2017 IEEE 30th Computer Security Foundations Symposium

Run-time Attack Detection in Cryptographic APIs

Riccardo Focardi and Marco Squarcina
Università Ca' Foscari Venezia, Italy
Cryptosense, France
Email: {focardi, squarcina}@unive.it

***Abstract*—Cryptographic APIs are often vulnerable to attacks that compromise sensitive cryptographic keys. In the literature we find many proposals for preventing or mitigating such attacks but they typically require to modify the API or to configure it in a way that might break existing applications. This makes it hard to adopt such proposals, especially because security APIs are often used in highly sensitive settings, such as financial and critical infrastructures, where systems are rarely modified and legacy applications are very common. In this paper we take a different approach. We propose an effective method to monitor existing cryptographic systems in order to detect, and possibly prevent, the leakage of sensitive cryptographic keys. The method collects logs for various devices and cryptographic services and is able to detect, offline, any leakage of sensitive keys, under the assumption that a *key fingerprint* is provided for each sensitive key. We define key security formally and we prove that the method is sound, complete and efficient. We also show that without key fingerprinting completeness is lost, i.e., some attacks cannot be detected. We discuss possible practical implementations and we develop a proof-of-concept log analysis tool for PKCS#11 that is able to detect, on a significant fragment of the API, all key-management attacks from the literature.**

I. Introduction

Cryptography is one of the dominant technologies to provide security in various settings and cryptographic hardware and services are becoming more and more pervasive in everyday applications. The interfaces to cryptographic devices and services are implemented as *Security APIs* that allow untrusted code to access resources in a secure way. These APIs provide various functionalities such as: the creation or deletion of keys; the encryption, decryption, signing and verification of data under some key; the import and export of *sensitive* keys, i.e., keys that should never be revealed as plaintext outside smartcards and hardware security modules [18], [20].

Cryptographic APIs have been found vulnerable to many attacks that compromise sensitive cryptographic keys (see, e.g., [2], [4], [5], [8]). Some attacks are related to the key wrapping operation: for example, attacks on the IBM CCA interface are due to the improper way of binding a cryptographic key to its usage rules through the XOR function [4], and attacks on security tokens can be mounted by assigning particular sets of attributes to the keys, and by performing particular sequences of (legal) API calls [5]. Other attacks, e.g., the ones on PIN processing APIs, are based on formats used for message encryption [9], or on the lack of integrity of user data [6].

In the literature we find many proposals for preventing or mitigating such attacks, but they typically require to modify the API or to configure it in a way that might break existing applications (see, e.g. [5], [7], [11], [12], [15], [17]). This makes it hard to adopt such proposals, especially because security APIs are often used in highly sensitive settings, such as financial and critical infrastructures, where systems are rarely modified and legacy applications are very common. Notice that, in these settings, the leakage of a cryptographic key might have very serious consequences such as decrypting confidential data, breaking integrity or forging digitally signed documents and transactions. It is thus of ultimate importance to introduce mechanisms that can detect or prevent attacks and that can be also deployed in practice.

In this paper we explore a different approach. Instead of trying to fix the API or developing a new secure one, we propose an effective method that can be used to monitor existing systems in order to detect, and possibly prevent, the leakage of sensitive cryptographic keys. The method collects logs for various devices and is able to detect, offline, any leakage of sensitive keys. For example, by tracking keys we may discover that a sensitive key is being wrapped under an untrusted one, that might be known to the attacker; whenever a sensitive key is leaked in the clear, as in the so-called *wrap/decrypt* attack [10], we are able to identify the problem through a special *key fingerprinting* functionality, that allows for an efficient offline log analysis without affecting in any way the cryptographic application.

Challenges: Devising a run-time analysis of cryptographic APIs presents many challenges. First of all, it needs to track the usage of any sensitive key, without exposing its value. Cryptographic APIs store cryptographic keys securely and give access to them through handles so that it is not necessary to know key values to perform operations. Monitoring keys that are referred through handles can be tricky. In particular, when a key is leaked it is not possible to discover this immediately, since the key value is not available.

The analysis must be very accurate: false positives might result in unnecessary key updates of all keys that are believed to be leaked, while false negatives would miss actual key leakages, with possible serious consequences. In this respect, it is of ultimate importance that any proposed monitoring method is supported by a formal proof of soundness and completeness with respect to a class of attacks.

The analysis should work on distributed executions across many devices or applications. Cryptographic keys are often shared among devices and services and API level attacks can

DOI 10.1109/CSF.2017.33

thus be effectively run in a distributed fashion, with the aim of bypassing any local monitoring. Thus, it is important that the analysis is able to collect logs from various sources and check them consistently, in order to find attacks that might leak a key of one cryptographic service through another one, in a different physical location.

Finally, the analysis should be efficient and should be able to scale on fairly big logs. Ideally, the monitor should continuously collect distributed logs and perform the analysis in real-time. Once the analysis has been proved accurate and suitably tested, the monitor might run "in the middle" of the API calls, and could be able to spot attacks on the fly and prevent them, by blocking the call right before the key is leaked.

Contributions: We contribute to the state of the art in various respects: (i) we model the problem of run-time detection of cryptographic API attacks. Our model captures distributed attacks, i.e., attacks performed by executing API calls on different devices and services; (ii) we provide a sound and complete characterization of attacks based on the monitoring of a subset of API calls; (iii) we prove that the problem of finding attacks cannot be decided in general because, intuitively, it is not possible to distinguish sensitive keys from non-sensitive ones just by their values; (iv) we propose a key fingerprinting abstract mechanism and a run-time analysis that is sound, complete and efficient; key fingerprinting is only used for logging purposes in order to make the analysis feasible and accurate and it does not affect the cryptographic applications invoking the API; (v) we discuss practical implementations and we develop a proof-of-concept log analysis tool for PKCS#11, the RSA standard interface for cryptographic tokens [18], [20]. The tool is able to detect, on a significant fragment of the API, all key-management attacks reported in [12], [14].

Related Work: The first paper that has applied general analysis tools to the analysis of security APIs is [22], but no formal statement of the security guarantees provided by the analysis was done. The first automated analysis of PKCS#11 with a formal statement of the underlying assumptions has been presented in [12]. In [5], the model of [12] has been generalized and provided with a reverse-engineering tool that automatically refines the model depending on the actual behavior 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 [5] has successfully found attacks that leak the value of sensitive keys on real devices. In [1], [7], type-based techniques have been used to statically analyze the security of cryptographic API specification. Computational security guarantees of cryptographic APIs have been studied in [16], [17], [21].

All of above works aim at analyzing a given API specification or configuration, looking for attack sequences or proving the absence of attacks. None of them perform a run-time analysis of API invocation sequences.

Caml Crush [3] is a PKCS#11 Filtering Proxy that can be configured to prevent dangerous PKCS#11 commands and mechanisms. Caml Crush performs a run-time analysis, but there are important differences with respect to our proposal: (i) Caml Crush modifies the API behavior by imposing restrictions that prevent attacks. For example it prevents keys to be assigned conflicting roles so that attacks such as Clulow's wrap/decrypt are prevented. Our approach is different: we do not impose any restriction on how keys are configured and used, and in fact we do not even consider key attributes, since our method is independent of the specific API. We just monitor the API calls that can be responsible of leaking a sensitive key. While Caml Crush can break applications that do not adhere to the imposed policy, even when no key leakage happens, our approach is more accurate and only stops the calls that are responsible of key leakages greatly reducing the number of false positives; (ii) Caml Crush does not track keys on multiple devices and is thus unable to prevent the distributed attacks we discuss in Section III, unless the wrapping API is modified. More precisely, in Caml Crush it is possible to enable an ad hoc modification of the wrapping API that tracks key attributes but makes the API incompatible with the standard PKCS#11 one: keys wrapped under Caml Crush modified wrapping API cannot be unwrapped on standard devices; (iii) our method can work offline by simply analyzing logs, while Caml Crush requires an online component to actively monitor API calls. Together with (i), we believe that this is a fundamental feature that might facilitate the adoption of the method, since it would never interfere with (and possibly break) applications, a crucial requirement in critical settings (e.g., banks); (iv) Camel Crush is tailored to PKCS#11 while our approach is more general in principle. Indeed, the method we propose does not rely on any specific feature of PKCS#11 such as key attributes. It only requires the specification of which keys are sensitive (i.e., not accessible in the clear), a basic security property useful in any key management API (e.g., Microsoft CAPI and CNG, Java JCA).

The model presented in this work is based on the one in [12] but there are important differences: we remove from the model any detail that is specific of PKCS#11, in order to model generic cryptographic APIs. We define a notion of local and distributed secure execution that formalizes when a specific execution is secure with respect to a set of sensitive keys. In particular, our executions are not regulated by any key policy or attributes as in [12]. Finally, the focus of [12] is the discovery of sequences of API calls that might leak a key. Here, instead, we study how to detect attacks on given (distributed) sequences of API calls by log inspection, i.e., without knowledge of the actual key values.

Structure of the Paper: In Section II we present the core formal model, which is based on [12]; Section III introduces our notion of secure distributed execution that we characterize in terms of the analysis of a subset of the API calls; Section IV presents our method for the run-time analysis based on key fingerprinting and we prove that it has linear complexity with

respect to the length of logs and the number of sensitive keys; in Section V we report on a prototype implementation in PKCS#11, where we implement key fingerprinting through standard API calls. We show that the tool can effectively detect and prevent known attacks on a significant fragment of PKCS#11 key management; Section VI draws some concluding remarks.

