A Secure Cryptographic Token Interface

Christian Cachin[*] Nishanth Chandran[†]

April 22, 2009

Abstract

Cryptographic keys must be protected from exposure. In real-world applications, they are often guarded by cryptographic tokens that employ sophisticated hardware-security measures. Several logical attacks on the key management operations of cryptographic tokens have been reported in the past, which allowed to expose keys merely by exploiting the token API in unexpected ways.

This paper proposes a novel, provably secure, cryptographic token interface that supports multiple users, implements symmetric cryptosystems and public-key schemes, and provides operations for key generation, encryption, authentication, and key wrapping. The token interface allows only the most important operations found in real-world token APIs; while flexible to be of practical use, it is restricted enough so that it does not expose any key to a user without sufficient privileges. An emulation of the security policy in the industry-standard PKCS #11 interface is demonstrated.

1 Introduction

Cryptographic tokens or *hardware security modules (HSM)* are an important part of many security infrastructures that use cryptography. They protect cryptographic keys in hostile environments using physical tamper-protection measures. Cryptographic tokens store keys and use them for cryptographic operations, but the keys typically never leave the physical security perimeter established by the secure hardware. These tokens are used because they can be controlled better than the hosts who invoke operations on them. HSMs exist in many environments today, ranging from smartcards and the Trusted Platform Module [26] found in many personal computers, to high-security HSMs used by the finance industry such as IBM's 4764 cryptoprocessor [18]. Two prominent token interfaces used in industry are PKCS #11 [25] and IBM CCA [18].

A cryptographic token acts as an auxiliary device for an application running on a host computer; it performs various cryptographic tasks, such as key generation, key derivation, encryption, decryption, signing, signature verification and so on. Tokens also enforce a security policy on the stored keys and other cryptographic objects. When an application invokes a command of the token, the operation may access inputs supplied by the application as well as objects stored by the token. The application may not always receive all results of the operation, as, for example, when a key is generated and stored only in the token. The security policy of the token interface defines which operations are permitted to whom. Tokens usually have at least two security levels, one for the payload application, which has restricted privileges, and one for the administrator, who may access all keys in the token.

Designing a secure cryptographic token interface is a challenging problem that has not been solved today [3]. One has to find a balance between flexibility on the one hand, for satisfying the demands of the application programmers, and a restrictive policy on the other hand, for guaranteeing a meaningful

[*] IBM Zurich Research Laboratory, Säumerstr. 4, CH-8803 Rüschlikon, Switzerland. cca@zurich.ibm.com.

[†] University of California, Los Angeles, Department of Computer Science, 3714 Boelter Hall, Los Angeles CA 90095, USA. nishanth@cs.ucla.edu. Work done at IBM Zurich Research Laboratory.

notion of security. Many attacks on token interfaces have been reported in the literature over the recent years [4, 10, 12, 1], which illustrate this problem. By exploiting an interface in unexpected ways, these attacks typically allow an adversary to undermine the (intended) security policy of a token. They are dangerous because an adversary only needs logical access to the token, and completely bypasses the physical protection.

In this paper, we introduce a cryptographic token interface that supports multiple users and provides operations for key generation, encryption, authentication, and key wrapping. Our token model defines an access control list (ACL) for every key. We show how to implement our token securely from a set of cryptographic primitives and prove that it respects the security policy expressed by the ACL.

Our proposal constitutes the first formal model of a universal token interface with an explicit security policy, and it defines security in a strong and cryptographically sound way. It supports common key-management operations and cryptographic functions, including encryption and key wrapping, which may interact in unexpected ways and were a common source of problems in earlier token interfaces. It models multiple users and goes beyond most previous token interfaces, which distinguish only between two user roles (application and administrator).

We intend this model to give a basis for the key management interfaces of future *hardware tokens* and beyond that, for future networked *key management servers*. Key management in enterprises is moving towards key servers accessible online [8], and several emerging standardization efforts address key-management interfaces [14]. In particular, the recently published OASIS Key Management Interoperability Protocol [21] contains all relevant key-management operations of a cryptographic token interface. Available to multiple users, such key managers must implement sophisticated access-control policies and provide all relevant key-management functions addressed here, including key generation, wrapping, and key derivation. At the same time, our model also isolates a subset of the features in existing cryptographic token interfaces, such that they have a clearly defined security policy and offer provable cryptographic security. As a starting point, we identify a safe subset of PKCS #11 in this paper.

Approach. In existing token interfaces, seemingly benign operations may sometimes have unexpected consequences. One problematic function is *key wrapping*, whose goal is to encrypt a target key under a wrapping key for export and transport to another token or re-import at a later time. But some interfaces allow an unprivileged user to request that a privileged key be wrapped and exported under a key accessible to the user, so that the user may obtain the cleartext of the privileged key [12].

Key wrapping is also dangerous if the token does not bind the attributes of a target key in a secure way to the wrapping. An attack may exploit this vulnerability by re-importing the key with different attributes, which subsequently allow an unprivileged user to execute privileged operations. To protect the attributes from modification, our wrapping scheme must satisfy the notion of authenticated semantically secure encryption with associated data [7]. Key wrapping with *deterministic* algorithms was formalized by Rogaway and Shrimpton [24], but we take the liberty to require a stronger notion of wrapping with randomization.

Token interfaces typically support operations for *deriving* a new (symmetric) key from a parent key. Keys in tokens often form a hierarchy constructed like this, where knowledge of one key implies knowledge of all keys below in the hierarchy.

In our interface, we model such dependencies explicitly. We say that a key $\boldsymbol{l}$ *depends* on another key $\boldsymbol{k}$ when revealing $\boldsymbol{k}$ discloses also the value of $\boldsymbol{l}$ to an adversary; a dependency may exist because $\boldsymbol{l}$ was wrapped under $\boldsymbol{k}$ or $\boldsymbol{l}$ was derived from $\boldsymbol{k}$, for example. By storing a history of all operations in the token, we are able to keep track of all dependencies and avoid unexpected consequences of an operation.

Our token interface models separate privileges for most cryptographic operations offered by the token. We enforce a separation on the usage of keys and distinguish between key derivation, encryption, authentication, and key wrapping tasks; every key may only be used for one of the four tasks.

We also require that no key used to import keys through the unwrapping operation has ever been disclosed to a user. This is necessary to maintain the security policy of the token because an adversary might modify the attributes of a key otherwise. It also prevents so-called key-conjuring attacks, whereby an attacker creates a spurious key in the token without invoking an operation to create the key [10, 12, 3]. Such keys are dangerous because they may not satisfy the security policy of the token.

The security policy of our token interface takes into account the interaction of the users with the token and the corresponding state changes; the policy is more complex than previous models because users may, for instance, modify the access-control data associated to a key. But the security policy satisfies some easily stated invariants: When a user never had the privilege to read a certain key, for example, then that user cannot break any encryption with the key or forge a signature under the key; or, when a key has been flagged as "not extractable," no user may obtain the cryptographic value of the key afterwards.

In summary, the contributions of this paper are:

1. The first formal model of a stateful cryptographic token interface that supports an arbitrary number of users and provides all common cryptographic operations.
2. An explicit security policy for the key servers and for key management in cryptographic tokens, which models access control for keys and guarantees the security of the cryptographic operations in a cryptographically sound way.
3. A description of a subset of PKCS #11 that satisfies a formal security policy.

Organization of the Paper. Section 2 describes the problem and previous work. We base our token implementation on a set of cryptographic primitives presented in Section 3. Section 4 introduces our token interface and Section 5 defines its security and analyzes it. Section 6 describes how our multi-user token model can be mapped to the industry-standard PKCS #11 interface. The paper concludes in Section 7.

2 Background and Related Work

IBM's *Common Cryptographic Architecture (CCA)* [19, 18] is a token interface primarily used by banks for securing ATM networks and financial transactions; it is provided by the IBM 4764 cryptoprocessor and its predecessors. CCA defines a control vector for every key that models key usage and other attributes, and introduced a method for binding the control vector securely to a wrapped key.

The *PKCS #11* standard by RSA [25] defines a generic cryptographic token interface that is implemented by many products in software and in hardware. It is widely used today and available on a variety of platforms. HSMs providing a PKCS #11 interface are often used in heterogeneous environments for building public-key infrastructures (PKIs), securing network links using TLS, running VPNs and so on.

Attacks on cryptographic token interface have been pioneered by Anderson, Bond, Clulow and others [4, 10, 12, 1], and mostly address tokens used by the finance industry. These attacks have triggered other researchers to study token interfaces with formal methods. A number of efforts have since analyzed parts of CCA and PKCS #11 using model checkers, theorem provers, and other tools [27, 13, 15]. These works either discover new attacks in existing interfaces automatically or demonstrate the absence of attacks for certain (usually small) configurations.

The recent paper of Delaune et al. [15] comes closest to our approach. It formalizes a subset of PKCS #11's key management commands and addresses the subtle interactions of key wrapping functions with key import, encryption, decryption, and attribute manipulation. The work identifies a number of new unsafe aspects in PKCS #11 discovered with the help of a model checker. Typically, they consist of initial configurations and sequences of commands that lead to unexpected consequences.

Every tool-supported analysis starts from an abstract model of the token interface, usually derived from its specification. These abstract models are similar to our token model. However, there are two differences between those abstractions and our work:

1. Our goal is constructive as opposed to analytic. These formalizations take an existing token interface and isolate a part of it with all its idiosyncrasies. In contrast, our design is not bound to the details of existing designs.

2. Our model uses the notions of modern cryptography to express the security requirements, whereas the formal-model approaches rely on a Dolev-Yao abstraction [16] of cryptography.

On a higher level, though, the goals of all these works is the same: To lay a foundation for the design of future token interfaces with provable security guarantees.

3 Cryptographic Primitives

This section describes the cryptographic primitives that are used by the token and by users.

Let κ be the security parameter. Let $\mathcal{U}_\kappa$ denote the uniform distribution on bit-strings of length κ. The keys used for encryption, authentication, and so on in symmetric cryptosystems are called *secret keys* and are κ-bit strings drawn according to $\mathcal{U}_\kappa$. All *public keys* and *private keys* used for public-key or asymmetric cryptosystems are also represented as κ-bit strings for simplicity; public key/private key pairs are generated by a randomized algorithm PKG and the same key pair may be used for encryption and for signatures. All algorithms are public.

Let $\mathsf{F} : \{0,1\}^\kappa \times \{0,1\}^d \rightarrow \{0,1\}^\kappa$ be an adaptively secure *pseudo-random function* family. Functions in the family are distinguished by a *seed*, which acts as a secret key. We write $\mathsf{F}_s(t)$ for evaluating F with seed s on input $t \in \{0,1\}^d$.

Let SENC denote a randomized *symmetric encryption scheme* with the following two algorithms: an encryption algorithm $\mathsf{SE}(k,m)$ that takes as input a secret key k and a message m and outputs a ciphertext c, and a decryption algorithm $\mathsf{SD}(k,c)$ that takes as input a secret key k and ciphertext c and outputs the decryption of c using k. If the decryption fails, then $\mathsf{SD}(k,c)$ outputs $\perp$. The correctness condition of the symmetric encryption scheme requires that $\mathsf{SD}(k,\mathsf{SE}(k,m)) = m$ for all m and k. We require SENC to be secure against adaptive chosen-ciphertext attacks [20].

Let PENC denote a *public-key encryption scheme*, consisting of an encryption algorithm $\mathsf{PE}(p,m)$ that takes as input a public key p and a message m and outputs a ciphertext c, and a decryption algorithm $\mathsf{PD}(k,c)$ that takes as input a private key k and ciphertext c and outputs the decryption of c using k. Again, if the decryption fails, then $\mathsf{PD}(k,c)$ outputs $\perp$. The correctness condition of the public-key encryption scheme requires that $\mathsf{PD}(p,\mathsf{PE}(k,m)) = m$ for all m and for all pairs (p,k) that are generated by PKG. We require PENC to be secure against adaptive chosen-ciphertext attacks [23, 22, 6].

Define SAUTH to be a *symmetric message authentication scheme* consisting of the following two algorithms: a signing algorithm $\mathsf{SS}(k,m)$ that takes as input a key k and a message m and outputs a signature σ, and a verification algorithm $\mathsf{SV}(k,m,\sigma)$ that takes as input a key k, a message m and a purported signature σ, and outputs either *accept* or $\perp$. The correctness condition requires that $\mathsf{SV}(k,m,\mathsf{SS}(k,m)) =$ *accept* for all m and k. We require SAUTH to be a strong existentially unforgeable message-authentication code (MAC) under adaptive chosen-message attack.

