Type-based Analysis of PKCS#11 Key Management[⋆]

Matteo Centenaro[1] and Riccardo Focardi[1] and Flaminia L. Luccio[1]

DAIS, Università Ca' Foscari Venezia, Italy
{mcentena,focardi,luccio}@dsi.unive.it

Abstract. PKCS#11, is a security API for cryptographic tokens. It is known to be vulnerable to attacks which can directly extract, as cleartext, the value of sensitive keys. In particular, the API does not impose any limitation on the different roles a key can assume, and it permits to perform conflicting operations such as asking the token to wrap a key with another one and then to decrypt it. Fixes proposed in the literature, or implemented in real devices, impose policies restricting key roles and token functionalities. In this paper we define a simple imperative programming language, suitable to code PKCS#11 symmetric key management, and we develop a type-based analysis to prove that the secrecy of sensitive keys is preserved under a certain policy. We formally analyse existing fixes for PKCS#11 and we propose a new one, which is type-checkable and prevents conflicting roles by *deriving* different keys for different roles.

1 Introduction

PKCS#11, also known as Cryptoki, defines a widely adopted API for cryptographic tokens [18]. It provides access to cryptographic functionalities while, in principle, providing some security properties. More specifically, the value of keys stored on a PKCS#11 device and tagged as *sensitive* should never be revealed outside the token, even when connected to a compromised host. Unfortunately, PKCS#11 is known to be vulnerable to attacks that break this property [4,8,10].

An application initiates a *session* with a PKCS#11 compliant device by first supplying a PIN, and then accessing the functionalities provided by the token. There may be various *objects* stored in the token, such as cryptographic keys and certificates. Objects are referenced via *handles* to permit, e.g., that a cryptographic key is used without necessarily knowing its value: we can ask a token to encrypt some data just providing a handle to the encryption key. The value of a key is one of the *attributes* of the enclosing object. There are other attributes to specify the various roles a key can assume: each different API call can, in fact, require a different role. For example, decryption keys are required to have attribute `CKA_DECRYPT` set, while key-encrypting keys, i.e., keys used to encrypt other keys, must have attribute `CKA_WRAP` set.

[⋆] Work partially supported by the RAS Project "*TESLA: Techniques for Enforcing Security in Languages and Applications*".

The attacks on PKCS#11 we consider in this paper are at the API level [1,2,3,4,7,8,10,16], i.e., the attacker is assumed to control the host on which the token is connected and to perform any sequence of (legal) API calls. The crucial functionalities of PKCS#11 are the ones for exporting and importing sensitive keys, called `C_WrapKey` and `C_UnwrapKey`. The former performs the encryption of a key under another one, giving as output the resulting ciphertext, and the latter performs the corresponding decrypt and import into the token. They allow for exporting and reimporting keys, in an encrypted form. Note that, having a wrapping key (`CKA_WRAP`) which can also be used for decryption (`CKA_DECRYPT`) is dangerous and leads to the following simple 'wrap-decrypt' API-level attack:

```
h_myKey = C_GenerateKey({CKA_DECRYPT, CKA_WRAP});
wrapped = C_WrapKey(h_sensitiveKey, h_myKey);
leak = C_Decrypt(wrapped, h_myKey);
```

First, we ask the token to generate a new key with attributes `CKA_DECRYPT`, `CKA_WRAP` set. Then, we use this key to wrap an existing sensitive key referenced by `h_sensitiveKey`. Finally, we ask the token to decrypt the resulting ciphertext using again the freshly generated key. Since it is the same key used for wrapping, we obtain the value of the sensitive key in the clear.

A recent work [4] has shown that the state of the art in PKCS#11 security tokens is rather poor: many existing commercially available devices are vulnerable to attacks similar to the above one; the secured ones, instead, prevent the attacks by completely removing wrapping functionalities. However, it has been shown that the API can be 'patched' without necessarily cutting down so much on its functionalities [4,10]: this can be done by (*i*) imposing a policy on the attributes so that a key cannot be used for conflicting operations; (*ii*) limiting the way attributes can be changed so to avoid that conflicting attributes are set at two different instants; (*iii*) either adding a wrapping format which binds attributes to wrapped keys [10] or limiting very carefully the usage of imported keys to a subset of non-critical functions [4].

In our opinion, formal tools to reason about the security of different implementations of PKCS#11 APIs, such as Tookan [4], are fundamental to help developers and hardware producers to detect and better understand the causes of the bugs affecting the implementations, and they are very important for the testing of new patches.

Our contribution. In this paper we (*i*) define a simple imperative programming language, suitable to code PKCS#11 APIs for symmetric key management; (*ii*) formalize a Dolev-Yao attacker and API security in this setting; (*iii*) present a type system to statically enforce API security; (*iv*) propose a new fix for PKCS#11 based on key-diversification; (*v*) apply the type system to validate our new fix and one previously proposed in [4,5]. We only consider functions for encryption/decryption of data and wrap/unwrap of keys as these are the most relevant ones for what concerns API-level attacks.

The language is, by itself, an original contribution as PKCS#11 is typically modelled following a 'black-box' approach: each API function takes some input values and a (representation of a) device, and returns new values possibly modifying the device state. This is done in one step, disregarding the internal single steps (see, e.g., [10]). Our target is to perform a language-based analysis of the API specification, and this requires that APIs are specified as sequences of internal commands and lower level calls to the device. The attacker is modelled in a classic Dolev-Yao style: he can perform any cryptographic operation once he knows the corresponding key. He can also execute any API call passing, as parameters, values that he knows, and incrementing his knowledge with the returned value. API security requires that sensitive keys that are not already known by the attacker, and *always-sensitive* keys (special sensitive keys that have been generated inside the token) will never be disclosed to the attacker.

Our type system statically enforces API security by checking that keys can only be wrapped using *trusted* keys and every key has a clear, unambiguous role. Typing is parametrized with respect to a policy dictating the possible attributes that can be simultaneously set on a key and the ones that are set when unwrapping/importing a new key in a device. We prove that type-checked APIs are secure against a Dolev-Yao attacker. Using the proposed type system we analyse the Secure Templates fix proposed in [4,5], and we prove it secure. We then propose a new patch, based on key-diversification, a standard cryptographic technique to derive a new key from a known one. Our idea, is to explicitly require that keys for different roles will always be different. To the best of our knowledge, key-diversification has previously never been adopted as a systematic mechanism to secure key management of cryptographic tokens. We finally prove that this new patch type-checks.

Related work. The most established work on formal analysis of PKCS#11 is [10]. In this paper, it is given a model of a fragment of PKCS#11 and a model-checking procedure to look for possible attack sequences. Interesting abstractions to reduce state explosion and to analyse unbounded fresh data have been given in [14]. In [4], the theory has been engineered into Tookan, a tool for the analysis of real devices. The tool is able to build a formal model of a real token, perform model-checking and try the theoretical attacks on a real device. Once the model is extracted from the token, it is also possible to try new fixes are check again for existing attacks.

Our present contribution extends this line of research by exploring a language-based, static analysis technique that allows for proving the security of PKCS#11 APIs and their fixes. We in fact intend to integrate this type-based analysis in Tookan. The contribution is also in the line of other type-based analyses on different settings: For what concerns Bank APIs in [6] it is studied the security of PIN managements Hardware Security Modules and it is given a type system to prove their security; in [11] we have given a type system for the security of rechargeable disposable RFID tickets.

A recent line of research [12,13] investigates models of PKCS#11 based on first-order linear time logic extended by past operators. The motivation is, again, to check the security of the PKCS#11 configuration, but the underlying model is completely different. A comparison between the two models is for sure an interesting future issue.

In [15] Keighren, Aspinall, and Steel propose a type system to check information flow properties for cryptographic operations in security APIs. There seem to be many differences with our contribution: (*i*) the target property is different: Here we consider confidentiality of sensitive keys while in [15] the authors investigate *noninterference*, a much stronger property. In this sense their result is more in the line of [6]; (*ii*) their model is very general and allows for reasoning on cryptographic operations so that the wrap/decrypt attack is modelled as a forbidden information flow from secret to public. No language is given to express internal commands. Our language allows for specifying PKCS#11 key management APIs at a fine granularity, and the same attack is prevented by avoiding conflicting roles for the same key. This is why we can avoid the complex treatment of noninterference and only focus on key confidentiality; (*iii*) Keighren, Aspinall, and Steel only considers confidentiality and do not treat integrity (or trust) that is one of the crucial ingredient of our analysis: only trusted keys should be used to wrap sensitive keys. A more detailed comparison will be the subject of future work.

Paper structure. The paper is organized as follows. In section 2 we introduce the simple imperative language for PKCS#11 key management, the attacker model and the notion of API security; in section 3 we present the type system statically enforcing API security; in section 4 we type-check known implementations of PKCS#11 key management APIs, and we propose our new fix based on key-diversification, which we prove to be secure. We conclude in section 5.

2 A language for PKCS#11 key management

In this section we first introduce a simple imperative language suitable to specify PKCS#11 key management APIs. We then formalize the attacker model and define API security.

