A Provably Secure PKCS#11 Configuration Without Authenticated Attributes

Ryan Stanley-Oakes(✉)

University of Bristol, Bristol, UK
ryan.stanley@bristol.ac.uk

Abstract. Cryptographic APIs like PKCS#11 are interfaces to trusted hardware where keys are stored; the secret keys should never leave the trusted hardware in plaintext. In PKCS#11 it is possible to give keys conflicting roles, leading to a number of key-recovery attacks. To prevent these attacks, one can authenticate the attributes of keys when wrapping, but this is not standard in PKCS#11. Alternatively, one can configure PKCS#11 to place additional restrictions on the commands permitted by the API.

Bortolozzo *et al.* proposed a configuration of PKCS#11, called the Secure Templates Patch (STP), supporting symmetric encryption and key wrapping. However, the security guarantees for STP given by Bortolozzo *et al.* are with respect to a weak attacker model. STP has been implemented as a set of filtering rules in *Caml Crush*, a software filter for PKCS#11 that rejects certain API calls. The filtering rules in *Caml Crush* extend STP by allowing users to compute and verify MACs and so the previous analysis of STP does not apply to this configuration.

We give a rigorous analysis of STP, including the extension used in *Caml Crush*. Our contribution is as follows:

(i) We show that the extension of STP used in *Caml Crush* is insecure.
(ii) We propose a strong, computational security model for configurations of PKCS#11 where the adversary can adaptively corrupt keys and prove that STP is secure in this model.
(iii) We prove the security of an extension of STP that adds support for public-key encryption and digital signatures.

1 Introduction

In high-risk environments, particularly where financial transactions take place, secret and private keys are often stored inside trusted, tamper-proof hardware such as HSMs and cryptographic tokens. Then ordinary host machines, which could be compromised by malware or malicious users, can issue commands to the trusted hardware via an interface called a cryptographic API. The operations that can be carried out using the API often include key wrapping, which is the encryption of one key under another to enable the secure exchange and storage of

R. Stanley-Oakes—The author is supported by an EPSRC Industrial CASE studentship.

A. Kiayias (Ed.): FC 2017, LNCS 10322, pp. 145–162, 2017.
https://doi.org/10.1007/978-3-319-70972-7_8

keys. The API can also be used to add new keys to the trusted hardware, either by issuing a key generation command or unwrapping a wrapped key. The API refers to each key by a handle, which has attributes used to specify the intended use of the key. By wrapping and unwrapping, it is possible for different handles, each with different attributes, to point to the same key. This could cause a key to have conflicting roles within the API.

The study of cryptographic APIs was initiated by Bond and Anderson in 2001, when they described attacks against ATMs and prepayment utility meters, exploiting weaknesses in the *interfaces* to the trusted hardware, rather than in the cryptographic algorithms performed by the hardware: "The basic idea is that by presenting valid commands to the security processor, but in an unexpected sequence, it is possible to obtain results that break the security policy envisioned by its designer" [3].

While Bond and Anderson identified vulnerabilities in particular devices with bespoke APIs, Clulow then used their approach to find devastating key recovery attacks against a widely-used, generic API [6]. This API, called PKCS#11[1] is independent of the hardware with which it communicates and was designed to enable interoperability between the trusted hardware from different manufacturers [10].

In 2008, Delaune *et al.* presented a formal, Dolev–Yao style model of PKCS#11 and used model-checking tools to find new attacks [7,8]. Bortolozzo *et al.* then developed an automated tool called *Tookan*, built on the model by Delaune *et al.*, that found and executed attacks against real hardware devices using PKCS#11 [4]. As a result of these attacks, an important research question has been to find a *configuration* of PKCS#11, i.e. a set of restrictions on the commands that can be issued to the API, such that the API is secure with these restrictions.

Bortolozzo *et al.* suggested a configuration of PKCS#11, supporting just symmetric encryption and symmetric key wrapping, called the Secure Templates Patch (STP) [4]. In STP, newly-generated keys are separated into encryption/decryption keys and wrapping/unwrapping keys, while keys imported by unwrapping can be used for encryption and unwrapping, but not decryption or wrapping. STP has been implemented as a set of filtering rules in *Caml Crush*, a software filter that rejects certain PKCS#11 calls [1]. However, the filtering rules in *Caml Crush* allow users to compute and verify MACs, which is not captured by the model from Delaune *et al.* [7,8]. Therefore the previous analysis of STP does not apply to what is implemented in *Caml Crush*. Furthermore, while STP is resistant to attack by *Tookan*, there has not yet been a formal proof of security for this configuration, which is the problem we address here.

[1] PKCS#11 is actually the name of the cryptographic standards document that describes the API, which is called *Cryptoki*. However, it is conventional to refer to the API itself as PKCS#11.

1.1 Our Contribution

As a first result, we show that the filtering rules in *Caml Crush* are not sufficient to secure PKCS#11. The attacker is assumed to have knowledge of how the filter operates, but can only interact with the API via the filter. Two sets of filtering rules are offered; the first set is trivially broken if the attacker can read the source code of the filter. The second set of rules is designed to emulate STP, but offers MAC functionality that was not modelled by Delaune *et al.* and hence is not exploited by *Tookan*. We show that the filtering does not enforce a separation between encryption and MAC keys. We also show that there exist encryption and MAC schemes that are individually secure, but completely insecure when the same keys are used for both primitives. Therefore STP, as implemented in *Caml Crush*, is only safe to use if one is certain that the encryption and MAC schemes are *jointly* secure.

Our second contribution is a computational security model for configurations of PKCS#11, where certain API calls are rejected according to the *policy* in the configuration. The policy may determine, for example, what attributes newly-generated or newly-imported keys can have. Our model captures the use of both symmetric and asymmetric variants of encryption and signing primitives within the API. We say that an API is secure if, for any cryptographic primitives used by the API, encrypting and signing data using the API is as secure as using the primitives themselves in isolation. This is strictly stronger than the model from Delaune *et al.*, where an API is considered secure if the attacker cannot learn the values of honestly-generated secret keys [7,8]. Moreover, the adversary in our model is allowed to adaptively corrupt certain keys.

Our main result is a PKCS#11 configuration that is provably secure in our model. We first show that STP as proposed by Bortolozzo *et al.* is *not* secure; STP allows the same keys to be used for encryption and unwrapping, so an attacker can *encrypt* (rather than wrap) their own key, import this key by unwrapping and use this key to encrypt or sign data. Since keys used by the API could have been generated by the adversary, there can be no guarantees for data protected by the API, even if the cryptographic primitives are secure. However, we prove that if the policy prevents the encryption (rather than wrapping) of keys, then the configuration is secure. Moreover, our main result holds for an extension of STP that supports public-key encryption and digital signatures.

