2013 IEEE 26th Computer Security Foundations Symposium

Type-Based Analysis of Generic Key Management APIs

Pedro Adão
SQIG–Instituto de Telecomunicações
IST, TULisbon, Lisboa, Portugal
Email: pedro.adao@ist.utl.pt

Riccardo Focardi
DAIS, Università Ca' Foscari
Venezia, Italy
Email: focardi@dsi.unive.it

Flaminia L. Luccio
DAIS, Università Ca' Foscari
Venezia, Italy
Email: luccio@unive.it

***Abstract*—In the past few years, cryptographic key management APIs have been shown to be subject to tricky attacks based on the improper use of cryptographic keys. In fact, real APIs provide mechanisms to declare the intended use of keys but they are not strong enough to provide key security. In this paper, we propose a simple imperative programming language for specifying strongly-typed APIs for the management of symmetric, asymmetric and signing keys. The language requires that type information is stored together with the key but it is independent of the actual low-level implementation. We develop a type-based analysis to prove the preservation of integrity and confidentiality of sensitive keys and we show that our abstraction is expressive enough to code realistic key management APIs.**

I. Introduction

In the recent years cryptography is becoming a key technology to provide security in various settings, and cryptographic hardware and services are becoming more and more pervasive in everyday applications. The interfaces to cryptographic devices and services are implemented as *Security APIs* whose main aim is to allow untrusted code to access resources in a secure way. Typically, these APIs provide *key management* operations such as: the creation or deletion of keys; the encryption/decryption, signing and verification of data through some keys; the import/export of *sensitive* keys, i.e., keys that should never be revealed outside a smart card or hardware security modules (HSMs). These last operations are usually implemented by encrypting these sensitive keys under other keys, operation which is called *key wrapping*.

API calls may be executed on untrusted machines, thus a very important issue is to design secure APIs that enforce a *policy*, that is, security properties have to be maintained no matter what the parameters are, and which sequence of legal API calls is executed. A *key usage* policy is defined by some *key attributes* stored in the key. Examples are the *wrap* attribute that is associated to keys used to encrypt other keys, or the attribute *decrypt* associated to decryption keys. Objects such as cryptographic keys or certificates in tokens, are referenced via *handles*, that are pointers to or names for the objects in the secure memory. These handles do not reveal any information about the actual values of the objects, e.g., of a key, thus objects may be used without necessarily knowing their values but just providing a handle to them. Although these APIs are very powerful, all the proposed implementations are not capable of precisely defining the different roles and uses that objects should have.

In the last decade this has lead to many different attacks both on HSMs and smart cards (see, e.g., [2], [3], [4], [8]). Many of these attacks are related to the key wrapping operation. For example, attacks on the IBM CCA interface are related to the improper bound, provided by the XOR function, between the attributes of a wrapping key and the usage rules [3], and attacks on the PKCS#11 security tokens can be mounted by assigning particular sets of attributes to the keys, and by performing particular sequences of (legal) API calls [4]. In this context some 'patches' have been presented that rely on: imposing a policy on the attributes so that a key cannot be used for conflicting operations; imposing that conflicting attributes are not set at two different instants by limiting the usage of imported keys to some non-critical functions [4], or by adding a wrapping format that binds attributes to wrapped keys [12], [15]. Other attacks on PIN processing APIs are, e.g., on formats used for message encryption [10], or on the lack of integrity of user data [7].

In our opinion, formal and general tools to reason about the security of cryptographic APIs are very important in order to find attacks to real APIs and to test new patches.

Our contribution: In this paper we present an abstract and simple imperative programming language for specifying strongly-typed APIs for the management of symmetric, asymmetric and signing keys. Starting from the definition of an abstract key management language which is strongly typed, i.e., that associates objects to types, we provide a concrete semantics, in which concrete key properties are stored in place of types. We then investigate conditions that allow us to map concrete APIs to the proposed types so that security results are preserved. In particular, we prove that if the translation of the concrete API into the typed one is well-typed then security of keys is guaranteed.

We then study realistic implementations of the APIs. We consider PKCS#11 v2.20 that allows to specify the attributes of wrapped and unwrapped keys [19]. We show that PKCS#11 attributes can be mapped into types preserving the above mentioned conditions, and this allow us to prove security through the general type-checking.

DOI 10.1109/CSF.2013.14

Related work: The current literature proposes different solutions for the designs of new secure token interfaces and the proofs of their security. In [5] secure token interfaces are proposed together with security proofs in the cryptographic model. The security relies on the access to a log of all the operations, solution that seems to be not very practical when applied, e.g., to limited memory devices. Moreover, it does not cover the set of all the possible security properties. In [11] secure token interfaces are proposed for a distributed setting, together with security proofs in the symbolic model. This approach however assumes a limited set of functionalities. In [16] the authors introduce a general security model for cryptographic APIs. They define a new notion of security for cryptographic APIs, and apply this notion to the security proofs both in the symbolic and in the computational model. This new model is able to separate key management from key usage, thus avoiding some of the previous attacks. It is also flexible enough to be adapted to some real security APIs. The main difference with respect to our proposal is the use that we do of types to statically enforce security properties on general APIs.

Our type system is partially inspired in the one in [13], proposed for the different setting of spi-calculus processes for protocol analysis. Apart from the completely different setting, there is another important technical difference with respect to [13]. In our case, we do not assume any integrity check when performing encryption and decryption. When we decrypt with a wrong key we still get a valid term. This is what typically happens in many real world implementations.

In [6] the authors propose a simple language, for the coding of PKCS#11 APIs, and they develop a type-based analysis to prove that the secrecy of sensitive keys is preserved under a certain policy. This solution, is however limited to PKCS#11 cryptographic APIs and to symmetric keys, whereas in this paper we propose a new language that is applicable to general cryptographic APIs, that is, any key storage that is managed through handles, and that manages also asymmetric and signing keys, in the style of [16]. As we will show we will be able to instantiate the PKCS#11 APIs in this new model.

The paper is organized as follows. In Section II we introduce a simple imperative programming language for specifying strongly-typed APIs for the management of symmetric, asymmetric and signing keys, the attacker model and the notion of API security; in Section III we present the type system that enforces API security and the type soundness results; in Section IV we modify the language in order to code real API implementations. In Section V we show how PKCS#11 can be modeled in our framework. We draw some conclusions in Section VI.

II. A LANGUAGE FOR KEY MANAGEMENT APIS

In this section we first introduce a simple imperative language suitable to specify key management APIs. We then formalize the attacker model and define API security. The API language is inspired in [6] but here we allow more expressive types that dictate how keys should be used and what is their security level. Moreover we consider asymmetric encryption and digital signatures that are not accounted for in [6].

Values: We let $\mathcal{C}$ and $\mathcal{G}$, with $\mathcal{C} \cap \mathcal{G} = \emptyset$, respectively be the set of atomic *constants* and *fresh* values. The former is used to model any public data, including non-sensitive keys, while 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:

$$\begin{array}{rcl} v & ::= & val \mid enc(v, v') \mid dec(v, v') \\ & & \mid ek(v) \mid enc^a(v, v') \mid dec^a(v, v') \\ & & \mid vk(v) \mid sig(v, v') \end{array}$$

Intuitively, $enc(v, v')$ (resp. $enc^a(v, v')$) and $dec(v, v')$ (resp. $dec^a(v, v')$) denote value v respectively encrypted and decrypted under key v' in a symmetric (resp. asymmetric) cipher; $ek(v)$ denotes the public *encryption* key corresponding to the private *decryption* key v, and $vk(v)$ is the *verification* key corresponding to the *signing* key v; finally, $sig(v, v')$ denotes the signature of v using key 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; on the other hand, the decryption under v' of $enc(v, v'')$, with $v'' \neq v'$ will be explicitly represented as $dec(enc(v, v''), v')$. This allow us to model cryptosystems with no integrity checks as decrypting with a wrong key never returns a failure. Signature verification, instead, only succeeds when the verification key corresponds to the signing one.

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:

$$\begin{array}{rcl} e & ::= & x \mid \mathsf{enc}(e, x) \mid \mathsf{dec}(e, x) \\ & & \mid \mathsf{ek}(x) \mid \mathsf{enc}^a(e, x) \mid \mathsf{dec}^a(e, x) \\ & & \mid \mathsf{vk}(x) \mid \mathsf{sig}(e, x) \mid \mathsf{ver}(e, x) \end{array}$$

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. The semantics of expressions is defined inductively in Table I. As already mentioned, the modeled encryption mechanism does not perform any integrity check on the messages, so the decryption of a ciphertext under a wrong key gives $dec(v'', v')$. Signature verification, instead, evaluates to the

$$
\begin{array}{rll}
x \downarrow^{\mathsf{M}} \mathsf{M}(x) & & \text{if } \mathsf{M}(x) \text{ is defined} \\
\mathsf{e}(e_1,\ldots,e_n) \downarrow^{\mathsf{M}} \mathsf{e}(v_1,\ldots,v_n) & & \text{if } e_i \downarrow^{\mathsf{M}} v_i, i \in [1,n] \\
\mathsf{enc}(v,v') \downarrow^{\mathsf{M}} enc(v,v') & & \\
\mathsf{dec}(enc(v,v'),v') \downarrow^{\mathsf{M}} v & & \\
\mathsf{dec}(v'',v') \downarrow^{\mathsf{M}} dec(v'',v') & & \text{if } v'' \neq enc(v,v') \\
\mathsf{enc}^a(v,v') \downarrow^{\mathsf{M}} enc^a(v,v') & & \\
\mathsf{ek}(v) \downarrow^{\mathsf{M}} ek(v) & & \\
\mathsf{dec}^a(enc^a(v,ek(v')),v') \downarrow^{\mathsf{M}} v & & \\
\mathsf{dec}^a(v'',v') \downarrow^{\mathsf{M}} dec^a(v'',v') & & \text{if } v'' \neq enc^a(v,ek(v')) \\
\mathsf{sig}(v,v') \downarrow^{\mathsf{M}} sig(v,v') & & \\
\mathsf{vk}(v) \downarrow^{\mathsf{M}} vk(v) & & \\
\mathsf{ver}(sig(v,v'),vk(v')) \downarrow^{\mathsf{M}} v & &
\end{array}
$$

Table I
THE SEMANTICS OF EXPRESSIONS

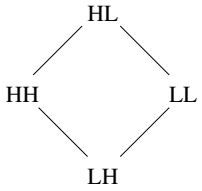

Figure 1. Security lattice

signed message only when the verification key corresponds to the signing key.

Types: Our language is designed around powerful types that specify the intended usage and the security level of each key. A security level is a pair $\ell_C\ell_I$ specifying, separately, the confidentiality (C) and integrity (I) levels. We consider two possible levels: *High* (H) and *Low* (L). For example, HH denotes a high confidentiality and high integrity value, while LH a public (low confidentiality) and high integrity one. Intuitively, high confidentiality values should never be read by opponents while high integrity values should not be modified by opponents, i.e., when high integrity data is received it is expected to have been originated at some trusted source.

Figure 1 is a standard security lattice showing that confidentiality and integrity levels are contra-variant [18]. Moving up is safe while moving down is unsafe, thus it is safe to consider a public datum as secret, while it is unsafe to promote low integrity to high integrity. More formally, the confidentiality and integrity preorders are such that $L \sqsubseteq_C H$ and $H \sqsubseteq_I L$. We let ℓ_C and ℓ_I range over $\{L,H\}$, while we let ℓ range over the pairs $\ell_C\ell_I$ with $\ell_C^1\ell_I^1 \sqsubseteq \ell_C^2\ell_I^2$ if and only if $\ell_C^1 \sqsubseteq_C \ell_C^2$ and $\ell_I^1 \sqsubseteq_I \ell_I^2$.

