A Formally Verified Configuration for Hardware Security Modules in the Cloud

Riccardo Focardi
focardi@unive.it
DAIS, Ca' Foscari University
Venice, Italy

Flaminia L. Luccio
luccio@unive.it
DAIS, Ca' Foscari University
Venice, Italy

ABSTRACT

Hardware Security Modules (HSMs) are trusted machines that perform sensitive operations in critical ecosystems. They are usually required by law in financial and government digital services. The most important feature of an HSM is its ability to store sensitive credentials and cryptographic keys inside a tamper-resistant hardware, so that every operation is done internally through a suitable API, and such sensitive data are never exposed outside the device. HSMs are now conveniently provided in the cloud, meaning that the physical machines are remotely hosted by some provider and customers can access them through a standard API. The property of keeping sensitive data inside the device is even more important in this setting as a vulnerable application might expose the full API to an attacker. Unfortunately, in the last 20+ years a multitude of practical API-level attacks have been found and proved feasible in real devices. The latest version of `PKCS#11`, the most popular standard API for HSMs, does not address these issues leaving all the flaws possible. In this paper, we propose the first secure HSM configuration that does not require any restriction or modification of the `PKCS#11` API and is suitable to cloud HSM solutions, where compliance to the standard API is of paramount importance. The configuration relies on a careful separation of roles among the different HSM users so that known API flaws are not exploitable by any attacker taking control of the application. We prove the correctness of the configuration by providing a formal model in the state-of-the-art Tamarin prover and we show how to implement the configuration in a real cloud HSM solution.

CCS CONCEPTS

• **Security and privacy** → **Key management**; **Cryptanalysis and other attacks**; **Formal security models**; • **Theory of computation** → **Cryptographic primitives**.

KEYWORDS

Hardware Security Modules, `PKCS#11`, Cryptographic APIs, Automated analysis.

ACM Reference Format:
Riccardo Focardi and Flaminia L. Luccio. 2021. A Formally Verified Configuration for Hardware Security Modules in the Cloud. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3460120.3484785

1 INTRODUCTION

The increasingly massive digitization we have been witnessing in recent years is leading to a pervasive use of cryptography for daily tasks. This phenomenon is even more pronounced in critical ecosystems such as financial and governmental ones. In these contexts it is often mandatory to use special machines called Hardware Security Modules (HSMs) that allow applications to perform critical operations internally, without exposing the cryptographic keys. In fact, the main feature of an HSM is to keep sensitive data inside a tamper-resistant hardware so that in the event of a physical attack it is not possible to extract the cryptographic keys and produce an identical copy of the HSM for malicious purposes. In other words, resistance to intrusion makes an HSM unique, similarly to a smartcard: an attacker who wants to carry out illicit cryptographic operations should necessarily have access to the device since its cloning, which requires a copy of the secret data kept in the device, should not be possible.

This special hardware is very expensive and is not tailored to a specific application. Its standard API, named `PKCS#11`, is in fact a general purpose API for cryptographic operations with some peculiar features. Keys have attributes that determine their role and usage so, for example, the value of a `sensitive` key cannot be directly accessed from outside the device. However, keys can be exported encrypted under so called `wrap` keys, in order to be securely stored off the device or imported and shared with other HSMs. Unfortunately, in the last 20+ years a number of API level attacks on `PKCS#11` have appeared in the literature. In his MSc thesis, Clulow [11] showed that an astonishingly trivial attack was possible: given a target sensitive key, it was possible to generate another key with both `wrap` and `decrypt` attributes set. This allowed for a trivial *wrap-then-decrypt* attack: the sensitive key was first wrapped with the `wrap` key, producing a ciphertext outside the device which was then sent back to the device for decryption under the very same key. The coexistence of these conflicting roles for the same key made it possible to obtain the plaintext value of a sensitive key off the device after two simple API calls: first a *wrap* and then a *decrypt*.

After Clulow's seminal work, many researchers have pointed out more attack variants and investigated possible fixes to the API. A practical implementation of a secure wrapping mechanism have been even suggested to Oasis, the current maintainer of the PKCS#11, with no success [37]: none of the proposed fixes have received enough attention, and the latest version of the PKCS#11 standard [35] does not contain any guideline on how to prevent this attacks in practice and, in fact, does not even mention them. On the contrary, referring in a broad sense to API-level attacks, the latest version of PKCS#11 usage guide [33, Section 3.1], which dates back to November 2014, surprisingly states that: "We note that none of the attacks just described can compromise keys marked sensitive, since a key that is sensitive will always remain sensitive". While this is certainly true when it comes to modifying attributes, the attack described above allows `sensitive` keys to be leaked in the clear, breaking the required security property. Hence, we are faced with a disturbing situation where expensive hardware enforced by law in security-critical environments is subject to attacks that are not even mentioned in the standard API documentation.

The only available defense was introduced in version 2.40 of the PKCS#11 standard [34]: a special attribute `wrap_with_trusted` that, when set on `sensitive` keys, only allows wrapping under keys marked as `trusted` by a privileged HSM user called Security Officer. However, this attribute is not mandatory for sensitive keys and it is unclear how these special trusted keys should be generated and managed. As previously discussed, the documentation does not provide guidance on this important attribute, and does not mention the fact that not using it exposes the HSM to well-known vulnerabilities. In fact, not using `wrap_with_trusted` completely voids the adoption of an HSM as any attacker gaining access to the API could leak any key with `wrap_with_trusted` attribute unset.

Particular installations might use ad-hoc solutions to prevent the above mentioned vulnerabilities, such as disabling key wrapping and any other dangerous key management functionalities in production devices, or replace them with nonstandard proprietary APIs as, e.g., [7, 9, 17, 29, 36]. However, these solutions break portability and, in general, it would be preferable to use the full API functionality in a secure way rather then cutting it down to a secure subset, or modifying it in nonstandard ways. Moreover, proprietary solutions are not publicly documented and cannot be validated by the scientific community. In many cases, this *security-by-obscurity* approach only provides a false sense of security and many proprietary solutions exhibit vulnerabilities once they have been reverse-engineered or just leaked to the public. The numerous attacks on cryptographic protocols for automotive systems are a representative example (e.g., [24, 39]).

In recent years, we have witnessed the rapidly growing phenomenon of cloud HSMs. Several cloud service providers give the possibility to use real HSM clusters managed entirely in the cloud and accessible via remote APIs [2, 26, 27, 38]. This phenomenon radically changes the attacker model. With classic, physically reachable HSMs, it was possible to use specific procedures for managing keys, which could only be performed by users with physical access to the machines; with cloud HSMs these procedures must necessarily take place remotely via the API offered by the service provider. Furthermore, with classic HSMs it was possible to customize their configuration based on the particular application, for example by disabling some critical functionalities, and possibly using proprietary APIs; with cloud HSMs this possibility no longer exists as they only provide standard APIs in order to work with any PKCS#11-based application. In fact, the advent of cloud HSMs makes the weakness of their APIs even more critical and requires an accurate and correct use of `trusted` keys. In addition, it becomes extremely important to apply the least privilege principle to applications to prevent a vulnerability from allowing the attacker to access the (vulnerable) key management functions, giving the ability to extract sensitive keys as cleartext. Unfortunately, as already discussed, the API documentation does not contain any guidelines in this regard.

In this paper, we propose the first configuration for HSMs based on a careful separation of roles so that the critical key management operations are only performed by special users in a way that production applications cannot exploit the (flawed) API to extract keys. As we discussed earlier, this is very important, especially in a cloud setting, where the HSM is operated remotely and a vulnerability in one application might compromise the security of the whole device, leaking all of the sensitive keys. Interestingly, our solution does not require any limitation of the API for the applications: any HSM user has access to the full API, however only the Security Officer can mark a key as `trusted` as defined in the standard. We define the following categories of users:

Normal Users (NU) These users are accessible by production applications, so it is of paramount importance that: (*i*) they can use the full API, in order to maximize compatibility; (*ii*) they cannot exploit API-level attacks to leak sensitive keys, as they are the most exposed HSM users: in fact, an attacker might exploit a vulnerability in production application to gain access to the HSM API. Intuitively, the keys of these users that need to be wrapped under other keys will always be marked as `wrap_with_trusted`.

Key Manager (KM) These users perform key management operations using the standard API in a controlled way. They are responsible for handling the `trusted` keys that will be used to wrap other sensitive keys. The life-cycle of these keys is very delicate. For example, they should never be wrapped under other keys and never be allowed to encrypt or decrypt data. The KM accounts should only be accessed by special management applications (or directly by humans), and their credentials should never be used in production applications.

Security Officer (SO) The Security Officer is a special user which adheres to the PKCS#11 standard and has no access to the full API [33, Section 2.4]. As such, the SO mainly performs administrative tasks, such as creating other HSM users and marking keys as `trusted`. Specifically, we require the SO to mark as `trusted` only the keys generated by a Key Manager. As for KMs, the SO account should only be accessed by special management applications (or directly by humans), and its credentials should never be used by production applications.

Intuitively, if we assume that the management of `trusted` keys is done by special users (KMs and SO) and is not accessible by production applications, `wrap_with_trusted` keys will never be leaked in the clear, even using all known API-level attacks. The tricky part of the configuration is to define precisely what the KMs and SO should or should not do. For example, enabling decryption on a trusted wrapping key would allow the trivial Clulow's wrap-then-decrypt attack previously discussed. Encryption is also dangerous as an attacker could encrypt a known key and then import it to the device using unwrap: these malicious keys should never be marked as trusted. Finally, if a trusted key can be wrapped under another trusted key then an attacker might *reimport* it in the device with different attributes enabling the previous attacks [18]. Our separation of roles is based on the crucial assumption that keys generated by one user can be made available to other HSM users but key management operations, such as attribute change, will only be permitted to the key owner. We will show that this form of mild access control is implemented in real cloud HSM solutions.

Given the well-documented subtleties behind this unfortunate API, it is of paramount importance to provide evidence of the correctness of the proposed configuration. We decided to model it using the state-of-the-art Tamarin prover, a powerful tool that allows for proving security properties of cryptographic protocols. In our model, the attacker has full access to the API but cannot impersonate neither the Security Officer nor the Key Manager. In this way, we cover any vulnerable application with access to the HSM APIs which cannot tamper with the management of trusted keys. The model represents a significant subset of PKCS#11 including symmetric key cryptography, key management, core key attributes and users, and we prove our configuration correct under various assumptions on the behavior of the KMs and SO. Our policy is simple to implement and we show how to apply it to the AWS CloudHSM solution [2] which supports the required form of access control between users and keys. The simplicity of the policy comes from the simplicity of the wrap_with_trusted mechanism that, unfortunately, does not offer enough flexibility to implement more sophisticated secure configurations.

Contributions. The main contributions of this work are summarized below:

(1) We explore for the first time the usage of users' roles for securing PKCS#11. In particular, we introduce special users called Key Managers that are in charge of creating and managing the trusted keys. The keys generated by the KMs are made available to other HSM users but key management will only be permitted to the key owner;
(2) Based on the users' roles, we propose the first practical HSM configuration that does not require any restriction or modification of the PKCS#11 API and is suitable to cloud HSM solutions. The configuration comprises a set of rules for the secure usage of trusted and wrap_with_trusted attributes and we claim its correctness through an informal analysis based on existing API-level attacks;
(3) We show how the rules can be directly applied to the AWS CloudHSM which provides a suitable access control over users and keys. Interestingly, AWS CloudHSM allows for separating key usage from key management, which is one of the crucial assumption of our configuration;
(4) Finally, we provide a formal model of a significant core of PKCS#11 and we use it to model the proposed secure configuration rules as constrains on the behavior of KMs and SO. We use the Tamarin tool to automatically verify that sensitive keys are never leaked to normal users, even when they are in full control of the API, assuming that KMs and SO behave correctly. The proof is fully automated and the model can be easily adapted to check variants of the proposed configuration.

Related work. The first security vulnerability that may properly be called an API-level attack was discovered by Longley and Rigby in the early 1990s [30]. Although the device was not identified at the time, it was later discovered that it was an HSM manufactured by Eracom and used in the ATM network.