The proof of our main result is highly non-trivial since we allow the adversary to adaptively corrupt keys. Adaptive corruption captures the realistic threat scenario that certain keys are leaked through side-channel attacks, which, due to the key wrapping operation, can have devastating consequences for the API. Nevertheless, most existing analyses of cryptographic APIs avoid this strong attacker model because traditional proof techniques cannot be used; for a standard cryptographic reduction, one has to know in advance which keys will be corrupted to correctly simulate the environment of the adversary. Instead, our security proof uses techniques from Panjwani's proof that the IND-CPA security of encryption implies its Generalised Selective Decryption (GSD) security [11]. This is a com-

plex hybrid argument where one first guesses a *path*, in the wrapping graph that will be adaptively created by the adversary, from a source node (corresponding to a key that does not appear in a wrap) to a *challenge* node (corresponding to a key used for encryption of data, or signing, etc.). Then the way in which one responds to wrap queries depends on the positions of the corresponding nodes relative to the guessed path. To our knowledge, we are the first to adapt Panjwani's result to the API setting. A detailed discussion of related work is given in the full version of the paper [14].

2 Preliminaries

We use the term *token* to refer to any trusted hardware carrying out cryptographic operations. All keys are stored inside the token and the user has an API used to issue commands to the token.

We assume the API used by the token is compliant with at least v2.20 of the PKCS#11 standard.[2] While the PKCS#11 specification distinguishes between normal users and security officers, we conflate these roles and assume the adversary can perform any operations permitted by the API. Security in this sense automatically implies security against adversaries who can only interact with the API as normal users or security officers.

We assume that tokens store no keys in their initial state. Then keys can be added to the device using one of the following commands: `C_GenerateKey` or `C_GenerateKeyPair`, which cause the token to generate a new key or key pair using its own internal randomness; `C_UnwrapKey`, which causes the token to decrypt the supplied ciphertext and store the plaintext as a new key (without revealing it); `C_CreateObject`, which we used to model importing public keys from other tokens; or `C_TransferKey`, which we use to model an out-of-band method for securely transferring long-term secret keys between tokens (this could happen during the manufacturing process, for example).

The API refers to keys using *handles*; these are public identifiers. So, for example, if the user issues the command $\texttt{C\_Encrypt}(h, m)$, they expect to receive the encryption of the message m under the key pointed to by the handle h. The *class* of a key is whether it is public, private or secret. For each handle, the token stores the corresponding key, the class of this key and its *template*, which is a set of *attributes* that determine how the key can be used. Attributes are either *set* or *unset*. For example, PKCS#11 mandates that the command $\texttt{C\_Encrypt}(h, m)$ must fail if the attribute `CKA_ENCRYPT` is not set in the template associated to h.

In the language of PKCS#11, the value of a key is also an attribute of its handle, and the API has to prevent the reading of this attribute if the attribute `CKA_SENSITIVE` is set, i.e. the API should not reveal the values of keys that are supposed to be secret. For simplicity we say that templates do not contain the value of keys. This way all attributes are binary and can be disclosed to the user.

[2] Version 2.20 of the standard was published in 2004, and was the first to introduce the attributes `CKA_TRUSTED` and `CKA_WRAP_WITH_TRUSTED`, which we use to prevent key cycles.

Accordingly we have no need for the attribute CKA_SENSITIVE; all public keys will be returned to the user at generation time and other keys can only be revealed by corruption.

PKCS#11 allows an incomplete template to be supplied when a new handle is created, forcing the API to choose whether to set or unset the unspecified attributes; we simply assume that the operation fails if the template is incomplete. For convenience, we also assume that the template of a handle contains the class of the corresponding key.

In PKCS#11, some attributes can be changed by the user (or by the API). For example, perhaps the attribute CKA_ENCRYPT is not initially set in the template of some handle h pointing to the key k, but later the user wishes to use k to encrypt data. We exclude this from our model, preferring to assume that the intended use of all keys is known at generation time. In the language of PKCS#11, all our attributes are *sticky*.

There are nine attributes relevant to our analysis, as follows: CKA_EXTRACTABLE, which we abbreviate by CKA_EXTR, is used to identify those keys that can be wrapped (in the case of private or secret keys), or given out (in the case of public keys). CKA_WRAP_WITH_TRUSTED, which we abbreviate by CKA_WWT, is used to identify those keys that can only be wrapped by keys with CKA_TRUSTED set. CKA_TRUSTED is used to identify those keys that are considered trusted wrapping keys. CKA_WRAP, CKA_UNWRAP, CKA_ENCRYPT, CKA_DECRYPT, CKA_SIGN and CKA_VERIFY are used to identify those keys that can wrap keys, unwrap keys, encrypt data, decrypt data, sign (or MAC) data and verify signatures (or MAC tags), respectively.

PKCS#11 specifies some rules, which we call the *policy*, about how attributes must be used (like how the template of h must have CKA_ENCRYPT set in order for C_Encrypt(h, m) to succeed). But the standard also allows manufacturers, in their own *configurations* of PKCS#11, to impose additional restrictions on how the API operates. For example, the PKCS#11 policy allows a symmetric key to be generated with both CKA_WRAP and CKA_DECRYPT set, leading to the famous wrap/decrypt attack [6]. Manufacturers should therefore disable this command in their configuration. We assume that the policy in the manufacturer's configuration allows a subset of commands allowed by the PKCS#11 policy (so that the configuration is actually compliant with the specification) and therefore we use a single policy algorithm to capture both the standard PKCS#11 policy and any additional restrictions, i.e. any command not rejected by our policy algorithm is automatically allowed within PKCS#11.

3 Vulnerabilities in Caml Crush

In *Caml Crush*, the idea is that the interface to some trusted hardware is a PKCS#11-compliant, but insecure, API [1]. The software is then used to filter out API calls that could lead to attacks. This is rather like having a more restrictive policy *within* the API and so the authors adapt the PKCS#11 configurations suggested by Bortolozzo *et al.* to filtering rules. Bortolozzo *et al.* suggested two

configurations of PKCS#11 that are resistant to attack by *Tookan* [4], both of which are implemented in *Caml Crush* as sets of filtering rules [1]:

1. In the *Wrapping Formats Patch (WFP)*, the attributes of a key are transmitted as part of a wrap of the key and authenticated using a MAC.
2. In the *Secure Templates Patch (STP)*, wrapping and encryption keys are separated at generation time and imported symmetric keys can be used for unwrapping and encryption, but not wrapping or decryption.

We remark that the first patch is actually a violation of the PKCS#11 standard: the standard mandates that a wrap of a key is solely the encryption of the value of the key, i.e. the attributes of the key are not included in the output and no MAC tag is added. Tokens whose APIs use WFP are not interoperable with tokens using PKCS#11-compliant APIs.

