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 trough 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 security 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 e.g., cryptographic keys or certificates in tokens, are referenced via *handles*, that are pointers to or names for the objects in secure memory. 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 object should have.

In the last decade this has lead to many different attacks both on HSMs and smart cards (see, e.g., [1], [2], [3], [7]). 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 [2], 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 [3]. In this context some 'patches' have been presented and 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 to some non-critical functions the usage of imported keys [3], or by adding a wrapping format that binds attributes to wrapped keys [11], [13]. Other attacks on PIN processing APIs are, e.g., on formats used for message encryption [9], or on the lack of integrity of user data [6].

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 then provide a concrete semantics, in which concrete key properties are stored in place of types. We then investigate conditions that allow to map concrete APIs over the proposed types so that security results are preserved. In particular, we prove that if the translation of the concrete API to the typed one is well-typed than 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 [16]. We show that PKCS#11 attributes can be mapped into types preserving the above mentioned conditions, and this allows to prove security through the general type-checking.

Related work: The literature proposes different solutions for the designs of new secure token interfaces and the proofs of their security. In [4] secure token interfaces are proposed together with security proofs in the cryptographic model. The security relies on the access of 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 [10] secure token interfaces are proposed for a distributed setting, together with security proofs in the symbolic model. However, this approach assumes a limited set of functionalities. [14] introduces a general security model for cryptographic APIs: it defines a new notion of security for cryptographic APIs, and it applies this notion to the security proofs both in the symbolic and 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 from the one in [12], 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 [12]: here 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 happen in many real implementations.

In [5] 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 which is applicable general cryptographic APIs, that is, any key storage which is managed through handles, and manages also asymmetric and signing keys, in the style of [14]. 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; 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 out framework. We conclude 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 from [5] but here we develop it around more expressive types for keys that dictate how key should be used and what is their security level. Moreover we consider asymmetric encryption and digital signatures which are not accounted for in [5].

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

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

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, $vk(v)$ is the *verification* key corresponding to the *signature* 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; instead, the decryption under v' of $enc(v, v'')$, with $v'' \neq v'$ will be explicitly represented as $dec(enc(v, v''), v')$. This allows us to model cryptosystems with no integrity checks: decrypting with a wrong key never gives a failure. Signature verification, instead, only succeeds when the verification key corresponds to the signature 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 as 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 signed message only when the verification key corresponds to the signing key.

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

Table I
THE SEMANTICS OF EXPRESSIONS

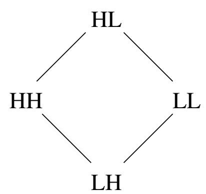

Figure 1. Security lattice

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 are received they are expected to be originated at some trusted source.

Figure 1 is a standard security lattice showing that confidentiality and integrity levels are contra-variant [15]. Moving up is safe while moving down is unsafe, thus it is safe to consider a public datum as secret, while it is unsafe promoting 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 $\ell_C^1\ell_I^1 \sqsubseteq \ell_C^2\ell_I^2$ iff $\ell_C^1 \sqsubseteq_C \ell_C^2$ and $\ell_I^1 \sqsubseteq_I \ell_I^2$.

We define the following types:

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

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 for generic data at security level ℓ; and type $\mu\mathsf{K}^{\ell}[T]$ is 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 are stored, retrieved and checked at run-time in order to authorize specific cryptographic operations. Type variables allows 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 $\ell_I(\ell) = \ell_C(\mu\mathsf{K}^{\ell}[T]) = \ell_I^\star$ 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 $\mu\mathsf{K}^{\ell}[T]$ 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:

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

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 function 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 modelled 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 reported in Table II, where ϵ denotes the empty API. Assignment $x := e$ evaluates expression e on M and stores the result in variable x, noted $\mathsf{M}[x \mapsto v]$. In case x is not defined in M the domain of M is extended to include the new variable, otherwise the value stored in x is overwritten. 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 note 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 now show 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,*

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

which gives $\mathsf{SymWrap}(\mathsf{g}, \mathsf{g}') \downarrow^{\mathsf{H},\mathsf{H}} enc(v, v')$ *meaning that the value returned invoking the wrap command is thus the encryption of v under v'. Notice how X gets bound 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 $v'' \in \mathcal{K}(V)$.

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

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

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

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

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

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

Table II
API SEMANTICS

Given a handle map H, representing tokens, 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 key 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:*

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

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 permit only a

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

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

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

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

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

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

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

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

Table III
TYPING EXPRESSIONS

subset of key types in Γ (other types for keys are derived by these ones).

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* $T = LL$*.*

Type judgment for expressions is noted $\Gamma \vdash_e e : T$ meaning that expression e is of type T under Γ. Typing rules are reported 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 nor more than once in a sequence, that is, given the sequence $T_1 \leq T_2, \ldots, T_{n-1} \leq T_n$, 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 the confidentiality to L. In fact, these keys can be be safely made public. Rule [*enc*] encrypts the result of an expression of type T, as required by the key

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

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

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

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

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

Table IV
TYPING APIS

type. The ciphertext has low confidentiality and the same integrity level as the encryption key. 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. In fact, since encryption key is public the plaintext might come by the attacker. This is not the case only when the ciphertext has high integrity. Finally, rule [*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. In order to protect its confidentiality we have to preserve the confidentiality level in the signature.

APIs: We now type APIs 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 of the same type; rules [*genkey*] and [*setkey*] store keys and return a LL handle. 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:*

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

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 typecheck the return command, completing the typing.

A. Type soundness

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 very 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. We can show that private/symmetric keys either are 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 $T = LL$.

Proposition 12. *Suppose that $v \neq dec(v', v''), dec^a(v', v'')$, and 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

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

Proposition 13. *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 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 says 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; the values in V are recorded 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 14 (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 $\Theta(val) = T$.*

It follows from this definition, and Lemma 30 that σ is such that all the $T\sigma$ above are ground.

Having defined the properties of keys and the notion of well-formed memory and handle-maps we characterize which values an adversary may derive. We show that with the rules from Section II, given a set of values of type LL an 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 15. *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 $\Theta \models_v v : LL$.

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

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

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

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

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

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

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

Table V
VALUE WELL-FORMEDNESS

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

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 18. *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* $\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}_{\mathsf{ok}}(\mathsf{H}, V)$.

Lemma 19. *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 18 ■

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

Theorem 20. *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 19 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 15 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 15 $\Theta \models_v val : LL$ 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 give a general theorem stating that whenever we assign types to key properties in a unique way, security results are preserved in the new, concrete semantics.

Key properties: Properties of keys have the following syntax:

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

where Y is a property variable that will be bound at runtime, p is a value specifying the actual properties, $p[P]$ represents a key with properties p which can perform cryptographic operations on keys with properties P. It typically happens that many concrete properties are treated the same when authorizing cryptographic operations. For example, an encryption key is always allowed to perform encryption independently of its actual length or of the algorithm it is bound to. Of course these details are important when actual cryptography takes place but they are irrelevant in our analysis. For this reason, we assume to have an equivalence relation on concrete properties $\equiv$ that relates properties which make the APIs behave the same way.

