Towards a Type System for Security APIs

Gavin Keighren[1], David Aspinall[1], and Graham Steel[2]

[1] Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, UK
gavin.keighren@ed.ac.uk david.aspinall@ed.ac.uk

[2] LSV, INRIA & CNRS & ENS de Cachan
61, avenue du Président Wilson
94235 CACHAN Cedex – France
graham.steel@lsv.ens-cachan.fr

Abstract. Security API analysis typically only considers a subset of an API's functions, with results bounded by the number of function calls. Furthermore, attacks involving partial leakage of sensitive information are usually not covered. Type-based static analysis has the potential to alleviate these shortcomings. To that end, we present a type system for secure information flow based upon the one of Volpano, Smith and Irvine [1], extended with types for cryptographic keys and ciphertext similar to those in Sumii and Pierce [2]. In contrast to some other type systems, the encryption and decryption of keys does not require special treatment. We show that a well-typed sequence of commands is non-interferent, based upon a definition of indistinguishability where, in certain circumstances, the adversary can distinguish between ciphertexts that correspond to encrypted public data.

1 Introduction

It is common for computer systems which store, process and manipulate sensitive data to use a dedicated security hardware device (e.g., IBM 4758 [3] and nCipher nShield [4]). The set of functions provided by a security device is termed its *security API*, as they are intended to enforce a security policy as well as provide an interface to the device. A security policy describes the restrictions on the access to, use of, and propagation of data in the system. These restrictions, therefore, must follow as a direct consequence of the API functions which are available to users of the security device.

The analysis of security APIs has traditionally been carried out by enumerating the set of data items which the adversary (a malicious user) can obtain through repeated interactions with the API. While this approach has had reasonable success (e.g., [5–9]), results are typically bounded by the number of API calls, do not consider data integrity, and only detect flaws which involve the release of sensitive data items in their entirety.

In contrast, static analysis has the potential to provide unbounded results, identify flaws which allow for sensitive data to be leaked via covert control-flow channels, and also deal with data integrity. The type system presented in this paper is the foundation of one such approach, although it does not yet deal with integrity.

Our work builds upon the information-flow analysis capabilities of Volpano, Smith and Irvine's type system [1] by including cryptographic types similar to those from Sumii and Pierce's system for analysing security protocols [2]. Although there are many similarities between security APIs and security protocols, analysis methods for the latter are typically designed to deal with fixed-length specified interactions, and therefore generally do not scale well when applied to arbitrary sequences of interactions.

2 Background

Hardware Security Modules (HSMs) comprise some memory and a processor inside a tamper-proof enclosure which prevents the memory contents from being physically read — any breach causes the memory to be erased within a few micro-seconds. Additional storage is provided by the host system to which the HSM is attached. This leads to a natural partition of memory locations: those inside the HSM are high security, and those on the host system are low security.

Memory locations on the host system are deemed low security, since the attack model for security API analysis assumes that the adversary has full control of the host system. In addition, the adversary is assumed to be capable of calling certain functions provided by the HSM's security API (because, for example, they have hijacked a user's session, or they are a legitimate user themselves).

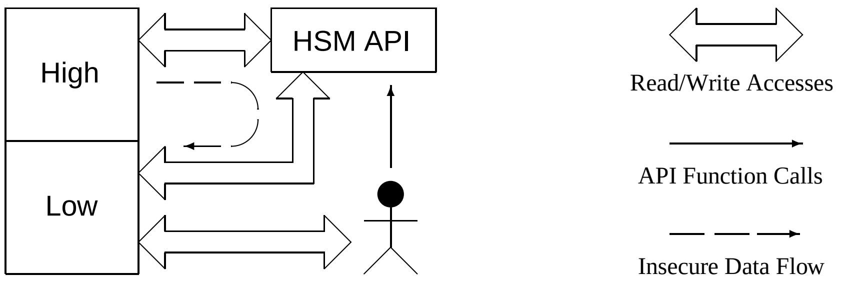

Fig. 1. The interactions between the adversary, the HSM API, and the memory locations.

Figure 1 shows the interactions between the adversary, HSM API, and memory locations in the standard attack scenario. HSM functions may access any memory locations, while the adversary can only access the low security locations. A similar setup applies in the case of software APIs, where the adversary is a malicious client program and the high memory locations correspond to those which are hidden by the API.

The adversary's goal is to execute a series of API function calls such that sensitive data is unintentionally written to the low security memory, or that sensitive data can be inferred from the API's low security output. The aim of security API analysis is to detect such insecure data flows, or to guarantee that no such flows exist.

n	$\in$	$\mathcal{N}$	*Names*
a	$\in$	$\mathcal{A}$	*Locations*
l	$\in$	$\mathcal{L}$	*Security levels* (where $\bot, \top \in \mathcal{L}$)
e	$::=$	$n \mid\, !a \mid \mathrm{senc}(e,e) \mid \mathrm{sdec}(e,e) \mid \mathrm{junk}(e)$	*Expressions*
c	$::=$	$a := e \mid c\, ; c \mid \varepsilon$	*Commands*
u	$::=$	$n \mid \mathrm{senc}(u,u)$	*Non-junk values*
v	$::=$	$u \mid \mathrm{junk}(u)$	*Values*
E	$::=$	$l\ \mathsf{data} \mid l\ \mathsf{key} \mid \mathsf{enc}(\mathsf{E})$	*Security types for expressions*
A	$::=$	$\mathsf{E}\ \mathsf{loc}$	*Security type for locations*
C	$::=$	cmd	*Security type for commands*
T	$::=$	$\mathsf{E} \mid \mathsf{A} \mid \mathsf{C}$	*All security types*
ϕ	$::=$	$\phi, a \mapsto v \mid \varepsilon$	*Store*
Δ	$::=$	$\Delta, n : l \mid \Delta, a : l \mid \varepsilon$	*Security levels environment*
Γ	$::=$	$\Gamma, n : l\ \mathsf{data} \mid \Gamma, n : l\ \mathsf{key} \mid \Gamma, a : \mathsf{A} \mid \varepsilon$	*Security types environment*

Fig. 2. Fundamental syntax definitions.

3 Type System

Figure 2 presents the fundamental syntax definitions upon which our type system is built. The set $\mathcal{N}$ comprises *names* representing regular data items and cryptographic keys; $\mathcal{A}$ is the set of abstract memory *locations*, and $\mathcal{L}$ is the set of security levels which may be associated to names and locations.[3] Although our type system allows for any number of security levels (where $l \in \mathcal{L} \rightarrow \bot \leq l \leq \top$), in this paper we only consider $\bot$ and $\top$ (i.e., low and high) in order to simplify the presentation and discussion.

An *expression* can be a name, the contents of a memory location, the result of a (symmetric) encryption or decryption, or 'junk' which denotes an incorrectly decrypted term. Junk terms contain the expression which would have resulted had the correct decryption key(s) been used, so we can ensure that a junk expression is treated in the same way as the intended non-junk term. A *command* is zero or more instances of the assignment operation. A *value* is a name, a fully evaluated ciphertext, or a junk value.

The security types for regular data items and cryptographic keys, $l\ \mathsf{data}$ and $l\ \mathsf{key}$ respectively, associate a security level with their terms. The level associated with regular data denotes the confidentiality level of that data item, whereas the one associated with a cryptographic key denotes the maximum security level of expressions which that key may encrypt. The different semantics are necessary because we allow the security level of expressions to be arbitrarily increased and therefore cannot determine what a key may encrypt based solely on its security level. For example, a low security key can be considered a high security key and thus could be used to encrypt other high security expressions. This approach also allows a high security key to encrypt and decrypt low security expressions without forcing the result of the decryption to be high security.

[3] Security levels are for analysis purposes only — they do not exist in practice (and even if they did, the adversary would not gain anything from knowing a term's security level).

To recover the precise type of encrypted data when it is subsequently decrypted, we use a type operator; enc(E) is the type of an encrypted expression of type E. This means that no type information is lost as a result of the encryption/decryption process and also allows us to handle nested encryptions. The security type for locations denotes the most general type of expression which may be stored in that location. We do not associate a security level with the command type, although we will do so in future work.

The store, ϕ, maps locations to values; the security levels environment, Δ, contains the security levels of names and locations (used dynamically), and the security types environment, Γ, contains the security types of names and locations (used statically). We assume there is no overlap between the identifiers used for names and for locations.