In 2001, Anderson published an attack on key loading procedures on another similar module manufactured by Visa [4], and the term *security API* was coined by Bond and Anderson [5, 6] in two subsequent papers presenting other attacks. Clayton and Bond showed how computationally intensive attacks could be implemented against a real IBM device using programmable FPGA hardware [10]. Independently from the Cambridge group, an MSc thesis by Clulow gave more examples of attacks, mostly specific to the PIN translation and verification commands offered by the API of Prism HSMs [11]. Clulow also published the first attacks on the industry standard for cryptographic key management APIs, PKCS#11 [12].

This proliferation of vulnerabilities has stimulated research results regarding the analysis of security APIs that has helped understanding in depth the nature of key management attacks. A first effort to apply general analysis tools appeared in [40], but the researchers were unable to discover any new attacks and could not conclude anything about the security of the device. The first automated analysis of PKCS#11 with a formal statement of the underlying assumptions was presented in [18]. When no attacks were found, the authors were able to derive precise security properties of the device. In [7], the model was generalized and provided with a reverse-engineering tool that automatically refined the model depending on the actual behaviour of the device. When new attacks were found, they were tested directly on the device to get rid of possible spurious attacks determined by the model abstraction. The automated tool of [7] successfully found attacks that leaked the value of sensitive keys on real devices. In [17], *authenticated wrapping* was analyzed and proved correct under some assumptions. The idea was that the attributes were wrapped together with the key value so that they were preserved when the keys were exported and reimported. This mechanism would provide a fundamental improvement in PKCS#11 security but, unfortunately, it has not yet been included in the standard. Other previous attempts to request its inclusion have so far failed (see, e.g., [37]).

The works discussed so far do not take into account the attribute wrap_with_trusted which was introduced in version 2.20 of PKCS#11. The first analysis of the wrap_with_trusted attribute appeared in [23], where the authors showed that when trusted keys have a fixed, pre-determined set of attributes, all keys with wrap_with_trusted set are secure. Intuitively, the work provided formal evidence that the basic mechanism was secure, when used in a very controlled and limited way. In [8] and [29] the analysis was generalized to other key configurations and was proved correct via typing and automated verification, respectively. In [36] a PKCS#11 configuration proposed in [7] was proved secure on a computational model of PKCS#11, using wrap_with_trusted to prevent key cycles which, by the way, is not the primary purpose of this attribute. All of these works assume that the attributes of keys are immutable. While this property is certainly desirable for a cryptographic API, PKCS#11 allows for attribute changes and removing this functionality might break a multitude of applications. Therefore, the applicability of these proposals is confined to specific applications and is rather limited for what concerns cloud HSMs which aim to offer a complete PKCS#11 interface.

Our work is also based on the `wrap_with_trusted` attribute, but our proposal requires minimal constraints on key management by leveraging user roles. To the best of our knowledge, our configuration is the first one that does not require any modification or reduction of PKCS#11: it simply requires some particular users to be trustworthy and behave according to strict guidelines, making it directly applicable to real cases. In particular, we do not impose any restriction on production users, who can use the full API including the change attribute functionality, and we only limit what Key Managers and the Security Officer can do.

The Tamarin prover [31] has been successfully used to prove the correctness of many real world protocols (see, e.g., [14–16, 20, 25]). Following this line of work, we model a core subset of PKCS#11 in Tamarin obtaining a fully mechanized proof of our secure configuration for an unbounded number of sessions, users and keys. This allows for easily validating possible model extensions and variants. The automated analyses of [29] was also performed using Tamarin as a backend, but the model was specified in a process calculus which was translated into a Tamarin model so, to the best of our knowledge, our model of the `wrap_with_trusted` mechanism is the first one that is directly expressed in the Tamarin prover language. A peculiar feature that makes the analysis of security APIs hard is that they have a non-monotonic mutable global state [18]. The assumption on immutable attributes of [29] simplifies this aspect. Our model, instead, supports non-monotonic attributes that we made treatable through new, suitable simplifications (cf. Section 4.2). Moreover, we model for the first time access control to keys. In [17] a model of authenticated wrapping was expressed and verified in Tamarin. The proposed authenticated wrapping mechanism is based on unique counters, a key hierarchy and authenticated handles, which are not in the PKCS#11 standard. In fact, as we discussed earlier, the aim of the paper was to propose new, provable secure extensions of the PKCS#11 standard and not, as we do here, to analyze the security of PKCS#11 configurations.

Paper organization. The paper is organized as follows: in Section 2 we recall some notions related to the PKCS#11 API and to the Tamarin prover. In Section 3 we present our new HSM configuration and we apply it to the AWS CloudHSM solution. In Section 4 we formalize our configuration in Tamarin and we automatically prove its security. In Section 5 we provide some concluding remarks.

2 BACKGROUND

In this section, we first introduce the PKCS#11 standard and API-level attacks (Section 2.1), then we briefly illustrate the Tamarin prover (Section 2.2) which will later be used to prove the security properties of the proposed HSM configuration.

Handle a_1, associated to key k_1, has attributes `sensitive` and `extractable` set and handle a_2, associated to key k_2, has attributes `wrap` and `decrypt` set; c denotes the encryption of k_1 under key k_2 using a symmetric key cipher. The attack leaks the sensitive key k_1 referenced by a_1 in the clear.

$$\begin{array}{rcl} \textbf{Wrap}(a_1,a_2) & \rightarrow & c \\ \textbf{Decrypt}(c,a_2) & \rightarrow & k_1 \end{array}$$

Figure 1: PKCS#11 wrap-then-decrypt attack.

2.1 The PKCS#11 Standard

The RSA Public Key Cryptography Standard #11 (PKCS#11), first proposed in 1995 as Version 1.0, describes an API named 'Cryptoki' for cryptographic hardware such as HSMs, cryptographic tokens and smart cards. It supports both cryptographic and key management functions with the general idea that all operations are performed inside the device so that cryptographic keys are never exposed in the clear to external applications. The latest version of the standard, taken over by OASIS in 2012, is Version 3.0 [35].

The PKCS#11 API is used by applications as follows. The application starts a *session* with the device identifying either as a Security Officer (SO), or as a normal user. In this phase, there is no defense against a rogue host or application, as the user PIN may be intercepted. However, this should not allow an attacker to compromise a sensitive key "since a key that is sensitive will always remain sensitive", as stated in the latest version of PKCS#11 usage guide [33, Section 3.1], which dates back to November 2014. After a session is established, the application may access the *objects*, such as keys and certificates, stored in the device. Objects are accessed through *handles* that point to them and do not disclose any information about the object they point to. New objects, pointed by fresh handles, are either created with a key generation command or by unwrapping an encrypted blob, as described below.

Objects have *attributes* that specify properties or roles. Many of them are boolean flags that can be set or unset. For example, attribute `sensitive`, when set, protects a key from being read in the clear, and cannot be unset, to prevent trivial attacks. A key can be exported from the device encrypted under another key only if its attribute `extractable` is set, and once unset it cannot be set again. The attributes `encrypt` and `decrypt` indicate keys that can be respectively used to encrypt and decrypt data. The attribute `wrap`, instead, is used to encrypt another key that has the `extractable` attribute set and then to export it as a ciphertext. Similarly, `unwrap` is used to import keys: it decrypts an encrypted key and imports it in the device as a new object, returning a corresponding fresh handle. Thus, wrap and unwrap functions allow for exporting and reimporting (`extractable`) keys encrypted under so called *wrapping* keys. All the cryptographic functions use key handles to refer to keys so that their value is never exposed outside the device. For example, wrap takes as input two handles that refer respectively to the key to be wrapped and the wrapping key, while encrypt takes as input data bytes and one handle that refers to the encryption key.

API-level attacks on PKCS#11. The first attacks to the PKCS#11 API were introduced by Clulow in 2003 [12]. The simplest one is called *wrap-then-decrypt*: the attacker first wraps a sensitive key, and then decrypts it. More precisely, the attacker has two handles: a_1, associated to a key k_1, with attributes sensitive and extractable set and a_2, associated to a key k_2, with attributes wrap and decrypt set. The attacker is able to leak the sensitive key k_1 in the clear by first wrapping it under k_2 obtaining ciphertext c, and then decrypting it using key k_2. Note that, wrap is possible because of the wrap attribute set on a_2 and the extractable attribute set on a_1, while decrypt is possible because of the decrypt attribute set on a_2. When the device executes the operations, it cannot distinguish between keys and plaintexts and the decrypted key is given as output to the attacker. The attack is illustrated in Figure 1.

There exists a *dual* version of this attack called *encrypt-then-unwrap* in which handle a_2 has attributes unwrap and encrypt set. The attacker encrypts a known key k_A under a_2 and then unwraps it, importing it in the device with fresh handle a_3 and attribute wrap set. Note that, when a key is unwrapped it is possible to specify its attributes. The attacker can now wrap a_1, which refers to the sensitive key k_1, under a_3 obtaining the encryption of $k_1$under the known key k_A. Decryption can be then performed by the attacker independently of the device.

In principle, these attacks could be prevented by forbidding conflicting roles on keys such as wrap, decrypt and unwrap, encrypt. However attributes can be set and unset liberally so, for example, wrap could be unset before decrypt is set and, even if these conflicts could be prevented on a single handle in one device, subtler attacks exist that exploit the presence of a key under multiple handles, possibly on different devices. For example, a wrapped key could be imported twice providing two copies with conflicting roles or, equivalently, an existing key could be wrapped and then unwrapped to provide a new instance of the key with a conflicting role. Note that, these latter attacks could be carried out on separate devices sharing a wrapping key, making it very difficult to track conflicting attributes on the same key: the sensitive key could be wrapped under one copy of the key on the first device and decrypted under another copy of the key on the second device. The interested reader can refer to [18] for a comprehensive analysis of all attack variants and to [22] for a more detailed introduction to API-level attacks.

Trusted wrapping keys. The only existing mechanism in the standard that can prevent API-level attacks is the wrap_with_trusted attribute, introduced in Version 2.20, that forces a key to be wrapped only under keys that have attribute trusted set. Once set, the wrap_with_trusted attribute cannot be unset and the only user that can mark a key as trusted is the Security Officer. Note that, while this attribute can potentially prevent previous attacks, in the standard it is not mandatory to set wrap_with_trusted on sensitive keys and there is no mention of how trusted keys should be generated and managed, vanishing all advantages of this mechanism. Some works proved that an accurate use of this attribute in a very controlled an limited way may work correctly, but they put limits to its application and assume that the attributes of keys are immutable, which is not realistic as it reduces the API functionality and breaks compliance with the PKCS#11 standard [1, 8, 23, 29, 36].

2.2 The Tamarin Prover

The Tamarin prover is a very powerful verification tool [31] that has been successfully used to prove the correctness of many real world protocols (see, e.g., [14–16, 20, 25]). Attackers and protocols are specified using multiset rewriting rules, and properties as first-order logic formulas that admit quantification over timepoints and messages. Proofs can either be done interactively, by combining automated proof search with manual guidance, or in a fully automated way using heuristic searches. In this second scenario, if Tamarin terminates, it either provides a proof of correctness of all system traces, or an attack as a counter-example trace of the system.

Terms, facts and special facts. In Tamarin, *terms* are used to symbolically represent data and messages and functions, e.g., $\mathsf{senc}(\_,\_)$, $\mathsf{sdec}(\_,\_)$ are used to express symmetric key encryption and decryption. Tamarin supports equational theories, e.g., $\mathsf{sdec}(\mathsf{senc}(m,k),k) =_E m$ that makes it possible to deconstruct $\mathsf{senc}(m,k)$ and recover m from the term $\mathsf{sdec}(\mathsf{senc}(m,k),k)$; *facts* $\mathsf{F}(t_1,\ldots,t_n)$ are used to store state information while *special facts* have a particular semantics. For example, $\mathtt{In}(m)$, $\mathtt{Out}(m)$ are used for receiving and sending message m and $\mathtt{Fr}(n)$ to generate a fresh name n, not known by the attacker.