Define PAUTH to be a *public-key authentication scheme* (also called a *digital signature scheme*), consisting of a signing algorithm $\mathsf{PS}(k,m)$ that takes as input a private key k and a message m and outputs a signature σ, and a verification algorithm $\mathsf{PV}(p, m, \sigma)$ that takes as input a public key p, a message m and a purported signature σ, and outputs either *accept* or $\perp$. The correctness condition requires that $\mathsf{PV}(p, m, \mathsf{PS}(k, m)) = \textit{accept}$ for all m and for all pairs (p, k) generated by PKG. We require PAUTH to be a strong existentially unforgeable signature scheme under adaptive chosen-message attack [17, 2].

A *key-wrapping scheme* is essentially a labeled authenticated encryption scheme that allows for the encryption of secret keys and private keys. A label is simply a bit-string included with the ciphertext in an unmodifiable way. Let WRAP denote a symmetric key-wrapping scheme with the following two components: a wrapping algorithm $\mathsf{W}(k, l, \ell)$ that takes as input a secret key k, a key l that is either a secret key or a private key, and a label ℓ, and outputs a wrapping w, and an unwrapping algorithm $\mathsf{U}(k, w, \ell)$ that takes as input a secret key k, a wrapping w, and a label ℓ and outputs the unwrapping of w with label ℓ using k. If the unwrapping fails, then $\mathsf{U}(k, w, \ell)$ outputs $\perp$. The correctness condition requires that $\mathsf{U}(k, \mathsf{W}(k, l, \ell), \ell) = l$ for all k, l, and ℓ. We assume WRAP is an authenticated symmetric-key encryption scheme [7]. In particular, WRAP provides not only secrecy but also ciphertext integrity. WRAP can be seen as a randomized version of deterministic key wrap [24].

Our set of *cryptographic primitives* P consists of F, SENC, PENC, SAUTH, PAUTH, and WRAP.

4 Token Model

4.1 Structure

We first describe the structure of a cryptographic token. The token stores several keys together with different attributes and performs cryptographic operations with the keys. We do not consider other objects stored by a token, such as data objects, since our focus is on keys and their management. The token uses a set P of secure cryptographic primitives according to Section 3.

There are multiple *users* in the system who may access the token. We denote the set of users by $\mathbb{U}$; there is a special element $\textit{any} \in \mathbb{U}$ that denotes any user in the system. Before a user may interact with the token, the user must *log in* to the token and may authenticate itself. The user remains logged in until another login operation occurs. When the user does not authenticate itself during login, we assume that user *any* is logged in. For simplicity, we assume that a trusted entity configures $\mathbb{U}$ during the initialization of the token and that $\mathbb{U}$ does not change afterwards.

The token internally stores a history of all its interactions with the environment in a *log* $\mathbb{L}$. The log contains a tuple for every operation that successfully completed; the tuple denotes the user who executed the operation, the operation itself, the input value, and the return value of the operation.

More precisely, let $\mathbb{O}$ denote the set of operations that can be performed by the token. When the token receives a request to perform an operation $o \in \mathbb{O}$ on input x from a user $u \in \mathbb{U}$ that is logged in, the token first checks if the operation is permitted. If so, the token executes the operation, possibly updates its state, and computes a return value y. Then it gives y to the user. An operation may either succeed or fail; for all operations that succeed, the token updates its history by appending (u, o, x, y) to $\mathbb{L}$. We often state a condition on the existence of some entry in the log for an operation o and input x with short-hand notation like $(\cdot, o, x, \cdot) \in \mathbb{L}$; it means that there exist $u \in \mathbb{U}$ and y such that $(u, o, x, y) \in \mathbb{L}$.

In Section 4.2, we shall describe the key objects in detail and specify their attributes that also include access control information. In Section 4.3, we shall describe the functionalities that can be performed on these keys using the token.

4.2 Key Objects and Attributes

We denote the set of key identifiers by $\mathbb{K}$ and use the convention that $\boldsymbol{k} \in \mathbb{K}$ merely denotes a handle for a key and $k \in \{0,1\}^{\kappa}$ denotes its cryptographic value. We often refer to $\boldsymbol{k}$ as a "key" even when its value k is not available. Let $\mathbb{G} \subseteq \mathbb{K}$ denote the set of keys that have been generated through *create* or *derive* operations (as defined in Section 4.3); initially $\mathbb{G}$ is empty.

The set $\mathbb{K}$ comprises the *secret keys* used by symmetric cryptographic primitives as well as the *public keys* and *private keys* used by public-key cryptosystems and digital signature schemes.

We introduce a relation between keys that is defined in terms of two operations *wrap* and *derive*, which are described more precisely in Section 4.3. The operation *wrap*$(\boldsymbol{k}, \boldsymbol{l})$ encrypts a target key $\boldsymbol{l}$ with a wrapping key $\boldsymbol{k}$ and outputs the resulting ciphertext. The operation *derive*$(\boldsymbol{k}, \boldsymbol{l})$ derives a new secret key $\boldsymbol{l}$ in a reproducible way from a parent key $\boldsymbol{k}$.

These operations create dependencies among the keys because the secrecy of the wrapped target key depends on the secrecy of the wrapping key and the secrecy of a derived key depends similarly on the parent key.

Definition 1 (Depends on). *We say that a key $\boldsymbol{l}$* depends on *a key $\boldsymbol{k}$ whenever one of the following conditions hold:*

1. *$\boldsymbol{l} = \boldsymbol{k}$; or*

2. *the token successfully executed an operation wrap$(\boldsymbol{k}, \boldsymbol{l})$; or*

3. *the token successfully executed an operation derive$(\boldsymbol{k}, \boldsymbol{l})$; or*

4. *there exists a key $\boldsymbol{j}$ such that $\boldsymbol{l}$ depends on $\boldsymbol{j}$ and $\boldsymbol{j}$ depends on $\boldsymbol{k}$.*

The token also provides an operation *read*$(\boldsymbol{k})$ that outputs the cryptographic value of $\boldsymbol{k}$ to the user (see Section 4.3). In our token model, we keep track of those users who may know the cryptographic value of $\boldsymbol{k}$ because they have executed *read* for some key on which $\boldsymbol{k}$ depends (note this includes in particular executing *read* for $\boldsymbol{k}$ itself).

Definition 2 (Reader). *We say that a user u is a* reader *of a key $\boldsymbol{k}$ whenever there exists a key $\boldsymbol{l} \in \mathbb{G}$ such that $\boldsymbol{k}$ depends on $\boldsymbol{l}$ and the token successfully executed an operation read$(\boldsymbol{l})$ with u logged in.*

Keys have one (or more) owners and several other attributes associated with them. These attributes are of three types: *simple attributes* that denote basic information, an *access control list* that specifies the privileges for every user on the key, and *derived attributes* that are computed from the simple attributes, the access control list, and the log. The simple attributes and the access control list of a key can be modified.

Let $\langle \boldsymbol{k} \rangle$ denote a representation of $\boldsymbol{k}$ and all its attributes; note that $\boldsymbol{k}$ does not include the cryptographic value k.

To simplify the notation, assume that a public key/private key pair $(\boldsymbol{p}, \boldsymbol{k})$ may be used for all asymmetric cryptographic schemes, in particular for public-key encryption and for digital signature schemes. This simplifies also the emulation of legacy token interfaces such as PKCS #11, which allow an RSA key pair to be used for encryption and for signatures, for example.

We now describe the simple attributes, the access control list, and the derived attributes of a key $\boldsymbol{k}$. We refer to some attribute a of $\boldsymbol{k}$ with $\boldsymbol{k}.a$.

Simple Attributes. The simple attributes of a key can be manipulated directly by users who have sufficient privileges. They are:

$\boldsymbol{k}.\mathit{type} \in \{\mathit{public}, \mathit{private}, \mathit{secret}\}$: This attribute describes the type of $\boldsymbol{k}$ and can be *public*, *private* or *secret*. This attribute can only be read by users but not modified.

$\boldsymbol{k}.\mathit{unextractable} \in \{\mathit{false}, \mathit{true}\}$: This is a boolean flag that is set to *true* if k must not be revealed to any user even if encrypted with a key. This value can only be set by an owner of $\boldsymbol{k}$; it may only be set to *true* once and not modified further once it is *true*.

$\boldsymbol{k}.\mathit{pub} \in \mathbb{G}$: When $\boldsymbol{k}$ is of type *private*, this attribute denotes the corresponding public key. If $\boldsymbol{k}$ is of another type, the attribute is not defined. The attribute is set when the key is created and cannot be modified.

$\boldsymbol{k}.\mathit{priv} \in \mathbb{G}$: When $\boldsymbol{k}$ is of type *public*, this attribute denotes the corresponding private key. If $\boldsymbol{k}$ is of another type, the attribute is not defined. The attribute is set when the key is created and cannot be modified.

Note that for a public key/private key pair $(\boldsymbol{p}, \boldsymbol{k})$ stored by the token, it holds $\boldsymbol{p}.\mathit{priv}.\mathit{pub} = \boldsymbol{p}$ and $\boldsymbol{k}.\mathit{pub}.\mathit{priv} = \boldsymbol{k}$ at all times.

Practical token interfaces for key management use many other attributes, such as the algorithm and scheme for which a key is intended, the state of the key in its lifecycle and related time information, and data about validity of the key and related certificates [5]. We do not consider these attributes here for simplicity.

Access Control List. The access control list attribute $\boldsymbol{k}.\mathit{acl}$ represents the access privileges on $\boldsymbol{k}$ for all users. The universe $\mathbb{P}$ of all privileges with respect to keys is described below. The access control list is a set containing tuples of the form (u, p), where $u \in \mathbb{U}$ and $p \in \mathbb{P}$. Privileges in $\mathbb{P}$ are:

admin: Denotes if a user may modify the attributes of the key.

read: Denotes if a user may read the cryptographic value of the key.

derive: Denotes if a user may derive another key from the key.

encrypt: Denotes if a user may encrypt with the key.

decrypt: Denotes if a user may decrypt with the key.

sign: Denotes if a user may sign or authenticate with the key.

verify: Denotes if a user may verify a signature or an authentication value with the key.

wrap: Denotes if a user may wrap another key with this key and export it.

unwrap: Denotes if a user may unwrap a ciphertext with this key and thereby import a wrapped key into the state of the token.

Derived Attributes. The following attributes are derived from the other attributes:

$\boldsymbol{k}.\mathit{owner} \subseteq \mathbb{U}$: This attribute is derived from the access control list and denotes the set of users who may modify the attributes of $\boldsymbol{k}$. These users are called the *owners* of $\boldsymbol{k}$. Formally,

$$\boldsymbol{k}.\mathit{owner} \;=\; \bigl\{u \in \mathbb{U} \,\big|\, (u, \mathit{admin}) \in \boldsymbol{k}.\mathit{acl}\bigr\}.$$

$\boldsymbol{k}.\mathit{dependent} \subseteq \mathbb{G}$: This attribute denotes the set of keys whose secrecy depends on the secrecy of $\boldsymbol{k}$. When $\boldsymbol{k}$ is a secret key or a private key, the set $\boldsymbol{k}.\mathit{dependent}$ consists of all keys that depend on $\boldsymbol{k}$ according to Definition 1. When $\boldsymbol{k}$ is a public key, no other key depends on it and $\boldsymbol{k}.\mathit{dependent} = \emptyset$. Note that the token can easily compute $\boldsymbol{k}.\mathit{dependent}$ from $\mathbb{L}$.

$\boldsymbol{k}.\mathit{readers} \subseteq \mathbb{U}$: This attribute denotes the set of all users $u \in \mathbb{U}$ who are readers of $\boldsymbol{k}$ according to Definition 2. Note that the token can easily compute this from $\boldsymbol{k}.\mathit{dependent}$ and from $\mathbb{L}$.

4.3 Operations

In this section, we describe the operations that a user can execute with the token. The operations are grouped into *key-management operations* that create and delete keys or manipulate their attributes, and into *cryptographic operations* that use keys for protecting data or other keys. The operations for confidentiality protection (encryption and decryption) are applicable to both secret-key cryptosystems and public-key cryptosystems. Analogously, the operations for protecting authenticity and integrity (signatures and verification) are applicable to both secret-key authentication schemes (i.e., MACs) and public-key authentication schemes (i.e., digital signatures).