3.1 Operational Semantics

Figure 3 presents the operational semantics for command sequences which we consider. We wish to enforce the restriction that only ciphertext is decrypted, therefore any term which evaluates to the decryption of non-encrypted data will get 'stuck.' That is, the term cannot be fully evaluated under the operational semantics.

Other terms which will get stuck include the encryption of data with a key whose security level is too low, the assignment of a value to a location whose security level is too low, and the dereferencing of something other than a location. The first two of these correspond to cases where continued evaluation would result in a security breach:

$$\Delta = \{a : \bot, v_k : \bot,\ v_m : \top\} \qquad \begin{array}{ll} a := \text{senc}(v_k, v_m) & (1) \\ a := v_m & (2) \end{array}$$

Getting stuck in case (1) guarantees that, if the adversary is able to decrypt some given piece of ciphertext, the result will not be sensitive, while getting stuck in case (2) guarantees that sensitive data cannot be written directly to a low security memory location. This latter property is known as 'no-write down,' and is enforced by the rule E-ASSIGN2. The 'no read-up' property follows from the assumption that an observer is only able to read the contents of locations whose associated security level is low enough. This is a legitimate assumption, since the sensitive locations will be those which are inside the tamper-proof HSM whose security API is being analysed, or in the case of software APIs, those locations which are hidden from client programs.

The junk term is returned when a piece of ciphertext is decrypted with a different key from the one which was used to create it (E-SDEC4), or when the key or message in an encryption or decryption operation is junk (E-SENC3, E-SENC4, E-SDEC5 and E-SDEC6). In all cases, the expression within the junk term is that which would have been returned had the correct decryption key(s) been used.

Encryption requires that the security level of the key is at least as high as that of the message. However, this restriction is not enforced when the ciphertext is actually created, but rather when it is subsequently decrypted or assigned to a location (i.e., when the security level of the ciphertext has to be determined). The security level of ciphertext is $\bot$ since encryption is used primarily as a means of securely declassifying sensitive data. If the result of an encryption should itself be sensitive, then this can be achieved simply by assigning the ciphertext to a location which stores sensitive data and returning a reference to that location.

Expressions $\boxed{e \rightarrow_{\Delta\phi} e'}$

$$\frac{e_k \rightarrow_{\Delta\phi} e_k{}'}{\mathrm{senc}(e_k, e_m) \rightarrow_{\Delta\phi} \mathrm{senc}(e_k{}', e_m)}\ \text{E-SEnc1}$$

$$\frac{e \rightarrow_{\Delta\phi} e'}{\mathrm{senc}(v, e) \rightarrow_{\Delta\phi} \mathrm{senc}(v, e')}\ \text{E-SEnc2}$$

$$\frac{}{\mathrm{senc}(u, \mathrm{junk}(u')) \rightarrow_{\Delta\phi} \mathrm{junk}(\mathrm{senc}(u, u'))}\ \text{E-SEnc3}$$

$$\frac{}{\mathrm{senc}(\mathrm{junk}(u), v) \rightarrow_{\Delta\phi} \mathrm{junk}(\mathrm{senc}(u, v))}\ \text{E-SEnc4}$$

$$\frac{e_k \rightarrow_{\Delta\phi} e_k{}'}{\mathrm{sdec}(e_k, e_m) \rightarrow_{\Delta\phi} \mathrm{sdec}(e_k{}', e_m)}\ \text{E-SDec1}$$

$$\frac{e \rightarrow_{\Delta\phi} e'}{\mathrm{sdec}(v, e) \rightarrow_{\Delta\phi} \mathrm{sdec}(v, e')}\ \text{E-SDec2}$$

$$\frac{\Delta \vdash u_k : l_k \quad \Delta \vdash u_m : l_m \quad l_m \leq l_k}{\mathrm{sdec}(u_k, \mathrm{senc}(u_k, u_m)) \rightarrow_{\Delta\phi} u_m}\ \text{E-SDec3}$$

$$\frac{u_k' \neq u_k \quad \Delta \vdash u_k', u_k : l_k \quad \Delta \vdash u_m : l_m \quad l_m \leq l_k}{\mathrm{sdec}(u_k', \mathrm{senc}(u_k, u_m)) \rightarrow_{\Delta\phi} \mathrm{junk}(u_m)}\ \text{E-SDec4}$$

$$\frac{}{\mathrm{sdec}(u, \mathrm{junk}(u')) \rightarrow_{\Delta\phi} \mathrm{junk}(\mathrm{sdec}(u, u'))}\ \text{E-SDec5}$$

$$\frac{}{\mathrm{sdec}(\mathrm{junk}(u), v) \rightarrow_{\Delta\phi} \mathrm{junk}(\mathrm{sdec}(u, v))}\ \text{E-SDec6}$$

$$\frac{}{!a \rightarrow_{\Delta\phi} \phi(a)}\ \text{E-Deref}$$

$$\frac{e \rightarrow_{\Delta\phi} e'}{\mathrm{junk}(e) \rightarrow_{\Delta\phi} \mathrm{junk}(e')}\ \text{E-Junk1}$$

$$\frac{}{\mathrm{junk}(\mathrm{junk}(u)) \rightarrow_{\Delta\phi} \mathrm{junk}(u)}\ \text{E-Junk2}$$

Commands $\boxed{\langle \phi, c \rangle \rightarrow_{\Delta} \langle \phi', c' \rangle}$

$$\frac{\langle \phi, c_1 \rangle \rightarrow_{\Delta} \langle \phi', c_1' \rangle}{\langle \phi, c_1 ; c_2 \rangle \rightarrow_{\Delta} \langle \phi', c_1' ; c_2 \rangle}\ \text{E-Cmds1}$$

$$\frac{}{\langle \phi, \varepsilon ; c \rangle \rightarrow_{\Delta} \langle \phi, c \rangle}\ \text{E-Cmds2}$$

$$\frac{e \rightarrow_{\Delta\phi} e'}{\langle \phi, a := e \rangle \rightarrow_{\Delta} \langle \phi, a := e' \rangle}\ \text{E-Assign1}$$

$$\frac{\Delta \vdash a : l_a \quad \Delta \vdash v : l_v \quad l_v \leq l_a}{\langle \phi, a := v \rangle \rightarrow_{\Delta} \langle \phi[a \mapsto v], \varepsilon \rangle}\ \text{E-Assign2}$$

Security Levels of Values

$$\frac{n : l \in \Delta}{\Delta \vdash n : l} \qquad \frac{a : l \in \Delta}{\Delta \vdash a : l} \qquad \frac{\Delta \vdash u_k : l_k \quad \Delta \vdash u_m : l_m \quad l_m \leq l_k}{\Delta \vdash \mathrm{senc}(u_k, u_m) : \bot} \qquad \frac{\Delta \vdash u : l}{\Delta \vdash \mathrm{junk}(u) : l}$$

Fig. 3. The operational semantics for command sequences.

3.2 Typing Rules

Figure 4 presents the rules of our type system. As noted previously, a location's type denotes the most general type of values which can be stored in that location. By design, the more general a type is, the greater its security level (i.e., $\mathsf{E} <: \mathsf{E}' \rightarrow lvl(\mathsf{E}) \leq lvl(\mathsf{E}')$). Therefore, the typing rule for assignment (T-Assign) guarantees the 'no write-down' property since the security level associated with the location will be no lower than the one associated with the expression.

Junk terms can have any expression type (T-Junk), since they are generated only as the result of a decryption with the wrong key, and we wish to consider a junk term as being equivalent to the intended result, had the correct decryption key been used. This is to prevent insecure information flows which may otherwise result from the use of an incorrect decryption key.

