the YubiKey authentication server accepts an OTP only if both it decrypts under the appropriate AES key and the token counter stored in the OTP is larger than the token counter stored in the last OTP received by the server. The token counter is used as a *Lamport clock* [21], i.e., it is used to determine the order of events in a distributed concurrent system by using a counter that both has a minimum value (e.g., 0) and has a minimum *tick* (increment of the counter).

The authentication protocol of YubiKey involves three roles: (i) the user, (ii) the service, and (iii) the verification server. The user can have access to the service if it provides its own valid OTP generated by the YubiKey; its validity is verified by the verification server as explained before. The following example shows a user (Browser), a service (YubiCloud), and a verification server running the YubiKey API.



Since both the YubiKey and the server need to store information, e.g., the last received token counter, different predicates are defined in [19]: (i) *SharedKey(pid,k)* to represent the key k that is shared with the Yubikey public ID *pid*, (ii) *Y(pid,sid)* that stores the corresponding secret ID *sid* associated to the Yubikey public ID *pid*, (iii) *Server(pid,sid,token-counter)* that links the Yubikey public ID *pid* with the secret ID *sid* and the value of the last received counter *token-counter*, and (iv) *YubiCounter(pid,token-counter)* that represents that the current counter value *token-counter* is stored on the Yubikey. Following [19], all predicates are stored together in a shared global memory.

The YubiKey OTP generation scheme can be described by the following interaction.

1. The initialization of the YubiKey device takes place. A fresh public ID (*pid*), a secret ID (*sid*) and a YubiKey key (k) are generated. Any interaction between the YubiKey and the server will involve all three elements *pid*, *sid* and *k*. There are also two token counters, one stored on the Server and another stored on the YubiKey. All predicates are initialized in the global memory.

2. The YubiKey is plugged in. Every time the YubiKey is plugged in, the YubiKey token counter must be increased. However, we consider the compromised scenario of [19] in which the attacker has temporary access to the authentication server and it can produce all counter values, thus adding a new token counter as an input to the command and checking that it must be bigger than the old stored token counter. Figure 1 shows a graphical representation of the plugin event, including the input, output, and updated predicate.

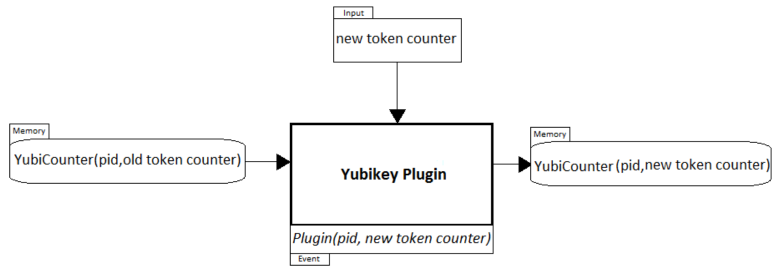


Figure 1: YubiKey Plugin API Command

3. The user pushes the YubiKey OTP generation button and generates a byte string formed by the *sid*, the YubiKey token counter, and a random number. The byte string is encrypted using a symmetric encryption operator and the saved key k. The YubiKey token counter is also increased. According to the compromised scenario, the YubiKey token counter must be provided as input. Figure 2 shows a graphical representation of the button-pressing event, including the input, output, and updated predicate.

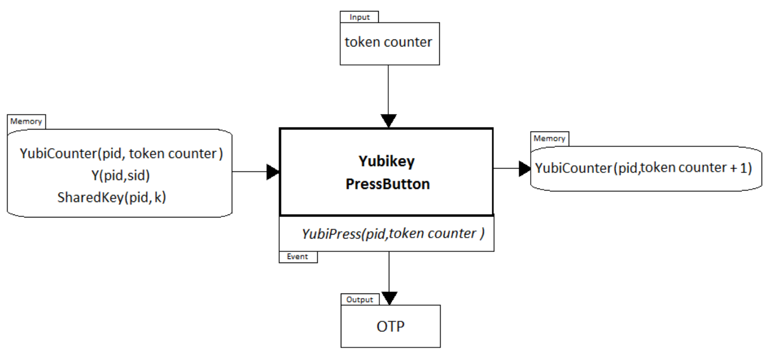


Figure 2: YubiKey Press Button Command

4. Upon reception of the generated OTP string, the basic verification steps are:

   4.1 The byte string is decrypted, and if it is not valid the OTP is rejected.

   4.2 The token counter stored in the OTP is compared with the server token counter. If smaller than or equal to the server token counter, the received OTP is rejected as a replay. According to the compromised scenario, the server token counter must be provided as input.

   4.3 A successful login must have been preceded by a button press for the same counter value, and there is not a second distinct login for this counter value. In this paper we omit this check and show (Section 5.1 below) that this property is always guaranteed, assuming that the checks on the byte string and token counter succeed.

   4.4 If all the checks succeed, the token counter stored in the OTP is stored as the server token counter and the OTP is accepted as valid.

   Figure 3 shows a graphical representation of the login event, including the input, output, and updated predicate.

In [19, 20], Künnemann and Steel were able to prove several properties:

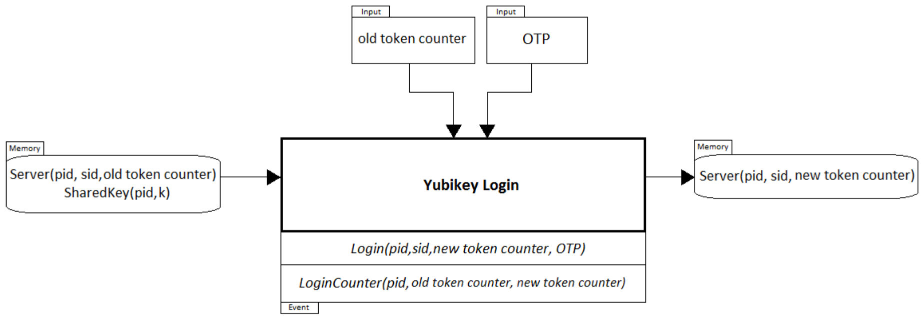


Figure 3: YubiKey Login Command

(a) Absence of replay attacks, i.e., there are no two distinct logins that accept the same counter.