Concrete syntax and semantics: We define a more concrete syntax which stores concrete key properties instead of types. In the code, only internal functions are affected:

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

The semantics is reported in Table VI and is close to the one of Table II. There are however some important differences: $\mathsf{H_p}$ notes the new concrete handles that store actual key properties; ρ is a substitution of key property variables into key properties; all occurrences of types T are replaced by key properties P; in getKey, when we match properties, we also allow for equivalent key properties.

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

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

Definition 21 (Typed key properties)**.** *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 $\mathcal{T}(\rho)(\mathcal{T}(Y)) = \mathcal{T}(\rho(Y))$;

Notice that 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 commands by simply applying it to all occurrences of key properties.

Definition 22 (Connecting concrete and typed semanics)**.** *Given a mapping $\mathcal{T}$ from key properties to types we apply it to handles, commands and substitutions as follows:*

- *$\mathcal{T}(\mathsf{H_p})(v) = (v', \mathcal{T}(P))$ whenever $\mathsf{H_p}(v) = (v', P)$;*
- *for internal functions, we have*

$$\begin{aligned} \mathcal{T}(\mathsf{getKey}(y, P)) &= \mathsf{getKey}(y, \mathcal{T}(P)) \\ \mathcal{T}(\mathsf{genKey}(P)) &= \mathsf{genKey}(\mathcal{T}(P)) \\ \mathcal{T}(\mathsf{setKey}(y, P)) &= \mathsf{setKey}(y, \mathcal{T}(P)) \end{aligned}$$

All other commands just apply $\mathcal{T}$ to subcommands, recursively.

Theorem 23 (Semantic correspondence)**.** *Let $\mathcal{T}$ be a typing for key properties, and pick the same fresh number generator $\mathcal{G}$ for the two semantics. Then,*

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

implies

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

Proof: By easy induction on the length of the derivation of the two reductions, applying Definitions 21 and 22. ∎

As a consequence we have that all the attacks in the concrete semantics are mimicked in the typed one:

Corollary 24. *Let $\langle \mathsf{H_p}, V \rangle \rightsquigarrow^*_{\mathcal{A}_p} \langle \mathsf{H}'_\mathsf{p}, V' \rangle$. Then we have $\langle \mathcal{T}(\mathsf{H_p}), V \rangle \rightsquigarrow^*_{\mathcal{T}(\mathcal{A}_p)} \langle \mathcal{T}(\mathsf{H}'_\mathsf{p}), V' \rangle$*

Definition of security can be done through the type associated to concrete key properties:

Definition 25 (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 26. *Let $\Gamma \vdash_\mathsf{c} \mathcal{T}(\mathcal{A}_p)$. Then $\mathcal{A}_p$ is secure.*

Proof: This is a direct consequence of Corollary 24. ∎

V. Case study: PKCS#11 v2.20

PKCS#11, also known as Cryptoki, defines a widely adopted API for cryptographic tokens [16]. It provides access to cryptographic functionalities while, in principle, providing some security properties. More specifically, the value of keys stored on a PKCS#11 device and tagged as *sensitive* should never be revealed outside the token, even when connected to a compromised host. Unfortunately, PKCS#11 is known to be vulnerable to attacks that break this property [3], [8], [11].

There may be various *objects* stored in the token, such as cryptographic keys and certificates. Objects are referenced via *handles*. The value of a key is one of the *attributes* of

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

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

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

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

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

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

Table VI
API CONCRETE SEMANTICS

the enclosing object. There are other attributes to specify the various roles a key can assume: each different API call can, in fact, require a different role. For example, decryption keys are required to have attribute `CKA_DECRYPT` set, while key-encrypting keys, i.e., keys used to encrypt other keys, must have attribute `CKA_WRAP` set.

PKCS#11 key properties: Properties and capabilities of keys are described by set of *attributes*. When a certain attribute is contained in the set of key properties p we will say that the attribute is set, it is unset otherwise. 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 revealed out 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 27. *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. It 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 $\mathcal{T}_{p11}$ 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.

$\mathcal{T}_{p11}(Y) = X_Y$

$\mathcal{T}_{p11}(p) = \mathcal{T}_{p11}(p[\{\}])$

$\mathcal{T}_{p11}(p[P]) =$

$$= \begin{cases} H & \begin{cases} PrivK & \begin{cases} D & \mathsf{DecK}^{HL}[LL] \\ U & \mathsf{DecK}^{HH}[\mathcal{T}_{p11}(P)] \\ S & \mathsf{SigK}^{HH}[\mathcal{T}_{p11}(P)] \end{cases} \\ SecK & \begin{cases} E,D & \mathsf{SymK}^{HL}[LL] \\ W,U & \mathsf{SymK}^{HH}[\mathcal{T}_{p11}(P)] \end{cases} \\ \neg C & HL \end{cases} \\ \neg H & \begin{cases} PubK & \begin{cases} E & \mathsf{EncK}^{LL}[LL] \\ W & \mathsf{EncK}^{LH}[\mathcal{T}_{p11}(P)] \\ V & \mathsf{VerK}^{LH}[\mathcal{T}_{p11}(P)] \end{cases} \\ \neg C & LL \end{cases} \end{cases}$$

Example 28. *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 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 $\mathcal{T}_{p11}(p \cup \{A\}) = \mathcal{T}_{p11}(p)$. Thus item 1 of Definition 21 trivially holds.

Now notice that, since $\mathcal{T}_{p11}(Y) = X_Y$ for all variables Y, $\mathcal{T}_{p11}(P\rho)$ is the same as $\mathcal{T}_{p11}(P)$ where all the occurrences of variables X_Y are replaced by $\mathcal{T}_{p11}(Y\rho)$, which is exactly the definition of $\mathcal{T}_{p11}(\rho)$. Thus, $\mathcal{T}_{p11}(P\rho) = \mathcal{T}_{p11}(P)\mathcal{T}_{p11}(\rho)$. This is exactly what item 2 of Definition 21 requires.

We can thus apply Theorem 26 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:*

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

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* $\mathcal{T}_{p11}$ *type-checks as we did in Example 9. Thus, by Theorem 26, this API is secure.*

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 show that if, under reasonable conditions, 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.

VII. Acknowledgements

This work was partially supported by FCT projects ComFormCrypt PTDC/EIA-CCO/113033/2009 and PEst-OE/EEI/LA0008/2011.

References

[1] 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.

[2] M. Bond. Attacks on cryptoprocessor transaction sets,. In *Proc. 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES01)*, volume 2162 of *LNCS*, page 220234. Springer, 2001.

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

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

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

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

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

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

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

[10] V. Cortier and G. Steel. A generic security API for symmetric key management on cryptographic devices. In *Proc. 14th European Symposium on Research in Computer Security (ESORICS09)*, LNCS, pages 605–620. Springer, 2009.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] RSA Security Inc. *PKCS #11 v2.20: Cryptographic Token Interface Standard*, July 2004. Available at http://www.rsa.com/rsalabs/node.asp?id=2133.