Rules, execution and properties. The systems starts from an *initial state* , i.e., the empty multiset, and transitions are specified by a *set of rules*, which are in the form:

$$[\ p_1,\ldots,p_m\ ]\quad -\![\ a_1,\ldots,a_k\ ]\!\!\rightarrow\quad [\ c_1,\ldots,c_n\ ]$$

where $p_1,\ldots,p_m$ are the premise facts that must belong to the state to enable the rule. Variables inside facts are instantiated once they match the facts in the state. When the rule is fired different actions are taken: the premise facts are removed from the state, unless they are specified as persistent using the symbol !; action facts $a_1,\ldots,a_k$ become part of the execution trace and can be used to express properties of the protocol using first-order logic formulas with timepoints; finally, conclusion facts $c_1,\ldots,c_n$ are added to the state. Variables of action and conclusion facts are bound to the premise facts variables and are instantiated accordingly. An alternating sequence of states and rule instances defines a protocol execution. Note that, there are built-in rules to model a standard Dolev-Yao attacker [19].

The following first-order formula is a simple example of a property that can be expressed in Tamarin:

$$\exists\, \mathtt{U,h,\#i}\ .\ \mathtt{SetAttr(U,h,a)@i}$$

It states that action fact SetAttr(U,h,a) has occurred at some time i. The symbol # in front of i is used in the quantifiers to indicate that variable i represents a timepoint. We will use this action fact in Section 4.1 to model a user U setting attribute a on handle h.

3 A SECURE HSM CONFIGURATION

As we have pointed out in the previous sections, the PKCS#11 API is flawed and there is no easy way to fix it without reducing its functionality or resorting to non-standard extensions. In the extensive literature on the subject, it has been pointed out many times that the excessive liberality of the API, especially in terms of changing key attributes, is the main source of the problem. Nevertheless, PKCS#11 is still the standard API for cryptographic hardware and HSMs in particular, and it is of paramount importance to investigate general guidelines to use it in a proper way, limiting the attack surface as much as possible. Using a flawed API to access a very expensive secure hardware is quite disturbing, and we believe that the growing adoption of cloud HSM solutions will eventually provide a huge attack surface, completely nullifying the promised enhanced security. We also have to keep in mind that the use of HSM is required by law in many settings, and discovering that the security of these devices is poor, after so many years of warnings from the academic community, would have very negative consequences.

In this section, we make a new proposal for a secure HSM configuration which does not require any change in the PKCS#11 API. As a consequence, our proposal is fully compliant with the standard API and can be adopted immediately. The configuration relies on few requirements for user management in HSMs that we believe might be easily incorporated in any cloud HSM solution. Interestingly, we found that AWS CloudHSM already provides the necessary ingredients to implement our configuration.

The section is organized as follows: in Section 3.1 we present the attacker model and in Section 3.2 we give a definition of HSM security; in Section 3.3 we describe the user roles and, based on them, in Section 3.4 we define our secure configuration; in Section 3.5 we show how known attacks are prevented by the secure configuration and claim its security in a general case; finally, in Section 3.6 we show how to implement our configuration in a real cloud HSM solution.

3.1 Attacker Model

Our attacker model is based on the classic API-level security approach where the attacker is assumed to gain access to the device's API, targeting its sensitive keys. Notice that, once the API is available to the attacker, any cryptographic operation will be granted. For example, the attacker will be able to decrypt any ciphertext encrypted under the HSM symmetric keys and to produce any digital signature using HSM private keys. Therefore, API access naturally gives the attacker the power to *use* sensitive keys in arbitrary ways. However, the advantage of HSMs is that the attacker should not be able to extract sensitive keys and *clone* the HSM functionality so to have unbound access to those keys.

The difference might seem insignificant but in fact is very important and, by the way, is the only extra layer of protection provided by this secure cryptographic hardware. The only place where sensitive keys should be used is inside the HSM: without access to the device it should be impossible to use its keys. A typical use case is an HSM used by a Certification Authority (CA) to sign public key certificates. An attacker gaining access to the HSM might create rogue certificates but should not be able to extract the root private key from the HSM. A temporary access to the HSM due to some vulnerability could be fixed by revoking the generated rogue certificates while leaking the root private key would require to revoke it, invalidating any existing certificate signed by that CA.

Moreover, continuous access to the actual HSM would be detectable in the medium/long term while having a copy of the HSM keys would make the attack persistent and definitively harder to spot, as the attacker could locally use the keys without any access to the HSM. By analogy, an HSM could be thought of as a very powerful smartcard: physically accessing the smartcard gives temporary privileges to an attacker, e.g. accessing a building, but cloning it would provide unlimited access.

Refining the attacker model. Since the API is flawed, full API access allows for extracting sensitive keys and makes any attempt of defining a secure configuration void. In other words, the general API-level attacker model is too strong and requires changing the API in order to achieve some security (see, e.g., [1, 7, 9, 17, 29, 36]). Our strategy is to exploit a form of mild access control over keys that makes it possible to define different HSM users with specialized roles. In this way, we can effectively isolate critical key management operations that require some trust. In particular, we assume that keys are shared among users, and we let the owner of the key do whatever operation with it, while we forbid other users to change the attributes of the keys they do not own:

Definition 3.1 (Key owner). An HSM user creating a key is its *owner* and can perform any operation over that key. Other users can use keys that they do not own but they cannot modify their attributes.

We refine accordingly the attacker model by assuming that the attacker only impersonates a subset of the HSM users. This makes sense in practice as accessing the HSM from production applications will be more vulnerable to attacks than accessing the HSM from management applications used by administrators. We let $\mathcal{U}$ denote all the HSM users:

Definition 3.2 (Attacker model). Given a subset of the HSM users $\mathcal{A} \subseteq \mathcal{U}$, we say that an HSM is under $\mathcal{A}$-attack if the attacker can access the HSM API as one of the users in $\mathcal{A}$.

Intuitively, we will investigate HSM security assuming that a portion of the users, namely $\mathcal{U} \setminus \mathcal{A}$, has not been compromised. This is necessary, as full compromise would allow for full access to the API that we know to be flawed. As we explain next, we do not resort to any API modification and, instead, we base our separation of roles on the standard `wrap_with_trusted` attribute (cf. Section 3.4).

3.2 Security Property

The primary aim of an HSM is to never expose sensitive keys outside the device: cryptographic operations are performed inside the HSM and keys should only be exported, when necessary, wrapped under other secure keys. Since key usage is allowed by the API, we cannot guarantee any security property about it: attacker accessing the API as user $u \in \mathcal{A}$ will be able to perform any operation on behalf of u such as encrypting, decrypting and signing data. In fact, the latest version of PKCS#11 usage guide [33, Section 3.1] explicitly states that an authenticated user may perform any operation supported by the token. So, even when the keys are secured we do not have any guarantee about data security, when the HSM is under $\mathcal{A}$-attack (cf. Definition 3.2).

Therefore, our focus is on key confidentiality: any sensitive key of the HSM should remain confidential, even when the device is under attack. One might wonder which keys should be regarded as sensitive and which should not. Typically, long term keys stored in the device and reused across sessions should be regarded as sensitive. Session keys, generated on the fly and with a limited validity might be regarded as less important and treated with more flexibility: an API-level attack compromising session keys would have a limited impact and would not allow to clone the HSM functionality. Notice that, `sensitive` is also a PKCS#11 attribute that is used to explicitly tag sensitive keys in the device, but we use the word sensitive in its broader sense, to indicate important keys as defined below:

Definition 3.3 (Sensitive keys). Any key that, if leaked, allows to clone the HSM functionality should be regarded as *sensitive* and should have the corresponding PKCS#11 attribute set.

We can now state our security property for HSMs in terms of Definitions 3.2 and 3.3. As discussed in Section 3.1, we define security assuming that a portion of the users, namely $\mathcal{U} \setminus \mathcal{A}$, has not been compromised:

Definition 3.4 (HSM security). Given a subset of the HSM users $\mathcal{A} \subseteq \mathcal{U}$, we say that an HSM is $\mathcal{A}$-secure if no *sensitive* key can be leaked even when the HSM is under $\mathcal{A}$-attack.

3.3 User Roles

Our secure configuration is based on a careful separation of roles so that the critical key management operations are only performed by special users, in a way that production applications cannot exploit the (flawed) API to extract keys. This is very important, especially in a cloud setting, where the HSM is operated remotely and a vulnerability in one application might compromise the security of the whole device, leaking all of the sensitive keys. Interestingly, our solution does not require any limitation of the API, which makes it fully compatible with standard HSM implementations, assuming a mild form of access control over keys. We define three categories of users:

Normal Users (NU) These users are accessible by production applications, so it is of ultimate importance that: (*i*) they can use the full API, so to maximize compatibility; (*ii*) they cannot exploit API-level attacks to leak sensitive keys, as they are the most exposed HSM users: in fact, an attacker might exploit a vulnerability in production application to gain access to the HSM API.

Key Manager (KM) These users perform key management operations using the standard API in a controlled way. They are responsible for handling the `trusted` keys that will be used to wrap other sensitive keys. The KM accounts should only be accessed by special management applications (or directly by humans), and their credentials should never be used in production applications.

Security Officer (SO) The Security Officer is a special user which adheres to the PKCS#11 standard and has no access to the full API [33, Section 2.4]. As such, the SO mainly performs administrative tasks, such as creating other HSM users. Interestingly, the SO can mark as `trusted` the keys generated by other users. As for KMs, the SO account should only be accessed by special management applications (or directly by humans), and its credentials should never be used by production applications.

It is important to notice that the PKCS#11 standard only defines two types of users: the normal users and the Security Officer (SO) [33, Section 2.4]. Thus, KMs will be implemented as a normal users whose credential should be regarded as very critical: compromising a KM would allow for API-level attacks. In practice, the SO and the KMs could be accessed by the same physical person using different accounts and combining cryptographic and administrative operations in order to generate and manage `trusted` keys.

3.4 Secure Configuration

We now present a secure configuration based on `trusted` and `wrap_with_trusted` attributes, and on the user roles defined in Section 3.3.

Protecting sensitive keys. The only available defense against API-level attacks was introduced in version 2.40 of the PKCS#11 standard: a special attribute `wrap_with_trusted` that, when set on *sensitive* keys, only allows wrapping under a `trusted` key which, in turn, can only be set by a special user of the HSM, namely the Security Officer. Not using `wrap_with_trusted` completely voids the adoption of an HSM as any attacker getting access to the API can potentially leak any key which has not been marked as `wrap_with_trusted` with a trivial wrap-then-decrypt attack. For this reason, we require this attribute on any sensitive key of the HSM (cf. Definition 3.3). As an alternative, the key should be generated with the `extractable` attribute unset, meaning that the key cannot be wrapped under other keys. Recall from Section 2.1, that `wrap_with_trusted` cannot be unset and `extractable` cannot be set, so if a key is generated with the suggested values the configuration is permanent and cannot be subverted: either the key is wrapped under a `trusted` key or it cannot be wrapped at all.

Rule 1 (Sensitive keys). Any sensitive key stored in the HSM should either be generated with attribute `wrap_with_trusted` set, or with attribute `extractable` unset.

Restricting KMs and SO behavior. In order to guarantee that `trusted` keys can really be trusted we need to impose constraints on the process of tagging a key as `trusted`. Recall that only the SO can set the `trusted` attribute, while cryptographic operations, such as key generation, should be done by a normal user. We elect KM to carry out this delicate task. The idea is that only the KM can generate a key that will be promoted to `trusted` by the SO.

Rule 2 (Trusted keys). The SO sets attribute `trusted` only on special *candidate* keys generated by one of the KMs.

These candidate keys should be treated very carefully so to avoid possible API-level attacks in production applications. In particular, during their lifetime they should never have roles that conflict with key wrapping, such as `encrypt` and `decrypt`:

Rule 3 (Roles of candidate keys). The candidate keys managed by the KMs should only admit wrap and unwrap cryptographic operations during their whole lifetime.

Moreover, candidate keys should not be wrapped under other keys, to prevent attacks in which keys are exported and then reimported with different attributes. Unfortunately, this uncovers the limitation of the `wrap_with_trusted` mechanism which does not allow for managing a hierarchy of keys.

Rule 4 (Management of candidate keys). The candidate keys managed by the KMs should be generated with the `extractable` attribute unset.

Candidate keys should be generated in the device to prevent them to be used differently from what is required by previous rules. Recall that, imported keys might exist in other devices with conflicting attributes. Cloud HSM solutions offer mechanisms to transparently synchronize keys across cluster of devices, out of the `PKCS#11` standard (e.g., [3, page 84]). So, in practical implementations it might be fine to generate fresh wrapping keys that are transparently replicated in other HSMs belonging to the same cluster.