We define the following types:

$$
\begin{array}{rcl}
T & ::= & X \mid \ell \mid \mu\mathsf{K}^{\ell}[T] \\
\mu & ::= & \mathsf{Sym} \mid \mathsf{Enc} \mid \mathsf{Dec} \mid \mathsf{Sig} \mid \mathsf{Ver}
\end{array} \tag{1}
$$

Intuitively, X is a type variable that will be bounded at runtime by a map $\sigma : X \to T$ from type variables to ground types; type ℓ is used for generic data at security level ℓ; and type $\mu\mathsf{K}^{\ell}[T]$ is used for keys at security level ℓ that are used to perform cryptographic operations on terms of type T. Depending on the label μ, this type may describe symmetric keys, encryption/decryption asymmetric keys, or signing and verification keys. We will see that type information is stored, retrieved and checked at run-time in order to authorize specific cryptographic operations. Type variables allow for some degree of polymorphism so that static analysis can be performed on types that are partially specified. We write $var(T)$ to note the variables occurring in type T.

We allow symmetric, decryption and verification keys to have a payload different from LL only if their level is HH, i.e., when they can really be trusted.

Definition 1 (Types well-formedness)**.** *Let $T = \mu\mathsf{K}^{\ell}[T]$ with $\mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}$. Then $\ell \neq HH$ implies $T = LL$.*

Given a type T we will use $\ell_C(T)$ and $\ell_I(T)$ to denote respectively its confidentiality and integrity levels. Let $\ell = \ell_C^\star\ell_I^\star$. Define $\ell_C(\ell) = \ell_C(\mu\mathsf{K}^{\ell}[T]) = \ell_C^\star$ and $\ell_C(X) = H$; similarly $\ell_I(\ell) = \ell_I(\mu\mathsf{K}^{\ell}[T]) = \ell_I^\star$ and $\ell_I(X) = L$.

We define the notion of subtyping $\leq$ as the least preorder such that:

(1) $\ell_1 \leq \ell_2$ whenever $\ell_1 \sqsubseteq \ell_2$;
(2) $LL \leq \mu\mathsf{K}^{\ell_C L}[LL]$;
(3) $\mu\mathsf{K}^{\ell}[T] \leq \ell$ for any type T.

Intuitively, (1) states that subtyping extends the security level preorder; (2) public and low integrity (LL) terms are regarded as keys performing cryptographic operations on public and low integrity (LL) terms. For example, it is allowed to encrypt a LL term under a LL key; (3) keys can be thought as generic data at the same level. Notice that the opposite would be unsafe, apart from the special case of LL stated in item (2).

Lemma 2. *Let $\sigma : X \to T$ be a map from type variables to ground type. Then, $T \leq T'$ implies $T\sigma \leq T'\sigma$.*

Proof: Conditions (1) and (2) of $\leq$ are on ground types so $T\sigma = T \leq T' = T'\sigma$. Condition (3) we have $T = \mu\mathsf{K}^{\ell}[T] \leq \ell = T'$. In this case $\mu\mathsf{K}^{\ell}[T]\sigma = \mu\mathsf{K}^{\ell}[T\sigma] \leq \ell = T' = T'\sigma$ ∎

Notice that when we have type $\mu\mathsf{K}^{\ell}[T]$ everything has to be ground except T. Even when we have $(\mu\mathsf{K}^{\ell_C(X)L}[T])$, the label $\ell_C(X)L$ is ground and so $(\mu\mathsf{K}^{\ell_C(X)L}[T])\sigma = \mu\mathsf{K}^{\ell_C(X)L}[T\sigma]$.

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:

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

We will only consider API commands in which 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 recorded (ground) type of the key is unifiable with T, the key is returned; any binding of type variables in T which is necessary to match the actual key type is recorded in a special environment σ; genKey generates a key with (ground) type $T\sigma$; finally, setKey imports a new key with plaintext value y and (ground) type $T\sigma$. The first function fails, i.e., is stuck, if the given handle does not exist or refers to a key with a non-matching type. The other functions are stuck if the given type is not ground, once we apply the environment binding σ. 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 return e.

Example 3 (Symmetric key wrapping)**.** *We specify a wrapping API that takes two handles: the wrapped key h_key and the wrapping key h_w. If the wrapped key has the expected type then it is encrypted under the wrapping key and the ciphertext is returned. For the sake of readability, we will always write* $\mathsf{a}(x_1, \ldots, x_k)$ c *in place of* $\mathsf{a} = \lambda x_1, \ldots, x_k.\mathsf{c}$ *to specify an API function:*

$$\begin{array}{l}\mathsf{SymWrap}(h\_key,\ h\_w)\\ \quad w := \mathsf{getKey}(h\_w,\ \mathsf{SymK}^{HH}[X]);\\ \quad k := \mathsf{getKey}(h\_key, X);\\ \quad \mathsf{return\ enc}(k, w);\end{array}$$

Notice the use of type variable X to allow for any type from wrapped key. What is important is that X matches the payload type for the wrapping key, as specified in $\mathsf{SymK}^{HH}[X]$.

Semantics: Device keys are modeled by an handle-map $\mathsf{H} : g \mapsto (v, T)$ that is a partial mapping from the atomic (generated) values to pairs of key values and ground types. Key values are referred by their handles and we allow multiple handles to refer to the same value with eventually different types, for instance, $\mathsf{H}(g) = (v, T)$ and $\mathsf{H}(g') = (v, T')$. By allowing this we are able to deal with multiple devices considering all keys available to the API as a unique 'universal' device. This corresponds to a worst-case scenario in which attackers can simultaneously access all the existing hardware.

An API command c working on a memory M, with a handle-map H and type variable substitution σ is denoted by $\langle \mathsf{M}, \mathsf{H}, \sigma, \mathsf{c}\rangle$. Semantics is presented in Table II, where ϵ denotes the empty API. Assignment $x := e$ evaluates expression e on M and stores the result in variable x, denoted $\mathsf{M}[x \mapsto v]$. 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. Internal function $\mathsf{getKey}(y, T)$ takes the (ground) type T' of the key referred to by y and extends the present binding σ of type variables with a new binding σ' that makes T the same as T'. Binding σ' is minimal, as it only operates on the variables of $T\sigma$. With $\uplus$ we denote the union of two disjoint substitutions.

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 that 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.

Example 4 (Semantics of symmetric key wrapping)**.** *To illustrate the semantics, we present the transitions of the symmetric key wrapping command specified in Example 3. Suppose that the device associates the handle g to $(v, \mathsf{SymK}^{HL}[LL])$ and g' to $(v', \mathsf{SymK}^{HH}[\mathsf{SymK}^{HL}[LL]])$. 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.,* $\mathsf{M} = \mathsf{M}_\epsilon[h\_key \mapsto g, h\_w \mapsto g']$. *Let also assume that $X \notin dom(\sigma)$. Then it follows,*

$$\begin{array}{l}\langle \mathsf{M}, \mathsf{H}, \sigma, w := \mathsf{getKey}(h\_w, \mathsf{SymK}^{HH}[X]));\\ \qquad k := \mathsf{getKey}(h\_key, X); \mathsf{return\ enc}(k, w)\rangle\\ \rightarrow \langle \mathsf{M}[w \mapsto v'], \mathsf{H}, \sigma \uplus [X \mapsto \mathsf{SymK}^{HL}[LL]],\\ \qquad k := \mathsf{getKey}(h\_key, X); \mathsf{return\ enc}(k, w)\rangle\\ \rightarrow \langle \mathsf{M}[w \mapsto v', k \mapsto v], \mathsf{H}, \sigma \uplus [X \mapsto \mathsf{SymK}^{HL}[LL]],\\ \qquad \mathsf{return\ enc}(k, w)\rangle\end{array}$$

which gives $\mathsf{SymWrap}(g, g') \downarrow^{\mathsf{H},\mathsf{H}} enc(v, v')$ *meaning that the value returned invoking the wrap command is thus the encryption of v under v'. Notice how X gets bounded to the type transported by the wrapping key* $\mathsf{SymK}^{HL}[LL]$ *which then matches the type of v stored in* H.

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

(1) $enc(v, v'), enc^a(v, v'), sig(v, v'), ek(v), vk(v) \in \mathcal{K}(V)$;
(2) if $v = enc(v'', v')$ or $v = enc^a(v'', ek(v'))$ then $v'' \in \mathcal{K}(V)$;
(3) if $v \neq enc(v'', v')$ then $dec(v, v') \in \mathcal{K}(V)$;
(4) if $v \neq enc^a(v'', ek(v'))$ then $dec^a(v, v') \in \mathcal{K}(V)$;
(5) if $v = sig(v'', v''')$ and $v' = vk(v''')$ then $v'' \in \mathcal{K}(V)$.

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

$$\frac{\mathsf{H}(\mathsf{M}(y)) = (v, T') \qquad T' = (T\sigma)\sigma' \qquad dom(\sigma') = var(T\sigma)}{\langle \mathsf{M}, \mathsf{H}, \sigma, x := \mathsf{getKey}(y, T)\rangle \to \langle \mathsf{M}[x \mapsto v], \mathsf{H}, \sigma \uplus \sigma', \varepsilon\rangle}$$

$$\frac{\begin{array}{c} g, g' \leftarrow \mathcal{G} \qquad T\sigma \text{ ground} \\ T\sigma = \mu\mathsf{K}^{\ell}[T'] \implies \mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\} \wedge (\ell = HH \vee T' = LL) \end{array}}{\langle \mathsf{M}, \mathsf{H}, \sigma, x := \mathsf{genKey}(T)\rangle \to \langle \mathsf{M}[x \mapsto g], \mathsf{H}[g \mapsto (g', T\sigma)], \sigma, \varepsilon\rangle}$$

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

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

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

Table II
API SEMANTICS

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

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

The initially knowledge of the adversary is given by an arbitrary subset $V_0 \subseteq \mathcal{C}$ and we consider an initial empty handle map H_0. In the following, we use the standard notation $\rightsquigarrow^*_{\mathcal{A}}$ for multi-step reductions.

API security.: We define *confidential* and *secure* keys by inspecting the security levels stored in the handle map. Recall that the same key value can appear under multiple handles. A key that is always stored at a high confidential level should be regarded as *confidential*, however there is no guarantee that the key is not known by the attacker. For example, the attacker might succeed importing a key as confidential in the device. The device will regard it as high confidential but the value comes from the attacker. The situation is different for keys that are stored as high confidential and high integrity (HH). High integrity means that the key cannot come from the attacker. Typically these keys are generated in the device or stored by a security officer in a secure environment. We expect these keys to be confidential in their entire life and we refer to them as *secure* keys.

Definition 5 (Confidential and secure keys)**.** *Let val be an atomic value and H a handle-map such that $val \notin dom(\mathsf{H})$. If val is such that $\mathsf{H}(g) = (val, T)$ implies $\ell_C(T) = H$ we say that val is* confidential *in H. If we additionally have that $T = \mu\mathsf{K}^{HH}[T^{\star}]$ we say that val is* secure *in H.*

The definition of API security follows.

Definition 6 (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 confidential in H implies $val \notin \mathcal{K}(V')$;

(2) val is secure in H implies $val \notin \mathcal{K}(V) \cup \mathcal{K}(V')$.

The above property is not enforced by the semantics as the following example illustrates.

Example 7. *Consider the following insecure API that takes a handle and leaks the corresponding key:*

$$\begin{array}{l} \mathsf{LeakKey}(h\_key) \\ \quad k := \mathsf{getKey}(h\_key, X); \\ \quad \mathsf{return}\ k; \end{array}$$

The key is copied into k and then returned, independently of the associated type. For example if the handle is associated to a secure $\mathsf{SymK}^{HH}[HH]$ key, the key value will be returned and leaked to the attacker, breaking API security definition.

In the next section we develop a type system that statically enforces the API security property.