(b) Correspondence between pressing the button on a YubiKey and a successful login. In other words, a successful login must have been preceded by a button pressed for the same counter value. Furthermore, there is no second distinct login for this counter value.

(c) Counter values are different over time, i.e., the counter values associated to logins are monotonically increasing in time. Therefore, if one login has a smaller counter than the other, then it must have occurred earlier.

Note that the verification of properties (b) and (c) in [20] using Tamarin involved additional user-defined lemmas (see Section 6.1).

3 The YubiHSM Device

Yubico also distributes a USB device that works as an application-specific Hardware Security Module (HSM) to protect the YubiKey AES keys. The YubiHSM [29] stores a very limited number of AES keys so that the server can use them to perform cryptographic operations without the key values ever appearing in the server's memory. The YubiHSM is designed to protect the YubiKey AES keys, when an authentication server is compromised, by encrypting the AES keys using a master key stored inside the YubiHSM.

In addition, the YubiHSM can decrypt an indefinite number of YubiKey's OTP's with secure storage of the AES keys on the host computer. The AES keys are only readable to the YubiHSM through the use of *Authenticated Encryption with Associated Data (AEAD)*. The AEAD uses a cryptographic method that provides both confidentiality and authenticity. An AEAD consists of two parts: (i) the encryption of a message using the cryptographic mode of operation, called *counter mode*, and (ii) a *message authentication code (MAC)* taken over the encrypted message. In order to construct, decrypt or verify an AEAD, a symmetrical cryptographic key and a piece of associated data are required. This associated data, called a *nonce* in the rest of the paper, can either be a uniquely generated handle or something that is uniquely related to the AEAD.

To encrypt a message using counter mode, one first divides it into blocks of equal length, each suitable for input to the block cipher AES, e.g., $data_1, \dots, data_n$. The counter mode $counter_1, \dots, counter_n$ is computed using an exclusive-or operator $\oplus$, where $counter_i = nonce \oplus i$ modulo 2^η and η is the length of a block in bits. The encrypted message is $(senc(counter_1, k) \oplus data_1); \dots; (senc(counter_n, k) \oplus data_n)$, where $senc$ is the encryption function and k the symmetrical cryptographic key; $senc(counter_1, k); \dots; senc(counter_n, k)$ is called the *keystream*. The

MAC is computed over the encrypted message and appended to obtain $senc(counter_1, k) \oplus data_1; \ldots; senc(counter_n, k) \oplus data_n); MAC$. The MAC is of fixed length, so it is possible to predict where it starts in an AEAD. However, since the two attacks considered below do not involve most of the details about block cipher AES, we follow the generalization of [19] and consider just messages of the form $senc(cmode(nonce), k) \oplus data; mac(data, k)$ where the counter mode is reduced to $cmode(nonce)$ for a chosen nonce, and the MAC is reduced to $mac(data, k)$.

In [19, 20], Künnemann and Steel reported two kinds of attacks on version 0.9.8 beta of YubiHSM API: (a) if the intruder has access to the server running YubiKey, where AES keys are generated, then it is able to obtain plaintext in the clear; (b) even if the intruder has no access to the server running YubiKey, it can use previous nonces to obtain AES keys. However, they were only able to find the first attack in Tamarin due to the limited support for exclusive-or in Tamarin at that time (see Section 6.1).

The first attack involves the YubiHSM API command depicted in Figure 4, which takes a handle to an AES key and the nonce and applies the raw block cipher. In order to perform

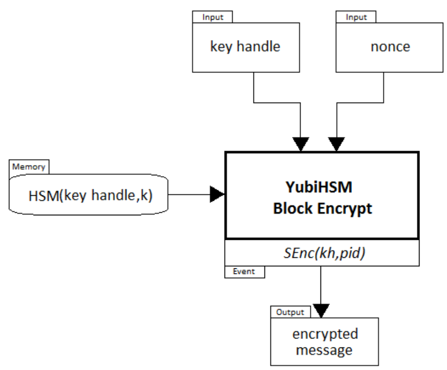


Figure 4: YubiHSM Block Encrypt API Command

this attack the intruder compromises the server to learn an $AEAD$ and the key-handle used to produce it. Then, using the Block Encrypt command shown in Figure 4, an intruder is able to decrypt an $AEAD$ by recreating the blocks of the key-stream: inputting $counter_i$ (the nonce) to the YubiHSM Block Encrypt API command. The intruder exclusive-ors the result with the $AEAD$ truncated by the length of the MAC and obtains the plaintext. Note that the verification of this attack in [20] using Tamarin involved additional user-defined lemmas (see Section 6.1).

The second attack involves the YubiHSM command depicted in Figure 5 that takes a nonce, a handle to an AES key and some data and outputs an $AEAD$. An intruder can produce an $AEAD$ for the same handle kh and a value $nonce$ that was previously used to generated another $AEAD$. An intruder can recover the keystream directly by using the AEAD Generate command to encrypt a string of zeros and then discarding the MAC. The result will be the exclusive-or of the keystream with a string of zeros, which is equal to the keystream itself. This attack is worse than the first one, because this command cannot be avoided or restricted (see [19]).

