Type-based Analysis of Key Management in PKCS#11 cryptographic devices[*]

Matteo Centenaro   Riccardo Focardi   Flaminia L. Luccio

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

June 21, 2013

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. We develop a prototype type-checker for a software token emulator written in C and we experiment on various working configurations.

1 Introduction

PKCS#11, also known as Cryptoki, defines a widely adopted API for cryptographic tokens [22]. 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 [6, 11, 13].

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

[*]This work is a revised and extended version of [8] and has been partially supported by the Miur Prin Project "*Security Horizons*".

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.

The attacks on PKCS#11 we consider in this paper are at the API level [2, 4, 5, 6, 10, 11, 13, 21], 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 (CKA_SENSITIVE), 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:

$$\begin{aligned}&h2 := \texttt{C\_GenerateKey}(\{\texttt{CKA\_SENSITIVE}, \texttt{CKA\_WRAP}, \texttt{CKA\_DECRYPT}\});\\ &wrapped := \texttt{C\_WrapKey}(h1, h2);\\ &leak := \texttt{C\_Decrypt}(wrapped, h2);\end{aligned}$$

First, we ask the token to generate a new key with attributes CKA_SENSITIVE, CKA_WRAP and CKA_DECRYPT set, referenced by the handle $h2$. Then, we use this key to wrap an existing sensitive key referenced by the handle $h1$. 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 [6] 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 [6, 13]: 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 [13, 18], or limiting very carefully the usage of imported keys to a subset of non-critical functions [6].

In our opinion, formal tools to reason about the security of different implementations of PKCS#11 APIs, such as Tookan [6], 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 [6, 7]; (vi) develop a prototype type-checker for a software token emulator written in C and we experiment on various working configurations. 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., [13]). 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 [6, 7], 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 prove that this new patch type-checks.

In order to investigate to which extent the proposed theory scales to real settings, we develop a type-checker for Opencryptoki,[1] a software emulator of a PKCS#11 device written in C. We describe our experience and we illustrate how we had to adapt the theory and instrument the code in to allow the type-checking of a real, working implementation. We experiment on various configurations and attacks. As expected, all flawed token configurations do not type-check, while for type-checked code we could not find any working attack. This prototype should be thought as a proof-of-concept that this kind of analysis might be useful in practice and could possibly be adapted to validate real device firmware.

Related work. The most established work on formal analysis of PKCS#11 is [13]. 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 [18]. In [6], 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

[1] `http://sourceforge.net/projects/opencryptoki/`

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 [9] it is studied the security of PIN managements Hardware Security Modules, and it is given a type system to prove their security; in [14] we have given a type system for the security of rechargeable disposable RFID tickets.

A recent line of research [16, 17] 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. In particular, it is shown that specific configurations preserve key security: keys can be configured as non-extractable to avoid them of being wrapped under other keys [16]. Since non-extractable keys have already been shown to be secure, in this work we implicitly assume that any key is extractable. If the device implements PKCS#11 v2.20 it is also possible to require that a key is only wrapped under security officer `CKA_TRUSTED` keys [17]. This results are important and hold for implementations that are compliant with the standard. We have shown, however, that this is not always the case for commercial devices [6] (see, in particular, attack a4).[2] Our starting point is to allow variations in how the standard is implemented, in particular for what concerns assigning attributes to keys, and devise techniques to prove that these implementations achieve the desired security goals. Our approach is 'language-based' which makes it appealing for application to the firmware of real devices, as we discuss in Section 5. Note also that v2.20 of the standard is available only on a small subset of commercial devices. For this reason we do not base our analysis on the `CKA_TRUSTED` attribute. However, our usage of `CKA_ALWAYS_SENSITIVE` (deviating a bit from the standard) is very close to `CKA_TRUSTED` and we are confident that our results would easily scale to PKCS#11 v2.20. We leave the formal comparison with [16, 17] as a future work.

In [19] 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 [19] the authors investigate *noninterference*, a much stronger property. In this sense their result is more in the line of [9]; (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 consider confidentiality and do not treat integrity (or trust) that is one of the crucial ingredients of our analysis: only trusted keys should be used to wrap sensitive keys. A formal comparison will be the subject of future work.

A recent line of research [3] shows that type-systems can be effective on real im-

[2]`http://secgroup.ext.dsi.unive.it/tookan`

plementations of cryptographic protocols. We believe that this direction is really interesting and important but in this paper we take a different approach. Instead of checking the whole implementation, we extract from the real code of a PKCS#11 software simulator written in C the relevant fragments that correspond to API specification, and we type check them against a dynamically enforced policy. While this solution is far from proving correctness of the actual implementation, as it assumes correctness of big parts of code, it is very manageable and amenable for implementation onto the firmware of real hardware tokens.

Some recent work has focused on strong information flow guarantees for general-purpose programs with cryptographic primitives [15, 20]. These techniques have been applied to a different setting, an interesting future work would be to study whether they could be applied to the problem of type-based analysis of key management APIs.

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. In Section 5 we describe the prototype implementation of a type-checker for a software PKCS#11 token, and we apply it to check various working configurations. We conclude in Section 6.

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. The language allows for performing symmetric key encryption, decryption and key diversification. It provides abstract commands to generate, retrieve and store a key and its attributes on a device. While retrieving a key it is possible to perform a check on the presence of specific attributes. There core operations are expressive enough to model interesting key management APIs.

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:

$$v \quad ::= \quad val \mid enc(v, v') \mid dec(v, v') \mid kdf(v, v')$$

where $enc(v, v')$ and $dec(v, v')$ denote value v respectively encrypted and decrypted under key v', and $kdf(v, v')$ represents a new key obtained via diversification from a value v and a 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:

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

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:

$$\begin{array}{rll}
x & \downarrow^{\mathsf{M}} M(x) & \text{if } M(x) \text{ is defined} \\
\mathsf{enc}(e, x) & \downarrow^{\mathsf{M}} enc(v, v') & \\
\mathsf{dec}(e, x) & \downarrow^{\mathsf{M}} \left\{ \begin{array}{ll} v'' & \text{if } v = enc(v'', v') \\ dec(v, v') & \text{otherwise} \end{array} \right. & \\
\mathsf{kdf}(val, x) & \downarrow^{\mathsf{M}} kdf(val, v') &
\end{array}$$

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:

$$\begin{array}{lll}
\mathsf{a} & ::= & \lambda x_1, \ldots, x_k.\mathsf{c} \\
\mathsf{c} & ::= & x := e \mid x := \mathsf{f} \mid \mathsf{return}\ e \mid \mathsf{c}_1; \mathsf{c}_2 \\
\mathsf{f} & ::= & \mathsf{getKey}(y, T) \mid \mathsf{genKey}(T) \mid \mathsf{setKey}(y, T)
\end{array}$$

We 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, getKey retrieves the plaintext value of a key stored in the device, given its handle y; 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, setKey imports a new key with plaintext value y and template T. The first function fails (i.e., is stuck) if the given handle does not exist or refers to a key with a template that is not a superset of T, i.e., a key that does not have all the attributes in T set. 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 $\mathtt{return}\ e$. Notice that the language does not have if-then-else branches since attribute check performed in getKey is enough for modelling common APIs. However there is no technical reason that prevents modelling branches in case they would be needed.