III. TYPE SYSTEM

Expressions: In order to type expressions and commands we introduce a typing environment $\Gamma : x \mapsto T$ which maps variables to their respective types. We allow only a subset of key types in Γ.

$$\frac{\Gamma(x) = T \quad \Gamma \vdash \diamond}{\Gamma \vdash_e x : T} \text{ [var]} \qquad \frac{\Gamma \vdash_e e : T' \quad T' \leq T}{\Gamma \vdash_e e : T} \text{ [sub]}$$

$$\frac{\Gamma \vdash_e x : \mathsf{DecK}^{\ell_C \ell_I}[T]}{\Gamma \vdash_e \mathsf{ek}(x) : \mathsf{EncK}^{L\ell_I}[T]} \text{ [ek]} \qquad \frac{\Gamma \vdash_e x : \mathsf{SigK}^{\ell_C \ell_I}[T]}{\Gamma \vdash_e \mathsf{vk}(x) : \mathsf{VerK}^{L\ell_I}[T]} \text{ [vk]}$$

$$\frac{\Gamma \vdash_e x : \mathsf{SymK}^{\ell_C \ell_I}[T] \quad \Gamma \vdash_e e : T}{\Gamma \vdash_e \mathsf{enc}(e, x) : L\ell_I} \text{ [enc]}$$

$$\frac{\Gamma \vdash_e x : \mathsf{SymK}^{\ell}[T] \quad \Gamma \vdash_e e : T'}{\Gamma \vdash_e \mathsf{dec}(e, x) : T} \text{ [dec]}$$

$$\frac{\Gamma \vdash_e x : \mathsf{EncK}^{\ell_C \ell_I}[T] \quad \Gamma \vdash_e e : T}{\Gamma \vdash_e \mathsf{enc}^a(e, x) : L\ell_I} \text{ [enca]}$$

$$\frac{\Gamma \vdash_e x : \mathsf{DecK}^{\ell}[T] \quad \Gamma \vdash_e e : T' \quad \ell_I(T') \neq H \implies T = LL}{\Gamma \vdash_e \mathsf{dec}^a(e, x) : T} \text{ [deca]}$$

$$\frac{\Gamma \vdash_e x : \mathsf{SigK}^{\ell_C \ell_I}[T] \quad \Gamma \vdash_e e : T}{\Gamma \vdash_e \mathsf{sig}(e, x) : \ell_C(T)\ell_I} \text{ [sig]}$$

$$\frac{\Gamma \vdash_e x : \mathsf{VerK}^{\ell_C \ell_I}[T] \quad \Gamma \vdash_e e : T' \quad \ell_C(T') = H \implies \ell_I = H}{\Gamma \vdash_e \mathsf{ver}(e, x) : T} \text{ [ver]}$$

Table III
TYPING EXPRESSIONS

$$\frac{\Gamma(x) = T \quad \Gamma \vdash_e e : T}{\Gamma \ \vdash_{\mathsf{c}} \ x := e} \text{ [assign]} \qquad \frac{\Gamma \ \vdash_{\mathsf{c}} \ \mathsf{c}_1 \quad \Gamma \ \vdash_{\mathsf{c}} \ \mathsf{c}_2}{\Gamma \ \vdash_{\mathsf{c}} \ \mathsf{c}_1; \mathsf{c}_2} \text{ [seq]}$$

$$\frac{\Gamma(x) = T \quad \Gamma \vdash_e y : LL}{\Gamma \vdash_{\mathsf{c}} x := \mathsf{getKey}(y, T)} \text{ [getkey]} \qquad \frac{\Gamma(x) = LL}{\Gamma \ \vdash_{\mathsf{c}} \ x := \mathsf{genKey}(T)} \text{ [genkey]}$$

$$\frac{\Gamma(x) = LL \quad \Gamma \vdash_e y : T}{\Gamma \ \vdash_{\mathsf{c}} \ x := \mathsf{setKey}(y, T)} \text{ [setkey]} \qquad \frac{\Gamma \vdash_e e : LL}{\Gamma \ \vdash_{\mathsf{c}} \ \mathsf{return}\ e} \text{ [return]}$$

$$\frac{\Gamma \vdash_e x_1 : LL \quad \ldots \quad \Gamma \vdash_e x_k : LL \quad \Gamma \ \vdash_{\mathsf{c}} \ \mathsf{c}}{\Gamma \ \vdash_{\mathsf{c}} \ \lambda x_1, \ldots, x_k.\mathsf{c}} \text{ [function]}$$

$$\frac{\forall \mathsf{a} \in \mathcal{A} \quad \Gamma \ \vdash_{\mathsf{c}} \ \mathsf{a}}{\Gamma \ \vdash_{\mathsf{c}} \ \mathcal{A}} \text{ [API]}$$

Table IV
TYPING APIS

Definition 8 (Gamma well-formedness)**.** *Let* $\Gamma : x \mapsto T$. *We say that* Γ *is well-formed, written* $\Gamma \vdash \diamond$*, if whenever* $\Gamma(x) = \mu\mathsf{K}^{\ell}[T]$ *it holds:*

(1) $\mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}$*;*

(2) $\ell \neq HH$ *implies* $T = LL$.

With the previous definition we ensure that we only record in Γ the *private* parts of keys, $\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}$, while deriving the corresponding *public* parts, $\mathsf{Enc}, \mathsf{Ver}$ only via the typing rules. Also, symmetric, decryption and signature keys need to be kept secret and can be trusted, for what concerns the transported type, only if they are high integrity. For that, we require that we only transport keys of type different from LL under HH keys.

Type judgment for expressions is denoted $\Gamma \vdash_e e : T$ meaning that expression e is of type T under Γ. Typing rules are presented in Table III. Rules [*var*] and [*sub*] are standard and derive types directly from Γ (for variables) or via subtyping. Without loss of generality we assume that [*sub*] is never applied uselessly in a derivation, that is, we never apply it to obtain the same type as we can just remove this rule from the derivation, nor more than once consecutively as given the sequence $T_1 \leq T_2 \leq \ldots \leq T_n$, we can always substitute it by a single application of [*sub*] with $T_1 \leq T_n$. Rules [*ek*] and [*vk*] derive the types for encryption and verification keys, respectively from decryption and signature ones, by changing their confidentiality level to L. This reflects the fact that these keys are public.

Rule [*enc*] encrypts the result of an expression of type T, as required by the key type. The integrity level of the ciphertext is the same as the integrity level of the encryption key while its confidentiality level becomes L reflecting the fact that it is going to be sent eventually over an untrusted channel. Symmetric decryption [*dec*] gives the original type T to the plaintext.

Rules [*enca*] and [*deca*] are similar but asymmetric decryption gives type LL to the plaintext unless the ciphertext has high integrity. The *rationale* behind this is that unless the ciphertext has high integrity then, since encryption key is public, the plaintext might have come from the attacker and so should be typed as LL.

Finally, rules [*sig*] and [*ver*] behave similarly but signature has the same confidentiality level as the signed expression. This is due to the fact that our verification function recovers the signed message from the signature and so, in order to protect its confidentiality, we have to preserve the confidentiality level in the signature.

APIs: We now type API commands via the judgment $\Gamma \vdash_{\mathsf{c}} \mathsf{c}$ meaning that c is well-typed under Γ. The judgment is formalized in Table IV. 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*] retrieves a key of type T from the device and assigns it to a variable with the same type as the retrieved key; rules [*genkey*] and [*setkey*] store keys and return a LL handle that can be safely sent outside the device. 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 9. *Let us consider again the API in Example 3:*

$$\begin{array}{l}\mathsf{SymWrap}(h\_key,\ h\_w)\\ \quad w := \mathsf{getKey}(h\_w,\ \mathsf{SymK}^{HH}[X]);\\ \quad k := \mathsf{getKey}(h\_key, X);\\ \quad \mathsf{return}\ \mathsf{enc}(k, w);\end{array}$$

In order to type the API we have to type all parameters as LL (rule [function]). Thus we let

$$\Gamma(h\_key) = \Gamma(h\_w) = LL$$

Now by applying rule [getkey] twice we also set

$$\begin{array}{rcl}\Gamma(w) & = & \mathsf{SymK}^{HH}[X]\\ \Gamma(k) & = & X\end{array}$$

Under this Γ we can apply rule [enc] and type $\mathsf{enc}(k, w)$ as LH, since the encryption key w has high integrity. By rule [sub] we can type $\mathsf{enc}(k, w)$ as LL which allows us to type-check the return command, completing the typing.

A. Type soundness

We can now state the main theorem of our paper. The main goal is to be able to show security of APIs via typing. For that, we need to track the consistency between the memory cells and the values recorded in it, as well as the values recorded by the handle-map and their associated types. Moreover, we need to show that this are preserved by any execution. Full proofs of the Propositions of this section can be found in [1].

In order to track the value integrity at run-time we define a notion of value well-formedness. This judgment is based on a mapping $\Theta : val \mapsto T$ from atomic values to ground types that satisfies the following conditions:

$$\begin{array}{l}\Theta(val) = \mu\mathsf{K}^{\ell}[T] \text{ implies } \mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}\\ \Theta(val) = \mu\mathsf{K}^{\ell}[T] \text{ and } \ell \neq HH \text{ then } T = LL\end{array} \quad (2)$$

Rules are given in Table V and follow closely the ones for expressions defined in Table III.

With this definition we may now characterize the run-time types associated with values that represent keys. Similarly to the requirements for a well-defined Γ, we can show that private/symmetric keys are either really trusted, of type HH, or can only transport payloads of level LL. For the case of public-keys their integrity-level needs to be H, meaning that they were derived from good private keys, of type HH, or they can only transport LL payloads.

Proposition 10. *Let $\Theta(val) = T$ and $\Theta \models_v val : T'$. Then $T \leq T'$.*

Proposition 11. *Suppose that $v \neq dec(v', v''), dec^a(v', v'')$ and that $\Theta \models_v v : \mu\mathsf{K}^{\ell}[T]$. Then*

1) *if $\ell = HH$ then v is atomic and $\Theta(v) = \mu\mathsf{K}^{HH}[T]$;*
2) *if $\mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}$ then $\ell = HH$ or $T = LL$;*
3) *if $\mu \in \{\mathsf{Enc}, \mathsf{Ver}\}$ then $\ell = LH$ or $T = LL$.*

We can now show that the type transported by symmetric, decryption and signature keys is unique and, when the level of the key is HH, the key type is also unique.

Proposition 12. *Suppose that $\Theta \models_v v : \mu\mathsf{K}^{\ell}[T]$ and $\Theta \models_v v : \mu'\mathsf{K}^{\ell'}[T']$, and $\mu, \mu' \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}$.*

Then $T = T'$ and $\ell \leq \ell'$ (or $\ell' \leq \ell$). Moreover if $v \neq dec(v_1, v_2), dec^a(v_1, v_2)$ we also have

1) *if $\ell = HH$, then $\ell' = HH$ and $\mu = \mu'$;*
2) *if $\ell \neq HH$ then $T = T' = LL$.*

We can now define when a typing-environment Γ, a well-formedness function Θ, and a map σ from types to ground-types are correct with respect to a particular memory M, a handle-map H, and a set of atomic values V. In short, this definition requires that memory cells of type T record values with (a ground) run-time type $T\sigma$; the handle-map associates properly the values with their correspondent type; and that the values in V are recorded in Θ with their exact type, and not a subtype of it. This last property is important in our main theorem as we want to be sure that the generated keys are recorded in Θ with their appropriated type and not a subtype of it. This way we will be able to construct a Γ that will record the minimum type that a value needs to have and distinguish well generated keys from arbitrarily generated ones.

Definition 13 (Well-formedness)**.** $\Gamma, \Theta, \sigma \vdash \mathsf{M}, \mathsf{H}, \mathsf{V}$ *if*

- $\Gamma, \Theta, \sigma \vdash_{\mathsf{M}} \mathsf{M}$*, i.e.,* $\mathsf{M}(x) = v$*,* $\Gamma(x) = T$ *implies* $\Theta \models_v v : T\sigma$*; and*
- $\Theta \models_{\mathsf{H}} \mathsf{H}$*, i.e.,* $\mathsf{H}(v') = (v, T)$ *implies* $\Theta \models_v v : T$*;*
- $\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}$*, i.e.,* $val \in \mathsf{V}$ *then* $\exists g.\mathsf{H}(g) = (val, T)$ *and* $\Theta(val) = T$.

We can easily show that all values typed by Θ are ground, and so it follows from the previous definition that σ is such that all the $T\sigma$ above are ground.