4 Maude-NPA

We begin by giving a brief overview of Maude-NPA. Then, in Sections 4.1, 4.2, and 4.3, we show how we used special features of Maude-NPA and Maude to model mutable memory, event

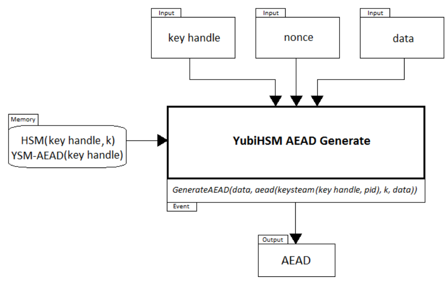


Figure 5: YubiHSM AEAD Generate API Command

lists, and Lamport clocks, respectively.

In Maude-NPA [1], as in most formal analysis tools for cryptographic protocols, a protocol is modeled as a set of rules that describe the actions of honest principals communication across a network controlled by an intruder. Given a protocol $\mathcal{P}$, states in Maude-NPA are modeled as elements of an initial algebra $\mathcal{T}_{\Sigma_\mathcal{P}/E_\mathcal{P}}$, where $\Sigma_\mathcal{P} = \Sigma_{SS} \cup \Sigma_\mathcal{C}$ is the signature defining the sorts and function symbols ($\Sigma_\mathcal{C}$ for the cryptographic functions and Σ_{SS} for all the state constructor symbols), $E_\mathcal{P} = E_\mathcal{C} \cup E_{SS}$ is a set of equations where $E_\mathcal{C}$ specifies the *algebraic properties* of the cryptographic functions and E_{SS} denotes properties of state constructors. The set of equations $E_\mathcal{C}$ may vary depending on different protocols, but the set of equations E_{SS} is always the same for all protocols. Therefore, a state is an $E_\mathcal{P}$-equivalence class $[t]_{E_\mathcal{P}} \in T_{\Sigma_\mathcal{P}/E_\mathcal{P}}$ with t a ground $\Sigma_\mathcal{P}$-term, i.e. a term without variables.

In Maude-NPA a *state pattern* for a protocol P is a term t of sort State which has the form $\{S_1 \,\&\, \cdots \,\&\, S_n \,\&\, \{IK\}\}$, where & is an infix associative-commutative union operator with identity symbol $\emptyset$. Each element in the set is either a *strand* S_i or the *intruder knowledge* $\{IK\}$ at that state.

The *intruder knowledge* $\{IK\}$ belongs to the state and is represented as a set of facts using comma as an infix associative-commutative union operator with identity element *empty*. There are two kinds of intruder facts: *positive* knowledge facts (the intruder knows m, i.e., $m \in \mathcal{I}$), and *negative* knowledge facts (the intruder *does not yet know* m but *will know it in a future state*, i.e., $m \notin \mathcal{I}$), where m is a message expression.

A *strand* [15] specifies the sequence of messages sent and received by a principal executing the protocol and is represented as a sequence $[msg_1^\pm, msg_2^\pm, msg_3^\pm, \ldots, msg_{k-1}^\pm, msg_k^\pm]$ with $msg_i^\pm$ either msg_i^- (also written $-msg_i$) representing an input message, or msg_i^+ (also written $+msg_i$) representing an output message. Note that each msg_i is a term of a special sort Msg.

In Maude-NPA, variables of sort Fresh will never be instantiated [1] during the analysis and, thus, are considered as constants. This ensures that if nonces are represented using variables of sort Fresh, they will never be made equal to each other and thus each nonce remains unique. Strands are extended with all the fresh variables $f_1, \ldots, f_k$ created by that strand, i.e., $:: f_1, \ldots, f_k :: [msg_1^\pm, msg_2^\pm, \ldots, msg_k^\pm]$.

Strands are used to represent both the actions of honest principals (with a strand specified for each protocol role) and the actions of an intruder (with a strand for each action an intruder is able to perform on messages). In Maude-NPA strands evolve over time; the symbol | is used to divide past and future. That is, given a strand $[\ msg_1^\pm,\ \ldots,\ msg_i^\pm \mid msg_{i+1}^\pm,\ \ldots,\ msg_k^\pm\ ]$, messages $msg_1^\pm, \ldots, msg_i^\pm$ are the *past messages*, and messages $msg_{i+1}^\pm, \ldots, msg_k^\pm$ are the *future messages* ($msg_{i+1}^\pm$ is the immediate future message). A strand $[msg_1^\pm, \ldots, msg_k^\pm]$ is shorthand for $[nil \mid msg_1^\pm, \ldots, msg_k^\pm, nil]$. An *initial state* is a state where the bar is at the beginning for all strands in the state, and the intruder knowledge has no fact of the form $m \in \mathcal{I}$. A *final state* is a state where the bar is at the end for all strands in the state and there is no intruder fact of the form $m \notin \mathcal{I}$.

Since the number of states in $T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}$ is in general infinite, rather than exploring concrete protocol states $[t]_{E_{\mathcal{P}}} \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}$, Maude-NPA explores *symbolic state patterns* $[t(x_1,\ldots,x_n)]_{E_{\mathcal{P}}} \in T_{\Sigma_{\mathcal{P}}/E_{\mathcal{P}}}(\mathcal{X})$ on the free $(\Sigma_{\mathcal{P}}, E_{\mathcal{P}})$-algebra over a set of variables $\mathcal{X}$. In this way, a state pattern $[t(x_1,\ldots,x_n)]_{E_{\mathcal{P}}}$ represents not a single concrete state (i.e., an $E_{\mathcal{P}}$-equivalence class) but a possibly infinite set of states (i.e., an infinite set of $E_{\mathcal{P}}$-equivalence classes), namely all the *instances* of the pattern $[t(x_1,\ldots,x_n)]_{E_{\mathcal{P}}}$ where the variables $x_1,\ldots,x_n$ have been instantiated by concrete ground terms.