II. Core Model

The core of our model is a variation of the one introduced by Delaune, Kremer and Steel (DKS) [12], in which anything specific to PKCS#11 has been removed, and with labels referring to API calls on the transitions. The latter will be required to formalize secure executions in Section III.

The attacker is assumed to be able to call commands of the API in any order providing any known value. Data and keys are modeled as terms and the rules of the API and the abilities of an attacker are written as rules that, given some terms, produce new ones. Cryptography is modeled symbolically: the intruder is assumed not to be able to break cryptography by brute-force or cryptanalysis, i.e., (s)he can only read an encrypted message if (s)he knows the correct key, along the standard Dolev-Yao approach [13]. Since the attacker is at the API level, we do not distinguish between malicious or legitimate users.

A. Syntax

As in DKS, we assume a given *signature* Σ, i.e., a finite set of *function symbols*, with an arity function $ar : \Sigma \to \mathbb{N}$, a (possibly infinite) set of *names* $\mathcal{N}$ and a (possibly infinite) set of *variables* $\mathcal{X}$. Names represent keys, data values, nonces, etc. Function symbols model cryptographic primitives. We also denote with Σ_{api} the set of API function symbols and extend the arity function to this set in the expected way. The set of *plain terms* $\mathcal{PT}(\Sigma, \mathcal{N}, \mathcal{X})$ is defined by the following grammar

$$\begin{array}{rcll} \mathsf{t} & := & \mathsf{x} & \mathsf{x} \in \mathcal{X} \\ & | & \mathsf{n} & \mathsf{n} \in \mathcal{N} \\ & | & \mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_j) & \mathsf{f} \in \Sigma \text{ and } ar(\mathsf{f}) = j \end{array}$$

The set $\mathcal{PT}(\Sigma, \mathcal{N}, \emptyset)$, also referred to as $\mathcal{PT}(\Sigma, \mathcal{N})$, is called the set of *ground terms*. We use $vars(\mathsf{t})$ and $names(\mathsf{t})$ for the set of variables and names that occur in the term t and extend the notations to set of terms.

We simplify the DKS model by removing the set of literals from each rule. As a result, user's capabilities are not restricted by the attributes assigned to key handles. Additionally, we make explicit the API function call used to fire a rule by including the function as a label. The description of the system is given as a finite set of rules $\mathcal{R}$ of the form

$$T \xrightarrow[\mathsf{new}\ \tilde{\mathsf{n}}]{\mathsf{f}} T'$$

where $T, T' \subseteq \mathcal{PT}$ are sets of plain terms, $\tilde{\mathsf{n}} \subseteq \mathcal{N}$ is a set of names and $\mathsf{f} \in \Sigma_{api}$ is an API function symbol. When $\tilde{\mathsf{n}} = \emptyset$, we omit $\mathsf{new}\,\tilde{\mathsf{n}}$ from the rule.

Intuitively, the rule can be fired when all the terms in T are in the user knowledge and the API function f is invoked. The effect of the rule is that the user knowledge is augmented with terms in T'. The $\mathsf{new}\,\tilde{\mathsf{n}}$ means that all the names in $\tilde{\mathsf{n}}$ need to be replaced by fresh names in T'. This models nonce or key generation: if the rule is executed several times, the effects are different as different names will be used each time.

We consider the signature $\Sigma = \{\mathsf{senc}, \mathsf{aenc}, \mathsf{pub}, \mathsf{priv}, \mathsf{h}\}$, as in DKS. The function symbols senc and aenc of arity 2 represent symmetric and asymmetric encryption, whereas pub and priv of arity 1 are constructors to obtain public and private keys, respectively. The symbol h allows to model key handles.

Example 1 (Ciphertext, Keys and Handles)**.** *We show a few examples of ciphertext, key and handle terms. Term* $\mathsf{senc}(\mathsf{k}_2, \mathsf{k}_1)$ *represents key* k_2 *encrypted under symmetric key* k_1*. Private key* $\mathsf{priv}(\mathsf{s})$ *and public key* $\mathsf{pub}(\mathsf{s})$ *represent a keypair generated from a common seed* s*. Finally* $\mathsf{h}(\mathsf{n}, \mathsf{k})$ *is a handle referring to key* k*. Nonce* n *is used to make it possible to have multiple handles, e.g.,* $\mathsf{h}(\mathsf{n}, \mathsf{k})$ *and* $\mathsf{h}(\mathsf{n}', \mathsf{k})$*, for the same key* k*. Notice that from a handle it is not possible to recover the value of the key.*

We consider the following set of API functions

$$\begin{aligned} \Sigma_{api} = \{ & \mathsf{KeyGen}, \mathsf{KeyPairGen}, \\ & \mathsf{Wrap}_{\mathsf{ss}}, \mathsf{Wrap}_{\mathsf{sa}}, \mathsf{Wrap}_{\mathsf{as}}, \\ & \mathsf{Unwrap}_{\mathsf{ss}}, \mathsf{Unwrap}_{\mathsf{sa}}, \mathsf{Unwrap}_{\mathsf{as}}, \\ & \mathsf{Encrypt}_{\mathsf{s}}, \mathsf{Encrypt}_{\mathsf{a}}, \\ & \mathsf{Decrypt}_{\mathsf{s}}, \mathsf{Decrypt}_{\mathsf{a}} \} \end{aligned}$$

Intuitively, KeyGen, $\mathsf{KeyPairGen}$ are nullary functions for generating symmetric keys and key pairs, respectively; $\mathsf{Wrap}_{\mathsf{ss}}, \mathsf{Wrap}_{\mathsf{sa}}, \mathsf{Wrap}_{\mathsf{as}}$ and $\mathsf{Unwrap}_{\mathsf{ss}}, \mathsf{Unwrap}_{\mathsf{sa}}, \mathsf{Unwrap}_{\mathsf{as}}$ are used to respectively wrap and unwrap keys under other keys. We model the common cases of wrapping a symmetric key under a symmetric and an asymmetric one plus the case of wrapping an asymmetric key under a symmetric one. Wrap operations take two key handles as arguments while unwrap operations take a handle and a ciphertext and generate a new handle in the device, pointing to the unwrapped key. Finally, $\mathsf{Encrypt}_{\mathsf{s}}, \mathsf{Encrypt}_{\mathsf{a}}$ and $\mathsf{Decrypt}_{\mathsf{s}}, \mathsf{Decrypt}_{\mathsf{a}}$ perform symmetric and asymmetric encryption and decryption. They respectively take as arguments a plaintext/ciphertext and the handle of the encryption/decryption key. The set of rules of our model are listed in Table I.

Example 2 (Wrap API)**.** *As an example, consider the rule*

$$\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1),\ \mathsf{h}(\mathsf{x}_2, \mathsf{y}_2) \xrightarrow{\mathsf{Wrap}_{\mathsf{ss}}} \mathsf{senc}(\mathsf{y}_2, \mathsf{y}_1)$$

used to wrap a symmetric key with another symmetric key. We have that $\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1)$ *and* $\mathsf{h}(\mathsf{x}_2, \mathsf{y}_2)$ *are handles for keys* y_1 *and* y_2*, respectively, while* $\mathsf{senc}(\mathsf{y}_2, \mathsf{y}_1)$ *is the symmetric encryption of* y_2 *under* y_1*. The rule states that the key* y_2 *can be wrapped, i.e., encrypted, with* y_1 *when the API function* $\mathsf{Wrap}_{\mathsf{ss}}$ *is fired and both handles for keys* $\mathsf{y}_1, \mathsf{y}_2$ *are known. The wrapped key is then added to the set of known terms.*

$$\xrightarrow[\text{new } n, k]{\mathsf{KeyGen}} \mathsf{h}(n,\, k)$$
$$\xrightarrow[\text{new } n, s]{\mathsf{KeyPairGen}} \mathsf{h}(n,\, \mathsf{priv}(s)),\ \mathsf{pub}(s)$$

$$\mathsf{h}(x_1,\, y_1),\ \mathsf{h}(x_2,\, y_2) \xrightarrow{\mathsf{Wrap_{ss}}} \mathsf{senc}(y_2,\, y_1)$$
$$\mathsf{h}(x_1,\, \mathsf{priv}(z)),\ \mathsf{h}(x_2,\, y_2) \xrightarrow{\mathsf{Wrap_{sa}}} \mathsf{aenc}(y_2,\, \mathsf{pub}(z))$$
$$\mathsf{h}(x_1,\, y_1),\ \mathsf{h}(x_2,\, \mathsf{priv}(z)) \xrightarrow{\mathsf{Wrap_{as}}} \mathsf{senc}(\mathsf{priv}(z),\, y_1)$$

$$\mathsf{h}(x,\, y_2), \mathsf{senc}(y_1,\, y_2) \xrightarrow[\text{new } n_1]{\mathsf{Unwrap_{ss}}} \mathsf{h}(n_1,\, y_1)$$
$$\mathsf{h}(x,\, \mathsf{priv}(z)), \mathsf{aenc}(y_1,\, \mathsf{pub}(z)) \xrightarrow[\text{new } n_1]{\mathsf{Unwrap_{sa}}} \mathsf{h}(n_1,\, y_1)$$
$$\mathsf{h}(x,\, y_2), \mathsf{senc}(\mathsf{priv}(z),\, y_2) \xrightarrow[\text{new } n_1]{\mathsf{Unwrap_{as}}} \mathsf{h}(n_1,\, \mathsf{priv}(z))$$

$$\mathsf{h}(x_1,\, y_1), y_2 \xrightarrow{\mathsf{Encrypt_s}} \mathsf{senc}(y_2,\, y_1)$$
$$\mathsf{h}(x_1,\, y_1), \mathsf{senc}(y_2,\, y_1) \xrightarrow{\mathsf{Decrypt_s}} y_2$$

$$\mathsf{h}(x_1,\, \mathsf{priv}(z)), y_1 \xrightarrow{\mathsf{Encrypt_a}} \mathsf{aenc}(y_1,\, \mathsf{pub}(z))$$
$$\mathsf{h}(x_1,\, \mathsf{priv}(z)), \mathsf{aenc}(y_2,\, \mathsf{pub}(z)) \xrightarrow{\mathsf{Decrypt_a}} y_2$$