Having defined the properties of run-time values for keys and the notion of well-formed memory and handle-maps we can now characterize which values are derivable by an adversary. We show that with the attacker model defined in Section II, given an initial set of values of type LL, the attacker can only derive values of type LL. Intuitively, having type LL, or LH via subtyping, is a necessary condition for a well-formed value to be deducible by the attacker.

Proposition 14. *Let Θ be a well-formedness mapping and V be a set of values such that $\Theta \models_v v : LL$ for all $v \in V$.*

Then, $v \in \mathcal{K}(V)$ implies $\Theta \models_v v : LL$.

$$\frac{\Theta(val) = T}{\Theta \models_v val : T}\ [\text{atom}] \qquad \frac{\Theta \models_v v : T' \quad T' \leq T}{\Theta \models_v v : T}\ [\text{sub}]$$

$$\frac{\Theta \models_v v : \mathsf{DecK}^{\ell_C \ell_I}[T]}{\Theta \models_v ek(v) : \mathsf{EncK}^{L\ell_I}[T]}\ [\text{ek}] \qquad \frac{\Theta \models_v v : \mathsf{SigK}^{\ell_C \ell_I}[T]}{\Theta \models_v vk(v) : \mathsf{VerK}^{L\ell_I}[T]}\ [\text{vk}]$$

$$\frac{\Theta \models_v v : \mathsf{SymK}^{\ell_C \ell_I}[T] \quad \Theta \models_v v' : T}{\Theta \models_v enc(v', v) : L\ell_I}\ [\text{enc}] \qquad \frac{\Theta \models_v v : \mathsf{SymK}^{\ell}[T] \quad \Theta \models_v v' : T' \quad v' \neq enc(v'', v)}{\Theta \models_v dec(v', v) : T}\ [\text{dec}]$$

$$\frac{\Theta \models_v v : \mathsf{EncK}^{\ell_C \ell_I}[T] \quad \Theta \models_v v' : T}{\Theta \models_v enc^a(v', v) : L\ell_I}\ [\text{enca}] \qquad \frac{\Theta \models_v v : \mathsf{SigK}^{\ell_C \ell_I}[T] \quad \Theta \models_v v' : T}{\Theta \models_v sig(v', v) : \ell_C(T)\ell_I}\ [\text{sig}]$$

$$\frac{\Theta \models_v v : \mathsf{DecK}^{\ell}[T] \quad \Theta \models_v v' : T' \quad v' \neq enc^a(v'', ek(v)) \quad \ell_I(T') \neq H \implies T = LL}{\Theta \models_v dec^a(v', v) : T}\ [\text{deca}]$$

Table V
VALUE WELL-FORMEDNESS

It is important that the type of expressions and the type of their corresponding values are consistent at runtime. The next Proposition states that when evaluating an expression with type T in a well-formed memory, the type of the returned value is $T\sigma$. Recall that the range of Θ are only the ground types whereas the range of Γ are all types. We thus need to have a map σ that accounts for this.

Proposition 15. *Let $\Gamma \vdash_e e : T$, $e \downarrow^{\mathsf{M}} v$, Θ a well-formedness function and σ a map from types to ground types. If $\Gamma, \Theta, \sigma \vdash_{\mathsf{M}} \mathsf{M}$ then it holds $\Theta \models_v v : T\sigma$.*

We are now ready to prove our subject-reduction Theorem that states that well-typed programs remain well-typed at run-time and preserve memory and handle-map well-formedness. We also have that all the atomic values associated with new-handles, and that were not already in memory, are recorded in Θ with their exact type.

Theorem 16. *Let $\Gamma, \Theta, \sigma \vdash \mathsf{M}, \mathsf{H}, \mathsf{V}$ and $\Gamma \vdash_{\mathsf{c}} \mathsf{c}$. If $\langle \mathsf{M}, \mathsf{H}, \sigma, \mathsf{c} \rangle \to \langle \mathsf{M}', \mathsf{H}', \sigma', \mathsf{c}' \rangle$ then*

(i) *if $\mathsf{c}' \neq \varepsilon$ then $\Gamma \vdash_{\mathsf{c}} \mathsf{c}'$;*

(ii) *$\exists \Theta' \supseteq \Theta$ such that $\Gamma, \Theta', \sigma' \vdash \mathsf{M}', \mathsf{H}', \mathsf{V}'$, where $\mathsf{V}' = \mathsf{V} \cup \{val \mid \exists g \in dom(\mathsf{H}') \setminus dom(\mathsf{H}).\mathsf{H}'(g) = (val, T)\} \setminus ran[\mathsf{M}]$.*

Proof: This proof can be found in the Appendix. ∎

We can now show that one can construct a Θ that types all the values known to the adversary as LL while at the same time typing all the values not known for the adversary with their exact type.

Let $\mathsf{V}_{\mathsf{ok}}(\mathsf{H}, V) = \{val \mid \exists g.\mathsf{H}(g) = (val, T)\} \setminus \mathcal{K}(V)$.

Lemma 17. *Let $\Gamma \vdash_{\mathsf{c}} \mathcal{A}$ and $\langle \mathsf{H}_0, V_0 \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}, V \rangle$.*

Then, there exists Θ such that $\Theta \models_{\mathsf{H}} \mathsf{H}$, $\Theta \models_v v : LL$ for each $v \in V$, and $\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}_{\mathsf{ok}}(\mathsf{H}, V)$.

Proof: This proof can be found in the Appendix. ∎

Lemma 18. *Let $\Gamma \vdash_{\mathsf{c}} \mathcal{A}$ and $\langle \mathsf{H}_0, V_0 \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}, V \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}', V' \rangle$. Then, there exists Θ, Θ' with $\Theta \subseteq \Theta'$ such that*

- *$\Theta \models_{\mathsf{H}} \mathsf{H}$ and $\Theta \models_v v : LL$ for each $v \in V$, and*
- *$\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}_{\mathsf{ok}}(\mathsf{H}, V)$*

and

- *$\Theta' \models_{\mathsf{H}} \mathsf{H}'$ and $\Theta' \models_v v : LL$ for each $v \in V'$, and*
- *$\Theta' \Vdash_{\mathsf{V}} \mathsf{H}', \mathsf{V}_{\mathsf{ok}}(\mathsf{H}', V')$*

Proof: Direct from the Lemma 17 ∎

We can now state the main result of the paper: well-typed APIs are secure, according to Definition 6.

Theorem 19. *Let $\Gamma \vdash_{\mathsf{c}} \mathcal{A}$. Then $\mathcal{A}$ is secure.*

Proof: Suppose that $\langle \mathsf{H}_0, V_0 \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}, V \rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}', V' \rangle$ and val is an atomic value confidential in H, that is, for all g where $\mathsf{H}(g) = (val, T)$ then $T = H\ell_I$ or $T = \mu\mathsf{K}^{H\ell_I}[T^\star]$.

By Lemma 18 one has that there exists $\Theta \subseteq \Theta'$ such that

- $\Theta \models_{\mathsf{H}} \mathsf{H}$ and $\Theta \models_v v : LL$ for each $v \in V$, and
- $\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}_{\mathsf{ok}}(\mathsf{H}, V)$
- $\Theta' \models_{\mathsf{H}} \mathsf{H}'$ and $\Theta' \models_v v : LL$ for each $v \in V'$, and
- $\Theta' \Vdash_{\mathsf{V}} \mathsf{H}', \mathsf{V}_{\mathsf{ok}}(\mathsf{H}', V')$

Since val is in the handle-map H and by hypothesis $val \notin \mathcal{K}(V)$ we have that $val \in \mathsf{V}_{\mathsf{ok}}(\mathsf{H}, V)$. Now since $\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}_{\mathsf{ok}}(\mathsf{H}, V)$ we have that $\exists g.\mathsf{H}(g) = (val, T)$ and $\Theta(val) = T$. Since val is confidential we have that $\Theta(val) = \Theta'(val) = H\ell_I$ or $\mu\mathsf{K}^{H\ell_I}[T^\star]$ which imply by Proposition 10 that $\Theta' \not\models_v val : LL$ (otherwise $H\ell_I \leq LL$ or $\mu\mathsf{K}^{H\ell_I}[T^\star] \leq LL$). Applying now Proposition 14 one gets $val \notin \mathcal{K}(V')$.

Suppose now that val is an atomic value secure in H, that is, for all g where $\mathsf{H}(g) = (val, T)$ then $T = \mu\mathsf{K}^{HH}[T^\star]$. Then by $\Theta \models_{\mathsf{H}} \mathsf{H}$ we have $\Theta \models_v val : \mu\mathsf{K}^{HH}[T^\star]$. By

Proposition 10 and definition of $\leq$ we have that $\Theta(val) = \mu\mathsf{K}^{HH}[T^\star]$.

Now, one can see that $val \notin \mathcal{K}(V)$ otherwise we would have by Proposition 14 $\Theta \models_v val : LL$ which is not possible by Proposition 10.

We now apply the same reasoning as in the first case to conclude that $val \notin \mathcal{K}(V')$. ∎

IV. Secure implementation

We now modify the language in order to get closer to realistic implementations of the APIs. So far, we have assumed that keys are typed and types are stored in the devices together with the key values. This abstraction allows to statically prove security but needs to be related to actual APIs implementation in order to be useful. To this aim, we give a new semantics in which keys are stored together with *key properties*, i.e. concrete data which specify the roles of the key, its security level, the cryptographic algorithm, the key length, etc.

We will show that that if we assign types to key properties in a unique way then the concrete semantics is mimicked by the typed semantics and so the security results can be carried over to this new concrete semantics. In this way we are able to deal with concrete examples while enjoying the properties of the formalism proposed in the previous sections.

Key properties: Properties of keys have the following syntax:

$$P ::= Y \mid \epsilon \mid p[P]$$

where Y is a property variable that will be bound at runtime, ϵ means no properties, $p[P]$ represents a key with properties p which can perform cryptographic operations on keys with properties P. We use $p[\epsilon]$, also denoted by p, to represent keys with properties p that do not operate on other keys, e.g., keys used to encrypt data.

It may happen that different concrete properties are treated the same when authorizing cryptographic operations. For example, an encryption key may be allowed to perform encryption independently of its actual length or of the algorithm it is bound to. These details are important when dealing with actual cryptography but should be irrelevant for our analysis. To encompass this, we assume to have an equivalence relation $\equiv$ on concrete properties that relates properties which make the APIs behave the same way.

Concrete syntax and semantics: We define a new syntax which stores concrete key properties rather than types. The only commands that are affected are the internal functions:

$$\mathsf{f} \quad ::= \quad \mathsf{getKey}(y, P) \mid \mathsf{genKey}(P) \mid \mathsf{setKey}(y, P)$$

The new semantics is presented in Table VI and is close to the one of Table II. There are however some important differences: $\mathsf{H_p}$ denotes the new concrete handle map that stores actual key properties instead of types; ρ is a substitution of key property variables into key properties; all occurrences of types T are replaced by key properties P; and in getKey, when we match properties, we also allow matching of equivalent key properties. Finally, we limit the properties of newly generated keys to a predefined set G.

Attacker configurations for a concrete API $\mathcal{A}_p = \{\mathsf{a}_1, \ldots, \mathsf{a}_n\}$ evolves by making calls on the concrete semantics:

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

In order to be able to study key properties in our general formalism, we need to ensure that they satisfy some conditions. We state a few conditions that allow us to "embed" key properties into types.

Definition 20 (Typed key properties)**.** *We say that key properties are typed if there exists a mapping $\mathcal{T}$ from key properties to types such that*

(1) $P_1 \equiv P_2$ implies $\mathcal{T}(P_1) = \mathcal{T}(P_2)$;

(2) $\mathcal{T}(P\rho) = \mathcal{T}(P)\mathcal{T}(\rho)$ for all substitutions ρ, where $\mathcal{T}(\rho)$ is defined as $\mathcal{T}(\rho)(\mathcal{T}(Y)) = \mathcal{T}(\rho(Y))$ for each $Y \in dom(\rho)$;

(3) $var(\mathcal{T}(P)) = \mathcal{T}(var(P))$, where $\mathcal{T}(\{Y_1, \ldots Y_n\}) = \{\mathcal{T}(Y_1), \ldots, \mathcal{T}(Y_n)\}$;

(4) whenever $P \in G$ and $\mathcal{T}(P) = \mu\mathsf{K}^\ell[T']$ then $\mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\} \wedge (\ell = HH \vee T' = LL)$.