APPENDIX

In this Appendix we present the proofs of the Propositions and Theorems Stated in the main body of the paper.

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

Proof: We prove by induction on the depth of the derivation. Since val is atomic the of the derivation is either [*atom*] or [*sub*].

Case [*atom*]: then $T = T'$ and we are done.

Case [*sub*]: then $\Theta \models_v val : T''$ with $T'' \leq T'$. By IH $T \leq T'' \leq T'$. ■

Restatement of 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 $T = LL$.*

Proof: 1). Suppose that $\ell = HH$. We prove by induction on the depth of the derivation $\Theta \models_v v : \mu\mathsf{K}^{HH}[T]$.

- *(i)* Case [*atom*]: in this case v is atomic and then $\Theta(v) = \mu\mathsf{K}^{HH}[T]$ automatically.
- *(ii)* Case [*sub*]: in this case $\Theta \models_v v : T'$ and $T' \leq \mu\mathsf{K}^{HH}[T]$ which by definition of $\leq$ implies that $T' = \mu\mathsf{K}^{HH}[T]$. As we assumed that we never apply rule [*sub*] uselessly we have to exclude this case.
- *(iii)* Cases [*ek*] and [*vk*] are immediate because they derive keys with level $L\ell_I$.
- *(iv)* Cases [*dec*] and [*deca*] are excluded by hypothesis.
- *(v)* All the other cases are excluded because they do not derive $\mu\mathsf{K}^{HH}[T]$.

2). We prove this by induction on the depth of the derivation $\Theta \models_v v : \mu\mathsf{K}^\ell[T]$.

- *(i)* Case [*atom*]: in this case v is atomic and then $\Theta(v) = \mu\mathsf{K}^\ell[T]$ that by (2) imply that $\ell = HH$ or $T = LL$.
- *(ii)* Case [*sub*]: in this case $\Theta \models_v v : T'$ and $T' \leq \mu\mathsf{K}^\ell[T]$. Then by definition of $\leq$ we have $T = LL$.
- *(iii)* Cases [*ek*] and [*vk*] are immediate because $\mathsf{Enc}, \mathsf{Ver} \notin \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}$.
- *(iv)* Cases [*dec*] and [*deca*] are excluded by hypothesis.
- *(v)* All the other cases are excluded because they do not derive $\mu\mathsf{K}^\ell[T]$.

3). We prove this also by induction on the depth of the derivation of $\Theta \models_v v : \mu\mathsf{K}^\ell[T]$.

- *(i)* Case [*atom*]: in this case v is atomic and then $\Theta(v) = \mu\mathsf{K}^\ell[T]$ which is impossible by (2) as $\mu \notin \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}$.
- *(ii)* Case [*sub*]: in this case $\Theta \models_v v : T'$ and $T' \leq \mu\mathsf{K}^\ell[T]$. Then by definition of $\leq$ we have $T = LL$.
- *(iii)* Case [*ek*]: in this case we have $\Theta \models_v ek(v') : \mathsf{EncK}^{L\ell_I}[T]$ and consequently $\Theta \models_v v' : \mathsf{DecK}^{\ell_C\ell_I}[T]$. By Case 2) above we get that $\ell_C\ell_I = HH$ or $T = LL$, hence our claim follows.
- *(iv)* Case [*vk*]: similar to case [*ek*] using Ver and Sig.

(v) Cases [*dec*] and [*deca*] are excluded by hypothesis.

(vi) All the other cases are excluded because they do not derive $\mu\mathsf{K}^{\ell}[T]$.

■

Restatement of Proposition 12. *Suppose that* $v \neq dec(v', v''), dec^a(v', v'')$*, and 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*

1) if $\ell = HH$*, then* $\ell' = HH$ *and* $\mu = \mu'$*;*

2) if $\ell \neq HH$ *then* $T = T' = LL$.

Proof: Since we have by hypothesis $\mu, \mu' \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}$, we know by Proposition 11(2) that $\ell = HH$ or $T = LL$ and $\ell' = HH$ or $T' = LL$.

Also, analyzing all possible rules that derive $\Theta \models_v v : \mu\mathsf{K}^{\ell}[T]$ with $\mu \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}$ we can immediately see that it has to be either [*atom*] or [*sub*]. All the other rules do not derive $\mu\mathsf{K}^{\ell}[T]$ or the resulting term is a decryption. Similarly for $\Theta \models_v v : \mu'\mathsf{K}^{\ell'}[T']$.

(i) Suppose that $\ell = HH$. By Proposition 11(1) we have that v is atomic and $\Theta(v) = \mu\mathsf{K}^{HH}[T]$. Since we do not have useless applications of [*sub*] there is no $T' \leq \mu\mathsf{K}^{HH}[T]$ and so, the only possible rule is [*atom*].

Let us analyze the derivation of $\Theta \models_v v : \mu'\mathsf{K}^{\ell'}[T']$ that can only be by [*atom*] or [*sub*].

(a) Case [*atom*]: then $\mu = \mu', \ell' = \ell = HH$, and $T' = T$;

(b) Case [*sub*]: then $\Theta \models_v v : T'_1$ with $T'_1 \leq \mu'\mathsf{K}^{\ell'}[T']$. By $\leq$ we have $T' = LL$ hence get $\Theta \models_v v : \mu'\mathsf{K}^{\ell'}[LL]$. By Proposition 10 and since v is atomic, $\mu\mathsf{K}^{HH}[T] \leq \mu'\mathsf{K}^{\ell'}[LL]$ which is impossible unless they are equal. Since we assumed we do not use the rule [*sub*] uselessly, the derivation cannot be by [*sub*].

(ii) Suppose now instead that $T = LL$ and $\Theta \models_v v : \mu\mathsf{K}^{\ell}[LL]$ is derived by [*atom*] or [*sub*].

(a) Case [*atom*]: in this case v is atomic and $\Theta(v) = \mu\mathsf{K}^{\ell}[LL]$.

• If $\Theta \models_v v : \mu'\mathsf{K}^{\ell'}[T']$ is also derived by [*atom*] we have $\mu = \mu', \ell' = \ell$, and $T' = T = LL$;

• If $\Theta \models_v v : \mu'\mathsf{K}^{\ell'}[T']$ is derived by [*sub*] then $\Theta \models_v v : T'_1$ and $T'_1 \leq \mu'\mathsf{K}^{\ell'}[T']$. By definition of $\leq$ we have $T' = LL = T$ and hence $\Theta \models_v v : \mu'\mathsf{K}^{\ell'}[LL]$.

By Proposition 10 and since v is atomic, $\mu\mathsf{K}^{\ell}[LL] \leq \mu'\mathsf{K}^{\ell'}[LL]$ which is possible only if $\ell \leq \ell'$.

(b) Case [*sub*]: in this case we have $\Theta \models_v v : T_1$ with $T_1 \leq \mu\mathsf{K}^{\ell}[LL]$ that by $\leq$ implies $T_1 \leq LL$ and $\ell \leq LL$.