Values. We let $\mathcal{C}$ and $\mathcal{G}$, with $\mathcal{C} \cap \mathcal{G} = \emptyset$, respectively be the set of atomic *constant* and *fresh* values. The former is used to model any public data, including non-sensitive keys; the latter models the generation of new fresh values such as sensitive keys. We associate to $\mathcal{G}$ an extraction operator $g \leftarrow \mathcal{G}$, representing the extraction of the first 'unused' value g from $\mathcal{G}$. Extracted values are always different: two, even non-consecutive, extractions $g \leftarrow \mathcal{G}$ and $g' \leftarrow \mathcal{G}$ are always such that $g \neq g'$. We let *val* range over the set of all atomic values $\mathcal{C} \cup \mathcal{G}$ and we define values v as follows:

$$v ::= \mathit{val} \mid \mathit{enc}(v, v') \mid \mathit{dec}(v, v') \mid \mathit{kdf}(v, v')$$

where $\mathit{enc}(v, v')$ and $\mathit{dec}(v, v')$ denote value v respectively encrypted and decrypted under key v', and $\mathit{kdf}(v, v')$ represents a new key obtained via diversification from a value v and another key v'. Key diversification may be implemented in many different ways. For example, using the encryption scheme, we can directly obtain $kdf(v, v')$ as $enc(v, v')$. We explicitly represent decrypted values in order to model situations in which a wrong key is used to decrypt an encrypted value: for example, the decryption under v' of $enc(v, v')$ will give, as expected, value v; instead, the decryption under v' of $enc(v, v'')$, with $v'' \neq v'$ will be explicitly represented as $dec(enc(v, v''), v')$. This allows us to model a cryptosystem with no integrity check, as the one used in PKCS#11 for symmetric keys: decrypting with a wrong key never gives a failure.

Expressions. Our language is composed of a core set of expressions for manipulating the above values. Expressions are based on a set of variables $\mathcal{V}$ ranged over by x, and have the following syntax:

$$e ::= x \mid \mathsf{enc}(e, x) \mid \mathsf{dec}(e, x) \mid \mathsf{kdf}(val, x)$$

The explicit tag val will simplify typing for key diversification. A memory $\mathsf{M} : x \mapsto v$ is a partial mapping from variables to values and $e \downarrow^{\mathsf{M}} v$ denotes that the evaluation of the expression e in memory M leads to value v. Let $e \downarrow^{\mathsf{M}} v$ and $\mathsf{M}(x) = v'$. The semantics of expressions follows:

$$\begin{aligned}
x \downarrow^{\mathsf{M}} &\; M(x) \qquad\qquad \text{if } M(x) \text{ is defined}\\
\mathsf{enc}(e, x) \downarrow^{\mathsf{M}} &\; enc(v, v')\\
\mathsf{dec}(e, x) \downarrow^{\mathsf{M}} &\begin{cases} v'' & \text{if } v = enc(v'', v')\\ dec(v, v') & \text{otherwise}\end{cases}\\
\mathsf{kdf}(val, x) \downarrow^{\mathsf{M}} &\; kdf(val, v')
\end{aligned}$$

The modeled encryption mechanism does not perform any integrity check on the messages, so the decryption of a ciphertext under a wrong key gives $dec(v, v')$.

Templates. Properties and capabilities of keys are described by templates, ranged over by T, represented as a set of *attributes*. When a certain attribute is contained in a template T we will say that the attribute is set, it is unset otherwise. A key can be *sensitive*, and a sensitive key can also be *always-sensitive* if it has been generated (as a sensitive key) by a secure device. These two properties are described by the attributes S (sensitive), and A (always-sensitive). Four attributes identify the capabilities of a key: data encryption (E) and decryption (D), wrap (W) and unwrap (U), i.e., encryption and decryption of other keys. Formally, a template T is a subset of $\{S, A, E, D, W, U\}$ under the constraint $S \notin T$ implies $A \notin T$, i.e., non-sensitive keys can never be always-sensitive.

APIs and tokens. An API is specified as a set $\mathcal{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_n\}$ of functions, each one composed of simple sequences of assignment commands:

$$\begin{aligned}
&\mathsf{a} ::= \lambda x_1, \ldots, x_k.\mathsf{c}\\
&\mathsf{c} ::= x := e \mid x := \mathsf{f} \mid \mathsf{return}\ e \mid \mathsf{c}_1; \mathsf{c}_2\\
&\mathsf{f} ::= \mathsf{getObj}(y) \mid \mathsf{checkTemplate}(y, T) \mid \mathsf{genKey}(T) \mid \mathsf{importKey}(y, T)
\end{aligned}$$

We will only consider API commands in which $\mathsf{return}\ e$ can only occur as the last command. Internal functions f represent operations that can be performed on the underlying devices. Note that these functions are used to implement the APIs and are not directly available to the users. Intuitively, getObj retrieves the plaintext value of a key stored in the device, given its handle y; $\mathsf{checkTemplate}$ is similar but it additionally queries the template of the stored key: if the key template 'matches', i.e., is a superset of, the given one T, the key is returned; genKey generates a key with template T; finally, $\mathsf{importKey}$ imports a new key with plaintext value y and template T. The first two functions fail (i.e., are stuck) if the given handle does not exists or refers to a key with a wrong template. A call to an API $\mathsf{a} = \lambda x_1, \ldots, x_k.\mathsf{c}$, written $\mathsf{a}(v_1, \ldots, v_k)$, binds $x_1, \ldots, x_k$ to values $v_1, \ldots, v_k$, executes c and outputs the value given by $\mathsf{return}\ e$.

Example 1 (PKCS#11 `C_WrapKey` *command).* The language introduced is suitable to implement PKCS#11 commands. Each API command will be modeled as a procedure reading inputs from pre-defined variables and returning a value as output. The following is a possible specification of the wrap command. It takes the handles of a key to be wrapped and the one pointing to the wrapping key (whose flags W and S have to be set, as it has to be a sensitive wrapping key) returning an encrypted byte-stream. For the sake of readability, we will always write $\mathsf{a}(x_1, \ldots, x_k)\ \mathsf{c}$ in place of $\mathsf{a} = \lambda x_1, \ldots, x_k.\mathsf{c}$ to specify an API function:

```
C_WrapKey($h\_key$, $h\_w$)
  $w := \mathsf{checkTemplate}(h\_w,\ \{S, W\})$;
  $k := \mathsf{getObj}(h\_key)$;
  $\mathsf{return\ enc}(k, w)$;
```

Device keys are modelled by the handle-map $\mathsf{H} : g \mapsto (v, T)$, a partial mapping from the atomic (generated) values to pairs of keys and templates. Each key has a handle to be referred with, and a template. Notice that we do not distinguish between one or many devices: we consider all keys available to the API as a unique 'universal' PKCS#11 token. This corresponds to a worst-case scenario in which attackers can simultaneously access all existing tokens. Notice, also, that this does not limit the multiple presence of the same key value under different handles or templates, as for example, with $\mathsf{H}(g) = (v, T)$ and $\mathsf{H}(g') = (v, T')$.

An API command c working on a memory M and handle-map H is noted as $\langle \mathsf{M}, \mathsf{H}, \mathsf{c} \rangle$. Semantics is reported in Table 1, where ϵ denotes the empty API. We explain the first rule for assignment $x := e$: it evaluates expression e on M and stores the results in variable x, noted $\mathsf{M}[x \mapsto v]$. In case x is not defined in M the domain of M is extended to include the new variable, otherwise the value stored in x is overwritten. Other rules are similar in spirit. Notice that genKey and $\mathsf{importKey}$ also modify the handle-map. The last rule is for API calls on an handle-map H: parameter values are assigned to variables of an empty memory M_ϵ, i.e., a memory with no variables mapped to values (recall memories are partial functions); then, the API commands are executed and the return value is given as a result of the call. This is noted $\mathsf{a}(v_1, \ldots, v_k) \downharpoonleft^{\mathsf{H},\mathsf{H}'} v$ where H' is the resulting handle map. Notice that at this API level we do not observe memories

$$\frac{e \downarrow^{\mathsf{M}} v}{\langle \mathsf{M}, \mathsf{H}, x := e\rangle \rightarrow \langle \mathsf{M}[x \mapsto v], \mathsf{H}, \varepsilon\rangle} \qquad \frac{\mathsf{H}(\mathsf{M}(y)) = (v, T)}{\langle \mathsf{M}, \mathsf{H}, x := \mathsf{getObj}(y)\rangle \rightarrow \langle \mathsf{M}[x \mapsto v], \mathsf{H}, \varepsilon\rangle}$$

$$\frac{\mathsf{H}(\mathsf{M}(y)) = (v, T') \quad T \subseteq T'}{\langle \mathsf{M}, \mathsf{H}, x := \mathsf{checkTemplate}(y, T)\rangle \rightarrow \langle \mathsf{M}[x \mapsto v], \mathsf{H}, \varepsilon\rangle}$$

