Security for Key Management Interfaces

Steve Kremer, Graham Steel, Bogdan Warinschi

Security for Key Management Interfaces

Steve Kremer Graham Steel
LSV, ENS Cachan & CNRS & INRIA
France

Bogdan Warinschi
University of Bristol
UK

***Abstract*—We propose a much-needed formal definition of security for cryptographic key management APIs. The advantages of our definition are that it is general, intuitive, and applicable to security proofs in both symbolic and computational models of cryptography. Our definition relies on an idealized API which allows only the most essential functions for generating, exporting and importing keys, and takes into account dynamic corruption of keys. Based on this we can define the security of more expressive APIs which support richer functionality. We illustrate our approach by showing the security of APIs both in symbolic and computational models.**

***Keywords*-Key management, security APIs, cryptography**

I. Introduction

Cryptographic key management, i.e. the secure creation, storage, backup, use and destruction of keys has long been identified as a major challenge in applied cryptography. In real-world applications, key management often involves the use of hardware security modules (HSMs) or cryptographic tokens, since these are considered easier to secure than commodity hardware, and indeed are mandated by standards in certain sectors [1]. There is also a growing trend towards enterprise-wide schemes based around key management servers offering cryptographic services over open networks [2]. All these solutions aim to enforce security by dividing the system into trusted parts (HSM, server) and untrusted parts (host computer, the rest of the network). The trusted part makes cryptographic functions available via an application program interface (API). This API has been identified as a security critical design point: in general one must assume that the untrusted host machines might execute malicious code, so the API must be designed to maintain its security policy no matter what sequence of API commands are called. Designing such an API is extremely tricky, and in the last decade serious flaws have been found in the APIs of HSMs [3]–[6] and authentication tokens and smartcards [7].

Recently, two academic papers have proposed new designs for secure token interfaces with security proofs [8], [9]. However, neither of these provide a satisfactory solution to the problem. We explain why in more detail in Section II, but in summary, one requires that only a single central key server be used by an entire organisation, since security depends on an up to date and accurate log of all operations [9], while the other is intended for distributed tokens, providing more limited functionality and security proofs only in the symbolic model [8]. Furthermore, both papers give their own models and notions of security and it is not clear how to compare the two, making it difficult to combine ideas to come up with a more generally applicable secure API.

In this paper, we set out to improve the situation by giving a general security model for cryptographic key management APIs. We exemplify our results for the case when such keys are used for encryption and decryption. The advantages of our approach are:

- The model that we propose is abstract. For example, we do not make assumptions about the global state to be stored by the API, giving only an abstract notion of state.
- Our model is flexible. It can be tuned to account for many possible different configurations used by practical APIs, which we exemplify in Sections IV and V.
- Our model is uniform. The same definitional ideas can be applied to both symbolic and computational models, as we demonstrate in our proofs.
- Our model strengthens existing ones which are clearly insufficient. In particular we consider dynamic corruption of keys, which neither of the two previous models account for.

The paper is organised as follows. In the next section (§II) we give some background on key management and security APIs. We then give an abstract model of a key management API together with a notion of security (§III). We define a symbolic model for APIs and exemplify the verification of the security of an API in the model with respect to our definition (§IV). We then do the same in the standard computational model for security (§V), in particular treating an implementation

using a deterministic key wrap primitive preferred by practitioners to randomized equivalents (§VI). Finally we discuss conclusions and further work (§VII).

II. Key Management APIs

The functionality provided by cryptographic APIs can broadly speaking be divided into key management and key usage. Key management typically involves generating and deleting keys, and importing and exporting them in a secure way, usually by encrypting them under other keys (an operation known as key wrapping). In key usage, we permit e.g. the encryption, decryption, signing and verification of data using the keys depending on some policy. Typically, every key under management will have its own usage policy, described by some metadata or *key attributes* stored with the key.

From this description we can already make some observations about the desirable characteristics of a secure API. First, it should not allow key usage operations to interfere with key management operations. Unfortunately, this is precisely what happens in the widely used industry standard interface RSA PKCS#11, leading to a variety of attacks [7], [10]. Second, if a key is wrapped, the correct key attributes should be cryptographically bound to the key so that when it is unwrapped, perhaps on a different device, the correct usage rules are adopted. Unfortunately, this was not the case in for example the IBM CCA interface, where the use of XOR to bind usage rules to keys allowed attacks on re-import [4]. Third, cryptography should be used following modern principles of provable security, and not, for example, in a way that allows meet in the middle attacks to be mounted on the device as was the case in the CCA [5]. More generally, we would like to have a sound theory allowing us to reason about what is and is not a secure API.

Efforts have already been made to automate the security analysis of APIs [11]–[14]. This has lead to some major successes, such as the discovery of new attacks, but this work suffers from two major limitations: the first is that all use a 'symbolic' or 'Dolev Yao' model of cryptography, where bitstrings are represented as terms in an abstract algebra and cryptographic operations as functions on those terms. We have seen that some existing attacks exploit vulnerabilities not captured by these models, so security proofs in these models do not assure their absence. The second is that there is no established notion of security for a cryptographic key management API, so it is hard to evaluate the significance of a security proof.

Recently, two articles have been published that set out to address some of these shortcomings. The first, by Cachin and Chandran, gives a design for a key management server together with a proof of security in the cryptographic model [9]. However, the security notion is tightly coupled to the design of the API, where security rests on a single log of operations which is used to decide which API calls should be permitted, preventing the design from being used in realistic applications where several distributed servers and other devices with limited memory may be used (the authors acknowledge this drawback [9, §7]). The security proof is also a little unsatisfactory: it is not clear what security policy has actually been proved. For example, their security game does not allow a wrapped key to be re-imported onto the device, which means that a design which fails to securely bind attributes to wrapped keys would still be proved secure. Additionally, their notion of security does not allow for corruption of keys, which must be considered a realistic possibility in an enterprise-scale solution. It does consider the legitimate reading of keys in clear by users, but this is a rather softer problem, since the API is aware of which keys have been read and so can adjust accordingly. Finally, they require the use of probabilistic encryption schemes for key wrap. Ideally one would like to avoid insisting on this: as Rogaway and Shrimpton remark, "practitioners have already 'voted' for [deterministic] key-wrap by way of protocol-design and standardization efforts, and it is simply not productive to say 'use a conventional AE scheme' after this option has been rejected." [15, p. 3].

The second article by Cortier and Steel proposes a very different API, designed for use on distributed tokens that contain very little state, together with a proof of security, but only in the symbolic model [8]. The usage policy for keys must be declared at generation time, which is not always convenient for applications: it is not trivial to see how, for example, to add a new user to the system. The proofs do deal with (static) corruption of keys, but cryptographic details are not considered. As we will discuss in Section V, there are tricky problems to solve at this level of detail: the specification of the security requirements for the wrap algorithm, for example.

It is not clear how to compare the security properties proven in these two works. It seems clear that a practical solution must involve a variety of devices, from key servers with a large storage capacity to cryptographic tokens with very little, and these devices will of course have different APIs. However, without a uniform notion of security, it seems impossible to combine ideas from these designs or others, or to compare solutions. At the same time, efforts to produce new industry standards

[16], [17], and indeed patents [18], [19], for such interfaces are proceeding apace. It is therefore a timely moment to investigate foundations for the study of the security of this problem.

III. IDEALIZED APIS

We describe APIs as state transition systems. As explained before, we consider three different settings that share significant commonalities. We start by presenting idealized APIs, which will be used to define the security notion in the symbolic and computational settings. Idealized executions are concerned strictly with the key management aspects of the API and are relevant for determining which keys can be learned by an adversary. Real APIs will of course allow more functionality, but our security definition requires that these additional functionalities do not interfere with the key management and do not compromise the security of the keys.

The idealized API allows applications to generate a new key, wrap a key under another key, and unwrap an encrypted key. We assume the API stores keys such that they cannot *a priori* be read by the calling program. To allow the application to refer to particular keys in function calls, each key stored is assigned a *handle*, which can be thought of intuitively as a name for or pointer to the key in secure memory. In order to, e.g. wrap a key under another, the calling program supplies the handles for the two keys. In addition to these three commands, we model explicitly, by the means of corruption queries, the possibility that the adversary may, through cryptanalysis or some other means, learn the key associated with a certain handle.

We will now formally define the idealized API. An idealized API is parametrized by 2 sets:

- Wraps: an abstract set of wraps;
- Handles: an abstract set of handles.

We refer to this API as API(Wraps, Handles).
A state of API(Wraps, Handles) is defined as a 5-tuple $\langle C, W, H, wr, \equiv \rangle$ where

- $C \subseteq$ Handles is the set of *insecure* handles;
- $W \subseteq$ Wraps is the set of wraps that have been computed by the API so far;
- $H \subseteq$ Handles is the set of current handles;
- $wr : W \to H \times H$ is a function that given a wrap returns the handle that was used for the wrapping key and the handle that was used for the payload or wrapped key;
- $\equiv \subseteq H \times H$ is an equivalence relation indicating which handles are equivalent. Intuitively, these are handles that point to the same key.

In order to define the security of a handle we need to take into account the fact that some handles have been explicitly corrupted, some other handles may be wrapped under a corrupted handle and some handles may be equivalent, in the sense that they refer to the same key. We therefore define a closure operation which reflects all the different ways for a handle to become insecure.