Commands

$$\frac{\Gamma \vdash c_1 : \mathsf{cmd} \quad \Gamma \vdash c_2 : \mathsf{cmd}}{\Gamma \vdash c_1\,;c_2 : \mathsf{cmd}}\ \text{T-CMDS}$$

$$\frac{\Gamma \vdash a : \mathsf{E\ loc} \quad \Gamma \vdash e : \mathsf{E}}{\Gamma \vdash a := e : \mathsf{cmd}}\ \text{T-ASSIGN} \qquad \frac{}{\Gamma \vdash \varepsilon : \mathsf{cmd}}\ \text{T-EMPTY}$$

Expressions

$$\frac{n : \mathsf{E} \in \Gamma}{\Gamma \vdash n : \mathsf{E}}\ \text{T-NAME} \qquad \frac{\Gamma \vdash e : \mathsf{E}}{\Gamma \vdash \mathrm{junk}(e) : \mathsf{E}}\ \text{T-JUNK}$$

$$\frac{a : \mathsf{E\ loc} \in \Gamma}{\Gamma \vdash a : \mathsf{E\ loc}}\ \text{T-LOC} \qquad \frac{\Gamma \vdash a : \mathsf{E\ loc}}{\Gamma \vdash !a : \mathsf{E}}\ \text{T-DEREF}$$

$$\frac{\Gamma \vdash e_k : l\ \mathsf{key} \quad \Gamma \vdash e_m : \mathsf{E} \quad lvl(\mathsf{E}) = l}{\Gamma \vdash \mathrm{senc}(e_k, e_m) : \mathsf{enc(E)}}\ \text{T-SENC}$$

$$\frac{\Gamma \vdash e_k : l\ \mathsf{key} \quad \Gamma \vdash e_m : \mathsf{enc(E)} \quad lvl(\mathsf{E}) = l}{\Gamma \vdash \mathrm{sdec}(e_k, e_m) : \mathsf{E}}\ \text{T-SDEC}$$

Subtyping

$$\frac{\Gamma \vdash t : \mathsf{T}' \quad \mathsf{T}' <: \mathsf{T}}{\Gamma \vdash t : \mathsf{T}}\ \text{T-SUB} \qquad \frac{}{\mathsf{T} <: \mathsf{T}}$$

$$\frac{\mathsf{T} <: \mathsf{T}'' \quad \mathsf{T}'' <: \mathsf{T}'}{\mathsf{T} <: \mathsf{T}'}$$

$$\frac{l \leq l'}{l\ \mathsf{data} <: l'\ \mathsf{data}} \qquad \frac{\mathsf{E} <: \mathsf{E}'}{\mathsf{enc(E)} <: \mathsf{enc(E')}}$$

$$\frac{}{l\ \mathsf{key} <: \top\ \mathsf{data}} \qquad \frac{}{\mathsf{enc(E)} <: \bot\ \mathsf{data}}$$

Security Levels of Types

$lvl(\mathsf{cmd}) = \bot$ $\qquad lvl(l\ \mathsf{data}) = l$

$lvl(l\ \mathsf{key}) = \top$ $\qquad lvl(\mathsf{enc(E)}) = \bot$

$lvl(\mathsf{E\ loc}) = lvl(\mathsf{E})$

Fig. 4. The typing rules of our system.

The contents of a location are given the type of the most general expression that can be stored in that location (T-DEREF). Thus, any security result is independent of the values which are actually stored in each memory location.

For encryption, the key used must be able to encrypt messages which are at least as secure as the actual message (T-SENC). For decryption, the message must be ciphertext and the security level associated with the key must be no lower than the security level associated with the result (T-SDEC).

Currently, we restrict keys to having the highest security level, and commands to having the lowest security level, since our focus is on security APIs with secret keys and public functions. Relaxing these restrictions will form part of our future work.

To prove the theorems presented in this paper, we require a couple of standard type-theoretic lemmas. The proofs are quite straightforward and have been omitted.

Lemma 1. *Generation Lemma (Inversion of the Typing Relation)*

1. If $\Gamma \vdash n : \mathsf{T}$ then $\mathsf{T} :> \mathsf{E}$ and $n : \mathsf{E} \in \Gamma$.
2. If $\Gamma \vdash a : \mathsf{T}$ then $\mathsf{T} \equiv \mathsf{E\ loc}$ and $a : \mathsf{E\ loc} \in \Gamma$.
3. If $\Gamma \vdash !a : \mathsf{T}$ then $\mathsf{T} :> \mathsf{E}$ and $\Gamma \vdash a : \mathsf{E\ loc}$.
4. If $\Gamma \vdash \mathrm{senc}(e_1, e_2) : \mathsf{T}$ then $\mathsf{T} :> \mathsf{enc(E)}$, $\Gamma \vdash e_1 : l\ \mathsf{key}$, $\Gamma \vdash e_2 : \mathsf{E}$ and $lvl(\mathsf{E}) = l$.
5. If $\Gamma \vdash \mathrm{sdec}(e_1, e_2) : \mathsf{T}$ then $\mathsf{T} :> \mathsf{E}$, $\Gamma \vdash e_1 : l\ \mathsf{key}$, $\Gamma \vdash e_2 : \mathsf{enc(E)}$ and $lvl(\mathsf{E}) = l$.
6. If $\Gamma \vdash \mathrm{junk}(e) : \mathsf{T}$ then $\Gamma \vdash e : \mathsf{T}$.
7. If $\Gamma \vdash a := e : \mathsf{T}$ then $\mathsf{T} \equiv \mathsf{cmd}$, $\Gamma \vdash a : \mathsf{E\ loc}$, and $\Gamma \vdash e : \mathsf{E}$.

8. If $\Gamma \vdash \varepsilon : \mathsf{T}$ then $\mathsf{T} \equiv \mathsf{cmd}$.
9. If $\Gamma \vdash c_1 \,;\, c_2 : \mathsf{T}$ then $\mathsf{T} \equiv \mathsf{cmd}$, $\Gamma \vdash c_1 : \mathsf{cmd}$, and $\Gamma \vdash c_2 : \mathsf{cmd}$.

Proof. Follows from induction on the typing derivations.

Lemma 2. *Canonical Forms Lemma*

1. If $\Gamma \vdash v : \mathsf{enc}(\mathsf{E})$ then $v \equiv \mathrm{senc}(u_k, u_m)$ or $\mathrm{junk}(\mathrm{senc}(u_k, u_m))$.
2. If $\Gamma \vdash v : l\,\mathsf{key}$ then $v \equiv n$ or $\mathrm{junk}(n)$.
3. If $\Gamma \vdash v : l\,\mathsf{data}$ then $v \equiv n$, $\mathrm{senc}(u_k, u_m)$, $\mathrm{junk}(n)$ or $\mathrm{junk}(\mathrm{senc}(u_k, u_m))$.

Proof. Follows from inspection of the typing rules and fundamental definitions.

4 Progress and Preservation

The standard way to establish *type safety* for a type system with respect to an operational semantics is to show that the *progress* and *preservation* properties hold. Preservation establishes that the type of a term is not changed by the evaluation rules, while progress demonstrates that well-typed terms will not get ‘stuck.’ Stuck terms represent certain error conditions that may arise during evaluation. In our system, for example, a term becomes stuck whenever further evaluation would result in a security leak. Such leaks are prevented in the operational semantics by checks carried out on the security levels in a number of the evaluation rules.

For the progress and preservation properties to hold, the initial store ϕ must be *well-typed*, and the security levels environment Δ must be *level-consistent* with respect to the typing context Γ. Informally, ϕ is well-typed if every value in ϕ has the type predicted by Γ, while Δ is level-consistent with respect to Γ if every name and location in Δ has the same security level as given to it by Γ.

Definition 1. *A store ϕ is* well-typed *with respect to a typing context Γ, written $\Gamma \vdash \phi$, if $dom(\phi) = dom(\Gamma \,|\, \mathsf{loc})$ and, $\forall a \in dom(\phi)$, $\exists E.\ \Gamma \vdash \phi(a) : E \wedge \Gamma \vdash a : E\,\mathsf{loc}$.*

Definition 2. *A security levels environment Δ is* level-consistent *with respect to a typing context Γ, written $\Gamma \vdash \Delta$, if $dom(\Delta) = dom(\Gamma)$, and*

- $\forall n \in dom(\Gamma \,|\, nam)$, $n : E \in \Gamma \rightarrow n : lvl(E) \in \Delta$
- $\forall a \in dom(\Gamma \,|\, loc)$, $a : E\,loc \in \Gamma \rightarrow a : lvl(E) \in \Delta$

Here, $\mathcal{S} \,|\, \mathsf{nam}$ and $\mathcal{S} \,|\, \mathsf{loc}$ denote the subsets of $\mathcal{S}$ containing only those elements which are names and locations respectively.