$$\frac{g, g' \leftarrow \mathcal{G}}{\langle \mathsf{M}, \mathsf{H}, x := \mathsf{genKey}(T)\rangle \rightarrow \langle \mathsf{M}[x \mapsto g], \mathsf{H}[g \mapsto (g', T)], \varepsilon\rangle}$$

$$\frac{g \leftarrow \mathcal{G}}{\langle \mathsf{M}, \mathsf{H}, x := \mathsf{importKey}(y, T)\rangle \rightarrow \langle \mathsf{M}[x \mapsto g], \mathsf{H}[g \mapsto (\mathsf{M}(y), T)], \varepsilon\rangle}$$

$$\frac{\langle \mathsf{M}, \mathsf{H}, \mathsf{c}_1\rangle \rightarrow \langle \mathsf{M}', \mathsf{H}', \varepsilon\rangle}{\langle \mathsf{M}, \mathsf{H}, \mathsf{c}_1; \mathsf{c}_2\rangle \rightarrow \langle \mathsf{M}', \mathsf{H}', \mathsf{c}_2\rangle} \qquad \frac{\langle \mathsf{M}, \mathsf{H}, \mathsf{c}_1\rangle \rightarrow \langle \mathsf{M}', \mathsf{H}', \mathsf{c}_1'\rangle}{\langle \mathsf{M}, \mathsf{H}, \mathsf{c}_1; \mathsf{c}_2\rangle \rightarrow \langle \mathsf{M}', \mathsf{H}', \mathsf{c}_1'; \mathsf{c}_2\rangle}$$

$$\frac{\mathsf{a} = \lambda x_1, \ldots, x_k.\mathsf{c} \quad \langle \mathsf{M}_\epsilon[x_1 \mapsto v_1 \ldots x_k \mapsto v_k], \mathsf{H}, \mathsf{c}\rangle \rightarrow \langle \mathsf{M}', \mathsf{H}', \mathsf{return}\ e\rangle \quad e \downarrow^{\mathsf{M}'} v}{\mathsf{a}(v_1, \ldots, v_k) \downarrow^{\mathsf{H},\mathsf{H}'} v}$$

Table 1. API Semantics

that are, in fact, used internally by the device to execute the function. The only exchanged data are the input parameters and the return value.

Example 2 (Semantics of `C_WrapKey`*).* To illustrate the semantics, we now show the transitions of the `C_WrapKey` command specified above. Suppose that the device associates the handle g to $(v, \{A, S, E, D\})$ and g' to $(v', \{S, W, U\})$. We consider a memory M where all the variables are set to zero except for h_key and h_w which store respectively g and g', i.e., $\mathsf{M} = \mathsf{M}_\epsilon[h\_key \mapsto g, h\_w \mapsto g']$. Then it follows,

$$\begin{aligned}
&\langle \mathsf{M}, \mathsf{H}, w := \mathsf{checkTemplate}(h\_w, \{S, W\}); k := \mathsf{getObj}(h\_key); \mathsf{return}\ \mathsf{enc}(k, w)\rangle \\
&\rightarrow \langle \mathsf{M}[w \mapsto v'], \mathsf{H}, k := \mathsf{getObj}(h\_key); \mathsf{return}\ \mathsf{enc}(k, w)\rangle \\
&\rightarrow \langle \mathsf{M}[w \mapsto v', k \mapsto v], \mathsf{H}, \mathsf{return}\ \mathsf{enc}(k, w)\rangle
\end{aligned}$$

which gives $\texttt{C\_WrapKey}(g, g') \downarrow^{\mathsf{H},\mathsf{H}} enc(v, v')$ meaning that the value returned invoking the wrap command is thus the encryption of v under v'. Obviously, this is safe as long as v' is not know outside the device, otherwise a user knowing the raw value of the key used to wrap could retrieve v by simply computing $\mathsf{dec}(enc(v, v'), v')$.

Attacker Model. We now formalize the attacker in a classic Dolev-Yao style. In particular, the attacker knowledge $\mathcal{K}(V)$ deducible from a set of values V is defined as the least superset of V such that $v, v' \in \mathcal{K}(V)$ implies

(1) $enc(v, v') \in \mathcal{K}(V)$;
(2) $kdf(v, v') \in \mathcal{K}(V)$;
(3) if $v = enc(v'', v')$ then $v'' \in \mathcal{K}(V)$;
(4) if $v \neq enc(v'', v')$ then $dec(v, v') \in \mathcal{K}(V)$.

Given a handle map H, representing tokens, and an API $\mathcal{A} = \{\mathsf{a}_1, \dots, \mathsf{a}_n\}$, the attacker can invoke any API function giving any of the known values as a parameter. The returned value is then added to the knowledge. Formally, an attacker configuration is represented as $\langle \mathsf{H}, V \rangle$ and evolves as follows:

$$\frac{\mathsf{a} \in \mathcal{A} \qquad v_1, \dots, v_k \in \mathcal{K}(V) \qquad \mathsf{a}(v_1, \dots, v_k) \downarrow^{\mathsf{H},\mathsf{H}'} v}{\langle \mathsf{H}, V \rangle \rightsquigarrow_{\mathcal{A}} \langle \mathsf{H}', V \cup \{v\} \rangle}$$

We assume that the attacker initially knows an arbitrary subset V_0 of the constant atomic values $\mathcal{C}$ and we consider an initial empty handle map H_0. In the following, we use the standard notation $\rightsquigarrow^*_{\mathcal{A}}$ to note multi-step reductions.

API security. The main property required by PKCS #11: "Sensitive keys cannot be revealed in plaintext off the token" [18, page 30], is modelled by requiring that sensitive keys, that are not already known by the attacker, should never be learned by the attacker. Moreover, we formalize the intuitive property that always-sensitive keys and all keys derived from them, are never known by the attacker. This will be useful to guarantee that such keys have not been imported by the attacker and can be trusted.

Formally, sensitive keys are the ones that only appear in the handle map with the attribute sensitive set. Always-sensitive keys additionally have the always-sensitive attribute set.

Definition 1 (Sensitive and always-sensitive values). *Let val be an atomic value and* H *a handle-map. If val is such that* $\mathsf{H}(g) = (val, T)$ *implies* $S \in T$ *we say that val is sensitive in* H*. If we additionally have that* $\mathsf{H}(g) = (val, T)$ *implies* $A \in T$ *we say that val is always-sensitive in* H*.*

The definition of API security follows.

Definition 2 (API Security). *Let* $\mathcal{A}$ *be an API. We say that* $\mathcal{A}$ *is secure if for all reductions* $\langle \mathsf{H}_0, V_0 \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}, V \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}', V' \rangle$ *and for all atomic values val we have*

1. *val* $\notin \mathcal{K}(V)$ *and val is sensitive in* H *imply val* $\notin \mathcal{K}(V')$*;*
2. *val is always-sensitive in* H *implies* $val, \mathsf{kdf}(v, val) \notin \mathcal{K}(V) \cup \mathcal{K}(V')$*.*

3 Type system

We enforce the security of an API through a type system requiring that (i) every key has a clear, unambiguous role, and (ii) keys can only be wrapped using trusted keys. This latter idea is, in fact, suggested in PKCS#11 v2.20 [18]: `CKA_TRUSTED` keys are added by the security officer in a protected environment. Keys with the `CKA_WRAP_WITH_TRUSTED` attribute (that we do not model here) set can only be wrapped via such security officer keys. In fact, here it is like we were assuming that `CKA_WRAP_WITH_TRUSTED` is always set.

Our type system generalizes this idea of trusted keys by also including the ones generated by the device (always-sensitive). Even in this case, in fact, we

$$\frac{A \notin T, S \in T \quad \neg data(T) \vee wrap(T)}{\vdash T : \mathsf{Any}} \qquad \frac{A \notin T, S \in T \quad data(T) \quad \neg wrap(T)}{\vdash T : \mathsf{Data}}$$

$$\frac{A, S \in T \quad data(T) \quad \neg wrap(T)}{\vdash T : \mathsf{TData}} \qquad \frac{A, S \in T \quad \neg data(T) \quad wrap(T)}{\vdash T : \mathsf{Wrap}}$$

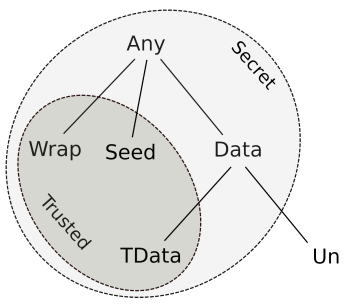
$$\frac{A, S \in T \quad data(T) \Leftrightarrow wrap(T)}{\vdash T : \mathsf{Seed}} \qquad \frac{A, S \notin T}{\vdash T : \mathsf{Un}}$$

Table 2. Typing templates

are guaranteed that their value has never appeared as plain-text outside the device. This will allow us to propose and analyse configurations in which always-sensitive keys can be exchanged by users. This is not allowed for trusted security officer keys. In the following we will then use the word trusted to refer to a key that is guaranteed to be unknown to the attacker. We will use the attribute always-sensitive to capture this fact, but we could easily extend the analysis to incorporate the above discussed attribute trusted. We consider the following types.

$$\rho ::= \mathsf{Any} \mid \mathsf{Data} \mid \mathsf{TData} \mid \mathsf{Wrap} \mid \mathsf{Seed} \mid \mathsf{Un}$$
$$\tau ::= \rho \mid \mathsf{Wrap}[\rho]$$

Intuitively, Any is the top type including all possible data and keys; type Data and TData are, respectively, for secret and trusted keys used to encrypt and decrypt data; Wrap is for trusted wrapping keys, i.e., keys used to encrypt other keys, and Seed is for trusted keys used to derive other keys via diversification; $\mathsf{Wrap}[\rho]$ is for trusted wrapping keys transporting keys of type ρ, obtained via diversification from some (trusted) seed; finally, Un represents untrusted values. Types are related by a subtyping relation $\leq$ depicted on the right. Notice that the level of secrecy can only grow while the level of trust can only decrease. Promoting a type via subtyping, in fact, should not compromise security. This will be proved in lemma 1 below.



