In proceedings of ARSPA-WITS'10. Springer LNCS.

Formal Analysis of Key Integrity in PKCS#11*

Andrea Falcone and Riccardo Focardi

Università Ca' Foscari di Venezia, Italy
andrea.falcone@tin.it, focardi@dsi.unive.it

Abstract. PKCS#11 is a standard API to cryptographic devices such as smarcards, hardware security modules and usb crypto-tokens. Though widely adopted, this API has been shown to be prone to attacks in which a malicious user gains access to the sensitive keys stored in the devices. In 2008, Delaune, Kremer and Steel proposed a model to formally reason on this kind of attacks. We extend this model to also describe flaws that are based on integrity violations of the stored keys. In particular, we consider scenarios in which a malicious overwriting of keys might fool honest users into using attacker's own keys, while performing sensitive operations. We further enrich the model with a *trusted key* mechanism ensuring that only controlled, non-tampered keys are used in cryptographic operations, and we show how this modified API prevents the above mentioned key-replacement attacks.

1 Introduction

PKCS#11 [9] defines an API to cryptographic devices, such as smartcards, hardware security modules and usb crypto-tokens. Apart from providing access to the functionalities offered by devices as, e.g., data encryption, the API is specifically developed to carefully manipulate cryptographic keys. As an example, keys marked as *sensitive* should never be accessible as plaintext outside the device: an application requiring the encryption of confidential data with a sensitive key stored in the device, will refer to the key via a *handle* pointing to the location where the key is stored; the cryptographic operation will then happen inside the device without exposing the key to the external, untrusted world. Access to devices is regulated via a PIN-based authentication check but RSA Security clearly states that PIN leakage, foreseeable, e.g., if the device is used on an untrusted machine, should not ruin the security of PKCS#11 devices.

Following this consideration, the attacker is assumed to execute any possible sequence of legal API calls. Even under this fairly strong attacker model, the secrecy of *sensitive* keys should still be preserved. Unfortunately, in [4] Clulow showed that this is not the case: there exist rather simple API call sequences that enable to recover the value of a sensitive key. The simplest example is the *wrap-decrypt* attack in which a sensitive key is wrapped, i.e. encrypted, with another key which has been generated with the double role of *wrapping* (sensitive) keys

* Work partially supported by Miur'07 Project SOFT: "*Security Oriented Formal Techniques*"

and *decrypting* data. These roles are, indeed, conflicting. It is clear, in fact, that once the sensitive key has been wrapped and exported, encrypted, out of the device it is sufficient to ask for decryption to obtain the key as plaintext.

In [5], Delaune, Kremer and Steel (DKS) proposed a model of PKCS#11 for formally reasoning on Clulow's attacks. We propose an extension of the DKS-model, in order to reason on how both an attacker and a regular user act on a cryptographic device. The model can therefore explore how the users' actions would influence each other's, accounting for some new capabilities we give to the attacker:

1. overwriting of keys in the device;
2. interception of messages sent on the network by the regular user;
3. disconnection from the system, interrupting the session with the device.

The first capability takes advantage of a vulnerability we found in the API, which may fool the regular user into using some untrusted key. The second capability is typical of attacker models for protocol analysis but is not considered in the DKS-model where there is no distinction between the API calls performed by the honest user and the ones performed by the attacker. For example, in DKS, the result of an encryption call is directly available to the attacker. In our model, this is true only if the call has been performed by the attacker, otherwise the ciphertext has to be explicitly intercepted. The third capability is useful to distinguish attacks where the opponent is always connected to the device, like the ones described in [5], from attacks requiring only a temporary connection. In this way we can describe rather convenient scenarios for the attacker who does not need to constantly break into the user's system in order to stay connected to the device.

Disconnection might also be caused by an external event. Suppose, for example, that the attacker is be able to break the integrity of a key in a token only when it is used on a specific compromised machine, at some public access point. Any subsequent use of that key will be potentially compromised, even when the user connects from her highly secure workstation, inaccessible to any attacker. Our extended model is able to capture this behaviour. In fact, we detail a scenario in which the attacker exploits the loss of one key integrity and manages to violate the secrecy of some sensitive data subsequently sent on the network, encrypted using the tampered key. The second part of the attack does not require the attacker to be connected to the device and is independent of *where* the token is used next.

It is worth noticing that dropping the explicit disconnect event would still find the same attack sequences, as disconnecting simply reduces the actions that the attacker can perform. However, discovering that an attack can be performed, even partially, in a disconnected state is interesting, as it enables the new scenarios described above and points out a dangerous flaw in the API enabling future exploits even when the token becomes inaccessible to the attacker.

We conclude by studying a modified API with a simple mechanism to ensure that only controlled, non-tampered keys are used in sensitive operations, and

then show how this countermeasure impedes the previously described attack. The mechanism is based on the *trusted key* attribute of PKCS#11 used to prevent exporting keys in an insecure way, and is extended to any key involved in sensitive operations, i.e., any such key is required to be trusted. As trusted keys cannot be set by regular users, this prevents the key integrity problems discussed above. It is quite surprising that a similar mechanism is not mandatory in the API. As a matter of fact, in all the experiments we have performed on real crypto-tokens and smartcard, we have ascertained that no one supports the trusted key attribute.

The paper is organized as follows. In Section 2 we introduce the DKS-model [5]. In Section 3 we present the vulnerability we found concerning the integrity of keys, we introduce our model and we illustrate its adequacy through a detailed example of attack; in Section 4 we refine this first model, tweaking a small but significant detail and giving birth to a variation of the model; we then formally compare the relative expressive power of the two proposals. Finally, in Section 5 we study a modified API preventing the previously described attack. Section 6 gives some concluding remarks.

2 Background: the DKS-model of PKCS#11

In [5], Delaune, Kremer and Steel (DKS) proposed a model of PKCS#11 for formally reasoning on API-level attacks. This model is the starting point of our work and we describe it in the following.