TABLE I
API RULES

B. Semantics

We enrich the semantics of DKS with labels, as they will be required for the run-time analysis. The semantics is thus defined in terms of a labeled transition system $(Q, A, \twoheadrightarrow, q_0)$. Q defines the set of possible states, where each state $q \subseteq \mathcal{PT}(\Sigma, \mathcal{N})$ is the set of ground terms in the user's knowledge. A is the set of actions such as

$$A = \{\mathsf{f}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \mid \mathsf{f} \in \Sigma_{api},\ n = ar(\mathsf{f}), \\ \forall\, i \in [1, n] : \mathsf{t}_i \in \mathcal{PT}(\Sigma, \mathcal{N})\}$$

Given a rule $a \in A$, we write $args(a) \subseteq \mathcal{PT}(\Sigma, \mathcal{N})$ for the set $\{\mathsf{t}_1, \ldots, \mathsf{t}_n\}$ of arguments of a. The initial state $q_0 \in Q$ represents the initial knowledge of the user. The transition relation $\twoheadrightarrow \subseteq Q \times A \times Q$ is defined as follows. We have that $q \xrightarrow{a}\!\!\!\!\twoheadrightarrow q'$ if

$$R := T \xrightarrow{\mathsf{f}} T'$$

is a fresh renaming w.r.t. $names(q)$ of a rule in $\mathcal{R}$ and there exists a grounding substitution θ for R such that $T\theta \subseteq q$ and given $a = \mathsf{f}'(t_1, \ldots, t_n)$ we have that $\mathsf{f}' = \mathsf{f}$, $ar(\mathsf{f}) = n$ and $args(a) = T\theta$. Then $q' = q \cup T'\theta$.

Given a LTS $P = (Q, A, \twoheadrightarrow, q_0)$, an execution is a sequence of transitions

$$q_0 \xrightarrow{a_1}\!\!\!\!\twoheadrightarrow q_1 \xrightarrow{a_2}\!\!\!\!\twoheadrightarrow \ldots \xrightarrow{a_n}\!\!\!\!\twoheadrightarrow q_n$$

that we abbreviate as $q_0 \xrightarrow{\alpha}\!\!\!\!\twoheadrightarrow^* q_n$, with $\alpha = a_1, a_2, \ldots, a_n$.

Example 3 (Wrap and Decrypt Attack)**.** *Consider, for example, the execution representing a wrap/decrypt attack in which the value of a key* k_2 *is exposed by wrapping* k_2 *with* k_1 *and then by decrypting the wrapped data with* k_1*. Given an initial state* $q_0 = \{\mathsf{h}(\mathsf{n}_1,\, \mathsf{k}_1), \mathsf{h}(\mathsf{n}_2,\, \mathsf{k}_2)\}$*, we have that*

$$q_0 \xrightarrow{\mathsf{Wrap_{ss}}(\mathsf{h}(\mathsf{n}_1,\, \mathsf{k}_1),\, \mathsf{h}(\mathsf{n}_2,\, \mathsf{k}_2))}\!\!\!\!\twoheadrightarrow q_1 \qquad q_1 = q_0 \cup \{\mathsf{senc}(\mathsf{k}_2,\, \mathsf{k}_1)\}$$
$$q_1 \xrightarrow{\mathsf{Decrypt_s}(\mathsf{h}(\mathsf{n}_1,\, \mathsf{k}_1),\, \mathsf{senc}(\mathsf{k}_2,\, \mathsf{k}_1))}\!\!\!\!\twoheadrightarrow q_2 \quad q_2 = q_1 \cup \{\mathsf{k}_2\}$$

That we write $q_0 \xrightarrow{\alpha}\!\!\!\!\twoheadrightarrow^* q_2$*, with*

$$\alpha = \mathsf{Wrap_{ss}}(\mathsf{h}(\mathsf{n}_1,\, \mathsf{k}_1),\, \mathsf{h}(\mathsf{n}_2,\, \mathsf{k}_2)), \\ \mathsf{Decrypt_s}(\mathsf{h}(\mathsf{n}_1,\, \mathsf{k}_1),\, \mathsf{senc}(\mathsf{k}_2,\, \mathsf{k}_1))$$

α defines the sequence of actions performed by the attacker to access the value of k_2 *by reaching the state* q_2*.*

III. Secure Executions

Our notion of secure execution is parametric with respect to a set of secure key, which might be different for different executions. Intuitively, we want to let each administrator to specify, locally, the set of sensitive keys that need to be monitored. Thus, what is sensitive is not established globally but we need to be able to compose local executions and recover a partial view of what is sensitive, so to capture distributed attacks, i.e., attacks in which one key might be leaked on a different device in order to bypass local monitoring.

A. Secure Local Executions

We let SK denote a set of sensitive keys that we want to monitor in a certain local execution. Secure keys are either

$$\begin{aligned} \mathsf{y}_1, \mathsf{y}_2 &\xrightarrow{\text{DY}} \mathsf{senc}(\mathsf{y}_2, \mathsf{y}_1) \\ \mathsf{senc}(\mathsf{y}_2, \mathsf{y}_1), \mathsf{y}_1 &\xrightarrow{\text{DY}} \mathsf{y}_2 \\ \mathsf{senc}(\mathsf{y}_2, \mathsf{y}_i) &\xrightarrow{\text{DY}} \mathsf{y}_2 \end{aligned}$$

$$\begin{aligned} \mathsf{y}_1, \mathsf{y}_2 &\xrightarrow{\text{DY}} \mathsf{aenc}(\mathsf{y}_2, \mathsf{y}_1) \\ \mathsf{aenc}(\mathsf{y}_2, \mathsf{pub}(\mathsf{z})), \mathsf{priv}(\mathsf{z}) &\xrightarrow{\text{DY}} \mathsf{y}_2 \\ \mathsf{aenc}(\mathsf{y}_2, \mathsf{pub}(\mathsf{z}_i)) &\xrightarrow{\text{DY}} \mathsf{y}_2 \end{aligned}$$

TABLE II
DOLEV-YAO SK-RULES, WITH INSECURE KEYS $\mathsf{k}_i, \mathsf{priv}(\mathsf{s}_i) \notin SK$ RANGED OVER BY $\mathsf{y}_i, \mathsf{priv}(\mathsf{z}_i)$

symmetric keys k or private asymmetric keys $\mathsf{priv}(\mathsf{s})$. In the following we let K range over k and $\mathsf{priv}(\mathsf{s})$. We only consider executions starting from a state q_0 in which all of the secure keys are safely stored in the device. They should not be publicly known or encrypted under an insecure key. Formally:

Definition 1 (SK-Secure Initial State)**.** *Let SK be a set of sensitive keys. An initial state q_0 is secure if any secure key K does not appear in q_0 in the forms k, $\mathsf{senc}(\mathsf{k}, \mathsf{k}_i)$ and $\mathsf{aenc}(\mathsf{k}, \mathsf{pub}(\mathsf{s}_i))$, with $\mathsf{k}_i, \mathsf{priv}(\mathsf{s}_i) \notin SK$.*

From now on we will only consider executions with secure initial states.

We consider a Dolev-Yao attacker parametrized by SK that can perform encryption and decryption operations using known keys (as usual) plus any insecure key $\mathsf{k}_i, \mathsf{priv}(\mathsf{s}_i) \notin SK$, ranged over by $\mathsf{y}_i, \mathsf{priv}(\mathsf{z}_i)$. Attacker is formalized by the rules in Table II. From now on we assume that executions can include attacker's actions.

An execution is secure if and only if it does not leak any of its secure key:

Definition 2 (SK-Secure Execution)**.** *Let $\sigma = q_0 \xrightarrow{\alpha} \!\!\!\!\rightarrow^* q_n$ be an execution. Then, σ is SK-*secure *if and only if $SK \cap q_n = \emptyset$.*

Notice that freshly generated keys may or may not be included in SK. There might be cases in which an administrator wants to monitor new keys, and other situations in which new keys are just session keys that are destroyed when the session is closed, and does not need to be monitored. Both these situations can be modeled by including or not new keys in the set SK.