Definition 1: Given a set of handles $C \subseteq$ Handles, a partial function wr : Wraps $\to$ Handles $\times$ Handles and an equivalence relation $\equiv \subseteq$ Handles $\times$ Handles we define insecure$(C, wr, \equiv)$ to be the smallest set such that

- $C \subseteq$ insecure$(C, wr, \equiv)$;
- if $h \in$ insecure$(C, wr, \equiv)$ and $h \equiv h'$ then $h' \in$ insecure$(C, wr, \equiv)$;
- if $w \in dom(wr)$, $wr(w) = \langle h_1, h_2 \rangle$ and $h_1 \in$ insecure$(C, wr, \equiv)$ then $h_2 \in$ insecure$(C, wr, \equiv)$.

The idealized API allows four operations which are defined by the following transitions between states.

- $\langle C, W, H, wr, \equiv \rangle \xrightarrow{\mathsf{corrupt}(h)} \langle C', W, H, wr, \equiv \rangle$ if $h \in H$ and $C' =$ insecure$(C \cup \{h\}, wr, \equiv)$.
- $\langle C, W, H, wr, \equiv \rangle \xrightarrow{\mathsf{new}(h)} \langle C, W, H \cup \{h\}, wr, \equiv \rangle$ if $h \in$ Handles $\setminus H$;
- $\langle C, W, H, wr, \equiv \rangle \xrightarrow{\mathsf{wrap}(h_1, h_2, w)} \langle C', W \cup \{w\}, H, wr', \equiv \rangle$ if
  - $h_1, h_2 \in H$,
  - either $wr(w) = \langle h_1, h_2 \rangle$
  - or $\forall w' \in W.\ wr(w') \neq \langle h_1, h_2 \rangle$ and $w \in$ Wraps $\setminus W$,
  - $$wr'(x) = \begin{cases} \langle h_1, h_2 \rangle & \text{if } x = w \\ wr(x) & \text{if } x \in W \end{cases}$$
  - $C' =$ insecure$(C, wr', \equiv)$
- $\langle C, W, H, wr, \equiv \rangle \xrightarrow{\mathsf{unwrap}(h, w, h')} \langle C', W, H \cup \{h'\}, wr, \equiv' \rangle$ if
  - $h \in H$, $h' \in$ Handles $\setminus H$ and
  - either $w \in W$, $wr(w) = \langle h_1, h_2 \rangle$, $h_1 \equiv h$, $\equiv'$ is the equivalence relation induced by $\equiv \cup \{\langle h', h_2 \rangle\}$ and $C' =$ insecure$(C, wr, \equiv')$
  - or $w \notin W, h \in C, \equiv' = \equiv, C' =$ insecure$(C \cup \{h'\}, wr, \equiv)$.

We can now formally define what it means for a handle to be insecure in a state s.

Definition 2 (insecure handles): Given API(Wraps, Handles) and a state $s = \langle C, W, H, wr, \equiv \rangle$. We say that a handle h is *insecure in state s* iff $h \in C$ and we say that h is *secure in state s*, otherwise.

Remark 1: An adversary may forge a valid wrap which was not generated by the device by using insecure keys. However, such a wrap will use an insecure

wrapping handle and a payload key pointed to by an insecure handle. Hence, using the unwrap command the adversary may introduce insecure handles (second case in the unwrap command) for which the $\equiv$ relation may not be updated. Nevertheless, the $\equiv$ relation remains correct on all *secure* handles, which is all we need for our purposes.

Definition 3 (Idealized adversary): Given $(\mathsf{Wraps}, \mathsf{Handles})$ a valid idealized adversary is a sequence of queries $q_1, \ldots, q_n$ such that $\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{q_1} s_1 \ldots \xrightarrow{q_n} s_n$ for $\mathsf{API}(\mathsf{Wraps}, \mathsf{Handles})$. A probabilistic idealized adversary is simply a distribution on valid idealized adversaries.

Having defined an idealized adversary and a notion of insecure handles, we will show in the next sections how this gives rise to a natural notion of security that can be used for proofs in the symbolic and computational models.

IV. Symbolic Model

In this section we define a symbolic model for APIs where messages are represented using a term algebra, together with a notion of security based on our definitions in the idealized model. We will give an example of a simple key management API and prove it secure in our model. We will comment on the relation to other APIs in the literature.

The term algebra will be built over a set of function symbols $\mathcal{F}$ and a set of variables $\mathcal{X}$. We suppose that the set $\mathcal{F}$ comes with an arity function $\mathsf{ar} : \mathcal{F} \to \mathbb{N}$. Function symbols of arity 0 are called constants. The set of terms that can be built over function symbols $\mathcal{F}' \subseteq \mathcal{F}$ and $\mathcal{X}' \subseteq \mathcal{X}$ is denoted $\mathcal{T}(\mathcal{F}', \mathcal{X}')$ and defined to be the smallest set such that $\mathcal{X}' \subseteq \mathcal{T}(\mathcal{F}', \mathcal{X}')$ and $f(t_1, \ldots, t_{\mathsf{ar}(f)}) \in \mathcal{T}(\mathcal{F}', \mathcal{X}')$ whenever $t_1, \ldots, t_{\mathsf{ar}(f)} \in \mathcal{T}(\mathcal{F}', \mathcal{X}')$. The set of closed terms over $\mathcal{F}' \subseteq \mathcal{F}$ is defined as $\mathcal{T}(\mathcal{F}', \emptyset)$ and denoted $\mathcal{T}(\mathcal{F}')$. Given a term t we write $st(t)$ for the set of subterms of t, defined as usual and $fv(t)$ for the (free) variables of t. These notations are also defined for sets of terms and formulas over terms as expected.

Informally, we define a general way of expressing APIs as sets of guarded rules, with an abstract notion of checking the state to see if a guard may be fired and updating the state when the rule goes through. Well-known symbolic formalisms that have been used for API modelling such as security protocol languages based on set rewriting can be expressed as special cases of our definition. We also define a function from symbolic handles to symbolic keys:

Definition 4: A symbolic API is a 7-tuple $(S, s_0, \Phi, \models, \mathcal{R}, \vdash, \mathsf{key})$ where

- S is a set of states;
- $s_0 \in S$ is the initial state;
- $\vdash \subseteq S \times \mathcal{T}(\mathcal{F})$ is the deduction relation which checks whether a given closed term can be deduced by the adversary in a given state.
- Φ is a set of guards, which are formulas built over the terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We denote by Φ^c the set of closed formulas that contain no free variables;
- $\models \subseteq S \times \Phi^c$ is a satisfaction relation which checks whether a closed guard is satisfied in a given state;
- $\mathcal{R}$ is a set of rules where each rule $r \in \mathcal{R}$ is a triple $(\varphi_r, \ell_r(\tilde{t}), \mathsf{upd}_r)$ and
  - $\varphi_r \in \Phi$ is a guard,
  - ℓ_r is a unique label and $\tilde{t} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$ is such that $fv(\tilde{t}) \subseteq fv(\varphi_r)$,
  - $\mathsf{upd}_r \subseteq S \times \Theta \times S$ is the update relation where Θ is the set of all substitutions from $fv(\varphi_r)$ to $\mathcal{T}(\mathcal{F})$.
- $\mathsf{key} : S \times \mathcal{T}(\mathcal{F}) \to (\mathcal{T}(\mathcal{F}) \cup \bot)$ maps a handle to its key in a given state. When applied to a term which is not a handle in this state, the function returns the special symbol $\bot$.

The set of rules $\mathcal{R}$ must at least contain rules $r_{\mathsf{corrupt}}, r_{\mathsf{new}}, r_{\mathsf{wrap}}, r_{\mathsf{unwrap}}$ such that these rules have labels $\ell_{\mathsf{corrupt}} = \mathsf{corrupt}(t, \tilde{u}), \ell_{\mathsf{new}} = \mathsf{new}(t, \tilde{u}), \ell_{\mathsf{wrap}} = \mathsf{wrap}(t_1, t_2, t_3, \tilde{u}), \ell_{\mathsf{unwrap}} = \mathsf{unwrap}(t_1, t_2, t_3, \tilde{u})$ where $\tilde{u}$ is a (possibly empty) sequence of terms.

A symbolic API induces a transition relation on states such that $s \xrightarrow{\ell} s'$ if and only if there exists a rule $(\varphi_r, \ell_r(\tilde{t}), \mathsf{upd}_r) \in \mathcal{R}$ and a substitution θ from $fv(\varphi_r)$ to $\mathcal{T}(\mathcal{F})$ such that

- $s \models \varphi_r\theta$,
- $\ell = \ell_r(\tilde{t})\theta$,
- $\mathsf{upd}_r(s, \theta, s')$ holds.

The notion of an insecure handle in our model corresponds to handles for which the key value is deducible:

Definition 5: Given a symbolic API $(S, s_0, \Phi, \models, \mathcal{R}, \vdash, \mathsf{key})$ and a state $s \in S$ we say that a handle h is insecure in s iff $\mathsf{key}(s, h) \neq \bot$ and $s \vdash \mathsf{key}(s, h)$.

We can now define a valid adversary, as we did in a similar manner in the ideal setting.

Definition 6: Given a symbolic API $(S, s_0, \Phi, \models, \mathcal{R}, \vdash, \mathsf{key})$ a valid symbolic adversary is a sequence of labels $\ell_1, \ldots \ell_n$ such that $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_n} s_n$.