Corollary 1. *If $\Gamma \vdash \Delta$, $\Gamma \vdash v : E$ and $\Delta \vdash v : l$, then $l \leq lvl(E)$.*

Proof. By definition, $v \equiv n$, $\mathrm{junk}(n)$, $\mathrm{senc}(u_k, u_m)$ or $\mathrm{junk}(\mathrm{senc}(u_k, u_m))$. If $v \equiv n$ or $\mathrm{junk}(n)$ then, by Lemma 1, $n : \mathsf{E}' \in \Gamma$, where $\mathsf{E}' <: \mathsf{E}$. By $\Gamma \vdash \Delta$, $n : lvl(\mathsf{E}') \in \Delta$ therefore $\Delta \vdash v : lvl(\mathsf{E}')$ and $l = lvl(\mathsf{E}')$. It then follows from $\mathsf{E}' <: \mathsf{E}$ that $lvl(\mathsf{E}') \leq lvl(\mathsf{E})$, so the result holds. If $v \equiv \mathrm{senc}(u_k, u_m)$ or $\mathrm{junk}(\mathrm{senc}(u_k, u_m))$ then $l = \bot$, so the result holds. $\square$

Theorem 1. *Progress*

- ***i)*** *If $\Gamma \vdash t : \mathsf{E}$, then either t is a value, or else for any security levels environment Δ and store ϕ such that $\Gamma \vdash \Delta$ and $\Gamma \vdash \phi$, there exists some t' such that $t \rightarrow_{\Delta\phi} t'$.*
- ***ii)*** *If $\Gamma \vdash t : \mathsf{C}$, then either t is the empty command ε or else, for any security types environment Δ and store ϕ such that $\Gamma \vdash \Delta$ and $\Gamma \vdash \phi$, there exists some t' and ϕ' such that $\langle \phi , t\rangle \rightarrow_{\Delta} \langle \phi', t'\rangle$.*

Proof. By induction on $\Gamma \vdash t : \mathsf{E}$ and $\Gamma \vdash t : \mathsf{C}$: *(selected cases only)*

- Case T-DEREF: $t : \mathsf{E} \equiv\, !a : \mathsf{E} \quad a : \mathsf{E\ loc}$
  The rule E-DEREF applies (it follows from $\Gamma \vdash \phi$ that $a \in \phi$).
- Case T-SENC: $t : \mathsf{E} \equiv \mathrm{senc}(e_k, e_m) : \mathsf{enc}(\mathsf{E}') \quad e_k : l\ \mathsf{key} \quad e_m : \mathsf{E}' \quad lvl(\mathsf{E}') = l$
  By the induction hypothesis, either e_k is a value, or else for any Δ and ϕ such that $\Gamma \vdash \Delta$ and $\Gamma \vdash \phi$, there exists some e'_k such that $e_k \rightarrow_{\Delta\phi} e'_k$. Similarly for e_m. If e_k is not a value then E-SENC1 applies; if e_m is not a value (but e_k is) then E-SENC2 applies; if e_k is a junk value then E-SENC4 applies; if e_m is a junk value (and e_k is a non-junk value) then E-SENC3 applies; if both e_k and e_m are non-junk values then t is a value.
- Case T-SDEC: $t : \mathsf{E} \equiv \mathrm{sdec}(e_k, e_m) : \mathsf{E} \quad e_k : l\ \mathsf{key} \quad e_m : \mathsf{enc}(\mathsf{E}) \quad lvl(\mathsf{E}) = l$
  By the induction hypothesis, either e_k is a value, or else for any Δ and ϕ such that $\Gamma \vdash \Delta$ and $\Gamma \vdash \phi$, there exists some e'_k such that $e_k \rightarrow_{\Delta\phi} e'_k$. Similarly for e_m. If e_k is not a value then E-SDEC1 applies, and if e_m is not a value (but e_k is) then E-SDEC2 applies. If e_k is a value then, by Lemma 2, it must be of the form n or junk(n). The former case is covered by the rules E-SDEC3, E-SDEC4 and E-SDEC5 as described below; in the latter case, E-SDEC6 applies. If e_m is a value then, by Lemma 2, it must be of the form $\mathrm{senc}(u_k, u_m)$ or $\mathrm{junk}(\mathrm{senc}(u_k, u_m))$. In the first case, it follows from Lemma 1 that $\Gamma \vdash u_k : l'\ \mathsf{key}$, $\Gamma \vdash u_m : \mathsf{E}'$ and $lvl(\mathsf{E}') = l'$, where $\mathsf{enc}(\mathsf{E}) :> \mathsf{enc}(\mathsf{E}')$ (therefore $\mathsf{E} :> \mathsf{E}'$). If $e_k = u_k$ then E-SDEC3 applies and if $e_k \neq u_k$ then E-SDEC4 applies. In the second case, where $e_m \equiv \mathrm{junk}(\mathrm{senc}(u_k, u_m))$, the rule E-SDEC5 applies. For rules E-SDEC3 and E-SDEC4, the inequality $l_m \leq l_k$ will be satisfied because it follows from Lemma 1 that $e_k : l\ \mathsf{key} \in \Gamma$, and from $\Gamma \vdash \Delta$ that $e_k : \top \in \Delta$, thus $l_k = \top$.
- Case T-ASSIGN: $t : \mathsf{C} \equiv a := e : \mathsf{cmd} \quad a : \mathsf{E\ loc} \quad e : \mathsf{E}$
  By the induction hypothesis for Part (i), either e is a value, or else for any Δ and ϕ such that $\Gamma \vdash \Delta$ and $\Gamma \vdash \phi$, there exists some e' such that $e \rightarrow_{\Delta\phi} e'$. If e is a value, then E-ASSIGN2 applies, otherwise E-ASSIGN1 applies. In the former case, the inequality will hold because, by Lemma 1, $a : \mathsf{E\ loc} \in \Gamma$, by $\Gamma \vdash \Delta$, $a : lvl(\mathsf{E}) \in \Delta$ therefore $l_a = lvl(\mathsf{E})$, and by Cor. 1, $l_v \leq lvl(\mathsf{E})$.
- Case T-CMDS: $t : \mathsf{C} \equiv c_1 \,;c_2 : \mathsf{cmd} \quad c_1 : \mathsf{cmd} \quad c_2 : \mathsf{cmd}$
  By the induction hypothesis, either c_1 is the empty command ε or else, for any Δ and ϕ such that $\Gamma \vdash \Delta$ and $\Gamma \vdash \phi$, there exists some c'_1 and ϕ' such that $\langle \phi, c_1\rangle \rightarrow_{\Delta} \langle \phi', c'_1\rangle$. If $c_1 \equiv \varepsilon$ then the rule E-CMDS2 applies, otherwise the rule E-CMDS1 applies. □

Theorem 2. *Preservation*

i) *If* $\Gamma \vdash t : E$*,* $\Gamma \vdash \Delta, \phi$ *and there exists some* t' *such that* $t \rightarrow_{\Delta\phi} t'$*, then* $\Gamma \vdash t' : E$*.*

ii) *If* $\Gamma \vdash t : C$*,* $\Gamma \vdash \Delta, \phi$ *and there exists some* t' *and* ϕ' *such that* $\langle \phi, t \rangle \rightarrow_{\Delta} \langle \phi', t' \rangle$*, then* $\Gamma \vdash \phi'$ *and* $\Gamma \vdash t' : C$*.*

Proof. By induction on $\Gamma \vdash t : \mathsf{E}$ and $\Gamma \vdash t : \mathsf{C}$: *(selected cases only)*

- Case T-DEREF: $t : \mathsf{E} \equiv\ !a : \mathsf{E}$ $\quad a : \mathsf{E\ loc}$
  E-DEREF is the only evaluation rule which may apply, therefore $t' \equiv \phi(a)$. By $\Gamma \vdash \phi$, $\exists \mathsf{E}'$ such that $\Gamma \vdash a : \mathsf{E}'\ \mathsf{loc}$ and $\Gamma \vdash \phi(a) : \mathsf{E}'$. It therefore follows that $\mathsf{E}' \equiv \mathsf{E}$ and so the result holds.

- Case T-SENC: $t : \mathsf{E} \equiv \mathrm{senc}(e_k, e_m) : \mathsf{enc(E)}$ $\quad e_k : l\ \mathsf{key}$ $\quad e_m : \mathsf{E}$ $\quad lvl(\mathsf{E}) = l$
  There are four evaluation rules which correspond to the transition $t \rightarrow_{\Delta\phi} t'$: E-SENC1 through E-SENC4. Subcase E-SENC2 has a similar proof to subcase E-SENC1, and subcase E-SENC4 has a similar proof to subcase E-SENC3.