A SK-secure execution is also secure with respect to any subset of SK, i.e., with respect to strictly less sensitive keys. This is proved formally in the following lemma:

Lemma 1. *Let σ be SK-secure. Then σ is SK'-secure for each $SK' \subseteq SK$.*

Proof: Trivial since $SK' \cap q_n \subseteq SK \cap q_n = \emptyset$. ∎

We now prove that in any insecure execution there is at least either a wrap operation of a secure key under an insecure one, or a decrypt operation of a secure key encrypted under another secure key.

Proposition 1. *Let $\sigma = q_0 \xrightarrow{\alpha} \!\!\!\!\rightarrow^* q_n$ be an execution. Then, σ is SK-secure if and only if none of the following is in σ:*

1) $\mathsf{Wrap}_{\mathsf{ss}}(\mathsf{h}(\mathsf{n}_i, \mathsf{k}_i), \mathsf{h}(\mathsf{n}_s, \mathsf{k}_s))$;
2) $\mathsf{Wrap}_{\mathsf{sa}}(\mathsf{h}(\mathsf{n}_i, \mathsf{priv}(\mathsf{s}_i)), \mathsf{h}(\mathsf{n}_s, \mathsf{k}_s))$;
3) $\mathsf{Wrap}_{\mathsf{as}}(\mathsf{h}(\mathsf{n}_i, \mathsf{k}_i), \mathsf{h}(\mathsf{n}_s, \mathsf{priv}(\mathsf{s}_s)))$;
4) $\mathsf{Decrypt}_{\mathsf{s}}(\mathsf{h}(\mathsf{n}_s, \mathsf{k}_s), \mathsf{senc}(\mathsf{k}'_s, \mathsf{k}_s))$;
5) $\mathsf{Decrypt}_{\mathsf{a}}(\mathsf{h}(\mathsf{n}_s, \mathsf{priv}(\mathsf{s}_s)), \mathsf{aenc}(\mathsf{k}_s, \mathsf{pub}(\mathsf{s}_s)))$;
6) $\mathsf{Decrypt}_{\mathsf{s}}(\mathsf{h}(\mathsf{n}_s, \mathsf{k}_s), \mathsf{senc}(\mathsf{priv}(\mathsf{s}_s), \mathsf{k}_s))$.

where $\mathsf{k}_i, \mathsf{priv}(\mathsf{s}_i) \notin SK$ and $\mathsf{k}_s, \mathsf{k}'_s, \mathsf{priv}(\mathsf{s}_s) \in SK$.

Proof:

($\Rightarrow$) We have to prove that if σ is SK-secure then none of the above API calls happens in σ. We in fact prove that if one of the calls is in σ then σ is not SK-secure. It is enough to observe that wrapping a secure key under an insecure one allows the attacker to decrypt it (cf. Table II), and decrypting a secure key clearly reveals it as plaintext. In both cases σ is not SK-secure, from which the thesis follows.

($\Leftarrow$) We have to prove that if none of the above API calls happens in σ then σ is SK-secure. We proceed by contradiction: assume that $\sigma = q_0 \xrightarrow{\alpha} \!\!\!\!\rightarrow^* q_n$ is not SK-secure. Then, by Definition 2, $SK \cap q_n \neq \emptyset$. We let $\{\mathsf{K}_1, \ldots, \mathsf{K}_m\} = SK \cap q_n$. By Definition 1 we know that $\{\mathsf{K}_1, \ldots, \mathsf{K}_m\} \cap q_0 = \emptyset$. We thus consider the shortest prefix of σ: $q_0 \xrightarrow{\alpha} \!\!\!\!\rightarrow^* q_{k-1} \xrightarrow{a_k} \!\!\!\!\rightarrow q_k$ such that $\{\mathsf{K}_1, \ldots, \mathsf{K}_m\} \cap q_{k-1} = \emptyset$ and $\exists i \in 1, \ldots, m$. $\mathsf{K}_i \in q_k$. Intuitively, q_k is the first state that contains a sensitive key in the clear. Recall that K_i is either k or $\mathsf{priv}(\mathsf{k})$. We now consider all the possible API calls that might have returned K_i in the clear.

From Table I we only have $\mathsf{Decrypt}_{\mathsf{s}}$ and $\mathsf{Decrypt}_{\mathsf{a}}$. Consider $\mathsf{Decrypt}_{\mathsf{s}}$: it requires: $\mathsf{h}(\mathsf{n}', \mathsf{k}'), \mathsf{senc}(\mathsf{K}_i, \mathsf{k}') \in q_{k-1}$. Now if $\mathsf{k}' \in SK$ we are in case 4 or 6, while if $\mathsf{k}' \notin SK$ there must have been a previous API call corresponding to case 1 or 3. In fact, since the term $\mathsf{senc}(\mathsf{K}_i, \mathsf{k}')$ cannot be in q_0 (cf. Definition 1) and cannot come from a previous application of DY rules (cf. Table II) because we have assumed that K_i is being leaked now, the only other way to obtain it is by invocation of the Wrap_* API. A similar reasoning applies to $\mathsf{Decrypt}_{\mathsf{a}}$ for cases 5 and 2. We thus get a contradiction.

From Table II the key can only come from the decryption of $\mathsf{senc}(\mathsf{K}_i, \mathsf{k}')$ or $\mathsf{aenc}(\mathsf{K}_i, \mathsf{pub}(\mathsf{s}))$. Consider $\mathsf{senc}(\mathsf{K}_i, \mathsf{k}')$: since it cannot be $\mathsf{k}' \in SK$ (because k' would be known by the attacker contradicting the fact that no sensitive key is in q_{k-1}) we necessarily have that $\mathsf{k}' \notin SK$. As before we have that there must have been a previous API call corresponding to case 1 or 3. Following a similar reasoning, for $\mathsf{aenc}(\mathsf{K}_i, \mathsf{pub}(\mathsf{s}))$ we can conclude that there must have been a previous API call corresponding to case 2, which gives a contradiction. ∎

The above proposition gives a precise characterization of insecure executions and has important implications: first of all, it is enough to monitor wrap and decrypt API calls and rise an alert any time one of the above cases occur. When one of the above cases occur we are guaranteed that it is going to be an attack. Moreover, no attack will be missed since any attack requires one of the above API calls. Finally, the proposition

shows that the complexity of Dolev-Yao reasoning disappears since it is enough to just focus on single API calls. This is very convenient in order to apply the theory to the analysis of real logs.

B. Secure Distributed Executions

Even if an execution is secure locally, with respect to its set SK of sensitive keys, it might be the case that the execution is leaking keys coming from other devices, that are not monitored in SK. It could also be possible that keys in SK are sent to other devices and are leaked remotely. In order to find these distributed attacks we define a notion of security with respect to a set of executions, each with a set of local sensitive keys. Intuitively, we require that each execution is secure with respect to the union of the set of sensitive keys.

Since executions might contain freshly generated keys, from now on we will always consider appropriate alpha-conversion of executions so that freshly generated keys never collide. So, if a freshly generated key is in a local SK it won't appear in any other local SK from a different execution.

Definition 3 (Secure Distributed Executions)**.** *Let $\mathcal{S}$ be a set of distinct executions starting from their own initial states with their respective sets of sensitive keys $\{(SK_1, \sigma_1), \ldots, (SK_n, \sigma_n)\}$. Let $SK = \bigcup_{i=1,\ldots,n} SK_i$. We say that $\mathcal{S}$ is secure iff $\sigma_1, \ldots, \sigma_n$ are SK-secure.*

It is quite immediate to see that if a set of executions is secure, then each execution is locally secure. In fact, secure distributed executions require security with respect to a bigger set of keys. Interestingly, the other implication does not hold: it might be the case that secure, local executions become insecure when taken together. This confirms the intuition that there exist distributed attacks that cannot be detected locally. Thus, collecting local executions from different devices might reveal attacks that cannot be detected locally. We illustrate through a simple example.

Example 4 (Distributed Wrap-and-Decrypt Attack)**.** *Consider the following two executions σ and σ':*

$$\sigma = q_0 \xrightarrow{\mathsf{Wrap_{ss}}(\mathsf{h}(\mathsf{n}_1,\mathsf{k}_1),\mathsf{h}(\mathsf{n}_2,\mathsf{k}_2))} q_0 \cup \{\mathsf{senc}(\mathsf{k}_2,\mathsf{k}_1)\}$$

$$\sigma' = q_0' \xrightarrow{\mathsf{Decrypt_s}(\mathsf{h}(\mathsf{n}_1,\mathsf{k}_1),\mathsf{senc}(\mathsf{k}_2,\mathsf{k}_1))} q_0' \cup \{\mathsf{k}_2\}$$

Intuitively, σ and σ' represent two executions on different devices. In σ a sensitive key k_2 is wrapped under another sensitive key k_1, which is the standard key for security exporting a sensitive key. We have, in particular, that σ is $\{\mathsf{k}_1, \mathsf{k}_2\}$-secure. In σ' we suppose to have a device with just k_1 key, meaning that the administrator is only monitoring that single sensitive key.