Typing keys. We now describe how PKCS#11 key templates are converted to key types. Key templates represent the 'types' of the keys stored in the devices. Attributes describe how keys are supposed to be used and which security properties the device enforces on them.

First we notice that attribute sensitive (S) indicates that the key should be regarded as secret. If, additionally, always-sensitive (A) is set we know that the key is trusted. In fact, the always-sensitive PKCS#11 attribute cannot be set by a user when generating or unwrapping a key (see [18], Table 15 footnotes 4 and 6). This attribute is meant to be automatically managed by the tamper resistant token whenever a key is generated as sensitive. Data and wrapping keys are instead determined by attributes E,D and W,U, respectively. We require that these pairs of attributes cannot coexists on data and wrapping keys, so to disambiguate key roles. Trusted keys that are neither wrapping not data keys are considered seeds while sensitive keys with mixed roles, e.g., E plus W, are given type Any.

We let $data(T)$ be $E \in T \vee D \in T$ and $wrap(T)$ be $W \in T \vee U \in T$. Types for keys are derived through the judgment $\vdash T : \tau$ formalized in Table 2. It is easy to see that any possible template is associated to exactly one type: non-sensitive keys are typed as Un; sensitive but not always-sensitive keys are typed Data if they have at least E or D set, and Any otherwise; always-sensitive keys are typed TData if they have E or D set, otherwise they are typed Wrap or Seed depending on the presence of W and U. Notice that no wrapping untrusted keys are allowed, in fact secret non-data keys are typed as Any.

The following lemma states that subtyping does not compromise the security of keys: non-sensitive keys can be regarded as sensitive and always-sensitive keys can be regarded as just sensitive ones. Intuitively, it is safe to increase the level of secrecy and decrease the level of trust.

Lemma 1 (Subtyping preserves security). *Let $\vdash T : \rho$ and $\vdash T' : \rho'$ with $\rho \leq \rho'$. Then $S \in T$ implies $S \in T'$ and $A \in T'$ implies $A \in T$.*

Proof. $S \in T$ implies that $\rho \neq \mathsf{Un}$ meaning that $\rho' \neq \mathsf{Un}$. Since Un is the only type for non-sensitive templates we have the thesis. Let $A \in T'$. We have $\rho' \in \{\mathsf{Wrap}, \mathsf{TData}, \mathsf{Seed}\}$ which implies $\rho \in \{\mathsf{Wrap}, \mathsf{TData}, \mathsf{Seed}\}$ giving the thesis.

Security policy. As we have already discussed in the introduction, PKCS#11 security tokens present different flaws, it is thus very important to fix them by imposing some extra security policies on them. In [4] it has been observed that real devices often limit the possible templates of keys, in order to have more control on their usage. It is possible that different operations such as key generation and key import restrict templates in different ways. At the level of static analysis, we abstract away the exact point where restrictions happen, and we consider $\mathbb{T}$ the set of all possible templates of keys.

Another very important aspect is to be clear about which keys are wrapped and unwrapped as the standards do not add any information about the template when encrypting a key with another one (one solution to this is, in fact, to add wrapping formats [9], solution which is however out of the standard). Types are useful here, as we can just establish a default type transported by wrapping keys. As we will see, thus this is limiting, it is however possible to rise the number of transported types via key diversification.

A security policy is thus defined as a pair $(\mathbb{T}, \rho)$, where $\mathbb{T}$ is the set of all possible templates of keys, and ρ is the default type for wrapped keys.

Expressions. In order to type expressions and commands we introduce a typing environment $\Gamma : x \mapsto \tau$ which maps variables to their respective types. Type judgment for expressions is noted $\Gamma \vdash_\rho e : \tau$ meaning that expression e is of type τ under Γ and assuming ρ as the default type for wrapped keys.

$$[var]\ \frac{\Gamma(x)=\tau}{\Gamma\vdash_\rho x:\tau}\quad [sub]\ \frac{\Gamma\vdash_\rho e:\tau' \quad \tau'\le\tau}{\Gamma\vdash_\rho e:\tau}\quad [kdf\text{-}w]\ \frac{\Gamma\vdash_\rho x:\mathsf{Seed}}{\Gamma\vdash_\rho \mathsf{kdf}(\mathsf{w}_{\rho'},x):\mathsf{Wrap}[\rho']}$$

$$[kdf\text{-}d]\ \frac{\Gamma\vdash_\rho x:\mathsf{Seed}}{\Gamma\vdash_\rho \mathsf{kdf}(\mathsf{d},x):\mathsf{Data}}\quad [kdf\text{-}un]\ \frac{\Gamma\vdash_\rho x:\mathsf{Un}\quad v=\mathsf{w}_{\rho'},\mathsf{d}}{\Gamma\vdash_\rho \mathsf{kdf}(v,x):\mathsf{Un}}$$

$$[enc]\ \frac{\Gamma\vdash_\rho x:\mathsf{Data}\quad \Gamma\vdash_\rho e:\mathsf{Un}}{\Gamma\vdash_\rho \mathsf{enc}(e,x):\mathsf{Un}}\quad [dec]\ \frac{\Gamma\vdash_\rho x:\mathsf{Data}\quad \Gamma\vdash_\rho e:\mathsf{Un}}{\Gamma\vdash_\rho \mathsf{dec}(e,x):\mathsf{Un}}$$

$$[wrap]\ \frac{\Gamma\vdash_\rho x:\mathsf{Wrap}\quad \Gamma\vdash_\rho e:\rho}{\Gamma\vdash_\rho \mathsf{enc}(e,x):\mathsf{Un}}\quad [unwrap]\ \frac{\Gamma\vdash_\rho x:\mathsf{Wrap}\quad \Gamma\vdash_\rho e:\mathsf{Un}}{\Gamma\vdash_\rho \mathsf{dec}(e,x):\rho}$$

$$[wrap\text{-}div]\ \frac{\Gamma\vdash_\rho x:\mathsf{Wrap}[\rho']\quad \Gamma\vdash_\rho e:\rho'}{\Gamma\vdash_\rho \mathsf{enc}(e,x):\mathsf{Un}}\quad [unwrap\text{-}div]\ \frac{\Gamma\vdash_\rho x:\mathsf{Wrap}[\rho']\quad \Gamma\vdash_\rho e:\mathsf{Un}}{\Gamma\vdash_\rho \mathsf{dec}(e,x):\rho'}$$

$$[enc\text{-}any]\ \frac{\Gamma\vdash_\rho x:\mathsf{Any}\quad \Gamma\vdash_\rho e:\mathsf{Un}\quad \rho\neq\mathsf{Wrap}}{\Gamma\vdash_\rho \mathsf{enc}(e,x):\mathsf{Un}}\quad [dec\text{-}any]\ \frac{\Gamma\vdash_\rho x:\mathsf{Any}\quad \Gamma\vdash_\rho e:\mathsf{Un}}{\Gamma\vdash_\rho \mathsf{dec}(e,x):\mathsf{Any}}$$

Table 3. Typing expressions

Typing rules are reported in Table 3. Rules [*var*] and [*sub*] are standard and derives types directly from Γ (for variables) or via subtyping. Rules [*kdf-w*] and [*kdf-d*] state that given a seed x we can derive a new wrapping key of type $\mathsf{Wrap}[\rho']$ as $\mathsf{kdf}(\mathsf{w}_{\rho'}, x)$, and a new data key as $\mathsf{kdf}(\mathsf{d}, x)$. Notice that we use values $\mathsf{w}_{\rho'}$ and d as tags to diversify keys, we can thus consider them as constant values established a-priori to this purpose. We do not assume any secrecy on them: security of this operation is given by the trusted seed x. Rule [*kdf-un*] allows for diversification from untrusted seeds, always generating an untrusted key. Rules [*enc*] and [*dec*] are for data encryption and decryption, and only work on untrusted values. Rules [*wrap*] and [*unwrap*] are more interesting: given a wrapping key we can wrap/unwrap other keys of type ρ, the default wrapping type specified in the security policy. Rules [*wrap-div*] and [*unwrap-div*] are similar but work on type ρ' given by the above rule [*kdf-w*]: diversification is in fact useful to obtain keys that can wrap keys of various types, as we will see in the case studies of section 4. Finally, rules [*enc-any*] and [*dec-any*] are conservative rules for cryptographic operation using generic keys of type Any. The former states that it is safe to encrypt with such keys as far as the default import type is not Wrap, otherwise we would be able to encrypt a broken key and then unwrap/import it as trusted in the device. The latter allows for decryption if the resulting value is considered of type Any. In section 4 we will see an example of application of these extremely conservative rules.