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

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 denoted by $\langle \mathsf{M}, \mathsf{H}, \mathsf{c} \rangle$. 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 setKey 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 $\mathsf{a}(v_1, \ldots, v_k) \downarrow^{\mathsf{H},\mathsf{H}'} v$ where H' is the resulting handle map. Notice that at this API level we do not observe memories 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.

$$\frac{e \downarrow^{\mathsf{M}} v}{\langle \mathsf{M}, \mathsf{H}, x := e\rangle \to \langle \mathsf{M}[x \mapsto v], \mathsf{H}, \varepsilon\rangle}$$

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

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

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

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

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

Table 1: API Semantics

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,*

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

which gives `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 known 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 $\mathcal{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_n\}$, 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:

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

We assume that the attacker initially knows all the constant atomic values $\mathcal{C}$ that we note V_0 and we consider an initial empty handle map H_0. This could be thought as a limitation as it does not allow to have keys pre-shared among different tokens. However this is not the case since we model many devices through a unique handle map. Thus, when a fresh key is generated it is implicitly shared among an arbitrary number of tokens. 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" [22, page 30], is modelled by requiring that sensitive keys, that are not already known by the attacker, should never be learned by the attacker. In fact, note that PKCS#11 allows for importing keys in the clear as sensitive: if these keys are known and imported by the attacker we cannot prove any security property about them. We also 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 such that $val \notin dom(\mathsf{H})$. If val is such that for all g, $\mathsf{H}(g) = (val, T)$ implies $S \in T$ we say that val is sensitive in H. If we additionally have that for all g, $\mathsf{H}(g) = (val, T)$ implies $A \in T$ we say that val is always-sensitive in H.*

Example 3. *Suppose that handle-map H associates the handle g to $(v, \{A, S, E, D\})$ and g' to $(v, \{S, W, U\})$, i.e., the same value v is stored in the device under two different handles g and g'. By Definition 1 we have that v is sensitive but not always-sensitive in H since S is set in both the occurrences of v while A is only set under handle g.*

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 $val, \mathsf{kdf}(v, val) \notin \mathcal{K}(V) \cup \mathcal{K}(V')$, for all values v.*

Example 4. *Consider the example of an attack described in the introduction:*

$$\begin{array}{l} \mathit{h2} := \texttt{C\_GenerateKey}(\{\texttt{CKA\_SENSITIVE}, \texttt{CKA\_WRAP}, \texttt{CKA\_DECRYPT}\}); \\ \mathit{wrapped} := \texttt{C\_WrapKey}(\mathit{h1}, \mathit{h2}); \\ \mathit{leak} := \texttt{C\_Decrypt}(\mathit{wrapped}, \mathit{h2}); \end{array}$$

where `C_WrapKey` *is specified as in Example 1, and*

```
C_GenerateKey($T$)              C_Decrypt($data, h$)
  $x :=$ genKey($T$);              $k :=$ getKey($h$, $\{D\}$);
  return $x$;                      return dec($data, k$);
```

Notice that we are slightly abusing the syntax, since templates cannot be passed as parameters. Thus, the definition of `C_GenerateKey`(T) *should be read as a family of definitions, one for each template T.*

Suppose now $\langle \mathsf{H}_0, V_0 \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}, V \rangle$ with $\mathsf{H} = [\mathit{h1} \mapsto (v_1, \{A, S\})]$ and $v_1 \notin \mathcal{K}(V)$, i.e., after some transitions the device contains a sensitive and always-sensitive key v_1 which is not known by the attacker. The first API call generates a new key v_2 with S, W and D set, then we wrap and decrypt key v_1 under this new key. These two operations succeed since S, W and D are respectively set on the newly generated key. Formally:

$$\begin{array}{lll} \langle \mathsf{H}, V \rangle & & \\ \quad \rightsquigarrow_{\mathcal{A}} \langle \mathsf{H}[\mathit{h2} \mapsto (v_2, \{S, W, D\})], V \rangle & & (\texttt{C\_GenerateKey}) \\ \quad \rightsquigarrow_{\mathcal{A}} \langle \mathsf{H}[\mathit{h2} \mapsto (v_2, \{S, W, D\})], V \cup \{\mathit{enc}(v_1, v_2)\} \rangle & & (\texttt{C\_WrapKey}) \\ \quad \rightsquigarrow_{\mathcal{A}} \langle \mathsf{H}[\mathit{h2} \mapsto (v_2, \{S, W, D\})], V \cup \{\mathit{enc}(v_1, v_2), v_1\} \rangle & & (\texttt{C\_Decrypt}) \end{array}$$

The attack is successful and $v_1 \in \mathcal{K}(V')$ with $V' = V \cup \{\mathit{enc}(v_1, v_2), v_1\}$. We conclude that the API is insecure since the above reduction violates item 1 of Definition 2.

To illustrate item 2 of the definition, consider the attacker trying to import a key as always-sensitive through the code setKey$(v, \{A, S\})$*. Since v is known to the attacker, this would lead to a configuration $\langle \mathsf{H}[h \mapsto (v, \{A, S\})], V \rangle$ with $v \in \mathcal{K}(V)$, which breaks item 2 of Definition 2. In fact, key v is always-sensitive, meaning that it should be unknown to the attacker during all if its life-cycle. While it is acceptable that a sensitive key is imported into the token, always-sensitive keys should only be generated by the token.*

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 [22]: two new attributes are introduced: `CKA_TRUSTED` and `CKA_WRAP_WITH_TRUSTED`. Keys with `CKA_TRUSTED` set can only be added by the security officer in a protected environment so that no attack on those keys is possible. Keys with the `CKA_WRAP_WITH_TRUSTED` attribute (that we do not model here) set can only be wrapped via such security officer keys. Intuitively, this prevents keys with `CKA_WRAP_WITH_TRUSTED` set to be

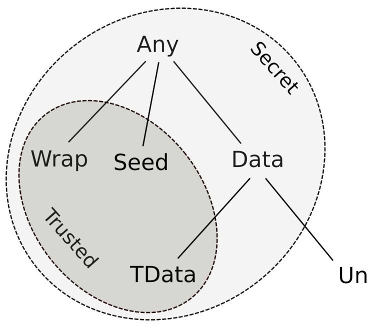

Figure 1: Subtyping relation

wrapped under compromised 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 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.

$$\begin{array}{lcl} \rho & ::= & \mathsf{Any} \mid \mathsf{Data} \mid \mathsf{TData} \mid \mathsf{Wrap} \mid \mathsf{Seed} \mid \mathsf{Un} \\ \tau & ::= & \rho \mid \mathsf{Wrap}[\rho] \end{array}$$

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 in Figure 1. Subtyping captures the fact that values and keys of one type can be given a bigger, supertype without breaking security. For example untrusted values of type Un can be promoted to secret keys of type Data, as this increases the secrecy level and will force programs to be more protective on those values, which is safe from a security perspective. At the same time, trusted keys of type TData, Wrap and Seed can be regarded as secret keys of type Data or Any, since this will restrict such keys to operate on untrusted values only. In fact, it is safe to increase the level of secrecy and decrease the level of trust. We will prove that subtyping does not compromise security in Lemma 1, below.

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

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

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

Table 2: Typing templates

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 [22], 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 coexist on data and wrapping keys, so to disambiguate key roles. Intuitively, trusted keys that are neither typed Wrap, nor typed Data are considered of type Seed, while sensitive keys with mixed roles, e.g., E plus W, are given type Any.

Formally, 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 only have E or D set, and Any otherwise; always-sensitive keys are typed TData if they only have E or D set, Wrap if they only have W or U set and Seed otherwise. Notice that no wrapping untrusted keys are allowed, in fact sensitive 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 [6] it has been observed that real devices often limit the allowed 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 let $\mathbb{T}$ be the set of all permitted 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 [12], 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 permitted templates of keys, and ρ is the default type for wrapped keys.

Example 5 (Security policy)**.** *An example of security policy that separates key roles is* $\mathbb{T} = \{\{W, U, S, A\}, \{E, D\}, \{E, D, S\}, \{E, D, S, A\}\}$ *with* $\rho = \mathsf{Data}$. *Notice that it is not possible to generate keys which are, at the same time, wrapping and decryption keys (W and D never occur together). The policy allows for wrapping and unwrapping data keys of type* $\rho = \mathsf{Data}$. *In Section 5.6 we will show that this simple role separation is not enough to provide API security.*

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.