All operations are described with respect to a user u who is logged in to the token. In most cases, the token checks some preconditions before executing an operation. If all conditions are satisfied, the token executes the operation and returns some value; otherwise, the operation fails. An operation also fails if some specified key does not exist. For every operation o with input x and return value y that succeeds, the token adds the tuple (u, o, x, y) to the log $\mathbb{L}$. We assume that the user or another entity that invokes an operation of the token learns whether the operation succeeded or failed.

We define a function *usage* on a key $\boldsymbol{k}$ from $\mathbb{G}$ and on $\mathbb{L}$; it denotes the set of all cryptographic operations from $\mathbb{O}$ for which the key has been used according to the log. Formally,

$$\begin{aligned}\mathit{usage}(\boldsymbol{k}, \mathbb{L}) \;&= \bigl\{o \in \mathbb{O} \,\big|\, \exists\, v, x, y:\; (v, o, \boldsymbol{k}, y) \in \mathbb{L} \text{ or } (v, o, (\boldsymbol{k}, \cdots), y) \in \mathbb{L}\bigr\} \\ &\quad \setminus\; \bigl\{\mathit{create}, \mathit{read}, \mathit{delete}, \mathit{getattr}, \mathit{setattr}\bigr\}\end{aligned}$$

Key-management operations. We first present the operations that manipulate keys and their attributes.

$\mathit{create}(\boldsymbol{k}, \mathit{type}, [\boldsymbol{p}])$: This operation creates a new secret key or a new public key/private key pair. It takes as inputs an identifier $\boldsymbol{k} \in \mathbb{K} \setminus \mathbb{G}$, a key type from $\{\mathit{public}, \mathit{private}, \mathit{secret}\}$, and optionally a second key identifier $\boldsymbol{p} \in \mathbb{K} \setminus \mathbb{G}$.

If $\mathit{type} = \mathit{public}$, the operation fails.

If $\mathit{type} = \mathit{secret}$, the token generates the cryptographic value k as a random string drawn from $\mathcal{U}_\kappa$, creates a new key object, and stores the object in the token memory. It sets $\boldsymbol{k}.\mathit{type}$ to *secret*, $\boldsymbol{k}.\mathit{acl}$ to $\{(u, \mathit{admin})\}$, and $\boldsymbol{k}.\mathit{unextractable}$ to *false*. It adds $\boldsymbol{k}$ to $\mathbb{G}$ and returns $\boldsymbol{k}$.

If $\mathit{type} = \mathit{private}$, the token runs PKG and obtains a public key p and a corresponding private key k; then it creates two new key objects accordingly and stores them in the token memory. It sets

$\boldsymbol{p}$.*type* to *public*, $\boldsymbol{p}$.*priv* to $\boldsymbol{k}$, $\boldsymbol{p}$.*acl* to $\{(u, \mathit{admin})\}$, and $\boldsymbol{p}$.*unextractable* to *false*. Furthermore, it sets $\boldsymbol{k}$.*type* to *private*, $\boldsymbol{k}$.*pub* to $\boldsymbol{p}$, $\boldsymbol{k}$.*acl* to $\{(u, \mathit{admin})\}$, and $\boldsymbol{k}$.*unextractable* to *false*. It adds $\boldsymbol{p}$ and $\boldsymbol{k}$ to $\mathbb{G}$ and returns $(\boldsymbol{p}, \boldsymbol{k})$.

read$(\boldsymbol{k})$: This operation returns the cryptographic value of a key to the user. It takes a key $\boldsymbol{k} \in \mathbb{G}$ as input.

The operation verifies the following condition:

1. $\boldsymbol{k}$.*type* $=$ *public*; or
2. for all $\boldsymbol{l} \in \boldsymbol{k}$.*dependent*, it holds $(u, \mathit{read}) \in \boldsymbol{l}$.*acl*.

If the condition is true, the operation returns the cryptographic value k. Otherwise, the operation fails.

delete$(\boldsymbol{k})$: This operation deletes $\boldsymbol{k}$, its cryptographic value, and all its attributes. It takes a key $\boldsymbol{k} \in \mathbb{G}$ as input. If $(u, \mathit{admin}) \in \boldsymbol{k}$.*acl*, then the token removes the key object identified by $\boldsymbol{k}$ from its memory and destroys k. Otherwise, the operation fails.

getattr$(\boldsymbol{k})$: This operation takes a key $\boldsymbol{k} \in \mathbb{G}$ as input and returns a list of all its attributes.

setattr$(\boldsymbol{k}, A)$: This operation takes a key $\boldsymbol{k} \in \mathbb{G}$ and a list of attributes A as input, and replaces the attributes of $\boldsymbol{k}$ with those in A. Recall that the attributes of a key also include the access control list.

Let *unextractable*$'$ and *acl*$'$ denote the new values of the *unextractable* and *acl* attributes in A, respectively. The token verifies the following condition:

1. $(u, \mathit{admin}) \in \boldsymbol{k}$.*acl*; and
2. $\neg\boldsymbol{k}$.*unextractable* or *unextractable*$'$; and
3. if *unextractable*$'$, then there is no $v \in \mathbb{U}$ such that $(v, \mathit{admin}) \in \mathit{acl}'$ or $(v, \mathit{read}) \in \mathit{acl}'$; and
4. if $\boldsymbol{k}$.*type* $=$ *public* and $(v, \mathit{admin}) \in \mathit{acl}'$ for some v, then for the corresponding private key $\boldsymbol{l} = \boldsymbol{k}$.*priv*, it holds $(v, \mathit{admin}) \in \boldsymbol{l}$.*acl*; and
5. for all v such that $(v, \mathit{read}) \in \mathit{acl}'$ and for all $\boldsymbol{l} \in \boldsymbol{k}$.*dependent*, it holds $(v, \mathit{read}) \in \boldsymbol{l}$.*acl*; and
6. if $(\mathit{any}, p) \in \boldsymbol{k}.\mathit{acl}'$ for some $p \in \mathbb{P}$, then $(u, p) \in \boldsymbol{k}.\mathit{acl}'$ for all $u \in \mathbb{U}$.

If the condition is true, then the token replaces all modifiable attributes of $\boldsymbol{k}$ with those in A and returns *attributes_updated*. Otherwise, the operation fails.

The second clause makes sure that if $\boldsymbol{k}$.*unextractable* is already *true*, it cannot be set to *false*. The fifth clause ensures that a privilege to read $\boldsymbol{k}$ does not compromise the secrecy of any key $\boldsymbol{l}$ that depends on $\boldsymbol{k}$, by requiring that $\boldsymbol{l}$ already has the corresponding *read* privilege.

Cryptographic operations. We now present the operations that use keys in cryptographic operations such as encryption and integrity protection.

derive$(\boldsymbol{k}, \boldsymbol{l})$: This operation derives a new secret key $\boldsymbol{l} \in \mathbb{K} \setminus \mathbb{G}$ from a secret key $\boldsymbol{k} \in \mathbb{G}$. The token verifies the following:

1. $\boldsymbol{k}$.*type* $=$ *secret*; and

2. $(u, \textit{derive}) \in \boldsymbol{k}.\textit{acl}$; and
3. $\textit{usage}(\boldsymbol{k}, \mathbb{L}) \subseteq \{\textit{derive}\}$.

If the condition is true, the token computes the cryptographic value $l = \mathsf{F}_k(\boldsymbol{l})$ using the pseudorandom function. The token creates a new key object, and stores it in the token memory. It sets $\boldsymbol{l}$.*type* to *secret*, $\boldsymbol{l}$.*acl* to $\boldsymbol{k}$.*acl*, and $\boldsymbol{l}$.*unextractable* to $\boldsymbol{k}$.*unextractable* and returns $\boldsymbol{l}$. If the condition is false, the operation fails.

Note that the attributes of a derived key $\boldsymbol{l}$ are the same as those of $\boldsymbol{k}$.

encrypt$(\boldsymbol{k}, m)$: This operation encrypts m under $\boldsymbol{k}$; it takes a key $\boldsymbol{k} \in \mathbb{G}$ and a plaintext bit string m as input. The token verifies the following:

1. $\boldsymbol{k}$.*type* = *secret* or $\boldsymbol{k}$.*type* = *public*; and
2. $(u, \textit{encrypt}) \in \boldsymbol{k}.\textit{acl}$; and
3. $\textit{usage}(\boldsymbol{k}, \mathbb{L}) \subseteq \{\textit{encrypt}, \textit{decrypt}\}$.

If the condition is true, then the token computes $c = \mathsf{SE}(k, m)$ if $\boldsymbol{k}$.*type* = *secret*, or $c = \mathsf{PE}(k, m)$ if $\boldsymbol{k}$.*type* = *public*, and returns c; otherwise, the operation fails.

The third clause of the condition ensures that $\boldsymbol{k}$ has not been used for any cryptographic operations by the token other than *encrypt* and *decrypt*.

decrypt$(\boldsymbol{k}, c)$: This operation decrypts c with $\boldsymbol{k}$. It takes a key $\boldsymbol{k} \in \mathbb{G}$ and a ciphertext bit string c as input. The token verifies the following:

1. $\boldsymbol{k}$.*type* = *secret* or $\boldsymbol{k}$.*type* = *private*; and
2. $(u, \textit{decrypt}) \in \boldsymbol{k}.\textit{acl}$; and
3. $\textit{usage}(\boldsymbol{k}, \mathbb{L}) \subseteq \{\textit{encrypt}, \textit{decrypt}\}$.

If the condition is true, then the token computes $d = \mathsf{SD}(k, c)$ if $\boldsymbol{k}$.*type* = *secret*, or $d = \mathsf{PD}(k, c)$ if $\boldsymbol{k}$.*type* = *private*, and returns d; otherwise, the operation fails.

sign$(\boldsymbol{k}, m)$: This operation authenticates m with k and returns a digital signature or an authentication tag. It takes a key $\boldsymbol{k} \in \mathbb{G}$ and a message bit string m as input. The token verifies the following:

1. $\boldsymbol{k}$.*type* = *secret* or $\boldsymbol{k}$.*type* = *private*; and
2. $(u, \textit{sign}) \in \boldsymbol{k}.\textit{acl}$; and
3. $\textit{usage}(\boldsymbol{k}, \mathbb{L}) \subseteq \{\textit{sign}, \textit{verify}\}$.

When the condition is true, the token computes $t = \mathsf{SS}(k, m)$ if $\boldsymbol{k}$.*type* = *secret*, or $t = \mathsf{PS}(k, m)$ if $\boldsymbol{k}$.*type* = *private*, and returns t; otherwise, the operation fails.

verify$(\boldsymbol{k}, m, \sigma)$: This operation verifies σ on m under $\boldsymbol{k}$. It takes a key $\boldsymbol{k} \in \mathbb{G}$, a message bit string m, and a purported signature or authentication tag σ as input. The token verifies the following:

1. $\boldsymbol{k}$.*type* = *secret* or $\boldsymbol{k}$.*type* = *public*; and
2. $(u, \textit{verify}) \in \boldsymbol{k}.\textit{acl}$; and
3. $\textit{usage}(\boldsymbol{k}, \mathbb{L}) \subseteq \{\textit{sign}, \textit{verify}\}$.

If the condition is true, then the token computes $v = \mathsf{SV}(k, m, \sigma)$ if $\boldsymbol{k}$.*type* = *secret*, or $v = \mathsf{PV}(k, m, \sigma)$ if $\boldsymbol{k}$.*type* = *public*, and returns v; otherwise, the operation fails.