Terms. Let $\mathcal{N}$ be a possibly infinite set of *names*, representing keys, data values, nonces, etc., and $\mathcal{X}$ a possibly infinite set of *variables*. Let also Σ be a finite set of *function symbols*, with arity $ar : \Sigma \to \mathbb{N}$, representing, e.g., handles to keys and cryptographic primitives. The set of *plain terms* $\mathcal{PT}$ is defined by the following grammar:

$$\begin{array}{rll} t, t_i := & x & x \in \mathcal{X} \\ \mid & n & n \in \mathcal{N} \\ \mid & f\,(t_1, \ldots, t_n) & f \in \Sigma \text{ and } ar\,(f) = n \end{array}$$

PKCS#11 specifies a number of different attributes for keys and data stored in the devices, dictating their respective usage. For example, a *sensitive* key should never be exported from the device as plaintext. Let now $\mathcal{A}$ be a set of unary function symbols disjoint from Σ, called *attributes*. The set of *attribute terms* $\mathcal{AT}$ is built by applying these attributes to plain terms, i.e., $\mathcal{AT} = \{att\,(t) \mid att \in \mathcal{A},\ t \in \mathcal{PT}\}$. Attribute terms are interpreted as propositions, meaning that there will be a truth value associated with them, representing the value of a given attribute of some plain term. A *literal* is an expression a or $\neg a$ where $a \in \mathcal{AT}$. The set of all terms is $\mathcal{T} = \mathcal{PT} \cup \mathcal{AT}$. Finally, we denote as σ substitutions of variables into terms and we write $t\sigma$ to note the application of σ to all variables of t.

Example 1. Term $senc\,(k_2, k_1) \in \mathcal{PT}$, with $k_1, k_2 \in \mathcal{N}$, represents the symmetric encryption of key k_2 under key k_1. Terms $h(n_1, k_1)$ and $h(n_2, k_2)$ specify two

handles n_1, n_2 to the respective keys k_1, k_2. Attributes $sensitive(n_2), extract(n_2)$ and $wrap(n_1)$ specifies that k_2 is sensitive and extractable and k_1 is a key for wrapping, i.e., encrypting other extractable, possibly sensitive keys. Wrapping keys, such as k_1, is useful to export sensitive and extractable keys in a secure, encrypted form.

Syntax. Rules for modelling APIs are expressed in the form

$$T; L \xrightarrow{\text{new } \tilde{n}} T'; L'$$

where $T, T' \subseteq \mathcal{PT}$ are plain terms, L and L' are sets of literals, $\tilde{n} \in \mathcal{N}$ is a set of names. T and L specify the conditions under which the rule is applicable, whereas T' and L' specify what will be the result of this rule, if applied. Intuitively, the rule can be executed if all the terms in T are present and all the literals in L are evaluated to true in the current state. We will see, in fact, that a *state* is a pair of two elements: a set of ground plain terms S, which represents data in the attacker knowledge, and a partial valuation function V, which maintains the values of the key attributes. The effect of the rule is that the terms in T' are added to the state and the valuation of the attributes is updated according to L'. Label 'new $\tilde{n}$' binds names $\tilde{n}$ in T' and L', forcing them to be different from all the other names in the state. This mechanism, named *fresh renaming*, allows for modelling nonce or key generation.

Example 2. Symmetric encryption is modelled as follows.

$$h\left(x_1, y_1\right), y_2;\ encrypt\left(x_1\right) \ \rightarrow\ senc\left(y_2, y_1\right)$$

Intuitively, if the attacker knows some data y_2 that he wants to encrypt and he also has a reference, represented by the *handle* term $h\left(x_1, y_1\right)$, to some internally stored key y_1 with the *encrypt* attribute set, then the attacker may perform an API call and obtain the ciphertext resulting from the encryption of y_2 under key y_1, i.e., $senc\left(y_2, y_1\right)$.

The DKS-model rules for more PKCS#11 functionalities are reported in Appendix A.

Semantics. As previously mentioned, A *state* (S, V) is a pair of two elements: a set of ground plain terms S, which represents data in the attacker knowledge, and a partial valuation function V, which maintains the boolean values of the key attributes. V is extended to literals and to sets of literals as expected, i.e., $V(\neg a) = \neg V(a)$ and $V\left(L\right) = \bigwedge_{l \in L} V\left(l\right)$ when $V(l)$ is defined for all $l \in L$.

Given a rule[1] $T; L \ \rightarrow\ T'; L'$ transition $(S, V) \rightsquigarrow (S', V')$ may take place if there exists a grounding substitution θ for the rule such that:

- $T\theta \subseteq S$, i.e., terms in T are in the enemy knowledge;

[1] We omit here the formalization of rule fresh renaming, consisting of removing label 'new $\tilde{n}$' from the arrow after having renamed names $\tilde{n}$ so that they differ from all the other names in the rule and in the state. For more detail see [5].

– $V(L\theta) = true$, i.e., all literals in L are evaluated to $true$.

Then, $S' = S \cup T'\theta$, and function V' is defined as follows:

$$dom\,(V') = dom\,(V) \cup \{a \mid (a \in L'\theta) \vee (\neg a \in L'\theta)\}$$

and $\forall a \in dom\,(V')$:

$$V'\,(a) = \begin{cases} true & \text{if } a \in L'\theta \\ false & \text{if } \neg a \in L'\theta \\ V\,(a) & \text{otherwise} \end{cases}$$

Intuitively, literals in L' override the evaluation function for their attributes. All the other attributes are untouched.

Example 3 (The wrap-decrypt attack, single key variant). We show how the models work by illustrating a variant of the wrap-decrypt attack of [4] with one single key. The DKS-rules for wrap and decrypt are as follows:

$$h\,(x_1, y_1)\,, h\,(x_2, y_2)\,; wrap\,(x_1)\,, extract\,(x_2) \rightarrow senc\,(y_2, y_1)$$
$$h\,(x_1, y_1)\,, senc\,(y_2, y_1)\,; decrypt\,(x_1) \rightarrow y_2$$

It now sufficient to consider an initial state (S_0, V_0) with $S_0 = \{h(n, k)\}$ and $V_0(wrap(n)) = V_0(extract(n)) = V_0(decrypt(n)) = V_0(sensitive(n)) = true$. Intuitively, we consider a single key k with handle n, with the attributes *wrap*, *extract*, *decrypt* and *sensitive* set. Notice that, being it sensitive, k should never be extracted from the device as plaintext. Intuitively, being k both a wrapping and extractable key, it can be wrapped under itself. To see this, formally, consider the first rule above (wrap) and the substitution $\theta = \{n/x_1, n/x_2, k/y_1, k/y_2\}$. We have:

$$\begin{aligned} T\theta &= \{h\,(x_1, y_1)\,, h\,(x_2, y_2)\}\theta \\ &= \{h(n, k)\} \subseteq S_0 \end{aligned}$$

$$\begin{aligned} V_0(L\theta) &= V_0(\{wrap\,(x_1)\,, extract\,(x_2)\}\theta) \\ &= V_0(\{wrap\,(n)\,, extract\,(n)\}) \\ &= V_0(wrap\,(n)) \wedge V_0(extract\,(n)) = true \end{aligned}$$

Thus $(S_0, V_0) \rightsquigarrow (S_0 \cup \{senc(k, k)\}, V_0)$, with term $senc(k, k)$ representing the wrapping of k under itself and obtained as $T'\theta = senc\,(y_2, y_1)\,\theta = senc(k, k)$. Now, being k also a decryption key it can be used to decrypt term $senc(k, k)$ giving the sensitive key k as a plaintext. We leave as an exercise the formal application of the decrypt rule using substitution $\theta' = \{n/x_1, k/y_1, k/y_2\}$ and giving transition $(S_0 \cup \{senc(k, k)\}, V_0) \rightsquigarrow (S_0 \cup \{senc(k, k), k\}, V_0)$. This attack shows that *wrap* and *decrypt* are conflicting attributes for keys: they should never be both set on the same key.