To link a symbolic API to an ideal one we define the sets of wraps and handles that are the parameters of an ideal API as $\mathsf{Wraps} = \{w_t \mid t \in \mathcal{T}(\mathcal{F})\}$ and $\mathsf{Handles} = \{h_t \mid t \in \mathcal{T}(\mathcal{F})\}$. Moreover, we define the

mapping from traces in the symbolic model to traces in the idealized model, which allows us to obtain a notion of security for symbolic APIs:

Definition 7: We define a mapping from symbolic to ideal adversaries $s2i$ as follows

$$s2i(\ell_1\ell_2\dots\ell_n) = \begin{cases} \mathsf{corrupt}(h_t)\cdot s2i(\ell_2\dots\ell_n) \\ \quad \text{if } \ell_1 = \mathsf{corrupt}(t,\tilde{u}) \\ \mathsf{new}(h_t)\cdot s2i(\ell_2\dots\ell_n) \\ \quad \text{if } \ell_1 = \mathsf{new}(t,\tilde{u}) \\ \mathsf{wrap}(h_{t_1},h_{t_2},w_{t_3})\cdot s2i(\ell_2\dots\ell_n) \\ \quad \text{if } \ell_1 = \mathsf{wrap}(t_1,t_2,t_3,\tilde{u}) \\ \mathsf{unwrap}(h_{t_1},w_{t_2},h_{t_3})\cdot s2i(\ell_2\dots\ell_n) \\ \quad \text{if } \ell_1 = \mathsf{unwrap}(t_1,t_2,t_3,\tilde{u}) \\ s2i(\ell_2\dots\ell_n) \\ \quad \text{else} \end{cases}$$

Definition 8: (Secure API in the symbolic model) Let $A = (S, s_0, \Phi, \models, \mathcal{R}, \vdash, \mathsf{key})$ be a symbolic API. A is secure iff for any symbolic adversary $\ell_1,\dots\ell_n$ such that $s_0 \xrightarrow{\ell_1} \dots \xrightarrow{\ell_n} s_n$ we have that

- $q_1\dots q_k = s2i(\ell_1,\dots\ell_n)$ is a valid ideal adversary for $\mathsf{API}(\mathsf{Wraps},\mathsf{Handles})$ such that $\langle\emptyset,\emptyset,\emptyset,\emptyset,\emptyset\rangle \xrightarrow{q_1} \dots \xrightarrow{q_k} t_k$, and
- if u is an insecure handle in s_n then h_u is an insecure handle in t_k.

The intuition here is that no keys should become deducible except exactly those that are 'inevitably' deducible as a result of the key management and corruption operations. We use our projection to the idealized API to make this notion formal.

A. Example: a simple symbolic API

We now give an example of a simple symbolic API (sAPI) which instantiates our generic symbolic model and show its security. The API facilitates key management in the same fashion as the idealized API, and additionally permits encryption and decryption of data. To enforce the security of the API each key has a *level*, which is a natural number. Keys with positive levels may be used for wrapping other keys while keys of level 0 are used to encrypt data (we will motivate this scheme in Section IV-B). When a key is wrapped we require that its level is encrypted together with the key in order to guarantee consistency of the internal state, i.e. to avoid having multiple copies of the same key with different associated levels. This exemplifies the binding of metadata to wrapped keys that seems essential for a secure API.

To define sAPI in our model, we will use the set of function symbols $\mathcal{F}$ consisting of . for pairing, $\{x\}_y$ for symmetric key encryption of x by y and a handle symbol h of arity 3. Informally, the handle term $\mathsf{h}(h,k,i)$ encodes that h is a handle to key k of level i. We assume that constants include the natural numbers $\mathbb{N}$ (natural numbers could alternatively be encoded using a constant 0 and a unary function symbol $succ(\,)$).

We now define sAPI formally, except for the initial state. As we will see we can state the security of this API for any initial state in which there are no handles already on the devices.

- A state s consists of a set of terms $\mathcal{T}(\mathcal{F})$. Hence, we define S_{sAPI} to be $2^{\mathcal{T}(\mathcal{F})}$.
- We define the relation $s \vdash_{\mathsf{sAPI}} t$ as the smallest relation satisfying the rules given in Figure 1. These model the fact that state information is considered public (first rule) and the standard 'Dolev-Yao' attacker for symbolic models [20].
- The set of guards Φ_{sAPI} is the set of expressions

  $$t_1,\dots,t_k;u_1,\dots,u_p;i \sim 0$$

  where $t_1,\dots,t_k,u_1,\dots,u_k,i \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and $\sim \in \{=,>\}$.
- The satisfaction relation $\models_{\mathsf{sAPI}}$ is defined as $s \models t_1,\dots,t_k;u_1,\dots,u_k;i \sim 0$ if $s \vdash_{\mathsf{sAPI}} t_j$ for $1 \le j \le k$, $u_j \in \mathcal{F}, ar(u_j) = 0, u_j \notin st(s)$ and $j \ne k \Rightarrow u_i \ne u_j$ for $1 \le j,k \le p$, i.e. u_j are distinct fresh constants. We interpret $i \sim 0$ as the usual interpretation of $=$ and $>$ over the naturals, evaluating to false when $i \notin \mathbb{N}$.
- We will use the syntax

  $$t_1,\dots,t_k;y_1,\dots,y_p;x_i \sim 0 \xrightarrow{\ell(\tilde{t})} v_1,\dots,v_m$$

  where $(fv(v_1,\dots,v_m) \cup fv(\tilde{t})) \subseteq (fv(t_1,\dots,t_k) \cup \{y_1,\dots,y_p\})$ to define the rule $(\varphi,\ell(\tilde{t}),\mathsf{upd})$ as follows. $\varphi = t_1,\dots,t_k;y_1,\dots,y_p;x_i \sim 0$ and the update relation upd is defined as

  $$\mathsf{upd}(s,\theta,s') \mathrel{\hat{=}} s' = s \cup \{v_1\theta,\dots,v_n\theta\}$$

  The set of rules $\mathcal{R}_{\mathsf{sAPI}}$ is defined by the rules given in Figure 2.
- The function $\mathsf{key}_{\mathsf{sAPI}}$ is defined as:

  $$\mathsf{key}_{\mathsf{sAPI}}(s,h) = \begin{array}{ll} k & \text{if } \exists i.\ \mathsf{h}(h,k,i) \in s \\ \bot & \text{otherwise} \end{array}$$

  To guarantee that this is indeed a function, we will require that the initial state will never contain any occurrences of h, and that the rules creating h terms should always use fresh handles h.

We are now ready to state and prove security for sAPI.

$$\frac{t \in s}{s \vdash_{\mathsf{sAPI}} t} \qquad \frac{s \vdash_{\mathsf{sAPI}} t_1, s \vdash_{\mathsf{sAPI}} t_2}{s \vdash_{\mathsf{sAPI}} \{t_1\}_{t_2}} \qquad \frac{s \vdash_{\mathsf{sAPI}} \{t_1\}_{t_2}, s \vdash_{\mathsf{sAPI}} t_2}{s \vdash_{\mathsf{sAPI}} t_1}$$

$$\frac{s \vdash_{\mathsf{sAPI}} t_1, S \vdash_{\mathsf{sAPI}} t_2}{s \vdash_{\mathsf{sAPI}} t_1.t_2} \qquad \frac{s \vdash_{\mathsf{sAPI}} t_1.t_2}{s \vdash_{\mathsf{sAPI}} t_1} \qquad \frac{s \vdash_{\mathsf{sAPI}} t_1.t_2}{s \vdash_{\mathsf{sAPI}} t_2}$$

Figure 1. $\vdash_{\mathsf{sAPI}}$ deduction rules

$$\begin{array}{rlll}
 & x_i\,;y_h,y_k\,; & \xrightarrow{\mathsf{new}(y_h,x_i)} & y_h,\ \mathsf{h}(y_h,y_k,x_i) \\
x_h,x'_h,\mathsf{h}(x_h,x_k,x_i),\mathsf{h}(x'_h,x'_k,x_j)\,; & ;x_i>0 & \xrightarrow{\mathsf{wrap}(x_h,x'_h,\{x'_k.x_j\}_{x_k})} & \{x'_k.x_j\}_{x_k} \\
x_h,\{x'_k.x_j\}_{x_k},\mathsf{h}(x_h,x_k,x_i)\,; \quad y_h & ;x_i>0 & \xrightarrow{\mathsf{unwrap}(x_h,\{x'_k.x_j\}_{x_k},y_h)} & \mathsf{h}(y_h,x'_k,x_j) \\
x_h,\mathsf{h}(x_h,x_k,x_i)\,; & ; & \xrightarrow{\mathsf{corrupt}(x_h,x_k))} & x_k \\
x_h,x_m,\mathsf{h}(x_h,x_k,x_i)\,; & ;x_i=0 & \xrightarrow{\mathsf{enc}(x_h,x_m,x_k)} & \{x_m\}_{x_k} \\
x_h,\{x_m\}_{x_k},\mathsf{h}(x_h,x_k,x_i)\,; & ;x_i=0 & \xrightarrow{\mathsf{dec}(x_h,x_m)} & x_m
\end{array}$$

Figure 2. sAPI rules

Theorem 1: Let $s_0 \in S_{\mathsf{sAPI}}$ such that h does not occur in s_0. Then $(S_{\mathsf{sAPI}}, s_0, \Phi_{\mathsf{sAPI}}, \models_{\mathsf{sAPI}}, \mathcal{R}_{\mathsf{sAPI}}, \vdash_{\mathsf{sAPI}}, \mathsf{key}_{\mathsf{sAPI}})$ is secure.