Moreover, the way WFP is implemented in Caml Crush is trivially insecure. Examining the source code, the MAC used to authenticate the attributes of the wrapped key is computed using a key that is stored in plaintext in the configuration file of the filter [2]. This is a clear violation of Kerckhoffs' principle: the attacker who knows how the filter is constructed (i.e. can read the source code of the filter) can immediately circumvent the additional protection provided by the MAC and use the wrap/decrypt attack to learn the value of any extractable secret key. The authors of *Caml Crush* acknowledge this vulnerability in a comment: "We use the key configured in the filter configuration file ... You might preferably want to use a key secured in a token". We feel this is an understatement of the insecurity of their solution.

We focus our attention on STP, as this is compliant with the PKCS#11 specification. Note that STP, as presented by Bortolozzo *et al.*, only enables the symmetric encryption, decryption, wrapping and unwrapping functions of the API and not, for example, the MAC and verify functions [4]. The implementation in *Caml Crush* adds MAC functionality to STP, but does so in a potentially insecure way. Their filtering rules allow freshly generated symmetric keys to be used for wrapping and unwrapping, encryption and decryption, or signing and verifying (using a MAC scheme). Then keys imported via the unwrap command can either unwrap and encrypt, or unwrap, sign and verify. At first glance, these restrictions appear to maintain a separation between encryption and MAC keys, but this is not the case. One can generate an encryption key, wrap it, and unwrap it as a MAC key. This configuration is only secure if the encryption and MAC schemes are *jointly* secure, i.e. it is safe to use the same key for both primitives. In the full version of the paper, we show that this assumption does not always hold [14].

4 Security Model and Assumptions

PKCS#11 supports both symmetric and asymmetric primitives for encrypting and signing data and for wrapping keys. For simplicity we will assume that all keys and key pairs are generated using the same two algorithms KG

and KPG. Moreover, we assume that the key wrap mechanisms use the same encryption schemes as for encrypting data. Therefore our model of a configuration of PKCS#11 is parameterised by four cryptographic primitives: a probabilistic symmetric encryption scheme $\mathcal{E} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$, a probabilistic public-key encryption scheme $\mathcal{PKE} = (\mathsf{KPG}, \mathsf{AEnc}, \mathsf{ADec})$, a MAC scheme $\mathcal{M} = (\mathsf{KG}, \mathsf{Mac}, \mathsf{MVrfy})$ and a digital signature scheme $\mathcal{S} = (\mathsf{KPG}, \mathsf{Sign}, \mathsf{SVrfy})$. The syntax of these primitives and the formal definitions of correctness and security are all given in the full version of the paper [14].

The API also has an algorithm NewHandle for generating fresh handles. This will be called when keys are imported via unwrapping or the `C_CreateObject` command or new keys are generated. This algorithm is assumed to be stateful so that it never returns the same value. For each handle h returned by NewHandle, the API stores a template $h.\mathsf{temp}$ and a pointer p to the token memory where the value of the key is stored. By abuse of notation, the contents of the token memory at p will be written $h.\mathsf{key}$ (even though this value is not directly accessible to the API). The class of the key, i.e. secret, public or private, is stored in $h.\mathsf{class}$.

The configuration of the API is defined by the *policy*. We model the policy by the algorithm P that takes the name of the API command and the inputs to that command as inputs, then returns 1 if this combination is permitted and 0 otherwise.

Before giving the formal security definition, we introduce a restriction which is necessary for security and considerably simplifies the model:

Remark 1. Asymmetric key wrapping must be disabled.

Even before a formal security definition is given, it should be clear that any mechanism for key wrapping must provide *integrity* as well as secrecy. If it were not the case, then an adversary could generate their own keys, forge wraps of these keys, unwrap them and use them to wrap honestly-generated keys or encrypt and sign data. If this attack is possible, there can be no guarantees for data and keys protected by the API, since any keys used by the API could be adversarially generated. Of course, the notion of integrity of ciphertexts makes no sense in the public key encryption setting without the sender needing a private key as well as a public key to encrypt. Therefore we make the standard assumptions from the literature that all key wrapping is symmetric and, for bootstrapping, there is an out-of-band method for securely exchanging long-term secret keys [4,9,12,13].

4.1 Security Definition

Following [9,12,13], we give a *computational*, rather than symbolic, security definition for a configuration of PKCS#11, where the adversary has access to a number of oracles and plays a game. Winning the game means violating the security of one of the cryptographic primitives used by the token. We say, informally, that a configuration of PKCS#11 is secure if using the API to encrypt

and sign data is as secure as encrypting and signing with the separate, individual primitives. This notion of security is similar to the one used by Cachin and Chandran [5].

Formally, for each adversary $\mathcal{A}$ and each $b \in \{0,1\}$, we define an experiment $\mathsf{API}^b(\mathcal{A}) := \mathsf{API}^b_{\mathcal{E},\mathcal{M},\mathcal{PKE},\mathcal{S},\mathtt{P}}(\mathcal{A})$ where the adversary has access to a number of oracles capturing the commands one can issue to the API, and some *challenge* oracles whose responses depend on b. The oracles all first check, using the policy $\mathtt{P}$, that the command from the adversary is allowed. If this succeeds, then the oracles perform the cryptographic operations that would be carried out by the token. Note that our formal model conflates the roles of the API and the token, which simplifies notation considerably, but is without loss of generality since we know how PKCS#11-compliant APIs interact with tokens. The only thing we do not know is how the token implements the cryptographic operations, and these details are abstracted away in our model.

After interacting with the API oracles, the adversary returns a guess b'. Provided that certain conditions are met whereby the adversary cannot trivially learn b, the experiment returns b'. Otherwise, the experiment returns 0. The *advantage* of $\mathcal{A}$ against the API is defined to be the following quantity:

$$\mathsf{Adv}^{\mathsf{API}}(\mathcal{A}) := \left|\mathbb{P}[\mathsf{API}^1(\mathcal{A}) = 1] - \mathbb{P}[\mathsf{API}^0(\mathcal{A}) = 1]\right|.$$

The experiment API^b is shown in Fig. 1, with the oracles available to $\mathcal{A}$ shown in Figs. 2 and 3.