The decision procedure arising from the model has been automated via model checking, leading to a series of experiments reproducing the attacks already shown by Clulow in [4], and also finding new ones [11]. The authors also proposed 'patches' on the API preventing the attacks in the model.

3 Extending the model: key integrity

We shift the focus of the analysis by taking into account a new vulnerability we observed on real tokens: the attacker is able to overwrite keys stored on the token, in such a way that the user application will refer the new substitute key as it was its own. Assume one application refers to keys by means of their *label* attribute, a short textual description, managed by the applications themselves, on which no security policy is enforced by the API. Thus, an attacker logged into the token can overwrite one of the regular user's keys, say k_1, with another key of his choice, k_2, by simply copying k_1's label onto k_2 and then deleting k_1 from the token. As a result, the next time a honest user will refer the target key k_1 by means of its label, he will be given access to the attacker's key k_2, in a transparent way. We have tested this attack on different USB crypto-tokens we possess.

We now extend the DKS-model of previous section so to also represent the above mentioned scenario. In particular, we describe the actions of two users: together with the enemy E we take into consideration a regular, trusted user T. Their sessions on the device may run concurrently, thus each step of the system represents an API call performed by one of them. States are enlarged to provide both users with their own sets of known terms, namely S_E and S_T. The states also maintain the attribute values for token keys and the *connection status* of the users. The state of the system may be altered as a result of the users' actions: for instance an encryption operation would generate in the model a new term, representing the resulting ciphertext, to be added to the knowledge set of the user requesting this operation.

Having both an attacker and a regular user acting at the same time on the token, this model can describe scenarios in which the two affect each other's actions. Based on this idea, we give the enemy new capabilities: the ability to intercept information sent on the network by the trusted user and the ability to indirectly affect the actions of the regular user, by modifying keys stored on the token. In fact, in our model the attacker has the ability to overwrite keys stored on the device, as discussed above.

Furthermore, the enemy has the ability to disconnect himself from the system, ceasing the session with the device. Disconnecting from the token means losing all the references to the keys stored on it and also losing the ability to require cryptographic operations from the device. Note that nonetheless, even if disconnected, the attacker can still perform cryptographic operations implemented in software libraries. As discussed in the Introduction, this allows for more practical and convenient attacks, in which a compromised key also compromises subsequent cryptographic operations based on such a key. The enemy, for example, does not need to be connected to the device to decrypt any sensitive data encrypted under a previously tampered key he possesses. For the sake of simplicity we omit modelling a complementary 'reconnect' action but this could easily accounted for in the model.

We formalize the above ideas by extending the DKS-model of section 2.

Key overwriting

$$h(x_1, y_2), senc(y_1, y_2); unwrap(x_1) \xrightarrow{\text{used } n} h(n, y_1); extract(n), L$$
$$h(x_1, priv(z)), aenc(y_1, pub(z)); unwrap(x_1) \xrightarrow{\text{used } n} h(n, y_1); extract(n), L$$
$$h(x_1, y_2), senc(priv(z), y_2); unwrap(x_1) \xrightarrow{\text{used } n} h(n, priv(z)); extract(n), L$$

where, $L = \neg wrap(n), \neg unwrap(n), \neg encrypt(n), \neg decrypt(n), \neg sensitive(n)$.

Disconnected

$$x, y \longrightarrow senc(x, y)$$
$$senc(x, y), y \longrightarrow x$$
$$pub(z), x \longrightarrow aenc(x, pub(z))$$
$$aenc(x, pub(z)), priv(z) \longrightarrow x$$

Table 1. Rules for key overwriting and disconnected users

Syntax. We extend rule syntax as follows:

$$T; L \xrightarrow{\text{new } \tilde{n},\ \text{used } \tilde{m}} T'; L'$$

The only difference is the new label 'used $\tilde{m}$' forcing the usage of names occurring in some handle already known by the user: when the rule is fired, all the occurrences in T' and L' of names in $\tilde{m}$ must be replaced by names already in use as the first element m of a handle term $h(m, k)$ in the actual user's knowledge set. This mechanism, named *used renaming*, allows us to model the key overwriting operation, because it pinpoints handle names already in use, thus selecting the handle to be replaced.

Key overwrite rules. Based on the extended syntax, we give new rules for modelling key overwrite. As an example:

$$h(x_1, y_2), senc(y_1, y_2); unwrap(x_1) \xrightarrow{\text{used } n} h(n, y_1); extract(n), \mathcal{L}$$

with $\mathcal{L} = \neg wrap(n), \neg unwrap(n), \neg encrypt(n), \neg decrypt(n), \neg sensitive(n)$, models an unwrap operation overwriting key referred by handle n. Knowing the handle $h(x_1, y_2)$ and the wrapped key $senc(y_1, y_2)$, and assuming x_1 refers to an unwrapping key, we can unwrap y_1 giving a used handle n to refer to it, thus overwriting a key already stored in the device. The appropriate attributes are set to the unwrapped key. Notice that this rule corresponds to the first unwrap DKS-rule of appendix A with 'new n' replaced by 'used n'.

The other rules to model key overwriting for asymmetric cryptography are given in table 1. In the same table we also give rules for operations implemented in software and called 'Disconnected', being them executable even when users are not connected to the token. These rules are not described in [5] but they are implicitly part of the DKS-model too, as they correspond to standard Dolev-Yao attackers.

Semantics. A state is now represented by a tuple (S_T, S_E, V, C_T, C_E), where

- $S_T, S_E \subseteq \mathcal{PT}$ represent the knowledge of the trusted user and the enemy;
- V, as before, is a partial boolean function evaluating attributes $\mathcal{AT}$;
- $C_T, C_E \in \{true, false\}$ are two booleans indicating the connection status of the two users.

We define the following operations on set of ground plain terms:

$$H(S) = \{h(n,k) \in S \mid n \in \mathcal{N}\}$$
$$\hat{H}(S_1, S_2) = \{h(n,k) \in S_1 \mid n \in \mathcal{N} \text{ and } \exists k' \text{ such that } h(n,k') \in S_2\}$$

$H(S)$ returns the set of all the handles in a given set S of ground plain terms. $\hat{H}(S_1, S_2)$ returns all the handles in S_1 also referring to keys in S_2.