$\mathit{wrap}(\boldsymbol{k},\boldsymbol{l})$: This operation takes a wrapping key $\boldsymbol{k} \in \mathbb{G}$ and a target key $\boldsymbol{l} \in \mathbb{G}$ as input; it wraps $\boldsymbol{l}$ together with its cryptographic value under $\boldsymbol{k}$ and outputs the resulting ciphertext, which is called a wrapping. The token verifies the following:

1. $\boldsymbol{k}.\mathit{type} = \mathit{secret}$; and
2. $(u, \mathit{wrap}) \in \boldsymbol{k}.\mathit{acl}$; and
3. for all $v \in \boldsymbol{k}.\mathit{readers}$ and for all $\boldsymbol{j} \in \boldsymbol{l}.\mathit{dependent}$, it holds $(v, \mathit{read}) \in \boldsymbol{j}.\mathit{acl}$; and
4. $\boldsymbol{k} \notin \boldsymbol{l}.\mathit{dependent}$; and
5. $\neg \boldsymbol{l}.\mathit{unextractable}$; and
6. $\boldsymbol{l}.\mathit{type} = \mathit{private}$ or $\boldsymbol{l}.\mathit{type} = \mathit{secret}$; and
7. $\mathit{usage}(\boldsymbol{k}, \mathbb{L}) \subseteq \{\mathit{wrap}, \mathit{unwrap}\}$.

If the condition is true, the token computes the wrapping $(w, \ell) = \mathsf{W}(k, l, \langle \boldsymbol{l} \rangle)$ using the wrapping algorithm and returns (w, ℓ). Otherwise, the operation fails.

$\mathit{unwrap}(\boldsymbol{k}, w, \ell)$: This operation takes a wrapping key $\boldsymbol{k} \in \mathbb{G}$ and a wrapping (w, ℓ) as input; it unwraps the key with $\boldsymbol{k}$ and label ℓ to obtain a key and stores it in the token memory. The token verifies the following:

1. $\boldsymbol{k}.\mathit{type} = \mathit{secret}$; and
2. $(u, \mathit{unwrap}) \in \boldsymbol{k}.\mathit{acl}$; and
3. $\boldsymbol{k}.\mathit{readers} = \emptyset$; and
4. $\mathit{usage}(\boldsymbol{k}, \mathbb{L}) \subseteq \{\mathit{wrap}, \mathit{unwrap}\}$.

If the condition is true, the token parses ℓ as the representation $\langle \boldsymbol{l}' \rangle$ of the attributes for some key $\boldsymbol{l}'$; otherwise, the operation fails.

The token then checks if there is already a key with identifier $\boldsymbol{l}'$ in the token memory. If yes, the token verifies that the attributes of the two keys are equal. If they match, the operation is complete; otherwise, the operation fails.

If no key $\boldsymbol{l}'$ is present in the token memory, then the token computes $l' = \mathsf{U}(k, w, \ell)$. If $l' = \perp$, indicating that U failed, then the operation fails. Otherwise, the token creates a new key object $\boldsymbol{l}'$ with cryptographic value l', extracts the ACL attribute value acl' from $\langle \boldsymbol{l}' \rangle$, and verifies the following condition:

1. if $\boldsymbol{l}'.\mathit{type} = \mathit{public}$ and $(v, \mathit{admin}) \in \mathit{acl}'$ for some v, then for the corresponding private key $\boldsymbol{l} = \boldsymbol{l}'.\mathit{priv}$, it holds $(v, \mathit{admin}) \in \boldsymbol{l}.\mathit{acl}$; and
2. for all v such that $(v, \mathit{read}) \in \mathit{acl}'$ and for all $\boldsymbol{l} \in \boldsymbol{l}'.\mathit{dependent}$, it holds $(v, \mathit{read}) \in \boldsymbol{l}.\mathit{acl}$.

If the condition is true, the token stores the key object in its memory, sets the attribute values $\boldsymbol{l}'.\mathit{unextractable} = \mathit{false}$ and $\boldsymbol{l}'.\mathit{acl} = \mathit{acl}'$, and returns $\boldsymbol{l}'$.

Note that a key imported through unwrapping is never *unextractable*.

Remarks. The cryptographic operations provided by the token are implemented by primitives P. As every primitive satisfies the standard correctness condition from the literature, the token operations inherit these. Hence, when an operation *encrypt*$(\boldsymbol{k}, m)$ returns c, for example, a user with the corresponding privilege may call *decrypt*$(\boldsymbol{k}, c)$ and will obtain m.

It is easy to verify that the *depends on* relation on $\mathbb{G}$ forms a directed acyclic graph (DAG). With the *derive* operation alone, it is not possible to create circular dependencies among keys. Furthermore, suppose a key $\boldsymbol{j}$ depends on a key $\boldsymbol{l}$. Then it is not possible to make $\boldsymbol{l}$ dependent on $\boldsymbol{j}$ by wrapping $\boldsymbol{j}$ with $\boldsymbol{l}$ because the fourth clause in the pre-condition of the *wrap* operation rules this out. Note that since the *depends on* relation forms a DAG, no circularity is created through the encryption, wrapping, or deriving of keys. In other words, one never reaches the case where $\boldsymbol{k}$ depends on $\boldsymbol{l}$ as well as $\boldsymbol{l}$ depends on $\boldsymbol{k}$. This allows us the luxury of not having to use encryption schemes that are secure for encrypting key-dependent messages [9, 11].

The fifth clause in the pre-condition for the *setattr* operation requires that when the *read* privilege is assigned to some user for a key, all dependent keys must already have that privilege. Further invocations of *setattr* may be needed to achieve this, but because the *depends on* relation forms a DAG, these operations can be carried out beforehand.

Keys may be exported from the token through the *wrap* operation and imported again through *unwrap*. If an imported key still exists, unwrapping succeeds only if the attributes of the key have not changed. Otherwise, that is, when the key has been deleted meanwhile, unwrapping resets *acl* to the value from the time when the key was wrapped. Since ACL changes underly some conditions (as checked in the *setattr* operation), the token has to verify two of these conditions on *acl*$'$ during *unwrap*.

When a key $\boldsymbol{k}$ is deleted from the token, the entries in the log are unaffected. This means that $\boldsymbol{k}$ and all entries in the log relating to it matter for computing the *dependent* and *readers* attributes of all keys. Removing these entries from the log could undermine the token security policy. Moreover, a deleted key might come into existence again through an *unwrap* operation.

We note that owners of a key $\boldsymbol{k}$ can modify the access control list of $\boldsymbol{k}$ to provide encryption, decryption, and other privileges to other users in the system. This allows in particular for delegation of rights from the owner to other users. Users who are not owners of $\boldsymbol{k}$, however, cannot further delegate rights to other users.

It is now easy to see why our security policy prevents some common attacks on cryptographic tokens. Take a *sensitive* key according to PKCS #11: its value must not be revealed to any user. In our model, this corresponds to a key for which no user has *read* privilege. But as shown by Clulow [12] and by Delaune et al. [15] in their Experiments 1–3, PKCS #11 allows the key to be wrapped under a key whose value is known to a user. Our *wrap* operation explicitly checks that every user who knows the wrapping key has *read* privilege for the key being wrapped, and thereby prevents this line of attacks. Other attacks that involve importing keys to tokens, manipulating keys with XOR operations, or splitting 3DES keys into two DES keys [10, 12, 1], in PKCS #11 and in IBM CCA as implemented on the 4758 coprocessor, are ruled out by design because our token interface does not support the necessary operations.

4.4 Design Rationale

Our token model represents a compromise between the flexibility of the cryptographic interfaces used in practice and the goal of providing provable security guarantees. We explain some of our choices here.

We assume that every $\boldsymbol{k} \in \mathbb{G}$ has a unique cryptographic value except with negligible probability; in order to enforce this, the token provides no operations for importing, duplicating or renaming keys. All keys are created randomly by the token or derived from another key by a pseudo-random function. This assumption considerably simplifies the safety checks performed by the operations.

For most operations, there exists a corresponding privilege in $\mathbb{P}$; only the *admin* privilege applies to multiple operations. A user may only execute an operation if the access control list allows it.

Our conditions involving the *usage* of a key ensure that every key is used for a single cryptographic purpose: key creation (*derive*), data confidentiality (*encrypt* and *decrypt*), data authenticity (*sign* and *verify*), and key transport (*wrap* and *unwrap*). A key is not *a priori* typed for a particular purpose, as is the case in other key management systems. But once a key has been used for some goal, it must not be used for another one. This reflects the established good practice [5] and also simplifies the security analysis in our model.

The *wrap* and *unwrap* operations serve two purposes in practice: to transport keys between multiple tokens that share the wrapping key and to store keys in a trusted manner on external storage because the token memory is bounded. In our model that contains only one token, we use the latter motivation for key wrapping exclusively; but this covers all interesting facets of the problem.

For clarity of exposition, we do not consider wrapping with public keys here; but this could be added easily as mentioned in Section 7.

In order to guarantee the safety of the token operations, it is important that every key imported through an *unwrap* operation was created by the token and previously exported by the *wrap* operation. For this purpose, *wrap* authenticates its payload during export. This approach works only if the (un-)wrapping key $\boldsymbol{k}$ remains a secret of the token and is not known to any user, which is enforced by the condition that $\boldsymbol{k}.\mathit{readers} = \emptyset$ in the *unwrap* operation. Otherwise, if $\boldsymbol{k}$ was known to some user, and since the wrapping mechanism is public, the user might create a fake wrapping and import a key value and attributes that violate the consistency rules of the token. This is an example of so-called key-conjuring attacks that have been found in practical cryptographic token interfaces [10, 12, 3].

Keys with the *unextractable* attribute must remain on the token forever, and cannot be read or wrapped; also its ACL can no longer be modified. We provide this policy to model similar functionality in PKCS #11. The token implements two mechanisms to support this: First, the conditions for the *setattr* operation enforce that no user has *read* or *admin* privilege for an unextractable key. Second, keys marked *unextractable* cannot hence be deleted (because no user has *admin* privilege for an unextractable key). This is necessary since a user who unwraps a key may thereby reset its attributes to their earlier values, as mentioned before. Otherwise, if unwrapping of a (now) unextractable key were allowed, an adversary could reset the *unextractable* attribute and undermine the security policy.

We note that when a key $\boldsymbol{k}$ is marked unextractable, the token does not take an action to set the *unextractable* attribute of another key $\boldsymbol{l}$ on which $\boldsymbol{k}$ depends. But because no user may any longer have *read* privilege for $\boldsymbol{k}$, key $\boldsymbol{l}$ may neither be read nor wrapped with a key that has been read. Hence, when $\boldsymbol{k}$ was derived from $\boldsymbol{l}$, for example, it would be possible for a user to obtain a wrapping of key $\boldsymbol{l}$.

4.5 Implementation

Although our description of the log $\mathbb{L}$ implies that it grows with the number of operations executed by the token and its size is unbounded, this is only for ease of exposition. One actually needs to store only the following attributes for every key $\boldsymbol{k}$: $\boldsymbol{k}.\mathit{dependent}$, $\boldsymbol{k}.\mathit{readers}$, and $\mathit{usage}(\boldsymbol{k}, \mathbb{L})$. Note that the size of $\boldsymbol{k}.\mathit{dependent}$ is at most $|\mathbb{K}|$, the cardinality of $\mathbb{K}$, the size of $\boldsymbol{k}.\mathit{readers}$ is at most $|\mathbb{U}|$, and $\mathit{usage}(\boldsymbol{k}, \mathbb{L})$ is of constant size. This means that the log can be represented with $\mathcal{O}\big(|\mathbb{K}|(|\mathbb{K}| + |\mathbb{U}|)\big)$ space, independent of the number of operations executed by the token.

5 Token Security

This section presents the cryptographic security notion implemented by the token and shows that our implementation is secure.

5.1 Definition of Security

We define the security of a token through an experiment with three phases, called the *token security game*, that is run between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. Both are efficient algorithms. The challenger runs a token T implemented according to Section 4. During a few critical operations, $\mathcal{C}$ modifies the implementation of T for the experiment, but mostly $\mathcal{C}$ accesses T as black box. The modifications eliminate the capability of the adversary to obtain any symmetric key or private key directly, i.e., by executing a *read* operation for them. Hence, $\mathcal{A}$ should not be able to break the security of any cryptographic primitive. The experiment allows $\mathcal{A}$ to corrupt all users.

The challenger offers all token operations to $\mathcal{A}$ through the interface described next. $\mathcal{A}$ will have to guess whether it interacts with a real token or with one that simulates the output of all cryptographic operations. Recall that the set $\mathbb{K}$ denotes the set of all key identifiers, and the sets $\mathbb{S} \subseteq \mathbb{K}$ and $\mathbb{A} \subseteq \mathbb{K}$ denote the set of secret keys and asymmetric keys (public and private keys) that have been generated, respectively. We assume the challenger maintains $\mathbb{S}$ and $\mathbb{A}$.