The semantics of Maude-NPA is expressed in terms of *rewrite rules* that describe how a protocol transitions from one state to another via the intruder's interaction with it. One uses Maude-NPA to find an attack by specifying an insecure state pattern called an *attack pattern*. Maude-NPA attempts to find a path from an initial state to the attack pattern via backwards narrowing (narrowing using the rewrite rules with the orientation reversed). That is, a narrowing sequence from an initial state to an attack state is searched *in reverse* as a *backwards path* from the attack state to the initial state. Maude-NPA attempts to find paths until it can no longer form any backwards narrowing step, at which point it terminates. If at that point it has not found an initial state, the attack pattern is shown to be *unreachable* modulo the equations $E_{\mathcal{P}}$. Note that Maude-NPA places *no bounds on the number of sessions*, so reachability is undecidable in general. Note also that Maude-NPA does not perform any data abstraction such as a bounded number of nonces. However, the tool makes use of various sound and complete state space reduction techniques that help to identify unreachable and redundant states [13], and thus make termination more likely.

4.1 Modeling Mutable Memory by means of Maude-NPA Strand Composition

Strands can be extended with *synchronization messages* [26] of the form $\{Role_1 \to Role_2 \;;;\; mode \;;;\; w\}$ where $Role_1, Role_2$ are constants of sort Role provided by the user, *mode* can be either `1-1` or `1-*` representing a one-to-one or one-to-many synchronization (whether an output message can synchronize with one or many input messages), and w is a term representing the information passed along in a synchronization message. Synchronization messages are limited to the beginning (resp. end) of a strand and are called input (resp. output) messages. Although originally intended for a different use, they are very useful for representing a strand of unspecified length as a concatenation of different fixed-length strands. For example, consider a module that receives i pieces of data, and then exclusive-ors them, i.e., $[-(M_1),\ldots,-(M_i),+(M_1 \oplus \cdots \oplus M_i)]$ for $i \geq 1$. This can be specified in Maude-NPA using three strands with synchronization messages:

1. $[\quad -(M_1), \{role_{\oplus} \to role_{\oplus} \;;;\; \texttt{1-1} \;;;\; M_1\} \quad]$
2. $[\{role_{\oplus} \to role_{\oplus} \;;;\; \texttt{1-1} \;;;\; M\}, -(M_2), \{role_{\oplus} \to role_{\oplus} \;;;\; \texttt{1-1} \;;;\; (M \oplus M_2)\}]$
3. $[\{role_{\oplus} \to role_{\oplus} \;;;\; \texttt{1-1} \;;;\; M\}, +(M) \quad]$

Composition is then performed by unifying output synchronization messages with input synchronization messages of instances of these three strands.

For the YubiKey and YubiHSM APIs, if each event is represented by a strand, then an execution (e.g., Plugin followed by Press followed by Login) can be represented by the concatenation of the strands associated to the execution. However, the YubiKey and YubiHSM APIs also require different information to be stored from one API command to the next. Some information is read-only, but other information is updated, such as the `YubiCounter(pid,counter)`. Maude-NPA, unlike Tamarin, does not natively support mutable memory; but it can be modeled using synchronization messages. That is, the old data will appear in the input synchronization message of an API strand, and the new information will appear in the output synchronization message of that strand, which will then become the input synchronization message of the next API strand.

We model the mutable memory used by YubiKey as a multiset of predicates, where we define a new multiset union symbol `@`, which is an infix associative-commutative symbol with an identity symbol `empty`. Thus, for the strand describing the YubiKey button press, the input synchronization message is as follows:

```
{yubikey -> yubikey ;; 1-1 ;; Y(pid,sid) @ YubiCounter(pid,c1) @
   Server(pid,sid,c2) @ SharedKey(pid,k)}
```

Updating the counter of the YubiKey after a button press is represented by updating the second argument of the `YubiCounter(pid,c1)` predicate in the multiset. This updated multiset becomes the output synchronization of the strand.

4.2 Modeling Event Lists by Means of Maude Built-in Lists

The YubiKey and YubiHSM APIs also keep a rigid control of the ordering of events, where an event is a state transition in the system, and a proper analysis of actions is mandatory. Maude-NPA, unlike Tamarin, does not natively support the representation and analysis of event sequences; but we have implemented it by storing event sequences in the synchronization messages. This is helped by the fact that Maude-NPA, via the Maude language, has recently been endowed with built-in lists (using any associative symbol provided by the user). We have defined a new infix associative symbol `++` with an identity symbol `nil` to represent an event list and also a new auxiliary infix symbol `|>` where the left-hand side contains the mutable memory and the right-hand side contains the event list. The input synchronization message for the button press strand has now the form:

```
{yubikey -> yubikey ;; 1-1 ;; Y(pid,sid) @ YubiCounter(pid,c1) @
  Server(pid,sid,c2) @ SharedKey(pid,k) |> Plugin(pid,c3) ++ Press(pid,c4)}
```

Every time a new event occurs, it is inserted as a new element at the end of the event list. The leftmost elements are the oldest ones, whereas the rightmost elements are the newest. Thus, if we want to say that event e_1 must occur before event e_2, we can express this with the event pattern L_1 `++` e_1 `++` L_2 `++` e_2 `++` L_3, where any of the L_i variables could be empty.

Unification modulo associativity is infinitary [6], e.g., the unification problem $a : X \stackrel{?}{=} X : a$ where : is an associative symbol, X is a variable, and a is a constant has an infinite number of most general solutions $\{X \mapsto a^n\}$ for a^n being a list of n consecutive a constants. However, the implementation of unification modulo associativity in Maude is guaranteed to terminate with a finite and complete set of most general unifiers for a fairly large class of unification problems occurring in practice [12]. For any problem outside this class, the algorithm returns a finite set of unifiers together with a *warning* that such set may be incomplete. The reader should be aware that no warning showed up during the experiments of Section 6 and, thus, all the analyses were complete, which is especially important for the security properties of YubiKey.