An attacker might decrypt the ciphertext obtained from σ on the first device using the second device. This is what happens in σ': $\mathsf{senc}(\mathsf{k}_2, \mathsf{k}_1)$ is decrypted under k_1. The local administrator cannot notice the leakage of k_2 if such a key is unknown locally. In particular, we have that σ' is $\{\mathsf{k}_1\}$-secure.

In summary, we have two executions that are secure with respect to the local knowledge of sensitive keys. However, it is clear that the two executions represent a distributed wrap-and-decrypt attack in which a sensitive key from the first device (k_2) is leaked on the second device. This attack is captured by putting together the two executions: we let $\mathcal{S} = \{(\{\mathsf{k}_1, \mathsf{k}_2\}, \sigma), (\{\mathsf{k}_1\}, \sigma')\}$ and we obtain that $\mathcal{S}$ is not secure since σ' is not secure with respect to $\{\mathsf{k}_1, \mathsf{k}_2\}$. This can be seen by observing that $\{\mathsf{k}_1, \mathsf{k}_2\} \cap (q_0' \cup \{\mathsf{k}_2\}) \supseteq \{\mathsf{k}_2\} \neq \emptyset$.

The relation between local and distributed security is proved formally in the following proposition:

Proposition 2. *Let $\mathcal{S} = \{(SK_1, \sigma_1), \ldots, (SK_n, \sigma_n)\}$. Then:*

- *$\mathcal{S}$ secure $\Rightarrow$ σ_i SK_i-secure for each $i = 1, \ldots, n$;*
- *$\mathcal{S}$ secure $\not\Leftarrow$ σ_i SK_i-secure for each $i = 1, \ldots, n$.*

Proof:

($\Rightarrow$) By definition, $\mathcal{S}$ secure means that $\sigma_1, \ldots, \sigma_n$ are SK-secure, with $SK = \bigcup_{i=1,\ldots,n} SK_i$. Since $SK_i \subseteq SK$, by Lemma 1 we directly obtain that σ_i is SK_i-secure for each $i = 1, \ldots, n$.

($\not\Leftarrow$) The implication does not hold because of the existence of distributed attacks coming from locally secure executions, as shown in Example 4. ∎

IV. Analysis

We present a way to analyze executions offline. This allows for monitoring devices without necessarily being online, i.e., in between the application and the security hardware. We believe this is important to make the proposal realistic. In fact, in our experience, it would be hard to add an online element in the chain of a critical application based on secure hardware. The offline analysis can detect leakage of keys so that administrators can take suitable actions. Of course, if the solution works offline it is also possible to place it actively in the middle of the API calls, taking decision in real time, and preventing key leakage.

Logs can be taken locally and analyzed directly but, as we have shown in Example 4, attacks might happen across multiple devices, so it is crucial to consider the possibility of collecting local logs to look for distributed attacks.

It is important to notice that in order to monitor the above calls we need a way to distinguish secure keys from insecure ones and we need to track wrapped secure keys. We will discuss how this can be achieved in the next section. There are two important aspects to consider, in order to make the analysis effective and implementable: (i) the information that is tracked in the logs should not be too complex and should not grow too much, in order for the analysis to scale in space and time; (ii) the analysis should not require the whole execution logs in order to detect attacks, i.e., it should detect attacks even when logs represent partial executions.

It is important to point out that our model of distributed execution already detects attacks even when relevant API calls are missing, i.e., even when logs are partial. Thus, the requirement (ii) is implicit in the model we consider. As a consequence, any log analysis will necessarily have to fulfill (ii) in order to detect all the attacks. We illustrate this crucial point through an example:

Example 5 (Partial Executions)**.** *Consider a variant of Example 4 in which σ does not contain the wrap operation used by the attacker to mount the distributed wrap-and-decrypt attack. Execution σ could contain other API calls but, for simplicity, we just take it empty:*

$$\sigma = q_0$$
$$\sigma' = q_0' \xrightarrow{\mathsf{Decrypt_s}(\mathsf{h}(\mathsf{n}_1,\mathsf{k}_1),\,\mathsf{senc}(\mathsf{k}_2,\mathsf{k}_1))} q_1' \quad q_1' = q_0' \cup \{\mathsf{k}_2\}$$

The point here is that σ' is an attack to k_2 but there is no information in the logs about ciphertext $\mathsf{senc}(\mathsf{k}_2,\,\mathsf{k}_1)$.

The attack is nevertheless captured by the model. As before, in the first device we assume to have two sensitive keys, and we trivially have that σ is $\{\mathsf{k}_1,\mathsf{k}_2\}$-secure. However, $\{(\{\mathsf{k}_1,\mathsf{k}_2\},\sigma),(\{\mathsf{k}_1\},\sigma')\}$ is not secure since σ' is not secure with respect to $\{\mathsf{k}_1,\mathsf{k}_2\}$. Intuitively, since the set of sensitive keys is composed of all the sensitive keys from the various devices, attacks on a remote device will be naturally captured by the model that simply checks for the leakage of sensitive (possible remote) keys.

A. The Log Analysis Problem

We now state precisely the problem of log analysis. It is important to observe that, for obvious reasons, we cannot log the actual values of sensitive keys. The obvious replacement for key values are handles but this will introduce a major challenge: how to detect the leakage of a key value without knowing it.

Definition 4 (Log Analysis Problem)**.** *Let $\mathcal{S}$ be the distributed execution $\{(SK_1,\sigma_1),\ldots,(SK_n,\sigma_n)\}$.* Log analysis *is the problem of deciding whether or not $\mathcal{S}$ is secure given the following inputs:*

- *The executions $\bar{\sigma} = \sigma_1,\ldots,\sigma_n$, that we call logs;*
- *The handles H referring to sensitive keys occurring in $\sigma_1,\ldots,\sigma_n$ that belongs to $SK = \bigcup_{i=1,\ldots,n} SK_i$, i.e.,*

$$H = \{\mathsf{h}(\mathsf{n},\,\mathsf{k}) \mid \mathsf{h}(\mathsf{n},\,\mathsf{k}) \text{ occurs in } \bar{\sigma} \text{ and } \mathsf{k} \in SK\}$$

and under the following assumptions:

1) *terms can only be compared by syntactic equality;*
2) *offline encryption and decryption operations are possible only when the corresponding key is known (in a standard Dolev-Yao fashion).*

We now show that the log analysis problem is not solvable in general, because of the impossibility of linking a key value to its handle(s). Notice that this result holds because of the assumption that log analysis is done offline. If we have the possibility of interacting with the devices then keys could be distinguished by performing operations with them.

Proposition 3 (Unsolvability of Log Analysis Problem)**.** *The log analysis problem cannot be solved for all possible $\mathcal{S}$'s.*

Proof: We consider an instance of Example 5 in which $q_0 = \{\mathsf{h}(\mathsf{n}_1,\,\mathsf{k}_1),\mathsf{h}(\mathsf{n}_2,\,\mathsf{k}_2)\}$ and $q_0' = \{\mathsf{h}(\mathsf{n}_1',\,\mathsf{k}_1),\mathsf{senc}(\mathsf{k}_2,\,\mathsf{k}_1)\}$. Intuitively, the initial states q_0 and q_0' only contain the key handles and q_0' additionally contains the ciphertext that will be decrypted to leak k_2. The input to the log analysis problem is thus σ,σ' and $H = \{\mathsf{h}(\mathsf{n}_1,\,\mathsf{k}_1),\mathsf{h}(\mathsf{n}_2,\,\mathsf{k}_2),\mathsf{h}(\mathsf{n}_1',\,\mathsf{k}_1)\}$. The final state q_1' additionally contains the key value k_2. Now, there is clearly no way to link k_2 to the key handles in H, since we have assumed that terms can only be compared when they are identical (up to standard Dolev-Yao operations). Key k_2 could be used to encrypt other terms but since there is no ciphertext encrypted under k_2 the produced terms would never match any existing term. Intuitively, k_2 is leaked but it is not possible to detect offline whether or not it is a sensitive key pointed by one of the handles in H. ■

B. Log Analysis with Key Fingerprinting

In order to be able to solve the log analysis problem we need to log additional information that can be used offline to track keys. We consider an abstract key fingerprinting function whose value will be logged together with handles and that will allow to link a key value to a handle.

Definition 5 (Key Fingerprinting)**.** *A key fingerprinting is a deterministic one-way function. Formally, we note it as a special term $\mathsf{kf}(\mathsf{k})$ and we assume that $\mathsf{kf}(\mathsf{k})$ can be computed by anyone who knows k, while k cannot be computed from $\mathsf{kf}(\mathsf{k})$.*

We add a corresponding API call that allows for obtaining key fingerprints from their handles, and a corresponding Dolev-Yao rule for offline computation:

$$\mathsf{h}(\mathsf{x},\,\mathsf{y}) \xrightarrow{\mathsf{KeyFprint}} \mathsf{kf}(\mathsf{y}) \qquad \mathsf{y} \xrightarrow{\mathsf{DY}} \mathsf{kf}(\mathsf{y})$$

We can prove that key fingerprinting does not introduce new attacks. In particular, by adding the above rules we obtain the same characterization of Proposition 1.