Initialization phase: The challenger picks a bit b at random. $\mathcal{A}$ picks a set of users $\mathbb{U}$; it sends $\mathbb{U}$ to $\mathcal{C}$. The challenger initializes T with $\mathbb{U}$.

Query phase: The adversary can make queries to $\mathcal{C}$ for any $u \in \mathbb{U}$ and $\boldsymbol{k}, \boldsymbol{l}, \boldsymbol{p} \in \mathbb{K}$ according to the following list:

1. $(u, \textit{create}, \textit{secret}, \boldsymbol{k})$ for some $\boldsymbol{k} \notin \mathbb{S} \cup \mathbb{A}$. The challenger executes the $\textit{create}(\boldsymbol{k}, \textit{secret})$ operation of T with u logged in. If the operation is successful, $\mathcal{C}$ responds with $\boldsymbol{k}$ and adds $\boldsymbol{k}$ to $\mathbb{S}$. Otherwise, $\mathcal{C}$ responds with $\perp$.
2. $(u, \textit{create}, \textit{private}, (\boldsymbol{p}, \boldsymbol{k}))$ for some keys $\boldsymbol{p}, \boldsymbol{k} \notin \mathbb{S} \cup \mathbb{A}$. The challenger executes operation $\textit{create}(\boldsymbol{k}, \textit{private}, \boldsymbol{p})$ of T with u logged in. If the operation is successful, $\mathcal{C}$ responds with $(\boldsymbol{p}, \boldsymbol{k})$ and adds $\boldsymbol{p}$ and $\boldsymbol{k}$ to $\mathbb{A}$. Otherwise, $\mathcal{C}$ responds with $\perp$.
3. $(u, \textit{derive}, \boldsymbol{k}, \boldsymbol{l})$ for keys $\boldsymbol{k} \in \mathbb{S}$ and $\boldsymbol{l} \notin \mathbb{S} \cup \mathbb{A}$. If $b = 0$, then $\mathcal{C}$ executes operation $\textit{derive}(\boldsymbol{k}, \boldsymbol{l})$ of T with u logged in. If $b = 1$, then $\mathcal{C}$ also executes operation $\textit{derive}(\boldsymbol{k}, \boldsymbol{l})$ of T with u logged in, but modifies it such that the cryptographic value l of $\boldsymbol{l}$ is a random string drawn from $\mathcal{U}_\kappa$ (and not the output of $\mathsf{F}_k(\boldsymbol{l})$). In both cases, $\mathcal{C}$ responds with $\boldsymbol{l}$ if the operation is successful (and adds $\boldsymbol{l}$ to $\mathbb{S}$) and with $\perp$ otherwise.

In parallel $\mathcal{A}$ can also make the following queries for any $u \in \mathbb{U}$ and any $\boldsymbol{k} \in \mathbb{S} \cup \mathbb{A}$:

4. $(u, \textit{read}, \boldsymbol{k})$: $\mathcal{C}$ executes operation $\textit{read}(\boldsymbol{k})$ of T with u logged in; if the operation succeeds, then $\mathcal{C}$ responds with the return value of the token, and otherwise $\mathcal{C}$ responds with $\perp$.
5. $(u, \textit{delete}, \boldsymbol{k})$: $\mathcal{C}$ executes operation $\textit{delete}(\boldsymbol{k})$ of the token with u logged in; if the operation succeeds, then $\mathcal{C}$ responds with the return value of the token, and otherwise $\mathcal{C}$ responds with $\perp$.
6. $(u, \textit{getattr}, \boldsymbol{k})$: $\mathcal{C}$ executes operation $\textit{getattr}(\boldsymbol{k})$ of the token with u logged in; if the operation succeeds, then $\mathcal{C}$ responds with the return value of the token, and otherwise $\mathcal{C}$ responds with $\perp$.

7. $(u, \textit{setattr}, \boldsymbol{k}, A)$: $\mathcal{C}$ examines the new value *acl′* of the ACL in A, and checks if $(u, \textit{read}) \in$ *acl′* for a symmetric or a private key $\boldsymbol{k} \in \mathbb{S} \cup \mathbb{A}$ and any user u. If yes, $\mathcal{C}$ responds with $\perp$. Otherwise, $\mathcal{C}$ executes operation *setattr*$(\boldsymbol{k}, A)$ of the token with u logged in; if the operation succeeds, then $\mathcal{C}$ responds with the return value of the token, and otherwise $\mathcal{C}$ responds with $\perp$.

8. $(u, \textit{encrypt}, \boldsymbol{k}, m)$: $\mathcal{C}$ does the following. If $b = 0$, it runs operation *encrypt*$(\boldsymbol{k}, m)$ of the token with u logged in; otherwise, if $b = 1$, it runs operation *encrypt*$(\boldsymbol{k}, \mathfrak{d})$ of the token with u logged in. If the operation succeeds, then $\mathcal{C}$ responds with the return value of the token, and otherwise $\mathcal{C}$ responds with $\perp$.

9. $(u, \textit{decrypt}, (\boldsymbol{k}, c))$: $\mathcal{C}$ does the following. If c was never a response from $\mathcal{C}$ to $\mathcal{A}$ to a query of the form $(v, \textit{encrypt}, \boldsymbol{k}, m)$ for $\boldsymbol{k} \in \mathbb{S}$ (or $\boldsymbol{l} \in \mathbb{A}$ that corresponds to $\boldsymbol{k} \in \mathbb{A}$) for some $v \in \mathbb{U}$ and message m, then $\mathcal{C}$ runs operation *decrypt*$(\boldsymbol{k}, c)$ of the token with u logged in; if the operation succeeds, then $\mathcal{C}$ responds with the return value of the token. In any other case, $\mathcal{C}$ responds with $\perp$.

10. $(u, \textit{sign}, \boldsymbol{k}, m)$: $\mathcal{C}$ executes operation *sign*$(\boldsymbol{k}, m)$ of the token with u logged in; if the operation succeeds, then $\mathcal{C}$ responds with the return value of the token, and otherwise $\mathcal{C}$ responds with $\perp$.

11. $(u, \textit{verify}, \boldsymbol{k}, m, \sigma)$: $\mathcal{C}$ executes operation *verify*$(\boldsymbol{k}, m, \sigma)$ of the token with u logged in; if the operation succeeds, then $\mathcal{C}$ responds with the return value of the token, and otherwise $\mathcal{C}$ responds with $\perp$.

12. $(u, \textit{wrap}, \boldsymbol{k}, \boldsymbol{l})$: $\mathcal{C}$ first runs operation *wrap*$(\boldsymbol{k}, \boldsymbol{l})$ of the token with u logged in. If the operation fails, then $\mathcal{C}$ responds with $\perp$. Otherwise, if $b = 0$, then $\mathcal{C}$ takes (w, ℓ) to be the return value from the *wrap* operation; if $b = 1$, then $\mathcal{C}$ computes $\ell = \langle \boldsymbol{l} \rangle$ and $w = \mathsf{W}(k, \mathfrak{d}, \ell)$. Finally, $\mathcal{C}$ responds with (w, ℓ) to $\mathcal{A}$.

13. $(u, \textit{unwrap}, \boldsymbol{k}, w, \ell)$: $\mathcal{C}$ does the following. If $b = 0$ and (w, ℓ) was never a response from $\mathcal{C}$ to $\mathcal{A}$ to a query of the form $(v, \textit{wrap}, \boldsymbol{k}, \boldsymbol{l})$ for some $v \in \mathbb{U}$ and some $\boldsymbol{l} \in \mathbb{S} \cup \mathbb{A}$, then $\mathcal{C}$ runs operation *unwrap*$(\boldsymbol{k}, w, \ell)$ of the token with u logged in; if the operation succeeds, then $\mathcal{C}$ responds with the return value of the token. In any other case, i.e., if the operation fails or $b = 1$, then $\mathcal{C}$ responds with $\perp$.

Final phase: $\mathcal{A}$ does one of the following:

1. $\mathcal{A}$ outputs a bit b^*. Let the advantage α of $\mathcal{A}$ be defined as $\left|\Pr[b^* = b] - \frac{1}{2}\right|$.

2. $\mathcal{A}$ outputs a triple $(\boldsymbol{k}, m^*, \sigma^*)$ for $\boldsymbol{k} \in \mathbb{S}$ such that σ^* was never returned by $\mathcal{C}$ as the response to a query $(u, \textit{sign}, \boldsymbol{k}, m^*)$ with $\boldsymbol{k}$ and any $u \in \mathbb{U}$. Let the advantage α of $\mathcal{A}$ denote the probability that when $\mathcal{C}$ runs operation *verify*(k, m^*, σ^*) of the token, the return value is *accept*.

3. $\mathcal{A}$ outputs a triple $(\boldsymbol{p}, m^*, \sigma^*)$ for a public key $\boldsymbol{p} \in \mathbb{A}$ such that σ^* was never returned by $\mathcal{C}$ as the response to a query $(u, \textit{sign}, \boldsymbol{k}, m^*)$ with the private key $\boldsymbol{k} \in \mathbb{A}$ that corresponds to $\boldsymbol{p}$ and any $u \in \mathbb{U}$. Let the advantage α of $\mathcal{A}$ denote the probability that $\mathsf{PV}(p, m^*, \sigma^*) = \textit{accept}$.

Definition 3 (Secure token). *We say that a cryptographic token is secure if for all efficient adversaries $\mathcal{A}$ the advantage α is negligible.*

Remarks. The security of the cryptographic operations of the token (encryption, authentication, and wrapping) depends on the condition that $\mathcal{A}$ obtains no cryptographic value for any key. According to the check by the *setattr* query, the adversary may never add the *read* privilege to the ACL of any key. A secure token ensures that without this privilege, in particular, the key cannot become known to the adversary, and, more generally, that all cryptographic operations with the key are secure.

The adversary may try to break the security of the token in two different ways, corresponding the outputs of $\mathcal{A}$ in the final phase:

1. by distinguishing the legitimate output of the key derivation function from uniformly random keys, or by distinguishing legitimate encryptions from encryptions of a dummy message, or by distinguishing legitimate wrappings from wrappings of a dummy message; or

2. by contradicting the integrity of either the symmetric authentication scheme or the public-key signature scheme through a forged signature.

Note how the *wrap* operation of the token is required to authenticate its payload and to provide ciphertext integrity. In particular, if $\mathcal{A}$ succeeds in forging a wrapping (w^*, ℓ^*) that was not computed by *wrap*, and includes (w^*, ℓ^*) in an *unwrap* query to $\mathcal{C}$ such that the *unwrap* operation of the token succeeds, then the response from the challenger trivially discloses that $b = 0$ with probability one half.

5.2 Analysis

Theorem 1. *Let* P *be a set of cryptographic primitives according to Section 3. Then the cryptographic token according to Section 4 using the primitives* P *is secure.*

Let $\mathsf{P} = \{\mathsf{F}, \mathsf{SENC}, \mathsf{PENC}, \mathsf{SAUTH}, \mathsf{PAUTH}, \mathsf{WRAP}\}$ be a set of cryptographic primitives secure according to the definitions in Section 3. We first define a game that combines all primitives and use it to define their joint security.

Primitives game. Consider the following three-phase experiment between a challenger $\mathcal{C}_\mathsf{P}$ and an adversary $\mathcal{A}_\mathsf{P}$; both are efficient algorithms.

Let $\mathbb{K}$ denote the set of key identifiers used by the primitives. Some $\boldsymbol{k} \in \mathbb{K}$ represents only an identifier of a key, its cryptographic value is denoted by k and not part of $\boldsymbol{k}$. The challenger $\mathcal{C}_\mathsf{P}$ keeps track of two set $\mathbb{S}, \mathbb{A} \subseteq \mathbb{K}$, which denote the secret keys and the asymmetric keys (public and private keys) that have been *generated* in the experiment, respectively. The sets $\mathbb{S}$ and $\mathbb{A}$ are empty initially.

Initialization phase: $\mathcal{C}_\mathsf{P}$ picks a bit b at random.

Query phase: The adversary interacts with the challenger through a number of queries.

Throughout the query phase, $\mathcal{C}_\mathsf{P}$ imposes the following restriction: $\mathcal{A}_\mathsf{P}$ must use a particular symmetric key only in queries relating to one of the four schemes; for example, a key used once in a symmetric encryption or decryption query must not be used for key derivation, signing, verifying, wrapping, or unwrapping queries. Analogously, $\mathcal{A}_\mathsf{P}$ must use the public and the private keys of a particular key pair only with either encryption and decryption queries or with signing and verification queries.