  - Subcase E-SENC1: $e_k \rightarrow_{\Delta\phi} e'_k$ $\quad t' \equiv \mathrm{senc}(e'_k, e_m)$
    The T-SENC rule has a subderivation whose conclusion is $e_k : l\ \mathsf{key}$ and the induction hypothesis gives us $e'_k : l\ \mathsf{key}$. Therefore, in conjunction with $e_m : \mathsf{E}$ and $lvl(\mathsf{E}) = l$, we can apply the rule T-SENC to conclude that $\mathrm{senc}(e'_k, e_m) : \mathsf{enc(E)}$.

  - Subcase E-SENC3: $e_k \equiv u_k$ $\quad e_m \equiv \mathrm{junk}(u_m)$ $\quad t' \equiv \mathrm{junk}(\mathrm{senc}(u_k, u_m))$
    The T-SENC rule has a subderivation whose conclusion is $\mathrm{junk}(u_m) : \mathsf{E}$, and by Lemma 1 we get $u_m : \mathsf{E}$. Therefore, in conjunction with $u_k : l\ \mathsf{key}$ and $lvl(\mathsf{E}) = l$, we can apply T-SENC and T-JUNK to conclude that $\mathrm{junk}(\mathrm{senc}(u_k, u_m)) : \mathsf{enc(E)}$.

- Case T-SDEC: $t : \mathsf{E} \equiv \mathrm{sdec}(e_k, e_m) : \mathsf{E}$ $\quad e_k : l\ \mathsf{key}$ $\quad e_m : \mathsf{enc(E)}$ $\quad lvl(\mathsf{E}) = l$
  There are six evaluation rules which correspond to the transition $t \rightarrow_{\Delta\phi} t'$: E-SDEC1 through E-SDEC6. Subcases E-SDEC1 and E-SDEC2 have similar proofs to subcase E-SENC1 above; subcases E-SDEC5 and E-SDEC6 have a similar proof to subcase E-SENC3 above.

  - Subcase E-SDEC3: $e_k \equiv n$ $\quad e_m \equiv \mathrm{senc}(n, t')$
    The T-SDEC rule has a subderivation whose conclusion is $\mathrm{senc}(n, t') : \mathsf{enc(E)}$. It follows from Lemma 1 that $\Gamma \vdash t' : \mathsf{E}'$ and $\mathsf{enc(E')} <: \mathsf{enc(E)}$. Thus $\mathsf{E}' <: \mathsf{E}$, and we can apply the T-SUB rule to conclude that $\Gamma \vdash t' : \mathsf{E}$.

  - Subcase E-SDEC4: $e_m \equiv \mathrm{senc}(u_k, u_m)$ $\quad e_k \neq u_k$ $\quad t' = \mathrm{junk}(u_m)$
    The T-SDEC rule has a subderivation whose conclusion is $\mathrm{senc}(u_k, u_m) : \mathsf{enc(E)}$. It follows from Lemma 1 that $\Gamma \vdash u_m : \mathsf{E}'$ and $\mathsf{enc(E')} <: \mathsf{enc(E)}$. Thus $\mathsf{E}' <: \mathsf{E}$, and we can apply T-SUB and T-JUNK to conclude that $\Gamma \vdash \mathrm{junk}(u_m) : \mathsf{E}$.

- Case T-ASSIGN: $t : \mathsf{C} \equiv a := e : \mathsf{cmd}$ $\quad a : \mathsf{E\ loc}$ $\quad e : \mathsf{E}$
  Two evaluation rules may correspond to the transition $\langle \phi, t \rangle \rightarrow_{\Delta} \langle \phi', t' \rangle$: E-ASSIGN1 and E-ASSIGN2. The proof for the latter is trivial.

  - Subcase E-ASSIGN1: $\langle \phi, e \rangle \rightarrow_{\Delta} \langle \phi, e' \rangle$ $\quad t' = a := e'$
    The T-ASSIGN rule has a subderivation whose conclusion is $e : \mathsf{E}$. Applying the induction hypothesis to this subderivation gives us $\Gamma \vdash e' : \mathsf{E}$. In conjunction

with the other subderivation $\Gamma \vdash a : \mathsf{E}\ \mathsf{loc}$, we can apply T-ASSIGN to conclude that $\Gamma \vdash a := e' : \mathsf{cmd}$. $\Gamma \vdash \phi'$ follows immediately from the fact that $\phi = \phi'$.

- Case T-CMDS: $\quad t : \mathsf{C} \equiv c_1 \,;\, c_2 : \mathsf{cmd} \qquad c_1 : \mathsf{cmd} \qquad c_2 : \mathsf{cmd}$
  Two evaluation rules may correspond to the transition $\langle \phi, t \rangle \rightarrow_{\Delta} \langle \phi', t' \rangle$: E-CMDS1 and E-CMDS2. The proof for the latter is trivial.
  - Subcase E-CMDS1: $\quad \langle \phi, c_1 \rangle \rightarrow_{\Delta} \langle \phi', c_1' \rangle \qquad t' = c_1' \,;\, c_2$
    The T-CMDS rule has a subderivation whose conclusion is $c_1 : \mathsf{cmd}$ and the induction hypothesis gives us $\Gamma \vdash \phi'$ and $\Gamma \vdash c_1' : \mathsf{cmd}$. Using the latter of these, in conjunction with the other subderivation $\Gamma \vdash c_2 : \mathsf{cmd}$, we can apply the rule T-CMDS to conclude that $\Gamma \vdash c_1' \,;\, c_2 : \mathsf{cmd}$. □

The following lemma states that the type of an expression is preserved under evaluation with respect to a well-typed store, independent of the actual values contained in the locations of that store, and is required to prove our non-interference result.

Lemma 3. *If* $(\Gamma, a : \mathsf{E}\ \mathsf{loc}) \vdash e : \mathsf{E}'$, $\Gamma \vdash v : \mathsf{E}$, $(\Gamma, a : \mathsf{E}\ \mathsf{loc}) \vdash \Delta, \phi$ *and* $e \rightarrow^{*}_{\Delta\phi'} v'$, *where* $\phi' \equiv \phi[a \mapsto v]$, *then* $\Gamma \vdash v' : \mathsf{E}'$.

Proof. By induction on $(\Gamma, a : \mathsf{E}\ \mathsf{loc}) \vdash e : \mathsf{E}'$: *(selected cases only)*

- Case T-DEREF: $\quad e : \mathsf{E}' \equiv\ !a' : \mathsf{E}' \qquad a' : \mathsf{E}'\ \mathsf{loc}$
  $!a' \rightarrow_{\Delta\phi'} \phi'(a')$. By Lemma 1, $a' : \mathsf{E}'\ \mathsf{loc} \in \Gamma$. If $a = a'$ then $v' = v$ and $\mathsf{E}' \equiv \mathsf{E}$, thus the result holds. If $a \neq a'$ then $v' = \phi(a')$ and the result follows from $(\Gamma, a : \mathsf{E}\ \mathsf{loc}) \vdash \phi$.
- Case T-SENC: $\quad e : \mathsf{E}' \equiv \mathrm{senc}(e_k, e_m) : \mathsf{enc}(\mathsf{E}'') \qquad e_k : l\ \mathsf{key} \qquad e_m : \mathsf{E}'' \qquad lvl(\mathsf{E}'') = l$
  $\mathrm{senc}(e_k, e_m) \rightarrow^{*}_{\Delta\phi'} \mathrm{senc}(v_k, v_m) \rightarrow^{*}_{\Delta\phi'} v'$, where $e_k \rightarrow^{*}_{\Delta\phi'} v_k$ and $e_m \rightarrow^{*}_{\Delta\phi'} v_m$. By the induction hypothesis, $\Gamma \vdash v_k : l\ \mathsf{key}$ and $\Gamma \vdash v_m : \mathsf{E}''$. If v_k and v_m are both non-junk values, then $v' \equiv \mathrm{senc}(v_k, v_m)$ and the the result follows from T-SENC. Otherwise, $v_k \equiv \mathrm{junk}(u_k)$ and/or $v_m \equiv \mathrm{junk}(u_m)$, therefore $v' \equiv \mathrm{junk}(\mathrm{senc}(u_k, u_m))$ and the result follows from T-JUNK and T-SENC.
- Case T-SDEC: $\quad e : \mathsf{E}' \equiv \mathrm{sdec}(e_k, e_m) : \mathsf{E}' \qquad e_k : l\ \mathsf{key} \qquad e_m : \mathsf{enc}(\mathsf{E}')$
  $\mathrm{sdec}(e_k, e_m) \rightarrow^{*}_{\Delta\phi'} \mathrm{sdec}(v_k, v_m) \rightarrow^{*}_{\Delta\phi'} v'$ where $e_k \rightarrow^{*}_{\Delta\phi'} v_k$ and $e_m \rightarrow^{*}_{\Delta\phi'} v_m$. By the induction hypothesis, $\Gamma \vdash v_k : l\ \mathsf{key}$ and $\Gamma \vdash v_m : \mathsf{enc}(\mathsf{E}')$, and by Lemma 2, v_k is of the form n or $\mathrm{junk}(n)$, and v_m is of the form $\mathrm{senc}(u_k, u_m)$ or $\mathrm{junk}(\mathrm{senc}(u_k, u_m))$. In both cases for v_m, it follows from Lemma 1 that $\Gamma \vdash u_m : \mathsf{E}''$, where $\mathsf{E}'' <: \mathsf{E}'$. By inspection of the evaluation rules, v' will be of the form u_m or $\mathrm{junk}(u_m)$. In the first case, we can apply T-SUB to $\Gamma \vdash u_m : \mathsf{E}''$ and $\mathsf{E}'' <: \mathsf{E}'$ to conclude that $\Gamma \vdash u_m : \mathsf{E}'$; in the second case, the result follows from T-SUB and T-JUNK. □