Experiment $\mathsf{API}^b_{\mathcal{E},\mathcal{M},\mathcal{PKE},\mathcal{S},\mathtt{P}}(\mathcal{A})$:
$i \leftarrow 0$
$\mathsf{Chal} \leftarrow \emptyset, \mathsf{Cor} = \{0\}$
$W \leftarrow \emptyset, E \leftarrow \emptyset, V \leftarrow \{0\}$
$P \leftarrow \emptyset, K \leftarrow \emptyset$
for all $j \in [n]$,
  $C_1[j], C_1^*[j], C_2[j], C_2^*[j], T[j], T^*[j], S[j], S^*[j] \leftarrow \emptyset$
$b' \leftarrow \mathcal{A}^{\mathcal{O}}$
if $\mathsf{Chal} \cap \mathsf{Comp} \neq \emptyset$ then return 0
if $\exists j \in [n]$ such that:
  $C_1[j] \cap C_1^*[j] \neq \emptyset$
  or $C_2[j] \cap C_2^*[j] \neq \emptyset$
  or $T[j] \cap T^*[j] \neq \emptyset$
  or $S[j] \cap S^*[j] \neq \emptyset$:
then return 0
else return b'

Fig. 1. The Security Experiment $\mathsf{API}^b(\mathcal{A})$ for a cryptographic API supporting symmetric and asymmetric encryption, a MAC scheme and a signature scheme. The oracles $\mathcal{O}$ are defined in Figs. 2 and 3.

Oracle $\mathcal{O}^{\texttt{C\_CreateObject}}(pk, t)$:
if $\texttt{P}(\texttt{C\_CreateObject}, pk, t)$:
  $h \leftarrow \mathsf{NewHandle}$
  $h.\mathsf{key} \leftarrow pk$
  $h.\mathsf{temp} \leftarrow t$
  $h.\mathsf{class} \leftarrow \mathsf{public}$
  $X \leftarrow \{h' \in P : h'.\mathsf{key} = pk\}$
  if $X \neq \emptyset$:
    $\mathsf{idx}(h) \leftarrow \min_{h' \in X} \mathsf{idx}(h')$
  else $\mathsf{idx}(h) \leftarrow 0$
  return h

Oracle $\mathcal{O}^{\texttt{C\_TransferKey}}(k, t)$:
if $\texttt{P}(\texttt{C\_TransferKey}, k, t)$:
  $h \leftarrow \mathsf{NewHandle}$
  $h.\mathsf{key} \leftarrow k$
  $h.\mathsf{temp} \leftarrow t$
  $h.\mathsf{class} \leftarrow \mathsf{secret}$
  $X \leftarrow \left\{ \begin{array}{l} h' \in K : \\ \quad h'.\mathsf{key} = k \wedge h'.\mathsf{temp} = t \end{array} \right\}$
  if $X \neq \emptyset$:
    $\mathsf{idx}(h) \leftarrow \min_{h' \in X} \mathsf{idx}(h')$
  else $\mathsf{idx}(h) \leftarrow 0$
  return h

Oracle $\mathcal{O}^{\texttt{C\_GenerateKey}}(t)$:
if $\texttt{P}(\texttt{C\_GenerateKey}, t)$:
  $i{+}{+}$
  $h \leftarrow \mathsf{NewHandle}$
  $K = K \cup \{h\}$
  $\mathsf{idx}(h) \leftarrow i$
  $V \leftarrow V \cup \{i\}$
  $h.\mathsf{key} \leftarrow \mathsf{KG}$
  $h.\mathsf{temp} \leftarrow t$
  $h.\mathsf{class} \leftarrow \mathsf{secret}$
  return h

Oracle $\mathcal{O}^{\texttt{C\_GenerateKeyPair}}(t, t')$:
if $\texttt{P}(\texttt{C\_GenerateKeyPair}, t, t')$:
  $i{+}{+}$
  $h \leftarrow \mathsf{NewHandle}$
  $h' \leftarrow \mathsf{NewHandle}$
  $P = P \cup \{h\}$
  $\mathsf{idx}(h) \leftarrow i$
  $\mathsf{idx}(h') \leftarrow i$
  $V \leftarrow V \cup \{i\}$
  $(h.\mathsf{key}, h'.\mathsf{key}) \leftarrow \mathsf{KPG}$
  $h.\mathsf{temp} \leftarrow t$
  $h'.\mathsf{temp} \leftarrow t'$
  $h.\mathsf{class} \leftarrow \mathsf{public}$
  $h'.\mathsf{class} \leftarrow \mathsf{private}$
  return $h, h', h.\mathsf{key}$

Oracle $\mathcal{O}^{\texttt{C\_WrapKey}}(h, h')$:
if $\texttt{P}(\texttt{C\_WrapKey}, h, h')$:
  if $h.\mathsf{class} = \mathsf{secret}$:
    if $h'.\mathsf{class} = \mathsf{private}$
    or $h'.\mathsf{class} = \mathsf{secret}$:
      $w \leftarrow \mathsf{Enc}(h.\mathsf{key}, h'.\mathsf{key})$
      $W \leftarrow W \cup \{(h, h', w)\}$
      $E \leftarrow E \cup \{(\mathsf{idx}(h), \mathsf{idx}(h'))\}$
      return w

Oracle $\mathcal{O}^{\texttt{C\_UnwrapKey}}(h, w, t)$:
if $\texttt{P}(\texttt{C\_UnwrapKey}, h, w, t)$:
  if $h.\mathsf{class} = \mathsf{secret}$:
    $k' \leftarrow \mathsf{Dec}(h.\mathsf{key}, w)$
    if $k' \in \mathsf{SecretKeys}$
    or $k' \in \mathsf{PrivateKeys}$:
      $h' \leftarrow \mathsf{NewHandle}$
      $h'.\mathsf{temp} \leftarrow t$
      $h'.\mathsf{key} \leftarrow k'$
      unwrapbookkeeping
      return h'

Macro unwrapbookkeeping:
$X \leftarrow \left\{ \begin{array}{l} h_2 : (h_1, h_2, w) \in W \\ \quad \wedge\, \mathsf{idx}(h_1) = \mathsf{idx}(h) \end{array} \right\}$
if $X \neq \emptyset$:
  $\mathsf{idx}(h') \leftarrow \min_{h_2 \in X} \mathsf{idx}(h_2)$
else if $\mathsf{idx}(h) \in \mathsf{Comp}$:
  $\mathsf{idx}(h') \leftarrow 0$
else:
  $i{+}{+}$
  $\mathsf{idx}(h') \leftarrow i$
  $V \leftarrow V \cup \{i\}$

Oracle $\mathcal{O}^{\texttt{Corrupt}}(h)$:
if $h.\mathsf{class} = \mathsf{private}$
or $h.\mathsf{class} = \mathsf{secret}$:
  $\mathsf{Cor} \leftarrow \mathsf{Cor} \cup \{\mathsf{idx}(h)\}$
  return $h.\mathsf{key}$

Fig. 2. Oracles Representing PKCS#11 Key Management Commands and Key Corruption

Oracle $\mathcal{O}^{\texttt{C\_Encrypt}}(h, m)$:
if $\texttt{P}(\texttt{C\_Encrypt}, h, m)$:
  if $h.\mathsf{class} = \mathsf{secret}$:
    return $\mathsf{Enc}(h.\mathsf{key}, m)$
  if $h.\mathsf{class} = \mathsf{public}$:
    return $\mathsf{AEnc}(h.\mathsf{key}, m)$