Notice that the definition of $\mathcal{T}(\rho)$ implicitly assumes that $\mathcal{T}$ maps different property variables into different type variables. This also implies that $\mathcal{T}(\rho \uplus \rho') = \mathcal{T}(\rho) \uplus \mathcal{T}(\rho')$.

Typing of key properties is extended to handles and API commands by simply applying it to all occurrences of key properties. Formally:

Definition 21 (Connecting concrete and typed semantics)**.** *Given a mapping $\mathcal{T}$ from key properties to types we apply it to handles and API commands as follows:*

$$\mathcal{T}(\mathsf{H_p})(v) = (v', \mathcal{T}(P)) \text{ whenever } \mathsf{H_p}(v) = (v', P)$$

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

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

$$\frac{\mathsf{H_p}(\mathsf{M}(y)) = (v, P') \qquad P' \equiv (P\rho)\rho' \qquad \mathit{dom}(\rho') = \mathit{var}(P\rho)}{\langle \mathsf{M}, \mathsf{H_p}, \rho, x := \mathsf{getKey}(y, P)\rangle \rightarrow_p \langle \mathsf{M}[x \mapsto v], \mathsf{H_p}, \rho \uplus \rho', \varepsilon\rangle}$$

$$\frac{g, g' \leftarrow \mathcal{G} \qquad P\rho \text{ ground} \qquad P\rho \in G}{\langle \mathsf{M}, \mathsf{H_p}, \rho, x := \mathsf{genKey}(P)\rangle \rightarrow_p \langle \mathsf{M}[x \mapsto g], \mathsf{H_p}[g \mapsto (g', P\rho)], \rho, \varepsilon\rangle}$$

$$\frac{g \leftarrow \mathcal{G} \qquad P\rho \text{ ground}}{\langle \mathsf{M}, \mathsf{H_p}, \rho, x := \mathsf{setKey}(y, P)\rangle \rightarrow_p \langle \mathsf{M}[x \mapsto g], \mathsf{H_p}[g \mapsto (\mathsf{M}(y), P\rho)], \rho, \varepsilon\rangle}$$

$$\frac{\langle \mathsf{M}, \mathsf{H_p}, \rho, \mathsf{c}_1\rangle \rightarrow_p \langle \mathsf{M}', \mathsf{H_p}', \rho', \varepsilon\rangle}{\langle \mathsf{M}, \mathsf{H_p}, \rho, \mathsf{c}_1; \mathsf{c}_2\rangle \rightarrow_p \langle \mathsf{M}', \mathsf{H_p}', \rho', \mathsf{c}_2\rangle} \qquad \frac{\langle \mathsf{M}, \mathsf{H_p}, \rho, \mathsf{c}_1\rangle \rightarrow_p \langle \mathsf{M}', \mathsf{H_p}', \rho', \mathsf{c}_1'\rangle}{\langle \mathsf{M}, \mathsf{H_p}, \rho, \mathsf{c}_1; \mathsf{c}_2\rangle \rightarrow_p \langle \mathsf{M}', \mathsf{H_p}', \rho', \mathsf{c}_1'; \mathsf{c}_2\rangle}$$

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

Table VI
API CONCRETE SEMANTICS

Notice that expressions e are not affected by $\mathcal{T}$ as they do not contain occurrences of properties. We can now prove that each step in the concrete semantics is mimicked by the typed semantics.

Theorem 22 (Semantic correspondence)**.** *Let $\mathcal{T}$ be a typing for key properties, and assume that the two semantics use an identical generator of fresh values $\mathcal{G}$. Then,*

$$\langle \mathsf{M}, \mathsf{H_p}, \rho, \mathsf{c}\rangle \rightarrow_p \langle \mathsf{M}', \mathsf{H_p}', \rho', \mathsf{c}'\rangle$$

implies

$$\langle \mathsf{M}, \mathcal{T}(\mathsf{H_p}), \mathcal{T}(\rho), \mathcal{T}(\mathsf{c})\rangle \rightarrow \langle \mathsf{M}', \mathcal{T}(\mathsf{H_p}'), \mathcal{T}(\rho'), \mathcal{T}(\mathsf{c}')\rangle$$

Proof: By induction on the length of the derivation of the reduction, applying Definitions 20 and 21. All assignments are base cases.

Case $x := e$. Since translation $\mathcal{T}$ does not apply to expressions and expressions do not access $\mathsf{H_p}$, this case is trivially proved. In fact, $\langle \mathsf{M}, \mathsf{H_p}, \rho, x := e\rangle \rightarrow_p \langle \mathsf{M}[x \mapsto v], \mathsf{H_p}, \rho, \varepsilon\rangle$ requires $e \downarrow^{\mathsf{M}} v$ which in turns implies

$$\langle \mathsf{M}, \mathsf{H}, \sigma, x := e\rangle \rightarrow \langle \mathsf{M}[x \mapsto v], \mathsf{H}, \sigma, \varepsilon\rangle$$

for all H and σ. By Definition 21 we obtain:

$$\langle \mathsf{M}, \mathcal{T}(\mathsf{H_p}), \mathcal{T}(\rho), \mathcal{T}(x := e)\rangle \rightarrow \langle \mathsf{M}[x \mapsto v], \mathcal{T}(\mathsf{H_p}), \mathcal{T}(\rho), \mathcal{T}(\varepsilon)\rangle$$

Case $x := \mathsf{getKey}(y, P)$. This case is more interesting as it accesses $\mathsf{H_p}$ and the command can refer to properties. $\langle \mathsf{M}, \mathsf{H_p}, \rho, x := \mathsf{getKey}(y, P)\rangle \rightarrow_p \langle \mathsf{M}[x \mapsto v], \mathsf{H_p}, \rho \uplus \rho', \varepsilon\rangle$ requires $\mathsf{H_p}(\mathsf{M}(y)) = (v, P')$, $P' \equiv (P\rho)\rho'$ and $\mathit{dom}(\rho') = \mathit{var}(P\rho)$ which implies $\mathcal{T}(\mathit{dom}(\rho')) = \mathcal{T}(\mathit{var}(P\rho))$.

Let $\mathcal{T}(P) = T$, $\mathcal{T}(P') = T'$, $\mathcal{T}(\rho) = \sigma$ and $\mathcal{T}(\rho') = \sigma'$. By Definition 20(2) applied twice, we have that $\mathcal{T}((P\rho)\rho') = (T\sigma)\sigma'$ Then, $\mathcal{T}(\mathsf{H_p})(\mathsf{M}(y)) = (v, T')$ and, by Definition 20(1) $T' = (T\sigma)\sigma'$. By Definition 20(2-3) we have $\mathcal{T}(\mathit{var}(P\rho)) = \mathit{var}(\mathcal{T}(P)\mathcal{T}(\rho)) = \mathit{var}(T\sigma)$. By definition of $\mathcal{T}(\rho)$ we also know that $\mathit{dom}(\mathcal{T}(\rho')) = \mathcal{T}(\mathit{dom}(\rho'))$. Thus, $\mathit{dom}(\sigma') = \mathit{dom}(\mathcal{T}(\rho')) = \mathcal{T}(\mathit{dom}(\rho')) = \mathcal{T}(\mathit{var}(P\rho)) = \mathit{var}(\mathcal{T}(P)\mathcal{T}(\rho)) = \mathit{var}(T\sigma)$. Together with $\mathcal{T}(\mathsf{H_p})(\mathsf{M}(y)) = (v, T')$ and $T' = (T\sigma)\sigma'$ this is enough to derive

$$\langle \mathsf{M}, \mathcal{T}(\mathsf{H_p}), \sigma, x := \mathsf{getKey}(y, T)\rangle \rightarrow \langle \mathsf{M}[x \mapsto v], \mathcal{T}(\mathsf{H_p}), \sigma \uplus \sigma', \varepsilon\rangle$$

which in turn gives

$$\langle \mathsf{M}, \mathcal{T}(\mathsf{H_p}), \mathcal{T}(\rho), \mathcal{T}(x := \mathsf{getKey}(y, P))\rangle \rightarrow \langle \mathsf{M}[x \mapsto v], \mathcal{T}(\mathsf{H_p}), \mathcal{T}(\rho \uplus \rho'), \mathcal{T}(\varepsilon)\rangle$$

Case $x := \mathsf{genKey}(P)$. We have that

$$\langle \mathsf{M}, \mathsf{H_p}, \rho, x := \mathsf{genKey}(P)\rangle \rightarrow_p \langle \mathsf{M}[x \mapsto g], \mathsf{H_p}[g \mapsto (g', P\rho)], \rho, \varepsilon\rangle$$

requires $g, g' \leftarrow \mathcal{G}$, $P\rho$ ground, i.e., $\mathit{var}(P\rho) = \emptyset$, and $P\rho \in G$. Since we are assuming to use the same generator $\mathcal{G}$ we will get the same g and g' in the typed semantics. We let $T = \mathcal{T}(P)$ and $\sigma = \mathcal{T}(\rho)$. By Definition 20(2) we get $\mathcal{T}(P\rho) = \mathcal{T}(P)\mathcal{T}(\rho) = T\sigma$. By Definition 20(3) we now have $\mathit{var}(T\sigma) = \mathcal{T}(\mathit{var}(P\rho)) = \emptyset$, i.e., $T\sigma$ is ground. Moreover, if $\mathcal{T}(P\rho) = T\sigma = \mu\mathsf{K}^{\ell}[T']$, since $P\rho \in G$ by Definition 20(4) we have $\mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\} \wedge (\ell = HH \vee T' = LL)$. We can thus derive:

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

which is

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

for each H. Now notice that by Definition 21 we have $\mathcal{T}(\mathsf{H_p}[g \mapsto (g', P\rho)]) = \mathcal{T}(\mathsf{H_p})[g \mapsto (g', \mathcal{T}(P\rho))]$. Thus we can pick $\mathsf{H} = \mathcal{T}(\mathsf{H_p})$ and get the thesis:

$$\langle \mathsf{M}, \mathcal{T}(\mathsf{H_p}), \mathcal{T}(\rho), \mathcal{T}(x := \mathsf{genKey}(T)) \rangle \\ \rightarrow \langle \mathsf{M}[x \mapsto g], \mathcal{T}(\mathsf{H_p}[g \mapsto (g', P\rho)]), \mathcal{T}(\rho), \mathcal{T}(\varepsilon) \rangle$$

Case $x := \mathsf{setKey}(y, P)$. This case is proved exactly as the previous one. The only difference is that we have $\mathsf{M}(y)$ in place of g'. Since the two semantics work on the same memory M this does not affect the proof.

Inductive cases. We have two rules for sequential composition.

