Analysis of Key Wrapping APIs: Generic Policies, Computational Security

Guillaume Scerri
University of Bristol
Bristol, United Kingdom
Email: guillaume.scerri@bristol.ac.uk

Ryan Stanley-Oakes
University of Bristol
Bristol, United Kingdom
Email: ryan.stanley@bristol.ac.uk

***Abstract*—We present an analysis of key wrapping APIs with generic policies. We prove that certain minimal conditions on policies are sufficient for keys to be indistinguishable from random in any execution of an API.**

Our result captures a large class of API policies, including both the hierarchies on keys that are common in the scientific literature and the non-linear dependencies on keys used in PKCS#11. Indeed, we use our result to propose a secure refinement of PKCS#11, assuming that the attributes of keys are transmitted as authenticated associated data when wrapping and that there is an enforced separation between keys used for wrapping and keys used for other cryptographic purposes.

We use the Computationally Complete Symbolic Attacker developed by Bana and Comon. This model enables us to obtain computational guarantees using a simple proof with a high degree of modularity.

I. Introduction

A cryptographic API is an interface between a user and some trusted hardware, such as an HSM or a cryptographic token. These are deployed in insecure environments where the user's machine (or the user herself) may be compromised. The cryptographic operations are performed inside the trusted hardware, keeping sensitive data such as secret keys out of reach of the host machine. One widely-used cryptographic API is PKCS#11, which is described in a cryptographic standards document consisting of hundreds of pages [OAS15].

Many cryptographic APIs, including PKCS#11, allow key management commands such as key wrapping, the encryption of one key under another to facilitate secure key transport. Unfortunately, numerous key recovery attacks on PKCS#11 have been found, many of which exploit key wrapping commands that are explicitly authorised by the policy in the standard [Clu03], [DKS08], [DKS10]. Indeed, these attacks have been replicated on real cryptographic tokens that implement PKCS#11 [BCFS10].

Key management APIs are widely deployed, but the attacks on PKCS#11 demonstrate the dangers of not rigorously defining an API and its security properties. This led to the development of a number of formal models for key management APIs that have appeared in the scientific literature. However, these existing models have a number of important shortcomings that we describe below.

One weakness of the existing work is that proofs of API security typically depend on a particular choice of API security policy. A key management API policy specifies how keys can be used. For example, the policy should determine which keys may be used to wrap a particular key. Many of the attacks on PKCS#11 exploit weaknesses in the policy. For example, the PKCS#11 policy allows keys to be used for both wrapping other keys and decrypting data. Therefore, an adversary can request the decryption of a wrap and receive a sensitive key in the clear. Nevertheless, there has so far been no *generic* analysis of key management policies themselves, where one gives conditions on the policy, rather than describing it in full, and proves that these conditions are sufficient to prevent attacks. Instead, previous security proofs for API designs have specified a particular policy for their particular design and proved some security property of the API with this policy [CC09], [CS09], [KSW11]. There have been type-based analyses of APIs that allow for more general statements, but these works only prove security against key recovery (not that keys are indistinguishable from random) and do not consider the effect of key corruption [CFL13], [AFL13].

Not analysing generic policies is an important gap in the literature: poor policy design results in devastating attacks that are easy to implement, but there are no general results on what constitutes a valid policy. Proprietary APIs are likely to implement their own policies, not the particular ones that have been used in API designs in the academic literature. Providing a generic framework for evaluating policies gives the ability to analyse both the API designs already in use and any future designs that might appear.

A different kind of policy, not considered at all in previous analyses of key management APIs, is a high-level key management policy, independent of the choice of key management API, that must be enforced by any API. For example, consider the use of cryptographic tokens in a large enterprise, where employees have a position in a hierarchy. Suppose the enterprise sees employees near the bottom of the hierarchy as more likely to be compromised than those higher up, so the high-level key management policy on keys is simply that the compromise of keys for employees lower down the hierarchy should not compromise keys higher up. To enforce this, the API policy would need to prevent keys of lower-ranked employees being used to wrap keys of higher-ranked employees. We call the high-level key management policy the *enterprise security policy* to reflect this example,

but in general an enterprise policy could be a complicated, non-linear relationship between different components of the organisation[1]. Since previous analyses in the literature make no link between API policies and enterprise policies, it is unclear how a security property proved for a key management API is sufficient for the security needs of the users of the API.

Typically, analyses of key management APIs are strongly tied to specific assumptions about cryptographic mechanisms, and the security proofs only hold under these assumptions. For example, while Kremer, Steel and Warinschi give an API design with strong computational security guarantees, their proof relies on the assumption that key wrapping is deterministic and that wraps are unforgeable and indistinguishable from random strings [KSW11]. It is not clear how making different assumptions on the wrapping mechanism would affect the proof. The same can be said for the API design by Cachin and Chandran, whose security definition, and hence the proof of security, is very closely tied to the design of the API and the specific cryptographic security notions assumed for the primitives used by the API [CC09]. Kremer, Künnemann and Steel also make restrictive assumptions about the wrapping primitive used by their key management functionality, namely that it is deterministic and has Key-Dependent Message (KDM) security [KKS13]. All of these results only verify the security of a particular key wrapping API with a particular policy and a particular wrapping mechanism. Additionally, the proof in the paper by Kremer, Künnemann and Steel is in the GNUC framework. This means that the security notion used for key management should compose naturally with the security of other functionalities offered by a cryptographic API, but this composability comes at the cost of having a complex model and a proof that is hard to verify.

Some APIs cThe Cortier-Steel and Cortier-Steel-Wiedling APIs allow arbitrary terms to be processed by the API and only have one type of encryption for, in particular, both keys and data [CS09], [CSW12]. As input, their APIs can take a term representing the encryption of a key, together with some data encrypted under another key. This is intended for running complex cryptographic protocols without ever having to give the value of keys to an untrusted machine. This is a desirable design for cryptographic APIs, however it is far from what is implemented in practice (in particular, what is implemented in PKCS#11).

The Cortier-Steel API uses a very rich policy that describes relationships between agents, but this policy is expressible using our simple definition of policy. The Cortier-Steel-Wiedling API additionally attaches timestamps to keys so that their validity can change over time. This allows for an API that recovers after corruption. We do not consider the issue of time here, and leave such an extension as future work. Finally, both these works perform their proofs in the Dolev-Yao model. In this respect the computational guarantees we give in this paper are strictly stronger.

We improve on existing work in a number of ways: we avoid specifying a particular API policy; we use an approach that is modular with respect to cryptographic assumptions; we prove strong, computational security guarantees with a proof that is easy to verify and we parameterise our definition of API security by the security needs of the users of the API. Ultimately, our result is significantly more general, and our proof is significantly more modular, than in previous analyses of key management APIs. We can verify the security, with respect to different enterprise security policies, of a wide variety of key wrapping APIs with different policies and different wrapping mechanisms.

In the next subsection we outline our results.

A. Our Contribution

We view a cryptographic API as the composition of a *key wrapping API* that carries out key wrapping, and an *external API* that carries out other cryptographic operations. All keys stored by the API are called using a public handle, and each handle has an associated *attribute* that is either *internal* or *external*, referring to the intended use of the key pointed to by the handle. The key wrapping API will reject calls to wrap under keys with external attributes, and the external API will reject calls to carry out the other cryptographic operations using keys with internal attributes.

Our security goal for a key wrapping API is that external keys are indistinguishable from freshly generated keys, even after being wrapped and unwrapped using internal keys. As we exemplify in Section VII, this notion of security for key wrapping APIs is sufficient to preserve the security of the other cryptographic operations carried out by the external API.

We remark that indistinguishability from random is impossible to achieve for any key that can be used for wrapping[2] so the key wrapping API will need to enforce the separation between internal and external keys. While we assume the API does not explicitly allow attributes to be changed from external to internal, it is possible that an adversary could implicitly change the attribute of a key by wrapping and unwrapping it, giving it a new handle and possibly a new attribute. It is therefore necessary to use a secure AE-AD scheme to wrap keys, so that the wrapping mechanism binds the attributes of the wrapped key to the ciphertext.

From now on, unless stated otherwise, we will use the term API to refer to the key wrapping API inside a larger cryptographic API, since this is the main focus of our analysis.

An API has a *security policy*, or simply *policy*, that determines whether one key can be used to wrap another. This policy is not simply how the API ought to behave, but rather the actual rules for wrapping implemented in the code of the API.

[1]As another example, suppose an organisation has a CEO (A), a CFO (B), an HR Manager (C) and a Product Designer (D). D is lower-ranked than B and C, who are equally-ranked and lower-ranked than A. If C's trusted hardware is compromised, and hence C's secret keys are compromised, then one might expect the company's employee data to be compromised, but not the company's financial data. Both (respectively, neither) sets of data are expected to be compromised if A (resp., D) is compromised.

[2]There is a trivial distinguishing attack in this case: the distinguisher creates a wrap under the true key, then attempts to decrypt this wrap using its challenge key.

We analyse a *generic* key wrapping API in which the security policy is not fixed. That is, rather than choosing a particular policy as is done elsewhere in the literature, we leave it underspecified and find conditions on the policy that are sufficient for secure key management. While previous works conflate two roles of policies - defining the actual internal *behaviour* of the API and defining what properties are *expected* from the API - we clearly separate these roles into the API security policy and the enterprise policy, respectively, and determine when the former is sufficient to satisfy the latter.

Furthermore, our API design does not insist on a particular wrapping mechanism. Unlike other API designs, the wrapping mechanism can be implemented in a variety of ways without affecting our security theorem or its proof - in particular, both deterministic and randomised wrapping mechanisms are supported. As with the policy, we specify conditions on the security of the mechanism, capturing confidentiality and integrity, that are sufficient for the security of the API.

Our approach gives strong, computational guarantees but with a simple proof in the symbolic model that holds under a number of different computational assumptions.

Now we detail some of the key technical aspects of our result.

STRONG SECURITY GUARANTEES. We prove that certain minimal conditions on the policy and the wrapping mechanism are sufficient to guarantee the secrecy of external keys. We prove security in a strong, cryptographic sense, namely that external keys are indistinguishable from random keys, even in the presence of powerful Probabilistic Polynomial-time Turing machine (PPT) adversaries. We additionally prove that composing a secure key wrapping API with a secure encryption scheme results in a secure cryptographic API where users are able to wrap keys and encrypt data. As an application, we propose a secure refinement of PKCS#11, forbidding certain attribute combinations and forcing the wrap mechanism to be a secure AE-AD scheme that correctly transmits the attributes of wrapped keys.[3]

GENERIC POLICIES. We assume the existence of an enterprise security policy and an API security policy but do not fix either one. The enterprise policy is specified as relationships between the attributes of keys. Intuitively, these relationships determine how the compromise of keys will propagate. Then we give conditions that say whether or not the security policy of an API is valid with respect to the enterprise security policy; simply separating internal and external keys is not enough for security. For example, suppose the enterprise security policy says that the key k_1 is of a higher security level than k_2. Then a valid API security policy will not allow k_1 to be wrapped under k_2 since, if k_2 is compromised, a wrap of k_1 under k_2 is insecure, leading to the trivial compromise of k_1.

MODULAR PROOF. We use the Computationally-Complete Symbolic Attacker for equivalence properties (CCSA), developed by Bana and Comon in 2014 [BC14]. This model uses symbolic formalism to prove meaningful, computational security statements. Proofs in this model are easy to verify, especially compared to the cryptographic proofs like those found in [CC09], [KSW11] and [KKS13]. In CCSA one expresses assumptions (on the policy and the wrapping mechanism) as axioms in first-order logic and deduces the security property from these axioms. This means that the theorem holds under any computational assumptions such that the axioms hold, giving a high degree of modularity.