We can now derive $\Theta \models_v v : \mu'\mathsf{K}^{\ell'}[T']$ by rules [*atom*] and [*sub*].

• Suppose that $\Theta \models_v v : \mu'\mathsf{K}^{\ell'}[T']$ is derived by [*atom*]. Then v is atomic, $\Theta(v) = \mu'\mathsf{K}^{\ell'}[T']$, and by Proposition 10 $\mu'\mathsf{K}^{\ell'}[T'] \leq LL$ which implies that $T' = \ell' = LL = T$ or $\ell' = LH$ that by (2) implies $T' = LL = T$ because $\mu' \in \{\mathsf{Sym}, \mathsf{Dec}, \mathsf{Sig}\}$.

• Suppose that the last rule of $\Theta \models_v v : \mu'\mathsf{K}^{\ell'}[T']$ is also [*sub*] and $\Theta \models_v v : T'_1$ with $T'_1 \leq \mu'\mathsf{K}^{\ell'}[T']$. Then by $\leq$ we have $T' = LL = T$ and $\ell' \in \{LL, HL\}$.

■

Restatement of Proposition 13. *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* $T = T' = LL$.

Proof: For all $v'' \neq dec(v', v), dec^a(v', v)$ the result is immediate from Proposition 12.

For the case $v'' = dec(v', v)$ let us prove by induction on the length of the derivation $\Theta \models_v dec(v', v) : \mu\mathsf{K}^{\ell}[T]$. One immediately see that the last rule has to be [*dec*] or [*sub*] as no other rule derives values of the form $dec(v', v)$. Similarly for the case $\Theta \models_v dec(v', v) : \mu'\mathsf{K}^{\ell'}[T']$

(i) Let us consider first the case [*dec*]

$$\frac{\Theta \models_v v : \mathsf{SymK}^{\ell^\star}[\mu\mathsf{K}^{\ell}[T]] \quad \Theta \models_v v' : T_1 \quad v' \neq enc(v'', v)}{\Theta \models_v dec(v', v) : \mu\mathsf{K}^{\ell}[T]} \text{[dec]}$$

and let analyze the possible cases for $\Theta \models_v dec(v', v) : \mu'\mathsf{K}^{\ell'}[T']$.

(a) Case [*dec*]: In this case

$$\frac{\Theta \models_v v : \mathsf{SymK}^{\ell'^\star}[\mu'\mathsf{K}^{\ell'}[T']] \quad \Theta \models_v v' : T'_1 \quad v' \neq enc(v'', v)}{\Theta \models_v dec(v', v) : \mu'\mathsf{K}^{\ell'}[T']} \text{[dec]}$$

and by IH one has $\mu\mathsf{K}^{\ell}[T] = \mu'\mathsf{K}^{\ell'}[T']$.

(b) Case [*sub*]: In this case we have

$$\cfrac{\cfrac{\Theta \models_v v : \mathsf{SymK}^{\ell'^\star}[T'_1] \quad \ldots}{\Theta \models_v dec(v', v) : T'_1}\text{[dec]} \qquad T'_1 \leq \mu'\mathsf{K}^{\ell'}[T']}{\Theta \models_v dec(v', v) : \mu'\mathsf{K}^{\ell'}[T']} \text{[sub]}$$

and by IH it follows that $\mu\mathsf{K}^{\ell}[T] = T'_1 \leq \mu'\mathsf{K}^{\ell'}[T']$ which implies by definition of $\leq$ that (a) $\ell = T = T' = LL$ and $\ell' = \ell_C L$; or *(b)* $\ell = LL$, $\ell' = \ell_C L$ and $T' = LL$; or *(c)* $\ell = LH$, $\ell' = \ell_C L$ and $T' = LL$. The first immediately proves our result. For the other two cases, since we do not allow payloads different from LL for keys different from HH, and since $\ell \neq HH$ we have $T = LL = T'$.

(ii) Let us consider now the case when $\Theta \models_v dec(v', v) : \mu\mathsf{K}^{\ell}[T]$ is derived by [*sub*]. Analogously we have

$$\dfrac{\dfrac{\Theta \models_v v : \mathsf{SymK}^{\ell^\star}[T_1] \quad \dots}{\Theta \models_v dec(v', v) : T_1}\text{[dec]} \qquad T_1 \leq \mu\mathsf{K}^{\ell}[T]}{\Theta \models_v dec(v', v) : \mu\mathsf{K}^{\ell}[T]}\text{[sub]}$$

and let us analyze the possible cases for $\Theta \models_v dec(v', v) : \mu'\mathsf{K}^{\ell'}[T']$.

(a) Case [*dec*]: In this case

$$\dfrac{\Theta \models_v v : \mathsf{SymK}^{\ell'^\star}[\mu'\mathsf{K}^{\ell'}[T']] \quad \Theta \models_v v' : T_1' \quad v' \neq enc(v'', v)}{\Theta \models_v dec(v', v) : \mu'\mathsf{K}^{\ell'}[T']}\text{[dec]}$$

and by IH one has $\mu'\mathsf{K}^{\ell'}[T'] = T_1 \leq \mu\mathsf{K}^{\ell}[T]$ which implies by the same reasoning as case *(i)*-[*sub*] above that $T = T' = LL$ and $\ell' \leq \ell$.

(b) Case [*sub*]: In this case

$$\dfrac{\dfrac{\Theta \models_v v : \mathsf{SymK}^{\ell'^\star}[T_1'] \quad \dots}{\Theta \models_v dec(v', v) : T_1'}\text{[dec]} \qquad T_1' \leq \mu'\mathsf{K}^{\ell'}[T']}{\Theta \models_v dec(v', v) : \mu'\mathsf{K}^{\ell'}[T']}\text{[sub]}$$

and since $T_1 \leq \mu\mathsf{K}^{\ell}[T]$ and $T_1' \leq \mu'\mathsf{K}^{\ell'}[T']$ one has by definition of $\leq$ that $T = T' = LL$ and $\ell, \ell' \in \{HL, LL\}$ and the result follows.

The case $v'' = dec^a(v', v)$ is analogous using [*deca*] and Dec instead of [*dec*] and Sym. ■

Restatement of Proposition 15. *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 $\Theta \models_v v : LL$.

Proof: We prove by induction on the number of steps needed to derive $v \in \mathcal{K}(V)$.

Base case is when the derivation has length 0 hence $v \in V$. By hypothesis the result follows.

Suppose now that $v, v' \in \mathcal{K}(V)$ and by IH $\Theta \models_v v : LL$ and $\Theta \models_v v' : LL$. We consider all the cases in the definition of $v'' \in \mathcal{K}(V)$ and show that $\Theta \models_v v'' : LL$.

(1) $v'' = enc(v', v), enc^a(v', v), sig(v', v), ek(v), vk(v)$.