Rule 1. We have $\langle \mathsf{M}, \mathsf{H_p}, \rho, \mathsf{c}_1; \mathsf{c}_2 \rangle \rightarrow_p \langle \mathsf{M}', \mathsf{H_p}', \rho', \mathsf{c}_2 \rangle$ because of $\langle \mathsf{M}, \mathsf{H_p}, \rho, \mathsf{c}_1 \rangle \rightarrow_p \langle \mathsf{M}', \mathsf{H_p}', \rho', \varepsilon \rangle$. By induction

$$\langle \mathsf{M}, \mathcal{T}(\mathsf{H_p}), \mathcal{T}(\rho), \mathcal{T}(\mathsf{c}_1) \rangle \rightarrow \langle \mathsf{M}', \mathcal{T}(\mathsf{H_p}'), \mathcal{T}(\rho'), \mathcal{T}(\varepsilon) \rangle$$

Since $\mathcal{T}(\varepsilon) = \varepsilon$ and $\mathcal{T}(\mathsf{c}_1; \mathsf{c}_2) = \mathcal{T}(\mathsf{c}_1); \mathcal{T}(\mathsf{c}_2)$ we get

$$\langle \mathsf{M}, \mathcal{T}(\mathsf{H_p}), \mathcal{T}(\rho), \mathcal{T}(\mathsf{c}_1; \mathsf{c}_2) \rangle \\ \rightarrow \langle \mathsf{M}', \mathcal{T}(\mathsf{H_p}'), \mathcal{T}(\rho'), \mathcal{T}(\mathsf{c}_2) \rangle$$

Rule 2 is proved analogously. ■

As a consequence of the previous theorem we have that all the attacks in the concrete semantics are mimicked in the typed one. Given $\mathcal{A}_p = \{\mathsf{a}_1, \ldots, \mathsf{a}_n\}$ we let $\mathcal{T}(\mathcal{A}_p) = \{\mathcal{T}(\mathsf{a}_1), \ldots, \mathcal{T}(\mathsf{a}_n)\}$.

Theorem 23. *Let* $\langle \mathsf{H_p}, V \rangle \rightsquigarrow^*_{\mathcal{A}_p} \langle \mathsf{H}'_\mathsf{p}, V' \rangle$. *Then we have* $\langle \mathcal{T}(\mathsf{H_p}), V \rangle \rightsquigarrow^*_{\mathcal{T}(\mathcal{A}_p)} \langle \mathcal{T}(\mathsf{H}'_\mathsf{p}), V' \rangle$

Proof: By induction on the number of steps in $\rightsquigarrow^*_{\mathcal{A}_p}$. Base case is length 0 with $\mathsf{H}'_\mathsf{p} = \mathsf{H_p}$ and $V' = V$, and there is nothing to prove.

Now let $\langle \mathsf{H_p}, V \rangle \rightsquigarrow^*_{\mathcal{A}_p} \langle \hat{\mathsf{H}}_\mathsf{p}, \hat{V} \rangle \rightsquigarrow_{\mathcal{A}_p} \langle \mathsf{H}'_\mathsf{p}, V' \rangle$. By induction we have $\langle \mathcal{T}(\mathsf{H_p}), V \rangle \rightsquigarrow^*_{\mathcal{T}(\mathcal{A}_p)} \langle \mathcal{T}(\hat{\mathsf{H}}_\mathsf{p}), \hat{V} \rangle$. We now prove that $\langle \mathcal{T}(\hat{\mathsf{H}}_\mathsf{p}), \hat{V} \rangle \rightsquigarrow_{\mathcal{T}(\mathcal{A}_p)} \langle \mathcal{T}(\mathsf{H}'_\mathsf{p}), V' \rangle$ which give the thesis $\langle \mathcal{T}(\mathsf{H_p}), V \rangle \rightsquigarrow^*_{\mathcal{T}(\mathcal{A}_p)} \langle \mathcal{T}(\mathsf{H}'_\mathsf{p}), V' \rangle$.

We have that $\langle \hat{\mathsf{H}}_\mathsf{p}, \hat{V} \rangle \rightsquigarrow_{\mathcal{A}_p} \langle \mathsf{H}'_\mathsf{p}, V' \rangle$ requires $\mathsf{a}(v_1, \ldots, v_k) \downarrow_p^{\hat{\mathsf{H}}_\mathsf{p}, \mathsf{H}'_\mathsf{p}} v$ with $\mathsf{a} \in \mathcal{A}_p$ and $v_1, \ldots, v_k \in \mathcal{K}(\hat{V})$. By Table VI we know that $\mathsf{a} = \lambda x_1, \ldots, x_k.\mathsf{c}$, $e \downarrow^{\mathsf{M}'} v$ and $\langle \mathsf{M}_\epsilon[x_1 \mapsto v_1 \ldots x_k \mapsto v_k], \hat{\mathsf{H}}_\mathsf{p}, \emptyset, \mathsf{c} \rangle \rightarrow_p \langle \mathsf{M}', \mathsf{H}'_\mathsf{p}, \rho', \mathsf{return}\ e \rangle$. By Theorem 22 we obtain

$$\langle \mathsf{M}_\epsilon[x_1 \mapsto v_1 \ldots x_k \mapsto v_k], \mathcal{T}(\hat{\mathsf{H}}_\mathsf{p}), \emptyset, \mathcal{T}(\mathsf{c}) \rangle \\ \rightarrow \langle \mathsf{M}', \mathcal{T}(\mathsf{H}'_\mathsf{p}), \mathcal{T}(\rho'), \mathsf{return}\ e \rangle$$

which gives $\mathcal{T}(\mathsf{a})(v_1, \ldots, v_k) \downarrow^{\mathcal{T}(\hat{\mathsf{H}}_\mathsf{p}), \mathcal{T}(\mathsf{H}'_\mathsf{p})} v$. Since, by definition, $\mathcal{T}(\mathsf{a}) \in \mathcal{T}(\mathcal{A}_p)$ we obtain the thesis $\langle \mathcal{T}(\hat{\mathsf{H}}_\mathsf{p}), \hat{V} \rangle \rightsquigarrow_{\mathcal{T}(\mathcal{A}_p)} \langle \mathcal{T}(\mathsf{H}'_\mathsf{p}), V \cup \{v\} \rangle$ ■

We can now define security of concrete APIs through the type associated to concrete key properties:

Definition 24 (Concrete API Security)**.** *Let $\mathcal{A}_p$ be a concrete API. We say that $\mathcal{A}_p$ is secure if for all reductions $\langle \mathsf{H}_0, V_0 \rangle \rightsquigarrow^*_{\mathcal{A}_p} \langle \mathsf{H_p}, V \rangle \rightsquigarrow^*_{\mathcal{A}_p} \langle \mathsf{H}'_\mathsf{p}, V' \rangle$ and for all atomic values val we have*

(1) $val \notin \mathcal{K}(V)$ and val is confidential in $\mathcal{T}(\mathsf{H_p})$ implies $val \notin \mathcal{K}(V')$;

(2) val is secure in $\mathcal{T}(\mathsf{H_p})$ implies $val \notin \mathcal{K}(V) \cup \mathcal{K}(V')$.

Thus, all security results hold on the concrete semantics based on actual key properties once an appropriate typing $\mathcal{T}$ is provided.

Theorem 25. *Let $\Gamma \vdash_\mathsf{c} \mathcal{T}(\mathcal{A}_p)$. Then $\mathcal{A}_p$ is secure.*

Proof: Consider $\langle \mathsf{H}_0, V_0 \rangle \rightsquigarrow^*_{\mathcal{A}_p} \langle \mathsf{H_p}, V \rangle \rightsquigarrow^*_{\mathcal{A}_p} \langle \mathsf{H}'_\mathsf{p}, V' \rangle$. By Theorem 23 we have $\langle \mathsf{H}_0, V_0 \rangle \rightsquigarrow^*_{\mathcal{T}(\mathcal{A}_p)} \langle \mathcal{T}(\mathsf{H_p}), V \rangle \rightsquigarrow^*_{\mathcal{T}(\mathcal{A}_p)} \langle \mathcal{T}(\mathsf{H}'_\mathsf{p}), V' \rangle$. Now it is enough to observe that the requirements of Definition 6 (API Security) on this latter reduction are exactly the same as the requirements of Definition 24 (Concrete API Security) on the former reduction. By Theorem 19 we obtain that such requirements hold for each reduction, from which we obtain the thesis. ■

V. Case study: PKCS#11 v2.20

PKCS#11, also known as Cryptoki, defines a widely adopted API for cryptographic tokens [19]. It provides access to cryptographic functionalities while, in principle, satisfying some security properties. One such property is that the value of keys stored on a PKCS#11 device and tagged as *sensitive* should never be revealed outside the token, even when connected to a compromised host. Unfortunately, PKCS#11 is known to be vulnerable to attacks that break this property [4], [9], [12]. In this section we will see that we can encode PKCS#11 attributes as key properties and hence study its security using Theorem 25.

A token may store many different *objects* such as cryptographic keys and certificates, and these objects are referenced via *handles*. The value of a key is one of the *attributes* of the enclosing object but 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.

PKCS#11 key properties: Properties and capabilities of keys are described by a set of *attributes*. When a certain attribute is contained in the set of key properties p we will say that the attribute is set, otherwise we say that it is unset. In our analysis we consider the following subset of PKCS#11 attributes:

`CKA_CLASS` (C) The object class which can be one among

`CKO_PUBLIC_KEY` ($PubK$) Public keys;

`CKO_PRIVATE_KEY` ($PrivK$) Private keys;
`CKO_SECRET_KEY` ($SecK$) Secret (symmetric) keys;

`CKA_SENSITIVE` (H) The key should never be revealed outside of the token;

`CKA_ENCRYPT` (E) The key can be used to encrypt data;

`CKA_DECRYPT` (D) The key can be used to decrypt data;

`CKA_SIGN` (S) The key supports signature;

`CKA_VERIFY_RECOVER` (V) The key can be used to verify signatures, recovering data from the signature;

`CKA_WRAP` (W) The key can be used to wrap another key stored in the token;

`CKA_UNWRAP` (U) The key can be used to unwrap a key and import it in the token;

`CKA_WRAP_TEMPLATE` For wrapping keys (W set) specifies the attributes of any wrapped key. It is the P component of $p[P]$;

`CKA_UNWRAP_TEMPLATE` For unwrapping keys (U set) specifies the attributes of unwrapped key. For simplicity, we will assume that wrap and unwrap templates coincide.

Example 26. *The key property $p = \{PubK, E\}$ represents a public key that can be used to encrypt data. The key property $p' = \{H, PrivK, U\}[\{H, SecK, E\}]$, instead, represents a private (sensitive) unwrapping key that can be used to import symmetric (sensitive) encryption keys.*

From properties to types: We now define the mapping $\mathcal{T}_{p11}$ of PKCS#11 key properties into types. This mapping follows the informal description of attributes. For example, whenever E and $SecK$ are in p the key is typed as a symmetric key for encrypting data. Notice that this will force us to reduce the possible attribute assignments to sets with no *conflicting* attributes. For example, a key with $W, D, SecK$ set is dangerous as it can be used to wrap a sensitive key and then decrypt it as if it were simple data, leaking it outside the token. PKCS#11 is very flexible and allows for insecure operations, such as encrypting data under symmetric keys that are not sensitive and thus readable from anyone. We will discipline this more, by requiring that sensitive is always set.

The formal definition of $\mathcal{T}_{p11}$ is given in Table VII. Property ϵ is translated into LL, variables Y are translated into distinct type variables X_Y, and properties $p[P]$ are translated as follows: at each layer we specify the attributes to inspect in p. For example we first split depending on sensitive (H); then we inspect the class and so on. If the set of attributes match exactly one line of the table we have the corresponding type. If none or more than one match, we have no type and $\mathcal{T}_{p11}$ is undefined. Notice that for data keys like $\mathsf{DecK}^{HL}[LL]$, we only admit cryptographic operations on terms of type LL, which corresponds to requiring that $P = \emptyset$. Thus, in these cases $p[P] = p[\epsilon] = p$.