Rule 5 (Freshness of candidate keys). The candidate keys managed by the KMs should be generated as fresh in the device.

3.5 Security Analysis

We now informally discuss how the configuration of Section 3.4 prevents the known API-level attacks from the literature. We limit our analysis to the attacks that exploit key management APIs to extract sensitive keys in the clear presented in Section 2.1. Note that, Rule 1 requires to wrap sensitive keys only under `trusted` keys which, according to Rule 2, can only be KM's candidate keys. So we get the following:

Fact 1. If Rules 1 and 2 are respected, then sensitive keys can only be wrapped under keys that respect Rules 3, 4 and 5.

We now show how the various attacks are prevented by one of these rules on candidate keys.

Wrap-then-decrypt. This attack is based on wrapping a sensitive key k_1 with a `wrap` key k_2 and subsequently decrypting it. It requires that the `wrap` key k_2 can be also used to decrypt data. Because of Fact 1, sensitive keys can only be wrapped under keys that respect Rule 3 which, in turns, forbids decryption with k_2 and prevents the attack.

Encrypt-then-unwrap. This attack is based on encrypting a known attacker key k_A and then importing it in the device as a malicious wrapping key. Because of Fact 1, sensitive keys can only be wrapped under keys that respect Rule 5 which, in turns, forbids `wrap` keys that have not been generated in the device, such as k_A. This prevents the attack.

Importing with conflicting roles. A key k_A is imported twice, setting `wrap` attribute on one copy and `decrypt` on the other so to execute a wrap-then-decrypt attack. As for the previous case, Fact 1 and Rule 5 prevent wrapping under a key k_A that has not been generated in the device.

Reimporting with conflicting roles. A wrapping (`trusted`) key k_T is wrapped and then unwrapped, setting the `decrypt` attribute on the copy, so to execute a wrap-then-decrypt attack. In this case Rule 5 does not help as the key could have been generated in the device. However, Fact 1 and Rule 4 prevent that the `trusted` wrapping key k_T is wrapped in the first place, blocking the attack.

A fundamental ingredient of our configuration is that both SO and KM behave accordingly to Rules 2 and 3, 4, 5, respectively. Therefore, we assume that SO and KM users are immune to attacks (cf. Section 3.1). NUs only have to respect Rule 1 in order to have their sensitive keys protected by the `wrap_with_trusted` mechanism, but this rule does not restrict their behavior in any way: once keys are tagged as `wrap_with_trusted` these users can access the full API and those keys will be immune to API-level attacks. Recall that, we implicitly assume that only the owner of the key can modify its attribute (cf. Definition 3.1) so, for example, NUs cannot modify the attributes of KM's keys. This is fundamental to prevent trivial attacks on trusted keys.

We have shown that some of the prominent API-level attacks are blocked by the rules of our configuration and, in particular, we have seen that all of the rules are necessary in order to prevent the attacks. We claim that our configuration guarantees that the HSM is $\mathcal{A}$-secure (cf. Definition 3.4) when SO and KM users are not under attack, i.e., they do not belong to $\mathcal{A}$. In other words, if the attacker only accesses the HSM as a normal user, sensitive keys are never leaked even when the attacker access the full API. More precisely:

Claim 1 (Secure configuration). Let $\mathcal{U}_{SO}$ and $\mathcal{U}_{KM}$ respectively denote all SO and KM users. If all the configuration rules of Section 3.4 are respected then the HSM is $\mathcal{A}$-secure for each $\mathcal{A} \subseteq \mathcal{U}$ such that $\mathcal{A} \cap (\mathcal{U}_{SO} \cup \mathcal{U}_{KM}) = \emptyset$.

To substantiate our claim, in Section 4 we will prove that the configuration is indeed correct for an unbounded number of users, sessions and keys, on a significant core of `PKCS#11`.

3.6 Implementation on Real Cloud HSMs

We now discuss how to implement the secure configuration of Section 3.4 in real cloud HSM solutions. As far as we know, AWS CloudHSM [2] is the only one suggesting in its documentation the adoption of the `wrap_with_trusted` mechanism. However, the suggestion is very general and does not refer to API-level attacks. Instead, it emphasizes the possibility offered by trusted keys of defining the so called *unwrap template*, i.e., the set of attributes that the imported key will have to adhere to. However, the *unwrap template* only offers a false sense of security since, once a key is unwrapped, its attributes can be changed. For this reason, we have not based our solution on this particular mechanism.

Interestingly, the AWS CloudHSM solution also provides a form of access control that is expressive enough to implement our secure configuration.

Key sharing. The AWS CloudHSM solution allows to share keys among different HSM users through a proprietary command called `shareKey` [3, page 140]. The manual explicitly states that "Users who share the key can use the key in cryptographic operations, but they cannot ... change its attributes". The latter part is what makes the key sharing mechanism a viable implementation of our configuration: it is enough for KMs to share the candidate keys with NUs so that they can be used as wrapping keys but can never be modified. Our attacker model, in fact, assumes a worst-case scenario in which keys are implicitly shared with any other HSM users (cf. Section 3.1).

Users. The AWS CloudHSM solution defines various users, two of which, named CryptoOfficer (CO) and CryptoUser (CU), can be mapped directly to the SO and normal users of `PKCS#11`. Our KM can be implemented as a CU, given that we are assuming it is a normal user whose credential should not be available to production applications.

Implementation. We assume to have special KM users that create easy to identify candidate keys, e.g., using particular labels, that we refer to as $\mathcal{K}$. The secure configuration rules of Section 3.4 can be directly instantiated in AWS CloudHSM as follows:

- **Rule 1** Attribute `wrap_with_trusted` should be set for any sensitive key, as already suggested in [3, page 88];
- **Rule 2** The CO should set the `trusted` attribute on $\mathcal{K}$ keys only;
- **Rule 3** Relatively to cryptographic operations, KMs should only set `wrap` and `unwrap` attributes on $\mathcal{K}$ keys;
- **Rule 4** Keys in $\mathcal{K}$ should be generated with `extractable` attribute unset;
- **Rule 5** Keys in $\mathcal{K}$ should be generated as fresh in the HSM. Note that, fresh keys can be shared between HSMs using external mechanisms not included in PKCS#11 [3, page 84].

4 FORMAL ANALYSIS

In this section, we formalize a significant subset of PKCS#11 in the Tamarin prover [31], and we automatically prove Claim 1 for an unbounded number of users, keys and sessions. The portion of PKCS#11 that we analyze includes symmetric key encryption and key wrapping. We model `extractable`, `wrap_with_trusted` and `trusted` attributes, as they are crucial for the secure configuration, and we model user roles mapping them to standard PKCS#11 users, i.e., Security Officer and normal users [33, Section 2.4], so that we do not need to modify in any way the standard API.

We introduce two simplifications in our model that make the analysis more manageable and fully automated. First, instead of explicitly modeling the encryption and decryption functionalities we leak keys with attribute `encrypt` or `decrypt` set to the attacker. Notice that, this does not reduce the attacker capabilities: on the contrary, it gives more power to the attacker since unsetting `encrypt` and `decrypt` will not disable the use of those keys. In our experiments, we found that this simplification did not introduce false attacks but allowed us to automatically prove all the lemmas. The second simplification is related to the fact that we do not model wrap and unwrap operations with untrusted keys, as we know they are subject to existing API-level attacks. In order to implicitly cover those attacks we just leak any `extractable` key that does not have attribute `wrap_with_trusted` set. Intuitively, this is equivalent to attacking each of those keys with, e.g., a wrap-then-decrypt attack.

The section is organized as follows: we describe the model of user roles, keys and attributes in Section 4.1; we discuss model simplifications in Section 4.2; we present the model of key management operations in Section 4.3 and in Section 4.4 we model our HSM configuration and prove its security.

4.1 Modeling User Roles, Keys and Attributes

User roles. Standard PKCS#11 normal users and Security Officer are modeled with facts `!NU(U1)` and `!SO(U2)`, respectively. As explained in Section 2.2, the `!` symbol makes a fact persistent. We will use these facts in the precondition of rules that can be fired by such users. Figure 2 reports, directly in the Tamarin syntax, the three rules for generating the three categories of users NU, KM and SO (cf. Section 3.3). Recall that `Fr` is a special fact in Tamarin used to generate fresh names that are not known by the attacker (cf. Section 2.2). For example, rule `NewNU` generates the persistent fact `!NU(U)`, for a fresh username `U`, producing an action fact `NewNU(U)`. This action fact appears in the protocol execution trace and can be used in logical formulas to state theorems over the model execution. Freshness of name `U` ensures that each user has a unique role. Note that, rule `NewKM` also generates a normal user `!NU(U)` but the action is different: `NewKM(U)`. This allows KMs to use the full API as if they were NUs, but the different action will allow us to set special restrictions over KMs (cf. Section 4.4).

Cryptographic keys. Recall that PKCS#11 uses handles to refer to cryptographic keys that are stored in the device (cf. Section 2.1). According to Definition 3.1, we also need to store the owner of the key. Therefore, we model keys as a persistent fact `!Key(U,ha,k)` with three terms: the owner `U`, the handle `ha` and the key value `k`. In PKCS#11 it is possible to specify a *template* that assigns values to attributes when a key is created/imported. The most generic template is one that only sets attribute `extractable`. Recall, in fact, that this attribute cannot be set but only unset, so if a key is generated with `extractable` unset it will never be exportable. In principle, from this template it is possible to reach any possible combination

```
rule NewNU:
  [ Fr(U) ] --[ NewNU(U) ]-> [ !NU(U) ]

rule NewKM:
  [ Fr(U) ] --[ NewKM(U) ]-> [ !NU(U) ]

rule NewSO:
  [ Fr(U) ] --[ NewSO(U) ]-> [ !SO(U) ]
```

Figure 2: Rules for new users.

```
rule CreateKey:
  [ !NU(U), Fr(ha), Fr(k) ]
--[ CreateKey(U,ha,k),
    SetAttr(U,ha,'extractable') ]->
  [ !Key(U,ha,k) ]

rule CreateWWTKey:
  [ !NU(U), Fr(ha), Fr(k) ]
--[ CreateWWTKey(U,ha,k),
    SetAttr(U,ha,'wrap_with_trusted'),
    SetAttr(U,ha,'extractable') ]->
  [ !Key(U,ha,k) ]

rule CreateNEKey:
  [ !NU(U), Fr(ha), Fr(k) ]
--[ CreateNEKey(U,ha,k) ]->
  [ !Key(U,ha,k) ]

rule ImportKey:
  [ !NU(U), Fr(ha), In(k) ]
--[ ImportKey(U,ha,k),
    SetAttr(U, ha, 'extractable') ]->
  [ !Key(U,ha,k) ]
```

Figure 3: Rules for key creation and import.

```
// For A in {'wrap','unwrap','encrypt','decrypt'}
rule SetAttrA:
  [ !Key(U,ha,k), !NU(U) ] --[ SetAttr  (U,ha,A) ]-> [ ]
rule UnsetAttrA:
  [ !Key(U,ha,k), !NU(U) ] --[ UnsetAttr(U,ha,A) ]-> [ ]

rule UnsetAttrExtractable:
  [ !Key(U,ha,k), !NU(U) ]
--[ UnsetAttr(U,ha,'extractable') ]-> [ ]

rule SetAttrWrapWithTrusted:
  [ !Key(U,ha,k), !NU(U) ]
--[ SetAttr(U,ha,'wrap_with_trusted') ]-> [ ]

rule SetAttrTrusted:
  [ !Key(U,ha,k), !SO(W) ]
--[ SetAttr  (W,ha,'trusted') ]-> [ ]
rule UnsetAttrTrusted:
  [ !Key(U,ha,k), !SO(W) ]
--[ UnsetAttr(W,ha,'trusted') ]-> [ ]
```

Figure 4: Rules for changing attributes. The first two rules apply to all attributes `wrap`, `unwrap`, `encrypt` and `decrypt`.

of attributes since, as we will describe next, attributes can be set and unset (with some exceptions). However, keys generated with this generic template are subject to known API-level attacks, so we need to define two more templates that we will use to generate secure keys. In fact, these templates adhere to Rule 1: The former has both `wrap_with_trusted` and `extractable` set and the latter has attribute `extractable` unset.