APIs. We now type APIs via the judgment $\Gamma\vdash_{\mathbb{T},\rho} \mathsf{c}$ meaning that c is well-typed under Γ and the policy $\mathbb{T},\rho$. The judgment is formalized in Table 4. Rules [*assign*] and [*seq*] are standard, and they amount to recursively type the expression and

$$[API]\ \frac{\forall \mathsf{a} \in \mathcal{A} \quad \Gamma \vdash_{\mathbb{T},\rho} \mathsf{a}}{\Gamma \vdash_{\mathbb{T},\rho} \mathcal{A}} \quad [assign]\ \frac{\Gamma(x) = \tau \quad \Gamma \vdash_\rho e : \tau}{\Gamma \vdash_{\mathbb{T},\rho} x := e} \quad [seq]\ \frac{\Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}_1 \quad \Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}_2}{\Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}_1; \mathsf{c}_2}$$

$$[getobj]\ \frac{\Gamma(x) = \mathsf{Any} \quad \Gamma \vdash_\rho y : \mathsf{Un}}{\Gamma \vdash_{\mathbb{T},\rho} x := \mathsf{getObj}(y)} \quad [checktmp]\ \frac{\Gamma(x) = \mathsf{LUB}(T, \mathbb{T}) \quad \Gamma \vdash_\rho y : \mathsf{Un}}{\Gamma \vdash_{\mathbb{T},\rho} x := \mathsf{checkTemplate}(y, T)}$$

$$[genkey]\ \frac{\Gamma(x) = \mathsf{Un} \quad T \in \mathbb{T}}{\Gamma \vdash_{\mathbb{T},\rho} x := \mathsf{genKey}(T)} \quad [impkey]\ \frac{\Gamma(x) = \mathsf{Un} \quad \vdash T : \tau \quad \Gamma \vdash_\rho y : \tau \quad T \in \mathbb{T}}{\Gamma \vdash_{\mathbb{T},\rho} x := \mathsf{importKey}(y, T)}$$

$$[return]\ \frac{\Gamma \vdash_\rho e : \mathsf{Un}}{\Gamma \vdash_{\mathbb{T},\rho} \mathsf{return}\ e} \quad [function]\ \frac{\Gamma \vdash_\rho x_1 : \mathsf{Un} \quad \dots \quad \Gamma \vdash_\rho x_k : \mathsf{Un} \quad \Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}}{\Gamma \vdash_{\mathbb{T},\rho} \lambda x_1, \dots, x_k.\mathsf{c}}$$

Table 4. Typing APIs

the sequential sub-part of a program, respectively. Rule [*getobj*] states that when getting a key from the token with no template check, we need to be conservative and assign the result to a variable of type Any. In fact, we cannot deduce any specific usage or security level for the obtained key; rule [*checktmp*], instead, approximates the type of the obtained key by getting the least upper bound of all types for templates T' matching T, i.e., such that $T \subseteq T'$:

$$\mathsf{LUB}(T, \mathbb{T}) \ = \ \bigsqcup \{\tau' \mid \exists T' \in \mathbb{T}. T \subseteq T' \wedge \vdash T' : \tau'\}$$

Rule [*genkey*] checks that the template for the new key is in the set of the admitted template $\mathbb{T}$, while [*impkey*] additionally checks that the type of the imported value is consistent with the given template. Rules [*return*] and [*function*] state that the return value and the parameter of an API call must be untrusted. In fact they are the interface to the external, possibly malicious users. Finally, by rule [API] we have that an API is well-typed if all of its functions are well-typed.

3.1 Type soundness

We give a notion of value well-formedness in order to track the value integrity at run-time. The judgment is based on a mapping $\Theta : val \mapsto \rho$ from atomic values to types, excluding $\mathsf{Wrap}[\rho]$ that is derived for diversified non-atomic keys. Tags $\mathsf{w}_{\rho'}$ and d for key diversification are implicitly assumed to be untrusted, i.e., $\Theta(\mathsf{w}_{\rho'}) = \Theta(\mathsf{d}) = \mathsf{Un}$. Rules are given in Table 5 and follow very closely the ones of Table 3 used for expressions.

Definition 3 (Well-formedness). $\Gamma, \Theta \vdash_{\mathbb{T},\rho} \mathsf{M}, \mathsf{H}$ *if*

- $\Gamma, \Theta \vdash_{\mathbb{T},\rho} \mathsf{M}$*, i.e.,* $\mathsf{M}(x) = v$*,* $\Gamma(x) = \tau$ *implies* $\Theta \vdash_\rho v : \tau$*,*
- $\Theta \vdash_{\mathbb{T},\rho} \mathsf{H}$*, i.e.,* $\mathsf{H}(v') = (v, T)$*,* $\vdash T : \tau$ *implies* $\Theta \vdash_\rho v : \tau$ *and* $T \subseteq \mathbb{T}$

We now prove that if we only give the attacker untrusted atomic values, all the values he will be able to derive (according to section 2) will also be untrusted. Intuitively, having type Un is a necessary condition for a well-formed value to be deducible by the attacker. The following holds:

$$[atom]\ \frac{\Theta(val)=\rho'}{\Theta\vdash_\rho val:\rho'}\quad [sub]\ \frac{\Theta\vdash_\rho v:\tau'\quad \tau'\le\tau}{\Theta\vdash_\rho v:\tau}\quad [kdf\text{-}w]\ \frac{\Theta\vdash_\rho v:\mathsf{Seed}}{\Theta\vdash_\rho kdf(\mathsf{w}_\rho,v):\mathsf{Wrap}[\rho]}$$

$$[kdf\text{-}d]\ \frac{\Theta\vdash_\rho v:\mathsf{Seed}}{\Theta\vdash_\rho kdf(\mathsf{d},v):\mathsf{Data}}\quad [kdf\text{-}un]\ \frac{\Theta\vdash_\rho v,v':\mathsf{Un}}{\Theta\vdash_\rho kdf(v',v):\mathsf{Un}}$$

$$[enc]\ \frac{\Theta\vdash_\rho v:\mathsf{Data}\quad \Theta\vdash_\rho v':\mathsf{Un}}{\Theta\vdash_\rho enc(v',v):\mathsf{Un}}\quad [dec]\ \frac{\Theta\vdash_\rho v:\mathsf{Data}\quad \Theta\vdash_\rho v':\mathsf{Un}\quad v'\ne enc(v'',v)}{\Theta\vdash_\rho dec(v',v):\mathsf{Un}}$$

$$[wrap]\ \frac{\Theta\vdash_\rho v:\mathsf{Wrap}\quad \Theta\vdash_\rho v':\rho}{\Theta\vdash_\rho enc(v',v):\mathsf{Un}}\quad [unwrap]\ \frac{\begin{array}{c}v'\ne enc(v'',v)\\ \Theta\vdash_\rho v:\mathsf{Wrap}\quad \Theta\vdash_\rho v':\mathsf{Un}\end{array}}{\Theta\vdash_\rho dec(v',v):\rho}$$

$$[wrap\text{-}div]\ \frac{\Theta\vdash_\rho v:\mathsf{Wrap}[\rho']\quad \Theta\vdash_\rho v':\rho'}{\Theta\vdash_\rho enc(v',v):\mathsf{Un}}\quad [unwrap\text{-}div]\ \frac{\begin{array}{c}v'\ne enc(v'',v)\\ \Theta\vdash_\rho v:\mathsf{Wrap}[\rho']\quad \Theta\vdash_\rho v':\mathsf{Un}\end{array}}{\Theta\vdash_\rho dec(v',v):\rho'}$$

$$[enc\text{-}any]\ \frac{\Theta\vdash_\rho v:\mathsf{Any}\quad \Theta\vdash_\rho v':\mathsf{Un}\quad \rho\ne\mathsf{Wrap}}{\Theta\vdash_\rho enc(v',v):\mathsf{Un}}\quad [dec\text{-}any]\ \frac{\begin{array}{c}v'\ne enc(v'',v)\\ \Theta\vdash_\rho v:\mathsf{Any}\quad \Theta\vdash_\rho v':\mathsf{Un}\end{array}}{\Theta\vdash_\rho dec(v',v):\mathsf{Any}}$$

Table 5. Value well-formedness

Proposition 1. *Let $\Theta\vdash_{\mathbb{T},\rho}\mathsf{H}$ and let V be a set of atomic values such that $val\in V$ implies $\Theta(val)=\mathsf{Un}$. Then, $v\in\mathcal{K}(V)$ implies $\Theta\vdash_\rho v:\mathsf{Un}$.*

Proof. By an easy induction on the length of the derivation of values in $\mathcal{K}(V)$. For length 0 we trivially have that $v\in V$ which gives the thesis. We assume the proposition holds for length i and we prove it for length $i+1$. We show case $enc(v,v')\in\mathcal{K}(V)$ because of $v,v'\in\mathcal{K}(V)$. The other cases are analogous. By rule $[enc]$ and observing that $\mathsf{Un}\le\mathsf{Data}$ we obtain the thesis.