Informally, our cryptographic axioms say that genuine wraps of keys are indistinguishable from wraps of fresh, random keys and that wraps are unforgeable. We prove that our axioms are sound if the wrapping scheme is built on an Authenticated Encryption with Associated Data (AE-AD) encryption scheme; this is a standard requirement in modern, symmetric cryptography. However, we also prove that the axioms are sound when wrapping satisfies a *deterministic* variant of this notion (i.e. not using random nonces). Since our proof depends only on the axioms, not on the computational assumptions, we can obtain results about APIs that use completely different wrapping mechanisms without any changes to the proof. One of our conditions for an API policy to be valid is that it forbids the creation of key cycles. However, if we assume KDM security for the wrapping scheme as in [KKS13], then we can relax this condition on the policy (see Remark 5). This is in contrast to existing works, where relaxing the assumption on key cycles or the assumption that encryption is randomised would require substantial changes to the security proof.

AUGMENTING THE CCSA FRAMEWORK. In previous works, CCSA has only been applied to cryptographic protocols. Ours is the first use of CCSA to analyse a system that is outside the scope of the original paper on CCSA. To prove our substantial result, we required novel, computationally-sound axioms. One such new axiom captures how a proof can be split into disjoint cases. Our new axioms can be used in security proofs outside the key management setting without needing to reprove their computational soundness. In this respect we have augmented the CCSA model.

Another recent work has developed CCSA axioms for various cryptographic primitives in order to tackle a larger class of cryptographic protocols [BC16]. Their work independently developed axioms for manipulating branching and 'if' statements in the CCSA model; most of the core axioms related to these constructions are similar in our work and in theirs. The main difference between our core axioms and those developed in [BC16] is that our case disjunction axiom is strictly stronger than theirs, at the cost of a more involved soundness proof. Our stronger case disjunction axiom is required to deal with the very rich branching that arises in the context of key-management.

CAVEATS. We prove security when the number of API queries made by the adversary is arbitrary but, because of our reliance on CCSA, this number must be independent of the security parameter used by the underlying cryptographic primitives. Nevertheless, all of the attacks found on real APIs

[3]We are aware that this is likely to be far from what is currently implemented on real-world cryptographic tokens running PKCS#11-compliant APIs. Our result should be viewed simply as guidance for future token designs.

either use a fixed number of queries, as captured by our model, or are attacks on weak cryptographic primitives (such as in [BFK+12]) for which our axioms would not be sound.

We accept that our separation of external and internal keys is difficult to enforce in practice; in particular, the attributes of keys must be securely bound to the keys when wrapping (such as with AE-AD) so that external and internal keys can never be confused. However, this separation enforces a standard industry practice, as recommended by NIST: "a single key should be used for only one purpose (e.g., encryption, authentication, key wrapping, random number generation, or digital signatures)" [NIS12]. Indeed, most of the API designs in the literature enforce this separation [CC09], [KSW11], [CS09], [CSW12].

We remark that our composition theorem in Section VII only applies to cryptographic APIs with wrap, unwrap, encrypt, decrypt and corrupt actions, rather than arbitrary cryptographic primitives. However, the proof is largely independent of the specific cryptographic game for encryption and would therefore be easy to adapt to other primitives.

II. Key Wrapping APIs

In this Section we define the execution model for a key wrapping API and the security one should expect of such an API.

A. Execution Model

We assume the existence of the sets $\mathcal{K}$ of *keys*, $\mathcal{H}$ of *handles* and $\mathcal{D}$ of *attributes* (data). The set $\mathcal{D}$ has a particular subset $\mathcal{E}$ of *external* attributes. A *wrapping mechanism* $\mathtt{wm}$ consists of the triple $(\mathtt{keygen}, \mathtt{wrap}, \mathtt{unwrap})$ of algorithms. The algorithm $\mathtt{keygen}$ takes a security parameter η (in unary) as input and returns an element of $\mathcal{K}$. We assume that $\mathtt{keygen}\,(1^\eta)$ always returns a key of length $\mathtt{keylen}\,(1^\eta)$. The algorithm $\mathtt{wrap}$ takes as input a key $k \in \mathcal{K}$, an attribute $a \in \mathcal{D}$ and a second key $k' \in \mathcal{K}$, and returns $\mathtt{wrap}\,(k, a, k')$, the *wrap packet* of the key k with attribute a under the key k'. We do not specify how the wrap packet depends on the attribute of the wrapped key, nor how it uses any randomness. The algorithm $\mathtt{unwrap}$ takes a wrap packet and a key as input and returns a key and an attribute. Both $\mathtt{wrap}$ and $\mathtt{unwrap}$ can have access to the security parameter, if necessary. We assume that all wrapping mechanisms are *correct*, that is, for all keys $k, k' \in \mathcal{K}$ and all attributes $a \in \mathcal{D}$, $\mathtt{unwrap}\,(\mathtt{wrap}\,(k, a, k')\,, k') = (k, a)$.

A *security policy* $\mathtt{P}$ is an algorithm that takes two attributes as input and returns a bit. We make this choice since a policy ought not to depend on the values of the keys themselves (otherwise it could leak unintended information about the keys and would have to be evaluated by the secure hardware, not the API). We remark that a more general wrapping policy could use the value of a global clock as an additional argument (to handle key lifecycles and key revocation, etc.), but we leave this for future work.

Definition 1. A *key wrapping API* $\mathtt{API}$ is a program parameterised by the tuple $(\mathcal{K}, \mathcal{D}, \mathcal{E}, \mathcal{H}, \mathtt{wm}, \mathtt{P})$. The API maintains, in its state $\mathbf{st}$, a map $\mathbf{st}.\mathbf{val} : \mathcal{H} \to (\mathcal{K} \cup \{\lightning\}) \times (\mathcal{D} \cup \{\lightning\})$ that records the association between handles, which are public names, and pairs consisting of a key and its attribute. In what follows, we specify how an adversary interacts with an API.

An adversary, interacting with the API, can request wraps and unwraps via the handles of keys. If $\mathbf{st}.\mathbf{val}\,(h) = (\lightning, \lightning)$ then we say h is unused in state $\mathbf{st}$. There is also a function $\mathbf{freshhdl}$ that takes a state $\mathbf{st}$ as input and returns a handle $\mathbf{freshhdl}\,(\mathbf{st})$ that is unused in $\mathbf{st}$.

Let $\mathbf{st}_0$ be an initial state of the API. Then the initial configuration of the API is the map $\mathbf{val}$ encoded by $\mathbf{st}_0$. An initial configuration is called *honest* if, for all handles h, $\mathbf{st}_0.\mathbf{val}\,(h)$ is either not initialised or initialised with an output of $\mathtt{keygen}(1^\eta)$. Intuitively, this corresponds to a device with honestly generated keys already present. From now on, all key wrapping APIs are assumed to have honest initial configurations. This assumption is reflected in the generation of the initial state of APIs in the experiment used to define security.

Note that we assume that each key has a single attribute. We will also assume the API does not explicitly allow the user to change the attribute of a key. These assumptions are without loss of generality, since any attributes that can be easily changed by the adversary cannot be used to preserve meaningful security and so are omitted from this discussion. Note that any "attributes" irrelevant to security can be encoded in the state and we simply do not allow the policy to depend on these. Furthermore, if a key could be generated without an attribute, then the attribute would need to be fixed before the key could be used and so we merely assume for convenience that this decision takes place before the key is generated.

Intuitively, if a key's attribute belongs to $\mathcal{E}$ then that key is intended to protect data, as opposed to other keys. Our security goal will be that external keys (keys with attributes in $\mathcal{E}$) remain indistinguishable from random after being managed by the API and therefore are ideal for cryptographic use. Note that this goal is impossible to achieve for any keys used for key wrapping, since an adversary can distinguish the real key from a random key by attempting to unwrap with its test key.

Even though separating external and wrapping keys might seem restrictive, as soon as a wrapping key can also be used for decryption, keys can be recovered by the adversary (as demonstrated in [BCFS10]). Therefore such a separation must be enforced by any secure policy. While we only consider here the wrapping module used by cryptographic APIs, in Section VII we give a formal argument that a secure key wrapping API can be composed with an encryption scheme without undermining the security of the encryption scheme, assuming this separation of key roles.

There are three actions that a user can perform in its interaction with a key wrapping API: wrapping, unwrapping and corruption. Obviously, the corruption action is not intended to be implemented on real APIs, but is used here to reason about the security of APIs in the presence of an adversary who can obtain the values of particular keys, for example through

side-channel attacks.

When a key is corrupted, we add its attribute to a list of corrupted attributes. We settle for recording corrupted attributes rather than corrupted keys as, by construction, every key with the same attribute has the same capabilities. In other words, if one corrupts a key with attribute a, every key that can be wrapped by a key with attribute a will be compromised, irrespective of exactly which key was corrupted. It is therefore enough to log that *some* key with attribute a has been corrupted. As a consequence, if one wants to distinguish the corruption of two keys, these keys should be given different attributes.

A *fully-specified action* is an action (wrap, unwrap or corrupt), together with the handles of the keys relevant to the action. Fully-specified actions define the execution of the API. Formally, if $\mathbf{A}$ is a fully-specified action of the API, then $\mathbf{A}$ is an element of one of the sets $\{\mathbf{W}(h_1, h_2) \mid h_1, h_2 \in \mathcal{H}\}$, $\{\mathbf{U}(h) \mid h \in \mathcal{H}\}$ or $\{\mathbf{C}(h) \mid h \in \mathcal{H}\}$. Obviously the unwrap action will require an additional argument corresponding to the ciphertext to be unwrapped, but the value of this argument does not determine the overall structure of the API execution (as it could depend on some random coins) and so it is not considered part of the fully-specified action $\mathbf{U}(h)$.

- If $\mathbf{A} = \mathbf{W}(h_1, h_2)$, then the API computes $(k_1, a_1) \leftarrow \mathbf{st}.\mathbf{val}(h_1)$ and $(k_2, a_2) \leftarrow \mathbf{st}.\mathbf{val}(h_2)$. If $k_1, k_2 \neq \lightning$ and $\mathtt{P}(a_1, a_2) = 1$, then the API returns $\mathtt{wrap}(k_1, a_1, k_2)$. Otherwise, the API returns $\lightning$.
- If $\mathbf{A} = \mathbf{U}(h_1)$, then the API takes an additional input x from the user, computes $(k_1, a_1) \leftarrow \mathbf{st}.\mathbf{val}(h_1)$ and $(k_2, a_2) \leftarrow \mathtt{unwrap}(x, k_1)$. If $k_2, a_2 \neq \lightning$ and $\mathtt{P}(a_2, a_1) = 1$, then $\mathbf{st}.\mathbf{val}(\mathbf{freshhdl}(\mathbf{st})) \leftarrow (k_2, a_2)$. Otherwise, $\mathbf{st}.\mathbf{val}(\mathbf{freshhdl}(\mathbf{st})) \leftarrow (\lightning, \mathsf{bad})$ where bad is a particular attribute (used to denote handles that are not fresh, but do not point to a key). In either case, $\mathbf{freshhdl}(\mathbf{st})$ is returned to the user.
- If $\mathbf{A} = \mathbf{C}(h_1)$, then the user is asking to learn the value of the key pointed to by h_1. The API computes $(k_1, a_1) \leftarrow \mathbf{st}.\mathbf{val}(h)$, does not carry out any checks, and returns k_1. By definition, if h is unused in $\mathbf{st}$ then the API returns $\lightning$. For our security property, the state maintains a list of *corrupt attributes*. Therefore, for this action, the API updates its state as follows: $\mathbf{st}.\mathbf{cor} \leftarrow \{a_1\} \cup \mathbf{st}.\mathbf{cor}$.