Oracle $\mathcal{O}^{\texttt{C\_Decrypt}}(h, c)$:
if $\texttt{P}(\texttt{C\_Decrypt}, h, c)$:
  if $h.\mathsf{class} = \mathsf{secret}$:
    $c \leftarrow \mathsf{Dec}(h.\mathsf{key}, c)$
    $C_1[\mathsf{idx}(h)] \leftarrow C_1[\mathsf{idx}(h)] \cup \{c\}$
    return c
  if $h.\mathsf{class} = \mathsf{private}$:
    $c \leftarrow \mathsf{ADec}(h.\mathsf{key}, c)$
    $C_2[\mathsf{idx}(h)] \leftarrow C_2[\mathsf{idx}(h)] \cup \{c\}$
    return c

Oracle $\mathcal{O}^{\texttt{C\_Sign}}(h, m)$:
if $\texttt{P}(\texttt{C\_Sign}, h, m)$:
  if $h.\mathsf{class} = \mathsf{secret}$:
    $\tau \leftarrow \mathsf{Mac}(h.\mathsf{key}, m)$
    $T[\mathsf{idx}(h)] \leftarrow T[\mathsf{idx}(h)] \cup \{\tau\}$
    return τ
  if $h.\mathsf{class} = \mathsf{private}$:
    $\sigma \leftarrow \mathsf{Sign}(h.\mathsf{key}, m)$
    $S[\mathsf{idx}(h)] \leftarrow S[\mathsf{idx}(h)] \cup \{\sigma\}$
    return σ

Oracle $\mathcal{O}^{\texttt{C\_Verify}}(h, m, s)$:
if $\texttt{P}(\texttt{C\_Verify}, h, m, s)$:
  if $h.\mathsf{class} = \mathsf{secret}$:
    return $\mathsf{MVrfy}(h.\mathsf{key}, m, s)$
  if $h.\mathsf{class} = \mathsf{public}$:
    return $\mathsf{SVrfy}(h.\mathsf{key}, m, s)$

Oracle $\mathcal{O}_b^{\texttt{Enc-Challenge}}(h, m_0, m_1)$:
if $\texttt{P}(\texttt{C\_Encrypt}, h, m_0)$:
  if $\texttt{P}(\texttt{C\_Encrypt}, h, m_1)$:
    if $|m_0| = |m_1|$:
      if $h.\mathsf{class} = \mathsf{secret}$:
        $\mathsf{Chal} \leftarrow \mathsf{Chal} \cup \{\mathsf{idx}(h)\}$
        $c \leftarrow \mathsf{Enc}(h.\mathsf{key}, m_b)$
        $C_1^*[\mathsf{idx}(h)] \leftarrow C_1^*[\mathsf{idx}(h)] \cup \{c\}$
        return c
      if $h.\mathsf{class} = \mathsf{public}$:
        $\mathsf{Chal} \leftarrow \mathsf{Chal} \cup \{\mathsf{idx}(h)\}$
        $c \leftarrow \mathsf{AEnc}(h.\mathsf{key}, m_b)$
        $C_2^*[\mathsf{idx}(h)] \leftarrow C_2^*[\mathsf{idx}(h)] \cup \{c\}$
        return c

Oracle $\mathcal{O}_b^{\texttt{Sign-Challenge}}(h, m, s)$:
if $\texttt{P}(\texttt{C\_Verify}, h, m, s)$:
  if $h.\mathsf{class} = \mathsf{secret}$:
    $T^*[\mathsf{idx}(h)] \leftarrow T^*[\mathsf{idx}(h)] \cup \{s\}$
    $\mathsf{Chal} \leftarrow \mathsf{Chal} \cup \{\mathsf{idx}(h)\}$
    if $b = 0$ return $\mathsf{MVrfy}(h.\mathsf{key}, m, s)$
    else return 0
  if $h.\mathsf{class} = \mathsf{public}$:
    $S^*[\mathsf{idx}(h)] \leftarrow S^*[\mathsf{idx}(h)] \cup \{s\}$
    $\mathsf{Chal} \leftarrow \mathsf{Chal} \cup \{\mathsf{idx}(h)\}$
    if $b = 0$ return $\mathsf{SVrfy}(h.\mathsf{key}, m, s)$
    else return 0

Fig. 3. Oracles Representing PKCS#11 Cryptographic Operations and the IND-CCA and EUF-CMA Games

Now we explain some of the rationale behind the security game. We have two challenge oracles $\mathcal{O}_b^{\texttt{Enc-Challenge}}$ and $\mathcal{O}_b^{\texttt{Sign-Challenge}}$, corresponding to confidentiality (of public key and symmetric encryption) and authenticity (of signatures and MACs), respectively. These oracles closely resemble the IND-CCA and EUF-CMA games. For encryption, the bit b determines which of the messages m_0 and m_1 is encrypted under the challenge key. As usual, to avoid trivial wins we have to record the ciphertexts output by $\mathcal{O}_b^{\texttt{Enc-Challenge}}$ and the queries made

to the decryption oracle $\mathcal{O}^{\texttt{C\_Decrypt}}$, and check that the two sets corresponding to the same key are disjoint. For signing and MACs, the bit b determines whether the adversary sees the genuine result of the verification algorithm, or always sees the bit 0 (indicating that the verification has failed). To avoid trivial wins here, we record the signatures and tags output by $\mathcal{O}^{\texttt{C\_Sign}}$ and the candidate signatures and tags submitted to $\mathcal{O}_b^{\texttt{Sign-Challenge}}$ and check that the two sets corresponding to the same key are disjoint.

In our model, we include an oracle $\mathcal{O}^{\texttt{Corrupt}}$ that allows the adversary to adaptively corrupt certain keys. This captures the situation where some keys may be leaked, for example through side-channel attacks. Obviously, if such keys are used by the challenge oracles, then $\mathcal{A}$ can trivially recover the bit b. Moreover, if the adversary were to wrap a key under a corrupt key, then the wrapped key must be assumed *compromised*, since it can be trivially recovered by the adversary. Like corrupt keys, compromised keys are not safe for use by the challenge oracles. Therefore we keep track of a set Comp of corrupt and compromised keys and a set Chal of keys used by the challenge oracles, and the experiment only returns the guess b' from $\mathcal{A}$ if Comp and Chal are disjoint.

The situation is complicated by the fact that the adversary queries $\mathcal{O}^{\texttt{Corrupt}}$ with *handles*, not keys, and learns the value of the key pointed to by the handle. But by wrapping and unwrapping a key, the adversary obtains a new handle for the same key and clearly all handles pointing to the same key are compromised by the corruption of just one of them. Therefore we keep track of which handles point to the same key by giving them the same *index* $\mathsf{idx}(h)$ and store which *indices* are compromised, rather than which handles. This is based on the security model by Shrimpton *et al.* [13].