In order to prove this theorem we prove a stronger property that we will call Sec^+ which makes explicit the tight link between the symbolic and ideal states and is defined as follows.

Definition 9: Let $s_0 \in S_{\mathsf{sAPI}}$ and $\ell_1 \dots \ell_n$ be a symbolic adversary such that $s_0 \xrightarrow{\ell_1} s_1 \cdots \xrightarrow{\ell_n} s_n$ and $q_1 \cdots q_m = s2i(\ell_1 \dots \ell_n)$. Property Sec^+ holds iff

1) $\langle \emptyset, \dots, \emptyset \rangle \xrightarrow{q_1} \cdots \xrightarrow{q_m} \langle C_m, W_m, H_m, wr_m, \equiv_m \rangle$
2) if $\mathsf{h}(h,k,i) \in st(s_n)$ and $s_n \vdash k$ then $\mathsf{h}_h \in C_m$
3) $\mathsf{h}(h,k,i) \in st(s_n)$ iff $\mathsf{h}(h,k,i) \in s_n$ iff $\mathsf{h}_h \in H_m$
4) if $\mathsf{h}(h,k,i) \in s_n$, $\mathsf{h}(h',k,j) \in s_n$ and $s \nvdash k$ then $\mathsf{h}_h \equiv_m \mathsf{h}_{h'}$
5) if $\{t_1\}_{t_2} \in st(s_n)$ and $t_1 = k.j$ and $\mathsf{h}(h,k,j) \in s_n$ and $s_n \nvdash k$, then $t_2 = k'$ and $\mathsf{h}(h',k',i) \in s_n$ and $i > 0$ and $w_{\{t_1\}_{t_2}} \in W_m$ and $wr_m(w_{\{t_1\}_{t_2}}) = \langle h', h \rangle$
6) if $\{t_1\}_{t_2} \in st(s_n)$ and $s_n \nvdash t_2$ and $\mathsf{h}(h,t_2,i) \in s_n$ and $i > 0$ then $\mathsf{h}(h',t_1,j) \in s_n$ and $w_{\{t_1\}_{t_2}} \in W_m$, $wr_m(w_{\{t_1\}_{t_2}}) = \langle h', h \rangle$ and $t_1 = k'.j$
7) if $\{t_1\}_{t_2} \in st(s_n)$ and $\mathsf{h}(h,t_2,0) \in s_n$ then $s_n \vdash t_1$
8) if $\{t_1\}_{t_2} \in st(s_n)$ and $\forall h,i,\mathsf{h}(h,t_2,i) \notin s_n$, then either h does not occur in t_1, or $s \vdash t_1$

Lemma 1: Let $s_0 \in S_{\mathsf{sAPI}}$ such that h does not occur in s_0. Let $\ell_1 \dots \ell_n$ be a symbolic adversary such that $s_0 \xrightarrow{\ell_1} s_1 \cdots \xrightarrow{\ell_n} s_n$ and $q_1 \cdots q_m = s2i(\ell_1 \dots \ell_n)$. Then property Sec^+ holds.

Proof: By induction on n (see technical report for details [21]). ■

Theorem 1 directly follows from this lemma.

B. Application to other APIs

In our API we used natural numbers as attributes, but tested only that attributes were zero or non-zero in the guards. This corresponds to the subset of PKCS#11 proved secure (for a weaker notion of security) by Fröschle and Steel [22], where *ed* (encryption and decryption) keys are mapped to 0, and *uw* (wrap and unwrap) keys are mapped to (say) 1. One can easily imagine a more refined API where the guards for wrap and unwrap fire only when the level of the payload key is strictly less than that of the wrapping key. This models a key hierarchy, which is a common feature of key management APIs [1, A.1]. Executions in this API are a subset of the executions of sAPI, hence security directly follows from the security of sAPI. Furthermore, we can show more refined properties, such as that if the highest level key corrupted has level n, then all deducible keys have level m where $m \leq n$. One can refine further by adding sets of users to the attributes of a key, and allowing wrap to fire only when the payload key is associated to a set of users that is a superset of the wrapping key, thus ensuring that wrap does not reveal the value of the payload key to anyone outside its user set. Thus we recover a version of the Cortier-Steel API [8] and the associated security

notion proposed there. We could also drop the explicit integer labels and instead build a hierarchy based on history, where we store in the symbolic state a table of *depends(k,k')* relations, expressing the fact that the security of k depends on the security of k'. Then we recover the core of Cachin and Chandran's design [9]. We could also consider asymmetric encryption, with an appropriate authenticity check for keys wrapped under public keys. However we leave all this for future work and move on to consider cryptographic details.

V. Computational model

In this section we present computational definitions for APIs and show how the definitions of Section III can be applied to give a computational notion of security. The syntax that we impose on APIs requires that they be equipped with algorithms for generating, wrapping, and unwrapping keys. These algorithms correspond to the key-management part of the API. Some of the keys can then be used to carry out cryptographic operations (whose result is observed outside the API). Roughly speaking, security of the API is defined in terms of adversaries that attempt to defeat the tasks for which the various keys of the API are intended. Specifically, we ask that the adversary cannot distinguish honestly created wrappings of keys from wrappings of a random key. Additionally, encryption under uncorrupted encryption keys should be secure, in a standard cryptographic sense (i.e. IND-CCA). Note that it is only the encryption of data that we require to be IND-CCA secure, not the wrapping of keys, which might use a deterministic scheme.

The presentation in this section assumes that the API is to be used for encryption. A more general and abstract treatment is also possible. In such a setting, key management stays unchanged but one leaves the set of cryptographic operations that the API should perform unspecified. Security is defined in terms of the typical cryptographic games that those primitives should satisfy.

A. Syntax

As is typical in cryptography executions depend on a security parameter η. The definition below uses families of sets $\{\mathsf{Keys}_\eta\}_\eta$ and $\{\mathsf{CWraps}_\eta\}_\eta$ which keys and key wrappings belong to. Both families are indexed by a security parameter $\eta \in \mathbb{N}$ and each individual set is a subset of $\{0,1\}^*$. When the security parameter is clear from the context we often omit it. Each API depends on a set of attributes $\mathsf{Attributes}$ and a set of handles $\mathsf{Handles}$, both subsets of $\{0,1\}^*$. We assume these sets are fixed and that the size of the set $\mathsf{Handles}$ is polynomial in η.

Definition 10: A computational API CA is defined by a tuple of algorithms (as specified below). In addition to handles and attributes, these algorithms also take as input (and produce as output) states from a set of states $\mathsf{States} = \{\mathsf{States}_\eta\}_{\eta\in\mathbb{N}}$, which is just some subset of $\{0,1\}^*$. The algorithms are as follows.

- $\mathsf{CA.init}$ is a (possibly) randomized initialization function that takes as input a security parameter η and returns a state $s_0 \in \mathsf{States}_\eta$.
- algorithms $\mathsf{CA.key}$ and $\mathsf{CA.attr}$ take as input a state $s \in \mathsf{States}$ and a handle $h \in \mathsf{Handles}$ and return bitstrings. These are the key and attributes associated to the handle h in state s.
- algorithm $\mathsf{CA.new}$ takes as input an attribute $a \in \mathsf{Attributes}$ and a state s and returns a pair $(\bar{s}, \bar{h}) \in \mathsf{States} \times \mathsf{Handles}$. We write $(\bar{s}, \bar{h}) \leftarrow \mathsf{CA.new}(s, a)$ for this process. This corresponds to generating a new key with attribute a. The handle $\bar{h}$ points to the key.
- algorithm $\mathsf{CA.wrap}$ takes as input a triple $(s, h_1, h_2) \in \mathsf{States} \times \mathsf{Handles} \times \mathsf{Handles}$ and returns a pair $(\bar{s}, w) \in \mathsf{States} \times (\mathsf{CWraps} \cup \{\perp\})$. The result w is the wrapping of the key associated to h_2 under the key associated to h_1 (in state s).
- algorithm $\mathsf{CA.unwrap}$ takes as input a tuple $(s, h, w) \in \mathsf{States}_\eta \times \mathsf{Handles} \times \{0,1\}^*$ and returns a pair $(\bar{s}, \bar{h})$. Intuitively, this command unwraps w with the key associated to handle h. Handle $\bar{h}$ points to the resulting key (if unwrapping succeeds).
- algorithm $\mathsf{CA.enc}$ is randomized. It takes as input $(s, h, p) \in \mathsf{States}_\eta \times \mathsf{Handles} \times \{0,1\}^*$ and returns a ciphertext $c \in \{0,1\}^*$. The decryption algorithm $\mathsf{CA.dec}$ takes as input $(s, h, c) \in \mathsf{States}_\eta \times \mathsf{Handles} \times \{0,1\}^*$ and returns $p \in \{0,1\}^*$.

Remark 2: All of the algorithms take as an additional input the security parameter (which we only show for the initialization algorithm) and all of the algorithms may also return an error symbol $\perp$. In a more abstract incarnation of the above syntax, the encryption and decryption algorithms could be replaced by some arbitrary cryptographic function.