For any $\boldsymbol{k} \in \mathbb{K}$, the adversary $\mathcal{A}_\mathsf{P}$ may invoke the following query:

1. A *key creation* query for $\boldsymbol{k}$: If $\boldsymbol{k} \notin \mathbb{S} \cup \mathbb{A}$, then $\mathcal{C}_\mathsf{P}$ adds $\boldsymbol{k}$ to $\mathbb{S}$, sets its cryptographic value k to a random string drawn from $\mathcal{U}_\kappa$, and responds with *done*. Otherwise, $\mathcal{C}_\mathsf{P}$ responds with $\perp$.

For any $\boldsymbol{p}, \boldsymbol{k} \in \mathbb{K}$, the adversary $\mathcal{A}_{\mathsf{P}}$ may invoke the following query:

2. A *key-pair creation* query for $(\boldsymbol{p}, \boldsymbol{k})$: If $\boldsymbol{p}, \boldsymbol{k} \notin \mathbb{S} \cup \mathbb{A}$, then $\mathcal{C}_{\mathsf{P}}$ adds $\boldsymbol{p}$ and $\boldsymbol{k}$ to $\mathbb{A}$, runs PKG, sets the cryptographic values (p, k) to the output of PKG, and returns the public-key value p to $\mathcal{A}_{\mathsf{P}}$. Otherwise, $\mathcal{C}_{\mathsf{P}}$ responds with $\perp$.

   For key pairs created like this, we say that $\boldsymbol{k}$ *corresponds to* $\boldsymbol{p}$ and vice versa.

Adversary $\mathcal{A}_{\mathsf{P}}$ may invoke the following queries:

3. A *key derivation* query from a base key $\boldsymbol{k} \in \mathbb{S}$ to a derived key $\boldsymbol{l} \in \mathbb{K}$: If $\boldsymbol{l} \in \mathbb{S} \cup \mathbb{A}$, then $\mathcal{C}_{\mathsf{P}}$ responds with $\perp$. If $b = 0$, then $\mathcal{C}_{\mathsf{P}}$ sets the cryptographic value l to $\mathsf{F}_k(\boldsymbol{l})$; otherwise, $\mathcal{C}_{\mathsf{P}}$ sets l to a random string drawn from $\mathcal{U}_\kappa$. Then $\mathcal{C}_{\mathsf{P}}$ adds $\boldsymbol{l}$ to $\mathbb{S}$ and responds with *done*.
4. An *encryption query* $\mathsf{E}(\boldsymbol{k}, m)$ for $\boldsymbol{k} \in \mathbb{S}$ or a public key $\boldsymbol{k} \in \mathbb{A}$: If $b = 0$, then $\mathcal{C}_{\mathsf{P}}$ responds with $\mathsf{SE}(k, m)$ if $\boldsymbol{k} \in \mathbb{S}$ and with $\mathsf{PE}(k, m)$ if $\boldsymbol{k} \in \mathbb{A}$. Otherwise, if $b = 1$, then $\mathcal{C}_{\mathsf{P}}$ responds with $\mathsf{SE}(k, \mathfrak{d})$ if $\boldsymbol{k} \in \mathbb{S}$ and with $\mathsf{PE}(k, \mathfrak{d})$ if $\boldsymbol{k} \in \mathbb{A}$.
5. A *decryption query* $\mathsf{D}(\boldsymbol{k}, c)$ for $\boldsymbol{k} \in \mathbb{S}$ or a private key $\boldsymbol{k} \in \mathbb{A}$: If c was never returned by $\mathcal{C}_{\mathsf{P}}$ as the response to some encryption query under key $\boldsymbol{k}$ or under the public key $\boldsymbol{p} \in \mathbb{A}$ that corresponds to $\boldsymbol{k}$, then $\mathcal{C}_{\mathsf{P}}$ responds with $\mathsf{SD}(k, c)$ if $\boldsymbol{k} \in \mathbb{S}$ and with $\mathsf{PD}(k, c)$ if $\boldsymbol{k} \in \mathbb{A}$. Otherwise, $\mathcal{C}_{\mathsf{P}}$ responds with $\perp$.
6. A *signing query* $\mathsf{S}(\boldsymbol{k}, m)$ for $\boldsymbol{k} \in \mathbb{S}$ or a private key $\boldsymbol{k} \in \mathbb{A}$: $\mathcal{C}_{\mathsf{P}}$ responds with $\mathsf{SS}(k, m)$ if $\boldsymbol{k} \in \mathbb{S}$ and with $\mathsf{PS}(k, m)$ if $\boldsymbol{k} \in \mathbb{A}$.
7. A *verification query* $\mathsf{V}(\boldsymbol{k}, m, \sigma)$ for $\boldsymbol{k} \in \mathbb{S}$: $\mathcal{C}_{\mathsf{P}}$ responds with $\mathsf{SV}(k, m, \sigma)$.
8. A *wrapping query* $\mathsf{W}(\boldsymbol{k}, \boldsymbol{l}, \ell)$ for $\boldsymbol{k} \in \mathbb{S}$ and for a symmetric key $\boldsymbol{l} \in \mathbb{S}$ or a private key $\boldsymbol{l} \in \mathbb{A}$: If $b = 0$, then $\mathcal{C}_{\mathsf{P}}$ computes a value w as $\mathsf{W}(k, l, \ell)$. Otherwise, if $b = 1$, then $\mathcal{C}_{\mathsf{P}}$ computes w as $\mathsf{W}(k, \mathfrak{d}, \ell)$. Finally, $\mathcal{C}_{\mathsf{P}}$ responds with (w, ℓ).
9. An *unwrapping query* $\mathsf{U}(\boldsymbol{k}, w, \ell)$ for $\boldsymbol{k} \in \mathbb{S}$: If $b = 0$ and (w, ℓ) was never returned by $\mathcal{C}_{\mathsf{P}}$ as the response to a wrapping query with first argument equal to $\boldsymbol{k}$, then $\mathcal{C}_{\mathsf{P}}$ responds with $\mathsf{U}(k, w, \ell)$. Otherwise, $\mathcal{C}_{\mathsf{P}}$ responds with $\perp$.

Final phase: $\mathcal{A}_{\mathsf{P}}$ does one of the following:

1. $\mathcal{A}_{\mathsf{P}}$ outputs a bit b^*. Let the advantage α of $\mathcal{A}_{\mathsf{P}}$ be defined as $\left|\Pr[b^* = b] - \frac{1}{2}\right|$.
2. $\mathcal{A}_{\mathsf{P}}$ outputs a triple $(\boldsymbol{k}, m^*, \sigma^*)$ for $\boldsymbol{k} \in \mathbb{S}$ such that σ^* was never returned by $\mathcal{C}_{\mathsf{P}}$ as the response to a signing query with m^* for $\boldsymbol{k}$. Let the advantage α of $\mathcal{A}_{\mathsf{P}}$ denote the probability that $\mathsf{SV}(k, \sigma^*, m^*)$ outputs *accept*.
3. $\mathcal{A}_{\mathsf{P}}$ outputs a triple $(\boldsymbol{p}, m^*, \sigma^*)$ for a public key $\boldsymbol{p} \in \mathbb{A}$ such that σ^* was never returned by $\mathcal{C}$ as the response to a signing query with m^* for the key $\boldsymbol{k} \in \mathbb{A}$ that corresponds to $\boldsymbol{p}$. Let the advantage α of $\mathcal{A}_{\mathsf{P}}$ denote the probability that $\mathsf{PV}(p, m^*, \sigma^*) = \textit{accept}$.

Note that $\mathcal{A}_{\mathsf{P}}$ obtains all public keys and can generate encryptions and verify signatures on its own. The query to encrypt under a public key serves to produce challenge ciphertexts.

Lemma 2. *Let* $\mathsf{P} = \{\mathsf{F}, \mathsf{SENC}, \mathsf{PENC}, \mathsf{SAUTH}, \mathsf{PAUTH}, \mathsf{WRAP}\}$ *be a set of cryptographic primitives according to Section 3. Then the advantage of any efficient adversary* $\mathcal{A}_{\mathsf{P}}$ *in the above primitives game is negligible.*

Proof sketch. We have to show that the advantage α of any efficient adversary in the primitives game is negligible in the security parameter κ.

The proof of this lemma follows from a simple hybrid argument. We outline a sketch of the proof below. Let p_i be the advantage of the adversary $\mathcal{A}_{\mathsf{P}}$ in Game i below.

Game 0. This game is the primitives game with $b = 0$.

Game 1. This game is exactly like Game 0, except that when responding to an unwrap query $\mathsf{U}(\boldsymbol{k}, w, \ell)$ for $\boldsymbol{k} \in \mathbb{S}$, $\mathcal{C}_{\mathsf{P}}$ always responds with $\perp$.

By the ciphertext integrity of the authenticated encryption scheme WRAP, it follows that $|p_1 - p_0|$ is negligible in κ.

Game 2. This game is exactly like Game 1, except that when responding to a wrap query $\mathsf{W}(\boldsymbol{k}, \boldsymbol{l}, \ell)$ for $\boldsymbol{k} \in \mathbb{S}$ and for a symmetric key $\boldsymbol{l} \in \mathbb{S}$ or a private key $\boldsymbol{l} \in \mathbb{A}$, $\mathcal{C}_{\mathsf{P}}$ always computes $w = \mathsf{W}(k, \mathfrak{d}, \ell)$ and responds with (w, ℓ).

By the semantic security of the authenticated encryption scheme WRAP, it follows that $|p_2 - p_1|$ is negligible. Once again, note that since the *depends on* relation between keys forms a DAG, we do not require an encryption scheme secure against key-dependent message encryptions as no circularity is encountered.

Game 3. This game is exactly like Game 2, except that when responding to an encryption query $\mathsf{E}(\boldsymbol{k}, m)$ for $\boldsymbol{k} \in \mathbb{S}$ or a public key $\boldsymbol{k} \in \mathbb{A}$, $\mathcal{C}_{\mathsf{P}}$ always responds with $\mathsf{SE}(k, \mathfrak{d})$ if $\boldsymbol{k} \in \mathbb{S}$ and with $\mathsf{PE}(k, \mathfrak{d})$ if $\boldsymbol{k} \in \mathbb{A}$.

By the adaptive chosen ciphertext security of the encryption schemes SENC and PENC, it follows that $|p_3 - p_2|$ is negligible.

Game 4. This game is exactly like Game 3, except that when responding to responding to a key derivation query from a base key $\boldsymbol{k}$ to a derived key $\boldsymbol{l}$, $\mathcal{C}_{\mathsf{P}}$ always sets l to to a random string drawn from $\mathcal{U}_{\kappa}$.

By the adaptive security of the pseudorandom function family F, it follows that $|p_4 - p_3|$ is negligible.

Clearly, Game 4 is the primitives game with $b = 1$. Let α_f be the probability that in the primitives game, $\mathcal{A}_{\mathsf{P}}$ outputs a triple $(\boldsymbol{k}, m^*, \sigma^*)$ for $\boldsymbol{k} \in \mathbb{S}$ such that σ^* was never returned by $\mathcal{C}_{\mathsf{P}}$ as the response to a signing query with m^* for $\boldsymbol{k}$ or that $\mathcal{A}_{\mathsf{P}}$ outputs a triple $(\boldsymbol{p}, m^*, \sigma^*)$ for $\boldsymbol{p} \in \mathbb{A}$ such that σ^* was never returned by $\mathcal{C}_{\mathsf{P}}$ as the response to a signing query with m^* for the key $\boldsymbol{k} \in \mathbb{A}$ that corresponds to $\boldsymbol{p}$. Let α_d be the probability that $\mathcal{A}_{\mathsf{P}}$ in the primitives game outputs a bit b^* such that $b^* = b$. Clearly, $\alpha = \alpha_f + \alpha_d$. Now, $\alpha_d = |p_4 - p_0|$ and this is negligible by the hybrid argument shown above. From combining the strong existential unforgeability conditions of SAUTH and of PAUTH, it follows that α_f is negligible. Hence, α is also negligible. □

We now show the security of our token by reducing it to the security of the primitives games. In other words, given an adversary $\mathcal{A}$ that succeeds in the security game of the token (described in Section 5) with non-negligible probability, we construct an adversary $\mathcal{A}_{\mathsf{P}}$, that succeeds in the security game of the primitives game (described above), also with non-negligible probability. We note that $\mathcal{A}$ simply announces the user under which it logs in and authenticates to the token. Since the adversary controls all users of the token and we do not concern ourselves with the security of the authentication mechanism, we assume that user authentication is ideal.