Example 27. *Consider again the key property $p = \{PubK, E\}$ representing a public key that can be used to encrypt data. It matches $\neg H, PubK, E$ in the table*

$$
\begin{array}{l}
\mathcal{T}_{p11}(\epsilon) = LL \\
\mathcal{T}_{p11}(Y) = X_Y \\
\mathcal{T}_{p11}(p[P]) = \\
= \left\{ \begin{array}{lll}
H & \left\{ \begin{array}{ll}
PrivK & \left\{ \begin{array}{ll} D & \mathsf{DecK}^{HL}[LL] \text{ with } P = \epsilon \\ U & \mathsf{DecK}^{HH}[\mathcal{T}_{p11}(P)] \\ S & \mathsf{SigK}^{HH}[\mathcal{T}_{p11}(P)] \end{array} \right. \\
SecK & \left\{ \begin{array}{ll} E, D & \mathsf{SymK}^{HL}[LL] \text{ with } P = \epsilon \\ W, U & \mathsf{SymK}^{HH}[\mathcal{T}_{p11}(P)] \end{array} \right. \\
\neg C & HL \text{ with } P = \epsilon
\end{array} \right. \\
\neg H & \left\{ \begin{array}{ll}
PubK & \left\{ \begin{array}{ll} E & \mathsf{EncK}^{LL}[LL] \text{ with } P = \epsilon \\ W & \mathsf{EncK}^{LH}[\mathcal{T}_{p11}(P)] \\ V & \mathsf{VerK}^{LH}[\mathcal{T}_{p11}(P)] \end{array} \right. \\
\neg C & LL \text{ with } P = \epsilon
\end{array} \right.
\end{array} \right.
\end{array}
$$

Table VII
DEFINITION OF $\mathcal{T}_{p11}$

giving type $\mathsf{EncK}^{LL}[LL]$, as expected. Key property $p' = \{H, PrivK, U\}[\{H, SecK, E\}]$ instead, represents a private (sensitive) unwrapping key that can be used to import symmetric (sensitive) encryption keys. From the table we get type $\mathsf{DecK}^{HH}[\mathcal{T}_{p11}(P)]$ with $P = \{H, SecK, E\}$ which, in turns, gives $\mathsf{DecK}^{HH}[\mathsf{SymK}^{HL}[LL]]$.

Proving security: We define the equivalence relation $\equiv_{p11}$ over key properties by simply equating properties that are mapped to the same types, i.e.,

$$P \equiv_{p11} P' \text{ iff } \mathcal{T}_{p11}(P) = \mathcal{T}_{p11}(P')$$

For example, $\{H, S\} \equiv \{H, PrivK, S\}$, since H and S are enough to univocally identify a private signature key. Moreover, if we consider an extra attribute A that is not relevant in our analysis we have that its addition does not affect the semantics, i.e., $p \cup \{A\} \equiv_{p11} p$ since $\mathcal{T}_{p11}(p \cup \{A\}) = \mathcal{T}_{p11}(p)$.

Proposition 28. *Let $G = \{P \mid \mathcal{T}(P) = \mu\mathsf{K}^{\ell}[T']$ implies $\mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}\}$. Then, $\mathcal{T}_{p11}$ respects Definition 20.*

Proof: We prove the three items of the definition.

(1) There is nothing to prove as $P \equiv_{p11} P'$ implies $\mathcal{T}_{p11}(P) = \mathcal{T}_{p11}(P')$ by definition of $\equiv_{p11}$.

(2) We prove $\mathcal{T}_{p11}(P\rho) = \mathcal{T}_{p11}(P)\mathcal{T}_{p11}(\rho)$ by induction on the structure of P.

Base cases. If P is ϵ we have $var(P) = \emptyset$. Moreover $\mathcal{T}_{p11}(\epsilon) = LL$ thus, $var(\mathcal{T}_{p11}(P)) = \emptyset$. Thus, trivially $\mathcal{T}_{p11}(P\rho) = \mathcal{T}_{p11}(P) = \mathcal{T}_{p11}(P)\mathcal{T}_{p11}(\rho)$.

If P is Y, since $\mathcal{T}_{p11}(Y) = X_Y$ we directly have $\mathcal{T}_{p11}(P\rho) = \mathcal{T}_{p11}(\rho(Y)) = \mathcal{T}_{p11}(\rho)(X_Y) = \mathcal{T}_{p11}(P)\mathcal{T}_{p11}(\rho)$.

Inductive case. Let P be $p[P']$. Then, by inspecting Table VII we notice that in some cases P' is

required to be ϵ and the corresponding types contain no variables. Thus, we have $\mathcal{T}_{p11}(P\rho) = \mathcal{T}_{p11}(P) = \mathcal{T}_{p11}(P)\mathcal{T}_{p11}(\rho)$ as for base case ϵ. In all the remaining cases, $\mathcal{T}_{p11}(p[P']) = \mu\mathsf{K}^{\ell}[\mathcal{T}_{p11}(P')]$, where the choice of μ and ℓ does not depend on ρ, only on p. Thus $\mathcal{T}_{p11}(P\rho) = \mathcal{T}_{p11}(p[P']\rho) = \mathcal{T}_{p11}(p[P'\rho]) = \mu\mathsf{K}^{\ell}[\mathcal{T}_{p11}(P'\rho)]$. By inductive hypothesis, $\mu\mathsf{K}^{\ell}[\mathcal{T}_{p11}(P'\rho)] = \mu\mathsf{K}^{\ell}[\mathcal{T}_{p11}(P')\mathcal{T}_{p11}(\rho)] = \mu\mathsf{K}^{\ell}[\mathcal{T}_{p11}(P')]\mathcal{T}_{p11}(\rho) = \mathcal{T}_{p11}(P)\mathcal{T}_{p11}(\rho)$.

(3) We prove $var(\mathcal{T}_{p11}(P)) = \mathcal{T}_{p11}(var(P))$ by induction on the structure of P.

Base cases. If P is ϵ then $var(P) = \emptyset = \mathcal{T}_{p11}(var(P))$ and $var(\mathcal{T}_{p11}(P)) = var(LL) = \emptyset$. If P is Y then $var(\mathcal{T}_{p11}(P)) = X_Y = \mathcal{T}_{p11}(var(P))$.

Inductive case. If P is $p[P']$ then we notice that $var(P) = var(P')$. For the cases where P' is ϵ, the corresponding types have no variables so we trivially have $var(\mathcal{T}_{p11}(P)) = \emptyset = \mathcal{T}_{p11}(var(P))$. For the remaining cases, $var(\mathcal{T}_{p11}(p[P'])) = var(\mathcal{T}_{p11}(P')) \stackrel{ih}{=} \mathcal{T}_{p11}(var(P')) = \mathcal{T}_{p11}(var(p[P']))$.

(4) Since $G = \{P \mid \mathcal{T}(P) = \mu\mathsf{K}^{\ell}[T']$ implies $\mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}\}$ it is enough to observe that types in Table VII with $\mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}$ are such that $\ell = HH \vee T' = LL$. ■

We can thus apply Theorem 25 to prove security of PKCS#11 API specifications.

Example 29. *We revise once more the symmetric key wrapping example. We specify it using PKCS#11 attributes as follows:*

$$\begin{array}{l}\mathsf{SymWrap}(h\_key,\ h\_w)\\ \quad w := \mathsf{getKey}(h\_w, \{SecK, W\}[Y]);\\ \quad k := \mathsf{getKey}(h\_key, Y);\\ \quad \mathsf{return}\ \mathsf{enc}(k, w);\end{array}$$

We check that the wrapping key is symmetric ($SecK$) and is authorized to wrap (W). The transported key has an unspecified property Y that is matched in the second call to getKey. *We have that $\mathcal{T}_{p11}(\{SecK, W\}[Y]) = \mathsf{SymK}^{HH}[X_Y]$ and $\mathcal{T}_{p11}(Y) = X_Y$. The program translated under $\mathcal{T}_{p11}$ typechecks as we did in Example 9. Thus, by Theorem 25, this API is secure.*

We have shown that a significant fragment of PKCS#11 key management attributes can be translated into our typed model so that well-typed translations ensure security of PKCS#11 API specifications. We only had to prove, in Proposition 28, that there exists a suitable set G of properties of the generated keys for which the translation respects Definition 20. This method is general and can be applied to other APIs.

VI. Conclusions

In the past few years, many attacks against cryptographic key management APIs have been presented and most of them were based on the improper use of cryptographic keys. In this paper, we proposed a simple imperative programming language for specifying strongly-typed APIs for the management of symmetric, asymmetric and signing keys. The main idea is to have expressive key types directly stored in the device, however independent of the implementation, that are matched at run-time when managing keys. We then developed a type-based analysis to prove the preservation of integrity and confidentiality of sensitive keys and have shown that this abstraction is expressive enough to code realistic key management APIs.

In order to code realistic key-management API's in our framework we defined a more concrete version of the language that allows for storing real key properties. We then showed that, under reasonable conditions, if the concrete properties are mapped into types, the general security results on typing are preserved.

As a case study we have shown an encoding of PKCS#11 v2.20 by mapping the standard attributes into our types in a version that can be type-checked and thus proved secure.

Some recent work has focused on strong information flow guarantees for general-purpose programs with cryptographic primitives [14], [17]. These techniques have been applied to a different setting and it would be interesting as future work to study whether they could be applied to the problem of type-based analysis of key management APIs.

Acknowledgements

This work was partially supported by FCT projects ComFormCrypt PTDC/EIA-CCO/113033/2009 and PEst-OE/EEI/LA0008/2011 and by PRIN 2010 Project "Security Horizons" 2010XSEMLC_007.

References

[1] P. Adão, R. Focardi, and F.L. Luccio. Type-Based Analysis of Generic Key Management APIs (Long Version). *IACR Cryptology ePrint Archive*, 2013.

[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] M. Bond. Attacks on cryptoprocessor transaction sets,. In *Proc. of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 220–234. Springer Verlag, 2001.

[4] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proc. of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 260–269. ACM, 2010.

[5] C. Cachin and J. Camenisch. Encrypting keys securely. *IEEE Security & Privacy*, 8(4):66–69, 2010. IEEE Computer Society.

[6] M. Centenaro, R. Focardi, and F.L. Luccio. Type-based Analysis of PKCS#11 Key Management. In *Proc. POST*, volume 7215 of *LNCS*, pages 349–368. Springer Verlag, 2012.

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

[8] 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 Verlag, 2003.

[9] 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 Verlag, 2003.

[10] J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.

[11] V. Cortier and G. Steel. A generic security API for symmetric key management on cryptographic devices. In *Proc. of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, LNCS, pages 605–620. Springer Verlag, 2009.

[12] 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. IOS Press.

[13] R. Focardi and M. Maffei. Types for security protocols. *Cryptology and Information Security Series, Formal Models and Techniques for Analyzing Security Protocols*, 5:143–181, 2011. IOS Press.

[14] 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.

[15] 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 Verlag.

[16] S. Kremer, G. Steel, and B. Warinschi. Security for key management interfaces. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*, pages 266–280. IEEE Computer Society Press, June 2011.

[17] 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.

[18] A. Myers and A. Sabelfeld. Language-based information-flow security . *IEEE Journal on Selected Areas in Communications,*, 21(1):519, January 2003.

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

APPENDIX

In this Appendix we present the proofs referred in the main body of the paper.

Restatement of Theorem 16. *Let* $\Gamma, \Theta, \sigma \vdash \mathsf{M}, \mathsf{H}, \mathsf{V}$ *and* $\Gamma \vdash_{\mathsf{c}} \mathsf{c}$. *If* $\langle \mathsf{M}, \mathsf{H}, \sigma, \mathsf{c}\rangle \to \langle \mathsf{M}', \mathsf{H}', \sigma', \mathsf{c}'\rangle$ *then*

(i) *if* $\mathsf{c}' \neq \varepsilon$ *then* $\Gamma \vdash_{\mathsf{c}} \mathsf{c}'$*;*

(ii) $\exists \Theta' \supseteq \Theta$ *such that* $\Gamma, \Theta', \sigma' \vdash \mathsf{M}', \mathsf{H}', \mathsf{V}'$*, where* $\mathsf{V}' = \mathsf{V} \cup \{val \mid \exists g \in dom(\mathsf{H}') \backslash dom(\mathsf{H}).\mathsf{H}'(g) = (val, T)\} \backslash ran[\mathsf{M}]$.

Proof: Suppose that $\langle \mathsf{M}, \mathsf{H}, \sigma, \mathsf{c}\rangle \to \langle \mathsf{M}', \mathsf{H}', \sigma', \mathsf{c}'\rangle$.

We prove (i) by induction on c. We analyze the only two rules where $\mathsf{c}' \neq \varepsilon$.

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

Since by hypothesis $\Gamma \vdash_{\mathsf{c}} \mathsf{c}_1; \mathsf{c}_2$ we have that $\Gamma \vdash_{\mathsf{c}} \mathsf{c}_1$ and $\Gamma \vdash_{\mathsf{c}} \mathsf{c}_2$ which automatically implies our result.