Example 4. Let $S_E = \{k_1, h(n_1, k_2), h(n_2, k_3)\}$ and $S_T = \{k_1, h(n_1, k_4)\}$, then $H(S_E) = \{h(n_1, k_2), h(n_2, k_3)\}$, $H(S_T) = \{h(n_1, k_4)\}$ give the handles in S_E and S_T. Moreover, $\hat{H}(S_E, S_T) = \{h(n_1, k_2)\}$ since n_1 refers to a key also in S_T. Similarly, $\hat{H}(S_T, S_E) = \{h(n_1, k_4)\}$.

In the following we will use $\sigma \in \{T, E\}$ to select one of the users, with the complement $\bar{\sigma}$ defined as expected, i.e. $\bar{T} = E$ and $\bar{E} = T$. The system can evolve through one of the following transitions:

- **σ-call**: user σ performs a call to the API or an operation in software, the former only possible if she is connected;
- **Send**: one or more terms of the trusted user's knowledge become part of the attacker's knowledge. This mimics the attacker intercepting data sent over the network by the regular user;
- **Disconnect**: the attacker closes his session with the token and loses all his handles.

We describe the three kinds of transition in detail.

The σ-call transition. This transition extends the execution of a rule in the DKS-model: first, if the rule is not one of the 'Disconnected' (table 1), i.e. if the rule uses handles referring to keys stored in the device, the user must be connected. Second, the premises of the rule must be in the knowledge of user σ and the non-handle terms generated by the rule-firing are visible only to such user, i.e. the one performing the call, while new handles are visible to both. As for DKS-model, we assume rules have been renamed so that new names are different from any other name around (fresh renaming). Moreover, for each label 'used m', we rename m so that there exists $h(m,k) \in H(S_T) \cup H(S_E)$, i.e., m is the handle of some key in one of the knowledge sets (used renaming).

Given $T; L \rightarrow T'; L'$ transition $(S_T, S_E, V, C_T, C_E) \rightsquigarrow (S'_T, S'_E, V', C_T, C_E)$ takes place if there exists a grounding substitution θ for the rule such that

- $C_\sigma \vee H(T\theta) = L\theta = H(T'\theta) = L'\theta = \emptyset$, i.e., either the user is connected or the rule does not refer to any handles and attributes (it is a 'Disconnected');

– $T\theta \subseteq S_\sigma$, i.e., terms in $T\theta$ are in the knowledge of user σ;
– $\mathrm{V}(L\theta) = true$, i.e., all literals in L are evaluated to $true$.

Then $S'_\sigma = S_\sigma \setminus \hat{H}\left(S_\sigma, T'\theta\right) \cup T'\theta$ and

$$S'_{\bar{\sigma}} = \begin{cases} S_{\bar{\sigma}} \setminus \hat{H}\left(S_{\bar{\sigma}}, T'\theta\right) \cup H\left(T'\theta\right) & \text{if } C_{\bar{\sigma}} \\ S_{\bar{\sigma}} & \text{if } \neg C_{\bar{\sigma}} \end{cases}$$

Intuitively, the terms in the rule consequence are added to the knowledge set of the σ-user; terms that represent homonym handles, i.e. key handles that share their first argument with some handle in the rule consequence, must be subtracted from the system, since they are going to be overwritten. The manipulation of token keys is immediately visible to all the sessions on the token, thus the other user's knowledge set (i.e. $S_{\bar{\sigma}}$) must be updated accordingly: only handle terms in the rule consequence are added to it and homonym handles are subtracted; of course, if instead $\bar{\sigma}$ is disconnected, there will be no change on user's knowledge.

Function V' is defined as follows

$$dom\left(V'\right) = dom\left(V\right) \setminus \hat{A} \cup \{a \mid (a \in L'\theta) \vee (\neg a \in L'\theta)\}$$

where $\hat{A} = \left\{att\left(n\right) \mid \exists h\left(n,k\right) \in \hat{H}\left(S_\sigma, T'\theta\right),\ att \in \mathcal{A},\ n \in \mathcal{N}\right\}$ is the set of the attribute-terms in the V function domain that will be "forgotten" due to their respective handles being overwritten in this transition. Also, attribute terms in the rule consequence are added to the function domain. V' values then are defined as in DKS-model (section 2). Finally, notice that C_T and C_E are unchanged since no disconnection can happen within this transition.

Example 5 (σ-call transition). We illustrate how a symmetric encryption operation performed by the attacker is represented in the model. Consider the names set $\mathcal{N} = \{m, n, k_1, k_2\}$. Assume, also, that at step number i the state is $q_i = (S_{T,i}, S_{E,i}, V_i, true, true)$ with $S_{E,i} = \{h\left(n, k_1\right), k_2\}$, $V_i\left(encrypt\left(n\right)\right) = true$. In state q_i the attacker has thus access to key k_1 through the handle named n, which is entitled for encryption, and he knows the value of key k_2; also, both the users are connected to the token. The values of the other state elements are irrelevant here.

To carry out the next step in computation, an applicable transition must be looked for, on behalf of either the regular user or the attacker. Of all the applicable transitions in this state, an *E-call* for the symmetric encryption is chosen. We report the corresponding rule here for convenience:

$$h\left(x_1, y_1\right), y_2; encrypt\left(x_1\right) \rightarrow senc\left(y_2, y_1\right)$$

There is no need for a renaming operation on this rule before execution, and it is applicable in the current state q_i with the substitution $\theta_i = \{n/x_1, k_1/y_1, k_2/y_2\}$: in fact, we have that $\{h\left(x_1, y_1\right), y_2\}\,\theta_1 \subseteq S_{E,i}$ and $V_i\left(\{encrypt\left(x_1\right)\}\,\theta_1\right) = true$. Therefore, $q_i \rightsquigarrow q_{i+1} = (S_{T,i+1}, S_{E,i+1}, V_{i+1}, true, true)$ such that:

– $S_{T,i+1} = S_{T,i}$
– $S_{E,i+1} = S_{E,i} \cup \{senc\,(y_2, y_1)\}\,\theta_i = S_{E,i} \cup \{senc\,(k_2, k_1)\}$
– V_{i+1} is such that
  • $dom\,(V_{i+1}) = dom\,(V_i)$
  • $V_{i+1}\,(encrypt\,(x_1)) = true$, thus $V_{i+1} = V_i$