We assume that there is an authenticated channel for transmitting public keys using the `C_CreateObject` command. Therefore we check that any public keys imported via $\mathcal{O}^{\texttt{C\_CreateObject}}$ had at some point been honestly generated by a token. If so, the new handle is given the same index as the handle that was given out when the key was first generated. If not, the new handle is given index 0, which is used to represent automatically compromised keys (and therefore if this new public key is used in the challenge oracles, the guess output by $\mathcal{A}$ will be ignored). Note that we do not check that the template of the imported public key matches the template of the key when it was first generated. This is because we are not assuming that the attributes of keys are always authenticated. Therefore the policy of our configuration will have to restrict the roles of imported public keys.

Similarly, we assume there is a secure out-of-band method for transferring long-term wrapping keys, modelled by the `C_TransferKey` command, so we check that keys imported via $\mathcal{O}^{\texttt{C\_TransferKey}}$ were previously generated on the token. If this check fails, the new handle is given index 0. Unlike with $\mathcal{O}^{\texttt{C\_CreateObject}}$, we check that the template of the key matches the template it had when it was first generated. This is because the transfer mechanism is designed for keys of the highest privilege, so we must ensure that keys imported this way were always intended to have this role. As a result, the transfer mechanism cannot really

benefit the adversary, since they can only import a key with the same value and role as it had previously. We only include this oracle to model a system with multiple tokens.

Finally, when a key is imported via $\mathcal{O}^{\texttt{C\_UnwrapKey}}$, we check if the wrap had been previously generated by the token. To carry out this check, we maintain a list W of triples (h, h', w) such that the query $\mathcal{O}^{\texttt{C\_WrapKey}}(h, h')$ received the response w.[3] If the wrap submitted to $\mathcal{O}^{\texttt{C\_UnwrapKey}}$ was indeed generated by the token, we know the contents of the wrap, so the new handle is given the same index of the originally wrapped handle.[4] If the wrap submitted to $\mathcal{O}^{\texttt{C\_UnwrapKey}}$ was not generated by the token, then it was forged by the adversary. If the unwrapping key is compromised, then the new handle is assumed compromised and given index 0. This is because it is trivial to forge a wrap under a compromised key and so we do not allow the adversary to win the security game this way. However, if the unwrapping key is not already compromised, then the new handle is given a fresh (non-zero) index, even though there can be no security guarantees for the imported key. This allows the adversary to benefit from creating forged wraps without compromising the wrapping key, which is a realistic attack. It will be necessary for security to prove that this can never happen, using the integrity of the wrapping mechanism.

Now we give the formal definition of the security of a PKCS#11 configuration. Suppose $\mathsf{Adv}^{\mathsf{API}}(\mathcal{A}) \leq \epsilon$ for all adversaries $\mathcal{A}$ running in time at most t, making at most q oracle queries and such that the number of non-zero handle indices used in API^b, i.e. the number of keys generated by the token or imported into the token by forgeing a wrap under an uncompromised key, is at most n. Then we say the API is (t, q, n, ϵ)-*secure*.

4.2 Security Assumptions

In order for an API to securely support both symmetric and asymmetric cryptographic primitives, we have to assume that the encoding of keys is such that the three key classes cannot be confused.[5] More precisely, algorithms that are supposed to use secret keys will automatically fail if one tries to use a public key or a private key instead, and so on. This is necessary to avoid otherwise secure primitives exhibiting insecure behaviour (such as returning the value of the key) when used with a key of the wrong class. Moreover, when one imports a new key using the `C_CreateObject` command or the `C_UnwrapKey` command, the class of the new key will be automatically determined by the input to the command. We capture these assumptions in our formal syntax by having the keyspaces SecretKeys, PublicKeys and PrivateKeys be disjoint sets. These assumptions mean that,

[3] A real API does not need to maintain such a list; it is purely for preventing trivial attacks in our model.

[4] Actually it is given the minimal index of all wrapped handles satisfying these conditions, but if the API is secure then all these indices will agree, or they will all be in Comp.

[5] In practice, the length of the bitstring could determine the class of the key.

for example, a secure symmetric encryption scheme and a secure digital signature scheme are automatically jointly secure, but different primitives using the same class of keys, e.g. a symmetric encryption scheme and a MAC scheme, could still interfere with each other.

Furthermore, as explained above, the wrapping mechanism must provide *integrity* (in addition to secrecy) to prevent the adversary from importing their own keys. While we assume the wrapping mechanism authenticates the *values* of keys, we do not assume that the *attributes* of keys are authenticated. We remark that some wrapping mechanisms supported by early versions of PKCS#11, e.g. LYNKS from v2.20, attempted to authenticate the values of keys by adding an encrypted checksum to the ciphertext, which was then checked when unwrapping. On the other hand, even the latest version of PKCS#11 does not explicitly support including and authenticating the attributes of keys when wrapping. While we assume the use of a strong wrapping mechanism, we show how security can be achieved without any changes to the PKCS#11 standard.

5 Secure Templates

Since we do not assume that the PKCS#11 wrapping mechanism authenticates the attributes of keys, we have no way of knowing what the attributes of imported keys were when the keys were first generated. This means the API must impose attributes on imported keys regardless of user input.

Furthermore, it is very difficult to separate the roles of imported keys of the same class without authenticated attributes. This is because forcing the adversary to choose between templates of imported keys (such as unwrap and encrypt or unwrap and sign/verify) does not limit the adversary at all, since the adversary can just unwrap the same wrapped key twice with different roles. Moreover, if one tries to prevent this attack by rejecting unwraps of a ciphertext that has previously been unwrapped on the same token, the adversary can just unwrap the same key on multiple tokens and use them together. The only way to avoid this entirely is with a central log of all the operations performed on any token, as suggested by Cachin and Chandran, which is impractical for more than one token [5]. Since we do not assume that attributes are authenticated or that there is a central log of all operations, our configuration must have exactly one template for all imported keys of the same class. Under our assumption that the three classes of keys cannot be confused, we can have a different template for each class.

Recall that, in STP, imported secret keys can be used for encryption, but not decryption [4]. This is because these keys may be stored under a different handle with the ability to wrap other keys and so we must prevent the wrap/decrypt attack. Similarly, such keys can be used for unwrapping, but not wrapping, since they may be stored under a different handle with the ability to decrypt ciphertexts. However, this does not prevent all the attacks that we consider: STP is actually not secure in our model.