Key creation/import rules are reported in Figure 3. The first three rules correspond to the three templates previously described and generate a new key `!Key(U,ha,k)` respectively producing actions `CreateKey(U,ha,k)`, `CreateWWTKey(U,ha,k)` and `CreateNEKey (U,ha,k)`. Setting an attribute is modeled by `SetAttr(U,ha,a)`, representing user `U` setting attribute `a` on handle `ha`. In particular, `a` is a label containing the attribute name. For example, the first rule creates a key with the generic template in which only the attribute `extractable` is set, written `SetAttr(U,ha,'extractable')`; similarly, the second rule creates a key with `wrap_with_trusted` set (`extractable` is also set in order to make the key exportable under wrapping); note that, the third rule does not set any attribute: not setting `extractable` makes the key not exportable even through wrap operations. All three rules have the same precondition facts `!NU(U)`, `Fr(ha)`, `Fr(k)`, requiring a normal user `U` and two fresh values `ha` and `k`. The fourth rule is for importing keys in the clear. Notice that, `k` in this case is read as input through the special fact `In(k)`. These keys are imported with the most generic, insecure template. In fact, this rule lets the attacker import any known key in the device and use it in subsequent attacks.

Attributes. Similarly to attribute setting, attribute unsetting is modeled with action `UnsetAttr(U,ha,a)`. For each attribute we add a rule for setting and one for unsetting except for `extractable` and `wrap_with_trusted` that, respectively, can only be unset and set. In Figure 4 we report the rules for attribute change. Apart from `trusted`, the precondition is always `Key(U,ha,k)`, `!NU(U)`, i.e., the key is owned by the normal user `U` who is setting/unsetting the attribute. For attribute `trusted`, instead, the user performing the operation is SO who is not the key owner. In fact, only the Security Officer can set this attribute but cannot create the keys, that will necessarily be owned by other users.

Notice that, attributes are bound to handles and not directly to keys. This is very important and reflects the actual PKCS#11 specification: the same key can appear in the device many times under different handles, for example after many unwrap operations, and each occurrence of the key can have different attributes. This is what enables attacks in which a key is reimported with conflicting roles, which are very hard to spot in practice (cf. Section 2.1).

Action `IsSet(h,a)` is a special action that we use to check that attribute `a` is set on key with handle `h`. In Tamarin this can be done by imposing a restriction over the action as follows:

```
restriction IsSet:
  "
  All ha a #i . IsSet(ha,a)@i ==>
  (
    Ex U #j . SetAttr(U,ha,a)@j & j<i &
    (All W #w . UnsetAttr(W,ha,a)@w & w<i ==> w<j)
  )
  "
```

In the restriction we universally quantify over all handles `ha`, attributes `a` and timepoints `i` (`#` is used to indicate that `i` is a time variable). Intuitively, an attribute `a` is set to handle `ha` at time `i`, written `IsSet(ha,a)@i` if the attribute has been set before, written `SetAttr(U,ha,a)@j & j<i`, and in case the attribute has been also unset before, written `UnsetAttr(W,ha,a)@w & w<i`, then the set operation happened after the unset operation, written `w<j`. Intuitively, if there are many occurrences of set and unset operations before time `i`, it must be that set is the last one. In fact, an attribute can be set and unset many times but what matters is the last assignment. Similarly, we define an action `IsUnset` that will be used to check when an attribute is not set. The only difference with respect to `IsSet` is that an attribute is unset also when it has never be set, i.e., we assume that attributes are unset by default.[1] This is formalized in the last part of the formula stating that no `SetAttr(U,ha,a)@j` with `j<i` exists:

```
restriction IsUnset:
  "
  All ha a #i . IsUnset(ha,a)@i ==>
  (
    (Ex U #j . UnsetAttr(U,ha,a)@j & j<i &
      (All W #w . SetAttr(W,ha,a)@w & w<i ==> w<j))
    |
    (not Ex U #j . SetAttr(U,ha,a)@j & j<i)
  )
  "
```

4.2 Model Simplification

As previously discussed, instead of explicitly modeling encryption and decryption APIs we leak keys with attributes `encrypt` or `decrypt` set to the attacker. Moreover, since we are not interested in rediscovering known attacks, we only model wrapping under trusted keys and incorporate everything else in the standard Dolev-Yao attacker model offered by Tamarin. In fact, we do not need to model untrusted wrapping in detail because we know that it is

[1]Another possibility for modeling attributes would be to use non-permanent facts, but this would require re-adding an attribute each time it is used with the effect of over-complicating the model and the automated analysis.

```
rule LeakEncKey:
  [ !Key(U,ha,k) ]
--[ IsSet(ha,'encrypt') ]->
  [ Out(h(k)) ]

rule LeakDecKey:
  [ !Key(U,ha,k) ]
--[ IsSet(ha,'decrypt') ]->
  [ Out(h(k)) ]

rule LeakExtractable:
  [ !Key(U,ha,k) ]
--[ IsSet(ha,'extractable'),
    IsUnset(ha,'wrap_with_trusted') ]->
  [ Out(h(k)) ]
```

Figure 5: Model simplification through explicit key leakage.

insecure by construction. More precisely, we make the following simplifications:

- Keys with attributes `encrypt` or `decrypt` set are directly available to the attacker; as we already discussed, the attacker accessing the API can freely use the HSM keys and perform any encryption and decryption of her choice. We found out that modeling encrypt and decrypt operations complicated a lot the automated reasoning of the tool as the same term could be obtained both from the implicit Dolev-Yao attacker and from the explicit API.
- Keys with `extractable` set and `wrap_with_trusted` unset are directly available to the attacker. These keys are subject to a wrap-then-decrypt attack. Making them directly available to the attacker makes it possible to only model trusted wrap and unwrap functionalities.

Note that, leaking some of the keys to the attacker makes that attacker model strictly more powerful: any proof in this model will also hold in a model where those keys are not leaked. However, in order to express and prove the secrecy lemmas (cf. Section 4.4) we need a way to distinguish between keys implicitly leaked by the model and keys leaked in actual attacks. In other words, we need to distinguish the actual key from its *value*, used to perform cryptographic operations. Of course, knowing a key should allow to know its value but not vice-versa. A natural way to implement this idea is to hash the key `k` under a one-way hash function, written `h(k)`, any time it is used as cryptographic key but not when it is used as data, e.g., when it is wrapped. For example in Section 4.3 (Figure 6) we will use the symmetric encryption term `senc(k1,h(k2))` to represent key `k1` wrapped under `k2`, in which only the actual cryptographic key `k2` is hashed. Notice that, leaking `h(k)` allows the attacker to perform any cryptographic operation based on key `k` but does not reveal `k`. This modeling of `PKCS#11` is new and it is an original contribution by itself.

In Figure 5 we show the rules that leak keys: rules `LeakEncKey` and `LeakDecKey` leak respectively encryption and decryption keys, i.e., keys with `encrypt` and `decrypt` set. Rule `LeakExtractable` leaks extractable keys that do not have `wrap_with_trusted` set.

```
rule Wrap:
  [ !NU(U),!Key(U1,ha1,k1),!Key(U2,ha2,k2) ]
--[ Wrap(U,ha1,ha2),
    IsSet(ha1,'wrap_with_trusted'),
    IsSet(ha1,'extractable'),
    IsSet(ha2,'trusted'),
    IsSet(ha2,'wrap') ]->
  [ Out(senc(k1,h(k2))) ]

rule Unwrap:
  [ !NU(U1),!Key(U2,ha2,k2),In(senc(k1,h(k2))),Fr(ha1) ]
--[ Unwrap(U1,ha1,ha2),
    IsSet(ha2,'trusted'),
    IsSet(ha2,'unwrap'),
    SetAttr(U1,ha1,'wrap_with_trusted'),
    SetAttr(U1,ha1,'extractable') ]->
  [ !Key(U1,ha1,k1) ]
```

Figure 6: Wrap and unwrap of `wrap_with_trusted` keys.

4.3 Modeling Key Management

As already mentioned, we are not interested in rediscovering known API-level attacks so we explicitly model wrap and unwrap operations with trusted keys, while we implicitly leak keys with attribute `wrap_with_trusted` unset (cf. Section 4.2). In Figure 6, we model wrap and unwrap operations in which the wrapped key has attributes `wrap_with_trusted` and `extractable` set and the wrapping key has `trusted` and `wrap`/`unwrap` set. Intuitively, in rule `Wrap` a normal user `U` wraps key `k1` belonging to user `U1` under key `k2` belonging to user `U2`. Handle `ha1` of `k1` has both `wrap_with_trusted` and `extractable` set while handle `ha2` of `k2` has both `trusted` and `wrap` set. The encryption of `k1` under `k2`, noted `senc(k1,h(k2))`, is sent as output using the special Tamarin fact `Out`. Similarly, in rule `Unwrap` a normal user `U1` unwraps key `k1` from the ciphertext `senc(k1,h(k2))` read as input through the special Tamarin fact `In`. The key is imported under a fresh handle `ha1`. The wrapping key is required to have both `trusted` and `unwrap` set and the new key is imported with both `wrap_with_trusted` and `extractable` set. The reason why we use `h(k2)` instead of `k2` is due to the model simplification explained in Section 4.2.

4.4 Proof of Security

We now restrict the behavior of SO and KMs to adhere to the secure configuration of Section 3.4.

Trusted keys. Rule 2 states that SO should only mark as `trusted` special candidate keys generated by KMs. We let these keys be all the ones generated with `extractable` unset (rule `CreateNEKey` of Figure 3). This also complies with Rule 5, since the key is generated as fresh, and with Rule 4, since the key is generated with `extractable` unset. We prove security for all of these keys but, in a real implementation, candidate keys can be a subset of them.

```
restriction SO:
  "
  All W ha #i . SetAttr(W,ha,'trusted')@i
  ==>
  Ex k U #j #w .
    CreateNEKey(U,ha,k)@j & j<i &
    NewKM(U)@w & w<i
  "
```

The restriction states that whenever a handle ha is marked as trusted by W at time i, written SetAttr(W,ha,'trusted')@i, then the key must have been created with extractable unset, written CreateNEKey(U,ha,k)@j & j<i, by a Key Manager U, written NewKM(U)@w & w<i.

Candidate keys. Rules 3, 4 and 5 state how KMs should handle candidate keys that are marked as trusted by SO. Rule 5 and 4 are already fulfilled by the fact that SO only marks as trusted freshly generated keys with extractable unset. As for Rule 3, we only need to require that KM only admits wrap and unwrap cryptographic operations on these keys, obtaining the following restriction on KM's behavior:

```
restriction KM:
  "
  All U ha k a #i #j #w.
    NewKM(U)@i &
    CreateNEKey(U,ha,k)@j &
    SetAttr(U,ha,a)@w
  ==>
    ( a='wrap' | a='unwrap' )
  "
```

The restriction states that whenever an attribute a is set by a Key Manager U on a non extractable key k with handle ha, the attribute can only be wrap or unwrap. Note that, there is no constraint on the timing of actions, so this restriction holds independently of the ordering of actions for the entire lifetime of the key, as required by Rule 3.

HSM security. We now instantiate and prove Claim 1 on the Tamarin model presented so far that we have intuitively shown to adhere to the secure configuration rules of Section 3. Claim 1 states that if all the rules are respected then the HSM is $\mathcal{A}$-secure when any user except SO and KMs are under the control of the attacker. This assumption is implemented in the model by the restrictions over SO and KM presented above. From Definition 3.4 we know that $\mathcal{A}$-secure means that no *sensitive* key is leaked. Trusted keys should be certainly regarded as sensitive, as they are used to wrap other sensitive keys. Moreover, by Rule 1 we require that sensitive keys should either be generated with attribute wrap_with_trusted set, or with attribute extractable unset, i.e., with CreateWWTKey or CreateNEKey of Figure 3. In the following, we prove that the confidentiality of all of these keys is preserved.