Next lemma proves that we can never type a value with two types that are not related via subtyping. As a consequence, we have that trusted values can never be typed as untrusted and vice-versa.

Lemma 2. *$\Theta\vdash_\rho v:\tau$ and $\Theta\vdash_\rho v:\tau'$ implies $\tau\le\tau'$ or $\tau'\le\tau$.*

Proof. By an easy induction on the (sum of the) length of the derivations of $\Theta\vdash_\rho v:\tau$ and $\Theta\vdash_\rho v:\tau'$. Base case is length 0 and trivially gives $\tau=\tau'=\Theta(v)$. We show once case of the inductive step. Suppose $v=kdf(v',v'')$. We have three different rules for typing v: $[kdf\text{-}w]$,$[kdf\text{-}d]$,$[kdf\text{-}un]$. For example, types Data and $\mathsf{Wrap}[\tau]$ given by the first two rules are unrelated, however the typed values differ for a tag which excludes the case. More interestingly, Un and $\mathsf{Wrap}[\tau]$ are also unrelated but, by induction, we know that the key v should be typed with two related types, which is not the case since Seed and Un are not in the subtyping relation. Other cases follow similarly.

This last lemma states that evaluating an expression of type τ on a well-formed memory, gives a value of type τ.

Lemma 3. *Let $\Gamma \vdash_\rho e : \tau$ and $e \downarrow^{\mathsf{M}} v$. If $\Gamma, \Theta \vdash_{\mathbb{T},\rho} \mathsf{M}$ then it holds $\Theta \vdash_\rho v : \tau$.*

Proof. By induction on the structure of e. Base case is when e is x. Thesis directly follows by memory well-formedness. For the inductive step, in some cases we use Lemma 2: for example, when dealing with decryption [*dec*] we might have a value encrypted under the right key that we know to type Un. Now looking at the possible cases in Table 5 we see that the encrypted values can be obtained in different ways, but the information about the type of the key (Data) allows us to pick either [*enc*] or [*enc-any*], both of which prove the plaintext to be of type Un as required. Other cases follow analogously.

We now give a subject-reduction result stating that well-typed programs remain well-typed at run-time and preserve memory and handle-map well-formedness.

Theorem 1. *Let $\Gamma, \Theta \vdash_{\mathbb{T},\rho} \mathsf{M}, \mathsf{H}$ and $\Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}$. If $\langle \mathsf{M}, \mathsf{H}, \mathsf{c}\rangle \to \langle \mathsf{M}', \mathsf{H}', \mathsf{c}'\rangle$ then*

(i) if $\mathsf{c}' \neq \varepsilon$ then $\Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}'$;
(ii) $\exists \Theta' \supseteq \Theta$ such that $\Gamma, \Theta' \vdash_{\mathbb{T},\rho} \mathsf{M}', \mathsf{H}'$.

Proof. (Sketch.) Proof of item (i) is by trivial induction on the structure of c. In fact almost all commands reduce to ε. Item (ii) is again by induction on the structure of c: for expressions we just apply Lemma 3. For genKey and $\mathsf{importKey}$ the returned handle and the new key are fresh names that we add to Θ in order to type the new memory (this is why we have Θ' in the thesis). Template T is checked to be compatible with respect to $\mathbb{T}$, and the type of the imported key value is checked to be the same as the one derived from the template. getObj assigns to type Any so there is nothing to prove, while $\mathsf{checkTemplate}$ approximates the type of the key using a least upper bound which guarantees that the value can be typed the same as the variable x via subtyping.

We can now state the main result of the paper: well-typed APIs are secure, according to definition 2.

Theorem 2. *Let $\Gamma \vdash_{\mathbb{T},\rho} \mathcal{A}$. Then $\mathcal{A}$ is secure.*

Proof. We first prove, by induction on the length of reduction $\langle \mathsf{H}_0, V_0\rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}, V\rangle$, that there exists Θ such that $\Theta \vdash_{\mathbb{T},\rho} \mathsf{H}$ and $\Theta \vdash_\rho v : \mathsf{Un}$ for each $v \in V$.

Base case is length 0, meaning that $\mathsf{H}_0 = \mathsf{H}$ and $V_0 = V$. If we take Θ such that $\Theta(v) = \mathsf{Un}$ for each $v \in V_0$, since H is empty, we easily obtain the thesis.

For the inductive case we have $\langle \mathsf{H}_0, V_0\rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}_n, V_n\rangle \rightsquigarrow_{\mathcal{A}} \langle \mathsf{H}, V\rangle$. By inductive hypothesis there exists Θ such that $\Theta \vdash_{\mathbb{T},\rho} \mathsf{H}_n$ and $\Theta(v) = \mathsf{Un}$ for each $v \in V_n$. We consider the last step $\langle \mathsf{H}_n, V_n\rangle \rightsquigarrow_{\mathcal{A}} \langle \mathsf{H}, V\rangle$. By definition, this is due to a call to a function $\mathsf{a} \in \mathcal{A}$. In particular, we have $\mathsf{a}(v_1, \ldots, v_k) \downarrow^{\mathsf{H}_n,\mathsf{H}} v$ with $v_1, \ldots, v_k \in \mathcal{K}(V)$ and $V = V_n \cup \{v\}$. This, in turns, happens because $\mathsf{a} = \lambda x_1, \ldots, x_k.\mathsf{c}$ and $\langle \mathsf{M}_\epsilon[x_1 \mapsto v_1 \ldots x_k \mapsto v_k], \mathsf{H}_n, \mathsf{c}\rangle \to \langle \mathsf{M}', \mathsf{H}, \mathsf{return}\ e\rangle$ with $e \downarrow^{\mathsf{M}'} v$. From $\Gamma \vdash_{\mathbb{T},\rho} \mathcal{A}$ we have $\Gamma \vdash_{\mathbb{T},\rho} \mathsf{a}$ which requires $\Gamma \vdash_\rho x_1 : \mathsf{Un} \ \ \ldots \ \ \Gamma \vdash_\rho x_k : \mathsf{Un}$ and $\Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}$. Since $x_1, \ldots, x_k$ are the only variables in the domain of $M_0 = \mathsf{M}_\epsilon[x_1 \mapsto v_1 \ldots x_k \mapsto v_k]$, we easily obtain that $\Gamma, \Theta \vdash_{\mathbb{T},\rho} \mathsf{M}_0$. We have proved that $\Gamma, \Theta \vdash_{\mathbb{T},\rho} \mathsf{M}_0, \mathsf{H}$ and $\Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}$, thus by Theorem 1 we obtain $\Gamma \vdash_{\mathbb{T},\rho} \mathsf{return}\ e$ and $\exists \Theta' \supseteq \Theta$ such that $\Gamma, \Theta' \vdash_{\mathbb{T},\rho} \mathsf{M}', \mathsf{H}$. Now, $\Gamma \vdash_{\mathbb{T},\rho} \mathsf{return}\ e$ requires $\Gamma \vdash_\rho e : \mathsf{Un}$, by Lemma 3 we have $\Theta' \vdash_\rho v : \mathsf{Un}$ which gives the thesis.

We have proved that there exists Θ such that $\Theta \vdash_{\mathbb{T},\rho} \mathsf{H}$ and $\Theta \vdash_\rho v : \mathsf{Un}$ for each $v \in V$. For item 1, if $\Theta \vdash_\rho \mathit{val} : \mathsf{Un}$, meaning that $\Theta(\mathit{val}) = \mathsf{Un}$, since we have $\mathit{val} \notin \mathcal{K}(V)$ and val is sensitive, we can change Θ into $\tilde{\Theta} = \Theta[\mathit{val} \mapsto \mathsf{Data}]$ while preserving $\tilde{\Theta} \vdash_{\mathbb{T},\rho} \mathsf{H}$ and $\tilde{\Theta} \vdash_\rho v : \mathsf{Un}$ for each $v \in V$. In fact, if S appears in all the templates for value val and val is different from all values in V, we have that its type is never required to be Un, since none of the templates will be typed as Un. Notice that $\tilde{\Theta}(\mathit{val}) = \mathsf{Data}$ implies that $\tilde{\Theta} \nvdash_\rho \mathit{val} : \mathsf{Un}$. We consider now $\langle \mathsf{H}, V\rangle \leadsto^*_{\mathcal{A}} \langle \mathsf{H}', V'\rangle$. By following the same proof scheme as above, we can prove that $\Theta' \vdash_{\mathbb{T},\rho} \mathsf{H}'$ and $\Theta' \vdash_\rho v : \mathsf{Un}$ for each $v \in V'$ with $\Theta' \supseteq \tilde{\Theta}$. Thus, $\Theta'(\mathit{val}) = \mathsf{Data}$ meaning that $\Theta' \nvdash_\rho \mathit{val} : \mathsf{Un}$. From Proposition 1 we obtain that $\mathit{val} \notin \mathcal{K}(V')$ which gives the thesis.