Proposition 4. *Let $\sigma = q_0 \xrightarrow{\alpha}\!\!\!\!\!\rightarrow^{*} q_n$ be an execution possibly containing key fingerprint API calls and direct (Dolev-Yao) fingerprinting computations. Then, Proposition 1 holds.*

Proof: Proof is the same as the one of Proposition 1. In fact, fingerprinting does not add any new way of leaking a key in the clear, nor it can produce cryptographic terms. As a consequence, adding fingerprinting does not add any new case to the proof of Proposition 1. ■

Notice that the above proposition holds in our symbolic model because there is no notion of cost for the attack and, in general, we do not take into account cryptanalytic issues. It is important to observe that just using a standard one-way cryptographic hash to implement kf would provide the attacker a faster way to bruteforce cryptographic keys since hash functions are usually much faster than encryption algorithms. We will discuss possible implementations later on.

With key fingerprinting we can solve the log analysis problem efficiently. It is enough to assume that each sensitive key is fingerprinted in each local log. Intuitively, whenever we

Algorithm 1 Log Analysis using Key Fingerprinting.

```
1: procedure LogAnalysis(σ̄, H)
2:     FSK = [ ]                                  ▷ Initialize the list of fingerprints of sensitive keys as empty
3:     for (a, ret) ∈ σ̄ do                        ▷ Collect all the fingerprints of sensitive keys
4:         if a == KeyFprint(h) and h ∈ H then    ▷ If the API call is KeyFprint in sensitive handle
5:             FSK ← FSK + [ret]                  ▷ The actual fingerprint ret is added to FSK
6:         end if
7:     end for
8:     for (a, ret) ∈ σ̄ do                        ▷ Search for insecure wrap and decrypt
9:         if a == Wrap*(h1, h2) and h1 ∉ H and h2 ∈ H then   ▷ Insecure wrap of sensitive key
10:            return a                           ▷ The insecure wrap is returned: S is insecure
11:        end if
12:        if a == Decrypt*(h, t) and h ∈ H and kf(ret) ∈ FSK then   ▷ Decrypt of a sensitive key
13:            return a                           ▷ The insecure decrypt is returned: S is insecure
14:        end if
15:    end for
16:    return None                                ▷ No attack found: S is secure
17: end procedure
```

have a decrypt operation we test the leaked key against all the available fingerprints.

Our solution is coded as Algorithm 1: the algorithm first computes the set of fingerprints for sensitive keys (FSK) by looking for the actual calls to $\mathsf{KeyFprint}(\mathsf{h})$ where $\mathsf{h} \in H$ is a handle to sensitive keys.[1] This is done by the for loop from line 3 to line 7. The notation $(a, ret) \in \bar{\sigma}$ means that we loop over all possible API calls and a ranges over the actual call while ret ranges over the returned value. In fact, when we find a handle that belongs to H we add the returned value, i.e., the fingerprint, to the set FSK.

Then, the algorithm looks for attacks, in terms of the characterization of Proposition 4. In particular it searches for any $\mathsf{Wrap}_*(\mathsf{h}_1, \mathsf{h}_2)$ API call in which a sensitive key referred by h_2 is wrapped under a non-sensitive key referred by h_1. When this happens, the algorithm terminates and returns the call responsible for the attack. Similarly the algorithm looks for any $\mathsf{Decrypt}_*(\mathsf{h}, \mathsf{t})$ call that returns a key ret whose fingerprint $\mathsf{kf}(ret)$ belongs to the ones computed in the first phase (FSK). Again, when this situation is spotted, the responsible call is returned as a witness of the attack. If none of the above is found in all the executions the algorithm returns "None" to indicate that no attack has been found and that $\mathcal{S}$ is in fact secure.

Theorem 1 (Log Analysis through Key Fingerprinting)**.** *Let $\mathcal{S} = \{(SK_1, \sigma_1), \ldots, (SK_n, \sigma_n)\}$ and $SK = \bigcup_{i=1,\ldots,n} SK_i$, such that for each $\mathsf{K} \in SK$ we have that $\mathsf{kf}(\mathsf{K})$ occurs in $\bar{\sigma}$. Then, the log analysis problem can be solved in $O(|\bar{\sigma}| + |H|)$ steps.*

Proof: The algorithm correctly computes the set FSK of all fingerprints of sensitive keys because of the assumption that those fingerprints are all in the logs. Then, the correctness of solving log analysis directly derives from the characterization of Proposition 4. Both loops take, in the worst case, $|\bar{\sigma}|$ iterations while lookup in sets H and FSK can be done in constant time building appropriate hashtables, from which we get linear complexity. ■

Notice that, since Algorithm 1 only inspects a subset of the API calls, it is enough to just log those calls. This would greatly reduce the size of $|\bar{\sigma}|$ and, consequently, the execution time of the analysis.

C. Practical Considerations

Our approach requires the specification of which keys are considered sensitive. This decision must be definitely taken by the administrator, who is supposed to know what are the important cryptographic keys. It is worth noticing that key management APIs usually have a way to specify what keys should be regarded as sensitive, i.e., not accessible in the clear, so it is reasonable to assume that this property is going to be specified in some way.

Theorem 1 proves that log analysis can be solved efficiently when fingerprints for sensitive keys are available. Thus, in order to implement the proposed analysis, it is necessary that the relative fingerprint API calls are performed, in each local log. For long-term keys, it is reasonable to assume that key fingerprints will stabilize over time and could be reliably shared after an initial startup phase. For new keys that are freshly generated during the execution we can imagine that the logging system is instrumented so to ask for the key fingerprint of each new key, i.e., every time a KeyGen or KeyPairGen is invoked. We believe this is a mild, realistic assumption that does not significantly impact on the applicability of the methods. In fact, recall that without fingerprinting we know that log analysis is not even solvable (cf. Proposition 3). Additionally, a practical way to prevent that logs grow indefinitely is to delete part of them when there is a consensus that the knowledge on sensitive keys is synchronized, i.e., that no key occurring in the deleted logs will be discovered to be sensitive in the future.

[1] Notice that, for the sake of readability, we abbreviate handles as h.

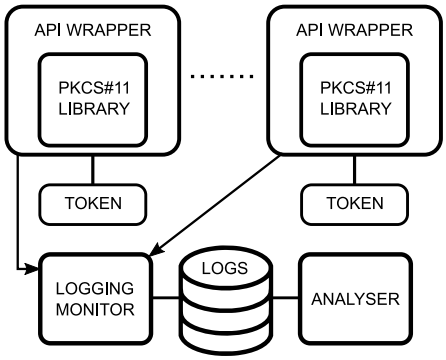

Fig. 1. Diagram of our log analysis system for PKCS#11

Another consideration regards fingerprints. One problem with fingerprints that is not captured by our symbolic model is the possible exploitation of fingerprints in cryptanalytic attacks. Using cryptographic hash functions, for example, would speed up key bruteforcing. Moreover, using a fixed function for all devices allows for precomputing fingerprints which, in turns, would reduce the key space to bruteforce. A reasonable alternative would be to digitally sign the fingerprint using a dedicated private key, different for each device. This would (i) slow down bruteforcing and (ii) prevent fingerprint precomputation. It is out of the scope of this paper to prove the security of practical implementations of key fingerprint and we leave this as a future work.

V. Prototype Implementation

To show the feasibility of our approach we discuss the implementation of a proof-of-concept log analysis tool for PKCS#11. The tool[2] is able to identify all the key-management attacks found in [12], [14] involving symmetric encryption operations. We plan to support the detection of attacks using asymmetric keys on PKCS#11 as a future work.

Our solution consists of three components, as outlined in Figure 1: (i) a software layer that wraps the existing PKCS#11 library interface. The wrapper allows the instrumentation of selected API calls to record the operations executed by the underlying library. It also computes key fingerprints to solve the log analysis problem; (ii) a logging facility to store the logs of each session in a central repository; (iii) the analyzer that parses the logs generated by the first two components and applies Algorithm 1 to discover attacks aimed at leaking the value of secure keys.

A. Fingerprint Computation

Before giving full details of the system, we introduce our fingerprint approach for PKCS#11. Fingerprint computation is indeed a challenging problem, given that it is not possible to access the value of a sensitive key directly. Additionally, since our solution does not extend the existing API with an ad-hoc fingerprint function, the fingerprint must be produced by the device reusing existing PKCS#11 functions.

Of course, a simple way to perform fingerprint computation would be to use the `C_DigestKey` API call. Given a key handle, this function returns a digest of the key value using a cryptographic hash function. According to the considerations made in Section IV, the use of a hash function would weaken the security of the system by decreasing the cost of a bruteforce attack to recover the key value. For this reason, we devised a different approach that allows to compute key fingerprints using keys as intended.

The functions allowed to be executed under a key are determined by the set of its attributes. For instance, a key with the `CKA_ENCRYPT` attribute enabled is allowed to perform encryption calls via `C_Encrypt`. Given a key k, we exploit the capabilities of each key to compute multiple fingerprints depending on the allowed operations. We devise three possible fingerprints for key k:

- if `CKA_ENCRYPT` is enabled, we pick a random value r and we encrypt it with k. We say that the *encryption fingerprint* of k, denoted by $\mathsf{kf}(\mathsf{k})_E$, is the pair $(\mathsf{r}, \mathsf{enc\_data})$ where $\mathsf{enc\_data}$ denotes r encrypted under k using the `C_Encrypt` function with a compatible mechanism;
- similarly, if `CKA_DECRYPT` is enabled, we let $\mathsf{dec\_data}$ be a random value r decrypted with k using the `C_Decrypt` function with a compatible mechanism. The *decryption fingerprint* of k, denoted by $\mathsf{kf}(\mathsf{k})_D$, is the pair $(\mathsf{r}, \mathsf{dec\_data})$;
- if k is a wrapping key, i.e., the `CKA_WRAP` attribute is enabled, we state that the *wrap fingerprint* of k, denoted by $\mathsf{kf}(\mathsf{k})_W$, is $\mathsf{wrap\_data}$, where $\mathsf{wrap\_data}$ is the result of wrapping the key with itself via a call to `C_WrapKey`.