There are two reasons why we will not be able to reduce the security of STP to the confidentiality and integrity of the underlying symmetric encryption scheme. The first is technical: STP allows the creation of key cycles, since any key with `CKA_WRAP` set can wrap any key with `CKA_EXTR` set, and key cycles cannot be modelled by standard, computational security notions for encryption. However, one can prevent key cycles using the attributes `CKA_TRUSTED` and `CKA_WWT`: we allow the creation of *trusted* wrapping keys that are not extractable and *untrusted* wrapping keys that are extractable but can only be wrapped under trusted wrapping keys. Moreover, all imported secret keys must have `CKA_WWT` set, since they may be stored under a different handle as an untrusted wrapping key.

The second security flaw is more serious. While *Tookan* found no attacks against STP, this was with respect to a weak security notion that honestly-generated keys cannot be recovered by the adversary. Our stronger security notion requires that all keys on the token that are not trivially compromised are safe to use for encryption and signing. This means the attacker should not be able to import their own keys, which is why we need INT-CTXT security for the wrapping mechanism. However, since STP allows the same keys to be used for encryption and wrapping, the adversary could *encrypt* their own key and then *unwrap* the ciphertext, without violating the integrity property of the wrapping mechanism. The newly-imported key, known to the adversary, can then be used by the encryption challenge oracle, trivially leaking the hidden bit b. To prevent this attack, our policy must not allow the encryption (as opposed to wrapping) of any element of $\mathsf{SecretKeys}$.

Let STP+ be the PKCS#11 configuration obtained by restricting STP as described above, thereby preventing the creation of key cycles and the encryption, rather than wrapping, of secret keys. We will extend STP+ by enabling public-key encryption and signatures and our main result (Theorem 1) is a security reduction for this configuration to the security of the underlying primitives. As an immediate corollary, we see that the security of STP+ is implied by the confidentiality and integrity of the underlying symmetric encryption scheme.

In describing STP, Bortolozzo *et al.* did not consider MAC functionality [4]. As mentioned in Sect. 3, the extension of STP used in *Caml Crush* is such that secret keys can have both MAC and encrypt functionality. We also show in the full version of the paper that a secure MAC scheme and a secure encryption scheme are not always jointly secure [14]. Therefore, if we do not assume the joint security of the encryption and MAC schemes, we cannot prove the security of our configuration of PKCS#11 if it allows unwrapped secret keys to compute or verify MACs. Thus there is no generically secure way to exchange MAC keys between tokens and so we must only use (asymmetric) signatures to provide data authenticity.

Then, since unwrapped private keys need to be used to create signatures, such keys cannot be allowed to decrypt messages (without assuming the joint security of public key encryption and signing). So private decryption keys must be unextractable, meaning there is no way to safely transmit such keys between

tokens. However we do not need to disable public-key encryption altogether, since tokens can exchange public encryption keys over an authenticated channel and decrypt ciphertexts using their unextractable, locally-generated private keys.

Since tokens are required to transmit public keys for encryption and verifying signatures, it is quite possible for the adversary to use an encryption key to verify signatures, by generating the key in one role and then re-importing it with a different role. However, this does not affect the joint security of the encryption scheme and the signature scheme. The verification algorithm has no way of knowing that the key it uses was 'intended' as an encryption key and will function as normal. Moreover, as the key is public there is no risk from leaking parts of the key not needed for verification. Similarly there is no risk from encrypting data using keys intended for signature verification. In summary, it is not necessary to authenticate the attributes of public keys, only the *values* of these keys. As a result our configuration of PKCS#11 allows all imported public keys to have both encryption and verification capabilities.

Bringing together this analysis, we obtain a set of attribute templates that, without assuming the joint security of different primitives, is maximal among those with which the API is secure:

1. Generated secret keys must have one of the following templates:
   (a) `TRUSTED`: trusted wrapping keys that are unextractable and cannot be used for encryption or decryption,
   (b) `UNTRUSTED`: untrusted wrapping keys that can themselves be wrapped under trusted wrapping keys, but cannot be used for encryption or decryption,
   (c) `ENC`: keys that can be wrapped and used for encryption and decryption, but cannot wrap other keys.
2. Imported secret keys have the template `IMPORTSECRET`: they can encrypt data and unwrap keys, but cannot decrypt data or wrap keys. To prevent key cycles, imported secret keys must only be wrapped under trusted wrapping keys.
3. Only trusted wrapping keys, i.e. keys with template `TRUSTED`, can be transferred using the secure out-of-band mechanism `C_TransferKey` (for bootstrapping).
4. The templates of generated public and private key pairs must be one of the following:
   (a) `AENC`, `ADEC`: the public key can encrypt data and the private key can decrypt data; neither can wrap or unwrap and the private key is *not extractable*.
   (b) `VERIFY`, `SIGN`: the public key can verify signatures and the private key can create signatures; neither can wrap or unwrap and both are extractable.
5. Finally, imported public keys must have the template `IMPORTPUBLIC`: such keys can encrypt data and verify signatures, but cannot wrap or unwrap keys.

In Tables 1 and 2, we define our set of secure templates with respect to the PKCS#11 attributes `CKA_EXTR`, `CKA_WWT`, `CKA_TRUSTED`, `CKA_WRAP`, `CKA_UNWRAP`, `CKA_ENCRYPT`, `CKA_DECRYPT`, `CKA_SIGN`, and `CKA_VERIFY`. Any attributes from this

Table 1. Templates for Secret Keys (note that CKA_SIGN and CKA_VERIFY are always unset). The attribute CKA_TRUSTED, not shown here, is set in the template TRUSTED and unset in all other templates.

Template Name	CKA_EXTR	CKA_WWT	CKA_WRAP	CKA_UNWRAP	CKA_ENCRYPT	CKA_DECRYPT
TRUSTED			✓	✓		
UNTRUSTED	✓	✓	✓	✓		
ENC	✓				✓	✓
IMPORTSECRET	✓	✓		✓	✓	

Table 2. Templates for Public and Private Keys (note that CKA_TRUSTED, CKA_WRAP and CKA_UNWRAP are always unset).

Template Name	CKA_EXTR	CKA_WWT	CKA_ENCRYPT	CKA_DECRYPT	CKA_SIGN	CKA_VERIFY
AENC	✓		✓			
ADEC				✓		
SIGN	✓				✓	
VERIFY	✓					✓
IMPORTPUBLIC	✓		✓			✓

set that are not shown in the tables, or not marked with ✓, are unset. The only exception to this rule is CKA_TRUSTED, which is not shown in any of the tables due to limitations on space, but is set in the template TRUSTED and unset in all other templates.

The policy P used in our configuration is given in Table 3. We remark that $\mathtt{P}(\mathtt{C\_UnwrapKey}, h, w, t)$ sometimes depends on the value of $\mathsf{Dec}(h.\mathsf{key}, w)$. Since $h.\mathsf{key}$ is not accessible to the API, what this means is that the API makes the relevant decryption call to the token, receives a response, and then determines whether or not to release the response to the user based on its value. Note that this policy could not be achieved by simply using a filter (like *Caml Crush*). For comparison, we also give the default PKCS#11 policy and the STP+ policy in the full version of the paper [14]. One can see that our configuration is indeed PKCS#11 compliant and STP+ is a special case of our configuration.