Proof sketch for Theorem 1. Adversary $\mathcal{A}_{\mathsf{P}}$ interacts with a primitives-game challenger $\mathcal{C}_{\mathsf{P}}$ as an adversary and simulates to $\mathcal{A}$ an interaction with challenger $\mathcal{C}$ such that $\mathcal{A}$ cannot distinguish this interaction with $\mathcal{A}_{\mathsf{P}}$ from an interaction with $\mathcal{C}$ according to the token security game.

We now describe the interaction of $\mathcal{C}_{\mathsf{P}}$ with $\mathcal{A}_{\mathsf{P}}$ and the interaction of $\mathcal{A}_{\mathsf{P}}$ with $\mathcal{A}$ simulating $\mathcal{C}$ with the token. $\mathcal{A}_{\mathsf{P}}$ works by running a token T and replacing the cryptographic primitives in the token with challenger $\mathcal{C}_\mathsf{P}$. More precisely, whenever the simulated token calls one of the cryptographic primitives, $\mathcal{A}_\mathsf{P}$ queries $\mathcal{C}_\mathsf{P}$ to obtain the response.

The token implementation should ensure that $\mathcal{A}_\mathsf{P}$ does not learn the value of any symmetric key or any private key, and that $\mathcal{A}_\mathsf{P}$ cannot break the security notions of any of the cryptographic primitives that the token provides.

Initialization phase: $\mathcal{C}_\mathsf{P}$ picks a bit b at random. $\mathcal{A}_\mathsf{P}$ acts as a challenger $\mathcal{C}$ in the game with $\mathcal{A}$ and maintains the set of generated symmetric keys $\mathbb{S}$ and public/private keys $\mathbb{A}$. Initially, $\mathbb{S}$ and $\mathbb{A}$ are empty. When $\mathcal{A}$ picks a set of users $\mathbb{U}$ and sends $\mathbb{U}$ to $\mathcal{A}_\mathsf{P}$, then $\mathcal{A}_\mathsf{P}$ initializes T with $\mathbb{U}$. $\mathcal{A}_\mathsf{P}$ will simulate the responses to $\mathcal{A}$ without knowing the value of any of the keys in $\mathbb{S} \cup \mathbb{A}$.

Query phase: $\mathcal{A}$ may make any of the queries described in the query phase of the token security game according to Definition 3. For most queries, $\mathcal{A}_\mathsf{P}$ performs exactly the same operations to compute its responses as $\mathcal{C}$ does in the token security game (except for replacing the cryptographic primitives with calls to $\mathcal{C}_\mathsf{P}$). In particular, this means that no *setattr* query is allowed with an ACL that includes a read privilege for a symmetric key or for a private key. Only for certain queries, $\mathcal{A}_\mathsf{P}$ behaves differently, as described next:

1. $(u, \mathit{read}, \boldsymbol{k})$: $\mathcal{A}_\mathsf{P}$ executes operation $\mathit{read}(\boldsymbol{k})$ of T with u logged in; if the operation succeeds and $\boldsymbol{k} \in \mathbb{S}$ or $\boldsymbol{k} \in \mathbb{A}$ is a private key, then $\mathcal{A}_\mathsf{P}$ responds with *FAIL* to $\mathcal{A}$ and aborts the simulation. Otherwise, $\mathcal{A}_\mathsf{P}$ responds with the return value of T to $\mathcal{A}$.
2. $(u, \mathit{unwrap}, \boldsymbol{k}, w, \ell)$: $\mathcal{A}_\mathsf{P}$ does the following. If (w, ℓ) was never a response from $\mathcal{A}_\mathsf{P}$ to $\mathcal{A}$ to a query of the form $(v, \mathit{wrap}, \boldsymbol{k}, \boldsymbol{l})$ for some $v \in \mathbb{U}$ and some $\boldsymbol{l} \in \mathbb{S} \cup \mathbb{A}$, then $\mathcal{A}_\mathsf{P}$ runs operation $\mathit{unwrap}(\boldsymbol{k}, w, \ell)$ of T with u logged in. If the operation succeeds and returns a key $\boldsymbol{l}' \neq \bot$, then $\mathcal{A}_\mathsf{P}$ proceeds to the final phase in the primitives game, outputs the bit 0 to $\mathcal{C}_\mathsf{P}$, and halts. Otherwise, if the *unwrap* operation fails, $\mathcal{A}_\mathsf{P}$ responds with $\bot$ to $\mathcal{A}$.

Final phase: Finally, $\mathcal{A}$ may do one of the following:

1. $\mathcal{A}$ outputs a bit b^*. In this case, $\mathcal{A}_\mathsf{P}$ outputs b^* to challenger $\mathcal{C}_\mathsf{P}$.
2. $\mathcal{A}$ outputs a triple $(\boldsymbol{k}, m^*, \sigma^*)$ for $\boldsymbol{k} \in \mathbb{S}$ such that σ^* was never returned by $\mathcal{A}_\mathsf{P}$ as the response to a query $(u, \mathit{sign}, \boldsymbol{k}, m^*)$ with $\boldsymbol{k}$ and any $u \in \mathbb{U}$. Then $\mathcal{A}_\mathsf{P}$ outputs $(\boldsymbol{k}, m^*, \sigma^*)$ to challenger $\mathcal{C}_\mathsf{P}$.
3. $\mathcal{A}$ outputs a triple $(\boldsymbol{p}, m^*, \sigma^*)$ for public key $\boldsymbol{p} \in \mathbb{A}$ such that σ^* was never returned by $\mathcal{A}_\mathsf{P}$ as the response to a query $(u, \mathit{sign}, \boldsymbol{k}, m^*)$ with the private key $\boldsymbol{k} \in \mathbb{A}$ that corresponds to $\boldsymbol{p}$ and any $u \in \mathbb{U}$ and. Then $\mathcal{A}_\mathsf{P}$ outputs $(\boldsymbol{k}, m^*, \sigma^*)$ to challenger $\mathcal{C}_\mathsf{P}$.

This completes the description of adversary $\mathcal{A}_\mathsf{P}$. Note that for answering many of $\mathcal{A}$'s queries, $\mathcal{A}_\mathsf{P}$ proceeds according to the token security game and executes an operation of the simulated token T. This involves the *depends on* relation for the *setattr* and for the *wrap* queries.

We now analyze the advantage of $\mathcal{A}_\mathsf{P}$ in the primitives game. We show that $\mathcal{A}_\mathsf{P}$ never responds with *FAIL* to $\mathcal{A}$ and aborts the simulation. This holds because $\mathcal{A}_\mathsf{P}$ responds with *FAIL* only when operation $\mathit{read}(\boldsymbol{k})$ of T with $u \in \mathbb{U}$ logged in succeeds for some $\boldsymbol{k} \in \mathbb{S}$ or private key $\boldsymbol{k} \in \mathbb{A}$. According to the specification of T, in particular the conditions that T verifies during the *read* operation, this succeeds only if $(u, \mathit{read}) \in \boldsymbol{k}.\mathit{acl}$. Note that this condition relies on the *depends on* relation.

There are two ways in which the value of $\boldsymbol{k}.\mathit{acl}$ may be modified:

1. With a *setattr* query: But this is not possible because $\mathcal{A}_\mathsf{P}$ explicitly rejects any *setattr* query with a value for $\boldsymbol{k}.\mathit{acl}$ that contains (u, read) for some $\boldsymbol{k} \in \mathbb{S}$ or some private key $\boldsymbol{k} \in \mathbb{A}$.

2. Through a successful *unwrap* query that creates key $\boldsymbol{k}$ in the token such that $\boldsymbol{k}.\mathit{acl}$ contains (u, read): But this is not possible, as we show by contradiction. Recall that the attributes of an unwrapped key are taken from the label ℓ of the wrapping. Regardless of whether $\boldsymbol{k}$ ever existed in T before, the *unwrap* query in this case is invoked with a value ℓ that was never part of a response from $\mathcal{A}_\mathsf{P}$ to a *wrap* query. Since the *unwrap* operation of T succeeded and added $\boldsymbol{k}$ to the token memory, $\mathcal{A}_\mathsf{P}$ would have output bit 0 to $\mathcal{C}_\mathsf{P}$ and halted.

Therefore, $\mathcal{A}_\mathsf{P}$ never outputs *FAIL* in the simulation.

Inspecting the conditions on the queries to $\mathcal{C}_\mathsf{P}$ invoked by $\mathcal{A}_\mathsf{P}$, it is easy to see that $\mathcal{A}_\mathsf{P}$ respects all rules of the primitives game and simulates the token game perfectly. It follows that the advantage of $\mathcal{A}_\mathsf{P}$ in the primitives game is at least the advantage of $\mathcal{A}$ in the token security game. In other words, the security policy implemented by the token preserves the insecurity of the cryptographic primitives perfectly. □

6 Emulation of PKCS #11

In this section, we briefly discuss how our token model can emulate the key management functions in the RSA PKCS #11 standard [25] in a way that respects our security policy. Because PKCS #11 allows some attacks through the interface (consult the paper of Delaune et al. [15] for examples), we have to omit some operations and restrict others. The goal is thus to identify a subset of PKCS #11 features that is *safe* according to our token model.

We begin with the attributes of PKCS #11, then address the cryptographic operations and key-management functions of PKCS #11, and finally examine the security features of PKCS #11 and show that they are achieved by our emulation.

PKCS #11 distinguishes between ordinary users, who all have the same permissions for accessing keys, and a *security officer* or *SO user* with additional privileges. It is therefore sufficient to consider only two users and a set $\mathbb{U} = \{\mathit{user}, \mathit{SO}\}$ in our emulation.

Attributes. Key objects in PKCS #11 may be persistent token objects or volatile session objects; we only address token objects. For most of their attributes, there is a straightforward one-to-one correspondence between PKCS #11 and our token model. For example, the CKA_ID and CKA_KEY_TYPE attributes in PKCS #11 are emulated by our key identifier $\boldsymbol{k}$ and the $\boldsymbol{k}.\mathit{type}$ attribute, respectively, and serve the same purpose. However, in contrast to PKCS #11, the value of CKA_ID cannot be modified in our emulation once it has been assigned.

Keys in PKCS #11 have several attributes that govern their cryptographic usage, like CKA_DERIVE, CKA_ENCRYPT, CKA_DECRYPT, and so on. They determine if the key may be used for the corresponding cryptographic operation. Our token model includes the same privileges (with the exception of CKA_SIGN_RECOVER and CKA_VERIFY_RECOVER) and the emulation maintains them in a one-to-one way in the ACL attribute; that is, a key $\boldsymbol{k}$ has CKA_ENCRYPT set in PKCS #11 if and only if $\{\mathit{user}, \mathit{encrypt}\} \in \boldsymbol{k}.\mathit{acl}$ in our token, and so on. Our emulation does not permit a key to be used in multiple cryptographic primitives according to the *usage* conditions.

The CKA_SENSITIVE attribute in PKCS #11 denotes whether a key $\boldsymbol{k}$ is *sensitive* in the sense that its cryptographic value must not be read by any user, i.e., the key “cannot be revealed in plaintext off the token” [25]. We model this by removing the *read* privilege for *user* from $\boldsymbol{k}.\mathit{acl}$ and by restricting the *admin* privilege to user *SO* whenever CKA_SENSITIVE is *true*.

The CKA_EXTRACTABLE attribute denotes if the cryptographic value of $\boldsymbol{k}$ can be *extracted* and may ever leave the token (in plaintext or wrapped), i.e., an unextractable key “cannot be revealed off the token even when encrypted” [25]. We model this by the attribute $\boldsymbol{k}.\mathit{unextractable}$ that takes on the inverse value of CKA_EXTRACTABLE.