B. Correctness and Security

In this section we define correctness and security for a computational API. We first discuss the rationale behind our definition of security. The definition that we present incorporates two main ideas: first, notice that on the one hand, unlike in the symbolic setting (Definition 8), security for the keys associated to handles cannot be defined in terms of key-recovery (the

resulting notion would be too weak by the standards of modern cryptography). On the other hand, the typical paradigm for defining security of keys by requiring that they be indistinguishable from random keys is too strong for the setting of APIs: as soon as a key is used this indistinguishability property is inevitably lost. Instead, we define security by asking the adversary to defeat the tasks for which keys are to be used. We model this idea using a standard real-or-fake definitional approach. We define two types of executions of the API. In the *real* execution the algorithms of the API are used as expected. In the *fake* execution the information that is supposedly secret, for example because it has been encrypted under a key unknown to the adversary, is completely (in an information theoretical way) hidden from the view of the adversary. Security then demands that the adversary cannot see a difference between the two executions.

The second main idea of our definition is that it asks for security only for those keys that are "ideally" secure, in the sense defined in Section III. More precisely, for any computational adversary we extract a probabilistic idealized adversary (by only considering the sub-sequence of calls related to key management). We then demand security for those keys that with overwhelming probability are secure in the face of the idealized adversary (as any other key is trivially known to the adversary with non-negligible probability).

Before giving the details we briefly discuss the games that we use in our formal definition. The execution of the APIs is driven by the adversary by the means of queries:

- query NEW allows the adversary to initialize keys with arbitrary attributes;
- queries ENC and WRAP allow the adversary to obtain encryptions and wrappings of his choice;
- queries DEC and UNWRAP allow the adversary to decrypt and unwrap ciphertexts and wrappings of his choice
- query CORRUPT allows the adversary to corrupt keys.

Real and fake executions: As explained above we will consider two types of executions: real executions and fake executions. These executions will be defined by the means of two experiments which take as parameter a handle h^*. The fake execution of APIs is similar to the real one, except that the API provides "fake" answers to wrap and encryption queries for handle h^*. Fake wraps are computed by replacing the key to be wrapped with a randomly chosen key k_0. Fake encryptions are produced by encrypting a string of 0s instead of the real plaintext. We write $s[\mathsf{key}(h) \overset{\$}{\mapsto} k_0]$ for the state s in which $\mathsf{key}(h)$ has been replaced with the randomly chosen k_0. The association between fake and real wrappings is maintained in a list L_w, similarly, the association between real ciphertexts and fake ones is maintained in list L_c. Before answering unwrapping or decryption requests, the fake execution uses these lists to convert fake wraps and ciphertexts to real ones. This allows us to avoid the weakness of the Cachin-Chandran proof described in Section II, where an attacker cannot unwrap a previously created wrap, and hence an insecure wrapping method is not ruled out by a security proof. The list L_w also allows to check if a key had been wrapped under the the key associated to h^* before; in this case, to ensure consistency with the real experiment, the fake experiment needs to return the previous (fake) wrapping.

Definition 11 (Real and fake executions of APIs): The real and fake executions of a computational API CA in the presence of adversary A are defined by experiments $\mathbf{Exp}^{\mathsf{sec},\mathsf{exe},h^*}_{\mathsf{CA},A}(\eta)$ where $\mathsf{exe} \in \{\mathsf{real}, \mathsf{fake}\}$ defined as follows. The initial state of the API is obtained via $s \leftarrow \mathsf{init}(\eta)$. The adversary interacts with the experiment via a set of queries described below. We explain how the experiment answers each type of query. (Notice the different font that we use to distinguish between the queries of the adversary and the executions of the algorithms that they trigger). Each query also defines a labeled transition between states (which we also describe below).

- NEW(a): $(\bar{s}, \bar{h}) \leftarrow \mathsf{CA.new}(s, a)$;
  define transition $s \xrightarrow{\mathsf{new}(\bar{h},a)} \bar{s}$;
  set s to $\bar{s}$; return $\bar{h}$.
- WRAP(h_1, h_2):
  $(\bar{s}, \bar{w}) \leftarrow \mathsf{CA.wrap}(s, h_1, h_2)$;
  if $\mathsf{exe} = \mathsf{fake}$ and $h_1 = h^*$ then
  if $\exists w.\ (w, \bar{w}) \in L_w$
  then set $\bar{w}$ to w
  else $(s', w) \leftarrow \mathsf{CA.wrap}(s[\mathsf{key}(h_2) \overset{\$}{\mapsto} k_0], h_1, h_2)$;
  add $(w, \bar{w})$ to L_w; set $\bar{w}$ to w;
  define transition $s \xrightarrow{\mathsf{wrap}(h_1,h_2,\bar{w})} \bar{s}$;
  set s to $\bar{s}$; return $\bar{w}$.
- UNWRAP(h, w):
  if $\mathsf{exe} = \mathsf{fake}$, $h = h^*$ and $\exists \bar{w}.\ (w, \bar{w}) \in L_w$
  then $(\bar{s}, \bar{h}) \leftarrow \mathsf{CA.unwrap}(s, h, \bar{w})$
  else $(\bar{s}, \bar{h}) \leftarrow \mathsf{CA.unwrap}(s, h, w)$;
  define transition $s \xrightarrow{\mathsf{unwrap}(h,w,\bar{h})} \bar{s}$;
  set s to $\bar{s}$, return $\bar{h}$.
- ENC(h, p):

$(\bar{s}, \bar{c}) \leftarrow \mathsf{CA.enc}(s, h, p)$
if $\mathsf{exe} = \mathsf{fake}$ and $h_1 = h^*$ then
$(s', c) \leftarrow \mathsf{CA.enc}(s, h, 0^{|p|})$;
add $(c, \bar{c})$ to L_c; set $\bar{c}$ to c;
define transition $s \xrightarrow{\mathsf{enc}(h,p,\bar{c})} \bar{s}$;
set s to $\bar{s}$, return $\bar{c}$.

- DEC(h, c):
if $\mathsf{exe} = \mathsf{fake}$, $h = h^*$ and $\exists \bar{c}.\ (c, \bar{c}) \in L_c$
then $(\bar{s}, p) \leftarrow \mathsf{CA.dec}(s, h, \bar{c})$
else $(\bar{s}, p) \leftarrow \mathsf{CA.dec}(s, h, c)$;
define transition $s \xrightarrow{\mathsf{dec}(h,c,p)} \bar{s}$;
set s to $\bar{s}$, return p.
- CORRUPT(h):
define transition $s \xrightarrow{\mathsf{corrupt}(h)} s$;
return $\mathsf{key}(s, h)$.

At the end of the execution the adversary has to output a bit, which we set to be the outcome of the experiment, i.e. the adversary's guess as to whether he is talking to the real or the fake API.

Note that each execution (fixed by the random coins used by parties) defines a sequence of labeled transitions $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_n} s_n$.

Correctness: Before moving on with defining security for APIs we use the game defined above to give a definition for the correctness of an API. An implementation of an API needs to satisfy some minimal correctness requirements. We require that newly created keys have the correct attribute associated with them. We require that ciphertexts created by the API should be decrypted correctly by the API at a later time. Finally, unwrapping a wrap that contains a key k produced by the API should result in a handle that points to the same key, and for which the associated attributes are those that key k originally had.

Definition 12: A computational API CA is correct if the following holds. Let $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_n} s_n$ be the transition sequence defined by an arbitrary execution of CA. Consider an arbitrary step $s_{i-1} \xrightarrow{\ell_i} s_i$ in the execution.

- If ℓ_i is $\mathsf{new}(h, a)$ then in any later state s_j, $\mathsf{attr}(s_j, h) = a$ and h is fresh, i.e., h did not occur in any label before.
- If ℓ_i is $\mathsf{wrap}(h_1, h_2, w)$ with $w \neq \perp$ then for any latter transition $s_{j-1} \xrightarrow{\mathsf{unwrap}(h_1,w,h_3)} s_j$, it holds that $\mathsf{CA.key}(s_j, h_3) = \mathsf{key}(s_{i-1}, h_2)$ and $\mathsf{CA.attr}(s_j, h_3) = \mathsf{attr}(s_{i-1}, h_2)$.
- If ℓ_i is $\mathsf{unwrap}(h_1, w, h_2)$ then h_2 is fresh.
- If ℓ_i is $\mathsf{enc}(h, p, c)$ with $c \neq \perp$, then for latter $s_{j-1} \xrightarrow{\mathsf{dec}(h_1,c,p')} s_j$ it holds that $p' = p$.

Security: As explained above, for a secure API an adversary should not be able to tell apart real and fake executions. Clearly, this task is easy if the adversary, for example, corrupts a key used for wrapping h^*, so some restrictions are needed. We only impose minimal conditions: we extract out of the interaction of the adversary with the API an idealized attacker (in the sense defined in Section III). Then, we demand that the attacker does not corrupt (in the ideal world) the handle under attack (except with negligible probability). Next we define the (probabilistic) ideal adversary associated to a computational adversary A.