For expressing the secrecy of trusted key it is useful to define an action IsHandle(ha,k) stating that ha is a valid handle for key k. We add a specific rule producing this action for each key:

```
rule IsHandle:
  [ !Key(U,ha,k) ] --[ IsHandle(ha,k) ]-> [ ]
```

Now, we can state the secrecy of trusted keys as follows:

```
lemma SecrecyTrusted:
  "
  All W ha k #i #j #w.
    IsHandle(ha,k)@i &
    SetAttr(W,ha,'trusted')@j &
    KU(k)@w
  ==> F
  "
```

where KU(k) is a special Tamarin fact expressing that the attacker knows k and F denotes *false*. This lemma states that any key k with handle ha marked as trusted cannot be discovered by the attacker (implication of *false* requires that the hypothesis is *false*).

The secrecy of CreateWWTKey and CreateNEKey can be stated in a similar way:

```
lemma SecrecyNE:
  "
  All U ha k #i #j .
    CreateNEKey(U,ha,k)@i &
    KU(k)@j
  ==> F
  "

lemma SecrecyWWT:
  "
  All U ha k #i #j .
    CreateWWTKey(U,ha,k)@i &
    KU(k)@j
  ==> F
  "
```

These three lemmas constitute an instance of Claim 1 and can be automatically checked in Tamarin, as detailed in the next section.

4.5 Automated Verification in Tamarin

The complete Tamarin model is in Appendix A and is publicly available at [21]. In addition to what was presented in the previous sections, it includes a helper lemma used to make the proof converge and a number of sanity lemmas checking that: (*i*) the rules of Section 3.4 are correctly modeled; (*ii*) all the actions are executable, i.e., the model is actually executable; (*iii*) some intuitive facts hold.

Helper lemma. Lemma Unwrap is a so called *source* lemma. Tamarin cannot compute all the possible sources of terms used in its backward search algorithm for our model and this lemma helps Tamarin to disambiguate some cases. In particular the lemma states that unwrapped keys come from one of the CreateKey, CreateWWTKey or ImportKey rules:

```
lemma Unwrap [sources, reuse]:
  "
  All U k1 ha1 ha2 #i .
    Unwrap(U,ha1,ha2)@i &
    IsHandle(ha1,k1)@i
  ==>
  (
    (Ex W ha #j . CreateKey(W,ha,k1)@j & j<i)
    |
    (Ex W ha #j . CreateWWTKey(W,ha,k1)@j & j<i)
    |
    (Ex W ha #j . ImportKey(W,ha,k1)@j & j<i)
  )
  "
```

Notice that, CreateNEKey is not possible as it generates non extractable keys. Once this lemma is proved all the sources are correctly computed and other lemmas can be proved based on this one because of the reuse label.

Sanity lemmas for rules. We check the following properties for the rules of Section 3.4:

Rule 1 Sensitive keys, i.e., keys that we proved to remain confidential in Section 4.4, have been either generated with attribute wrap_with_trusted set (SanityRule1_1), or with attribute extractable unset (SanityRule1_2 and 1_3). In the lemmas, we additionally check that these attributes cannot respectively be unset or set;

Rule 2 Only the SO can set the `trusted` attribute and the candidate keys are always generated by a KM with `extractable` unset (`SanityRule2_1` and `2_2`);

Rule 3 The candidate keys managed by the KMs should only admit `wrap` and `unwrap` cryptographic operations during their whole lifetime (`SanityRule3`);

Rule 4 The candidate keys have the `extractable` attribute unset during their whole lifetime (`SanityRule4`), which implies that they have been generated with the `extractable` attribute unset.

For what concerns Rule 5, the fact that the candidate keys managed by the KMs are generated as fresh in the device trivially derives from rule `CreateNEKey` that requires `Fr(k)` in the premise. So for this rule we do not prove any sanity lemma.

Sanity lemmas for actions. It is important to check that the model really does something and check that all the actions are possible. These sanity lemmas are all of the form `exists-trace` as they look for one trace satisfying the required formula. For example:

```
lemma SanityUsers:
exists-trace
"
Ex U1 U2 U3 #i1 #i2 #i3.
  NewSO(U1)@i1 & NewNU(U2)@i2 & NewKM(U3)@i3
"
```

checks that at least one user for each role can be created. We have similar lemmas for all the actions defined in the model.

Sanity lemmas for intuitive facts. We finally check some intuitive facts that are expected to hold:

`SanityUsersRole` checks that the same user can never have two distinct roles;

`SanityAttributesExtractable2` checks that the attribute `extractable` can be set only by `CreateKey`, `CreateWWTKey`, `ImportKey` and `Unwrap`;

`SanityAttributesWWT2` checks that `wrap_with_trusted` attribute can never be unset.

All lemmas can be proved automatically in about 1m30s on a MacBook Pro 2018. The full model and more detail about its automated verification are available at [21].

5 CONCLUSION

PKCS#11 is notoriously vulnerable and existing fixes require non-standard mechanisms and/or a reduced functionality. HSMs are increasingly offered in the cloud and they must adhere to PKCS#11 in order to offer a uniform and standard interface to applications. This makes proprietary fixes even less appealing and practical. Motivated by this urgency to configure HSMs in a secure way without loosing compliance with the PKCS#11 standard, we have proposed the first practical HSM configuration that does not require any restrictions or modifications of the PKCS#11 API.

In particular, we have explored for the first time users' roles for securing PKCS#11 by assuming a mild form of access control that only allows key owners to change key attributes. Based on this idea, we have defined five rules that we claim to be sufficient to prevent API-level attacks in PKCS#11 devices, preserving key confidentiality. We have substantiated our claim by providing a formal model of a significant subset of PKCS#11 and we have proved the security of the proposed rules on an unbounded number of users, keys and sessions. The model is expressive enough to cover all known API-level attacks, so the proof of security is promising and supports the claim that the proposed rules enforce security on the entire API. The proof is mechanized in Tamarin making it possible to easily verify further variants of the proposed configuration.

Interestingly, the simple access control underneath our configuration is offered by AWS CloudHSM and we have shown how our rules can be directly mapped and implemented in this real cloud HSM solution. In fact, the proposed configuration is rather simple to implement and one might wonder if more sophisticated configurations exist. For example, we might try to relax Rule 5 so that existing keys can be imported and used in the device as `trusted` keys. Unfortunately, this cannot be done in PKCS#11 as the only way to import a key securely is through unwrap and we know that unwrap can be abused to generate instances of the same key with conflicting attributes.

The simplicity of our solution in a sense reflects the simplicity of the wrap-with-trusted mechanism which unfortunately is not suitable for managing a key hierarchy in which the wrapping keys can themselves be wrapped by keys that are higher in the hierarchy (see, e.g., [13]). Note, in fact, that Rule 1 states that any sensitive key should either be generated with attribute `wrap_with_trusted` set, or with attribute `extractable` unset. Trusted keys are certainly sensitive and cannot be wrapped and unwrapped to prevent changes of attributes so the only possibility is to generate them with `extractable` unset (cf. Rule 4), meaning that they cannot be wrapped under other keys.

As a future work we intend to investigate if more sophisticated access control policies on keys can help building more complex and flexible secure configurations of PKCS#11. Moreover, we want to explore how to implement our configuration in other cloud HSM solutions. From a preliminary study we have seen that, unfortunately, some solutions such as Utimaco [38] and Microsoft [32] do not have a publicly available technical documentation. IBM Cloud HSM [27] implements an access management mechanism called Cloud Identity and Access Management (IAM) that regulates access to cryptographic operations. However, the form of key sharing required by our solution is only mentioned for key rings which do not support PKCS#11 keys [28]. Understanding whether our solution could be implemented in IBM Cloud HSM requires more investigation. Surprisingly, Google Cloud HSM documentation [26] does not mention PKCS#11 at all. From discussions in forums it seems like it is not (yet) supported. Finally, we plan to contact AWS and Oasis to discuss possible collaborations in terms of real implementations and standard inclusion.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their very interesting and insightful comments. This work has been partially supported by the European Regional Development Fund project *SAFE PLACE: Sistemi IoT per ambienti di vita salubri e sicuri* (POR FESR 2014-2020 AZIONE 1.1.4 DGR 822/2020 — ID 10288513).

REFERENCES

[1] P. Adão, R. Focardi, and F. L. Luccio. 2013. Type-Based Analysis of Generic Key Management APIs. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013.* 97–111.

[2] Amazon. 2021. AWS CloudHSM. Accessed in May 2021 at https://aws.amazon.com/cloudhsm/.

[3] Amazon 2021. *AWS CloudHSM User Guide.* Amazon. Accessed in May 2021 at https://docs.aws.amazon.com/cloudhsm/latest/userguide/cloudhsm-user-guide.pdf.

[4] R. Anderson. 2001. The Correctness of Crypto Transaction Sets. In *8th International Workshop on Security Protocols (LNCS, Vol. 2133).* Springer, 125–127. https://doi.org/10.1007/3-540-44810-1_17

[5] M. Bond. 2001. Attacks on Cryptoprocessor Transaction Sets. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01) (LNCS, Vol. 2162).* Springer, Paris, France, 220–234. https://doi.org/10.1007/3-540-44709-1_19.

[6] M. Bond and R. Anderson. 2001. API Level Attacks on Embedded Systems. *IEEE Computer Magazine* 34, 10 (October 2001), 67–75. https://doi.org/10.1109/2.955101

[7] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. 2010. Attacking and Fixing PKCS#11 Security Tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10).* ACM Press, Chicago, Illinois, USA, 260–269. https://doi.org/10.1145/1866307.1866337

[8] M. Centenaro, R. Focardi, and F.L. Luccio. 2013. Type-based analysis of key management in PKCS#11 cryptographic devices. *Journal of Computer Security* 21, 6 (2013), 971–1007. https://doi.org/10.3233/JCS-130479

[9] M. Centenaro, R. Focardi, F.L. Luccio, and G. Steel. 2009. Type-Based Analysis of PIN Processing APIs. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS 09)*, Springer LNCS vol. 5789/2009 (Ed.). 53–68. https://doi.org/10.1007/978-3-642-04444-1_4

[10] R. Clayton and M. Bond. 2003. Experience using a low-cost FPGA design to crack DES keys. In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded System (CHES'02) (LNCS, Vol. 2523).* Springer, 579–592. https://doi.org/10.1007/3-540-36400-5_42

[11] J. Clulow. 2003. *The Design and Analysis of Cryptographic APIs for Security Devices.* Master's thesis. University of Natal, Durban.

[12] J. Clulow. 2003. On The Security of PKCS#11. In *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03) (LNCS, Vol. 2779).* Springer, 411–425. https://doi.org/10.1007/978-3-540-45238-6_32

[13] V. Cortier and Steel G. 2014. A generic security API for symmetric key management on cryptographic devices. *Information and Computation* 238 (2014), 208–232. https://doi.org/10.1016/j.ic.2014.07.010

[14] C. Cremers and M. Dehnel-Wild. 2019. Component-Based Formal Analysis of 5G-AKA: Channel Assumptions and Session Confusion. In *26th Annual Network and Distributed System Security Symposium (NDSS'19) 2019, San Diego, California, USA, February 24-27, 2019.* The Internet Society. https://doi.org/10.14722/ndss.2019.23394

[15] C. Cremers, J. Fairoze, B. Kiesl, and A. Naska. 2020. Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20).* Association for Computing Machinery, New York, NY, USA, 1481–1495. https://doi.org/10.1145/3372297.3423354

[16] C. Cremers, B. Kiesl, and N. Medinger. 2020. A Formal Analysis of IEEE 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters. In *29th USENIX Security Symposium (USENIX'20).* USENIX Association, 1–17. https://www.usenix.org/conference/usenixsecurity20/presentation/cremers

[17] A. Dax, R. Künnemann, S. Tangermann, and M. Backes. 2019. How to Wrap it up - A Formally Verified Proposal for the use of Authenticated Wrapping in PKCS#11. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF'19).* 62–6215. https://doi.org/10.1109/CSF.2019.00012

[18] S. Delaune, S. Kremer, and G. Steel. 2010. Formal Analysis of PKCS#11 and Proprietary Extensions. *Journal of Computer Security* 18, 6 (Nov. 2010), 1211–1245. https://doi.org/10.3233/JCS-2009-0394

[19] D. Dolev and A. Yao. 1983. On the security of public key protocols. *IEEE Transactions in Information Theory* 2, 29 (March 1983), 198–208. https://doi.org/10.1109/TIT.1983.1056650

[20] R. Focardi and F. L. Luccio. 2020. Automated Analysis of PUF-based Protocols. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF).* IEEE Computer Society, Los Alamitos, CA, USA, 304–317. https://doi.org/10.1109/CSF49147.2020.00029

[21] R. Focardi and F. L. Luccio. 2021. A Formally Verified Configuration for Hardware Security Modules in the Cloud: Tamarin model. https://github.com/secgroup/CloudHSM-model.