4.3 Modeling Lamport Clocks in Maude-NPA Using Constraints

Lamport clocks require the testing of constraints: i.e., whether one counter is smaller than another. This is simple to do when the counters have concrete values. However, since Maude-NPA does not consider concrete protocol states but symbolic state patterns (terms with logical variables), the equality and disequality constraints handled by Maude-NPA are predicates defined over variables, whose domain, in the case of Lamport clocks, is the natural numbers.

In Maude-NPA strands can be extended with equality and disequality constraints [14] of the form "*Term1* `eq` *Term2*" and "*Term1* `neq` *Term2*". Whenever an equality constraint is found during the execution of a strand, the two terms in the equality constraint are unified modulo the set $E_{\mathcal{P}}$ of equations of the protocol and a new state is created for each possible unifier. Whenever a disequality constraint is found during the execution of a strand, it is simply stored in an internal repository of disequality constraints associated to each protocol state; but every time a new state is going to be generated during the state space exploration, all the disequality constraints in the internal repository are tested for satisfiability (see [14] for details).

We deal with Lamport clocks symbolically by representing the relations between clocks as constraints in *Presburger Arithmetic*. Although various *Satisfiability modulo theories (SMT)* [24] solvers such as CVC4[1], Yices[2], and Microsoft Z3[3] could be used for this purpose, we decided to avoid the complexities of invoking an external tool while executing Maude-NPA. Instead, we have used the variant-based decision procedure for Presburger Arithmetic already available in Maude [23]; but considered only positive numbers without zero.

Adding two natural numbers i and j is written as i + j. Checking whether a natural number i is smaller than another natural number j is represented in Maude-NPA by a constraint of the form j `eq` i + k, where k is an auxiliary variable. Disequality constraints are not needed.

5 Formal Specifications in Maude-NPA

5.1 Formal Specifications of YubiKey in Maude-NPA

In our specification, each command of the YubiKey API (Figures 1, 2, and 3) plus the initialization are specified in Maude-NPA as a strand.

The initialization strand is defined as follows. Three new Fresh values are defined: a YubiKey public ID (`rpid`), a secret ID (`rsid`), and a key '`rk`' shared with the server. Variables of sort Fresh are wrapped by symbol `Fr` as in [19].

```
:: rk,rpid,rsid ::
[ +(init),
  {yubikey -> yubikey ;; 1-1 ;;
   YubiCounter(Fr(rpid), 1) @ Server(Fr(rpid),Fr(rsid),1) @
   Y(Fr(rpid),Fr(rsid)) @ SharedKey(Fr(rpid),Fr(rk))
   |> Init(Fr(rpid),Fr(rk)) ++ ExtendedInit(Fr(rpid),Fr(rsid),Fr(rk))}]
```

The API command represented in Figure 1 shows what happens when a YubiKey is being plugged in. This command checks that the new received counter is smaller than the previous one, by using an equality constraint, and updates the predicate `YubiCounter`.

```
 :: nil ::
[{yubikey -> yubikey ;; 1-1 ;; YubiCounter(pid,otc) @ mem |> EL },
```

[1] Available at http://cvc4.cs.stanford.edu/web/.
[2] Available at http://yices.csl.sri.com.
[3] Available at https://github.com/Z3Prover/z3.

```
   -(tc), (tc eq (otc + extra)),
 {yubikey -> yubikey ;; 1-1 ;; YubiCounter(pid,tc) @ mem |> EL ++ Plugin(pid,tc)} ]
```

Note that the parameter `mem` denotes the rest of the mutable memory and the parameter `EL` denotes the previous event list. The variable `extra` is an auxiliary variable used just for testing the numerical constraint.

The command shown in Figure 2 represents what happens when the YubiKey button is pressed and the OTP is sent. The OTP is represented by message `senc(sid ; tc ; Fr(rnpr),k)` where `senc` denotes symmetric encryption using key `k` and symbol `;` denotes message concatenation[4].

```
:: rnpr,rnonce ::
[{yubikey -> yubikey ;; 1-1 ;;
   YubiCounter(pid,tc) @ Y(pid,sid) @ SharedKey(pid,k) @ mem |> EL },
   -(tc),
   +(pid ; Fr(rnonce) ; senc(sid ; tc ; Fr(rnpr),k)),
 {yubikey -> yubikey ;; 1-1 ;;
   YubiCounter(pid,tc + 1) @ Y(pid,sid) @ SharedKey(pid,k) @ mem |> EL ++ YubiPress(pid,tc)}]
```

Finally, the command shown in Figure 3 represents what happens when the server receives a login request. This request is accepted if the counter inside the encryption is larger than the last counter stored on the server, by using an equality constraint.

```
:: nil ::
[ {yubikey -> yubikey ;; 1-1 ;; Server(pid,sid,otc) @ SharedKey(pid,k) @ mem |> EL },
   -(pid ; nonce ; senc(sid ; tc ; pr, k)), -(otc),  (tc eq (otc + extra)),
  {yubikey -> yubikey ;; 1-1 ;; Server(pid,sid,tc) @ SharedKey(pid,k) @ mem
      |> EL ++ Login(pid,sid,tc,senc(sid ; tc ; pr, k)) ++ LoginCounter(pid,otc,tc) } ]
```