Example 6 (overwriting a key). We now illustrate what happens when firing the previously described key overwriting rule:

$$h\,(x_1, y_2)\,, senc\,(y_1, y_2)\,; unwrap\,(x_1) \xrightarrow{\text{used } n} h\,(n, y_1)\,; extract\,(n)\,, L$$

We give a synthetic representation of the scenario, detailing only the knowledge set of the attacker, who is executing the rule.

step	S_E
i	$h\,(n_1, k_1)\,, senc\,(k_3, k_2)\,, h\,(n_2, k_2)$
i+1	$h\,(n_1, \mathbf{k_3})\,, senc\,(k_3, k_2)\,, h\,(n_2, k_2)$

We see that key k_3 is unwrapped and *overwrites* key k_1. Formally, to fire the overwriting rule we first have to perform the 'used renaming' operation obtaining:

$$h\,(x_1, y_2)\,, senc\,(y_1, y_2)\,; unwrap\,(x_1) \rightarrow h\,(n_1, y_1)\,; extract\,(n_1)\,, L\,\{n_1/n\}$$

We have removed the 'used n' label by renaming n in a way it matches an existing handle, in this case $h\,(n_1, k_1)$. Now we can fire the rule using the substitution $\theta_i = \{n_2/x_1, k_2/y_2, k_3/y_1\}$. We obtain that

$$\begin{aligned} S_{E,i+1} &= S_{E,i} \setminus \hat{H}\,(S_{E,i}, T'\theta) \cup T'\theta = \\ &= S_{E,i} \setminus \hat{H}\,(S_{E,i}, \{h(n_1, k_3)\}) \cup \{h(n_1, k_3)\} = \\ &= S_{E,i} \setminus \{h(n_1, k_1)\} \cup \{h(n_1, k_3)\} = \\ &= \{h\,(n_1, k_3)\,, senc\,(k_3, k_2)\,, h\,(n_2, k_2)\} \end{aligned}$$

Notice that the existing conflicting handle $h(n_1, k_1)$ is correctly removed before the new overwriting key $h(n_1, k_3)$ is added to the knowledge.

The Send transition. When a user asks a device for generating a ciphertext it is plausible that she is going to send it on the network or store it in a place she does not completely trust. This transition models exactly the attacker intercepting such an encrypted data.

Given a subset $I \subseteq S_T$ of the trusted user's knowledge set, the system can perform a Send transition $(S_T, S_E, V, C_T, C_E) \rightsquigarrow (S_T, S_E \cup I, V, C_T, C_E)$.

Example 7 (Send transition). Suppose that the trusted user has to send a ciphertext on the network and that this will be intercepted by the attacker. Consider the state is $q = (S_T, S_E, V, C_T, C_E)$, where $S_T = \{h\,(n_1, k_1)\,, senc\,(d, k_1)\}$, meaning that the trusted user has access to the key k_1 through the handle named n_1 and he also knows the ciphertext resulting from the encryption of

some data d under key k_1. Furthermore, $S_E = \{h(n_1, k_1)\}$. The values of the other state elements are irrilevant here. In this state the Send transition is applicable, considering $I \subseteq S_T$, $I = \{senc(d, k_1)\}$. Thus, we have $q \rightsquigarrow q' = (S_T, S_E \cup \{senc(d, k_1)\}, V, C_T, C_E)$.

The Disconnect transition. Since this transition models the disconnection of the attacker from the device with the consequence of losing access to all the token keys, key handles must be deleted from the attacker's knowledge set in the process.

The system may evolve by a Disconnect transition $(S_T, S_E, V, C_T, C_E) \rightsquigarrow (S_T, S_E \setminus H(S_E), V, C_T, false)$.

Example 8 (Disconnect transition). Let $q = (S_T, S_E, V, true, true)$, where $S_E = \{h(n_1, k_1), h(n_2, k_2)\}$, meaning that the attacker has access to two keys through their respective handles. A disconnect operation is issued and reflected in the model as $q_i \rightsquigarrow q_{i+1} = (S_T, \emptyset, V, true, false)$.

A complete key overwriting attack. The features introduced in our model make it possible to describe a scenario in which a key integrity violation is exploited by the attacker to illegally obtain secret data. The regular user possesses some secret data d, which he needs to send on the net, after having encrypted it to protect its secrecy. Also, recall that the attacker knows the regular user's PIN code for the token and he is also able to send commands to the token. Furthermore, the attacker knows which key will be used for the encryption operation and has access to the corresponding handle (but not to the key value).

In Table 2 we give a summary of the attack sequence, detailing only the knowledge sets; in bold font we emphasize what is new at each step. In the initial state the device holds two keys and both users have handles to them: k_t is the trusted user's key and k_i is a wrap/unwrap key that will be exploited by the attacker to import in the device his own key k_e (initially not stored on the device); the trusted user knows the sensitive data d.

During steps 1 and 2 the attacker illegally imports his key in the device, through a process already shown in [5, section 5.3, experiment 2]: he encodes key k_e in the same format used for wrap operations and makes the device encrypt it under k_i. This yields the same result as k_e was regularly wrapped by the token. Then, the attacker calls an unwrap of this piece of data, creating as a result a copy of k_e in the device and obtaining a new handle for it. We adapted this process to model the key overwriting operation: while unwrapping key k_e, the trusted user's key is deleted after having copied its label onto k_e. Note that having key k_i is not mandatory, it's just for clarity: key k_t itself could be used to import k_e, in case its *unwrap* attribute is set to *true*.

Having done this, at step 3 the attacker can disconnect himself from the system: the user key k_t as been replaced and any subsequent encryption (apparently) based on such a key will be compromised, as show below.

At step 4 the trusted user encrypts d: she (unconsciously) refers to k_e because her reference is now pointing to it. At step 5 the trusted user sends the ciphertext resulting from encryption on the net; the attacker intercepts it. Finally, at

step	transition	σ	S_T	S_E
0	-	-	$d, h(t, k_t), h(i, k_i)$	$h(t, k_t), h(i, k_i), k_e$
1	encrypt	E	$d, h(t, k_t), h(i, k_i)$	$h(t, k_t), h(i, k_i), k_e, \mathbf{senc}(\mathbf{k_e}, \mathbf{k_i})$
2	overwrite	E	$d, h(t, \mathbf{k_e}), h(i, k_i)$	$h(t, \mathbf{k_e}), h(i, k_i), k_e, senc(k_e k_i)$
3	Disconnect	-	$d, h(t, k_e), h(i, k_i)$	$k_e, senc(k_e k_i)$
4	encryption	T	$d, h(t, k_e), h(i, k_i),$ $\mathbf{senc}(\mathbf{d}, \mathbf{k_e})$	$k_e, senc(k_e k_i)$
5	Send	-	$d, h(t, k_e), h(i, k_i),$ $senc(d, k_e)$	$k_e, senc(k_e k_i), \mathbf{senc}(\mathbf{d}, \mathbf{k_e})$
6	decryption (disconn.)	E	$d, h(t, k_e), h(i, k_i),$ $senc(d, k_e)$	$k_e, senc(k_e k_i), senc(d, k_e), \mathbf{d}$

Table 2. Key overwriting attack

step 6 the attacker can decrypt the data d. Notice that this last operation is performed off-line, possibly using some software implementation of the decryption algorithm.