Keys in PKCS #11 have two further attributes CKA_ALWAYS_SENSITIVE and CKA_NEVER_EXTRACTABLE that denote if a key has always been sensitive and if a key has never been extractable, respectively. Their goal is to give information about the past of a key. Since our token model is richer, we know more about the history and emulate them with slightly different semantics from the *readers* of a key:

- We let the CKA_ALWAYS_SENSITIVE attribute of $\boldsymbol{k}$ be *true* whenever $\boldsymbol{k}.\mathit{readers} = \emptyset$.
- We let the CKA_NEVER_EXTRACTABLE attribute of $\boldsymbol{k}$ be *true* whenever $\boldsymbol{k}.\mathit{unextractable}$ and $\boldsymbol{k}.\mathit{readers} = \emptyset$.

We will discuss these choices shortly. Note that a literal implementation, reflecting the past values of CKA_SENSITIVE and CKA_EXTRACTABLE, could alternatively be modeled with the help of the log.

The CKA_TRUSTED attribute is not provided by our emulation, as it relates to wrapping under public keys and our model does not contain that. The attribute CKA_LOCAL in PKCS #11 is always *true* because no key can be generated outside of our token. The key life-cycle attributes CKA_START_DATE and CKA_END_DATE are not relevant for our token security policy and therefore omitted.

Functions. Only the functions of PKCS #11 dealing with object management, key management, and cryptographic operations are relevant here and provided by the emulation; all others are omitted.

For object management, PKCS #11 provides functions C_CreateObject, C_DestroyObject, C_GetAttributeValue and C_SetAttributeValue. The C_CreateObject function creates a key (and other objects) and sets its value to one supplied in the call; it is not provided by our emulation. The other three functions are implemented in our model by the *delete*, *getattr* and *setattr* operations, respectively. Reading key attributes containing cryptographic values with C_GetAttributeValue is mapped to the *read* operation. The functions for reading and modifying attribute values are restricted according to the conditions in our model.

The C_CreateObject function is missing because our model does not allow to create keys with user-specified values. Instead, all keys are generated randomly or derived from another key on the token; they cannot be biased by a user. This restriction is critical for maintaining our security policy because user-supplied key values might be abused; in particular, they might compromise the security of other keys in the token since the token cannot accurately determine the users who have read the value of a particular key. PKCS #11 also implements a C_CopyObject function to duplicate a key object and to modify its attributes; our emulation does not provide it for the same reason.

We now turn to the cryptographic functions in PKCS #11. All keys in our emulation must be generated through one of the functions C_GenerateKey, C_GenerateKeyPair, and C_DeriveKey; they are provided by the token operations $\mathit{create}(\boldsymbol{k}, \mathit{secret})$, $\mathit{create}(\boldsymbol{k}, \mathit{private}, \boldsymbol{l})$, and $\mathit{derive}(\boldsymbol{k}, \boldsymbol{l})$, respectively. The functions C_Encrypt, C_Decrypt, C_Sign, C_Verify, C_WrapKey, and C_UnwrapKey are implemented in the token by operations *encrypt*, *decrypt*, *sign*, *verify*, *wrap*, and *unwrap*, respectively, and follow our token security model.

Security. We now discuss some elements of the PKCS #11 security policy and how our emulation achieves them.

A *sensitive* key must not be read by any user. This is enforced directly by removing the *read* and *admin* privileges for *user* in the ACL. Our implementation of the CKA_ALWAYS_SENSITIVE attribute matches the intention behind marking a key “sensitive,” because the key has never been read by a user.

An *unextractable* key must not be wrapped or read by any user, and cannot be made extractable ever again. Our model achieves this because when an owner of a key $\boldsymbol{k}$ sets $\boldsymbol{k}$.*unextractable* to *true*, there must not be any *admin* or *read* privilege in the ACL. This ensures that the key can never be read in future and also $\boldsymbol{k}$.*acl* can no longer be modified. Furthermore, the *wrap* operation fails if $\boldsymbol{k}.\mathit{unextractable} = \mathit{true}$. Our implementation of the CKA_NEVER_EXTRACTABLE attribute is a slight departure from the idea of non-extractable keys because such a key in our emulation may have been wrapped previously under a key that has not been read by any user. But assuming that the token uses a secure wrapping scheme, the original intention behind marking a key "not extractable" is implemented correctly because our token ensures that the wrapping key can no longer become known to a user.

When deriving a key $\boldsymbol{l}$ from a key $\boldsymbol{k}$ in PKCS #11, if $\boldsymbol{k}$ has CKA_NEVER_EXTRACTABLE set to *false*, then so does $\boldsymbol{l}$; if $\boldsymbol{k}$ has CKA_NEVER_EXTRACTABLE set to *true*, then the value of CKA_NEVER_EXTRACTABLE for $\boldsymbol{l}$ is the opposite value of CKA_EXTRACTABLE for $\boldsymbol{k}$.

When deriving $\boldsymbol{l}$ from $\boldsymbol{k}$ in our token, the token sets the attributes of $\boldsymbol{l}$ to those of $\boldsymbol{k}$. Because $\boldsymbol{l}$ depends on $\boldsymbol{k}$ and therefore inherits its *readers* attribute, the value of the emulated CKA_NEVER_EXTRACTABLE attribute of $\boldsymbol{k}$ is propagated to the CKA_NEVER_EXTRACTABLE attribute of $\boldsymbol{l}$. This corresponds to the functionality of PKCS #11, as a simple case analysis shows.

PKCS #11 specifies that when a wrapping is unwrapped to obtain a key $\boldsymbol{k}$, then CKA_EXTRACTABLE is set to *true* and CKA_NEVER_EXTRACTABLE is set to *false*. But in our emulation we obtain again the attributes of the wrapped key from the time when it was wrapped. Since $\boldsymbol{k}$ had been wrapped in the first place, however, it must be that $\boldsymbol{k}$.*unextractable* is *false*. Hence, the emulated CKA_NEVER_EXTRACTABLE attribute of $\boldsymbol{k}$ after unwrapping is also *false*, consistent with its value according to PKCS #11.

7 Conclusion

This paper has introduced the first model of a cryptographic token interface supporting multiple users (i.e., more than two), support for encryption, data authentication, access control lists, and a security policy expressed in terms of a cryptographic notion. We hope that our model lays the foundation for the design of future cryptographic interfaces in the industry, which will not allow interface attacks *by design*.

For simplicity, in our work we did not address key wrapping with public keys; this is an important feature of PKCS #11, for example, available in cryptographic token interfaces used in many PKI applications. We sketch how this may be done. One needs a public-key cryptosystem secure against adaptive chosen-ciphertext attacks. One also needs a strong existentially unforgeable digital signature scheme. Combining the two in the standard way such that the resulting scheme provides confidentiality and integrity yields a suitable public-key wrapping scheme.

An important extension of this work lies in formally verifying our approach with automated tools, such as model checkers and theorem provers.

In practice, a security infrastructure usually contains many distributed cryptographic tokens, and a central administrator synchronizes all relevant data among them. Future work should therefore address multiple tokens that share a common set of keys. In our model with only one token, attribute changes are instantaneous and the abstraction of the log can be implemented efficiently. Distributed tokens are not automatically synchronized; the way to a secure distributed token infrastructure lies in considering communication between tokens and refining our one-token security policy.

Acknowledgments

We are grateful to Victor Shoup, Tamás Visegrády, and Sebastian Mödersheim for valuable feedback and interesting discussions.

References

[1] B. Adida, M. Bond, J. Clulow, A. Lin, R. Anderson, and R. L. Rivest, "On the security of the EMV secure messaging API." Manuscript, Apr. 2007.

[2] J. H. An, Y. Dodis, and T. Rabin, "On the security of joint signature and encryption," in *EUROCRYPT*, pp. 83–107, 2002.

[3] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov, "Cryptographic processors — a survey," *Proceedings of the IEEE*, vol. 94, pp. 357–369, Feb. 2006.

[4] R. J. Anderson, "Why cryptosystems fail," in *Proc. 1st ACM Conference on Computer and Communications Security*, pp. 215–227, 1993.

[5] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management — Part 1: General (revised)," NIST special publication 800-57, National Institute of Standards and Technology (NIST), Mar. 2007. Available from `http://csrc.nist.gov/publications/PubsSPs.html`.

[6] M. Bellare, A. Boldyreva, and S. Micali, "Public-key encryption in a multi-user setting: Security proofs and improvements," in *Advances in Cryptology: EUROCRYPT 2000* (B. Preneel, ed.), vol. 1087 of *Lecture Notes in Computer Science*, pp. 259–274, Springer, 2000.

[7] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *Advances in Cryptology: ASIACRYPT 2000* (T. Okamoto, ed.), vol. 1976 of *Lecture Notes in Computer Science*, pp. 531–545, Springer, 2000.

[8] BITS Security Working Group, "Enterprise key management." Whitepaper, BITS Financial Services Roundtable, available from `http://www.bits.org/downloads/Publications%20Page/BITSEnterpriseKeyManagementMay2008.pdf`, May 2008.

[9] J. Black, P. Rogaway, and T. Shrimpton, "Encryption-scheme security in the presence of key-dependent messages," in *Proc. Workshop on Selected Areas of Cryptography (SAC 2002)* (K. Nyberg and H. M. Heys, eds.), vol. 2595 of *Lecture Notes in Computer Science*, pp. 62–75, 2002.

[10] M. Bond, "Attacks on cryptoprocessor transaction sets," in *Proc. Cryptographic Hardware and Embedded Systems (CHES 2001)*, vol. 2162 of *Lecture Notes in Computer Science*, pp. 220–234, 2001.

[11] J. Camenisch and A. Lysyanskaya, "An efficient system for non-transferable anonymous credentials with optional anonymity revocation," in *Advances in Cryptology: EUROCRYPT 2001* (B. Pfitzmann, ed.), vol. 2045 of *Lecture Notes in Computer Science*, Springer, 2001.

[12] J. Clulow, "On the security of PKCS#11," in *Proc. Cryptographic Hardware and Embedded Systems (CHES 2003)*, vol. 2779 of *Lecture Notes in Computer Science*, pp. 411–425, 2003.

[13] V. Cortier, S. Delaune, , and G. Steel, “A formal theory of key conjuring,” in *Proc. 20th IEEE Computer Security Foundations Symposium (CSF 2007)*, 2007.

[14] “Cryptographic key management.” Topic Document, Cover Pages, available from `http://xml.coverpages.org/keyManagement.html`, Feb. 2009.

[15] S. Delaune, S. Kremer, and G. Steel, “Formal analysis of PKCS#11,” in *Proc. 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, 2008.

[16] D. Dolev and A. C. Yao, “On the security of public key protocols,” *IEEE Transactions on Information Theory*, vol. 29, pp. 198–208, Mar. 1983.

[17] S. Goldwasser, S. Micali, and R. L. Rivest, “A digital signature scheme secure against adaptive chosen-message attacks,” *SIAM Journal on Computing*, vol. 17, pp. 281–308, Apr. 1988.

[18] International Business Machines Corp., *CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors*, 19th ed., Apr. 2008. Available from `http://www-03.ibm.com/security/cryptocards/pcicc/library.shtml`.

[19] D. B. Johnson, G. M. Dolan, M. J. Kelly, A. V. Le, and S. M. Matyas, “Common cryptographic architecture cryptographic application programming interface,” *IBM Systems Journal*, vol. 30, no. 2, pp. 130–150, 1991.

[20] J. Katz and M. Yung, “Complete characterization of security notions for probabilistic private-key encryption,” in *Proc. 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pp. 245–254, 2000.

[21] “KMIP — Key Management Interoperability Protocol.” OASIS Key Management Interoperability Protocol (KMIP) Technical Committee, available from `http://xml.coverpages.org/KMIP/`, Feb. 2009.

[22] M. Naor and M. Yung, “Public-key cryptosystems provably secure against chosen ciphertext attacks,” in *STOC*, pp. 427–437, 1990.

[23] C. Rackoff and D. R. Simon, “Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack,” in *CRYPTO*, pp. 433–444, 1991.

[24] P. Rogaway and T. Shrimpton, “A provable-security treatment of the key-wrap problem,” in *Advances in Cryptology: Eurocrypt 2006* (S. Vaudenay, ed.), vol. 4004 of *Lecture Notes in Computer Science*, pp. 373–390, 2006.

[25] RSA Laboratories, “PKCS #11 v2.20: Cryptographic Token Interface Standard.” Available from `http://www.rsa.com/rsalabs/`, 2004.

[26] Trusted Computing Group, “Trusted platform module specifications.” Available from `http://www.trustedcomputinggroup.org`, 2008.

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