[22] R. Focardi, F. L. Luccio, and G. Steel. 2011. An Introduction to Security API Analysis. In *Foundations of Security Analysis and Design VI - FOSAD Tutorial Lectures.* 35–65.

[23] S.B. Fröschle and N. Sommer. 2011. Concepts and Proofs for Configuring PKCS#11. In *Formal Aspects of Security and Trust - 8th International Workshop, (FAST'11), Leuven, Belgium, September 12-14, 2011. Revised Selected Papers (LNCS, Vol. 7140).* Springer, 131–147. https://doi.org/10.1007/978-3-642-29420-4_9

[24] F.D. Garcia, D.F. Oswald, Kasper T., and P. Pavlidès. 2016. Lock It and Still Lose It - on the (In)Security of Automotive Remote Keyless Entry Systems. In *25th USENIX Security Symposium (USENIX'16), Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/garcia

[25] G. Girol, L. Hirschi, Sasse R., D. Jackson, C. Cremers, and D. Basin. 2020. A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols. In *29th USENIX Security Symposium (USENIX'20).* USENIX Association, 1857–1874. https://www.usenix.org/conference/usenixsecurity20/presentation/girol

[26] Google. 2021. Cloud HSM. Accessed in May 2021 at https://cloud.google.com/kms/docs/hsm.

[27] IBM. 2021. Cloud HSM. Accessed in May 2021 at https://cloud.ibm.com/catalog/infrastructure/hardware-security-module.

[28] IBM. 2021. Cloud HSM documentation: Managing key rings. Accessed in September 2021 at https://cloud.ibm.com/docs/hs-crypto?topic=hs-crypto-managing-key-rings.

[29] R. Künnemann. 2015. Automated Backward Analysis of PKCS#11 v2.20. In *Principles of Security and Trust - 4th International Conference (POST'15) 2015, London, UK, April 11-18, 2015 (LNCS, Vol. 9036).* Springer, 219–238. https://doi.org/10.1007/978-3-662-46666-7_12

[30] D. Longley and S. Rigby. 1992. An Automatic Search for Security Flaws in Key Management Schemes. *Computers and Security* 11, 1 (March 1992), 75–89. https://doi.org/10.1016/0167-4048(92)90222-D

[31] S. Meier, Schmidt B., C. Cremers, and D. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Proceedings, of the 25th International Conference on Computer Aided Verification (CAV'13) 2013, Saint Petersburg, Russia, July 13-19, 2013.* 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

[32] Microsoft. 2021. Azure Dedicated HSM. Accessed in September 2021 at https://azure.microsoft.com/en-us/services/azure-dedicated-hsm/.

[33] OASIS 2014. *PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40.* OASIS. Accessed in May 2021 at http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.html.

[34] OASIS 2016. *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40.* OASIS. Accessed in May 2021 at http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html.

[35] OASIS 2020. *PKCS #11 Cryptographic Token Interface Base Specification Version 3.0.* OASIS. Accessed in May 2021 at https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html.

[36] R. Stanley-Oakes. 2017. A Provably Secure PKCS#11 Configuration Without Authenticated Attributes. In *Financial Cryptography and Data Security.* Springer International Publishing, 145–162. https://doi.org/10.1007/978-3-319-70972-7_8

[37] G. Steel. 2014. Proposal: Authenticated Attributes for Key Wrap in PKCS#11. Available at https://lists.oasis-open.org/archives/pkcs11/201408/msg00006/pkcs11-authenticated-encryption-key-transport.pdf.

[38] Utimaco. 2021. Cloud HSM. Accessed in May 2021 at https://hsm.utimaco.com/products-hardware-security-modules/form-factor/cloud-hsm-as-a-service/.

[39] R. Verdult, F.D. Garcia, and B. Ege. 2013. Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer. In *Proceedings of the 22th USENIX Security Symposium (USENIX'13), Washington, DC, USA, August 14-16, 2013*, Samuel T. King (Ed.). USENIX Association, 703–718. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/verdult

[40] P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. Rivest, and R. Anderson. 2005. *Robbing the bank with a theorem prover.* Technical Report UCAM-CL-TR-644. University of Cambridge. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-644.pdf.