Typing rules are reported in Table 3. Rules [*var*] and [*sub*] are standard and derive 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 operations 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 a supertype of Un, otherwise we would be able to encrypt a broken key and then unwrap/import it as

$$[\textit{var}] \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash_\rho x : \tau}$$

$$[\textit{sub}] \quad \frac{\Gamma \vdash_\rho e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash_\rho e : \tau}$$

$$[\textit{kdf-w}] \quad \frac{\Gamma \vdash_\rho x : \mathsf{Seed}}{\Gamma \vdash_\rho \mathsf{kdf}(\mathsf{w}_{\rho'}, x) : \mathsf{Wrap}[\rho']}$$

$$[\textit{kdf-d}] \quad \frac{\Gamma \vdash_\rho x : \mathsf{Seed}}{\Gamma \vdash_\rho \mathsf{kdf}(\mathsf{d}, x) : \mathsf{Data}}$$

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

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

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

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

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

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

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

$$[\textit{enc-any}] \quad \frac{\Gamma \vdash_\rho x : \mathsf{Any} \quad \Gamma \vdash_\rho e : \mathsf{Un} \quad \mathsf{Un} \leq \rho}{\Gamma \vdash_\rho \mathsf{enc}(e, x) : \mathsf{Un}}$$

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

Table 3: Typing expressions

[*assign*] $$\frac{\Gamma(x) = \tau \quad \Gamma \vdash_{\rho} e : \tau}{\Gamma \vdash_{\mathbb{T},\rho} x := e}$$

[*seq*] $$\frac{\Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}_1 \quad \Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}_2}{\Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}_1 ; \mathsf{c}_2}$$

[*getkey*] $$\frac{\Gamma(x) = \mathsf{LUB}(T, \mathbb{T}) \quad \Gamma \vdash_{\rho} y : \mathsf{Un}}{\Gamma \vdash_{\mathbb{T},\rho} x := \mathsf{getKey}(y, T)}$$

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

[*setkey*] $$\frac{\Gamma(x) = \mathsf{Un} \quad \vdash T : \tau \quad \Gamma \vdash_{\rho} y : \tau \quad T \in \mathbb{T}}{\Gamma \vdash_{\mathbb{T},\rho} x := \mathsf{setKey}(y, T)}$$

[*return*] $$\frac{\Gamma \vdash_{\rho} e : \mathsf{Un}}{\Gamma \vdash_{\mathbb{T},\rho} \mathsf{return}\ e}$$

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

[*API*] $$\frac{\forall \mathsf{a} \in \mathcal{A} \quad \Gamma \vdash_{\mathbb{T},\rho} \mathsf{a}}{\Gamma \vdash_{\mathbb{T},\rho} \mathcal{A}}$$

Table 4: Typing APIs

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 the sequential sub-part of a program, respectively. Rule [*getkey*], 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 $T \subseteq T'$. If no such a template exists, the least upper bound is undefined.

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

Rule [*genkey*] checks that the template for the new key is in the set of the admitted template $\mathbb{T}$, while [*setkey*] 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.

Example 6 (Typing rule [*getkey*])**.** *We now illustrate which type is assigned to an extracted key while using the* [*getkey*] *rule, and we want to enphasize how this is strictly related to the policy that is chosen. Let us first assume that* $\mathbb{T}$ *contains all the possible templates of keys, i.e.,* $\mathbb{T} = \{T \subseteq \{S, A, E, D, W, U\} \mid S \notin T \Rightarrow A \notin T\}$*. Then, for any possible template* T *we have that the type of the obtained key is* $\mathsf{LUB}(T, \mathbb{T}) =$ Any*. Intuitively, with this* $\mathbb{T}$ *we cannot derive any precise information about keys. By restricting the allowed templates we obtain a more refined and interesting typing. E.g., with* $\mathbb{T}' = \{\{W, U, A, S\}\{E, D, S\}\}$ *we have* $\mathsf{LUB}(\{W\}, \mathbb{T}') = \mathsf{Wrap}$ *and* $\mathsf{LUB}(\{E\}, \mathbb{T}') = \mathsf{Data}$*. Here, checking* W *and* E *respectively identifies* Wrap *and* Data *keys. Intuitively, this happens because* $\mathbb{T}'$ *clearly separates data and wrapping keys.*

3.1 Type soundness

We give a notion of value well-formedness in order to track the value integrity at runtime. 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., $\Theta(\mathsf{w}_{\rho'}) = \Theta(\mathsf{d}) =$ 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* $T \subseteq \mathbb{T}$*. If additionally,* $v \in \mathcal{G}$ *and* $v \notin dom(\mathsf{H})$ *then there exists* v'' *such that* $\mathsf{H}(v'') = (v, T')$ *,* $\vdash T' : \tau'$ *and* $\Theta(v) = \tau'$*.*

We now prove that if we only give the attacker untrusted 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:

Proposition 1. *Let* V *be a set of values such that* $v \in V$ *implies* $\Theta \vdash_\rho v : \mathsf{Un}$*. Then,* $v' \in \mathcal{K}(V)$ *implies* $\Theta \vdash_\rho v' : \mathsf{Un}$*.*

Proof. By 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 consider the case $enc(v_1, v_2) \in \mathcal{K}(V)$ because of $v_1, v_2 \in \mathcal{K}(V)$. By rule [*enc*] and observing that $\mathsf{Un} \leq \mathsf{Data}$ we obtain the thesis. The other cases are analogous except for $v' \in \mathcal{K}(V)$ because of $enc(v', v'') \in \mathcal{K}(V)$ and $v'' \in \mathcal{K}(V)$. By induction we know that $\Theta \vdash_\rho enc(v', v'') :$ Un and $\Theta \vdash_\rho v'' : \mathsf{Un}$. We now have to consider all the typing rules that can derive $\Theta \vdash_\rho enc(v', v'') : \mathsf{Un}$. Among them, the only ones admitting $\Theta \vdash_\rho v'' : \mathsf{Un}$ are [*enc*] and [*enc-any*]. In both cases we have $\Theta \vdash_\rho v' : \mathsf{Un}$, from which the thesis. □

$[\textit{atom}]$
$$\frac{\Theta(val) = \rho'}{\Theta \vdash_\rho val : \rho'}$$

$[\textit{sub}]$
$$\frac{\Theta \vdash_\rho v : \tau' \quad \tau' \leq \tau}{\Theta \vdash_\rho v : \tau}$$

$[\textit{kdf-w}]$
$$\frac{\Theta \vdash_\rho v : \mathsf{Seed}}{\Theta \vdash_\rho \mathit{kdf}(\mathsf{w}_\rho, v) : \mathsf{Wrap}[\rho]}$$

$[\textit{kdf-d}]$
$$\frac{\Theta \vdash_\rho v : \mathsf{Seed}}{\Theta \vdash_\rho \mathit{kdf}(\mathsf{d}, v) : \mathsf{Data}}$$

$[\textit{kdf-un}]$
$$\frac{\Theta \vdash_\rho v, v' : \mathsf{Un}}{\Theta \vdash_\rho \mathit{kdf}(v', v) : \mathsf{Un}}$$

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

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

$[\textit{wrap}]$
$$\frac{\Theta \vdash_\rho v : \mathsf{Wrap} \quad \Theta \vdash_\rho v' : \rho}{\Theta \vdash_\rho enc(v', v) : \mathsf{Un}}$$

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

$[\textit{wrap-div}]$
$$\frac{\Theta \vdash_\rho v : \mathsf{Wrap}[\rho'] \quad \Theta \vdash_\rho v' : \rho'}{\Theta \vdash_\rho enc(v', v) : \mathsf{Un}}$$

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

$[\textit{enc-any}]$
$$\frac{\Theta \vdash_\rho v : \mathsf{Any} \quad \Theta \vdash_\rho v' : \mathsf{Un} \quad \mathsf{Un} \leq \rho}{\Theta \vdash_\rho enc(v', v) : \mathsf{Un}}$$

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

Table 5: Value well-formedness

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 untrusted values can never be typed as trusted and trusted values can never be typed as Un.