5 Indistinguishability

Our type system is intended for analysing systems with ciphers that are *repetition concealing* and *which-key concealing* — also known as type-1 ciphers ([10], Sec. 4.2). Repetition concealing means that it is not possible to say whether two messages encrypted under the same key are equal. Which-key concealing means that it is not possible to say whether two keys used to encrypt the same message are equal. Both of these properties are possessed by standard block ciphers, such as DES and AES, when used in CBC or CTR mode ([10], Sec. 4.4). However, these definitions assume that the adversary is unable to correctly decrypt the ciphertexts. This is not strictly the case with security APIs: the API functions can be used to decrypt ciphertexts whose contents are public, whilst keeping the actual values of the keys secret. As a result, we have to capture the ability of the adversary to distinguish between ciphertexts which contain public data, under certain circumstances.

We use the notation $\Gamma \vdash v_1 \sim_l v_2 : \mathsf{E}$ to denote that the values v_1 and v_2 both have type E and are indistinguishable at the security level l, and the notation $\Gamma \vdash \phi \sim_l \phi'$ to denote that the stores ϕ and ϕ' are indistinguishable at the security level l. In both cases, l denotes the maximum security level associated with the locations that an observer can read directly.

Definition 3. *We define the* indistinguishability of two values, v_1 *and* v_2, *with respect to a typing environment* Γ *and observation level* l, *denoted* $\Gamma \vdash v_1 \sim_l v_2 : E$, *as the least symmetric relation closed under the following rules, where* $\Gamma \vdash v_1, v_2 : E$*:*

- $\Gamma \vdash n_1 \sim_l n_2 : l'\ \mathsf{data}$ *iff* $(l \geq l') \rightarrow (n_1 = n_2)$
- $\Gamma \vdash n_1 \sim_l n_2 : l'\ \mathsf{key}$ *iff* $(l \geq l') \rightarrow (n_1 = n_2)$
- $\Gamma \vdash senc(u_k, u_m) \sim_l senc(u'_k, u'_m) : enc(E)$ *iff* $(\Gamma \vdash u_m \sim_l u'_m : E) \wedge$
  $(\Gamma \vdash junk(u_m) \sim_l u'_m : E \vee \Gamma \vdash u_k \sim_l u'_k : lvl(E)\ \mathsf{key})$
- $\Gamma \vdash junk(u) \sim_l junk(u') : E$
- $\Gamma \vdash junk(n) \sim_l n' : l'\ \mathsf{data}$ *iff* $(l < l')$
- $\Gamma \vdash junk(n) \sim_l n' : l'\ \mathsf{key}$ *iff* $(l < l')$
- $\Gamma \vdash junk(senc(u_k, u_m)) \sim_l senc(u'_k, u'_m) : enc(E)$ *iff* $\Gamma \vdash junk(u_m) \sim_l u'_m : E$

If a value has a type which permits it to be observed by the adversary, we must assume that this will eventually occur. It then follows that unencrypted data items which can be observed must be equal for them to be considered indistinguishable. Keys will be distinguishable if the output from their use is distinguishable. That is, by encrypting a known value with each key, decrypting each ciphertext with both keys, then comparing the final results to the original input: if any of the outputs are distinguishable from the input, then the two keys cannot be the same, and are thus distinguishable.

Ciphertexts are indistinguishable if their messages are indistinguishable, and the keys must also be indistinguishable if the observer could otherwise determine when one of the ciphertexts has been incorrectly decrypted. That is, if the keys have a type which allows them to encrypt observable data, then we must assume that the adversary is able to correctly decrypt each ciphertext, and can thus determine whether or not the required keys are the same whenever he can predict the correct output. It follows from the definition that keys which operate on non-observable data are indistinguishable.

Two junk values are indistinguishable, since they are both essentially just random bit-strings. For this reason also, junk names are distinguishable from observable non-junk names. Junk ciphertext is indistinguishable from non-junk ciphertext if the results of decrypting each one cannot be distinguished.

Definition 4. *We define the* indistinguishability of two stores, ϕ_1 *and* ϕ_2*, with respect to a typing environment* Γ *and observation level* l*, denoted* $\Gamma \vdash \phi_1 \sim_l \phi_2$*, as the least relation closed under the following rules:*

- $\Gamma \vdash \varepsilon \sim_l \varepsilon$
- $\Gamma \vdash (\phi, a \mapsto v) \sim_l (\phi', a \mapsto v')$ *iff* $\Gamma \vdash \phi \sim_l \phi'$, $\Gamma \vdash v, v' : E$ *and* $\Gamma \vdash v \sim_l v' : E$

This definition states that two stores are indistinguishable if their domains are equal, and the values stored in equivalent locations are indistinguishable.

6 Non-Interference

Informally, non-interference states that changes to non-observable inputs should have no effect on observable outputs. For expressions, this means that given two indistinguishable stores (which differ in the contents of at least one non-observable location), the final values obtained by fully evaluating the same expression with respect to those stores should be indistinguishable. For command sequences, this means that given two indistinguishable stores (which again differ in the contents of at least one non-observable location), the stores which result from fully evaluating the same command sequence with respect to those stores should also be indistinguishable.

Theorem 3. *Non-Interference*

i) *If* $(\Gamma, a : \mathsf{E\ loc}) \vdash e : E'$, $\Gamma \vdash v_1, v_2 : E$ *and* $\Gamma \vdash \Delta, \phi_1, \phi_2$*, such that* $\Gamma \vdash v_1 \sim_l v_2 : E$ *and* $\Gamma \vdash \phi_1 \sim_l \phi_2$*, then it follows from* $e \rightarrow^*_{\Delta\phi_1'} v_1'$ *and* $e \rightarrow^*_{\Delta\phi_2'} v_2'$ *that* $\Gamma \vdash v_1' \sim_l v_2' : E'$*, where* $\phi_i' \equiv \phi_i[a \mapsto v_i]$.

ii) *If* $(\Gamma, a : \mathsf{E\ loc}) \vdash c : C$, $\Gamma \vdash v_1, v_2 : E$ *and* $\Gamma \vdash \Delta, \phi_1, \phi_2$*, such that* $\Gamma \vdash v_1 \sim_l v_2 : E$ *and* $\Gamma \vdash \phi_1 \sim_l \phi_2$*, then it follows from* $\langle c, \phi_1' \rangle \rightarrow^*_{\Delta} \langle \varepsilon, \phi_1'' \rangle$ *and* $\langle c, \phi_2' \rangle \rightarrow^*_{\Delta} \langle \varepsilon, \phi_2'' \rangle$ *that* $\Gamma \vdash \phi_1'' \sim_l \phi_2''$*, where* $\phi_i' \equiv \phi_i[a \mapsto v_i]$.