4 The $\mathcal{H}$-transition system

We give another formalization of the transition system, aiming at making it more rational in the way it manages the handle terms: as can be seen in examples of previous section the key handles are often redundantly maintained in both the knowledge sets. In the new transition system, the handle terms are made explicit from the knowledge sets and uniquely maintained in a new element in the state vector, named $\mathcal{H}$. The state becomes thus $(\mathcal{H}, S_T, S_E, V, C_T, C_E)$ and we note the new transition as $\rightsquigarrow_{\mathcal{H}}$. We now describe how **$\sigma$-call**, **Send** and **Disconnect** are modified to accommodate the new separate information on handles.

The $\mathcal{H}$ σ-call transition. The σ-call transition requires the same conditions as in the previous transition system to be met in order to execute a rule from $\mathcal{R}$ (recall that a call to the API is only possible for connected users). The only difference is that the handle terms are to be found in the $\mathcal{H}$ element of the state vector. More specifically, in place of condition $T\theta \subseteq S_\sigma$, requiring that terms in $T\theta$ are in the knowledge of user σ, we require:

- $H(T\theta) \subseteq \mathcal{H}$, i.e., all the required handles are in $\mathcal{H}$;
- $T\theta \setminus H(T\theta) \subseteq S_\sigma$, i.e., all the required terms that are not handles are in the knowledge of user σ.

If these conditions are met, then the system can evolve with the transition $(\mathcal{H}, S_T, S_E, V, C_T, C_E) \rightsquigarrow (\mathcal{H}', S_T', S_E', V', C_T, C_E)$, where:

$$\mathcal{H}' = \mathcal{H} \setminus \hat{H}(\mathcal{H}, T'\theta) \cup H(T'\theta)$$
$$K_\sigma' = K_\sigma \cup T'\theta \setminus H(T'\theta)$$
$$K_{\overline{\sigma}}' = K_{\overline{\sigma}}$$

Handle terms in the rule's consequence are added to the handle set, accounting for overwriting operations if needed. The non-handle terms in the rule's consequence are added to the knowledge set of the σ-user. Notice that for $\bar{\sigma}$-user nothing is done, since handles are stored in the 'centralized' set $\mathcal{H}$.

The attribute valuation function V is updated as before, carefully removing all the attributes of overwritten keys.

The $\mathcal{H}$ Send transition. The $\mathcal{H}$ Send transition is analogous to the Send. Given a subset $I \subseteq S_T$ of terms in the trusted user's knowledge set, the system can perform $(\mathcal{H}, S_T, S_E, V, C_T, C_E) \rightsquigarrow_{\mathcal{H}} (\mathcal{H}, S_T, S_E \cup I, V, C_T, C_E)$.

The $\mathcal{H}$ Disconnect transition. The $\mathcal{H}$ Disconnect transition differs from the Disconnect in that no handle terms are deleted from the state vector; just the connection flag is updated: $(\mathcal{H}, S_T, S_E, V, C_T, C_E) \rightsquigarrow_{\mathcal{H}} (\mathcal{H}, S_T, S_E, V, C_T, \mathit{false})$.

Expressiveness results We now show that the two presented models are equivalent regarding their ability to describe evolutions of the token-users system. We formalize and prove a theorem stating that, given a $\rightsquigarrow$ transition there exists a correspondent $\rightsquigarrow_{\mathcal{H}}$ transition, and vice-versa. First we give some preliminary definitions.

Let $\bar{H}(S) = \{S \setminus H(S)\}$ be the set of all the non-handle terms in a set S.. We let also Q and $Q_{\mathcal{H}}$ denote the set of states in the two models.

Definition 1 (State encoding). *The state encoding function* $[\![\cdot]\!] : Q \longrightarrow Q_{\mathcal{H}}$ *is defined as* $[\![(S_T, S_E, V, C_T, C_E)]\!] \triangleq \left(H(S_T), \bar{H}(S_T), \bar{H}(S_E), V, C_T, C_E\right)$.

This definition of state encoding is based on the observation that the trusted user never disconnects from the token: the $\mathcal{H}$ element of $Q_{\mathcal{H}}$ is filled with handles taken explicitly from the trusted user's knowledge set S_T.

Definition 2 (Inverse state encoding). *The inverse state encoding function* $[\![\cdot]\!]^{-1} : Q_{\mathcal{H}} \longrightarrow Q$ *is defined as*

$$[\![(\mathcal{H}, S_T, S_E, V, C_T, C_E)]\!]^{-1} \triangleq \left([\![S_T]\!]^{-1}, [\![S_E]\!]^{-1}, V, C_T, C_E\right)$$

$$\textit{where } [\![S_\sigma]\!]^{-1} = \begin{cases} S_\sigma \cup \mathcal{H}, & \textit{if } C_\sigma \\ S_\sigma & \textit{if } \neg C_\sigma \end{cases}$$

Next lemma states that $[\![\cdot]\!]^{-1}$ is the inverse of $[\![\cdot]\!]$. Proof is given in Appendix B.

Lemma 1. $\forall q \in Q$ we have $[\![[\![q]\!]]\!]^{-1} = q$.

Relying on these encodings, we formalize our main result.

Theorem 1 (models equivalence). *Let* $q_i, q_f \in Q$

$$q_i \rightsquigarrow q_f \quad \Longleftrightarrow \quad [\![q_i]\!] \rightsquigarrow_{\mathcal{H}} [\![q_f]\!]$$

The proof for this theorem is long, but rather mechanical; we give a sketch of it in Appendix B.

5 Adding trusted keys

In this section, we propose a way to remedy the PKCS#11 API vulnerability that lets an attacker break a key integrity, discussed above. The idea is based on the use of one of the cryptographic key attributes, already part of the API: the *trusted* attribute. As the name suggests, this attribute is used to discriminate between trusted and untrusted keys. It is designed to be part of a mechanism by which the API can prevent exporting keys in an insecure way, by forbidding the usage of untrusted keys for wrapping sensitive keys. The *trusted* attribute can only be set by the Security Officer in charge of the token. Since the Securiy Officer's PIN code is different from that of the normal user and is never used on untrusted hosts, the enemy cannot manipulate the *trusted* attribute.