Let t_{max} be the maximum run time of any of the following operations: Enc, AEnc, ADec, Sign, SVrfy, one call to NewHandle and one call to Dec; one call to NewHandle and two calls to KG; and two calls to NewHandle and two calls to KPG. Then, with the configuration presented here, we obtain our main result, which is proved in the full version of the paper [14]:

Theorem 1. *Suppose* P *is as defined in Table 3,* $\mathcal{E}$ *is* (t, ϵ_1)*-IND-CCA-secure and* (t, ϵ_2)*-INT-CTXT secure,* $\mathcal{PKE}$ *is* (t, ϵ_3)*-IND-CCA-secure and* $\mathcal{S}$ *is* (t, ϵ_4)*-EUF-CMA-secure. Then the API is* (t', q, n, ϵ')*-secure, where:*

$$t' = t - q \cdot t_{max}, \quad \epsilon' = n\left[\left(8n^2 + 4n + 1\right)\epsilon_1 + 2\epsilon_2 + \epsilon_3 + \epsilon_4\right].$$

Table 3. The policy of our configuration (where $a \in h.\mathsf{temp}$ means that the attribute a is set in $h.\mathsf{temp}$)

Function	Value
$\mathtt{P}(\mathtt{C\_CreateObject}, pk, t)$	1 if $t = \mathtt{IMPORTPUBLIC}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_TransferKey}, k, t)$	1 if $t = \mathtt{TRUSTED}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_GenerateKey}, t)$	1 if $t \in \{\mathtt{TRUSTED}, \mathtt{UNTRUSTED}, \mathtt{ENC}\}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_GenerateKeyPair}, t, t')$	1 if $(t, t') \in \{(\mathtt{AENC}, \mathtt{ADEC}), (\mathtt{VERIFY}, \mathtt{SIGN})\}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_WrapKey}, h, h')$	1 if $\mathtt{CKA\_WRAP} \in h.\mathsf{temp}, \mathtt{CKA\_EXTR} \in h'.\mathsf{temp}$ and if $\mathtt{CKA\_WWT} \in h'\mathsf{temp}$ then $\mathtt{CKA\_TRUSTED} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_UnwrapKey}, h, w, t)$	1 if $\mathtt{CKA\_UNWRAP} \in h.\mathsf{temp}$ and $\mathsf{Dec}(h.\mathsf{key}, w) \in \mathsf{SecretKeys}$ and $t = \mathtt{IMPORTSECRET}$ or $\mathsf{Dec}(h.\mathsf{key}, w) \in \mathsf{PrivateKeys}$ and $t = \mathtt{SIGN}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Encrypt}, h, m)$	1 if $\mathtt{CKA\_ENCRYPT} \in h.\mathsf{temp}$ and $m \notin \mathsf{SecretKeys}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Decrypt}, h, c)$	1 if $\mathtt{CKA\_DECRYPT} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Sign}, h, m)$	1 if $\mathtt{CKA\_SIGN} \in h.\mathsf{temp}$, 0 otherwise
$\mathtt{P}(\mathtt{C\_Verify}, h, m, s)$	1 if $\mathtt{CKA\_VERIFY} \in h.\mathsf{temp}$, 0 otherwise

6 Conclusion and Acknowledgements

We have given a security definition for configurations of PKCS#11, where the adversary can adaptively corrupt keys. We proved the security, in this strong attacker model, of a configuration of PKCS#11 that extends the Secure Templates Patch from Bortolozzo *et al.* [4]. Unlike most existing analyses of APIs in the literature, we do not assume the attributes of keys are authenticated when wrapping.

Our result holds under the assumption that private, public and secret keys cannot be confused. Moreover, since our configuration does not support asymmetric key wrapping, we have to assume for bootstrapping that there is a secure channel for transmitting long-term secret keys and also an authenticated channel for transmitting public keys. We feel that these assumptions are likely to hold in practice.

Our security proof is far from tight: the advantage of the adversary against the API is potentially n^3 times bigger than the advantage against the underlying

symmetric encryption scheme used for wrapping, where n is an upper-bound on the number of distinct keys stored on the token. Whether such losses can ever be avoided is the subject of ongoing research.

The author would like to thank Bogdan Warinschi, Martijn Stam and the anonymous reviewers for their useful feedback on the paper.

References

1. Benadjila, R., Calderon, T., Daubignard, M.: Caml crush: a PKCS#11 filtering proxy. In: Joye, M., Moradi, A. (eds.) CARDIS 2014. LNCS, vol. 8968, pp. 173–192. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16763-3_11
2. Benadjila, R., Calderon, T., Daubignard, M.: Source code for Caml Crush (2016). https://github.com/ANSSI-FR/caml-crush. Accessed 19 Oct 2016
3. Bond, M., Anderson, R.J.: API-level attacks on embedded systems. IEEE Comput. **34**(10), 67–75 (2001)
4. Bortolozzo, M., Centenaro, M., Focardi, R., Steel, G.: Attacking and fixing PKCS#11 security tokens. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, 4–8 October 2010, pp. 260–269 (2010)
5. Cachin, C., Chandran, N.: A secure cryptographic token interface. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, 8–10 July 2009, pp. 141–153 (2009)
6. Clulow, J.: On the security of PKCS #11. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 411–425. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45238-6_32
7. Delaune, S., Kremer, S., Steel, G.: Formal analysis of PKCS#11. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23–25 June 2008, pp. 331–344 (2008)
8. Delaune, S., Kremer, S., Steel, G.: Formal security analysis of PKCS#11 and proprietary extensions. J. Comput. Secur. **18**(6), 1211–1245 (2010)
9. Kremer, S., Steel, G., Warinschi, B.: Security for key management interfaces. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27–29 June 2011, pp. 266–280 (2011)
10. PKCS#11 cryptographic token interface base specification version 2.40, April 2015. http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/pkcs11-base-v2.40.html
11. Panjwani, S.: Tackling adaptive corruptions in multicast encryption protocols. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 21–40. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_2
12. Scerri, G., Stanley-Oakes, R.: Analysis of key wrapping APIs: generic policies, computational security. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, 27 June–1 July 2016, pp. 281–295. IEEE (2016)
13. Shrimpton, T., Stam, M., Warinschi, B.: A modular treatment of cryptographic APIs: the symmetric-key case. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9814, pp. 277–307. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53018-4_11
14. Stanley-Oakes, R.: A provably secure PKCS#11 configuration without authenticated attributes. Cryptology ePrint Archive, Report 2017/158 (2017). http://eprint.iacr.org/2017/134