Proof. By induction on $(\Gamma, a : \mathsf{E\ loc}) \vdash e : \mathsf{E}'$ and $(\Gamma, a : \mathsf{E\ loc}) \vdash c : \mathsf{C}$: *(selected cases only)*

- Case T-DEREF: $e : \mathsf{E}' \equiv !a' : \mathsf{E}' \quad a' : \mathsf{E}'\ \mathsf{loc}$
  $!a' \rightarrow_{\Delta\phi_i'} \phi_i'(a')$. If $a' = a$, then $v_i' = v_i$ and $\mathsf{E}' \equiv \mathsf{E}$, thus the result follows from $\Gamma \vdash v_1 \sim_l v_2 : \mathsf{E}$. If $a' \neq a$, the result follows from $\Gamma \vdash \phi_1 \sim_l \phi_2$.

- Case T-SENC: $e : \mathsf{E}' \equiv \mathrm{senc}(e_k, e_m) : \mathsf{enc}(\mathsf{E}'') \quad e_k : l'\ \mathsf{key} \quad e_m : \mathsf{E}'' \quad lvl(\mathsf{E}'') = l'$
  $\mathrm{senc}(e_k, e_m) \rightarrow^*_{\Delta\phi_1'} \mathrm{senc}(v_k, v_m) \rightarrow^*_{\Delta\phi_1'} v_1'$ where $e_k \rightarrow^*_{\Delta\phi_1'} v_k$ and $e_m \rightarrow^*_{\Delta\phi_1'} v_m$. $\mathrm{senc}(e_k, e_m) \rightarrow^*_{\Delta\phi_2'} \mathrm{senc}(v_k', v_m') \rightarrow^*_{\Delta\phi_2'} v_2'$ where $e_k \rightarrow^*_{\Delta\phi_2'} v_k'$ and $e_m \rightarrow^*_{\Delta\phi_2'} v_m'$. It follows from Lemma 3 that $\Gamma \vdash v_k, v_k' : l'\ \mathsf{key}$ and $\Gamma \vdash v_m, v_m' : \mathsf{E}''$, by Lemma 2, $v_k \equiv n$ or junk(n), and $v_k' \equiv n'$ or junk(n'), and by definition, $v_m \equiv u_m$ or junk(u_m), and $v_m' \equiv u_m'$ or junk(u_m'). If $v_k \equiv n$ and $v_m \equiv u_m$ then $v_1' \equiv \mathrm{senc}(n, u_m)$ [A]; if $v_k \equiv \mathrm{junk}(n)$ and $v_m \equiv u_m$ then, by E-SENC4, $v_1' \equiv \mathrm{junk}(\mathrm{senc}(n, u_m))$ [B]; if $v_k \equiv n$ and $v_m \equiv \mathrm{junk}(u_m)$ then, by E-SENC3, $v_1' \equiv \mathrm{junk}(\mathrm{senc}(n, u_m))$ [C], and if $v_k \equiv \mathrm{junk}(n)$ and $v_m \equiv \mathrm{junk}(u_m)$ then, by E-SENC4, E-JUNK1, E-SENC3 and E-JUNK2, $v_1' \equiv \mathrm{junk}(\mathrm{senc}(n, u_m))$ [D]. The equivalent outcomes for v_2' are denoted by [E] through [H]. There are 16 cases for $\Gamma \vdash v_1' \sim_l v_2' : \mathsf{E}'$ which we need to consider (resulting from the cross product of [A,B,C,D] and [E,F,G,H]):

  - Subcase [A]×[E]: $\Gamma \vdash \mathrm{senc}(n, u_m) \sim_l \mathrm{senc}(n', u'_m) : \mathsf{enc}(\mathsf{E}'')$
    By the induction hypothesis, $\Gamma \vdash n \sim_l n' : l'\ \mathsf{key}$ and $\Gamma \vdash u_m \sim_l u'_m : \mathsf{E}''$, and since $lvl(\mathsf{E}'') = l'$, the result follows immediately from Def. 3.
  - Subcase [A]×[F]: $\Gamma \vdash \mathrm{senc}(n, u_m) \sim_l \mathrm{junk}(\mathrm{senc}(n', u'_m)) : \mathsf{enc}(\mathsf{E}'')$
    By the induction hypothesis, $\Gamma \vdash n \sim_l \mathrm{junk}(n') : l'\ \mathsf{key}$ thus, by Def. 3, $l < l'$. $\Gamma \vdash \mathrm{senc}(n, u_m) \sim_l \mathrm{junk}(\mathrm{senc}(n', u'_m)) : \mathsf{enc}(\mathsf{E}'')$ iff $\Gamma \vdash u_m \sim_l \mathrm{junk}(u'_m) : \mathsf{E}''$ and this holds when $l < lvl(\mathsf{E}'')$. The result then follows from $lvl(\mathsf{E}'') = l'$ and $l < l'$.
  - Subcases [A]×[G,H]: $\Gamma \vdash \mathrm{senc}(n, u_m) \sim_l \mathrm{junk}(\mathrm{senc}(u'_k, u'_m)) : \mathsf{enc}(\mathsf{E}'')$
    By the induction hypothesis, $\Gamma \vdash u_m \sim_l \mathrm{junk}(u'_m) : \mathsf{E}''$, thus the result follows immediately from Def. 3.
  - Subcases [B,C,D]×[F,G,H]: $\Gamma \vdash \mathrm{junk}(\mathrm{senc}(n, u_m)) \sim_l \mathrm{junk}(\mathrm{senc}(n', u'_m)) : \mathsf{enc}(\mathsf{E}'')$
    The result follows immediately from Def. 3.

  Subcase [B]×[E] is similar to subcase [A]×[F] and subcases [C,D]×[E] are similar to subcases [A]×[G,H].

- Case T-SDec: $e : \mathsf{E}' \equiv \mathrm{sdec}(e_k, e_m) : \mathsf{E}' \quad e_k : l'\ \mathsf{key} \quad e_m : \mathsf{enc}(\mathsf{E}') \quad lvl(\mathsf{E}') = l'$
  $\mathrm{sdec}(e_k, e_m) \rightarrow^{*}_{\Delta\phi'_1} \mathrm{sdec}(v_k, v_m) \rightarrow^{*}_{\Delta\phi'_1} v'_1$, where $e_k \rightarrow^{*}_{\Delta\phi'_1} v_k$ and $e_m \rightarrow^{*}_{\Delta\phi'_1} v_m$. $\mathrm{sdec}(e_k, e_m) \rightarrow^{*}_{\Delta\phi'_2} \mathrm{sdec}(v'_k, v'_m) \rightarrow^{*}_{\Delta\phi'_2} v'_2$, where $e_k \rightarrow^{*}_{\Delta\phi'_2} v'_k$ and $e_m \rightarrow^{*}_{\Delta\phi'_2} v'_m$. It follows from Lemma 3 that $\Gamma \vdash v_k, v'_k : l'\ \mathsf{key}$ and $\Gamma \vdash v_m, v'_m : \mathsf{enc}(\mathsf{E}')$, and by Lemma 2, $v_k \equiv n$ or $\mathrm{junk}(n)$, $v'_k \equiv n'$ or $\mathrm{junk}(n')$, $v_m \equiv \mathrm{senc}(u_a, u_b)$ or $\mathrm{junk}(\mathrm{senc}(u_a, u_b))$, and $v'_m \equiv \mathrm{senc}(u'_a, u'_b)$ or $\mathrm{junk}(\mathrm{senc}(u'_a, u'_b))$. If $v_k \equiv n$ and $v_m \equiv \mathrm{senc}(n, u_b)$ then, by E-SDec3, $v'_1 = u_b$ [A]; if $v_k \equiv n$ and $v_m \equiv \mathrm{senc}(u_a, u_b)$ where $u_a \neq n$ then, by E-SDec4, $v'_1 = \mathrm{junk}(u_b)$ [B]; if $v_k \equiv \mathrm{junk}(n)$ and $v_m \equiv \mathrm{senc}(u_a, u_b)$ then, by E-SDec5, $v'_1 = \mathrm{junk}(u_b)$ [C]; if $v_k \equiv n$ and $v_m \equiv \mathrm{junk}(\mathrm{senc}(u_a, u_b))$ then, by E-SDec6, $v'_1 = \mathrm{junk}(u_b)$ [D], and if $v_k \equiv \mathrm{junk}(n)$ and $v_m \equiv \mathrm{junk}(\mathrm{senc}(u_a, u_b))$ then, by E-SDec6, E-Junk1, E-SDec5 and E-Junk2, $v'_1 = \mathrm{junk}(u_b)$ [E]. The equivalent outcomes for v'_2 are denoted by [F] through [J]. There are 25 subcases for $\Gamma \vdash v'_1 \sim_l v'_2 : \mathsf{E}'$ which we need to consider (resulting from the cross product of [A,B,C,D,E] and [F,G,H,I,J]):
  - Subcase [A]×[F]: $\Gamma \vdash u_b \sim_l u'_b : \mathsf{E}'$
    By the induction hypothesis, $\Gamma \vdash \mathrm{senc}(n, u_b) \sim_l \mathrm{senc}(n', u'_b) : \mathsf{enc}(\mathsf{E}')$, therefore it follows from Def. 3 that $\Gamma \vdash u_b \sim_l u'_b : \mathsf{E}'$.
  - Subcase [A]×[G]: $\Gamma \vdash u_b \sim_l \mathrm{junk}(u'_b) : \mathsf{E}'$
    By the induction hypothesis, we have $\Gamma \vdash \mathrm{senc}(n, u_b) \sim_l \mathrm{senc}(u'_a, u'_b) : \mathsf{enc}(\mathsf{E}')$ and $\Gamma \vdash n \sim_l n' : l'\ \mathsf{key}$. By Def. 3, it follows from the second of these that $l < l'$ or $n = n'$. From the first one, it follows that $\Gamma \vdash u_b \sim_l u'_b : \mathsf{E}'$ as well as either $\Gamma \vdash n \sim_l u'_a : lvl(\mathsf{E}')\ \mathsf{key}$ or $\Gamma \vdash u_b \sim_l \mathrm{junk}(u'_b) : \mathsf{E}'$. In the latter case, the result is immediate. In the former case, it follows from Def. 3 that $l < lvl(\mathsf{E}')$ or $n = u'_a$. However, since $lvl(\mathsf{E}') = l'$, it must be the case that $l < lvl(\mathsf{E}')$, otherwise we would have $n = n' = u'_a$ which is prevented by the requirement for [G] which states that $n' \neq u'_a$. The result then follows from Def. 3.
  - Subcases [A]×[H,J]: $\Gamma \vdash u_b \sim_l \mathrm{junk}(u'_b) : \mathsf{E}'$
    By the induction hypothesis, $\Gamma \vdash n \sim_l \mathrm{junk}(n') : l'\ \mathsf{key}$, therefore it follows from Def. 3 that $l < l'$. Since $l' = lvl(\mathsf{E}')$, the result follows from Def. 3.