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

Proof. By induction on the derivation of $\Theta \vdash_\rho v : \tau$. Base case is [*atom*] and trivially gives $\tau = \tau' = \Theta(v)$. Case [*sub*] is easily proved by induction, observing that $\tau'' \leq \tau'$ and $\tau'' \leq \tau$ imply $\tau \leq \tau'$ or $\tau' \leq \tau$. This observation is also helpful when we have [*sub*] as the last rule for deriving $\Theta \vdash_\rho v : \tau'$. We show one more case of the inductive step. Suppose $\Theta \vdash_\rho \mathit{kdf}(\mathsf{w}_\rho, v) : \mathsf{Wrap}[\rho]$ because of [*kdf-w*]. We have two possibilities for $\tau' \neq \tau$: [*kdf-d*] and [*kdf-un*]. The former, however, is ruled out by the different tag in the value. Consider then [*kdf-un*]. By induction we know that v should be typed with two related types, which is not the case since Seed and Un are not in the subtyping relation. This gives a contradiction and excludes also this case. We conclude that $\tau' = \tau$. 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 derivation of $\Gamma \vdash_\rho e : \tau$. Base case is [*var*], i.e. when e is x. Thesis directly follows by memory well-formedness $\Gamma, \Theta \vdash_{\mathbb{T},\rho} \mathsf{M}$. The inductive step mainly amounts to using inductive hypothesis and apply the corresponding rule for deriving $\Theta \vdash_\rho v : \tau$. The only exception is decryption as its behaviour depends on the correctness of the key. Consider for example [*dec*]. We have $\Gamma \vdash_\rho \mathsf{dec}(e, x) : \mathsf{Un}$ because of $\Gamma \vdash_\rho x : \mathsf{Data}$ and $\Gamma \vdash_\rho e : \mathsf{Un}$. We have two cases depending whether or not decryption will be successful, i.e., whether or not we have $e \downarrow^{\mathsf{M}} \mathit{enc}(v', v'')$ and $x \downarrow^{\mathsf{M}} v''$. If this is not the case (key is wrong), thesis is directly obtained by applying the corresponding typing rule [*dec*] for values. If instead decryption succeeds, we have that $\mathsf{dec}(e, x) \downarrow^{\mathsf{M}} v'$. By induction hypothesis we know that $\Theta \vdash_\rho v'' : \mathsf{Data}$. Now, in Table 5 the only rules for encrypted values that type the key v'' with a type related to Data (Lemma 2) are [*enc*] or [*enc-any*], both of which prove the plaintext to be of type Un as required.

Consider now [*unwrap*]. We have $\Gamma \vdash_\rho \mathsf{dec}(e, x) : \rho$ because of $\Gamma \vdash_\rho x : \mathsf{Wrap}$ and $\Gamma \vdash_\rho e : \mathsf{Un}$. Again, the interesting case in when decryption succeeds and we have $e \downarrow^{\mathsf{M}} \mathit{enc}(v', v'')$, $x \downarrow^{\mathsf{M}} v''$ and $\mathsf{dec}(e, x) \downarrow^{\mathsf{M}} v'$. By induction hypothesis we know that $\Theta \vdash_\rho v'' : \mathsf{Wrap}$. Now, in Table 5 the only rules for encrypted values that type the key v'' with a type related to Wrap (Lemma 2) are [*wrap*] or [*enc-any*]. The former requires the plaintext to be of type ρ as required, the latter type the plaintext as Un but can only be applied when $\mathsf{Un} \leq \rho$ which, by subtyping, gives the thesis. 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 $\langle \mathsf{M}, \mathsf{H}, \mathsf{c} \rangle \rightarrow \langle \mathsf{M}', \mathsf{H}', \mathsf{c}' \rangle$ 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. Proof of item (i) is by easy 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. Assignment is easily proved by applying Lemma 3. Function $\mathsf{getKey}(y, T)$ assigns the retrieved value v to a variable x typed as $\mathsf{LUB}(T, \mathbb{T})$. We thus need to prove that $\Theta \vdash_\rho v : \mathsf{LUB}(T, \mathbb{T})$. Let $\mathsf{H}(\mathsf{M}(y)) = (v, T')$ with $\vdash T' : \tau$. From $\Theta \vdash_{\mathbb{T},\rho} \mathsf{H}$ we know $\Theta \vdash_\rho v : \tau$ and $T' \subseteq \mathbb{T}$. By the semantics of $\mathsf{getKey}(y, T)$ we know that $T \subseteq T'$. Thus $\tau \leq \mathsf{LUB}(T, \mathbb{T})$ and by subtyping $\Theta \vdash_\rho v : \mathsf{LUB}(T, \mathbb{T})$, giving the thesis.

Cases genKey and setKey are easily proved by observing that the returned handle and the new key are fresh names that we add to Θ, with suitable types, in order to type the new memory (this is why we have a potentially bigger Θ' in the thesis). Template T is checked to be included in $\mathbb{T}$, and the type of the imported key value is checked to be the same as the one derived from the template. Inductive step for sequential composition is trivially proved. □

Next lemma proves that values learned by the attacker from well-typed APIs can always be typed Un.

Proposition 2. *Let Let* $\Gamma \vdash_{\mathbb{T},\rho} \mathcal{A}$ *and* $\langle \mathsf{H}_1, V_1 \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}, V \rangle$ *with* $\Theta_1 \vdash_{\mathbb{T},\rho} \mathsf{H}_1$ *and* $\Theta_1 \vdash_\rho v : \mathsf{Un}$ *for each* $v \in V_1$*. Then,* $\exists \Theta \supseteq \Theta_1$ *such that* $\Theta \vdash_{\mathbb{T},\rho} \mathsf{H}$ *and* $\Theta \vdash_\rho v : \mathsf{Un}$ *for each* $v \in V$*.*

Proof. Proof is by induction on the length of reduction $\langle \mathsf{H}_1, V_1 \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}, V \rangle$. Base case is length 0, meaning that $\mathsf{H}_1 = \mathsf{H}$ and $V_1 = V$. It is enough to take $\Theta = \Theta_1$.

For the inductive case we have $\langle \mathsf{H}_1, V_1 \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}_n, V_n \rangle \rightsquigarrow_{\mathcal{A}} \langle \mathsf{H}, V \rangle$. By inductive hypothesis there exists Θ such that $\Theta \vdash_{\mathbb{T},\rho} \mathsf{H}_n$ and $\Theta \vdash_\rho v : \mathsf{Un}$ 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 $\mathsf{a}(v_1, \ldots, v_k) \downharpoonleft^{\mathsf{H}_n, \mathsf{H}} v$ with $v_1, \ldots, v_k \in \mathcal{K}(V_n)$ 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 $\Gamma \vdash_{\mathbb{T},\rho} \mathsf{c}$. By Proposition 1, since $v_1, \ldots, v_k \in \mathcal{K}(V_n)$ we have that $\Theta \vdash_\rho v_1 : \mathsf{Un} \ldots \Theta \vdash_\rho v_n : \mathsf{Un}$. 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 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. Let $\langle \mathsf{H}_0, V_0 \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}, V \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}', V' \rangle$. Consider Θ_0 such that $\Theta_0(val) = \mathsf{Un}$ for each $val \in V_0$. Since H_0 is empty we trivially obtain $\Theta_0 \vdash_{\mathbb{T},\rho} \mathsf{H}_0$. By applying

Proposition 2 twice, we obtain that there exist $\Theta' \supseteq \Theta \supseteq \Theta_0$ such that $\Theta \vdash_{\mathbb{T},\rho} \mathsf{H}$, $\Theta' \vdash_{\mathbb{T},\rho} \mathsf{H}'$, $\Theta \vdash_\rho v : \mathsf{Un}$ for each $v \in V$ and $\Theta' \vdash_\rho v : \mathsf{Un}$ for each $v \in V'$.

Item 1 of Definition 2. Let $val \notin \mathcal{K}(V)$ with val sensitive in H. Since V_0 contains all constant atomic names and $V_0 \subseteq V$ we have that $val \in \mathcal{G}$. By definition, sensitive values cannot be in $dom(\mathsf{H})$ so by $\Theta \vdash_{\mathbb{T},\rho} \mathsf{H}$ we obtain that there exists v'' such that $\mathsf{H}(v'') = (val, T)$, $\vdash T : \tau$ and $\Theta(val) = \tau$. Since val is sensitive in H we know that $S \in T$ which implies $\tau \neq \mathsf{Un}$. Given that $\Theta' \supseteq \Theta$ we also have $\Theta'(val) = \tau \neq \mathsf{Un}$. From Proposition 1 we obtain that $val \notin \mathcal{K}(V')$ which gives the thesis.