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

In this second case, since by IH $\Gamma \vdash_{\mathsf{c}} \mathsf{c}_1'$ and by hypothesis $\Gamma \vdash_{\mathsf{c}} \mathsf{c}_2$ it follows $\Gamma \vdash_{\mathsf{c}} \mathsf{c}_1'; \mathsf{c}_2$.

Let us now address (ii) analyzing all the possible cases. We want to show that given

- $\mathsf{M}(x) = v$, $\Gamma(x) = T$ implies $\Theta \models_v v : T\sigma$; and
- $\mathsf{H}(v') = (v, T)$ implies $\Theta \models_v v : T$,
- $val \in \mathsf{V}$ then $\exists g.\mathsf{H}(g) = (val, T)$ and $\Theta(val) = T$,

there is a $\Theta' \supseteq \Theta$ such that

(a) $\mathsf{M}'(x) = v$, $\Gamma(x) = T$ implies $\Theta' \models_v v : T\sigma'$; and

(b) $\mathsf{H}'(v') = (v, T)$ implies $\Theta' \models_v v : T$,

(c) $val \in \mathsf{V}'$ then $\exists g.\mathsf{H}'(g) = (val, T)$ and $\Theta'(val) = T$,

(d) Θ' is well-defined, namely, it is a function, the image of Θ' only contains ground types, and verify conditions in (2).

(i) Case $\mathsf{c} = x := e$:

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

Consider $\Theta' = \Theta$. Since by hypothesis $\Gamma \vdash_{\mathsf{c}} x := e$ implies $\Gamma(x) = T$ and $\Gamma \vdash_e e : T$, and $e \downarrow^{\mathsf{M}} v$ then by Proposition 15 we have $\Theta \models_v v : T\sigma$.

Since the only difference from M to M' is in $x \mapsto v$ and $\Theta' = \Theta$ *(a)* follows.

Since in this case $\mathsf{H}' = \mathsf{H}$ and $\Theta' = \Theta$, *(b)*, *(c)*, and *(d)* follow immediately from the hypothesis.

(ii) Case $\mathsf{c} = x := \mathsf{getKey}(y, T)$:

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

Consider again $\Theta' = \Theta$. The only difference from M to M' is in $x \mapsto v$.

(a) Since $\mathsf{M}'(x) = v$ and by $\Gamma \vdash_{\mathsf{c}} x := \mathsf{getKey}(y, T)$ we get $\Gamma(x) = T$, what we want to show is that $\Theta' \models_v v : T\sigma\sigma'$.

By hypothesis $\mathsf{H}(\mathsf{M}(y)) = (v, T')$ which implies by well-formedness that $\Theta \models_v v : T'$. Since $\Theta = \Theta'$ and by hypothesis $T' = T\sigma\sigma'$, the result follows.

Again, since in this case $\mathsf{H}' = \mathsf{H}$ and $\Theta' = \Theta$, *(b)*, *(c)*, and *(d)* are immediate from the hypothesis.

(iii) Case $\mathsf{c} = x := \mathsf{genKey}(T)$:

$$\frac{\begin{array}{c} g, g' \leftarrow \mathcal{G} \qquad T\sigma \text{ ground} \\ T\sigma = \mu\mathsf{K}^{\ell}[T'] \implies \mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\} \wedge (\ell = HH \vee T' = LL) \end{array}}{\langle \mathsf{M}, \mathsf{H}, \sigma, x := \mathsf{genKey}(T)\rangle \to \langle \mathsf{M}[x \mapsto g], \mathsf{H}[g \mapsto (g', T\sigma)], \sigma, \varepsilon\rangle}$$

Since g, g' are a freshly random atomic values, define $\Theta' = \Theta \cup \{g \mapsto LL, g' \mapsto T\sigma\}$.

(a) The only difference from M to M' is in $x \mapsto g$. Since $\mathsf{M}'(x) = g$ and by $\Gamma \vdash_{\mathsf{c}} x := \mathsf{genKey}(T)$ we get $\Gamma(x) = LL$, what we want to show is that $\Theta' \models_v g : LL$ which is true by construction.

(b) Now from H to H' the difference is $g \mapsto (g', T\sigma)$. We want to show then that $\Theta' \models_v g' : T\sigma$ which is also true by construction.

(c) Since the difference from H to H' is $g \mapsto (g', T\sigma)$, g' is atomic, and $g' \notin \mathit{ran}[\mathsf{M}]$ we have that $\mathsf{V}' = \mathsf{V} \cup \{g'\}$. By construction $\exists g.\mathsf{H}'(g) = (g', T\sigma)$ with $\Theta'(g') = T\sigma$.

To prove *(d)* notice that g, g' are fresh, LL and $T\sigma$ are ground by hypothesis, and by construction and side-condition of the rule both LL and $T\sigma$ satisfy the conditions in (2).

(iv) Case $\mathsf{c} = x := \mathsf{setKey}(y, T)$:

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

Since g is a freshly random atomic value, define $\Theta' = \Theta \cup \{g \mapsto LL\}$.

(a) The only difference from M to M' is in $x \mapsto g$. Since $\mathsf{M}'(x) = g$ and by $\Gamma \vdash_{\mathsf{c}} x := \mathsf{setKey}(y, T)$ we get $\Gamma(x) = LL$ and $\Gamma \vdash_e y : T$, what we want to show is that $\Theta' \models_v g : LL$ that is true by construction.

(b) Now from H to H' the difference is $g \mapsto (\mathsf{M}(y), T\sigma)$. We want to show then that $\Theta' \models_v \mathsf{M}(y) : T\sigma$. By Proposition 15 since $\Gamma \vdash_e y : T$ by typing, $y \downarrow^{\mathsf{M}} \mathsf{M}(y)$ by definition, and $\Gamma, \Theta, \sigma \vdash \mathsf{M}, \mathsf{H}, \mathsf{V}$ by hypothesis one has $\Theta \models_v \mathsf{M}(y) : T\sigma$. Since $\Theta \subset \Theta'$ *(b)* follows.

(c) The difference from H to H' is $g \mapsto (\mathsf{M}(y), T\sigma)$ but $\mathsf{M}(y) \in \mathit{ran}(\mathsf{M})$ so $\mathsf{V}' = \mathsf{V}$ and the result follows by hypothesis;

To prove *(d)* notice that g is fresh, LL is ground, and LL satisfy the conditions in (2).

(v) Case $\mathsf{c} = \mathsf{return}\ e$ has no transition associated.

(vi) Case $\mathsf{c} = \mathsf{c}_1; \mathsf{c}_2$: both cases follow directly from IH.

■

Restatement of Lemma 17. *Let* $\Gamma \vdash_{\mathsf{c}} \mathcal{A}$ *and* $\langle \mathsf{H}_0, V_0\rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}, V\rangle$.

Then, there exists Θ *such that* $\Theta \models_{\mathsf{H}} \mathsf{H}$, $\Theta \models_v v : LL$ *for each* $v \in V$, *and* $\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}_{\mathsf{ok}}(\mathsf{H}, V)$.

Proof: We show the result by induction on the length of the attack.

The base case is when $\mathsf{H} = \emptyset$ and $V = V_0$. Given that H is empty and $V = V_0 \subseteq \mathcal{C}$, defining $\Theta(v) = LL$ for all $v \in V_0$ gives us immediately the result.

Consider now that $\langle \mathsf{H}_0, V_0\rangle \rightsquigarrow^*_{\mathcal{A}} \langle \mathsf{H}_n, V_n\rangle \rightsquigarrow_{\mathcal{A}} \langle \mathsf{H}, V\rangle$. By IH $\exists \Theta_n$ such that

- $\Theta_n \models_{\mathsf{H}} \mathsf{H}_n$,
- $\Theta_n \models_v v : LL$ for all $v \in V_n$, and
- $\Theta_n \Vdash_{\mathsf{V}} \mathsf{H}_n, \mathsf{V}_{\mathsf{ok}}(\mathsf{H}_n, V_n)$.

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

Looking at the last step there was a call to some $\mathsf{a} \in \mathcal{A}$ with $v_1, \ldots, v_k \in \mathcal{K}(V_n)$, $\mathsf{a}(v_1, \ldots, v_k) \downarrow^{\mathsf{H}_n, \mathsf{H}} v$, and $V = V_n \cup \{v\}$.

Given that by IH $\Theta_n \models_v v : LL$ for all $v \in V_n$ and $v_1, \ldots, v_k \in \mathcal{K}(V_n)$ we get by Proposition 14 that $\Theta_n \models_v v_i : LL$.

Unfolding the operation call we get that $\mathsf{a} = \lambda x_1 \ldots x_k.\mathsf{c}$, $\langle \mathsf{M}_\epsilon[x_i \mapsto v_i], \mathsf{H}_n, \emptyset, \mathsf{c}\rangle \to \langle \mathsf{M}, \mathsf{H}, \sigma, \mathsf{return}\ e\rangle$ and $e \downarrow^{\mathsf{M}} v$, and consequently from $\Gamma \vdash_{\mathsf{c}} \mathsf{a}$ we get $\Gamma \vdash_{\mathsf{c}} \mathsf{c}$ and $\Gamma \vdash_e x_i : LL$.

Now, from $\mathsf{M}_\epsilon[x_i \mapsto v_i](x_i) = v_i$, $\Gamma \vdash_e x_i : LL$, and $\Theta_n \models_v v_i : LL$ we get by definition of well-formedness $\Gamma, \Theta_n, \emptyset \vdash_{\mathsf{M}} \mathsf{M}_\epsilon[x_i \mapsto v_i]$ that together with IH imply $\Gamma, \Theta_n, \emptyset \vdash \mathsf{M}_\epsilon[x_i \mapsto v_i], \mathsf{H}_n, \mathsf{V}_{\mathsf{ok}}(\mathsf{H}_n, V_n)$.

We can hence apply Theorem 16 to $\langle \mathsf{M}_\epsilon[x_i \mapsto v_i], \mathsf{H}_n, \emptyset, \mathsf{c}\rangle \to \langle \mathsf{M}, \mathsf{H}, \sigma, \mathsf{return}\ e\rangle$ and obtain

(i) $\Gamma \vdash_{\mathsf{c}} \mathsf{return}\ e$ and consequently $\Gamma \vdash_e e : LL$;

(ii) $\exists \Theta \supseteq \Theta_n$ such that $\Gamma, \Theta, \sigma \vdash \mathsf{M}, \mathsf{H}, \mathsf{V}$ where $\mathsf{V} = \mathsf{V}_{\mathsf{ok}}(\mathsf{H}_n, \mathsf{V}_n) \cup \{val \mid \exists g \in \mathit{dom}(\mathsf{H}) \setminus \mathit{dom}(\mathsf{H}_n).\mathsf{H}(g) = (val, T)\} \setminus \mathit{ran}[\mathsf{M}_\epsilon[x_i \mapsto v_i]]$

From (ii) it follows immediately that $\Gamma, \Theta, \sigma \vdash_{\mathsf{M}} \mathsf{M}$, $\Theta \models_{\mathsf{H}} \mathsf{H}$, and $\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}_{\mathsf{ok}}$.

Given that $\Gamma \vdash_e e : LL$, $e \downarrow^{\mathsf{M}} v$, and $\Gamma, \Theta, \sigma \vdash_{\mathsf{M}} \mathsf{M}$, we apply Proposition 15 to get $\Theta \models_v v : LL$.

Finally since by IH $\Theta_n \models_v v' : LL$ for all $v' \in V_n$, $\Theta_n \subseteq \Theta$ and $\Theta \models_v v : LL$ we get that $\Theta \models_v v : LL$ for all $v \in V_n \cup \{v\} = V$.

Since $\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}$ and $\mathsf{V}_{\mathsf{ok}}(\mathsf{H}_n, V_n) \subseteq \mathsf{V}$ we have $\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}_{\mathsf{ok}}(\mathsf{H}_n, V_n)$. ■