For item 2, we have that that all templates of val are typed with one of $\mathsf{Wrap}, \mathsf{TData}, \mathsf{Seed}$. This, by lemma 2, implies $\Theta \nvdash_\rho \mathit{val} : \mathsf{Un}$ which by Proposition 1 gives $\mathit{val} \notin \mathcal{K}(V)$. We can now apply item 1 to obtain $\mathit{val} \notin \mathcal{K}(V')$. Recall, in fact, we have assumed that $A \in T$ implies $S \in T$.

4 Type-based analysis

In this section we consider different implementations of (a subset of) PKCS#11 API and we analyse them using our type-based approach. We only consider the functions for encryption/decryption of data and wrap/unwrap of keys.

RSA PKCS#11 Standard. We show that an implementation of PKCS#11 that exactly follows the standard, fails to type-check, as expected, since it is known to be vulnerable to attacks. This is useful to show how these attacks can be prevented by statically requiring a precise unambiguous role for each key, as done by our type system.

The API is defined in the RSA standard, which specifies what are the input parameters and the result of each function. `C_Encrypt` takes a byte-stream and a handle to a key having the encrypt (E) flag set, and returns an encrypted byte-stream. Similarly `C_Decrypt` takes a byte-stream and decrypts it using the key pointed by the given handle, with the decrypt (D) flag set; it then returns to the user the decrypted message:

```
C_Encrypt($data$, $h\_key$)                      C_Decrypt($data$, $h\_key$)
  $k :=$ checkTemplate($h\_key$, $\{E\}$)          $k :=$ checkTemplate($h\_key$, $\{D\}$);
  return enc($data, k$);                           return dec($data, k$);
```

`C_WrapKey` takes the handle of a key to be wrapped and the one pointing to the wrapping key, having the wrap (W) flag set, and returns an encrypted byte-stream. The unwrap command (`C_UnwrapKey`) reads a byte-stream, decrypts it using a key having the unwrap (U) flag set, imports the resulting key in the device and returns a handle to it. The standard allows the user to specify the template for the new key. In this example, we assume the key is imported as sensitive (S).

```
C_WrapKey($h\_key$, $h\_w$)
  $w := \mathsf{checkTemplate}(h\_w,\ \{W\})$
  $k := \mathsf{getObj}(h\_key)$;
  return $\mathsf{enc}(k, w)$;

C_UnwrapKey($data$, $h\_w$)
  $w := \mathsf{checkTemplate}(h\_w,\ \{U\})$
  $k := \mathsf{dec}(data, w)$;
  return $\mathsf{importKey}(k, \{S\})$;
```

The standard does not impose any rule on the usage of encrypt, decrypt, wrap and unwrap attributes. Thus the policy is the most permissive one, i.e., $\mathbb{T}$ is the set of all the possible templates T. In section 1 we have seen an attack that exploits `C_Decrypt` and `C_WrapKey`. We now show that the latter does not type-check, confirming that we cannot prove the security of the API. Command $\mathsf{return\ enc}(k, w)$ requires $\Gamma \vdash_\rho \mathsf{return\ enc}(k, w) : \mathsf{Un}$. Command $k := \mathsf{getObj}(h\_key)$ requires that $\Gamma \vdash_\rho k : \mathsf{Any}$. Typing $w := \mathsf{checkTemplate}(h\_w,\ \{W\})$ requires w to have type $\mathsf{LUB}(\{W\}, \mathbb{T}) = \mathsf{Any}$ since the permissive policy allows for templates with mixed roles such as $\{S, E, D, W, U\}$. Since there is no rule for typing expressions of type Any with key of type Any we can never obtain $\Gamma \vdash_\rho \mathsf{return\ enc}(k, w) : \mathsf{Un}$, giving a contradiction.

Secure Templates. We now analyse and prove the security of a fix proposed in [4,5]. Note that, it is the first proposed patch that does not require the addition of any cryptographic mechanisms to the standard. The idea is to limit the set of admissible attribute combinations for keys in order to avoid that they ever assume conflicting roles at creation time. This is configurable at the level of the specific PKCS#11 operation. For example, different secure templates can be defined for different operations such as key generation and unwrapping.

More precisely, the fix includes three templates for the key generation command: a wrap and unwrap one for importing/exporting other keys, here mapped into $\{A, S, W, U\}$ with type Wrap; an encrypt and decrypt template for cryptographic operations, here encoded as $\{S, E, D\}$ with type Data and an empty template, corresponding to $\{\}$, i.e., Un. The unwrap command is instead allowed to set either an empty template or one which has the unwrap and encrypt attributes set and the wrap and decrypt ones unset. This is a mixed-role template that corresponds to type Any that we pick as the default unwrapping type ρ.

We use the policy $\mathbb{T}$ such that $T \in \mathbb{T}$ and $\{W\} \in T$ implies $T = \{A, S, W, U\}$, moreover $\{D\} \in T$ implies $T = \{S, E, D\}$, i.e., wrapping and decryption keys are respectively encoded with the unique templates $\{A, S, W, U\}$,$\{S, E, D\}$. With such a policy, whenever a checkTemplate expression queries a handle for a decryption key ($\{D\}$) then the type returned is Data, since the only matching template is $\{S, E, D\}$. When we query for an encryption key ($\{E\}$) then the type returned is Any since, for example, $\{S, E, U\} \in \mathbb{T}$. When querying for a wrapping key ($\{W\}$) the result will be typed as Wrap since the only template satisfying the query is $\{A, S, W, U\}$. Finally, when querying for an unwrapping key ($\{U\}$) the results is Any since, again, $\{S, E, U\} \in \mathbb{T}$. We now show that the standard API as defined above type-checks under the above more restrictive policy. Recall that we let $\rho = \mathsf{Any}$, i.e., the default type for wrapped key is Any.

```
C_Encrypt($data$, $h\_key$)
  $k := \mathsf{checkTemplate}(h\_key,\ \{E\})$     $(\Gamma(k) = \mathsf{Any})$
  return $\mathsf{enc}(data, k)$;                   $(\Gamma \vdash_\rho \mathsf{enc}(data, k) : \mathsf{Un})$

C_Decrypt($data$, $h\_key$)
  $k := \mathsf{checkTemplate}(h\_key,\ \{D\})$     $(\Gamma(k) = \mathsf{Data})$
  return $\mathsf{dec}(data, k)$;                   $(\Gamma \vdash_\rho \mathsf{dec}(data, k) : \mathsf{Un})$

C_WrapKey($h\_key, h\_w$)
  $w := \mathsf{checkTemplate}(h\_w,\ \{W\})$       $(\Gamma(w) = \mathsf{Wrap})$
  $k := \mathsf{getObj}(h\_key)$;                   $(\Gamma(k) = \mathsf{Any})$
  return $\mathsf{enc}(k, w)$;                      $(\Gamma \vdash_\rho \mathsf{enc}(k, w) : \mathsf{Un})$

C_UnwrapKey($data$, $h\_w$)
  $w := \mathsf{checkTemplate}(h\_w,\ \{U\})$       $(\Gamma(w) = \mathsf{Any})$
  $k := \mathsf{dec}(data, w)$;                     $(\Gamma(k) = \mathsf{Any})$
  return $\mathsf{importKey}(k, \{S, E, U\})$;      $(\Gamma \vdash_\rho \mathsf{importKey}(k, \{S, E, U\}) : \mathsf{Un})$
```

By theorem 2 we have that this fix is secure and never leaks sensitive and always-sensitive keys. It strongly limits, however, the set of possible templates, and this could be an issue if an application in use on a given system fails to obey such requirements. On the other hand, compatibility with other devices is not broken, since the implementation of the above functions is the same as in the standard. However, even if interoperability is guaranteed, the usage of an unsafe token would obviously expose the keys to attacks.

Finally, notice that the patch is presented here in an extended version: originally it allowed the generation of sensitive keys only, we instead let non-sensitive keys to be accepted by the policy.

Key Diversification. We present a novel fix to PKCS#11. The idea is to use key diversification to avoid the same key to be used for conflicting purposes. This ensures that the same key will never be used for encrypting and decrypting both data and other keys. The fix is completely transparent to the user as far as all the devices implement it. It must be noted, in fact, that a key wrapped by a token implementing this patch cannot be correctly imported by one acting as described by the standard, i.e., not using key diversification (and vice versa). The same holds for encrypted data. To the best of our knowledge, this is the only patch that correctly enforces the security of sensitive keys and, at the same time, is transparent to existing applications.