If all the operations required to produce a fingerprint are forbidden, i.e., the attributes are disabled, we temporarily alter the `C_Encrypt` attribute to generate a valid encryption fingerprint.

The proposed fingerprint approach for PKCS#11 allows to precisely identify a key by performing an offline computation of the fingerprint, once the plain text value of the key is known. Moreover, the results of the operations are unique for each key, in practice, since the probability of obtaining the same result given two different keys is negligible. With respect to the practical considerations mentioned in the previous section regarding fingerprints, we claim that this solution does not speed up key bruteforcing and thus it does not decrease the security of fingerprinted keys. If one of the functions used to compute the fingerprint is allowed for a given key, then guessing the key from the encryption (decryption) of a random value would not be faster than doing the same with a value chosen by the attacker for which (s)he might have precomputed offline encryptions (decryptions) with a large number of keys.

B. Working Principles

We implement a wrapper of the full PKCS#11 API. However, it follows from Proposition 1 and Proposi-

[2]https://github.com/secgroup/p11d

tion 4 that monitoring only a small subset of PKCS#11 functions, i.e., `C_WrapKey` and `C_Decrypt`, is enough to detect all possible attacks. We also instrument the `C_GetAttributeValue` function that allows to directly read the value of a non-sensitive key. We do not assume the set of secure keys SK to be static during each execution, thus we need to track functions that allow the creation of new sensitive objects such as `C_GenerateKey`. For convenience, we also instrument `C_Login` to list long-term keys stored in the device and we initialize SK with sensitive keys found among them, even if in general we want to let each administrator to specify, locally, the set of sensitive keys that need to be monitored. For all the remaining functions, the wrapper transparently performs the corresponding call.

For each new key found during a `C_Login` call at the start of each session, or generated using `C_GenerateKey`, the wrapper computes the fingerprints and sends them to the logging facility along with the list of publicly-readable attributes and the object handle assigned to the key. Recall that handles are not guaranteed to be fixed for the lifetime of an object, still they allow to access the same object for the entire session duration [18]. The `C_WrapKey` function is instrumented to track the handles of the wrapping key and the wrapped key. `C_Decrypt` tracks the handle of the decryption key and the value of the decrypted data. Similarly, the `C_GetAttributeValue` tracks the handle of the actual key and the accessed value.

The logging component allows multiple sessions to be tracked in a centralized repository at the same time. Each log file produced during this step represents a single execution within a session. Since the `C_Login` function is called every time a new session is initialized by PKCS#11 applications, the first entry of every log file contains the list of long-term keys found in the device. As an example, Table III provides the textual representation of a possible log entry produced by a call to the login function. In this case only one key is found in the device. The listed key is sensitive and the handle that points to it in this session has value `0x00`. The key has the encrypt and decrypt attributes enabled, thus fingerprints are computed by encrypting and decrypting random values according to the method described before.

The analyzer then parses the collected logs and applies Algorithm 1. For each sensitive key found in the logs as a result of either a `C_Login` or `C_GenerateKey`, the component updates the set H with its handle paired with the identifier of the current session. In parallel, the analyzer stores the fingerprints of this key in FSK. When the set H and FSK are initialized, the program iterates over all the logged `C_WrapKey`, `C_Decrypt` and `C_GetAttributeValue` calls looking for attacks:

- insecure wraps of secure keys are easily detected by checking if the pairs in the form $(\mathsf{session}, \mathsf{h})$ of the wrapping key and wrapped key handles belong to H. If a secure key is wrapped under an insecure one, the operation is marked as an attack and the analysis terminates;

```
["C_Login", [
  {"extractable": 0x00, "decrypt": 0x01,
   "sensitive": 0x01, "encrypt": 0x01,
   "wrap": 0x00, "unwrap": 0x00,
   "label": 0x4d7950726563696f7573,
   "keytype": 0x1300000000000000,
   "handle": 0x01,
   "fingerprint": {
     "decrypt": [
       0x96ccb41274a8adbf, 0x0c2e551a66cb4d86
     ],
     "encrypt": [
       0xd387b0b818a52d2a, 0xea1b3c934ed860f5
     ]
   }
]]
```

TABLE III
LOG ENTRY FOR THE `C_LOGIN` CALL

- decryptions of secure keys are identified in two steps. The analyzer first checks if the handle used in the decryption operation points to a secure key. In this case, it tests the decrypted data ret found in the log entry of `C_Decrypt` against all the fingerprints in FSK, by simulating calls to `C_Decrypt` and `C_Encrypt`. To perform the comparison with a wrap fingerprint, the application encrypts ret under itself and checks if the result matches the wrapped data in the fingerprint. Otherwise, if the fingerprint is in the form $(\mathsf{r}, \mathsf{data})$, depending on the type of the fingerprint, the tool executes an encryption or a decryption of the random value r under the key ret and compares the result with data. If no match is found after iterating the process over all the fingerprints in FKS, the operation is considered safe, otherwise it is marked as an attack and the analysis stops;
- direct accesses to the value of a non-sensitive key via a `C_GetAttributeValue` are threat of practical importance if the attacker manages somehow to alter the `CKA_SENSITIVE` attribute of a secure key. These attacks are easily detected by the analyzer by testing the value returned by the API call against all the fingerprints in FSK, as in the previous case.

C. Experimental Tests

We now show how our solution is effective against a range of key-management attacks, also in a distributed setting. All the attacks reported in this section, as well as others from the literature, can be simulated using our tool and a software token provided by openCryptoki [19]. Unless stated otherwise, in the following examples we denote by h_i the handle pointing to a key k_i.

Example 6 (Wrap and Decrypt Attack)**.** *We discuss how the wrap/decrypt attack outlined in Example 3 is detected. In this simulation, the attacker calls `C_GenerateKey` to generate a non-sensitive key k_2 with the `CKA_WRAP` and `CKA_DECRYPT` attributes enabled. Using this key, (s)he leaks the value of*

```
$ LD_PRELOAD=./p11d.so ./attack 0 12345
[I] Found 1 key(s)
[A] Wrap-Decrypt attempt
[*] Generate a key k2 for wrapping and
    decrypting
[*] Wrap the sensitive key k1 with the key k2
[*] Decrypt the wrapped key k1 with the key k2
[*] Recovering k1 value: "0102030405060708"

$ ./analyzer.py
[*] Computing H and FSK
[*] Searching for insecure Wrap and Decrypt
    operations
[!] Attack detected in session-1000.log
    The sensitive key h1 has been wrapped with
    the insecure key h2
```

TABLE IV
WRAP/DECRYPT ATTACK DETECTION

the long-term sensitive key k_1 *by wrapping* k_1 *under* k_2 *and decrypting the result again with* k_2*. As pointed out in Table IV, the attack is detected by our tool on the* `C_WrapKey` *operation since the attacker is wrapping a secure key pointed by* h_1 *with an insecure one pointed by* h_2*.*

Example 7 (Re-import Attack)**.** *In our implementation of the re-import attack, the attacker executes* `C_GenerateKey` *to generate a key* k_2 *with the* `CKA_UNWRAP` *attribute set. (S)he then unwraps a random value* r *with this key to create a new key* k_3 *in the device with the* `CKA_WRAP` *attribute set. Notice that* k_3 *is the decryption of* r *under* k_2*. The value* r *is unwrapped again using* k_2 *to re-import* k_3*, this time with the* `CKA_DECRYPT` *attribute set. We let* h_3 *and* h_4 *be the handles returned by the first and the second unwrap, respectively. Now, to leak the sensitive key* k_1*, the attacker wraps* k_1 *under* k_3 *pointed by* h_3 *and decrypts the wrapped key with* k_3 *pointed by* h_4*. As shown in Table V, our tool detects the attack on the* `C_WrapKey` *operation since* h_1 *points to a secure key, while* h_3 *does not.*

Example 8 (Wrap and Unwrap Attack)**.** *We assume the existence of a long-term sensitive key* k_1 *in the device. The key has the attributes* `CKA_WRAP` *and* `CKA_UNWRAP` *enabled. The attack consists in wrapping* k_1 *with itself and reimporting the key as a non-sensitive one under a new handle* h_2*. Then, by using the* `C_GetAttributeValue` *on* h_2*, the attacker can directly read the value of* k_1*. Our tool is able to detect the attack by testing the plain value of* k_1 *against the fingerprints in FSK. The attack trace and the log analysis performed by the tool are provided in Table VI.*