B. Security

We consider three objectives of an adversary in its execution of a key wrapping API. First, an adversary may try to learn (part of) the value of an external key, in order to compromise a cryptographic scheme using this key in the wider cryptographic API. Second, the adversary may try to change the attributes of a key (external or otherwise), in order to circumvent the security policy. Third, the adversary may try to import its own keys in order to wrap with these keys. The second and third objectives can be combined: the adversary succeeds in either objective if there is a handle pointing to a key, attribute pair that was not pointed to by a handle in the initial configuration of the API (either because the key is new, or the attribute of an honestly generated key has changed).

We consider an API to be *secure* if and only if no adversary can achieve these objectives with *uncompromised* handles. That is, if a handle is uncompromised and it points to an external key, then that key cannot be distinguished from a random key; we call this the *key secrecy* property. Furthermore, any uncompromised handle must point to a key, attribute pair from the initial configuration of the API; we call this the *handle consistency* property.

What is meant by calling a handle compromised (or uncompromised) is that, for example, an organisation might consider keys assigned to lower-status employees compromised if the keys of senior management are compromised, but not vice-versa. This is an example of an *enterprise security policy*. So, given the list of attributes of keys lost via the corrupt action, the enterprise security policy determines which handles are compromised according to their attributes. We formalise this below.

In general, we model an enterprise security policy by a *sacrifice function* $\mathbf{sacr} : \mathcal{D} \to \mathcal{P}(\mathcal{D})$. Intuitively, the set $\mathbf{sacr}(a)$ should be thought of as the set of attributes of keys compromised as a result of compromising a key with attribute a. This has nothing to do with the use of the API; it is a pre-existing relationship between attributes that the API needs to respect, by not allowing the compromise of keys to propagate beyond what is specified by the sacrifice function.

We assume without loss of generality that $a \in \mathbf{sacr}(a)$ for all $a \in \mathcal{D}$ and that $\mathbf{sacr}$ is *transitive* in the following sense: if $a \in \mathbf{sacr}(b)$ and $b \in \mathbf{sacr}(c)$ then $a \in \mathbf{sacr}(c)$.

The sacrifice function is derived from the existing relationships between users of the API. The example enterprise security policy given above, where compromise propagates down the employee hierarchy (but not up), would be captured by giving an ordering $<$ on attributes and putting $\mathbf{sacr}(a) = \{a' \in \mathcal{D} \mid a' \leq a\}$ for all $a \in \mathcal{D}$. A more refined enterprise security policy would separate the organisation into departments and say that the compromise of keys must not propagate across different departments. In this case, one would partition the set of attributes into classes $(\mathcal{D}_i)$ and, for each i, put $\mathbf{sacr}(a) = \{a' \in \mathcal{D}_i \mid a' \leq a\}$ for all $a \in \mathcal{D}_i$.

There could be a very conservative enterprise security policy, where the compromise of a key does not compromise any other keys. In this case, each key would have a unique attribute and $\mathbf{sacr}(a) = \{a\}$ for each attribute a. We will see that the only APIs secure for this sacrifice function are ones that disable key wrapping altogether.

Now we formalise the notion of *compromise*, which follows easily from the sacrifice function and the list of corrupt attributes.

Definition 2. If $a \in \mathbf{st}.\mathbf{cor}$ then we say a is a *corrupt attribute* in state $\mathbf{st}$. If a is corrupt in state $\mathbf{st}$ and $a' \in \mathbf{sacr}(a)$, then we say a' is a *compromised attribute* in state $\mathbf{st}$.

If $\mathbf{st}.\mathbf{val}(h) = (k, a)$ and a is a corrupt (respectively, compromised) attribute in state $\mathbf{st}$, then we say h is a *corrupt*

handle (respectively, *compromised handle*) in state $\mathbf{st}$.

We have two security experiments, capturing the secrecy of uncompromised keys and the consistency of uncompromised attributes.

Our security experiments take an adversary $\mathcal{A}$, integers m and n and attributes $a_1, \ldots, a_n$ as input. The handles $h_1, \ldots, h_n$ are initialised with randomly generated keys and the attributes $a_1, \ldots, a_n$. The integer m is the exact number of oracle queries that $\mathcal{A}$ makes in either experiment.

In both security experiments, the adversary has access to three oracles capturing the wrap, unwrap and corrupt actions of the API. There is also a fourth *test* oracle, which may only be queried once, whose behaviour depends on a bit b. The adversary submits a handle and, if $b = 0$, the oracle returns the key pointed to by this handle. If $b = 1$, a new key is randomly generated and returned to the adversary. When performing this test query, the API checks that the attribute of the submitted handle is *external* and has not been compromised. After the test query has been made, the API checks that any corrupt query does not compromise the tested handle.

If the tested handle is compromised at the end of the key secrecy experiment, or its attribute is not external, then the output of this experiment is a random bit. Otherwise, the output of the experiment is the output of the adversary.

In the handle consistency experiment, the output of the experiment is 1 if, once the adversary has given its output, the state of the API contains an uncompromised handle that points to a key, attribute pair not in the initial configuration of the API. Otherwise, the output of the experiment is 0.

The formal description of the security experiments are given in Figures 1 and 2.

Oracle $\mathcal{O}^{\mathtt{wrap}}(x_1, x_2)$:
$\mathbf{st}.C \leftarrow \mathbf{st}.C + 1$
If $\mathbf{st}.C > m$ Return $\lightning$
$(k, a) \leftarrow \mathbf{st}.\mathbf{val}(x_1)$
$(k', a') \leftarrow \mathbf{st}.\mathbf{val}(x_2)$
If $k, k' \neq \lightning$ and $\mathtt{P}(a, a')$
  Return $\mathtt{wrap}(k, a, k')$
Return $\lightning$

Oracle $\mathcal{O}^{\mathtt{unwrap}}(x_1, x_2)$:
$\mathbf{st}.C \leftarrow \mathbf{st}.C + 1$
If $\mathbf{st}.C > m$ Return $\lightning$
$(k, a) \leftarrow \mathbf{st}.\mathbf{val}(x_1)$
$(k', a') \leftarrow \mathtt{unwrap}(x_2, k)$
$h \leftarrow \mathbf{freshhdl}(\mathbf{st})$
$\mathbf{st}.H \leftarrow \mathbf{st}.H \cup \{h\}$
If $k' \neq \lightning$ and $\mathtt{P}(a', a)$
  $\mathbf{st}.\mathbf{val}(h) \leftarrow (k', a')$
Else $\mathbf{st}.\mathbf{val}(h) \leftarrow (\lightning, \mathsf{bad})$
Return h

Oracle $\mathcal{O}^{\mathtt{corrupt}}(x_1)$:
$\mathbf{st}.C \leftarrow \mathbf{st}.C + 1$
If $\mathbf{st}.C > m$ Return $\lightning$
$(k, a) \leftarrow \mathbf{st}.\mathbf{val}(x_1)$
If $\mathbf{st}.T \in \mathbf{sacr}(a)$
  $\mathbf{st}.V \leftarrow \bot$
$\mathbf{st}.\mathbf{cor} \leftarrow \{a\} \cup \mathbf{st}.\mathbf{cor}$
Return k

Oracle $\mathcal{O}_b^{\mathtt{test}}(x_1)$:
If $\mathbf{st}.C > m$ or $\mathbf{st}.T \neq \bot$ Return $\lightning$
$(k, a) \leftarrow \mathbf{st}.\mathbf{val}(x_1)$
$k' \leftarrow\$ \mathtt{keygen}(1^\eta)$
If $a \notin \mathcal{E}$ then $\mathbf{st}.V \leftarrow \bot$
For $c \in \mathbf{st}.\mathbf{cor}$
  If $a \in \mathbf{sacr}(c)$ then $\mathbf{st}.V \leftarrow \bot$
$\mathbf{st}.T \leftarrow a$
If $b = 0$ Return k
Return k'

Figure 1. Wrap, Unwrap, Corrupt and Test Oracles

Definition 3. Let $\mathtt{API} = (\mathcal{K}, \mathcal{D}, \mathcal{E}, \mathcal{H}, \mathtt{wm}, \mathtt{P})$. We say the $\mathtt{API}$ is secure if, for all integers m and n, all $\vec{a} \in \mathcal{D}^n$ and all polynomial-time adversaries $\mathcal{A}$, the following advantages are both negligible functions of η:

$$\mathrm{Adv}^{\mathrm{KEYSEC}}_{\mathtt{API}^m}(\mathcal{A}) := \mathbb{P}_b\left[\mathrm{Exp}^{\mathrm{KEYSEC}}_b(\mathcal{A}, \vec{a}, m) = b\right] - \frac{1}{2}$$

Experiment $\mathrm{Exp}^{\mathrm{KEYSEC}}_b(\mathcal{A}, a_1, \ldots, a_n, m)$:
$\mathbf{st}.C \leftarrow 0$
$\mathbf{st}.T \leftarrow \bot$
$\mathbf{st}.V \leftarrow \top$
$\mathbf{st}.\mathbf{cor} \leftarrow []$
For $1 \leq i \leq n$
  $k_i \leftarrow\$ \mathtt{keygen}(1^\eta)$
  $\mathbf{st}.\mathbf{val}(h_i) \leftarrow (k_i, a_i)$
$\mathbf{st}.H \leftarrow \{h_1, \ldots, h_n\}$
$\mathcal{O}_b \leftarrow (\mathcal{O}^{\mathtt{wrap}}, \mathcal{O}^{\mathtt{unwrap}}, \mathcal{O}^{\mathtt{corrupt}}, \mathcal{O}_b^{\mathtt{test}})$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_b}(1^\eta)$
If $\mathbf{st}.V = \top$ Return b'
$b'' \leftarrow\$ \{0, 1\}$
Return b''

Experiment $\mathrm{Exp}^{\mathrm{HDLCON}}(\mathcal{A}, a_1, \ldots, a_n, m)$:
$b \leftarrow 1$
$\mathbf{st}.C \leftarrow 0$
$\mathbf{st}.T \leftarrow \bot$
$\mathbf{st}.V \leftarrow \top$
$\mathbf{st}.\mathbf{cor} \leftarrow []$
$\mathbf{st}.\mathbf{val}_0 \leftarrow []$
For $1 \leq i \leq n$
  $k_i \leftarrow\$ \mathtt{keygen}(1^\eta)$
  $\mathbf{st}.\mathbf{val}(h_i) \leftarrow (k_i, a_i)$
  $\mathbf{st}.\mathbf{val}_0 \leftarrow \mathbf{st}.\mathbf{val}_0 \cup \{(k_i, a_i)\}$
$\mathbf{st}.H \leftarrow \{h_1, \ldots, h_n\}$
$\mathcal{O}_b \leftarrow (\mathcal{O}^{\mathtt{wrap}}, \mathcal{O}^{\mathtt{unwrap}}, \mathcal{O}^{\mathtt{corrupt}}, \mathcal{O}_b^{\mathtt{test}})$
$x \leftarrow \mathcal{A}^{\mathcal{O}_b}(1^\eta)$
For $h \in \mathbf{st}.H$
  $(k, a) \leftarrow \mathbf{st}.\mathbf{val}(h)$
  If $(k, a) \notin \mathbf{st}.\mathbf{val}_0$
    If for all $a_c \in \mathbf{st}.\mathbf{cor}$, $a \notin \mathbf{sacr}(a_c)$
      Return 1
Return 0

Figure 2. Key Secrecy Experiment b and Handle Consistency Experiment

$$\mathrm{Adv}^{\mathrm{HDLCON}}_{\mathtt{API}^m}(\mathcal{A}) := \mathbb{P}\left[\mathrm{Exp}^{\mathrm{HDLCON}}(\mathcal{A}, \vec{a}, m) = 1\right]$$

III. Statement of the Main Theorem

In this Section we define the class of *valid* API security policies, recall the definition of a secure AE-AD scheme and state the main theorem of the paper: if a key wrapping API uses a secure AE-AD scheme for its wrapping mechanism and its security policy is valid, then the API is secure.