Item 2 of Definition 2. Let val be always-sensitive in H. Thus, $\mathsf{H}(g) = (val, T)$ implies $A \in T$ which implies $\vdash T : \tau$ with $\tau \in \{\mathsf{Wrap}, \mathsf{TData}, \mathsf{Seed}\}$, i.e., all templates of val are typed with one of $\mathsf{Wrap}, \mathsf{TData}, \mathsf{Seed}$. By $\Theta \vdash_{\mathbb{T},\rho} \mathsf{H}$ we have that $\Theta \vdash_\rho v : \tau$ with τ one of $\mathsf{Wrap}, \mathsf{TData}, \mathsf{Seed}$. This, by Lemma 2, implies $\Theta \not\vdash_\rho val : \mathsf{Un}$. From Proposition 1 we obtain that $val \notin \mathcal{K}(V)$. Since $\Theta' \supseteq \Theta$ implies $\Theta' \vdash_\rho v : \tau$ with τ one of $\mathsf{Wrap}, \mathsf{TData}, \mathsf{Seed}$, again by Lemma 2 and Proposition 1 we obtain $\Theta' \not\vdash_\rho val : \mathsf{Un}$ and $val \notin \mathcal{K}(V')$.

Consider now $\mathsf{kdf}(v, val)$. Since we have proved $\Theta \not\vdash_\rho val : \mathsf{Un}$ and $\Theta' \not\vdash_\rho val : \mathsf{Un}$, we have that $\Theta \not\vdash_\rho \mathsf{kdf}(v, val) : \mathsf{Un}$ and $\Theta' \not\vdash_\rho \mathsf{kdf}(v, val) : \mathsf{Un}$, given that the only typing rule that would give $\Theta \vdash_\rho \mathsf{kdf}(v, val) : \mathsf{Un}$ is [*kdf-un*] which requires $\Theta \vdash_\rho val : \mathsf{Un}$ (and similarly for Θ'). As before, by Proposition 1 we obtain that $\mathsf{kdf}(v, val) \notin \mathcal{K}(V)$ and $\mathsf{kdf}(v, val) \notin \mathcal{K}(V')$. □

4 Type-based analysis

In this section we consider different implementations of (a subset of) PKCS#11 APIs 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($data$, $h\_key$)
  $k :=$ getKey($h\_key$, $\{E\}$)       $k :=$ getKey($h\_key$, $\{D\}$);
  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($h\_key$, $h\_w$)                  C_UnwrapKey($data$, $h\_w$)
  $w :=$ getKey($h\_w$, $\{W\}$)               $w :=$ getKey($h\_w$, $\{U\}$)
  $k :=$ getKey($h\_key$, $\{\}$);             $k :=$ dec($data, w$);
  return enc($k, w$);                          return setKey($k, \{S\}$);
```

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 return $\mathsf{enc}(k, w)$ requires $\Gamma \vdash_\rho$ return $\mathsf{enc}(k, w)$: Un. Command $k := \mathsf{getKey}(h\_key, \{\})$ requires that $\Gamma \vdash_\rho k$: Any. Typing $w := \mathsf{getKey}(h\_w, \{W\})$ requires w to have type $\mathsf{LUB}(\{W\}, \mathbb{T}) =$ 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 $\Gamma \vdash_\rho$ return $\mathsf{enc}(k, w)$: Un, giving a contradiction.

Secure Templates. We now analyse and prove the security of a fix proposed in [6, 7]. 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 getKey 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 type for wrapped key is Any.

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

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

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

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

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

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

Notice, in particular, that $\Gamma \vdash_\rho \mathsf{getKey}(h\_key,\ \{A,S\}) : \mathsf{Seed}$ since $\mathsf{LUB}(\{A,S\}, \mathbb{T}) =$ 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 $\mathsf{w}_{\rho'}$. 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 and typed as follows:

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

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

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 $\mathsf{w}_{\rho'}$ 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 Prototype implementation

We now investigate in which extent the formal framework developed so far scales to a real setting. Given that we do not have access to actual device firmware, we have performed our experiments on the Opencryptoki software simulator.[3] Opencryptoki is a fully-fledged open-source implementation of a PKCS#11 token for Linux. It is completely written in C. We have implemented a type-checker that is able to verify

[3]http://sourceforge.net/projects/opencryptoki/

the Opencryptoki implementation of wrap, unwrap, encrypt and decrypt operations. This has required some modification of the source code in order to both incorporate the security policy, and make the code more suitable to be type-checked. Moreover, typing rules have been slightly revisited in order to work on the real code. As discussed below, we do not aim at type-checking the whole C implementation but we extract from the actual code the interesting fragments which performs checks on attributes and implement calls to the relevant key management operations, which are then type-checked against the (dynamically) enforced policy. In summary we have:

1. coded the set $\mathbb{T}$ of allowed templates in C as an array of PKCS#11 key templates;
2. modified the source code so to enforce that key templates only belong to $\mathbb{T}$. If this is violated the error `CKR_TEMPLATE_INCONSISTENT` is returned;
3. adapted our type-checking to the finer granularity of cryptographic operations. Opencryptoki, as many existing real devices, implements encryption and decryption in two phases: *initialisation* of the cipher and the actual *cryptographic operation*;
4. defined coarser-grained key management operations that include cryptographic ones. This simplifies the handling of situations in which the actual loading of the key from the memory happens in low-level, mechanism-dependent functions;
5. implemented a prototype type-checker in Python that parses the policy, looks for predefined calls to cryptographic and key-management functions, and type-checks them with respect to the enforced policy. Type-checking is fully automatic;
6. run experiments with various policies and attacks. As expected, all flawed policies did not type-check, while for type-checked code we could not find any working attack.

These experimental results seem to suggest that our type-based analysis scales with not much effort to real settings. If implemented on the firmware or middlewere of real hardware this would allow manufacturers to easily re-profile their devices in secure, type-checkable ways. Our experience as consultants let us believe that this is an important feature since there is no generic secure configuration which works well with any existing application.

Even if our type-checker parses and analyses the actual C source code, we want to point out that our experiments do not aim at verifying the whole source code and that we do not perform any smart data-flow analysis. We are only interested in validating the fragment dealing with key-management, and our typing is based on reasonable assumptions on lower level calls. For example, we assume that our policy enforcement code is correct and cannot be bypassed. More importantly, we assume that the cryptographic and key-management functions that we type-check are the only ones manipulating keys and ciphertexts. Validating these assumptions is out of the scope of current research and is probably part, to some extent, of the standard firmware/middleware development life-cycle. Our prototype should be thought as a proof-of-concept that this kind of analysis might be useful in practice.

We now describe in more detail the above steps.

5.1 Security policy

Recall that a policy is expressed as a pair $(\mathbb{T}, \rho)$, where $\mathbb{T}$ is the set of all permitted templates of keys, and ρ is the default type for wrapped keys. The set $\mathbb{T}$ of allowed templates has been directly coded in C in order to make it enforceable by Opencryptoki. Table 6 reports an example. Types `CK_ULONG`, `CK_ATTRIBUTE` and `CK_BBOOL` are all defined in the PKCS#11 standard and correspond to unsigned longs, attributes and booleans, respectively. $\mathbb{T}$ is coded as T, an array of elements of type `ALLOWED_TEMPLATE` containing the size of the template, the template and a flag indicating whether or not the template is valid for unwrapped keys (see below). In the example of Table 6 we have three admissible templates starting at lines 13, 25 and 35.

Each template is defined as an array of attributes (type `CK_ATTRIBUTE`), as required by the standard. Attributes are triplets: the first element is the attribute name, the second is a pointer to the value, the third is the size of the value. For example, {`CKA_WRAP, &yes, sizeof(CK_BBOOL)`} encodes the fact that attribute `CKA_WRAP` should be set. Attributes that are neither set or unset, as in the second template of Table 6 which only defines the values of 4 attributes, are not constrained, meaning that they can be freely set or unset. We thus have that `T` corresponds to $\mathbb{T} = \{\{W, U, S, A\}, \{E, D\}, \{E, D, S\}, \{E, D, S, A\}\}$. Notice, in particular, that $\{E, D, S, A\}$ comes from the second template by setting S and A. Notice also that $\{E, D, A\}$ is not a valid template as non-sensitive keys can never be always-sensitive, as explained in Section 2.

The last element `is_unwrap` of `ALLOWED_TEMPLATE` is an extension with respect to the theoretical framework. It allows for more granularity when enforcing the policy on the token. In particular, only the templates with `is_unwrap` set to true are permitted when unwrapping a key. From this information we can easily derive the second component of the policy ρ as the greatest lower bound of the set of admissible unwrap templates. In this case, in fact, we are guaranteed that unwrapped keys will never be given a template whose type is not derivable from ρ. If the greatest lower bound does not exist, we let ρ be the default type Data meaning that only data keys can be wrapped and unwrapped, similarly to what happens in PKCS#11 v2.20 when `CKA_WRAP_WITH_TRUSTED` is set (see Section 6 for more details). This default type for ρ can be changed by the user, if needed. In this specific example we obtain $\rho = \mathsf{Data}$. In fact, only $\{E, D, S\}$ can be used when unwrapping a key and it is typed as Data.

5.2 Policy enforcement

Policy is enforced by checking the attributes of any newly generated/imported key. This is done by searching into `T` (coded as in Table 6) an admissible template that matches, on the defined attributes, the one of the actual key template `tmpl` assigned to the key. As discussed in previous section, attributes that are not defined can be freely set or unset. When performing an unwrap, only the templates admissible for that operation are checked. Policy enforcement is implemented by the following fragment of code:

```c
typedef struct ALLOWED_TEMPLATE {
   CK_ULONG len;           // size of template
   CK_ATTRIBUTE tmpl[6];  // template
   CK_BBOOL *is_unwrap;   // is an unwrap template?
 } ALLOWED_TEMPLATE;

CK_BBOOL yes = TRUE;
CK_BBOOL no = FALSE;

CK_ULONG T_len = 3;  // number of templates in T
ALLOWED_TEMPLATE T[] = {
    {
        6,  // length of the template
        {
            {CKA_WRAP, &yes, sizeof(CK_BBOOL)} ,
            {CKA_UNWRAP, &yes, sizeof(CK_BBOOL)} ,
            {CKA_ENCRYPT, &no, sizeof(CK_BBOOL)} ,
            {CKA_DECRYPT, &no, sizeof(CK_BBOOL)} ,
            {CKA_SENSITIVE, &yes, sizeof(CK_BBOOL)} ,
            {CKA_ALWAYS_SENSITIVE, &yes, sizeof(CK_BBOOL)}
        },
        &no // this is NOT an unwrap template
    },
    {
        4,  // length of the template
        {
            {CKA_WRAP, &no, sizeof(CK_BBOOL)} ,
            {CKA_UNWRAP, &no, sizeof(CK_BBOOL)} ,
            {CKA_ENCRYPT, &yes, sizeof(CK_BBOOL)} ,
            {CKA_DECRYPT, &yes, sizeof(CK_BBOOL)}
        },
        &no //  this is NOT an unwrap template
    },
    {
        6,  // length of the template
        {
            {CKA_WRAP, &no, sizeof(CK_BBOOL)} ,
            {CKA_UNWRAP, &no, sizeof(CK_BBOOL)} ,
            {CKA_SENSITIVE, &yes, sizeof(CK_BBOOL)},
            {CKA_ALWAYS_SENSITIVE, &no, sizeof(CK_BBOOL)},
            {CKA_ENCRYPT, &yes, sizeof(CK_BBOOL)} ,
            {CKA_DECRYPT, &yes, sizeof(CK_BBOOL)}
        },
        &yes // this IS an unwrap template
    }
 };
```

Table 6: Example of key-separation $\mathbb{T}$ coded in C

```
int i;
// loop over all templates in the policy until one matches
for (i=0; i<T_len && ( (mode==MODE_UNWRAP && !T[i].is_unwrap)
    || !template_compare(T[i].tmpl,T[i].len,tmpl) ) ; i++);
    if (i==T_len) // no matching template in the policy
        return CKR_TEMPLATE_INCONSISTENT;
```

5.3 Finer granularity of cryptography

Cryptographic operations are implemented by first initialising the cypher with the cryptographic key, and then performing the actual operation. So far, to uniformly treat data encryption and key wrapping we have specified cryptographic operations as single steps, $\mathsf{enc}(e, x)$ and $\mathsf{dec}(e, x)$, respectively. When implementing the typing we need to express these as the composition of two more refined steps: $\mathsf{enc}(e, x)$ as $\mathsf{enc\_init}(x); \mathsf{encrypt}(e)$, and $\mathsf{dec}(e, x)$ as $\mathsf{dec\_init}(x); \mathsf{decrypt}(e)$.

As far as the two steps are executed one after the other no change is needed in the typing. In this case it is convenient to still use the existing typing rules for $\mathsf{enc}(e, x)$ and $\mathsf{dec}(e, x)$. However, PKCS#11 APIs exhibit two different calls corresponding to the two more refined steps: `C_EncryptInit` and `C_Encrypt` for encryption and `C_DecryptInit` and `C_Decrypt` for decryption. To type-check these APIs we need to refine our typing rules so to separately deal with $\mathsf{enc\_init}(x)$, $\mathsf{encrypt}(e)$ and $\mathsf{dec\_init}(x)$, $\mathsf{decrypt}(e)$.

The needed typing rules are easy to derive by observing that data encryption and decryption are always applied on expressions of type Un. We require this constraint when typing $\mathsf{encrypt}(e)$ and $\mathsf{decrypt}(e)$. In order to type $\mathsf{enc\_init}(x)$, $\mathsf{dec\_init}(x)$ it is enough to consider the typing rules for $\mathsf{enc}(u, x)$ and $\mathsf{dec}(u, x)$ where u is a special variable that we type as Un, and such that the cryptographic expression is also typed Un. This latter fact is always true for $\mathsf{enc}(u, x)$ but needs to be checked for $\mathsf{dec}(u, x)$. In fact, it is unsafe to initialise decryption with a key that might decrypt sensitive values, since we have no control on when decryption will happen.

5.4 Coarser granularity of key-management

In order to type-check the C source code without making any complex data-flow analysis, we have identified calls to internal functions that can been expressed as a sequence of commands in our specification language (plus the refined cryptographic operations above). During this task, we have realised that the actual loading of the keys from the memory happens in low-level, mechanism-dependent functions. Type-checking would then require to track calls all the way down. By taking an abstraction step, instead, we have been able to directly type-check the top level call.

Intuitively, we have to 'merge' key-management with cryptographic operations. For example, $\mathsf{enc\_init}(x)$ is done by passing the handle instead of the actual key value. Thus, we consider a new function $\mathsf{enc\_init\_h}(h, T)$ taking the key handle and the template we want to check before performing the operation. In this specific example, it would typically be $\{E\}$ checking that the handle corresponds to an encryption key. This new, more complex operation can be implemented in our language

New function	Definition
$\mathsf{enc\_init\_h}(h, T)$	$x = \mathsf{getKey}(h, T)$; $\mathsf{enc\_init}(x)$;
$\mathsf{dec\_init\_h}(h, T)$	$x = \mathsf{getKey}(h, T)$; $\mathsf{dec\_init}(x)$;
$\mathsf{enc\_h}(h1, T1, h2, T2)$	$x = \mathsf{getKey}(h1, T1)$; $y = \mathsf{getKey}(h2, T2)$; return $\mathsf{enc}(x, y)$;
$\mathsf{dec\_h}(e,\ Tset, h, T)$	$y = \mathsf{getKey}(h, T)$; $k = \mathsf{dec}(e, y)$; $h2 = \mathsf{setKey}(k,\ Tset)$; return $h2$

Table 7: Cryptographic functions 'merged' with key-management

```
CK_RV  C_EncryptInit( ST_SESSION_HANDLE   *sSession ,
            CK_MECHANISM_PTR     pMechanism ,
            CK_OBJECT_HANDLE     hKey  )
{
     CK_ATTRIBUTE_TYPE  TKey[] = {CKA_ENCRYPT};
     CK_ULONG     TKeyLen = 1;