Definition 13 (Ideal adversary): Let r_Π and r_A be some fixed but arbitrary random coins used by the experiment and the adversary, respectively in the execution of $\mathbf{Exp}^{\mathsf{sec,real},h^*}_{\Pi,A}(\eta)$, and let $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_n} s_n$ be the resulting transition sequence. We define $I(A)(r_\Pi, r_A)$ as the ideal adversary obtained by applying the transformation $c2i$ (defined below labelwise) to $\ell_1, \ldots \ell_n$.

$$c2i(\ell_1 \ell_2 \ldots \ell_n) = \begin{cases} \mathsf{corrupt}(h) \cdot c2i(\ell_2 \ldots \ell_n) \\ \quad \text{if } \ell_1 = \mathsf{corrupt}(h) \\ \mathsf{new}(h) \cdot c2i(\ell_2 \ldots \ell_n) \\ \quad \text{if } \ell_1 = \mathsf{new}(a, h) \\ \mathsf{wrap}(h_1, h_2, w) \cdot c2i(\ell_2 \ldots \ell_n) \\ \quad \text{if } \ell_1 = \mathsf{wrap}(h_1, h_2, w) \\ \mathsf{unwrap}(h, w, h') \cdot c2i(\ell_2 \ldots \ell_n) \\ \quad \text{if } \ell_1 = \mathsf{unwrap}(h, w, h') \\ c2i(\ell_2 \ldots \ell_n) \\ \quad \text{else} \end{cases}$$

The randomized ideal adversary $I(A)$ (with sample space (r_A, r_Π)) is the ideal adversary associated to A.

Definition 14 (Ideally (in)secure handles): Let $I(A)$ be an arbitrary probabilistic ideal adversary that corresponds to a concrete adversary A, and let $h^* \in \mathsf{Handles}$ be an arbitrary handle. Adversary $I(A)$ induces a probability distribution on the states of the $\mathsf{API}(\{0,1\}^*, \mathsf{Handles})$ by considering the final state $s = \langle C, W, H, wr, \equiv \rangle$ of its executions. We say that handle h^* is ideally secure with respect to $I(A)$ if with overwhelming probability (over the same sample space as adversary $I(A)$), handle h^* is secure in s (in the sense of Definition 2).

We use the ideal adversary associated to a real adversary to characterize the class of *valid* adversaries. These are adversaries who do not win the real-versus-fake API game in a trivial manner. There are two types of trivial attacks. The simplest is to distinguish real from fake wrappings when the challenge handle is corrupt. We therefore simply ask that the challenge handle h^*

is not insecure (as captured by the definition above). The remaining attack relies on the fact that different handles may have associated equal keys (as captured by relation $\equiv$ of Definition 1). Assume that an adversary is allowed to ask for $\text{WRAP}(h, h_1)$ and $\text{WRAP}(h^*, h_1)$ for some handle h that is equivalent to h^*. In the real API both queries will trigger equal answers, whereas in the fake API the two answers will be different with overwhelming probability. The above situation is simply an artifact of our models and does not reflect real attacks, so we simply forbid the adversary to issue queries as above.

Definition 15 (Valid computational adversary): Let A be an adversary for experiment $\mathbf{Exp}^{\mathsf{sec},\mathsf{real},h^*}_{\mathsf{CA},A}(\eta)$, and let $I(A)$ its associated idealized adversary. We say that A is a valid adversary with respect to handle h^* if h^* is ideally secure with respect to $I(A)$ and the adversary does not query $\text{WRAP}(h, \cdot)$ for any $h \equiv h^*$. Here $\equiv$ is the equivalence relation on handles induced by $I(A)$.

The following definition says that an implementation of an API is secure if any handle that is not trivially known to the adversary is secure. Recall that a negligible function is a function that decreases faster than the inverse of any polynomial.

Definition 16 (Computationally secure API): A computational API CA is secure, if for all handles h^* and all probabilistic polynomial time adversaries A valid with respect to h^* it holds that $\mathbf{Adv}^{\mathsf{sec},\mathsf{API}}_{\mathsf{CA},A}(\eta) =$

$$\Big|\Pr\left[b \leftarrow \mathbf{Exp}^{\mathsf{sec},\mathsf{real},h^*}_{\mathsf{CA},A}(\eta) \ : b = 1\right] - \Pr\left[b \leftarrow \mathbf{Exp}^{\mathsf{sec},\mathsf{fake},h^*}_{\mathsf{CA},A}(\eta) \ : b = 1\right]\Big|$$

is a negligible function in η.

Remark 3: In the above experiment, fixing the handle h^* *a priori* (even though it is universally quantified) may seem restrictive compared to an experiment where the attacker decides adaptively which handle to attack. This is not the case. Since the set of handles is of polynomial size, for at least one handle h_0 the adaptive adversary would select that handle (as h^*) and win with non-negligible probability (as otherwise the overall advantage would be negligible). The adaptive adversary could then be converted into an adversary for the experiment $\mathbf{Exp}^{\mathsf{sec},\mathsf{realexec},h_0}$: the natural restrictions on an adaptive adversary would immediately imply that the same adversary is valid for h_0.

VI. An efficient key-wrap-based implementation

A. Cryptographic primitives

Notation: In the presentation below we use the following notation. We write $|x|$ for the size of x. If x is a bitstring, then $|x|$ is its length. If x is a set, then $|x|$ is its cardinality. We write $y \leftarrow A(x)$ for the process of executing the (possibly randomized) algorithm A on input x and obtaining y as a result. We write $\$^l$ for a string selected uniformly at random among the strings of length l.

The implementation of the API that we analyze uses a deterministic key-wrap scheme for wrapping keys, and a standard symmetric encryption scheme. Key-wrap schemes were first formalized by Rogaway and Shrimpton under the name of deterministic authenticated encryption [15]. Although the presentation and security notion in this section build on theirs, we term the primitive key-wrap as we are only concerned with the case when the scheme is used to encrypt other keys and not arbitrary plaintexts. Nonetheless, we use "encrypt" and "wrap" (and "decrypt" and "unwrap") interchangeably.

A key-wrap scheme KW is given by algorithms $(\mathsf{KW.KG}, \mathsf{KW.Wrap}, \mathsf{KW.UnWrap})$ for key generation, wrapping, and unwrapping, respectively. The scheme is parametrized by a key space $\mathcal{K}$ (from which keys are drawn), a header space $\mathcal{H}$ (that contains data that can be authenticated with each encryption), and a ciphertext space $\mathcal{Y}$. For simplicity we assume that for each security parameter keys are bitstrings of some fixed length, and that the key generation algorithm simply picks (uniformly at random) one of these keys. While the original work considers encryption schemes that can encrypt arbitrary plaintexts, here we only consider the case when one only needs to encrypt other keys. The space of plaintexts is therefore $\mathcal{K}$. The wrapping algorithm takes as input arguments $(k_1, k_2, a) \in \mathcal{K} \times \mathcal{K} \times \mathcal{H}$ and returns a ciphertext $c \in \mathcal{Y}$ – the encryption of k_2 under k_1 with authenticated header a. We write $c \leftarrow \mathsf{KW.Wrap}^a_{k_1}(k_2)$ for performing such an encryption and obtaining c. Unwrapping takes as input a key k, and attribute a, and a ciphertext c, and output a value in $\mathcal{K} \cup \{\perp\}$. We write $\mathsf{KW.UnWrap}^a_k(c)$ for the result of unwrapping c and authenticated header a with key k. Correctness of the wrapping scheme requires that for any $k_1, k_2 \in \mathcal{K}$ and any $a \in \mathcal{H}$, if $c \leftarrow \mathsf{KW.Wrap}^a_{k_1}(k_2)$ then $\mathsf{KW.UnWrap}^a_{k_1}(c) = k_1$.

We briefly discuss the intuition behind the security definition that we give next. Our definition builds on that of [15] where the authors only consider the case

when a single key is under attack by an adversary. The goal of the adversary is twofold: to break the secrecy of plaintexts encrypted under the key, and to create valid looking ciphertexts without knowledge of the secret key. Security is defined by comparing two worlds. In the real world the adversary has access to an encryption oracle and a decryption oracle that work as expected. In the fake world the encryption oracle returns random strings, and the decryption oracle simply rejects all ciphertexts (of course, the adversary is not allowed to submit ciphertexts obtained from the encryption oracle). Security demands that an adversary cannot tell whether it interacts with the real oracles or the fake ones. Secrecy of plaintexts is guaranteed as the adversary cannot tell apart real encryptions from random strings, and authenticity of ciphertexts is guaranteed since an adversary that could create a new valid ciphertext would immediately distinguish between the real and the fake world (in the latter the valid ciphertext would be rejected).

In this paper we present and use an extension to a setting where multiple keys are used by some system. Our model reflects the possibility that an adversary sees encryptions of messages that it chooses (with arbitrary associated authenticated data), and can decrypt whatever message he chooses. Furthermore, he can see encryptions of keys under other keys and can corrupt whichever keys he wants. In the model that we give below we assume that the encryption is length regular (the size of the ciphertext depends only on the sizes of the inputs).

The security notion that we give is directly motivated by the use of key-wrapping schemes in APIs: the original notion does not directly capture this usage and some attacks are not even indirectly captured. The situation has a parallel in the history of standard encryption schemes. The original notion of security for encryption was only concerned with security of ciphertexts in a single-user setting [23]. Attacks against encryption when used in a multi-user setting were only later considered. These attacks include adaptively corrupting keys [24], sending the same message under several different keys [25], and/or seeing key-dependent messages [26]. It became apparent only much later, and after a sustained research effort, that while in some cases security in the most basic (single-key) sense suffices to guarantee security against stronger attacks [25], [27] often, and perhaps counterintuitively, this is not always true [28].