There are three conditions required of the API policy. First, we require that if a key with attribute a can be wrapped under a key with attribute a', then the enterprise security policy should say that the compromise of a key with attribute a' compromises keys with attribute a. This is a reasonable assumption as any wrap under a key with attribute a' will no longer provide any security for the key with attribute a.

Second, we require that the policy forbids wrapping under external keys. This enforces the separation between keys used by the API and keys protected by the API.

Finally, we require that valid policies forbid the creation of key cycles. This is necessary, since standard security notions for encryption do not imply security in the presence of key cycles (see e.g. [CGH12]).

Definition 4. Let $\mathtt{P}$ be an API security policy. We say $\mathtt{P}$ is *valid* with respect to the sacrifice function $\mathbf{sacr}$ if:

- For all $a, a' \in \mathcal{D}$, if $\mathtt{P}(a, a') = 1$, then $a \in \mathbf{sacr}(a')$.
- For all $a \in \mathcal{D}$ and $e \in \mathcal{E}$, $\mathtt{P}(a, e) = 0$.
- The *policy graph*, i.e. the graph on $\mathcal{D}$ where there is an edge $a \to a'$ if and only if $\mathtt{P}(a', a) = 1$ (if and only if

a key with attribute a can be used to wrap a key with attribute a'), is acyclic.

Remark 5. This definition of a valid policy is minimal since, if any of the conditions are not met, then attacks are possible:

1) If the first condition is false, then it is possible to create a wrap of a key k such that corrupting the wrapping key does not compromise the handle pointing to k. So k is an uncompromised key (according to the enterprise policy) that is trivially distinguishable from random.
2) If the second condition is false, then there is an external key k such that the adversary can create a wrap under k and then try to decrypt the wrap with its challenge key.
3) The third condition is necessary since standard notions of encryption security do not imply security in the presence of key cycles. We could remove this condition and show that the key wrapping APIs are still secure, if we assume KDM security for the wrapping mechanism. In this case, we would simply weaken the constraint on the secrecy axiom to permit key cycles (see Section V), prove the soundness of this axiom (by slightly modifying the soundness proof of the original axiom) and then the proof of the symbolic security property from the axioms (Section VI) would need only minor changes to accommodate the new constraint.

Even though the above definition captures a large class of sensible security policies, it is all we need in order to prove that a key wrapping API is a secure API, provided that the wrapping mechanism is built using either a (randomised) *secure AE-AD scheme*, as defined in [Rog02] or a Deterministic Authenticated Encryption scheme as in [RS06]. We refer to these as randomised and deterministic AE-AD schemes, respectively. We recall these security notions here:

A triple $\Pi = (\mathtt{keygen}, \mathtt{enc}, \mathtt{dec})$ is called a randomised (resp. deterministic) *Secure Authenticated Encryption with Associated Data (AE-AD) scheme* if:

- Π is *correct*, i.e. for all messages m, data a, nonces r and all $k \leftarrow_{\$} \mathtt{keygen}\,(1^{\eta})$, $\mathtt{dec}\,(\mathtt{enc}\,(m, a, r, k), a, r, k) = m$ (where nonces are ignored in the deterministic case),
- Π is *private*, i.e. no PPT adversary can distinguish between an encryption oracle and an oracle returning random strings of the appropriate length,
- Π is *authenticated*, i.e. given access to an encryption oracle, no PPT adversary may produce triples (c, a, r) (or pairs (c, a) in the deterministic case) that decrypt successfully but where the c was not output by the encryption oracle.

Full definitions are provided in the long version of the paper [SSO16].

Now we state the main Theorem of this paper.

Theorem 1. *Let* $\mathtt{API} = (\mathcal{K}, \mathcal{D}, \mathcal{E}, \mathcal{H}, \mathtt{wm}, \mathtt{P})$ *be a key wrapping API and let* $\Pi = (\mathtt{keygen}, \mathtt{enc}, \mathtt{dec})$ *be a secure deterministic or randomised AE-AD scheme. Suppose* $\mathtt{wm} = (\mathtt{keygen}, \mathtt{wrap}, \mathtt{unwrap})$ *where* `wrap`*, on input* (k, a, k') *generates a fresh nonce* r *and returns the wrap packet* $(\mathtt{enc}\,(k, a, r, k'), a, r)$ *and* `unwrap`*, on input* (w, k)*, parses* w *as* (c, a, r) *and returns* $\mathtt{dec}\,(c, a, r, k)$*. Then, if* `P` *is valid with respect to* **sacr***, the key wrapping API is secure.*

Remark 6. The hypotheses of this theorem are minimal. By Remark 5, a valid policy is necessary to prevent attacks. Furthermore, without assuming AE-AD security for the wrapping mechanism, or some other way of securely binding attributes to keys when wrapping, the first two attacks given in Remark 5 are still possible: the adversary simply wraps a key and unwraps it with new attributes, circumventing any restrictions given by the policy.

In order to prove Theorem 1, we define *symbolic* APIs, using the language of CCSA, in Section IV. We give axioms in Section V and prove these axioms sound under the assumptions of Theorem 1. In particular we give axioms that are sound for both secure deterministic and randomised AE-AD schemes. Therefore, due to the computational soundness of CCSA, if one can show that a symbolic security property is implied by the axioms, then the corresponding computational security property holds in any computational API where the wrapping mechanism is as in Theorem 1.

In Section VI, we show that the axioms entail the symbolic security properties corresponding to the security definition given in Definition 3. This proves Theorem 1.

IV. Symbolic Model and Soundness

To prove Theorem 1, we cast our problem in the setting of the computationally complete symbolic attacker model (CCSA) from [BC14]. The CCSA allows one to abstract away a lot of the complexity of the usual computational models, yet it entails computational guarantees.

In CCSA, we model the capabilities of an adversary as a list of first-order axioms that cannot be broken by any PPT adversary with non-negligible probability. For example, a secrecy property for encryption could be expressed as the formula $\{x\}_k \sim \{y\}_k$, meaning that no PPT adversary can distinguish an encryption of x from an encryption of y. This is in contrast to Dolev-Yao style symbolic models where one has to fully specify the abilities of the attacker as a finite list of inference rules.

A security property in CCSA is expressed as the indistinguishability of two lists of terms, which is itself written as a formula in first-order logic. For example, if ϕ is the list of terms output to the adversary by the API where h points to (k, a) in the initial state, and ϕ' is where h points to (k', a), then the key secrecy property could be written $\phi \sim \phi'$[4]. One then proves that the axioms entail the security property. In other words, an adversary can perform any action that does not contradict the axioms, but still cannot break the security property.

Assigning a Turing Machine to each of the function symbols used in a symbolic API is called a *computational interpretation* of the symbolic API. For example, the function

[4]In fact, our key secrecy property is more complicated than this, since we need to specify that h is uncompromised.

symbol $\{\_\}$ could be interpreted as the encryption algorithm enc. A *computational model* of the symbolic API is a computational interpretation where certain function symbols have fixed interpretations. For example, in a computational model, the function symbol EQ is interpreted as the machine that outputs 1 if its two inputs are equal (and outputs 0 otherwise). Crucially, the predicate symbol $\sim$ is interpreted as computational indistinguishability of bitstrings. That is, the formula $x_1, \ldots, x_n \sim y_1, \ldots, y_n$ is interpreted as the statement that no polynomial-time adversary can distinguish the interpretations of $x_1, \ldots, x_n$ from the interpretations of $y_1, \ldots, y_n$.

In a computational model, each axiom is interpreted as a statement about Turing Machines. If the statement is true, then we say the axiom is *sound* in this model. Let A be a set of axioms. The computational soundness theorem for CCSA says that if $A \to \phi \sim \phi'$ then, in any computational model where the axioms in A are sound, the computational interpretations of the terms ϕ and ϕ' are indistinguishable: no polynomial-time adversary can distinguish them. In an API, the sequences of terms ϕ and ϕ' output by the API will depend on the inputs of the (active) distinguishing adversary, and these inputs are modelled using free function symbols g_i; one must prove that $\phi \sim \phi'$ regardless of the interpretation of the symbols g_i.

The axiomatic approach of CCSA gives an inherent modularity to the model. In our case, we are able to specify only the properties of the API security policy that are necessary, rather than describing the whole policy, since these properties correspond to the axioms used in the proof. Thus, at no extra cost, we prove the security of multiple instantiations of key wrapping APIs.

A. Symbolic Execution

In CCSA, protocols[5] have a fixed structure: it is known *a priori* what messages are expected at what stage (and therefore what checks to carry out before returning a message, and so on). In our computational API, even though the number of oracle queries is constant in the security parameter, the adversary may query its oracles in any order. Each possible order of queries induces a different sequence of output terms from the symbolic API. The order does matter, since the inputs from the adversary (in particular, the inputs to the unwrap oracle) depend on the previous outputs seen by the adversary so far. Therefore we have to fix a sequence of oracle queries, obtaining a *restricted computational API*, and reason about the output terms of the symbolic API corresponding to this restricted computational API. Fixing a list of actions is not a significant restriction; we show in the long version of the paper [SSO16] how the computational security definition given in Section II reduces to the security of every possible restricted computational API.

[5]The authors of [BC14] use the term protocols, but we will use their model to describe APIs. Having fixed a list of API oracle queries, we obtain a transition system that behaves like a protocol as described in [BC14]. Therefore we use 'protocol' and 'API' interchangeably.

We now give the language needed to describe the symbolic API.

Terms are built over the sets $\mathcal{F}$ of function symbols, $\mathcal{G}$ of free adversarial function symbols and $\mathcal{N}$ of names. The set of all terms is written $\mathcal{T} := \mathcal{T}(\mathcal{F}, \mathcal{G}, \mathcal{N})$. Syntactic equality of the terms t and t' is written $t \equiv t'$. Every term has a sort. Within the sort message, there are subsorts key, nonce, longnonce, handle, attribute and bool. Names have sort key, nonce, longnonce, handle or attribute. There is a particular name bad of sort attribute. The set $\mathcal{F}$ is described in Figure 3.

- Constants: true, false, fail
- Conditional branching: if _ then _ else _
- Logical operators: $\_ \wedge \_$, $\_ \vee \_$, $\neg\_$
- Pairing and projections: $(\_,\_)$, $\pi_1(\_)$, $\pi_2(\_)$, $\pi_3(\_)$
- Encryption and decryption: $\mathsf{enc}(\_,\_,\_,\_)$, $\mathsf{dec}(\_,\_,\_,\_)$
- Equality test: $\mathsf{wf}(\_)$, $\mathsf{EQ}(\_,\_)$
- Random string: $\$(\_,\_)$
- Policy check: $\mathsf{P}(\_,\_)$

Figure 3. Functions

We remark that certain function symbols above, e.g. the boolean operators and the projections have obvious intended meanings. We formalise these intended meanings either as fixed computational interpretations in Section IV-C, or give assumptions on these interpretations, which are reflected as axioms, in Section V. This means that for functions like encryption that have more than one natural implementation, we permit any implementation satisfying the assumptions.