     ....

    rc = encr_mgr_init( sess , &sess->encr_ctx , OP_ENCRYPT_INIT,
          pMechanism , hKey , TKey , TKeyLen );
```

Table 8: Fragment of `C_EncryptInit` calling `encr_mgr_init`

as $x = \mathsf{getKey}(h, T); \mathsf{enc\_init}(x)$; and can be type-checked accordingly. We have a similar function for initialising decryption and two functions for key encryption and decryption (wrap and unwrap) working as expected, and summarised in Table 7.

In Table 8 we show a fragment of the actual C code for `C_EncryptInit`. Variable `Tkey` contains the template we want to check. Function `encr_mgr_init` implements $\mathsf{enc\_init\_h}$(`hKey, TKey`). There are extra parameters dealing with sessions and mechanism that we ignore in the analysis. The parameters `TKey, TKeyLen` have been added so to make the template check more explicit. In the original code the check of the attribute `C_EncryptInit` was hidden inside `encr_mgr_init`, based on the `OP_ENCRYPT_INIT` parameter.

5.5 The type-checker

We have developed a Python prototype that parses the policy coded as described in Sections 5.1 and 5.2. Then, it looks for predefined cryptographic and key-management functions such as the above described `encr_mgr_init`, parses the relevant parameters (handles and templates), looks for template initialisations and applies the typing rules. As explained before, we only need to implement typing rules for $\mathsf{enc}(e, x)$ and $\mathsf{dec}(e, x)$ since initialisation of encryption and decryption can be typed as a special case of those general rules. In this first version of the prototype we did not implement the key-diversification fix as we preferred to focus on standard PKCS#11 implementations of the APIs enhanced with our policy enforcement. The type-checker is based on pycparser,[4] a Python module for parsing C source code.

5.6 Experiments

We illustrate our approach with two examples of flawed policies, discussing real attacks and fixes.

Plain PKCS#11. Let us start with a 'clean' Opencryptoki token in which we allow any key template. This is coded by letting `T` contain exactly one empty template. Recall, in fact, that only the attributes that are explicitly set or unset are checked when generating/importing a key. If nothing is set no constraint is enforced.

```
CK_ULONG T_len = 1;
ALLOWED_TEMPLATE T[] = {
    {
        0,
        { },
        &yes
    }
  };
```

This is the standard configuration and the token is vulnerable to all known API level attacks on key management. We consider the simple wrap-decrypt discussed in the introduction. To exemplify, we have installed in the token a sensitive key labelled as 'SecKey'. The challenge is to extract it in the clear. Running a C implementation of the attack we obtain:

```
$ ./simple-wrap-decrypt 12345
Found 1 key labelled as SecKey
Cannot read the sensitive value (CKR_ATTRIBUTE_SENSITIVE)
==== Simple wrap-decrypt ====
Wrapped key: 873fa34795c13b3b
Key value:  [af2c85298a39c685]
$
```

The program first tries to directly read the key value, but this is not possible since the key is sensitive. Next it generates a fresh wrap/decrypt key that is used to wrap-and-then-decrypt the sensitive key. We finally obtain the sensitive key in the cleartext: `af2c85298a39c685`. In order to check that this is really the value of the key it is enough to use it to perform an encryption of a fixed cleartext both in software and on the token, and check that the result is the same.

[4] `http://code.google.com/p/pycparser/`

Since the token is flawed we expect it not to type-check. The type-checker correctly parses the set of allowed templates, showing all the possible 48 combinations of attribute values (16 of the 64 possible combinations are not considered given that always sensitive cannot be set on non sensitive keys). Then, the type-checker tries to infer ρ but it fails, since there is no greater lower bound for the set of all possible templates. In fact, there are conflicting key roles. The tool uses a default type $\rho = \mathsf{Data}$. Let us now see why type-checking fails. This is the output:

```
== Typing 'encr_mgr_init(... TKey)' in 'C_EncryptInit'
       TKey : { CKA_ENCRYPT }   : Any
    OK
== Typing 'decr_mgr_init(... TKey)' in 'C_DecryptInit'
       TKey : { CKA_DECRYPT }   : Any
    [TYPE-CHECK FAILED] bye!
```

The tool is able to type-check the call to `encr_mgr_init` in `C_EncryptInit` (see Table 8). The relevant parameter is the template of the encryption key `Tkey` that is initialized as $\{$`CKA_ENCRYPT`$\}$, which in our formalism is $\{E\}$. When we retrieve the key we have to apply rule *getkey* of Table 4 and the type is computed as $\mathsf{LUB}(\{E\}, \mathbb{T})$ giving type Any. This reflects the intuition that without restricting the allowed templates we cannot establish the type of encryption keys.

Nevertheless, `encr_mgr_init` type-checks. In Section 5.3 we have discussed that $\mathsf{enc\_init}(x)$ can be type-checked as $\mathsf{enc}(u, x)$ with u of type Un. We can thus apply rule *enc-any* of Table 3, since we have picked $\rho = \mathsf{Data}$. It is in fact safe to encrypt untrusted values under any key as far as they are never unwrapped and imported as trusted in the device.

The problem arises with `C_DecryptInit`. In fact, $\mathsf{dec\_init}(x)$ needs be type-checked as $\mathsf{dec}(u, x)$ and we require it to be of type Un. From Table 3 the only applicable rule would be *dec* which however requires x to be typed as Data. In fact we cannot safely decrypt with any key if we have no control on how the decrypted value will be dealt with. Intuitively, this rules out the wrap-decrypt attack where, matter of factly, decryption is exploited to leak sensitive keys.

Simple key-separation fix. The above problem can be solved by clearly separating the key roles. We use the definition of $\mathbb{T}$ coded in Table 6. We report below the output of the successful type-checking:

```
=== Policy parsed from policy.h (length 4):
{ CKA_SENSITIVE, CKA_ALWAYS_SENSITIVE, CKA_WRAP, CKA_UNWRAP }
{ CKA_ENCRYPT, CKA_DECRYPT }
{ CKA_SENSITIVE, CKA_ENCRYPT, CKA_DECRYPT }
{ CKA_SENSITIVE, CKA_ENCRYPT, CKA_ALWAYS_SENSITIVE, CKA_DECRYPT }
=== Unwrap policy parsed from policy.h (length 1):
{ CKA_SENSITIVE, CKA_ENCRYPT, CKA_DECRYPT }
=== You have the following wrapped key type: Data
== Typing 'encr_mgr_init(... TKey)' in 'C_EncryptInit'
      TKey : { CKA_ENCRYPT }   : Data
    OK
== Typing 'decr_mgr_init(... TKey)' in 'C_DecryptInit'
      TKey : { CKA_DECRYPT }   : Data
    OK
 == Typing 'key_mgr_wrap_key(... Twrapping, TKey)' in 'C_WrapKey'
        Twrapping : { CKA_WRAP }   : Wrap
        TKey : { CKA_ENCRYPT, CKA_DECRYPT }   : Data
    OK
 == Typing 'key_mgr_unwrap_key(... Tunwrapping, TKey)' in 'C_UnwrapKey'
        Tunwrapping : { CKA_UNWRAP }   : Wrap
        TKey : { }   : Data
    OK
```

The generated templates $\mathbb{T} = \{\{W, U, S, A\}, \{E, D\}, \{E, D, S\}, \{E, D, S, A\}\}$ correspond to the ones already discussed in Section 5.1. This time the type-checker is able to derive $\rho =$ Data which is the type of the only possible template for unwrapping $\{E, D, S\}$. All the functions type-check. Intuitively, checking the presence of attribute `CKA_ENCRYPT` or `CKA_DECRYPT` is now enough to type the key as Data. Similarly, checking `CKA_WRAP` or `CKA_UNWRAP` is enough to type the key as Wrap. Since $\rho =$ Data, it is also necessary to check that wrapped keys have that type. This is done in `key_mgr_wrap_key` by checking template {`CKA_ENCRYPT, CKA_DECRYPT`} before wrapping keys. This is not necessary when unwrapping as the unwrapping policy only allows to import {`CKA_SENSITIVE, CKA_ENCRYPT, CKA_DECRYPT`}, i.e., Data keys.

Running the wrap-decrypt attack, it now fails:

```
$ ./simple-wrap-decrypt 12345
Found 1 key labelled as SecKey
Cannot read the sensitive value (CKR_ATTRIBUTE_SENSITIVE)
==== Simple wrap-decrypt ====
Cannot generate key (CKR_TEMPLATE_INCONSISTENT)
Something went wrong. Maybe the conflict is detected
$
```

The token enforces the policy and disallows the generation of a key with both `CKA_WRAP` and `CKA_DECRYPT` set.

Unwrap to nonsensitive. We conclude by showing that even a tiny change in the policy may open a flaw in the token. We modify the set of allowed templates by changing line 39 of Table 6 into

```
            {CKA_SENSITIVE, &no, sizeof(CK_BBOOL)},
```

Type-checking fails as follows:

```
=== Policy parsed from policy.h (length 4):
{ CKA_SENSITIVE, CKA_ALWAYS_SENSITIVE, CKA_WRAP, CKA_UNWRAP }
{ CKA_ENCRYPT, CKA_DECRYPT }
{ CKA_SENSITIVE, CKA_ENCRYPT, CKA_DECRYPT }
{ CKA_SENSITIVE, CKA_ENCRYPT, CKA_ALWAYS_SENSITIVE, CKA_DECRYPT }
=== Unwrap policy parsed from policy.h (length 1):
{ CKA_ENCRYPT, CKA_DECRYPT }
=== You have the following wrapped key type: Un
== Typing 'encr_mgr_init(... TKey)' in 'C_EncryptInit'
       TKey : { CKA_ENCRYPT }   : Data
   OK
```

```
== Typing 'decr_mgr_init(... TKey)' in 'C_DecryptInit'
       TKey : { CKA_DECRYPT }  : Data
   OK
== Typing 'key_mgr_wrap_key(... Twrapping, TKey)' in 'C_WrapKey'
       Twrapping : { CKA_WRAP }  : Wrap
       TKey : { CKA_ENCRYPT, CKA_DECRYPT }  : Data
   [TYPE-CHECK FAILED] bye!
```

Notice that policy for unwrap has changed and consequently the derived ρ is now Un. It is in fact possible to unwrap keys without setting the attribute `CKA_SENSITIVE`. This breaks the typing of `C_WrapKey` as it tries to wrap Data keys that, in turn cannot be typed as Un. Intuitively, this suggests an attack where a sensitive Data key is wrapped and then unwrapped as non-sensitive, making it possible to directly read its value. We have implemented this attack and tried it on the token:

```
$ ./unwrap-to-nonsensitive 12345
found 1 key labelled as SecKey
Cannot read the sensitive value (CKR_ATTRIBUTE_SENSITIVE)
==== wrap and then unwrap to non-sensitive ====
Wrapped key: 2C0163E5D2892CA8
Unwrapping as nonsensitive ... SUCCESS
Directly reading the value of unwrapped key ...SUCCESS
Key value: [af2c85298a39c685]
$
```

The very same attack fails when the token is configured as in Table 6 giving

```
Unwrapping as nonsensitive ... FAILED
*** Cannot unwrap key (CKR_TEMPLATE_INCONSISTENT)
```

6 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 [22]. 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 [6, 7], formally proving their security.

Starting from version 2.20, RSA added the new attribute `CKA_WRAP_WITH_TRUSTED` to the standard. This attribute 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 generalised 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 [22] 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 [23]). There are, instead, many problematic issues that need to be considered [17]. 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 attribute 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 seem easy, we believe 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.

We have implemented the analysis in the Opencryptoki software token[5]. This has required minor modifications of the source code and the introduction of more refined cryptographic operations whose typing rules can be easily derived from the ones in the formal framework. We have obtained an enhanced token parametrized with respect to the security policy. The user can re-profile the token for a specific application by picking a different policy. If the new configuration type-checks then the token should rule out, to some extent, key-management API level attacks. Our implementation assumes that big parts of code behave correctly, thus there could be ways to circumvent our enhancements. Nevertheless, our experiment shows that developing a real type-based analysis with formal foundations for PKCS#11 cryptographic devices is promising and effective. In [6, 7] we have extended Opencryptoki with the secure template patch. The extension to public-key cryptography and the implementation of the key diversification fix are left as a future work.

The proposed model does not account for attacks based on timing, termination, power consumption, and does not capture indirect or partial information leakage. Moreover, cryptography is modelled symbolically, so the type-system does not automatically enforce computational security. We leave the study of these aspects as a future work. We have been recently working on an extension of the imperative language to a more general API setting that also allows for asymmetric key cryptography and is not specif-

[5]http://sourceforge.net/projects/opencryptoki/

ically targeted to PKCS#11. Results will be presented at [1].

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

References

[1] P. Adão, R. Focardi, and F. L. Luccio. Type-Based Analysis of Generic Key Management APIs. In *IEEE Computer Security Foundation Symposium*, 2013. To appear.

[2] R. Anderson. The correctness of crypto transaction sets (discussion). In *Revised Papers from the 8th International Workshop on Security Protocols*, pages 128–141, London, UK, 2001. Springer-Verlag.

[3] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing tls with verified cryptographic security. In *IEEE Symposium on Security & Privacy (Oakland)*, 2013.

[4] 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.

[5] M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, 34(10):67–75, October 2001.

[6] 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'10)*, pages 260–269. ACM, 2010.

[7] 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'10)*, Edinburgh, UK, July 2010.

[8] M. Centenaro, R. Focardi, and F.L. Luccio. Type-based Analysis of PKCS#11 Key Management. In *POST*, volume 7215 of *Lecture Notes in Computer Science*, pages 349–368. Springer, 2012.

[9] M. Centenaro, R. Focardi, F.L. Luccio, and G. Steel. Type-Based Analysis of PIN Processing APIs. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, volume 5789 of *LNCS*, pages 53–68. Springer, 2009.

[10] R. Clayton and M. Bond. Experience using a low-cost FPGA design to crack DES keys. In *Cryptographic Hardware and Embedded System (CHES'02)*, volume 2523 of *LNCS*, pages 579–592. Springer, 2003.

[11] J. Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, volume 2779 of *LNCS*, pages 411–425. Springer, 2003.

[12] 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.

[13] 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.

[14] R. Focardi and F.L. Luccio. Secure recharge of disposable RFID tickets. In *The 8th International Workshop on Formal Aspects of Security & Trust (FAST'11)*, volume 7140 of *LNCS*, pages 85–99. Springer, 2011.

[15] C. Fournet, J. Planul, and T. Rezk. Information-flow types for homomorphic encryptions. In *Proc. of the 18th ACM conference on Computer and communications security (CCS'11)*, pages 351–360. ACM, 2011.

[16] S.B. Fröschle and N. Sommer. Reasoning with Past to Prove PKCS#11 Keys Secure. In *The 8th International Workshop on Formal Aspects of Security & Trust (FAST'10)*, volume 6561 of *LNCS*, pages 96–110. Springer, 2010.

[17] S.B. Fröschle and N. Sommer. Concepts and Proofs for Configuring PKCS#11. In *The 8th International Workshop on Formal Aspects of Security & Trust (FAST'11)*, volume 7140 of *LNCS*. Springer, 2011.

[18] S.B. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security, (ARSPA-WITS'09)*, volume 5511 of *LNCS*, pages 92–106, York, UK, 2009. Springer.

[19] G. Keighren, D. Aspinall, and G. Steel. Towards a Type System for Security APIs. In *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security, (ARSPA-WITS'09)*, pages 173–192, 2009.

[20] R. Küsters, T. Truderung, and J. Graf. A framework for the cryptographic verification of java-like programs. In *Prof. of the IEEE 25th Computer Security Foundations Symposium (CSF'12)*, pages 198–212. IEEE Computer Society, 2012.

[21] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.

[22] RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.

[23] 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`.