All these attacks are unfortunately realistic possibilities against key-wrapping schemes as used in APIs, and the security notion that we give captures them directly. In light of the above discussion, we caution that our notion is significantly stronger than that of Rogaway and Shrimpton, as we demand (simultaneous) resistance to key-dependent encryption and adaptive corruption in a multi-user setting. Consequently, constructions that meet the notion for the single key case may actually be insecure under our notion. We thus open an important avenue of further research that aims to construct schemes secure under our notion (in the conclusions section we describe two possible directions), or to prove that such constructions are impossible. The formal definition follows.

Definition 17 (Multi-user setting for key wrapping): We define experiments $\mathbf{Exp}^{\mathsf{wrap},\mathsf{real}}_{\mathcal{A},\mathsf{KW}}(\eta)$ and $\mathbf{Exp}^{\mathsf{wrap},\mathsf{fake}}_{\mathcal{A},\mathsf{KW}}(\eta)$. In both experiments the adversary can access a number of keys $k_1, k_2, \ldots, k_n \ldots$ (which he can ask to be created via a query NEW). In his other queries, the adversary refers to these keys via symbols $K_1, K_2, \ldots, K_n$ (where the implicit mapping should be obvious). By abusing notation we often use K_i as a placeholder for k_i so, for example, $\mathsf{KW.Wrap}^a_{K_i}(K_j)$ means $\mathsf{KW.Wrap}^a_{k_i}(k_j)$. We now explain the queries that the adversary is allowed to make, and how they are answered in the two experiments.

- $\mathsf{NEW}(K_i)$: a new key k_i is generated via $k_i \leftarrow \mathsf{KW.KG}(\eta)$
- $\mathsf{ENC}(K_i, a, m)$ where $m \in \mathcal{K} \cup \{K_i \mid i \in \mathbb{N}\}$ and $h \in \mathcal{H}$. The experiment returns $\mathsf{KW.Wrap}^a_{k_i}(m)$.
- $\mathsf{TENC}(K_i, a, m)$ where $m \in \mathcal{K} \cup \{K_i \mid i \in \mathbb{N}\}$ and $a \in \mathcal{H}$. The real experiment returns $\mathsf{KW.Wrap}^a_{k_i}(m)$, whereas the fake experiment returns $\$^{|\mathsf{KW.Wrap}^a_{k_i}(m)|}$
- $\mathsf{DEC}(K_i, a, c)$: the real experiment returns $\mathsf{KW.UnWrap}^a_{k_i}(c)$, the fake experiment returns $\perp$.
- $\mathsf{CORR}(K_i)$: the experiment returns k_i.

Consider the directed graph whose nodes are the symbolic keys K_i and in which there is an edge from K_i to K_j if the adversary issues a query $\mathsf{ENC}(K_i, a, K_j)$. We say that a key K_i is corrupt if either the adversary issued query $\mathsf{CORR}(K_i)$, or if the key is reachable in the above graph from a corrupt key.

We make the following assumptions on the behaviour of the adversary.

- For all i the query $\mathsf{NEW}(K_i)$ is issued at most once.
- All the queries issued by the adversary contain keys that have already been generated by the experiment.
- The adversary never makes a test query $\mathsf{TENC}(K_i, a, K_j)$ if K_i is corrupted at the end of the experiment.

- If A issues test query $\mathsf{TENC}(K_i, a, m)$ then A does not issue $\mathsf{TENC}(K_j, a', m')$ or $\mathsf{ENC}(K_j, a', m')$ with $(K_i, m) = (K'_j, m')$
- The adversary never queries $\mathsf{DEC}(K_i, a, c)$ if c was the result of a query $\mathsf{TENC}(K_i, a, m)$ or of a query $\mathsf{ENC}(K_i, a, m)$.

At the end of the execution the adversary has to output a bit b which is also the result of the experiment. The advantage of adversary A in breaking the key-wrapping scheme KW is defined by:

$$\mathbf{Adv}^{\mathsf{wrap}}_{\mathsf{KW},A}(\eta) = \Big|\Pr\left[b \leftarrow \mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},A}(\eta) \ : b = 1\right] - \Pr\left[b \leftarrow \mathbf{Exp}^{\mathsf{wrap,fake}}_{\mathsf{KW},A}(\eta) \ : b = 1\right]\Big|$$

and KW is secure if the advantage of any probabilistic polynomial time algorithm is negligible.

Remark 4: The above definition ensures that wrappings look like random strings (from an appropriate domain). This level of security is very likely beyond what is really needed in applications, especially since wrappings may have some fixed format, come with tags, etc. We give this notion because it is the proper generalization of the notion proposed by Rogaway and Shrimpton. An alternative security notion, strictly weaker than the one above but sufficient for our application is as follows. The experiment is kept mostly unchanged, except that the answer to a test encryption query $\mathsf{TENC}(K_i, h, m)$ is calculated, in the fake game, as $\mathsf{KW.Wrap}^h_{k_i}(\$^{|m|})$ (where we define $\mid K_i \mid$ as $\mid k_i \mid$). That is, instead of requiring that encryptions/wrappings look random, we require that they are indistinguishable from encryptions/wrappings of random plaintexts/keys (of appropriate lengths). In the rest of the paper we use this weaker requirement.

Symmetric encryption schemes: A symmetric encryption scheme SE is given by algorithms $(\mathsf{SE.KG}, \mathsf{SE.Enc}, \mathsf{SE.Dec})$. We are interested in schemes that satisfy the standard IND-CCA notion of security. We recall this notion in the style of real-or-fake world used above. We define experiments $\mathbf{Exp}^{\mathsf{IND\text{-}CCA,real}}_{\mathsf{SE},A}(\eta)$ and $\mathbf{Exp}^{\mathsf{IND\text{-}CCA,fake}}_{\mathsf{SE},A}(\eta)$ in which an adversary has access to a set of oracles keyed with a key k generated via $k \leftarrow \mathsf{SE.KG}(\eta)$. The oracles are as follows. The encryption oracle expects to receive a message m. In the real experiment the oracle answers with an encryption $c \leftarrow \mathsf{SE.Enc}(k, m)$. In the fake experiment the answer is $c \leftarrow \mathsf{SE.Enc}(k, 0^{|m|})$ (an encryption of the all-zero string). The decryption oracle receives a ciphertext c and returns the result p of $p \leftarrow \mathsf{SE.Dec}(k, c)$. Ciphertexts obtained from the encryption oracle are not allowed to be sent to the decryption oracle. At the end of its execution, the adversary outputs a bit, which is also set to be the output of the experiment. The advantage of adversary A in breaking IND-CCA security of the encryption scheme SE is defined as $\mathbf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{SE},A}(\eta) =$

$$\Big|\Pr\left[b \leftarrow \mathbf{Exp}^{\mathsf{IND\text{-}CCA,real}}_{\mathsf{SE},A}(\eta) \ : b = 1\right] - \Pr\left[b \leftarrow \mathbf{Exp}^{\mathsf{IND\text{-}CCA,fake}}_{\mathsf{SE},A}(\eta) \ : b = 1\right]\Big|$$

We say that SE is IND-CCA secure if for all probabilistic polynomial time attacker A, $\mathbf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{SE},A}(\eta)$ is a negligible function in η.

B. A computational API

The computational API that we specify and analyze in this section is similar in design to the symbolic API presented in Section IV-A. The construction uses a wrapping scheme KW to wrap and bind attributes to keys and a standard symmetric encryption scheme SE to perform encryptions. We write $\mathsf{CA}_{\mathsf{SE,KW}}$ (or simply CA) for the computational API that we define. In the implementation that we consider the set of handles is the set of natural numbers, i.e. $\mathsf{Handles} = \mathbb{N}$. The set of attributes is $\mathsf{Attributes} = \{0, 1, 2, \ldots, n\}$ for some fixed n. Intuitively, like the symbolic API in Section IV-A, the keys with attribute 0 are used for standard encryption, whereas keys with any other attribute are used for wrapping. We also impose a hierarchy on these keys based on their level, for two reasons. The first is to show that such hierarchies, commonly used in APIs can be enforced cryptographically, so no global state needs to be stored. The second reason is more indirect. Below, we prove the security of this API construction based on the strong notion of security for key wrapping defined earlier in the paper. The additional restriction may allow a proof based on weaker security for the key wrapping scheme (e.g. as defined originally by Rogaway and Shrimpton).

The internal states of the API that we consider are given by:

- a partial map $\mathsf{attr} : \mathbb{N} \to \mathsf{Attributes}$ that associates to each handle an attribute
- a partial map $\mathsf{key} : \mathbb{N} \to \{0,1\}^*$ that associates to each handle a bitstring (that is a key)
- for simplicity, we leave unspecified a method by which the API keeps track of all handles used, and also assume a method for selecting a fresh new handle if needed.

The implementation CA is as follows (although we do not show this explicitly, the state of the API and the security parameter are inputs to all of the algorithms).