We introduce some abbreviations for ease of notation. We will write $[b]x : y$ for $(\mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ y)$ and $[b]x$ for $(\mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{fail})$. This notation is extended to lists of terms in the natural way: if $\vec{v} \equiv v_1, \ldots, v_n$ and $\vec{w} \equiv w_1, \ldots, w_n$, then $[b]\vec{v} : \vec{w} \equiv [b]v_1 : w_1, \ldots, [b]v_n : w_n$.

We write $\{m\}^r_{k,a} \equiv (\mathsf{enc}(m, a, r, k), a, r)$. This is just for convenience: when unwrapping, the adversary must submit a ciphertext, some associated data and a nonce to the API and we prefer to express this triple as a single term called the *wrap packet*. In the case of deterministic encryption, the nonce will simply be ignored. Furthermore, we write

$$\mathsf{triple}(x) \equiv \mathsf{EQ}(x, (\pi_1(x), \pi_2(x), \pi_3(x)))$$
$$\mathsf{uwp}(x, k) \equiv [\mathsf{triple}(x)]\,\mathsf{dec}(\pi_1(x), \pi_2(x), \pi_3(x), k)$$

This is so, when the adversary submits a wrap packet to uwp, one checks that this packet is a triple (x', x'', x''') and, if so, its components are taken for the inputs to dec. We use $\mathsf{wf}(x) \equiv \neg\mathsf{EQ}(x, \mathsf{fail})$ to test whether terms (in particular, ciphertexts) are *well-formed*.

The set $\mathcal{G}$ consists of infinitely many function symbols $g_i : \mathsf{message}^i \to \mathsf{message}$ for each $i \geq 0$, corresponding to the computation of the adversary on the terms it has seen previously.

Subterms are defined naturally as the internal components of terms, and $\mathrm{st}(x)$ denotes the set of subterms of x. *Positions* of subterms within terms are defined as usual.

Formulae may be built over terms using predicate symbols. There is a predicate symbol $\sim^i$ for each natural number i;

each may be used between lists of i terms. We abbreviate $x_1, \ldots, x_i \sim^i y_1, \ldots, y_i$ by $x_1, \ldots, x_i \sim y_1, \ldots, y_i$. We use the symbols $\neg, \vee, \wedge, \rightarrow$ to construct compound formulae as normal, but these should not be confused with the *function symbols* used to operate on terms of sort bool. For example, $b \rightarrow b'$ is a term of sort bool, whereas $(b \sim b') \wedge (b' \sim b'') \rightarrow (b \sim b'')$ is a (compound) formula.

We assume that each state q of a symbolic API is labelled by the fully-specified actions used to reach it from the initial state q_0. Then, given an *initial configuration* $\mathbf{val}_{q_0} : \mathcal{H} \rightarrow (\mathcal{K} \cup \{\mathsf{fail}\}) \times (\mathcal{D} \cup \{\mathsf{fail}\})$, we can recover the function $\mathbf{val}_q : \mathcal{H} \rightarrow \mathsf{message}$ by modifying $\mathbf{val}_{q_0}$ in the manner described below. For convenience of notation, we write $\mathbf{k}_q(h)$ and $\mathbf{a}_q(h)$ for $\pi_1(\mathbf{val}_q(h))$ and $\pi_2(\mathbf{val}_q(h))$ respectively.

We remark that $\mathbf{val}_q$ is an inherent property of the state q of the API and *not* a function symbol to be interpreted in a model: $\mathbf{val}_q(h)$ will always be instantiated by a particular term of sort message, intended to be a key, attribute pair.

Finally we recall that, upon successfully unwrapping, the adversary is given a handle that points (via $\mathbf{val}$) to the result of the unwrap. The new handle is determined by a function $\mathbf{freshhdl}$ that takes a state as input and returns a handle that is *unused* in state q, in the sense that $\mathbf{val}_q(\mathbf{freshhdl}(q)) \equiv (\mathsf{fail}, \mathsf{fail})$. Again, this function depends on the internal state of the API (in particular on $\mathbf{val}_q$) and not on the interpretation: in each state q, $\mathbf{freshhdl}(q)$ will be instantiated by a particular handle and the $\mathbf{freshhdl}$ symbol will not appear in the terms output to the adversary.

Security properties in CCSA are expressed as the indistinguishability of sets of terms. We use the language introduced above to describe the sets of terms that will correspond to our key secrecy and handle consistency properties.

First, assume we are given a list of fully-specified actions $\mathbf{A}_1, \ldots, \mathbf{A}_m$. This list induces a state transition system $q_0 \rightarrow q_1 \rightarrow \cdots \rightarrow q_m$ as follows:

- The adversary's initial knowledge, ϕ_0, is the set of pairs $\{(h_j, \mathbf{a}_{q_0}(h_j)) \mid 1 \leq j \leq m\}$.
- In state q_i, the adversary has executed the actions $\mathbf{A}_1, \ldots, \mathbf{A}_{i-1}$ and received the list of terms ϕ_i. Then $\phi_{i+1} := \phi_i; s$ where s is the output from the next action $\mathbf{A}_i$, described below.

If $\mathbf{A}_i = \mathbf{W}(h_1, h_2)$, then

$$s \equiv [\mathsf{P}(\mathbf{a}_{q_i}(h_1), \mathbf{a}_{q_i}(h_2))] \{\mathbf{k}_{q_i}(h_1)\}^r_{\mathbf{k}_{q_i}(h_2), \mathbf{a}_{q_i}(h_1)}$$

where r, a name of sort nonce used to indicate the (fresh) randomness of the wrapping scheme, occurs in no other output terms. The state q_{i+1} is such that $\mathbf{val}_{q_{i+1}} = \mathbf{val}_{q_i}$.

If $\mathbf{A}_i = \mathbf{U}(h)$, then

$$s \equiv [\mathsf{wf}(u) \wedge \mathsf{P}(\pi_2(g_i(\phi_i)), \mathbf{a}_{q_i}(h))] \, \mathbf{freshhdl}(q_i)$$

where $u \equiv \mathsf{uwp}(g_i(\phi_i), \mathbf{k}_{q_i}(h))$. The state is updated as follows: if $h' \not\equiv \mathbf{freshhdl}(q_i)$ then $\mathbf{val}_{q_{i+1}}(h') \equiv \mathbf{val}_{q_i}(h')$ and, with u as above,

$$\mathbf{val}_{q_{i+1}}(\mathbf{freshhdl}(q_i)) \equiv [\mathsf{wf}(u) \wedge \mathsf{P}(\pi_2(g_i(\phi_i)), \mathbf{a}_{q_i}(h))] \, (u, \pi_2(g_i(\phi_i))) : (\mathsf{fail}, \mathsf{bad}),$$

where $\mathsf{bad} \in \mathcal{D}$ is a fixed attribute such that $\mathbf{sacr}(\mathsf{bad}) = \{\mathsf{bad}\}$.

If $\mathbf{A}_i = \mathbf{C}(h)$, then $s \equiv \mathbf{k}_{q_i}(h)$ and q_{i+1} is such that $\mathbf{val}_{q_{i+1}} = \mathbf{val}_{q_i}$. Note that if h does not point to a key in state q_i, then $s \equiv \mathsf{fail}$.

We remark that these output terms correspond to the API execution model in Section II.

B. Symbolic Security

In order to define the symbolic security properties, we define the symbolic analogues to the notions of *corrupt* and *compromised* attributes and handles introduced in Section II. We remark that, if the function symbols EQ, $\vee$ and $\wedge$ are interpreted in the obvious way (which we will later insist on), the following symbolic definitions are equivalent to the computational definitions given previously.

Definition 7. Let $q \in \mathbf{Q}$. Let $\mathbf{C}(\mathcal{H})$ be the set of handles h such that the action $\mathbf{C}(h)$ occurred in the transitions $q_0 \rightarrow \cdots \rightarrow q$. Then let $x \in \mathcal{T}$. We say x is a *corrupt attribute* in state q if $\mathbf{cor}_q(x) \sim \mathsf{true}$, where

$$\mathbf{cor}_q(x) \equiv \mathsf{EQ}(x, \mathsf{bad}) \vee \bigvee_{h \in \mathbf{C}(\mathcal{H})} \mathsf{EQ}(x, \mathbf{a}_q(h)).$$

Then for any $y \in \mathcal{T} \setminus \mathcal{H}$ we say y is a *compromised attribute* in state q if $\mathbf{comp}_q(y) \sim \mathsf{true}$, where

$$\mathbf{comp}_q(y) \equiv \bigvee_{a \in \mathcal{D}} \bigvee_{b \in \mathbf{sacr}(a)} \mathbf{cor}_q(a) \wedge \mathsf{EQ}(y, b).$$

Recall that $a \in \mathbf{sacr}(a)$ for any $a \in \mathcal{D}$ and, if $b \in \mathbf{sacr}(a)$ and $c \in \mathbf{sacr}(b)$, then $c \in \mathbf{sacr}(a)$. With the axiom $\mathsf{EQ}(x, x) \sim \mathsf{true}$ (which is obviously computationally sound for any term x if EQ is interpreted as equality of bitstrings), we can show that if a is a corrupt attribute in q, then a is a compromised attribute in q and if a is compromised in q and $b \in \mathbf{sacr}(a)$, then b is compromised in q.

Definition 8. Let $q \in \mathbf{Q}$ and $h \in \mathcal{H}$. We say h is a *compromised handle* in state q and write $\mathbf{comp}_q(h)$ if and only if $\mathbf{a}_q(h)$ is a compromised attribute in q.

By convention, bad is a corrupt attribute in any state. We can now state the symbolic version of security, and hence the symbolic version of our main Theorem. This Theorem states that that, given a handle h^* pointing to an *external* key k^*, the output ϕ^* of an API execution in which h^* is not compromised is indistinguishable from the output of the same execution but in which h^* points to a key $k^\dagger$ not appearing in ϕ^*. Moreover this is true even if the adversary is given k^* in plaintext at the start of both executions. Additionally, we require that any uncompromised handle points to an honestly generated key together with the attribute that key had in the initial state.

Theorem 2. *Let* $\mathrm{API} = (\mathcal{K}, \mathcal{D}, \mathcal{E}, \mathcal{H}, wm, \mathsf{P})$ *be a key management API such that* P *is valid with respect to* $\mathbf{sacr}$ *and the axioms of Section V are sound. Say* $\mathbf{val}_{q_0^*}(h^*) = (k^*, a^*)$ *where* $a^* \in \mathcal{E}$. *Let* $q_0^\dagger$ *be identical to* q_0^* *except that* $\mathbf{val}_{q_0^\dagger}(h^*) = (k^\dagger, a^*)$, *where* $k^\dagger$ *is a key not appearing in* ϕ^* *(in particular* $k^\dagger \neq k^*$*). Finally, let* t *be a sequence of fully-specified actions inducing the API executions* $(q_0^*, (k^*, \phi_0)) \to \cdots \to (q^*, \phi^*)$ *and* $(q_0^\dagger, (k^*, \phi_0)) \to \cdots \to (q^\dagger, \phi^\dagger)$. *Then the following security properties hold:*

1) *key secrecy:* $\left[\neg\mathbf{comp}_{q^*}(h^*)\right]\phi^* \sim \left[\neg\mathbf{comp}_{q^\dagger}(h^*)\right]\phi^\dagger$
2) *handle consistency:*

$$\phi^\dagger, \neg\mathbf{comp}_{q^\dagger}(h) \to \bigvee_{h_0 \in \mathcal{H}} \mathsf{EQ}(\mathbf{val}_{q^\dagger}(h), \mathbf{val}_{q_0^\dagger}(h_0)) \sim \phi^\dagger, \mathsf{true}$$

In order to relate this symbolic theorem to the computational security properties, we recall the computational interpretation of the logic from [BC14].