Key integrity can be thus checked by just reading the value of its *trusted* attribute: if the *trusted* attribute value is found to be *false*, it means the key has possibly been tampered. Note that this check can be done either explicitly by the user application, or implicitly by the API or the device itself.

In order to model this new mechanism, we simply check the *trusted* attribute of every key appearing in a rule, before its application. To this aim, we need to update all the rules for the σ-call transition which yield new key handles and have them explicitly set the *trusted* attribute of the newly created handles to *false*, as a default value. The rules affected by this update are all the key generation rules and all the unwrap rules. To this aim it is sufficient to extend $\mathcal{L}$ with $\neg trusted(n)$.

$\mathcal{T}$ σ-call semantics We let $Tr\,(S) = \{trusted\,(n) \mid \exists h\,(n,k) \in S \text{ with } n \in \mathcal{N}\}$ be the set of *trusted* attribute terms relative to all the handle terms in S. We add to the rule applicability conditions a check for the *trusted* attribute value for all the key handle terms in rule premises T. This new condition will only allow the execution of a rule if each of the required key handles has the *trusted* attribute set to *true*. Formally, this is achieved by asking $V\,(Tr\,(T)) = true$.

Note that this new behavior does not directly forbid the key overwriting operation but it makes it possible to detect it in a simple way, as shown below.

Key overwriting attack and trusted keys. Given the above modifications to the semantics, the example attack showed in Example 3 on page 11 is prevented. In Table 3 on the next page we show how the attack sequence is revealed and blocked. Suppose that initially the trusted attribute on the handle for k_t is set to *true*. After step 2 the attacker has successfully overwritten k_t with k_e, $h(t, k_t)$ being replaced by $h(t, k_e)$; nonetheless, the new handle $h(t, k_e)$ does not possess the *trusted* attribute. Thus, at step 4 the normal user's request for a symmetric encryption is refused, because the supplied key, referenced by the name t, fails the *trusted* attribute check. Therefore, the same attack will not be possible. Note that the key still gets overwritten, but the loss of integrity is disclosed.

st.	transition	σ	S_T	S_E	$trusted\,(t)$
0	-	-	$d, h\,(t, k_t)\,, h\,(i, k_i)$	$h\,(t, k_t)\,, h\,(i, k_i)\,, k_e$	$true$
1	encryption	E	$d, h\,(t, k_t)\,, h\,(i, k_i)$	$h\,(t, k_t)\,, h\,(i, k_i)\,, k_e,$ $\mathbf{senc}\,(\mathbf{k_e}, \mathbf{k_i})$	$true$
2	unwrap	E	$d, h\,(t, \mathbf{k_e})\,, h\,(i, k_i)$	$h\,(t, \mathbf{k_e})\,, h\,(i, k_i)\,, k_e,$ $senc\,(k_e k_i)$	**false**
3	Disconnect		$d, h\,(t, k_e)\,, h\,(i, k_i)$	$k_e, senc\,(k_e k_i)$	$false$
4	encryption (**not exec**)	T	-	-	-

Table 3. Key overwriting attack

6 Conclusions

We have proposed an extension to an existent model for the PKCS#11 API, in order to broaden its descriptive capabilities, taking into account the integrity of cryptographic keys. We have granted the attacker some new capabilities, among which the overwriting of keys on the token. We have given a result reasoning on the expressiveness of two variations of the model semantics. Finally, we have proposed a simple mechanism to ensure in the model that only non-tampered keys are used in cryptographic operations.

The PKCS#11 API does not provide a way to satisfactorily manage the integrity of keys stored on a device, in a transparent manner for the user applications. We believe this is a major flaw of the standard. As discussed in the introduction, a user might need to perform some critical operation connecting a crypto-token to an untrusted host and she would expect no one is allowed to tamper with her crypto-device in a way that might compromise *any* further usage of it. Connecting a crypto-device to an untrusted host should not, in fact, compromise what is done/stored inside the device. We have shown that, apart from known attacks where sensitive keys are extracted, an enemy might also substitute a user key with one he knows and learn any data subsequently encrypted with such compromised key. This is, in our opinion, a rather irritating behaviour for what is expected to be a 'secure' crypto-device.

Alternative ways to check key integrity, other than the proposed use of the *trusted* attribute, could be: the use of cryptographic functions to validate key values (e.g. digital signatures or MACs), of course having computational costs; the introduction of a new attribute to specify that a key cannot be deleted, making a key impossible to be overwritten.

The use of the *trusted* attribute we suggest as a countermeasure to the integrity issue has an advantage in that it is simple and fast. Also, it can be implemented at several levels: at user application level, without the need to alter the API or the devices, or at API or device level, gaining transparency for the applications. A main disadvantage is that, since only keys with the *trusted* attribute set to *true* are authorized for any operation, to be of use, a key needs to

be "promoted"right after its generation by the Security Officer in charge of the token. This might become quite restrictive especially in applications that need to establish new session-keys. We intend to explore alternative, less-restrictive, solutions.

We are glad that the automated DKS framework of [11] has been extended by Graham Steel so to include our model and analysis. In [12] it is possible to find the result of the performed experiments. First, a device patched to prevent confidentiality attack is modelled and analysed in the standard DKS-model. As expected no attacks are found. Then, the model is extended so to capture key overwriting and the same patched device is found to be vulnerable. The attack sequence pointed out by the SATMC model checker is essentially the same as the one presented in Table 2. It only differs on what is encrypted with the tampered key and then disclosed by the attacker: a freshly generated sensitive key `ks1` instead of generic sensitive data d.