We define a policy that allows for templates typed as Seed, Any, Data, Un. Formally $\mathbb{T} = \{T \mid\ \vdash T : \rho \text{ and } \rho \in \{\mathsf{Seed}, \mathsf{Any}, \mathsf{Data}, \mathsf{Un}\}\ \}$. We now specify the fixed functions and the typing for each variable/expression.

```
C_Encrypt($data$, $h\_key$)
  $k := \mathsf{checkTemplate}(h\_key,\ \{A, S\})$  $(\Gamma(k) = \mathsf{Seed})$
  $dk := \mathsf{kdf}(\mathsf{d}, k)$;              $(\Gamma(dk) = \mathsf{Data})$
  return $\mathsf{enc}(data, dk)$;                 $(\Gamma \vdash_\rho \mathsf{enc}(data, dk) : \mathsf{Un})$
```

```
C_Decrypt$(data,\ h\_key)$
  $k :=$ checkTemplate$(h\_key,\ \{A, S\})$ $(\Gamma(k) = \mathsf{Seed})$
  $dk :=$ kdf$(\mathsf{d}, k)$;                 $(\Gamma(dk) = \mathsf{Data})$
  return dec$(data, dk)$;              $(\Gamma \vdash_\rho \mathsf{dec}(data, dk) : \mathsf{Un})$
```

Notice, in particular, that $\Gamma \vdash_\rho$ checkTemplate$(h_key,\ \{A, S\})$: Seed since $\mathsf{LUB}(\{A, S\}, \mathbb{T}) = \mathsf{Seed}$. In fact, Seed is the only type in T with A set (we have excluded from the policy Wrap and TData).

Key diversification allows to choose at run-time the wrapping and unwrapping of different kind of keys: different instances of each command will be provided, each of them using a different tag when diversifying the seed retrieved from the device. Since the code is exactly the same, we just parametrize it on the tag value w_ρ. With $T_{\rho'}$ we identify a template such that $\mathsf{LUB}(T_{\rho'}, \mathbb{T}) = \rho'$. For $\rho' = \mathsf{Seed}, \mathsf{Any}, \mathsf{Data}$ we respectively have $T_{\rho'} = \{A, S\}, \{S\}, \{S, E, D\}$. Wrap and unwrap are specified an typed as follows:

```
C_WrapKey$^{\mathsf{w}_{\rho'}}(h\_key,\ h\_w)$
  $w :=$ checkTemplate$(h\_w,\ \{A, S\})$ $(\Gamma(w) = \mathsf{Seed})$
  $k :=$ checkTemplate$(h\_w,\ T_{\rho'})$     $(\Gamma(k) = \rho')$
  $dk :=$ kdf$(\mathsf{w}_{\rho'}, w)$;               $(\Gamma(dk) = \mathsf{Wrap}[\rho'])$
  return enc$(k, dk)$;                 $(\Gamma \vdash_\rho \mathsf{enc}(k, dk) : \mathsf{Un})$

C_UnwrapKey$^{\mathsf{w}_{\rho'}}(data,\ h\_w)$
  $w :=$ checkTemplate$(h\_w,\ \{A, S\})$ $(\Gamma(w) = \mathsf{Seed})$
  $dk :=$ kdf$(\mathsf{w}_{\rho'}, w)$;               $(\Gamma(dk) = \mathsf{Wrap}[\rho'])$
  $k :=$ dec$(data, dk)$;                $(\Gamma(k) = \rho')$
  return importKey$(k, T_{\rho'})$;          $(\Gamma \vdash_\rho \mathsf{importKey}(k, T_{\rho'}) : \mathsf{Un})$
```

Since the API type-checks, by theorem 2 we have that it is secure and never leaks sensitive and always-sensitive keys. Notice that, since it is possible to exchange seeds we have that new wrapping keys can be easily shared between users. Notice also that, in practice, the parameter w_ρ needs to be somehow fixed, in order to have a single implementation of wrap and unwrap commands. The way this value is picked is not relevant, since we prove that all these instances are secure even if they coexist on the device. For example, it might be derived at run-time from the `CKA_UNWRAP_TEMPLATE` attribute which specifies, for each wrapping key, the template to be assigned to the unwrapped key.

5 Conclusions

We have presented a type system to statically enforce the security of PKCS#11 key management APIs. We believe that a formal tool working at the language-level might help developers and hardware producers to better understand the crucial issues and limits affecting the design and implementation of this standard. For example, we have shown that `C_Decrypt` and `C_WrapKey` commands cannot be both type-checked if implemented as prescribed by the standard [18]. More precisely, it has been shown that the requirements on the templates of the keys used to perform such operations are not enough restrictive to avoid keys having conflicting purposes. Thus, failing to type-check corresponds, in this case, to the intuitive problematic issue, well understood by developers and hardware producers, of conflicting roles assigned to a single key.

We have also presented a new fix to PKCS#11, based on key diversification: Intuitively, the token avoids conflicting roles for one key by diversifying it depending on the actual role. We have type-checked both this new fix and the 'secure templates' one [4,5], formally proving their security.

Starting from version 2.20, RSA added to the standard the new attribute `CKA_WRAP_WITH_TRUSTED`, that could potentially be used to prevent the API-level attacks discussed in this work. However, a big limitation is that trusted keys, i.e., keys whose `CKA_TRUSTED` attribute is set, may be imported into a token only by a security officer, a special privileged user operating in a protected environment. Moreover, in order to prevent attacks on a sensitive key, it is required that its `CKA_WRAP_WITH_TRUSTED` attribute is set, meaning that it can only be wrapped under a key imported by the security officer. Here we have generalized this idea of wrapping keys only under trusted keys. We have used the always-sensitive attribute, even if the standard does not foresee any special usage for it, in order to show that what is important is 'trust', and not who has imported the key: a key that has always been sensitive (and has never been known by the attacker) can be considered trusted the same as one imported by the security officer. So, intuitively, in our model the always-sensitive and trusted attributes collapse into the A attribute. This allows for dynamically exchanging new always-sensitive, trusted keys, wrapped under the one initially imported by the security officer.

Quite surprisingly, in [18] RSA does not discuss any security implication of the two new attributes and does not provide any guideline about how to correctly use them to prevent attacks (in fact, attacks are not mentioned even in the most recent draft of the standard [19]). There are, instead, many problematic issues that need to be considered. We give a partial list here: (*i*) trusted keys should be non-extractable, i.e., not wrappable even under another trusted key. This is to avoid they are unwrapped with a different template and then leaked; (*ii*) a sensitive key with `CKA_WRAP_WITH_TRUSTED` set might be wrapped under a trusted key and then unwrapped with `CKA_WRAP_WITH_TRUSTED` unset, making it attackable; (*iii*) trusted keys should not have conflicting roles (such as wrap and decrypt). While this might be obvious, it is not a good idea to leave the security officer the freedom of freely configuring such crucial keys. Our type-based analysis solves all the above issues by enforcing a controlled usage of roles and templates for keys.

The extension to public-key cryptography and the implementation of the key diversification fix on a software emulated token are left as a future work. As already done for the secure template patch [4,5] the starting point for the implementation would be the open-source project openCryptoki [17].

Acknowledgements. We would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. R. Anderson. The correctness of crypto transaction sets. In *8th International Workshop on Security Protocols*, April 2000. `http://www.cl.cam.ac.uk/ftp/users/rja14/protocols00.pdf`.
2. M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 220–234, Paris, France, 2001. Springer.
3. M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, 34(10):67–75, October 2001.
4. M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 260–269. ACM, 2010.
5. M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. CryptokiX: a cryptographic software token with security fixes. In *Proceedings of the 4th International Workshop on Analysis of Security APIs (ASA)*, Edinburgh, UK, July 2010.
6. M. Centenaro, R. Focardi, F.L. Luccio, and G. Steel. Type-Based Analysis of PIN Processing APIs. In *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2009.
7. R. Clayton and M. Bond. Experience using a low-cost FPGA design to crack DES keys. In *Cryptographic Hardware and Embedded System - CHES 2002*, pages 579–592, 2002.
8. J. Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2003.
9. S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.
10. S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, November 2010.
11. R. Focardi and F.L. Luccio. Secure recharge of disposable RFID tickets. In *FAST*, volume 7140 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2011.
12. S.B. Fröschle and N. Sommer. Reasoning with Past to Prove PKCS#11 Keys Secure. In *FAST*, volume 6561 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2010.
13. S.B. Fröschle and N. Sommer. Concepts and Proofs for Configuring PKCS#11. In *FAST*, volume 7140 of *Lecture Notes in Computer Science*. Springer, 2011.
14. S.B. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In *ARSPA-WITS*, volume 5511 of *LNCS*, pages 92–106, York, UK, 2009. Springer.
15. G. Keighren, D. Aspinall, and G. Steel. Towards a Type System for Security APIs. In *ARSPA-WITS*, pages 173–192, 2009.
16. D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.
17. openCryptoki. `http://sourceforge.net/projects/opencryptoki/`.
18. RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.
19. RSA Security Inc., Draft v2.30. *PKCS #11: Cryptographic Token Interface Standard.*, July 2009. Available at `http://www.rsa.com/rsalabs/node.asp?id=2133`.