C. Computational Interpretation

To express the computational semantics of functions and predicates, we fix the computational interpretation of some functions. For example, the function EQ is supposed to express semantic equality, but without any restrictions it could be interpreted by any Turing machine and it would be effectively impossible to reason about it. Therefore we give fixed interpretations for certain symbolic objects and define a *computational model* to be any model satisfying our restrictions. Crucially, there is a great deal of freedom in how a computational model interprets the adversarial function symbols $g_i \in \mathcal{G}$, so that all realistic attacks (i.e. those that run in polynomial-time and do not have access to the internal randomness of the API) can be captured by computational models. Once these interpretation of terms are defined, the interpretation of the formula $\vec{x} \sim \vec{y}$ is simply that no PPT adversary can distinguish between the interpretation of $\vec{x}$ and the interpretation of $\vec{y}$.

Intuitively, we ensure that the encryption and decryption symbols are interpreted as (possibly deterministic) authenticated encryption and decryption algorithms and we also enforce some straightforward restrictions on pairs, projections, equality and boolean symbols. Details can be found in the long version of the paper [SSO16]. If the computational interpretation of a formula holds for any interpretation (with the aforementioned restrictions) of the function symbols, we say that this formula is *sound*.

Given these restrictions on the computational interpretation, we show that Theorem 2 entails Theorem 1. First we show that we can reduce the security game to the case of a single list of actions and then further reduce to when the test handle is chosen at the start of the execution. Details are provided in the long version of the paper [SSO16].

V. Axioms

As in [BC14], we define axiom schemes $\gamma \parallel \theta(\vec{v})$ where γ is a *constraint* and $\vec{v}$ is the vector of free variables in the formula θ. This is the set of all instances $\theta(\vec{v})\sigma$ where σ is an assignment of the free variables $\vec{v}$ to ground terms, and is such that $\sigma \models \gamma$. For example, we define a constraint $\mathrm{fresh}(r; \vec{w})$ such that $\sigma \models \mathrm{fresh}(r; \vec{w})$ if and only if r does not occur in $\vec{w}\sigma$.

In this Section we give our axioms A. We use $\theta(\vec{v})$ as a shorthand for $\mathsf{true} \parallel \theta(\vec{v})$ (i.e. axioms that are satisfied in any assignment of the free variables in $\vec{v}$).

A. Basic Properties

In Figure 4 we give basic axioms, mostly proposed in [BC14]. These state that $\sim$ forms an equivalence relation on sets of terms, together with some essential properties of equality and conditional branching. In particular, we make regular use of the axiom $\vec{v}, [\mathsf{EQ}(x, y)]u(x) : z \sim \vec{v}, [\mathsf{EQ}(x, y)]u(y) : z$. The soundness proof of the basic axioms not already present in [BC14] can be found in the long version of the paper [SSO16]. We remark that Axiom 9 can be immediately derived from axiom IFEVAL in [BC16].

$$z_1, [\mathsf{true}]x : y, z_2 \sim z_1, x, z_2 \quad (1)$$

$$z_1, [\mathsf{false}]x : y, z_2 \sim z_1, y, z_2 \quad (2)$$

$$\text{for } f : s_1 \times \cdots \times s_n \to s \quad \vec{v_1}, x_1, \ldots, x_n \sim \vec{v_2}, y_1, \ldots, y_n \to \vec{v_1}, f(x_1, \ldots, x_n) \sim \vec{v_2}, f(y_1, \ldots, y_n) \quad (3)$$

$$\vec{x} \sim \vec{x} \quad (4)$$

$$\vec{x} \sim \vec{y} \to \vec{y} \sim \vec{x} \quad (5)$$

$$\vec{x} \sim \vec{y} \wedge \vec{y} \sim \vec{z} \to \vec{x} \sim \vec{z} \quad (6)$$

$$\text{If } p \text{ projects and permutes onto a sublist, } \vec{x} \sim \vec{y} \to p(\vec{x}) \sim p(\vec{y}) \quad (7)$$

$$\mathsf{EQ}(x, x) \sim \mathsf{true} \quad (8)$$

$$\vec{v}, [\mathsf{EQ}(x, y)]u(x) : w \sim \vec{v}, [\mathsf{EQ}(x, y)]u(y) : w \quad (9)$$

$$\text{For } c \in \{\mathsf{true}, \mathsf{false}, \mathsf{fail}\} \cup \mathcal{D} \cup \mathcal{H} \quad x \sim c \leftrightarrow \vec{v}, x \sim \vec{v}, c \quad (10)$$

Figure 4. Basic axioms

We now state an important lemma that allow us to talk about semantic equality of terms.

Lemma 9. *(Rewriting With Equalities)* $\mathsf{EQ}(x, y) \sim \mathsf{true}$ *if and only if* $\vec{v}, x \sim \vec{v}, y$ *for any sequence of terms* $\vec{v}$. *That is,* $\mathsf{EQ}(x, y) \sim \mathsf{true}$ *if and only if* x *and* y *are indistinguishable* in any context.

This Lemma follows from the basic axioms. The complete proof can be found in the long version of the paper [SSO16]. We remark that this Lemma, seen here as a consequence of the axioms, also follows immediately from axiom EQCONG in [BC16].

Remark 10. Since $\mathsf{EQ}(x, y) \sim \mathsf{true}$ means we can replace x by y everywhere, we use the notation $x = y$ in this case. This considerably simplifies long formulae, but one must note the distinction between *syntactic* equality $x \equiv y$, which means that x and y are the same terms from $\mathcal{T}$, and *semantic* equality $x = y$, which means that x and y are (possibly)

different terms representing, *up to a negligible probability*, the same underlying object. Obviously by the computational interpretation of EQ, if $x \equiv y$ then $x = y$.

Function symbols applied to terms of sort bool in the symbolic model behave exactly as the corresponding operations on the booleans $\{0,1\}$ in the computational model. We state the corresponding axioms in Figure 5 and prove them sound in the long version of the paper [SSO16]. Note that the boolean axioms described here could be easily derived from the ones in [BC16], given a proper axiomatisation of the logical connectives.

As a consequence of these axioms, we can manipulate terms of sort bool within sets of terms in an intuitive way. We use this manipulation regularly, without citing particular axioms, in the proof of the symbolic theorem in Section VI.

$$b = [b]\mathsf{true} : \mathsf{false} \tag{11}$$

$$x = [b]x : x \tag{12}$$

$$[b_1]\left([b_2]u : v\right) : w = [b_1]\left([b_1 \to b_2]u : v\right) : w \tag{13}$$

$$[b_1]u : \left([b_2]v : w\right) = [b_1]u : \left([\neg b_1 \to b_2]v : w\right) \tag{14}$$

- For any $f : s_1 \times \cdots \times s_n \to s$ and for any $\vec{v} \equiv v_1, \ldots, v_n$ and $\vec{w} \equiv w_1, \ldots, w_n$ with each $v_i, w_i \in s_i$,

$$f\left([b]\vec{v} : \vec{w}\right) = [b]f(\vec{v}) : f(\vec{w}) \tag{15}$$

- Let $f_1\left(X_1, \ldots, X_n\right), f_2\left(Y_1, \ldots, Y_m\right)$ be *boolean functions* (propositional terms built using the connectives $\{\neg, \wedge, \vee, \to\}$ and with n and m propositional variables, respectively). If $f_1(X_1, \ldots, X_n) \Leftrightarrow f_2(Y_1, \ldots, Y_m)$,

$$f_1\left(b_1, \ldots, b_n\right) = f_2\left(b_1, \ldots, b_m\right) \tag{16}$$

Figure 5. Boolean axioms

We state the following as an example of the kind of formal deduction that is possible:

Example 11. The formula $b_1 \to b_2 = [b_1]\, b_2 : \mathsf{true}$ is valid in any model for the axioms previously listed.

Next we present an axiom that is non-trivial and has not appeared in the literature before. The new axiom is used to prove a formula $\vec{v_1}, u_1 \sim \vec{v_2}, u_2$ by splitting it into multiple cases. In order for this strategy to work, the cases must form, with overwhelming probability, a perfect partition of the space of possibilities. This is formalised in the following definition:

Definition 12. Let I be a finite indexing set such that $\left(b^i\right)_{i \in I}$ is a family of terms of sort bool. If the formula

$$\left[\bigvee_{i \in I} b^i \wedge \bigwedge_{i,j \in I, i \neq j} \neg\left(b^i \wedge b^j\right)\right] = \mathsf{true}$$

holds, then we say $\left(b^i\right)_{i \in I}$ is a *partition*.

We remark that if $\left(b^i\right)_{i \in I}$ is a partition, then, with overwhelming probability, there is exactly one $i \in I$ such that $\left[\!\left[b^i\right]\!\right]^{\sigma}_{\eta,\rho} = 1$.

If $\left(b_1^i\right)_{i \in I}$ and $\left(b_2^i\right)_{i \in I}$ are partitions (with the same indexing set), then we define the *case disjunction axiom*:

$$\bigwedge_{i \in I}\left(\vec{v}_1, b_1^i, [b_1^i]u_1 \sim \vec{v}_2, b_2^i, [b_2^i]u_2\right) \to \vec{v}_1, u_1 \sim \vec{v}_2, u_2 \tag{17}$$

As we prove in the long version of the paper [SSO16], this axiom is sound in all computational models.

B. Cryptographic Axioms

The core of the proof relies on axioms representing our *cryptographic* assumptions. To our knowledge, these axioms have not appeared in this form before, but the soundness proofs closely resemble others in the literature.

In Section V-C, we present some logical consequences of these axioms that are what we actually use in the symbolic proof.

The randomised version of the strong secrecy axiom states that if an encryption key k appears in a sequence of terms in essentially the way described in the AE-AD Privacy game - only as an encryption key and never with the same nonce for different plaintexts (and associated data) - then every encryption under k can be replaced with a random string (derived from a term of sort longnonce) while preserving indistinguishability. We formalise the constraint on k as follows:

Let the constraint $\mathrm{enckey}(k; \vec{v})$ be true if and only if the following conditions are satisfied:

- In $\vec{v}$, the term k only appears in the position of an encryption key with a term of sort nonce in the position of the nonce and a term of sort attribute in associated data position,
- If two encryptions under k appearing in $\vec{v}$ share the same term in the position of the nonce, then the terms in the plaintext and associated data positions are also identical.

Given a sequence of terms $\vec{v}$, let $\$_k\left(\vec{v}\right)$ be the sequence of terms obtained by replacing every encryption $\mathsf{enc}\left(m, a, r, k\right)$ under k appearing in $\vec{v}$ with a term $\$\left(\mathsf{enc}\left(m, a, r, k\right), r'\right)$, where r' is a fresh term of sort longnonce that is indexed by the encryption term. Indexing the long nonces this way means that, if the same encryption term appears more than once in $\vec{v}$, then it is replaced with the same random string in $\$_k\left(\vec{v}\right)$.