A THE TAMARIN MODEL

The complete model in Tamarin is listed below and is publicly available at [21]:

```
/*
 * Tamarin model for the ACM CCS'21 paper:
 *
 * A Formally Verified Configuration for Hardware Security Modules in
 * the Cloud by R. Focardi and F. L. Luccio.
 *
 * Rationale: we model the trusted part and keep everything else Dolev-Yao.
 * We do not model untrusted keys in detail because we know they are
 * insecure-by-construction. Modeling just the trusted part of the API
 * simplifies the model and is a new contribution by itself.
 *
 * Some keys are leaked explicitly:
 *
 * - encrypt/decrypt keys (they can be used arbitrarily);
 *
 * - extractable and non wrap_with_trusted keys, as they are subject to
 *   known API-level attacks.
 *
 * Attacks found here are attacks in the full model. In order to distinguish
 * between a key and its value we actually leak a one-way hash of the key
 * and we always use the hash to perform cryptographic operations.
 *
 * Check with:
 * tamarin-prover --prove HSM_model_CCS_cameraready.spthy
 * (about 1m30s on a MacBook pro 2018)
 *
 * Check only the secrecy lemmas with:
 * tamarin-prover --prove=Secrecy* HSM_model_CCS_cameraready.spthy
 * (about 22s on a MacBook pro 2018)
 */

theory HSM_model_CCS

begin

builtins: symmetric-encryption, hashing

/*
 * User roles:
 *
 * NU: Normal user (for production applications)
 * KM: Key Manager (implemented as a normal PKCS11 user)
 * SO: Security Officer
 *
 * NOTE: users are fresh names to prevent role clashes. See also
 * lemma SanityUsersRole below.
 */
rule NewNU:
  [ Fr(U) ] --[ NewNU(U) ]-> [ !NU(U) ] // Normal User

rule NewKM:
  [ Fr(U) ] --[ NewKM(U) ]-> [ !NU(U) ] // Key Manager, a special NU

rule NewSO:
  [ Fr(U) ] --[ NewSO(U) ]-> [ !SO(U) ] // Security Officer

/*
 * Key creation and import.
 *
 * NOTE: Keys are created/imported with extractable set,
 * since it cannot be set later on, only unset.
 *
 * We also need keys generated:
 * - with extractable unset, that will become trusted (CreateNEKey)
 * - with wrap_with_trusted set, that will remain secure (CreateWWTKey)
 */
rule CreateKey:
  [ !NU(U), Fr(ha), Fr(k) ]
--[ CreateKey(U,ha,k),
    SetAttr(U,ha,'extractable') ]->
  [ !Key(U,ha,k) ]

rule CreateWWTKey:
  [ !NU(U), Fr(ha), Fr(k) ]
--[ CreateWWTKey(U,ha,k),
    SetAttr(U,ha,'wrap_with_trusted'),
    SetAttr(U,ha,'extractable') ]->
  [ !Key(U,ha,k) ]

rule CreateNEKey:
  [ !NU(U), Fr(ha), Fr(k) ]
--[ CreateNEKey(U,ha,k) ]->
  [ !Key(U,ha,k) ]

rule ImportKey:
  [ !NU(U), Fr(ha), In(k) ]
--[ ImportKey(U,ha,k),
    SetAttr(U, ha, 'extractable') ]->
  [ !Key(U,ha,k) ]

/*
 * This is never used in the model but just in lemmas/restrictions.
 * In some lemmas we need to check the mapping handle->key
 */
rule IsHandle:
  [ !Key(U,ha,k) ] --[ IsHandle(ha,k) ]-> [ ]

/*
 * Attributes
 *
 * - extractable: key is extractable, can only be unset;
 * - wrap: key can wrap other keys;
 * - unwrap: key can unwrap other keys;
 * - encrypt: key can encrypt data;
 * - decrypt: key can decrypt data;
 * - wrap_with_trusted: key can be wrapped under trusted keys,
 *   can be set but not unset.
 *   Note that we only model trusted wrapping so these are the
 *   only keys that are wrapped in the model;
 * - trusted: keys that can wrap wrap_with_trusted ones.
 *   These can only be set by the SO.
 */
rule UnsetAttrExtractable:
  [ !Key(U,ha,k), !NU(U) ] --[ UnsetAttr(U,ha,'extractable') ]-> [ ]

rule SetAttrWrap:
  [ !Key(U,ha,k), !NU(U) ] --[ SetAttr  (U,ha,'wrap')   ]-> [ ]
rule UnsetAttrWrap:
  [ !Key(U,ha,k), !NU(U) ] --[ UnsetAttr(U,ha,'wrap')   ]-> [ ]

rule SetAttrUnwrap:
  [ !Key(U,ha,k), !NU(U) ] --[ SetAttr  (U,ha,'unwrap') ]-> [ ]
rule UnsetAttrUnwrap:
  [ !Key(U,ha,k), !NU(U) ] --[ UnsetAttr(U,ha,'unwrap') ]-> [ ]

rule SetAttrEncrypt:
  [ !Key(U,ha,k), !NU(U) ] --[ SetAttr  (U,ha,'encrypt') ]-> [ ]
rule UnsetAttrEncrypt:
  [ !Key(U,ha,k), !NU(U) ] --[ UnsetAttr(U,ha,'encrypt') ]-> [ ]

rule SetAttrDecrypt:
  [ !Key(U,ha,k), !NU(U) ] --[ SetAttr  (U,ha,'decrypt') ]-> [ ]
rule UnsetAttrDecrypt:
  [ !Key(U,ha,k), !NU(U) ] --[ UnsetAttr(U,ha,'decrypt') ]-> [ ]

rule SetAttrWrapWithTrusted:
  [ !Key(U,ha,k), !NU(U) ] --[ SetAttr(U,ha,'wrap_with_trusted') ]-> [ ]

rule SetAttrTrusted:
  [ !Key(U,ha,k), !SO(W) ] --[ SetAttr  (W,ha,'trusted') ]-> [ ]
rule UnsetAttrTrusted:
  [ !Key(U,ha,k), !SO(W) ] --[ UnsetAttr(W,ha,'trusted') ]-> [ ]

/*
 * Trusted wrap: Key(U1,ha1,k1) can be wrapped under Key(U2,ha2,k2) if
 * - ha1 has wrap_with_trusted and extractable set;
 * - ha2 has trusted and wrap set.
 * The ciphertext senc(k1,k2) is sent as output.
 */
rule Wrap:
  [ !NU(U), !Key(U1,ha1,k1), !Key(U2,ha2,k2) ]
--[ Wrap(U,ha1,ha2),
    IsSet(ha1,'wrap_with_trusted'),
    IsSet(ha1,'extractable'),
    IsSet(ha2,'trusted'),
    IsSet(ha2,'wrap') ]->
  [ Out(senc(k1,h(k2))) ]

/*
 * Trusted unwrap: Key(U1,ha1,k1) can be unwrapped under Key(U2,ha2,k2) if
 * ha2 has trusted and unwrap set.
 * The new key Key(U1,ha1,k1) has wrap_with_trusted and extractable set.
 */
rule Unwrap:
  [ !NU(U1), !Key(U2,ha2,k2), In(senc(k1,h(k2))), Fr(ha1) ]
--[ Unwrap(U1,ha1,ha2),
    IsHandle(ha1,k1), // used in lemma
    IsSet(ha2,'trusted'),
    IsSet(ha2,'unwrap'),
    SetAttr(U1, ha1, 'wrap_with_trusted'),
    SetAttr(U1, ha1, 'extractable') ]->
  [ !Key(U1,ha1,k1) ]

/*
 * We leak (a hash of) encrypt/decrypt key so that attacker does
 * any crypto operation.
 */
rule LeakEncKey:
  [ !Key(U,ha,k) ]
--[ IsSet(ha,'encrypt') ]->
  [ Out(h(k)) ]

rule LeakDecKey:
  [ !Key(U,ha,k) ]
--[ IsSet(ha,'decrypt') ]->
  [ Out(h(k)) ]

/*
 * Non wrap_with_trusted keys that are extractable are implicitly
 * compromised.
 */
rule LeakExtractable:
  [ !Key(U,ha,k) ]
--[ IsSet(ha,'extractable'),
    IsUnset(ha,'wrap_with_trusted') ]->
  [ Out(h(k)) ]

/*
 * An attribute a is set over ha, written IsSet(ha,a) if there exist
 * a SetAttr(U,ha,a) before IsSet(ha,a) and any occurrence of
 * UnsetAttr(W,ha,a) before IsSet(ha,a) is also before SetAttr(U,ha,a).
 *
 * Intuitively: if there are many occurrences of SetAttr(U,ha,a) and
 * UnsetAttr(U,ha,a), it must be that SetAttr(U,ha,a) is the last one
 * before IsSet(ha,a).
 */
restriction IsSet:
"
All ha a #i . IsSet(ha,a)@i ==>
(
  Ex U #j . SetAttr(U,ha,a)@j & j<i &
  (All W #w . UnsetAttr(W,ha,a)@w & w<i ==> w<j)
)
"

/*
 * An attribute a is unset over ha, written IsUnset(ha,a) if
 * there exist an UnsetAttr(U,ha,a) before IsSet(ha,a) and any
 * SetAttr(W,ha,a) before IsSet(ha,a) is also before UnsetAttr(U,ha,a)
 * (this part is the dual of IsSet above)
 * OR
 * there exist no SetAttr(U,ha,a) before IsUnset(ha,a).
 */
```

```
restriction IsUnset:
"
All ha a #i . IsUnset(ha,a)@i ==>
(
  (Ex U #j . UnsetAttr(U,ha,a)@j & j<i &
    (All W #w . SetAttr(W,ha,a)@w & w<i ==> w<j))
  |
  (not Ex U #j . SetAttr(U,ha,a)@j & j<i)
)
"

/*
 * Restricting SO behavior:
 *
 * If a SO makes ha trusted then the key have been created as
 * non extractable by a KM.
 */
restriction SO:
"
All W ha #i . SetAttr(W,ha,'trusted')@i
==>
Ex k U #j #w .
  CreateNEKey(U,ha,k)@j & j<i &
  NewKM(U)@w & w<i
"

/*
 * Restricting KM behavior:
 *
 * KM can only make candidate keys wrap or unwrap.
 */
restriction KM:
"
All U ha k a #i #j #w.
  NewKM(U)@i &
  CreateNEKey(U,ha,k)@j &
  SetAttr(U,ha,a)@w
==>
  ( a='wrap' | a='unwrap' )
"

/*
 * This lemma is necessary to make the analysis convergent and efficient.
 * Tamarin cannot compute the possible sources of terms (because of partial
 * deconstructions) and the lemma tells that unwrapped keys come from previous
 * API operations such as CreateKey, CreateWWTKey and ImportKey. CreateNEKey
 * is not possible as it generates non extractable keys.
 */
lemma Unwrap [sources, reuse]:
"
All U k1 ha1 ha2 #i .
  Unwrap(U,ha1,ha2)@i &
  IsHandle(ha1,k1)@i
==>
(
  (Ex W ha #j . CreateKey(W,ha,k1)@j & j<i)
  |
  (Ex W ha #j . CreateWWTKey(W,ha,k1)@j & j<i)
  |
  (Ex W ha #j . ImportKey(W,ha,k1)@j & j<i)
)
"


/*
 * Sanity:
 *
 * Besides simple sanity lemmas that check that the model really does something
 * (i.e., they check that all the actions are possible, see below) we prove
 * that the five rules presented in the paper (cf. Section 3.4) are respected.
 */

/*
 * Rule 1: (Sensitive keys). Any sensitive key stored in the HSM should either
 * be generated with attribute wrap_with_trusted set, or with attribute
 * extractable unset.
 *
 * We have two specific actions corresponding to the above cases which set the
 * appropriate attributes. Notice that we will prove the secrecy of these
 * particular cases (cf. lemmas SecrecyWWT and SecrecyNE). We also have the
 * case of trusted keys. In particular:
 *
 * - CreateWWTKey: creates a key that has wrap_with_trusted set. We additionally
 *                 check that the attribute cannot be unset (SanityRule1_1)
 *
 * - CreateNEKey:  creates a key that has extractable unset. We actually
 *                 check that the attribute can never be set (SanityRule1_2)
 *
 * - Trusted keys: trusted keys should be sensitive. We prove that they are
 *                 are never set as extractable in their whole life-cycle
 *                 (SanityRule1_3)
 */

lemma SanityRule1_1:
"
All U ha k #i .
  CreateWWTKey(U,ha,k)@i
==> (
  SetAttr(U,ha,'wrap_with_trusted')@i &
  not Ex W #j . UnsetAttr(W,ha,'wrap_with_trusted')@j
)
"
```

```
lemma SanityRule1_2:
"
All U ha k #i .
  CreateNEKey(U,ha,k)@i
==> not Ex W #j . SetAttr(W,ha,'extractable')@j
"

lemma SanityRule1_3:
"
All U ha k #i #j.
  IsHandle(ha,k)@i &
  SetAttr(U,ha,'trusted')@j
==> not Ex W #w . SetAttr(W,ha,'extractable')@w
"

/*
 * Rule 2: (Trusted keys). The SO sets attribute trusted only on special
 * candidate keys generated by one of the KMs.
 *
 * We check that:
 *
 * - only the SO can set the attribute trusted (SanityRule2_1)
 *
 * - candidate keys are generated by a KM with extractable unset (SanityRule2_2)
 */
lemma SanityRule2_1:
"
All W ha #i . SetAttr(W,ha,'trusted')@i
==> Ex #j . NewSO(W)@j& j<i
"

lemma SanityRule2_2:
"
All W ha #i . SetAttr(W,ha,'trusted')@i
==> (
  Ex U k #j #w .
    NewKM(U)@j & j<i &
    CreateNEKey(U,ha,k)@w & j<w & w<i
)
"

/*
 * Rule 3: (Roles of candidate keys). The candidate keys managed by the KMs
 * should only admit wrap and unwrap cryptographic operations during their
 * whole lifetime.
 */
lemma SanityRule3:
"
All U W ha a #i #j #w.
  SetAttr(W,ha,'trusted')@i &
  SetAttr(U,ha,a)@j &
  NewKM(U)@w
==> ( a = 'wrap' | a = 'unwrap' )
"

/*
 * Rule 4: (Management of candidate keys). The candidate keys managed by the
 * KMs should be generated with the extractable attribute unset.
 *
 * We actually check that candidate keys have the extractable attribute unset
 * during their whole lifetime.
 */

lemma SanityRule4:
"
All W ha #i . SetAttr(W,ha,'trusted')@i
==> (
  (Ex U #j . NewKM(U)@j & j<i) &
  (not Ex U #j . SetAttr(U,ha,'extractable')@j )
)
"

/*
 * Rule 5: (Freshness of candidate keys). The candidate keys managed by the
 * KMs should be generated as fresh in the device.
 *
 * This comes implicitly from the definition of rule CreateNEKey that requires
 * Fr(k) in the premise.
 */

/*
 * The following sanity lemmas are important to check that the model really
 * does something.
 * We check that
 * - all the actions are possible (the model can be executed). These lemmas
 *   are of the form exists-trace as they look for one trace satisfying the
 *   required formula;
 * - some intuitive constraints hold (see comments).
 */

lemma SanityUsers:
exists-trace
"
Ex U1 U2 U3 #i1 #i2 #i3.
  NewSO(U1)@i1 &
  NewNU(U2)@i2 &
  NewKM(U3)@i3
"

/*
 * Users can only have one role. This is guaranteed by the freshness
 * of U in rules NewNU, NewKM, NewSO
 */
```

```
lemma SanityUsersRole:
"
All U1 U2 #i1 #i2 . NewSO(U1)@i1 & NewNU(U2)@i2 ==> not U1 = U2 &
All U1 U2 #i1 #i2 . NewSO(U1)@i1 & NewKM(U2)@i2 ==> not U1 = U2 &
All U1 U2 #i1 #i2 . NewNU(U1)@i1 & NewKM(U2)@i2 ==> not U1 = U2
"

lemma SanityKeys:
exists-trace
"
Ex U ha1 k1 ha2 k2 ha3 k3 ha4 k4 #i1 #i2 #i3 #i4
  .
  CreateKey   (U,ha1,k1)@i1 &
  CreateWWTKey(U,ha2,k2)@i2 &
  CreateNEKey (U,ha3,k3)@i3 &
  ImportKey   (U,ha4,k4)@i4
"

lemma SanityAttributesWrap:
exists-trace
"
Ex U ha #i #j.
  SetAttr  (U,ha,'wrap')@i &
  UnsetAttr(U,ha,'wrap')@j
"

lemma SanityAttributesUnwrap:
exists-trace
"
Ex U ha #i #j.
  SetAttr  (U,ha,'unwrap')@i &
  UnsetAttr(U,ha,'unwrap')@j
"

lemma SanityAttributesEncrypt:
exists-trace
"
Ex U ha #i #j.
  SetAttr  (U,ha,'encrypt')@i &
  UnsetAttr(U,ha,'encrypt')@j
"

lemma SanityAttributesDecrypt:
exists-trace
"
Ex U ha #i #j.
  SetAttr  (U,ha,'decrypt')@i &
  UnsetAttr(U,ha,'decrypt')@j
"

lemma SanityAttributesTrusted:
exists-trace
"
Ex U ha #i #j.
  SetAttr  (U,ha,'trusted')@i &
  UnsetAttr(U,ha,'trusted')@j
"

lemma SanityAttributesExtractable1:
exists-trace
"
Ex U ha #i.
  UnsetAttr(U,ha,'extractable')@i
"

/*
 * extractable can be set only by the following rules:
 * - CreateKey
 * - CreateWWTKey
 * - ImportKey
 * - Unwrap
 */
lemma SanityAttributesExtractable2:
"
All U ha #i. SetAttr  (U,ha,'extractable')@i
==> (
  (Ex k . CreateKey(U,ha,k)@i    ) |
  (Ex k . CreateWWTKey(U,ha,k)@i ) |
  (Ex k . ImportKey(U,ha,k)@i    ) |
  (Ex ha2 . Unwrap(U,ha,ha2)@i   )
)
"

lemma SanityAttributesWWT1:
exists-trace
"
Ex U ha #i .
  SetAttr  (U,ha,'wrap_with_trusted')@i
"

/* wrap_with_trusted cannot be unset */
lemma SanityAttributesWWT2:
"
All U ha #i .
  UnsetAttr(U,ha,'wrap_with_trusted')@i
  ==> F
"

lemma SanityWrap :
exists-trace
"
Ex U ha1 ha2  #i . Wrap(U,ha1,ha2)@i
"

lemma SanityWrapWWT :
exists-trace
"
Ex U W ha1 k1 ha2  #i #j.
  Wrap  (U,ha1,ha2)@i & CreateWWTKey(W,ha1,k1)@j & j<i
"

lemma SanityUnwrap :
exists-trace
"
Ex U ha1 ha2  #i .
  Unwrap(U,ha1,ha2)@i
"

/*
 * Proof of security:
 *
 * SecrecyTrusted: trusted keys are never leaked
 * SecrecyNE: non extractable keys are never leaked
 * SecrecyWWT: wrap_with_trusted keys are never leaked
 */


/*
 * Keys which are generated as with extractable unset are never leaked.
 * This lemma speeds-up consistently the proof of next ones so we prove it
 * as first with the "reuse" label.
 */
lemma SecrecyNE [reuse]:
"
All U ha k #i #j .
  CreateNEKey(U,ha,k)@i &
  KU(k)@j
==> F
"

/*
 * Keys which, at some point, are marked as trusted are never leaked.
 */
lemma SecrecyTrusted:
"
All W ha k #i #j #w.
  IsHandle(ha,k)@i &
  SetAttr(W,ha,'trusted')@j &
  KU(k)@w
==> F
"

/*
 * Keys which are generated with wrap_with_trusted set are never leaked.
 */
lemma SecrecyWWT:
"
All U ha k #i #j .
  CreateWWTKey(U,ha,k)@i &
  KU(k)@j
==> F
"

end
```