5.2 Formal Specification of YubiHSM in Maude-NPA

We consider only the two commands associated to the attacks, which were shown in Figures 4 and 5 above. Each command is specified in Maude-NPA as a strand. YubiHSM makes extensive use of exclusive-or, denoted by the symbol $*$, which satisfies the following equations:

$$
\begin{aligned}
x * (y * z) &= (x * y) * z && \text{(associativity)}\\
x * y &= y * x && \text{(commutativity)}\\
x * \mathit{null} &= x && \text{(identity element)}\\
x * x &= \mathit{null} && \text{(self-cancellation)}
\end{aligned}
$$

The YubiHSM command of Figure 4 is defined as follows.

```
:: nil ::
[ {YubiHSM -> YubiHSM ;; 1-1 ;; HSM(kh,k) @ mem |> EL },
   -(kh), -(nonce),
   +(senc(cmode(nonce),k)),
  {YubiHSM -> YubiHSM ;; 1-1 ;; HSM(kh,k) @ mem |> EL ++ SEnc(kh,nonce) } ]
```

We use two alternative definitions of the YubiHSM command of Figure 5, one to represent what happens when the command processes plaintext from the intruder, and another to represent what happens when the command processes plaintext from a legitimate principal. This

[4]Note that `;` is not an associative symbol and it is used as "message cons" symbol using Maude label "`gather (e E)`" that concatenates a single element to the left of a list.

is possible because, in contrast to the traditional Dolev-Yao model, honest principals communicate with the YubiHSM devices directly, not through the intruder. This means that we can represent an honest principal's input data as internal to the system. Moreover, in this instance such a representation is necessary, since we are asking whether the intruder can learn the input data. We maximize the intruder's advantage, however, by giving it control over the other input data.

The following strand represents the intruder learning an honest principal's input plaintext data. We assume that the plaintext data is a `Fresh` value. In this way, we can later ask whether the intruder is able to learn that `Fresh` value. We use the following macro: `aead(n,k,d) = (senc(cmode(n),k) * d) ; mac(d,k)`.

```
:: data ::
[ {YubiHSM -> YubiHSM ;; 1-1 ;; HSM(kh,k) @ mem |> EL },
   -(kh), -(nonce),
   +(aead(nonce,k,Fr(data))),
   {YubiHSM -> YubiHSM ;; 1-1 ;; HSM(kh,k) @ mem
      |> EL ++ GenerateAEAD(Fr(data),aead(nonce,k,Fr(data)))}]
```

In the second strand we represent the `Fresh` value (`data`) associated to the plaintext data by an input from the intruder.

```
:: nil ::
[ {YubiHSM -> YubiHSM ;; 1-1 ;; HSM(kh,k) @ mem |> EL },
   -(data), -(kh), -(nonce),
   +(aead(nonce,k,data)),
   {YubiHSM -> YubiHSM ;; 1-1 ;; HSM(kh,k) @ mem
      |> EL ++ GenerateAEAD(data,aead(nonce,k,data)) }]
```

6 Experiments

We have been able to automatically prove secrecy and authentication properties (a,b,c below) of YubiKey and to find both attacks (d,e below) on YubiHSM:

(a) Absence of replay attacks in YubiKey, i.e., there are no two distinct logins that accept the same counter.

(b) Correspondence between pressing the button on a YubiKey and a successful login. In other words, a successful login must have been preceded by a button pressed for the same counter.

(c) Counter values of YubiKey are different over time, where a successful login invalidates previous OTPs.

(d) If the intruder has access to the server running YubiKey, it can use previous YubiHSM nonces to obtain AES keys.

(e) If the intruder has no access to the server running YubiKey, it can use previous YubiHSM nonces to decrypt a previously generated AEAD.

Table 1 summarizes the result of the analyses of the YubiKey and YubiHSM APIs specified in Maude-NPA showing the number of generated nodes in each step. In addition to the attacks (a)-(e), we analyzed a standard login sequence of the YubiKey API. The notation "(1)" represents that the tool found 1 solution to the question asked by the attack pattern. When the number of generated nodes is 0, the attack pattern is *unreachable*.

Attack Pattern	Depth								
	1	2	3	4	5	6	7	8	9
YubiKey (a)	4	4	9	21	88	160	0		
YubiKey (b)	4	7	16	14	2	2	5	0	
YubiKey (c)	4	4	6	18	55	80	0		
YubiKey Login	1	1	2	1	1	1	1	1	1(1)
YubiHSM (d)	1	2	3	4	7	13	24	40	76(1)
YubiHSM (e)	4	6	11	26(1)					

Table 1: Output YubiKey and YubiHSM Experiments

All the details on how the attack patterns are specified and which was the returned output are available at http://personales.upv.es/sanesro/Maude-NPA-YubiKey-YubiHSM. The analyses were completely automatic and we obtained finite search graphs for all the attack patterns. This was achieved thanks to Maude's associative unification (i.e., event list expressions are included within the attack patterns) and the variant-based SMT solving for Lamport clocks (i.e., specific counter constraints are included). Note that Maude-NPA uses a full specification of exclusive-or, an unbounded session model, and an active Dolev-Yao intruder model. Moreover, it does not perform any data abstraction such as a bounded number of nonces, so there are no false positives or negatives.

6.1 Experiments using Tamarin

In [19, 20], the authors needed some user-defined lemmas to prove properties (b) and (c) of YubiKey and property (d) of YubiHSM, and they could not find the attack of property (e) due to the limited support for exclusive-or in Tamarin at that time. However, Tamarin has recently been endowed with exclusive-or in [11]. In this section, we report on some experiments that we have performed with it. In summary, nothing has changed for properties (b) and (c), and property (e) can *now* be carried out by Tamarin using a lemma.