When encryption is randomised, using fresh nonces in the encryption terms ensures that ciphertexts created at different times will not collide (up to a negligible probability). So encryptions created at different times can be replaced by different random strings, regardless of the underlying plaintext, without an adversary being able to distinguish the change. We do not have this guarantee when encryption is deterministic: encryptions of the same message (i.e. wraps of the same key) created at different times will give the same ciphertexts and must be replaced by the same random string. So for the deterministic version of our strong secrecy axiom, instead of replacing all encryptions under k at once we only replace the encryptions of a particular term. In addition we need to check in the premise of the axiom that the computational interpretation of this term is different to the interpretations of other (syntactically different) terms encrypted under k. For example, the encryption of $\pi_1\left(x, y\right)$ will need to be replaced

by the same random string as the encryption of x, even though the plaintexts are syntactically different.

Let us define $\$_k^d(\vec{v}, m, a)$ as the sequence of terms obtained by replacing every encryption $\mathsf{enc}(m, a, r, k)$ under k appearing in $\vec{v}$ with a term $\$\left(\mathsf{enc}(m, a, r, k), r'\right)$, where r' is a fresh term of sort $\mathsf{longnonce}$ that is indexed by (m, a, k). Note that r' must not depend on r as, in the deterministic interpretation of encryption, r is ignored.

- *Strong Secrecy of Wrapping (randomised):*

$$\mathrm{enckey}(k; \vec{v}) \parallel \vec{v} \sim \$_k\left(\vec{v}\right) \tag{18}$$

- *Strong Secrecy of Wrapping (deterministic):*

$$\mathrm{enckey}(k; \vec{v}) \parallel \left(\bigwedge_{\substack{\{m'\}^r_{k,a} \in \mathrm{st}(\vec{v}) \\ m' \not\equiv m}} \left(\mathsf{EQ}(m, m')\right) = \mathsf{false} \right) \rightarrow \vec{v} \sim \$_k^d(\vec{v}, m, a) \tag{19}$$

- *Integrity of Wrapping:*

$$\mathrm{enckey}(k; x) \parallel \mathsf{wf}\left(\mathsf{uwp}(x, k)\right) = \left(\mathsf{wf}(x) \wedge \bigvee_{\{m\}^r_{k,a} \in \mathrm{st}(x)} \mathsf{EQ}\left(x, \{m\}^r_{k,a}\right) \right) \tag{20}$$

- *Correctness of Wrapping:*

$$\mathsf{uwp}\left(\{m\}^r_{k,a}, k\right) = m \tag{21}$$

- *Correctness of Projections:*

$$\pi_1\left(x, y, z\right) = x \tag{22}$$
$$\pi_2\left(x, y, z\right) = y \tag{23}$$
$$\pi_3\left(x, y, z\right) = z \tag{24}$$

- *Random String Length Consistency:* If $K \subseteq \mathcal{K}$

$$\$_k\left(\mathsf{o}_{k,A}\left(\vec{v}\right)\right) \sim \$_k\left(\vec{v}\right) \tag{25}$$

- *Failure Propagation:*

$$\{\mathsf{fail}\}^z_{x,y} = \mathsf{fail} = \mathsf{uwp}(\mathsf{fail}, x) \tag{26}$$

Figure 6. Cryptographic axioms

Provided that the encryption and decryption symbols are together interpreted as a secure deterministic (resp. randomised) AE-AD scheme with some simple correctness assumptions, the deterministic (resp. randomised) version of these axioms are sound. We prove this in the long version of the paper [SSO16].

C. Consequences of the Axioms

While assuming AE-AD security gives us that ciphertexts are indistinguishable from random strings, we actually require a significantly weaker result in order to prove the security of key wrapping APIs: namely, that the encryptions of keys are indistinguishable from *encryptions* of fresh random strings (of the correct length). We call this the weak secrecy axiom.

For this axiom we have to introduce notation for replacing plaintexts inside encryptions. For a vector of terms $\vec{v}$, we denote by $\mathsf{o}_{k,A}(\vec{v})$ the result of replacing any wrap $\{k'\}^r_{k,a} \in \mathrm{st}(\vec{v})$ with $a \in A$ and $k \in \mathcal{K}$, by $\{r_{k,k'}\}^r_{k,a}$ where $r_{k,k'}$ denotes a fresh key (indexed by (k, k') in the same way as for $\$_k$ and $\$_k^d$). For example, if $\vec{v} \equiv \pi_1\left(\{k'\}^r_{k,a}\right), \{k''\}^{r'}_{k,a'}, \{k\}^r_{k',a}$, then $\mathsf{o}_{k,\{a,a'\}}\left(\vec{v}\right) \equiv \pi_1\left(\{r_{k,k'}\}^r_{k,a}\right), \{r_{k,k''}\}^{r'}_{k,a'}, \{k\}^r_{k',a}$.

So that the weak secrecy axiom holds for both randomised and deterministic encryption, we only allow this substitution for terms where the plaintexts are in fact constants of sort key (as opposed to more general terms whose computational interpretations are equal to those of keys). This is the $\mathrm{wellused}$ constraint, formally defined below.

We also prove that the weak secrecy axiom and the integrity axiom, which hold when a key only occurs in the encryption key position in a certain set of terms, also hold in a larger class of terms where the key is *usable*: as well as encryption key positions, the key might also occur in the plaintext position of an encryption, provided that the encryption key is itself usable. This is the case we will mostly encounter in our proofs. The formal definition of the $\mathrm{usablekey}$ constraint is given below.

- *Weak Secrecy of Wrapping:*

$$\mathrm{enckey}(k; \vec{v}), \mathrm{wellused}(a; \vec{v}; k) \parallel \vec{v} \sim \mathsf{o}_{k,a}\left(\vec{v}\right) \tag{27}$$

- *Usable Secrecy of Wrapping*:

$$\mathrm{usablekey}(k; \vec{v}), \mathrm{wellused}(a; \vec{v}; k) \parallel \vec{v} \sim \mathsf{o}_{k,a}\left(\vec{v}\right) \tag{28}$$

- *Usable Integrity of Wrap*:

$$\mathrm{usablekey}(k; x) \parallel \mathsf{wf}\left(\mathsf{uwp}(x, k)\right) = \mathsf{wf}(x) \wedge \bigvee_{\{m\}^r_{k,a} \in \mathrm{st}(x)} \mathsf{EQ}\left(x, \{m\}^r_{k,a}\right) \tag{29}$$

- *Key Freshness*:

$$\mathrm{usablekey}(k, t) \parallel \mathsf{EQ}(k, t) = \mathsf{false} \tag{30}$$

Figure 7. Consequences of the cryptographic axioms

Definition 13. Let X be a set of terms. For all $k \in \mathcal{K}$, we say k is *usable* in X – and write $\mathrm{usablekey}\left(k; X\right)$ – if the following holds:

1) Either $\mathrm{enckey}(k; X)$,
2) or if k appears in a plaintext position $p.\alpha$ in X, then $X|_p \equiv \{k\}^r_{k',a}$ such that
   a) $k' \in \mathcal{K}$ is usable in X, a is well used for k' in X and r is a term of sort nonce
   b) if r occurs in a term $\{k''\}^r_{k',a'} \in \mathrm{st}(X)$, then $k'' \equiv k$ and $a' \equiv a$.

Definition 14. Let X be a set of terms. We say that the attribute a is *well-used* for the key k in X (written $\mathrm{wellused}(a; X; k)$) if, for every $\{m\}^r_{k,a} \in \mathrm{st}(X)$, m is a constant of sort key.

The proof that these axioms are consequences of the previous cryptographic axioms is proved in the long version of the paper [SSO16].

D. Policy Axioms

We conclude this Section by giving, in Figure 8, axioms reflecting the validity of a policy. These axioms are sound under the conditions of Section III.

As before, there are three criteria: the first is that the policy respects the sacrifice function, in the sense that if the policy accepts the pair (x, a), then x must be semantically equal to an attribute b such that $b \in \mathbf{sacr}(a)$. The second criterion is that the policy never allows external keys to be used for wrapping, so the policy will never accept (x, a) if $a \in \mathcal{E}$. Finally, the policy must forbid the creation of key cycles by enforcing a strict partial ordering on attributes. Additionally we give two useful consequences of these axioms, namely stating that compromise is monotonic in the API execution and that the compromise function respects the policy. These axioms are sound as soon as the function implementing the policy is valid with respect to $\mathbf{sacr}$.

Policy axioms:

$$\text{for } a \in \mathcal{D}\text{: } \mathsf{P}(x,a) = \left(\mathsf{P}(x,a) \wedge \bigvee_{b \in \mathbf{sacr}(a)} \mathsf{EQ}(x,b)\right) \tag{31}$$

$$\text{for } a \in \mathcal{E}\text{: } \mathsf{P}(x,a) = \mathsf{false} \tag{32}$$

- There exists a strict partial ordering $\prec$ on $\mathcal{D}$ such that:

$$(\mathsf{P}(a,b) = \mathsf{true}) \rightarrow a \prec b \tag{33}$$

Consequences of the policy axioms and the definition of $\mathbf{comp}$:

- If there is a sequence of actions inducing a transition from state q to state q':

$$\mathbf{comp}_q(x) \rightarrow \mathbf{comp}_{q'}(x) = \mathsf{true} \tag{34}$$

- For $x, y \in \mathcal{T} \setminus \mathcal{H}$:

$$\left(\mathbf{comp}_q(y) \wedge \mathsf{P}(x,y) \rightarrow \mathbf{comp}_q(x)\right) = \mathsf{true} \tag{35}$$

Figure 8. Policy axioms

VI. Proof of the Symbolic Theorem

A detailed proof is provided in the long version of the paper [SSO16]. We provide here an outline of the proof.

Before tackling the main Theorem, we prove that the frames ϕ^* and $\phi^\dagger$, which are *a priori* very complicated sequences of terms, can be rewritten in a relatively simple form. Let K_0 be the set of honestly-generated keys initially stored by the API.

We start by identifying sets of terms in which keys in K_0 appear only as expected:

1) If they are in a plaintext position, then they are protected by a properly-constructed wrap or known to have a compromised attribute.
2) If they appear in an unwrap position (which cannot be simplified using the integrity and correctness axioms), then they are known to have a compromised attribute.
3) If they appear in a wrapping position, then the key's attribute is internal and either the wrap is of a key from K_0 and its corresponding attribute, or the associated data of the wrap is a compromised attribute.

We remark that – by the policy axioms – uncompromised keys are usable in such sets.

We then identify a sequence $(F^i)_{i \in I}$ of boolean terms, in which keys in K_0 appear as above, that partition the space of possibilities and give sufficient information to determine whether or not any given handle or attribute is compromised. We prove a number of invariants showing that keys in K_0 only appear as expected in $[F^i]\phi$ and $[F^i]\mathbf{val}_q(h)$ and so uncompromised keys are usable in these terms. Additionally, we show that under the conditions of the (F^i), the internal state of the API is consistent with its initial configuration.

Since k^* is uncompromised, the usable secrecy axiom allows us to remove every instance of it from $[F^i]\phi$ until we reach a set of terms where replacing k^* by $k^\dagger$ has no effect. The main theorem then follows by the case disjunction axiom (Axiom 17) applied to the partition $(F^i)_{i \in I}$.

VII. Composing Key Wrapping with Other Functionalities

In this Section we demonstrate how our security property for a key wrapping API composes with the security of other functionalities offered by a cryptographic API, provided that keys used by the other functionalities are external. We will use encryption (of data, not keys) as an example, but the argument can be easily adapted for other cryptographic primitives.

First, we define a game in which the adversary interacts with a key wrapping API and an encryption scheme that uses the external keys managed by the key wrapping API. For simplicity, here the algorithms `enc` and `dec` used by the encryption scheme (not the wrapping mechanism) take just two arguments, the first being the key. Our game is very similar to the one used to define API security in [KSW11].