(i) Case $v'' = enc(v', v)$:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Theta \models_v v : LL}\text{[IH]} \quad LL \leq \mathsf{SymK}^{LL}[LL]}{\Theta \models_v v : \mathsf{SymK}^{LL}[LL]}\text{[sub]} \quad \dfrac{\vdots}{\Theta \models_v v' : LL}\text{[IH]}}{\Theta \models_v enc(v', v) : LL}\text{[enc]}$$

(ii) Case $v'' = enc^a(v', v)$: idem with Enc and rule [*enca*].

(iii) Case $v'' = sig(v', v)$: analogous with Sig and rule [*sig*].

(iv) Case $v'' = ek(v)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Theta \models_v v : LL}\text{[IH]} \quad LL \leq \mathsf{DecK}^{LL}[LL]}{\Theta \models_v v : \mathsf{DecK}^{LL}[LL]}\text{[sub]}}{\Theta \models_v ek(v) : \mathsf{EncK}^{LL}[LL]}\text{[ek]} \quad \mathsf{EncK}^{LL}[LL] \leq LL}{\Theta \models_v ek(v) : LL}\text{[sub]}$$

(v) Case $v'' = vk(v)$: analogous with Sig and Ver and rule [*vk*].

(2) $v'' \in \mathcal{K}(V)$ because $v = enc(v'', v')$ or $v = enc^a(v'', ek(v'))$.

(i) Case $v = enc(v'', v')$: analyzing the rules, [*enc*] and [*sub*] are the only possibilities for $\Theta \models_v enc(v'', v') : LL$.

(a) Considering the case [*enc*] we get

$$\dfrac{\Theta \models_v v' : \mathsf{SymK}^{\ell_C L}[T''] \quad \Theta \models_v v'' : T''}{\Theta \models_v enc(v'', v') : LL}\text{[enc]}$$

Since by IH $\Theta \models_v v' : LL$ we also have

$$\dfrac{\dfrac{\vdots}{\Theta \models_v v' : LL}\text{[IH]} \quad LL \leq \mathsf{SymK}^{LL}[LL]}{\Theta \models_v v' : \mathsf{SymK}^{LL}[LL]}\text{[sub]}$$

By Proposition 13 one has $T'' = LL$ which implies $\Theta \models_v v'' : LL$.

(b) Considering the case [*sub*] we get

$$\dfrac{\dfrac{\Theta \models_v v' : \mathsf{SymK}^{\ell_C \ell_I(T')}[T''] \quad \Theta \models_v v'' : T''}{\Theta \models_v enc(v'', v') : T'}\text{[enc]} \quad T' \leq LL}{\Theta \models_v enc(v'', v') : LL}\text{[sub]}$$

Similarly as in the case (2.*i.a*) above, by Proposition 13 one has $T'' = LL$ hence $\Theta \models_v v'' : LL$.

(ii) Case $v = enc^a(v'', ek(v'))$: analyzing the rules, [*enca*] and [*sub*] are only two possibilities for $\Theta \models_v enc^a(v'', ek(v')) : LL$.

(a) Considering the case [*enca*] we get

$$\dfrac{\Theta \models_v ek(v') : \mathsf{EncK}^{\ell_C L}[T''] \quad \Theta \models_v v'' : T''}{\Theta \models_v enc^a(v'', ek(v')) : LL}\text{[enca]}$$

By Proposition 11(3) we have $\ell_C L = LH$ or $T'' = LL$. Since the former is not true, the latter proves our claim.

(b) Considering now the case [*sub*] we get

$$\dfrac{\dfrac{\Theta \models_v ek(v') : \mathsf{EncK}^{\ell_C \ell_I(T')}[T''] \quad \Theta \models_v v'' : T''}{\Theta \models_v enc^a(v'', ek(v')) : T'}\text{[enca]} \quad T' \leq LL}{\Theta \models_v enc^a(v'', ek(v')) : LL}\text{[sub]}$$

Let us analyze the possible ways to derive $\Theta \models_v ek(v') : \mathsf{EncK}^{\ell_C \ell_I(T')}[T'']$. It is either by [*atom*], [*sub*] or [*ek*].

• Case [*atom*] is not possible because $ek(v')$ is not atomic.

• Case [*sub*] leads to $\Theta \models_v ek(v') : T'''$ for $T''' \leq \mathsf{EncK}^{\ell_C \ell_I(T')}[T'']$ which by definition of $\leq$ implies $T'' = LL$.

• Case [*ek*]:

$$\frac{\Theta \models_v v' : \mathsf{DecK}^{\ell_C{}' \ell_I(T')}[T'']}{\Theta \models_v ek(v') : \mathsf{EncK}^{\ell_C \ell_I(T')}[T'']}\ [\text{ek}]$$

By IH, similarly to case (2.i.a) one has $\Theta \models_v v' : \mathsf{DecK}^{LL}[LL]$ and by Proposition 13 we get $T'' = LL$.

(3) $v'' = dec(v', v) \in \mathcal{K}(V)$ and $v' \neq enc(v'', v)$.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Theta \models_v v : LL}\ [\text{IH}] \quad LL \leq \mathsf{SymK}^{LL}[LL]}{\Theta \models_v v : \mathsf{SymK}^{LL}[LL]}\ [\text{sub}] \quad \ldots \quad v' \neq enc(v'', v)}{\Theta \models_v dec(v', v) : LL}\ [\text{dec}]$$

(4) $v'' = dec^a(v', v) \in \mathcal{K}(V)$ and $v' \neq enc^a(v'', ek(v))$. Similar to case (3) using Dec and [*deca*] instead of Sym and [*dec*]. Notice that the side-condition of rule [*deca*] is verified since the payload $T = LL$.

(5) $v'' \in \mathcal{K}(V)$ because $v' = sig(v'', v''')$ and $v = vk(v''')$. By IH we know that $\Theta \models_v sig(v'', v''') : LL$. There are only 3 hypothesis to derive $v' = sig(v'', v''')$: either by [*atom*], by [*sig*], or by [*sub*].

(i) The first case, [*atom*], is not possible as $sig(v'', v''')$ is not atomic.

(ii) The second, [*sig*], implies that

$$\frac{\ldots \quad \Theta \models_v v'' : L{\ell_I}''}{\Theta \models_v sig(v'', v''') : LL}\ [\text{sig}]$$

and by [*sub*] $\Theta \models_v v'' : LL$.

(iii) Let us finally consider the case [*sub*]. In this case we have

$$\frac{\Theta \models_v sig(v'', v''') : T \quad T \leq LL}{\Theta \models_v sig(v'', v''') : LL}\ [\text{sub}]$$

Looking now at $\Theta \models_v sig(v'', v''') : T$ we know that it cannot be by [*atom*] because it is not atomic; cannot be by [*sub*] because we assumed we never apply rule [*sub*] twice in sequence; and so it can only be by [*sig*] which implies

$$\dfrac{\dfrac{\ldots \quad \Theta \models_v v'' : \ell_C(T){\ell_I}''}{\Theta \models_v sig(v'', v''') : T}\ [\text{sig}] \quad T \leq LL}{\Theta \models_v sig(v'', v''') : LL}\ [\text{sub}]$$

Now, if $T \leq LL$ we have $T = LL, LH, \mu\mathsf{K}^{LH}[T^\star]$, or $\mu\mathsf{K}^{LL}[T^\star]$ for some $T^\star$ which implies in any case that $\ell_C(T) = L$, hence $\Theta \models_v v'' : L{\ell_I}''$ that by [*sub*] proves our result.