References

1. Armando, A., Compagna, L. SAT-based model-checking for security protocols. International Journal of Information Security. Volume 7, Issue 1, January 2008.
2. Bond, M. and Anderson, R.: API-Level Attacks on Embedded Systems. IEEE Computer Magazine, pp. 67-75. (2001)
3. Cimatti, et al.: NuSMV version 2: an OpenSource Tool for Symbolic Model Checking. Proc. International Conference on Computer-Aided Verification (CAV'02). Vol. 2404 of LNCS, pp. 359-364. Springer. (2002)
4. Clulow, J.: On the security of PKCS#11. In Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03), volume 2779 of LNCS, pages 411-425, Cologne, Germany (2003). Springer.
5. Delaune, S. , Kremer, S., Steel, G.: 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.
6. Dolev, D.,Yao, A: On the security of public key protocols. IEEE Transactions in Information Theory, pp. 198-208. (1983)
7. International Telecommunication Union: X.690 - Abstract Syntax Notation One (ASN.1). (2002)
8. RSA Laboratories: PKCS#8: Private-Key Information Syntax Standard. (1993.)
9. RSA Security Inc: PKCS #11 v.2.20: Cryptographic Token Interface Standard (June 2004)
10. Steel, G.: Analysis of Security APIs FAQ. Available at `http://www.lsv.ens-cachan.fr/~steel/security\_APIs\_FAQ.html`.
11. Steel, G. Experiments: Secure Configuration of PKCS11. Available at `http://www.lsv.ens-cachan.fr/~steel/pkcs11/`
12. Steel, G. Experiments: Key Integrity in PKCS#11. Available at `http://www.lsv.ens-cachan.fr/~steel/pkcs11/replacement.php`

A DKS-Rules modelling cryptographic operations

Wrap

$$h(x_1, y_1), h(x_2, y_2); wrap(x_1), extract(x_2) \rightarrow senc(y_2, y_1)$$
$$h(x_1, priv(z)), h(x_2, y_2); wrap(x_1), extract(x_2) \rightarrow aenc(y_2, pub(z))$$
$$h(x_1, y_1), h(x_2, priv(z)); wrap(x_1), extract(x_2) \rightarrow senc(priv(z), y_1)$$

Unwrap

$$h(x_1, y_2), senc(y_1, y_2); unwrap(x_1) \xrightarrow{\text{new } n} h(n, y_1); extract(n), \mathcal{L}$$
$$h(x_1, priv(z)), aenc(y_1, pub(z)); unwrap(x_1) \xrightarrow{\text{new } n} h(n, y_1); extract(n), \mathcal{L}$$
$$h(x_1, y_2), senc(priv(z), y_2); unwrap(x_1) \xrightarrow{\text{new } n} h(n, priv(z)); extract(n), \mathcal{L}$$

Key generation

$$\xrightarrow{\text{new } n,k_1} h(n, k_1); \neg extract(n), \mathcal{L}$$
$$\xrightarrow{\text{new } n,s} h(n, priv(s)), pub(s); \neg extract(n), \mathcal{L}$$

Encryption

$$h(x_1, y_1), y_2; encrypt(x_1) \rightarrow senc(y_2, y_1)$$
$$h(x_1, priv(z)), y_1; encrypt(x_1) \rightarrow aenc(y_1, pub(z))$$

Decryption

$$h(x_1, y_1), senc(y_2, y_1); decrypt(x_1) \rightarrow y_2$$
$$h(x_1, priv(z)), aenc(y_2, pub(z)); decrypt(x_1) \rightarrow y_2$$

Attribute set

$$h(x_1, y_1); \neg wrap(x_1) \rightarrow wrap(x_1)$$
$$\vdots$$

Attribute unset

$$h(x_1, y_1); wrap(x_1) \rightarrow \neg wrap(x_1)$$
$$\vdots$$

where $\mathcal{L} = \neg wrap(n), \neg unwrap(n), \neg encrypt(n), \neg decrypt(n), \neg sensitive(n)$. The ellipsis in the attribute set and unset rules indicates that similar rules exist for other attributes.

B Proofs from section 4

To prove Lemma 1 we need the following simple lemma, stating that equality of the handles in S_T and S_E is preserved by transitions as long as the attacker stays connected. When he disconnects, the handles in S_E are permanently removed. This allows us to implicitly assume $H(S_T) = H(S_E)$ if C_E and $H(S_E) = \emptyset$ otherwise, for all states (S_T, S_E, V, C_T, C_E) in Q.

Lemma 2. *Let $(S_T, S_E, V, true, true)$ be such that $H(S_T) = H(S_E)$. Then, $(S_T, S_E, V, true, true) \rightsquigarrow^* (S'_T, S'_E, V', C'_T, C'_E)$ implies $H(S'_T) = H(S'_E)$ if C'_E and $H(S'_E) = \emptyset$, otherwise.*

Proof. Lemma 1. Let $q \in Q$, $q = (S_T, S_E, V, C_T, C_E)$. By definition of encoding, $[\![q]\!] = \left(H\left(S_T\right), \overline{H}\left(S_T\right), \overline{H}\left(S_E\right), V, C_T, C_E\right)$. To this we apply the inverse encoding; we have to distinguish whether the attacker is connected or disconnected from the device.

If the attacker is connected:

$$\begin{aligned}
&[\![\left(H\left(S_T\right), \overline{H}\left(S_T\right), \overline{H}\left(S_E\right), V, C_T, C_E\right)]\!]^{-1} = \\
&\quad = \left([\![\overline{H}\left(S_T\right)]\!]^{-1}, [\![\overline{H}\left(S_E\right)]\!]^{-1}, V, C_T, C_E\right) \\
&\quad = \left(\overline{H}\left(S_T\right) \cup H\left(S_T\right), \overline{H}\left(S_E\right) \cup H\left(S_T\right), V, C_T, C_E\right) \\
&\quad = \left(S_T, S_E, V, C_T, C_E\right) \text{ because } H\left(S_E\right) = H\left(S_T\right) \\
&\quad = q
\end{aligned}$$

If the attacker is disconnected:

$$\begin{aligned}
&[\![\left(H\left(S_T\right), \overline{H}\left(S_T\right), \overline{H}\left(S_E\right), V, C_T, C_E\right)]\!]^{-1} = \\
&\quad = \left([\![\overline{H}\left(S_T\right)]\!]^{-1}, [\![\overline{H}\left(S_E\right)]\!]^{-1}, V, C_T, C_E\right) \\
&\quad = \left(\overline{H}\left(S_T\right) \cup H\left(S_T\right), \overline{H}\left(S_E\right), V, C_T, C_E\right) \\
&\quad = \left(S_T, S_E, V, C_T, C_E\right) \text{ because } H\left(S_E\right) = \emptyset \\
&\quad = q
\end{aligned}$$

□

Proof sketch. Theorem 1.

($\Longrightarrow$) Let $q_i \rightsquigarrow q_f$. We explore case by case, for each kind of derivation applied. We express q_f as a function of q_i by the applied transition and we express q_i' as a function of q_i by the definition of state encoding. Then, given the applicability conditions of the assumed derivation, we verify that analogous conditions, required to apply the same transition in the $\mathcal{H}$ system, hold in state q_i'. Thus, we can apply the same transition to state q_i', which by hypothesis is the encoding of q_i, yielding state q_f'. Finally, we show that q_f' obtained in this way is the encoding of q_f: this is done checking the vector state elements one at a time, to show that the state encoding definition is complied with.

($\Longleftarrow$) This part of the proof is symmetric to the first part and it relies on the inverse state encoding.