The latest version of Tamarin with exclusive-or (version 1.4.0) is *now* available at https://github.com/tamarin-prover/tamarin-prover. Both YubiKey and YubiHSM specifications are also available at path "`examples/related_work/YubiSecure_KS_STM12`". Property (b) of YubiKey is specified as follows:

```
lemma one_count_foreach_login[reuse,use_induction]:
       "∀ pid sid x otp  #t2 . Login(pid,sid,x,otp)@#t2  →
         ( ∃  #t1 . YubiPress(pid,x)@#t1 ∧  #t1<#t2 )"
```

whereas property (c) of YubiKey is specified as follows:

```
lemma Login_invalidates_smaller_counters:
       "∀ pid otc1 tc1 otc2 tc2 #t1 #t2 #t3 .
            LoginCounter(pid,otc1,tc1)@#t1 ∧ LoginCounter(pid,otc2,tc2)@#t2
     ∧  Smaller(tc1,tc2)@#t3 →  #t1<#t2 "
```

Both properties (b) and (c) use a constraint `Smaller(tc1,tc2)` where `tc1` and `tc2` are token counters; for property (b) the constraint is not written explicitly but it is also necessary. In order for Tamarin to prove these two properties the following user-defined lemmas are necessary (called axioms in Tamarin).

```
axiom smaller:
        "∀ #i a b. Smaller(a,b)@#i →  Ex z. a+z=b"
axiom transitivity:
        "∀ #t1 #t2 a b c. IsSmaller(a,b)@#t1 ∧  IsSmaller(b,c)@#t2
        →  ∃  #t3 . IsSmaller(a,c)@#t3 "
axiom smaller_implies_unequal:
        " ¬ (∃  a #t . IsSmaller(a,a)@#t)"
```

Since these properties do not require exclusive-or, nothing changes from the earlier version of Tamarin to the latest one and, when proving properties (b) and (c) without these axioms either Tamarin is not able to terminate or terminates but without finding a proof.

For properties (d) and (e) of YubiHSM, we have rewritten the original specification to use exclusive-or following the examples published in [11]. The following axioms were necessary to find the attack of property (d) in [20], and they are still necessary when using the new specification of YubiHSM and the latest version of Tamarin.

```
axiom theory_before_protocol:
    "∀ #i #j. Theory() @ i & Protocol() @ j ==> i < j"
axiom onetime:
    "∀ #t3 #t4 . OneTime()@#t3 & OneTime()@t4 ==> #t3=#t4"
```

We encoded property (e) as follows:

```
lemma auth_intruder_obtain_AES[use_induction]: exists-trace
   "∃ data ks k mac #t1 #t2 .
   GenerateAEAD(data,<senc(ks,k),mac>)@#t1 ∧  K(senc(ks,k))@#t2  ∧  #t1<#t2"
```

We checked that the latest version of Tamarin was able to find the corresponding attack of the new specification of property (e), though our automated analysis in Maude-NPA was done [16] before [11] appeared.

7 Related Work

There is a vast amount of research on the formal analysis of APIs, so in this related work section we will concentrate on the work that is closest to ours, namely, the formal analysis of the YubiKey and YubiKey-like systems. Further related work on APIs and exclusive-or can be found in [17,18].

Besides the work on formalizing and verifying YubiKey that we have already discussed, there has been further work focused on building tools for analyzing policies for YubiKey and YubiKey-like systems.

In [3], Yubico presents some security arguments on their website. An independent analysis was given by blogger Fredrik Björck in 2009 [7,8], raising issues that Yubico responded to in a subsequent post. Oswald, Richter, et al. [25] analyze the YubiKey, generation 2, for side-channel attacks. They show that non-invasive measurements of the power consumption of the device allow retrieving the AES-key within approximately one hour of access. The authors mentioned a more recent version of the YubiKey, the YubiKey Neo which employs a certified smart-card controller that was designed with regard to implementation attacks and is supposed to be more resilient to power consumption analysis.

Künnemann et al. [19] performed a deep analysis of the different properties of YubiKey, but unlike our analysis using the Maude-NPA tool, they needed to use different lemmas, e.g., axioms `smaller`, `transitivity` or `smaller_implies_unequal` shown in Section 6.1, to check some properties that cannot be done automatically by the Tamarin prover, whereas these properties can be checked out in an automatic way by the Maude-NPA tool. Some properties were not proved due to limited support for exclusive-or. Mutable global state memory can be used in protocols that provide end-to-end encryption for instant messaging [9] as well as at the Trusted Platform Module (TPM) [10] that is a hardware chip designed to enable commodity computers to achieve greater levels of security than is possible by software alone.

8 Conclusions

The main contributions of this paper are to both prove properties of YubiKey generation 2 and find the known attacks on version 0.9.8 of YubiHSM in a completely automated way beyond the capabilities of previous work in the literature. This allowed us to perform the analysis of these APIs in a fully-unbounded session model making no abstraction or approximation of fresh values, and with no extra assumptions. These APIs involve several challenges: (1) handling of Lamport clocks, (2) modeling of mutable memory, (3) handling of constraints on the ordering of events, and (4) support for symbolic reasoning modulo exclusive or.

The main goal of this work has been to investigate whether Maude-NPA could complement and extend the formal modeling and analysis results about YubiKey and YubiHSM obtained in [19]. This is a non-obvious question: on the one hand, Maude-NPA has provided support for exclusive-or for years, so it is well-suited for meeting Challenge (4). But, on the other hand, previous applications of Maude-NPA have not addressed Challenges (1)-(3). The main upshot of the results we present can be summarized as follows: Challenge (2) can by met by expressing mutable memory in terms of *synchronization messages*, a notion used in Maude-NPA to specify protocol compositions [26], Challenge (3) can by met by the recently added unification modulo associativity, allowing an easy treatment of lists, and Challenge (1) can be met by a slight extension of Maude-NPA's current support for equality and disequality constraints [14], namely, by adding also support for constraints in Presburger Arithmetic. In this way, we show how challenges (1)-(4) can all be met by Maude-NPA, and how these results in automated formal analyses of YubiKey and YubiHSM substantially extend previous analyses. Very few tools are well equipped to simultaneously handle all these challenges.