■

Lemma 30. *Let Θ be a well-formedness function.*
If $\Theta \models_v v : T$ then T is ground.

Proof: The proof is by induction on the derivation of $\Theta \models_v v : T$.

Base case. Rule [*atom*]: In this case v is atomic and $\Theta(v) = T$. By construction of Θ, T is a ground type.

Inductive Step.

(i) Rule [*sub*]:

$$\frac{\Theta \models_v v : T' \quad T' \leq T}{\Theta \models_v v : T}\ [\text{sub}]$$

By induction hypothesis T' is ground. By definition of $\leq$, and since we do not use the rule [*sub*] uselessly we have that T is also ground and the result follows.

(ii) Rules [*ek*], [*vk*], [*dec*], and [*deca*]: follows directly from IH.

(iii) Rules [*enc*], [*enca*], and [*sig*]: immediate by construction.

■

Restatement of Proposition 16. *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 $\Theta \models_v v : T\sigma$.

Proof: By induction on the derivation $\Gamma \vdash_e e : T$.

Base case. Rule [*var*]: $e = x$ and $x \downarrow^{\mathsf{M}} v = \mathsf{M}(x)$.

$$\frac{\Gamma(x) = T \quad \Gamma \vdash \diamond}{\Gamma \vdash_e x : T}\ [\text{var}]$$

By well-formedness of M we get $\Theta \models_v v : T\sigma$.

Inductive Step.

(i) Rule [*sub*]: By hypothesis $\Gamma \vdash_e e : T$ and $e \downarrow^{\mathsf{M}} v$.

$$\frac{\Gamma \vdash_e e : T' \quad T' \leq T}{\Gamma \vdash_e e : T}\ [\text{sub}]$$

Applying IH one gets $\Theta \models_v v : T'\sigma$ and by Lemma 2 $T'\sigma \leq T\sigma$. So one can derive

$$\dfrac{\dfrac{\vdots}{\Theta \models_v v : T'\sigma}\ [\text{IH}] \quad T'\sigma \leq T\sigma}{\Theta \models_v v : T\sigma}\ [\text{sub}]$$

(ii) Rule [*ek*]: By hypothesis $\Gamma \vdash_e \mathsf{ek}(x) : \mathsf{EncK}^{L\ell_I}[T]$ and $\mathsf{ek}(x) \downarrow^{\mathsf{M}} ek(v)$ with $x \downarrow^{\mathsf{M}} v$.

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

By IH $\Theta \models_v v : \mathsf{DecK}^{\ell_C \ell_I}[T\sigma]$ and by [*ek*]

$$\dfrac{\dfrac{\vdots}{\Theta \models_v v : \mathsf{DecK}^{\ell_C \ell_I}[T\sigma]}\ [\text{IH}]}{\Theta \models_v ek(v) : \mathsf{EncK}^{L \ell_I}[T\sigma]}\ [\text{ek}]$$

(iii) Rule [*vk*]: Analogous replacing $\mathsf{Enc}, \mathsf{Dec}$, and [*ek*] by $\mathsf{Ver}, \mathsf{Sig}$, and [*vk*].

(iv) Rule [*enc*]: By hypothesis $\Gamma \vdash_e \mathsf{enc}(e, x) : L\ell_I$ and $\mathsf{enc}(e, x) \downarrow^{\mathsf{M}} enc(v', v)$ with $e \downarrow^{\mathsf{M}} v'$ and $x \downarrow^{\mathsf{M}} v$.

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

By IH $\Theta \models_v v' : T\sigma$ and $\Theta \models_v v : \mathsf{SymK}^{\ell_C \ell_I}[T\sigma]$ and by [*enc*]

$$\dfrac{\dfrac{\vdots}{\Theta \models_v v : \mathsf{SymK}^{\ell_C \ell_I}[T\sigma]}\ [\text{IH}] \quad \dfrac{\vdots}{\Theta \models_v v' : T\sigma}\ [\text{IH}]}{\Theta \models_v enc(v', v) : L\ell_I}\ [\text{enc}]$$

the result follows.

(v) Rule [*enca*]: Analogous to the case [*enc*].

(vi) Rule [*dec*]: By hypothesis $\Gamma \vdash_e \mathsf{dec}(e, x) : T$ and so

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

We have two cases:

(a) $x \downarrow^{\mathsf{M}} v$ and $e \downarrow^{\mathsf{M}} v''$ with $v'' \neq enc(v', v)$. In this case $\mathsf{dec}(e, x) \downarrow^{\mathsf{M}} dec(v'', v)$ and we want to show $\Theta \models_v dec(v'', v) : T\sigma$.
By IH $\Theta \models_v v : \mathsf{SymK}^{\ell}[T\sigma]$ and $\Theta \models_v v'' : T''\sigma$ and

$$\dfrac{\dfrac{\vdots}{\Theta \models_v v : \mathsf{SymK}^{\ell}[T\sigma]}\ [\text{IH}] \quad \dfrac{\vdots}{\Theta \models_v v'' : T''\sigma}\ [\text{IH}] \quad v'' \neq enc(v', v)}{\Theta \models_v dec(v'', v) : T\sigma}\ [\text{dec}]$$

proves the result.

(b) $x \downarrow^{\mathsf{M}} v$ and $e \downarrow^{\mathsf{M}} enc(v', v)$. In this case $\mathsf{dec}(e, x) \downarrow^{\mathsf{M}} v'$ and we want to show that $\Theta \models_v v' : T\sigma$.
By IH $\Theta \models_v v : \mathsf{SymK}^{\ell}[T\sigma]$ and $\Theta \models_v enc(v', v) : T''\sigma$. Let us analyze this last derivation. It was either by [*enc*] or [*sub*] because $enc(v', v)$ is not atomic.

- Suppose it was by [*enc*]:

$$\dfrac{\Theta \models_v v : \mathsf{SymK}^{\ell_C \ell_I(T''\sigma)}[T'] \quad \Theta \models_v v' : T'}{\Theta \models_v enc(v', v) : T''\sigma}\ [\text{enc}]$$

for some ground T'. Applying Proposition 13 to v we get $T\sigma = T'$ and so the result follows.

- If one used [*sub*] then

$$\dfrac{\Theta \models_v enc(v', v) : T''' \quad T''' \leq T''\sigma}{\Theta \models_v enc(v', v) : T''\sigma}\ [\text{sub}]$$

for some ground T'''. Now, since we do not have useless applications of [*sub*], the only possibility for $\Theta \models_v enc(v', v) : T'''$ is by [*enc*] and so, applying the reasoning just used before now with $\Theta \models_v enc(v', v) : T'''$ instead of $T''\sigma$ the result follows.