- $\mathsf{CA.init}(\eta)$. Set the security parameter for the API to η.
- $\mathsf{CA.new}(h, a)$. Set $\mathsf{CA.attr}(h) \leftarrow a$. If $a = 0$ then $k \leftarrow \mathsf{SE.KG}(\eta)$ else $k \leftarrow \mathsf{KW.KG}(\eta)$. Set $\mathsf{key}(h) \leftarrow k$.
- $\mathsf{CA.wrap}(h_1, h_2)$. If $\mathsf{attr}(h_1) \notin \{1, 2, 3, \ldots, n\}$ or $\mathsf{attr}(h_1) \leq \mathsf{attr}(h_2)$ then return $\perp$. Else, set $w \leftarrow \langle \mathsf{KW.Wrap}^{\mathsf{attr}(h_2)}_{\mathsf{key}(h_1)}(\mathsf{key}(h_2)), \mathsf{attr}(h_2)\rangle$. Output w
- $\mathsf{CA.unwrap}(h, w)$. Decode w as (c, a) (if decoding fails output $\perp$). $k \leftarrow \mathsf{KW.UnWrap}^{a}_{\mathsf{key}(h)}(c)$ (if unwrapping fails, output $\perp$). Select a fresh handle $\bar{h}$. Set $\mathsf{attr}(\bar{h}) \leftarrow a$, $\mathsf{key}(\bar{h}) \leftarrow k$. Output $\bar{h}$.
- $\mathsf{CA.enc}(h, p)$. If $\mathsf{attr}(h) \neq 0$ then return $\perp$. Else, set $c \leftarrow \mathsf{SE.Enc}(\mathsf{key}(h), p)$. Output c.
- $\mathsf{CA.dec}(h, c)$. Set $p \leftarrow \mathsf{SE.Dec}(\mathsf{key}(h), c)$. Output p.

The following definition says that the construction above is secure if its two main building blocks are.

Theorem 2: If SE is an IND-CCA symmetric encryption scheme, and KW is a secure key-wrapping mechanism, then CA is a secure API.
The proof is given in a technical report [21].

VII. Conclusions

We have defined a notion of security for key management APIs and demonstrated its utility in security proofs of APIs in the symbolic and computational models. Our notion captures the intuition of security in an API, where only keys that are inevitably lost as the result of dependencies and corruptions are insecure. It captures the separation of the key usage and key management functions, avoiding the kinds of failures seen in many previous APIs. By defining security in executions where keys may be wrapped and unwrapped, we ensure in a general way that key metadata or attributes - however they are expressed - are tracked properly, avoiding the drawback of the Cachin-Chandran API and associated proof [9]. By treating security in a modern cryptographic model, we obtain stronger assurances of correctness than is possible with purely symbolic treatments such as that of Cortier and Steel [8].

As explained earlier, when showing security of the computational API that we present, we identified a very strong requirement for key wrapping schemes, combining two notions which are notoriously difficult to achieve: security against key-dependent encryptions, and against adaptive corruption of keys. While elsewhere such attacks may be brushed-off as irrelevant, in the context of APIs they are of clear practical concern. In future work we plan to explore solutions for this problem. Immediate directions include defining restricted scenarios where a reasonable level of security can still be achieved (see e.g. the work of Panjwani for the case of randomized symmetric encryption [29]) and coming up with constructions that work heuristically, e.g. using random oracles [30]. Note that models for protocol verification that support key wrapping in the form of session keys encrypted under long term keys can usually avoid this problem [31], [32], since one can restrict to protocols that respect certain assumptions such as never wrapping a key after it has been used. For a general purpose API model, we have to accommodate applications that might want to e.g. take backups of keys that are in use, for example in encrypted storage applications.

Using our uniform notion of security that can be used for APIs with various functionalities and notions of state, we can move on in future work to proposing and analysing new designs for APIs that combine aspects of the distributed and centralised designs in the literature. We hope to contribute to the open standards processes currently considering the next generation of key management APIs.

References

[1] International Organization for Standardization, "ISO 9564-1: Banking personal identification number (PIN) management and security," 30 pages.

[2] C. Cachin and J. Camenisch, "Encrypting keys securely," *IEEE Security & Privacy*, vol. 8, no. 4, pp. 66–69, 2010.

[3] R. Anderson, "The correctness of crypto transaction sets," in *Proc. 8th International Workshop on Security Protocols*, ser. Lecture Notes in Computer Science, vol. 2133. Springer, 2000, pp. 125–127.

[4] M. Bond, "Attacks on cryptoprocessor transaction sets," in *Proc. 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, ser. Lecture Notes in Computer Science, vol. 2162. Springer, 2001, pp. 220–234.

[5] R. Clayton and M. Bond, "Experience using a low-cost FPGA design to crack DES keys," in *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, ser. Lecture Notes in Computer Science, vol. 2523. Springer, 2003, pp. 579–592.

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

[7] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel, "Attacking and fixing PKCS#11 security tokens," in *Proc. 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM Press, 2010, pp. 260–269.

[8] V. Cortier and G. Steel, "A generic security API for symmetric key management on cryptographic devices," in *Proc. 14th European Symposium on Research in Computer Security (ESORICS'09)*, ser. Lecture Notes in Computer Science, vol. 5789. Springer, 2009, pp. 605–620.

[9] C. Cachin and N. Chandran, "A secure cryptographic token interface," in *Proc. 22th IEEE Computer Security Foundation Symposium (CSF'09)*. IEEE Computer Society Press, 2009, pp. 141–153.

[10] J. Clulow, "On the security of PKCS#11," in *Proc. 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, ser. Lecture Notes in Computer Science, vol. 2779. Springer, 2003, pp. 411–425.

[11] D. Longley and S. Rigby, "An automatic search for security flaws in key management schemes," *Computers and Security*, vol. 11, no. 1, pp. 75–89, March 1992.

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

[13] V. Cortier, G. Keighren, and G. Steel, "Automatic analysis of the security of XOR-based key management schemes," in *Proc. 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'07)*, ser. Lecture Notes in Computer Science, no. 4424, 2007, pp. 538–552.

[14] S. Delaune, S. Kremer, and G. Steel, "Formal analysis of PKCS#11 and proprietary extensions," *Journal of Computer Security*, vol. 18, no. 6, pp. 1211–1245, Nov. 2010.

[15] P. Rogaway and T. Shrimpton, "Deterministic authenticated encryption: A provable-security treatment of the keywrap problem," in *Advances in Cryptology — EUROCRYPT'06*, ser. Lecture Notes in Computer Science, vol. 4004. Springer, 2006, pp. 373–390.

[16] IEEE 1619.3 Technical Committee, "IEEE storage standard 1619.3 (key management) (draft)," available from https://siswg.net/.

[17] OASIS Key Management Interoperability Protocol (KMIP) Technical Committee, "KMIP – key management interoperability protocol," available from http://xml.coverpages.org/KMIP/, february 2009.

[18] L. Noll and R. Lockhart, "Method and system for identifying and managing keys description/claims," U.S. Patent Application 20090092252.

[19] Quantum Corporation, "Cryptographic key management for stored data," U.S. Patent application 20080219449.

[20] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions in Information Theory*, vol. 2, no. 29, pp. 198–208, March 1983.

[21] S. Kremer, G. Steel, and B. Warinschi, "Security for key management interfaces," Laboratoire Spécification et Vérification, ENS-Cachan, France., Tech. Rep. 11-07, April 2011, available from http://www.lsv.ens-cachan.fr/Publis/.

[22] S. Fröschle and G. Steel, "Analysing PKCS#11 key management APIs with unbounded fresh data," in *Proc. Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, ser. Lecture Notes in Computer Science, vol. 5511. Springer, 2009, pp. 92–106.

[23] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of Computer and System Sciences*, vol. 28, pp. 270–299, April 1984.

[24] C. Dwork, M. Naor, O. Reingold, and L. Stockmeyer, "Magic functions," *Journal of the ACM*, vol. 50, no. 6, pp. 852–921, 2003.

[25] M. Bellare, A. Boldyreva, and S. Micali, "Public-key encryption in a multi-user setting: Security proofs and improvements," in *Advances in Cryptology — EUROCRYPT'00*. Springer, 2000, pp. 259–274.

[26] J. Black, P. Rogaway, and T. Shrimpton, "Encryption-scheme security in the presence of key-dependent messages," in *Poc. 9th Annual International Workshop on Selected Areas in Cryptography (SAC'02)*, vol. 2595. Springer, 2003, pp. 62–75.

[27] L. Mazaré and B. Warinschi, "Separating trace mapping and reactive simulatability soundness: The case of adaptive corruption," in *Proc. Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, ser. Lecture Notes in Computer Science, vol. 5511. Springer, 2009, pp. 193–210.

[28] D. Hofheinz and E. Kiltz, "Practical chosen ciphertext secure encryption from factoring," in *Advances in Cryptology — EUROCRYPT'09*, ser. Lecture Notes in Computer Science, vol. 5479, 2009, pp. 313–332.

[29] S. Panjwani, "Tackling adaptive corruptions in multicast encryption protocols," in *Proceed. 4th Theory of Cryptography ConferenceTheory of Cryptography Conference (TCC'07)*, ser. Lecture Notes in Computer Science, vol. 4392. Springer, 2007, pp. 21–40.

[30] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols," in *Proc. 1st ACM Conference on Computer and Communications Security (CCS'93)*. ACM, 1993, pp. 62–73.

[31] R. Küsters and M. Tuengerthal, "Universally composable symmetric encryption," in *Proc. 22nd IEEE Computer Security Foundations Symposium (CSF'09)*. IEEE Computer Society, 2009, pp. 293–307.

[32] M. Backes and B. Pfitzmann, "Symmetric encryption in a simulatable Dolev-Yao style cryptographic library," in *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW'04)*. IEEE Computer Society Press, 2004, pp. 204–218.