Example 9 (Distributed Wrap and Decrypt Attack)**.** *The last attack we discuss is wrap/decrypt in the distributed setting, as presented in Example 4. We assume two long-term sensitive keys* k_1 *and* k_2 *in the first device. We also assume* k_2 *to be found in a second device. The key* k_2 *has, at least, the attribute* `CKA_WRAP` *enabled in the first device and the attribute* `CKA_DECRYPT` *enabled in the second one. The attacker connects to the first device and wraps* k_1 *under* k_2 *and (s)he saves the wrapped data. Then, (s)he connects to the second device and decrypts the wrapped data with* k_2 *to access the value of* k_1*. When both* k_1 *and* k_2 *are secure keys, the* `C_WrapKey` *operation is not detected by our tool as an attack since we are wrapping a secure key with another secure key. Nevertheless, the* `C_Decrypt` *call returns the value of the secure key* k_1 *and thus allows our tool to match* k_1 *against fingerprints in FSK, revealing that an attack occurred. See Table VII for the detailed execution and the attack detection analysis.*

```
$ LD_PRELOAD=./p11d.so ./attack 0 12345
[I] Found 1 key(s)
[A] Re-import attempt
[*] Generate a key k2 for unwrapping
[*] Unwrap a random bytestream with k2 to
    import a new key k3 pointed by h3 that
    can wrap
[*] Unwrap a random bytestream with k2 to
    import a new key k3 pointed by h4 that
    can decrypt
[*] Wrap the sensitive key k1 with h3
[*] Decrypt the wrapped key k1 with h4
[*] Recovering k1 value: "0102030405060708"

$ ./analyzer.py
[*] Computing H and FSK
[*] Searching for insecure Wrap and Decrypt
    operations
[!] Attack detected in session-2000.log
    The sensitive key h1 has been wrapped with
    the insecure key h3
```

TABLE V
RE-IMPORT ATTACK DETECTION

```
$ LD_PRELOAD=./p11d.so ./attack 0 12345
[I] Found 1 key(s)
[A] Wrap-Unwrap attempt
[*] Wrap k1 with k1
[*] Re-import k1 as non-sensitive
[*] Recovering k1 value: "a1a2a3a4a5a6a7a8"

$ ./analyzer.py
[*] Computing H and FSK
[*] Searching for insecure Wrap and Decrypt
    operations
[!] Attack detected in session-3000.log
    The plaintext value of a sensitive key
    has been directly read
```

TABLE VI
WRAP/UNWRAP ATTACK DETECTION

VI. CONCLUSION

Attacks on cryptographic APIs are notoriously hard to detect and fix. Even simple key management operations may be subject to API level vulnerabilities that leak cryptographic keys in the clear. For example, an attacker can wrap a secure key under another secure key and then ask the device to decrypt

```
$ LD_PRELOAD=./p11d.so ./attack 0 12345
[I] Found 2 key(s)
[A] Distributed Wrap-Decrypt attempt (1)
[*] Wrap the sensitive key k1 with the
    sensitive key k2
[*] Wrapped data: "d6c22bb28cd93ec0"

$ LD_PRELOAD=./p11d.so ./attack 1 12345
[I] Found 1 key(s)
[A] Distributed Wrap-Decrypt attempt (2)
[*] Decrypt wrapped data "d6c22bb28cd93ec0"
    with the key k2
[*] Recovering k1 value: "0102030405060708"

$ ./analyzer.py
[*] Computing H and FSK
[*] Searching for insecure Wrap and Decrypt
    operations
[!] Attack detected in session-4001.log
    The plaintext value of a sensitive key
    has been leaked after decryption with
    key h1
```

TABLE VII
DISTRIBUTED WRAP/DECRYPT ATTACK DETECTION

the ciphertext, obtaining the former key in the clear. In the literature we find many proposals for preventing or mitigating this kind of attacks but they typically require to modify the API or to configure it in a way that might break existing applications. This makes it very hard to adopt these proposals for critical applications and infrastructures, where systems are rarely modified and legacy applications are very common. At the same time, in these critical settings, the leakage of a cryptographic key can cause serious consequences.

In this paper we have investigated a new method to analyze cryptographic API logs. Log analysis is interesting because it has a very low impact on existing systems and is frequently used in industrial systems, financial applications and critical infrastructures. Log analysis of cryptographic APIs is challenging since keys are never supposed to be leaked in the clear, meaning that tracking different keys might become hard, especially if we want to analyze logs offline without interacting with the cryptographic devices.

More specifically, we have extended an existing model for security API analysis in order to model API logs. We have given a formal definition of secure execution that scales to a distribute setting, in which logs from services and devices that are placed in different physical locations, can be collected and searched for distributed attack sequences. We have shown examples of distributed attacks that cannot be detected locally and we have proved that the problem of detecting these attacks offline is unsolvable, because of the impossibility of tracking keys. We have shown that by adding a simple API for key fingerprinting, log analysis becomes feasible and efficient. We actually proved that security can be characterized in term of absence of particular combination of parameters in a subset of the API calls, i.e., Wrap and Decrypt.

Finally, we have implemented a tool for PKCS#11 APIs that simulates key fingerprinting through the available cryptographic operations for a given key, and can detect all documented attacks on PKCS#11 that directly leak a key in the clear. The tool constitutes a proof-of-concept that the method is effective and that can be implemented even without a dedicated key fingerprinting API. It is worth noticing, that adding a key fingerprinting API for logging purposes would not affect existing applications. Compared to previous works, our approach does not require existing API functions to be modified, therefore legacy applications do not need to be updated. Instead, we propose to add a new fingerprinting function that is solely used by the monitoring solution to provide more informative logs. In this respect, extending existing devices with this new mechanism seems a realistic possibility and we would encourage producer to consider this idea for next generation devices.

As a future work, we intend to extend our tool to cover a more extensive fragment of PKCS#11 and we want to experiment with candidate key fingerprinting APIs on software emulators of PKCS#11. We also intend to characterize other cryptographic APIs by studying formally which rules are considered problematic and should be tracked in the logs. Intuitively, the problematic rules are the ones that either directly leak a key in the clear, or generate a term containing a sensitive key that can be deconstructed by the DY attacker, as when wrapping a sensitive key under a nonsensitive one. Lastly, we plan to cover cryptanalytic attacks related to weak cryptographic mechanisms and side channels.

Acknowledgments: This work has been partially supported by CINI Cybersecurity National Laboratory within the project FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks (www.filierasicura.it) funded by CISCO Systems Inc. and Leonardo SpA.

REFERENCES

[1] Pedro Adão, Riccardo Focardi, and Flaminia L. Luccio. Type-based analysis of generic key management apis. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, pages 97–111, 2013.

[2] R. Anderson. The correctness of crypto transaction sets. In *8th International Workshop on Security Protocols*, April 2000. http://www.cl.cam.ac.uk/ftp/users/rja14/protocols00.pdf.

[3] Ryad Benadjila, Thomas Calderon, and Marion Daubignard. Caml Crush: A PKCS#11 Filtering Proxy. In *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, pages 173–192, 2014.

[4] M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 220–234, Paris, France, 2001. Springer.

[5] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 260–269, Chicago, Illinois, USA, October 2010. ACM Press.

[6] M. Centenaro, R. Focardi, F.L. Luccio, and G. Steel. Type-based analysis of PIN processing APIs. In Springer LNCS vol. 5789/2009, editor, *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS 09)*, pages 53–68, 2009.

[7] Matteo Centenaro, Riccardo Focardi, and Flaminia L. Luccio. Type-based analysis of key management in PKCS#11 cryptographic devices. *Journal of Computer Security*, 21(6):971–1007, 2013.

[8] R. Clayton and M. Bond. Experience using a low-cost FPGA design to crack DES keys. In *Cryptographic Hardware and Embedded System - CHES 2002*, pages 579–592, 2002.

[9] J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.

[10] J. Clulow. On the security of PKCS#11. In *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, volume 2779 of *LNCS*, pages 411–425. Springer, 2003.

[11] V. Cortier and G. Steel. A generic security API for symmetric key management on cryptographic devices. In Michael Backes and Peng Ning, editors, *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, volume 5789 of *Lecture Notes in Computer Science*, pages 605–620, Saint Malo, France, September 2009. Springer.

[12] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, November 2010.

[13] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions in Information Theory*, 2(29):198–208, March 1983.

[14] R. Focardi, F.L. Luccio, and G. Steel. An introduction to security api analysis. In *FOSAD*, pages 35–65, 2010.

[15] S. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In Pierpaolo Degano and Luca Viganò, editors, *Revised Selected Papers of the Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, volume 5511 of *Lecture Notes in Computer Science*, pages 92–106, York, UK, August 2009. Springer.

[16] Steve Kremer, Robert Künnemann, and Graham Steel. Universally composable key-management. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pages 327–344, 2013.

[17] Steve Kremer, Graham Steel, and Bogdan Warinschi. Security for key management interfaces. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 266–280, 2011.

[18] OASIS Standard. *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40*, April 2015. http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html.

[19] openCryptoki. http://sourceforge.net/projects/opencryptoki/.

[20] RSA Laboratories. *PKCS #11 v2.30: Cryptographic Token Interface Standard*, April 2009. http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm.

[21] Guillaume Scerri and Ryan Stanley-Oakes. Analysis of key wrapping apis: Generic policies, computational security. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 281–295, 2016.

[22] P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, August 2005.