(vii) Rule [*deca*]: By hypothesis $\Gamma \vdash_e \mathsf{dec}^a(e, x) : T$ and so

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

We have again two cases:

(a) $x \downarrow^{\mathsf{M}} v$ and $e \downarrow^{\mathsf{M}} v''$ with $v'' \neq enc^a(v', ek(v))$. In this case $\mathsf{dec}^a(e, x) \downarrow^{\mathsf{M}} dec^a(v'', v)$ and we want to show $\Theta \models_v dec^a(v'', v) : T\sigma$.
By IH $\Theta \models_v v'' : T''\sigma$ and $\Theta \models_v v : \mathsf{DecK}^{\ell}[T\sigma]$ and so

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

proves the result if the side-condition is verified. Suppose that $\ell_I(T''\sigma) \neq H$, ie, $\ell_I(T''\sigma) = L$. We analyze the possible cases for T'':

- $T'' = X$ implies $\ell_I(T'') = \ell_I(X) = L = \ell_I(T''\sigma)$;
- $T'' = \ell$ implies $\ell_I(T'') = \ell_I(\ell) = \ell_I(T''\sigma)$;
- $T'' = \mu\mathsf{K}^{\ell}[T^\star]$ implies $\ell_I(T'') = \ell_I(\mu\mathsf{K}^{\ell}[T^\star]) = \ell_I(\ell) = \ell_I(\mu\mathsf{K}^{\ell}[T^\star\sigma]) = \ell_I(T''\sigma)$.

Hence $\ell_I(T''\sigma) = L$ implies $\ell_I(T'') = L$; by the side-condition of $\Gamma \vdash_e \mathsf{dec}^a(e, x) : T$ we have $T = LL$; and finally $T\sigma = T = LL$.

(b) $x \downarrow^{\mathsf{M}} v$ and $e \downarrow^{\mathsf{M}} enc^a(v', ek(v))$. In this case $\mathsf{dec}^a(e, x) \downarrow^{\mathsf{M}} v'$ and we want to show that $\Theta \models_v v' : T\sigma$.
By IH $\Theta \models_v v : \mathsf{DecK}^{\ell}[T\sigma]$ and $\Theta \models_v enc^a(v', ek(v)) : T''\sigma$. Let us analyze this last derivation. It was either by [*enca*] or [*sub*] because $enc^a(v', ek(v))$ is not atomic.

- Suppose it was by [*enca*]:

$$\dfrac{\Theta \models_v ek(v) : \mathsf{EncK}^{\ell_C \ell_I(T''\sigma)}[T'] \quad \Theta \models_v v' : T'}{\Theta \models_v enc^a(v', ek(v)) : T''\sigma}\ [\text{enca}]$$

We can now look at the possibilities for obtaining $ek(v)$ and see that we have again two cases: [*ek*] or [*sub*].

- for [*ek*] one has

$$\frac{\Theta \models_v v : \mathsf{DecK}^{\ell_C{}' \ell_I(T''\sigma)}[T']}{\Theta \models_v ek(v) : \mathsf{EncK}^{\ell_C \ell_I(T''\sigma)}[T']} \text{[ek]}$$

and applying Proposition 13 to v we get $T\sigma = T'$ and the result follows.

- for [*sub*] one has

$$\frac{\Theta \models_v ek(v) : T''' \quad T''' \leq \mathsf{EncK}^{\ell_C \ell_I(T''\sigma)}[T']}{\Theta \models_v ek(v) : \mathsf{EncK}^{\ell_C \ell_I(T''\sigma)}[T']} \text{[sub]}$$

for some ground T'''. Since we have no useless applications of [*sub*], we have by $\leq$ that $\ell_I(T''\sigma) = L$ and $T' = LL$.

As seen above, case *(vii.a)*, $\ell_I(T''\sigma) = L$ implies $\ell_I(T'') = L$ hence the side-condition of [*deca*] implies that $T = LL$ and so $\Theta \models_v v' : T' = LL = T = T\sigma$ as we wanted.

- if $\Theta \models_v enc^a(v', ek(v)) : T''\sigma$ was derived by [*sub*] then

$$\frac{\Theta \models_v enc^a(v', ek(v)) : T^\star \quad T^\star \leq T''\sigma}{\Theta \models_v enc^a(v', ek(v)) : T''\sigma} \text{[sub]}$$

for some ground $T^\star$. Now, the only possibility for $\Theta \models_v enc^a(v', ek(v)) : T^\star$ is by [*enca*] and so, applying the reasoning just used above in case *(vii.b)*-[*enca*] now with $\Theta \models_v enc^a(v', ek(v)) : T^\star$ instead of $T''\sigma$ the same argument follows up to the point in [*sub*] where we can show now that $\ell_I(T^\star) = L$ rather than the needed $\ell_I(T''\sigma) = L$. However, given that $\ell_I(T^\star) = L$ and $T^\star \leq T''\sigma$ one can see analyzing all possible cases that $\ell_I(T''\sigma) = L$ and so we can continue with the same argument to obtain our result.

(viii) Rule [*sig*]: By hypothesis $\Gamma \vdash_e \mathsf{sig}(e, x) : \ell_C(T)\ell_I$ and $\mathsf{sig}(e, x) \downarrow^{\mathsf{M}} sig(v', v)$ with $e \downarrow^{\mathsf{M}} v'$ and $x \downarrow^{\mathsf{M}} v$.

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

and so we want to show $\Theta \models_v sig(v', v) : \ell_C(T)\ell_I$. By IH $\Theta \models_v v' : T\sigma$ and $\Theta \models_v v : \mathsf{SigK}^{\ell_C \ell_I}[T\sigma]$ and by [*sig*]

$$\frac{\dfrac{\vdots}{\Theta \models_v v : \mathsf{SigK}^{\ell_C \ell_I}[T\sigma]} \text{[IH]} \quad \dfrac{\vdots}{\Theta \models_v v' : T\sigma} \text{[IH]}}{\Theta \models_v sig(v', v) : \ell_C(T\sigma)\ell_I} \text{[enc]}$$

If $T \neq X$ then $\ell_C(T\sigma) = \ell_C(T)$ and the result immediately follows. If not, notice that $T = X$ implies $\ell_C(T\sigma) \sqsubseteq_C H = \ell_C(T)$, hence $\ell_C(T\sigma)\ell_I \leq \ell_C(T)\ell_I$ and the result follows by [*sub*].

(ix) Rule [*ver*]: By hypothesis $\Gamma \vdash_e \mathsf{ver}(e, x) : T$ and so

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

Since $\mathsf{ver}(e, x) \downarrow^{\mathsf{M}} v'$ we also have that $e \downarrow^{\mathsf{M}} sig(v', v)$ and $x \downarrow^{\mathsf{M}} vk(v)$. We want to show that $\Theta \models_v v' : T\sigma$.