The game is parameterised by a handle h^* pointing to the key k^* with external attribute a^*. The adversary has access to a real-or-random encryption oracle for k^* and a (genuine) decryption oracle for k^*. Much as in the real-or-random IND-CCA game, the adversary tries to guess whether or not the encryption oracle for k^* is real, without passing any of its outputs to the decryption oracle for k^*. But in addition, the adversary is allowed to wrap and unwrap keys (according to the policy from the key wrapping API), encrypt and decrypt as normal under any external key other than k^*. Finally the adversary may also corrupt any key, provided that they do not compromise h^*. We formalise this below.

Definition 15. Let $\mathtt{API} = (\mathcal{K}, \mathcal{D}, \mathcal{E}, \mathcal{H}, \mathtt{wm}, \mathtt{P})$ be a key wrapping API and let $\Pi = (\mathtt{keygen}, \mathtt{enc}, \mathtt{dec})$ be an encryption scheme. Then we say $\mathtt{API}' = (\mathtt{API}, \Pi)$ is a secure encryption and wrapping API if, for all integers m and n, all $\vec{a} = a_1, \ldots, a_n \in \mathcal{D}^n$ with $a_1 = a^* \in \mathcal{E}$ and all polynomial-time adversaries $\mathcal{A}$, the following advantage is a negligible function of η:

$$\mathrm{Adv}^{\mathrm{IND-CCA}}_{\mathtt{API}'}(\mathcal{A}) := \mathbb{P}_b\left[\mathrm{Exp}^{\mathrm{COMP}}_b(\mathcal{A}, \vec{a}, m) = b\right] - \frac{1}{2}$$

where the experiments $\mathrm{Exp}^{\mathrm{COMP}}_b(\mathcal{A}, \vec{a}, m)$ for $b \in \{0,1\}$ and the new oracles are given in Figures 9 and 10.

Experiment $\mathrm{Exp}_b^{\mathrm{COMP}}(\mathcal{A}, a_1, \ldots, a_n, m)$:
$\mathbf{st}.C \leftarrow 0$
$\mathbf{st}.V \leftarrow \top$
$\mathbf{st}.\mathbf{cor} \leftarrow []$
$\mathbf{st}.e \leftarrow []$
For $1 \leq i \leq n$
  $k_i \leftarrow\$ \, \mathtt{keygen}(1^\eta)$
  $\mathbf{st}.\mathbf{val}(h_i) \leftarrow (k_i, a_i)$
$\mathbf{st}.H \leftarrow \{h_1, \ldots, h_n\}$
$(h^*, a^*, k^*) \leftarrow (h_1, a_1, k_1)$
$\mathbf{st}.T \leftarrow a^*$
$\mathcal{O}_{\mathtt{API}} \leftarrow (\mathcal{O}^{\mathtt{wrap}}, \mathcal{O}^{\mathtt{unwrap}}, \mathcal{O}^{\mathtt{corrupt}}, \mathcal{O}_b^{\mathtt{enc}}, \mathcal{O}^{\mathtt{dec}})$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathtt{API}}}$
If $\mathbf{st}.V = \top$ Return b'
$b'' \leftarrow\$ \{0,1\}$
Return b''

Figure 9. Cryptographic API Security Experiment

Oracle $\mathcal{O}_b^{\mathtt{enc}}(h, m)$:
$\mathbf{st}.C \leftarrow \mathbf{st}.C + 1$
If $\mathbf{st}.C > m$ Return $\lightning$
$(k, a) \leftarrow \mathbf{st}.\mathbf{val}(h)$
If $a \notin \mathcal{E}$ Return $\lightning$
If $h = h^*$
  If b
    $c \leftarrow \mathtt{enc}(k, m)$
  Else
    $m' \leftarrow\$ \{0,1\}^{|m|}$
    $c \leftarrow \mathtt{enc}(k, m')$
  $\mathbf{st}.e \leftarrow \mathbf{st}.e \cup \{c\}$
  Return c
Else
  Return $\mathtt{enc}(k, m)$

Oracle $\mathcal{O}^{\mathtt{dec}}(h, c)$:
$\mathbf{st}.C \leftarrow \mathbf{st}.C + 1$
If $\mathbf{st}.C > m$ Return $\lightning$
If $c \in \mathbf{st}.e$ then $\mathbf{st}.V \leftarrow \bot$
$(k, a) \leftarrow \mathbf{st}.\mathbf{val}(h)$
If $a \notin \mathcal{E}$ Return $\lightning$
Return $\mathtt{dec}(k, c)$

Figure 10. Encryption and Decryption Oracles

We obtain the following composition theorem (the proof is in the long version of the paper [SSO16]).

Theorem 3. *Let* $\mathtt{API}$ *be key wrapping API satisfying the hypotheses of Theorem 1 and let* Π *be an IND-CCA secure encryption scheme. Then* $\mathtt{API}' = (\mathtt{API}, \Pi)$ *is a secure encryption and wrapping API.*

VIII. Applications

A. Refinement of PKCS#11

We view PKCS#11 as the composition of a key wrapping API and an encryption scheme in the sense of Theorem 3. We describe a refinement of Version 2.4 of the PKCS#11 standard, such that this composition is secure in the sense of Theorem 3. Our refinement is less restrictive and our security guarantees are stronger than in [Kün15].

The PKCS#11 standard does not make a distinction between the key management module of a cryptographic token and its other functionalities [OAS15]. However, as the wrap and decrypt attack from [BCFS10] demonstrates, not separating keys for different functionalities can lead to attacks. Therefore to obtain a secure refinement of PKCS#11 we first forbid the creation of keys with attributes that can be used for multiple roles and we do not allow attributes to be modified. We remark that the standard explicitly supports the use of additional restrictions on attributes such as these.

Our final additional assumption is that the wrapping mechanism is a secure AE-AD encryption scheme (for example AES-GCM, supported in PKCS#11 since version 2.4 of the standard) that binds attributes properly when wrapping. Note that all our refinements are explicitly permitted by the PKCS#11 specification.

Then, Theorem 3 shows that an IND-CCA secure encryption mechanism using external keys can be securely combined with the key management functionalities of PKCS#11. More details of this analysis can be found in the long version of the paper [SSO16].

B. The Kremer-Steel-Warinschi API

In this Section we show how our result can be used to encapsulate security proofs of other computational APIs that have appeared in the literature. As an example we view the API in [KSW11] as an instance of a generic API described by our model and verify its security.

The authors of [KSW11] give a security definition for APIs with wrapping and encryption mechanisms. As in our composition result, the adversary is not supposed to distinguish between real and fake encryptions under a particular challenge key. The authors go on to describe an example implementation, in which the wrapping policy is a simple hierarchy on keys with external keys at the bottom level, and prove its security. Clearly this is a valid policy according to our definition. Therefore Theorem 3 subsumes the security proof given for this particular API design in [KSW11].

IX. Conclusion

We give here a general definition of a key wrapping API, parameterised by a wrapping mechanism and a wrapping policy. We provide a set of simple conditions for a key wrapping policy to be valid. Namely, forbidding creation of wrapping cycles, not allowing wraps under external keys and respecting the enterprise-level policy. We prove that if, in addition to these policy conditions, the key wrapping mechanism is implemented by an AE-AD encryption scheme and attributes are properly transmitted, the external keys are indistinguishable from random values.

Our strong secrecy notion for external keys, together with our generic notion of valid policies, allows us to prove that a secure key wrapping API may be securely composed with an encryption scheme. We are then able to give a configuration of the key wrapping functionality of PKCS#11 such that the encryption mechanism is as secure as when fresh keys are used.

Since our proof technique relies on the CCSA from [BC14], our theorem is derived from a set of relatively minimal axioms representing computational assumptions. These axioms are generic enough to be proven sound under a number of different cryptographic hypotheses, notably deterministic and randomised variants of authenticated encryption.

One aspect of API policies not captured by this work is how the roles of keys may change with time, as in [CSW12], allowing for APIs to recover from the compromise of short-lived session keys. We believe that our proof technique could

be extended to model time and that our results will generalise to this setting, but leave this extension as future work.

Ultimately, the hypotheses required for our security proof - particularly the need to securely bind the attributes of keys when wrapping - may be far from what currently happens in practice. Nevertheless, we argue that this assumption is essential for security, since one must enforce a separation between keys used for wrapping and keys used for other primitives. What we offer is a clear blueprint for building secure key management APIs in the future:

1) Explicitly set out which keys are intended for wrapping (internal) and which are intended for other uses (external) and the intended security relationships between keys (a sacrifice function).
2) Build an API that uses a secure AE-AD encryption scheme to bind the attributes of keys to the wraps and where, when wrapping keys, there is an *explicit* check performed on attributes (an API security policy).
3) Verify that the API security policy is acyclic, respects the sacrifice function and forbids the use of external keys for wrapping.

In this way, one guarantees that external keys are as secure as possible.

X. Acknowledgements

This work was supported by the European Union's 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE) and an EPSRC ICASE award n. 13440011.

The authors would like to thank Bogdan Warinschi, Martijn Stam and Karthikeyan Bhargavan for their useful feedback on the paper. The authors would also like to thank the anonymous reviewers for their valuable comments.

References

[AFL13] 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. IEEE, 2013.

[BC14] Gergei Bana and Hubert Comon-Lundh. A computationally complete symbolic attacker for equivalence properties. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 609–620, 2014.

[BC16] Gergei Bana and Rohit Chadha. Verification methods for the computationally complete symbolic attacker based on indistinguishability. Cryptology ePrint Archive, Report 2016/069, 2016. http://eprint.iacr.org/.

[BCFS10] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 260–269, 2010.

[BFK+12] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 608–625. Springer, 2012.

[CC09] Christian Cachin and Nishanth Chandran. A secure cryptographic token interface. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 141–153, 2009.

[CFL13] 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.

[CGH12] David Cash, Matthew Green, and Susan Hohenberger. New definitions and separations for circular security. In *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings*, volume 7293 of *Lecture Notes in Computer Science*, pages 540–557. Springer, 2012.

[Clu03] Jolyon Clulow. On the security of PKCS#11. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, pages 411–425, 2003.

[CS09] Véronique Cortier and Graham Steel. A generic security API for symmetric key management on cryptographic devices. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, pages 605–620, 2009.

[CSW12] Véronique Cortier, Graham Steel, and Cyrille Wiedling. Revoke and let live: a secure key revocation API for cryptographic devices. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 918–928, 2012.

[DKS08] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008*, pages 331–344, 2008.

[DKS10] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal security analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, 2010.

[KKS13] 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.

[KSW11] 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.

[Kün15] Robert Künnemann. Automated backward analysis of pkcs#11 v2.20. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9036 of *Lecture Notes in Computer Science*, pages 219–238. Springer, 2015.

[NIS12] NIST Special Publication 800-57: Recommendation for Key Management - Part 1: General (Revision 3), July 2012.

[OAS15] PKCS#11 cryptographic token interface base specification version 2.40, April 2015. Latest version: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html.

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 98–107, 2002.

[RS06] Phillip Rogaway and Thomas Shrimpton. Deterministic authenticated encryption: A provable-security treatment of the key-wrap problem. In *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 373–390. Springer, 2006.

[SSO16] Guillaume Scerri and Ryan Stanley-Oakes. Analysis of key wrapping apis: Generic policies, computational security. Cryptology ePrint Archive, Report 2016/433, 2016. http://eprint.iacr.org/.