What remains to be seen is how generally applicable these tools are to YubiKey and similar APIs. We note that previous work on analyzing API protocols in Maude-NPA did not achieve termination of the search space: the IBM CCA API in [17] and the PKCS#11 in [18]. In this work we have been able to achieve termination of many properties thanks to the use of Lamport clocks, mutable memory, and event lists. But more secure API case studies are needed to further test and advance the techniques presented here.

References

[1] Maude-NPA Manual v3.1.1. Available on: http://maude.cs.illinois.edu/w/images/d/d5/Maude-NPA_manual_v3.pdf.

[2] The Tamarin-Prover Manual (September 25, 2018). Available on: https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf.

[3] Yubico AB. YubiKey Security Evaluation: Discussion of security properties and best practices. v2.0. Available on: http://static.yubico.com/var/uploads/pdfs/Security_Evaluation_2009-09-09.pdf.

[4] Yubico customer list. Available on: http://www.yubico.com/references.

[5] Specifications Overview, FIDO Alliance. Available on: https://fidoalliance.org/specifications/overview/, December 2015.

[6] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[7] Fredrik Björck. Security DJ Blog: Increased security for Yubikey. Available on: http://web.archive.org/web/20100203110742/http://security.dj/?p=4, August 2009.

[8] Fredrik Björck. Security DJ Blog: Yubikey Security Weaknesses. Available on: http://web.archive.org/web/20100725005817/http://security.dj/?p=154, February 2009.

[9] Katriel Cohn-Gordon, Cas J. F. Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 451–466. IEEE, 2017.

[10] Stéphanie Delaune, Steve Kremer, Mark Dermot Ryan, and Graham Steel. Formal analysis of protocols based on TPM state registers. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 66–80. IEEE Computer Society, 2011.

[11] Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, and Ralf Sasse. Automated unbounded verification of stateful cryptographic protocols with exclusive OR. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 359–373. IEEE Computer Society, 2018.

[12] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. Associative unification and symbolic reasoning modulo associativity in Maude. In Vlad Rusu, editor, *Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018, Held as a Satellite Event of ETAPS, Thessaloniki, Greece, June 14-15, 2018, Proceedings*, volume 11152 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2018.

[13] Santiago Escobar, Catherine A. Meadows, José Meseguer, and Sonia Santiago. State space reduction in the Maude-NRL Protocol Analyzer. *Inf. Comput.*, 238:157–186, 2014.

[14] Santiago Escobar, Catherine A. Meadows, José Meseguer, and Sonia Santiago. Symbolic protocol analysis with disequality constraints modulo equational theories. In Chiara Bodei, Gian Luigi Ferrari, and Corrado Priami, editors, *Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, volume 9465 of *Lecture Notes in Computer Science*, pages 238–261. Springer, 2015.

[15] F. J. T. Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: why is a security protocol correct? In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No.98CB36186)*, pages 160–171, May 1998.

[16] Antonio González-Burgueño, Damián Aparicio, Santiago Escobar, Catherine A. Meadows, and José Meseguer. Formal verification of the YubiKey and YubiHSM APIs in Maude-NPA. *CoRR*, abs/1806.07209, 2018.

[17] Antonio González-Burgueño, Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. Analysis of the IBM CCA Security API Protocols in Maude-NPA. In Liqun Chen and Chris J. Mitchell, editors, *Security Standardisation Research - First International Conference, SSR 2014, London, UK, December 16-17, 2014. Proceedings*, volume 8893 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 2014.

[18] Antonio González-Burgueño, Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. Analysis of the PKCS#11 API Using the Maude-NPA Tool. In Liqun Chen and Shin'ichiro Matsuo, editors, *Security Standardisation Research - Second International Conference, SSR 2015, Tokyo, Japan, December 15-16, 2015, Proceedings*, volume 9497 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2015.

[19] Robert Künnemann. *Foundations for analyzing security APIs in the symbolic and computational model. Available on: https://tel.archives-ouvertes.fr/tel-00942459/file/Kunnemann2014.pdf*. Theses, École normale supérieure de Cachan - ENS Cachan, January 2014.

[20] Robert Künnemann and Graham Steel. YubiSecure? formal security analysis results for the Yubikey and YubiHSM. In Audun Jøsang, Pierangela Samarati, and Marinella Petrocchi, editors,

Revised Selected Papers of the 8th Workshop on Security and Trust Management (STM'12), volume 7783 of *Lecture Notes in Computer Science*, pages 257–272, Pisa, Italy, September 2012. Springer.

[21] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[22] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[23] José Meseguer. Variant-based satisfiability in initial algebras. *Sci. Comput. Program.*, 154:3–41, 2018.

[24] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, November 2006.

[25] David Oswald, Bastian Richter, and Christof Paar. Side-channel attacks on the yubikey 2 one-time password generator. In Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright, editors, *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, volume 8145 of *Lecture Notes in Computer Science*, pages 204–222. Springer, 2013.

[26] Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. Effective sequential protocol composition in Maude-NPA. *CoRR*, abs/1603.00087, 2016.

[27] L. Vamanu. Formal analysis of Yubikey. Master's thesis, École normale supérieure de Cachan (August 2011). Available on: `http://n.ethz.ch/~lvamanu/download/YubiKeyAnalysis.pdf`.

[28] Yubico. The YubiKey Manual. Usage, configuration and introduction of basic concepts. Version: 3.4. Available on: `https://goo.gl/zk5fbK`.

[29] Yubico. YubiHSM Manual v1.5. Available on: `https://www.yubico.com/wp-content/uploads/2015/04/YubiHSM-Manual_1_5_0.pdf`.