By IH $\Theta \models_v sig(v', v) : T'\sigma$ and $\Theta \models_v vk(v) : \mathsf{VerK}^{\ell_C \ell_I}[T\sigma]$. We now look at the possible derivations of $sig(v', v)$. It is either by [*sig*] or [*sub*] as $sig(v', v)$ is not atomic.

(a) [*sig*]: Since $\Theta \models_v sig(v', v) : T'\sigma$ we have

$$\frac{\Theta \models_v v' : \ell_C(T'\sigma)\ell_I{}' \quad \Theta \models_v v : \mathsf{SigK}^{\ell_C{}' \ell_I(T'\sigma)}[\ell_C(T'\sigma)\ell_I{}']}{\Theta \models_v sig(v', v) : T'\sigma} \text{[sig]}$$

Now, from $\Theta \models_v vk(v) : \mathsf{VerK}^{\ell_C \ell_I}[T\sigma]$ we know that this could only be obtained either by [*vk*] or [*sub*] since $vk(v)$ is not atomic.

- Consider the first case [*vk*]. We have then

$$\frac{\Theta \models_v v : \mathsf{SigK}^{\ell_C{}^\star \ell_I}[T\sigma]}{\Theta \models_v vk(v) : \mathsf{VerK}^{\ell_C \ell_I}[T\sigma]} \text{[vk]}$$

which implies by Proposition 13 that $T\sigma = \ell_C(T'\sigma)\ell_I{}'$ hence $\Theta \models_v v' : T\sigma$.

- Consider now the case [*sub*]. In this case

$$\frac{\Theta \models_v vk(v) : T'' \quad T'' \leq \mathsf{VerK}^{\ell_C \ell_I}[T\sigma]}{\Theta \models_v vk(v) : \mathsf{VerK}^{\ell_C \ell_I}[T\sigma]} \text{[sub]}$$

for some ground T''. This implies by definition of $\leq$ that $T\sigma = LL$ and $\ell_I = L$. Applying the side-condition in the contra-positive way one has $\ell_C(T') = L$ and consequently $\ell_C(T'\sigma) = L$.

Using this in the [*sig*]-rule above we get $\Theta \models_v v' : L\ell_I$ and by [*sub*] $\Theta \models_v v' : LL = T\sigma$.

(b) [*sub*]: Since $\Theta \models_v sig(v', v) : T'\sigma$ we have

$$\frac{\Theta \models_v sig(v', v) : T^\star \quad T^\star \leq T'\sigma}{\Theta \models_v sig(v', v) : T'\sigma} \text{[sub]}$$

for some ground $T^\star$. Now, the only possibility for $\Theta \models_v sig(v', v) : T^\star$ is by [*sig*] and so, applying the reasoning just used in *(ix.a)* now with $\Theta \models_v sig(v', v) : T^\star$ instead of $T'\sigma$ the argument follows until the point where we still derive by the side-condition that $\ell_C(T'\sigma) = \ell_C(T') = L$ but what we need for the argument to go through is that $\ell_C(T^\star) = L$. However, given that $\ell_C(T'\sigma) = L$ and $T^\star \leq T'\sigma$ one can analyze all possible cases and conclude that $\ell_C(T^\star) = L$ allowing us to substitute in the [*sig*]-rule above $\Theta \models_v v' : \ell_C(T^\star)\ell_I$ by $L\ell_I$ and conclude the argument in the same way.

■

Restatement of Theorem 17. *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: 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 16 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 $\mathsf{c} = x := \mathsf{genKey}(T)$:

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

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 $g' \notin ran[\mathsf{M}]$ 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 16 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 $\mathsf{M}(y) \in ran(\mathsf{M})$ 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 18. *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 15 that $\Theta_n \models_v v_i : LL$.

Unfolding the operation call we get that $\mathsf{a} = \lambda x_1 \ldots x_k.\mathsf{c}$, $\langle \mathsf{M}_\epsilon[x_i \mapsto v_i], \mathsf{H}_n, \emptyset, \mathsf{c} \rangle \rightarrow \langle \mathsf{M}, \mathsf{H}, \sigma, \mathsf{return}\ e \rangle$ 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 17 to $\langle \mathsf{M}_\epsilon[x_i \mapsto v_i], \mathsf{H}_n, \emptyset, \mathsf{c} \rangle \rightarrow \langle \mathsf{M}, \mathsf{H}, \sigma, \mathsf{return}\ e \rangle$ 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 $\mathsf{V} = \mathsf{V}_{\mathsf{ok}}(\mathsf{H}_n, \mathsf{V}_n) \cup \{ val \mid \exists g \in dom(\mathsf{H}) \setminus dom(\mathsf{H}_n).\mathsf{H}(g) = (val, T)\} \setminus ran[\mathsf{M}_\epsilon[x_i \mapsto v_i]]$

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 16 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 $\Theta \Vdash_{\mathsf{V}} \mathsf{H}, \mathsf{V}_{\mathsf{ok}}(\mathsf{H}_n, V_n)$ (see Table VII) ■

$$
\begin{array}{rl}
\mathsf{V} = & \mathsf{V_{ok}}(\mathsf{H}_n, \mathsf{V}_n) \cup \{val \mid \exists g \in dom(\mathsf{H}) \setminus dom(\mathsf{H}_n).\mathsf{H}(g) = (val, T)\} \setminus ran[\mathsf{M}_\epsilon[x_i \mapsto v_i]] \\
= & \mathsf{V_{ok}}(\mathsf{H}_n, \mathsf{V}_n) \cup \{val \mid \exists g \in dom(\mathsf{H}) \setminus dom(\mathsf{H}_n).\mathsf{H}(g) = (val, T)\} \setminus \{v_1, \ldots, v_k\} \\
\supseteq & \mathsf{V_{ok}}(\mathsf{H}_n, \mathsf{V}_n) \cup \{val \mid \exists g \in dom(\mathsf{H}) \setminus dom(\mathsf{H}_n).\mathsf{H}(g) = (val, T)\} \setminus \mathcal{K}(V_n) \\
= & \{val \mid \exists g.\mathsf{H}_n(g) = (val, T)\} \setminus \mathcal{K}(V_n) \cup \{val \mid \exists g \in dom(\mathsf{H}) \setminus dom(\mathsf{H}_n).\mathsf{H}(g) = (val, T)\} \setminus \mathcal{K}(V_n) \\
= & (\{val \mid \exists g.\mathsf{H}_n(g) = (val, T)\} \cup \{val \mid \exists g \in dom(\mathsf{H}) \setminus dom(\mathsf{H}_n).\mathsf{H}(g) = (val, T)\}) \setminus \mathcal{K}(V_n) \\
\supseteq & \{val \mid \exists g.\mathsf{H}(g) = (val, T)\} \setminus \mathcal{K}(V) \\
= & \mathsf{V_{ok}}(\mathsf{H}, V)
\end{array}
$$

Table VII
AUXILIARY TABLE FOR PROOF OF LEMMA 18