  - Subcase [A]×[I]: $\Gamma \vdash u_b \sim_l \mathrm{junk}(u'_b) : \mathsf{E}'$
    By the induction hypothesis, $\Gamma \vdash \mathrm{senc}(n, u_b) \sim_l \mathrm{junk}(\mathrm{senc}(v'_a, u'_b)) : \mathsf{enc}(\mathsf{E}')$ and so it follows from Def. 3 that $\Gamma \vdash u_b \sim_l \mathrm{junk}(u'_b) : \mathsf{E}'$.
  - Subcases [B,C,D,E]×[G,H,I,J]: $\Gamma \vdash \mathrm{junk}(u_b) \sim_l \mathrm{junk}(u'_b) : \mathsf{E}'$
    The result follows immediately from Def. 3.

  Subcase [B]×[F] is similar to subcase [A]×[G]; subcases [C,E]×[F] are similar to subcases [A]×[H,J], and subcase [D]×[F] is similar to subcase [A]×[I].

- Case T-ASSIGN: $c : \mathsf{C} \equiv a' := e : \mathsf{cmd} \qquad a' : \mathsf{E\,loc} \qquad e : \mathsf{E}$
  $\langle a' := e, \phi'_1 \rangle \xrightarrow[\Delta]{}^{*} \langle \varepsilon, \phi'_1[a' \mapsto v] \rangle$ where $e \xrightarrow[\Delta\phi'_1]{}^{*} v$, and $\langle a' := e, \phi'_2 \rangle \xrightarrow[\Delta]{}^{*} \langle \varepsilon, \phi'_2[a' \mapsto v'] \rangle$ where $e \xrightarrow[\Delta\phi'_2]{}^{*} v'$. By the induction hypothesis for part (i), $\Gamma \vdash v \sim_l v' : \mathsf{E}$ and thus, in conjunction with $\Gamma \vdash \phi'_1 \sim_l \phi'_2$ and $\Gamma \vdash v_1 \sim_l v_2 : \Gamma$, the result follows from Def. 4.

- Case T-CMDS: $c : \mathsf{C} \equiv c_1 ; c_2 : \mathsf{cmd} \qquad c_1 : \mathsf{cmd} \qquad c_2 : \mathsf{cmd}$
  $\langle c_1 ; c_2, \phi'_1 \rangle \xrightarrow[\Delta]{}^{*} \langle \varepsilon ; c_2, \phi'''_1 \rangle \xrightarrow[\Delta]{} \langle c_2, \phi'''_1 \rangle \xrightarrow[\Delta]{}^{*} \langle \varepsilon, \phi''_1 \rangle$ where $\langle c_1, \phi'_1 \rangle \xrightarrow[\Delta]{}^{*} \langle \varepsilon, \phi'''_1 \rangle$ and $\langle c_1 ; c_2, \phi'_2 \rangle \xrightarrow[\Delta]{}^{*} \langle \varepsilon ; c_2, \phi'''_2 \rangle \xrightarrow[\Delta]{} \langle c_2, \phi'''_2 \rangle \xrightarrow[\Delta]{}^{*} \langle \varepsilon, \phi''_2 \rangle$ where $\langle c_1, \phi'_2 \rangle \xrightarrow[\Delta]{}^{*} \langle \varepsilon, \phi'''_2 \rangle$. The result then follows by two applications of the induction hypothesis. □

Theorem 3 guarantees that well-typed expressions and command sequences are non-interferent. As an example, consider the following presentation of an API function for encrypting low security data stored in *msg_loc* with a key that is itself encrypted (by a master key, $\mathsf{k_m}$) and stored in a low security location, *ekey_loc*:[4]

$$\Gamma = \left\{ \begin{array}{c} \mathsf{k_m} : \top\,\mathsf{key},\ \mathit{ekey\_loc} : \mathsf{enc}(\bot\,\mathsf{key})\,\mathsf{loc}, \\ \mathit{key\_loc} : \bot\,\mathsf{key\,loc},\ \mathit{msg\_loc} : \bot\,\mathsf{data\,loc},\ \mathit{res\_loc} : \mathsf{enc}(\bot\,\mathsf{data})\,\mathsf{loc} \end{array} \right\}$$

$$\mathit{key\_loc} := \mathrm{sdec}(\mathsf{k_m}, !\mathit{ekey\_loc})\ ;\ \mathit{res\_loc} := \mathrm{senc}(!\mathit{key\_loc}, !\mathit{msg\_loc}) : \mathsf{cmd}$$

The non-interference theorem tells us that the above well-typed command sequence will not leak any information about the values of $\mathsf{k_m}$ and $!\mathit{key\_loc}$ into the low security locations.

7 Example: Wrap/Decrypt Attack

The wrap/decrypt attack ([11], Sec. 2.3) is one of the most basic attacks which a key management API can be susceptible to. In short, a sensitive key is altered in such a way as to be able to wrap (encrypt) other sensitive keys and also decrypt public data. This typically involves altering the key's 'type' so that it is accepted by each of the two required API functions. Alternatively, two copies of the key can be obtained such that each copy has one of the two necessary types. Both of these requirements can be quite straightforward to achieve (e.g., as discussed in [7]). The outcome is that a sensitive key can be discovered by first wrapping it, then decrypting the result:

$$\begin{array}{ll} x := \mathrm{senc}(\mathsf{k_1}, \mathsf{k_2}) & \ldots\text{'wrap' } \mathsf{k_2} \text{ with } \mathsf{k_1} \\ y := \mathrm{sdec}(\mathsf{k_1}, !x) & \ldots\text{recover } \mathsf{k_2} \end{array}$$

[4] Recall that we treat all keys as high security values, and the security level associated with a key's type denotes the level of data that it may encrypt.

$$\dfrac{\dfrac{\begin{array}{c}[\Rightarrow \mathsf{E}\equiv\mathsf{enc}(\top\,\mathsf{key})]\\ x:\mathsf{enc}(\top\,\mathsf{key})\,\mathsf{loc}\in\Gamma\end{array}}{\Gamma\vdash x:\mathsf{E}\,\mathsf{loc}} \qquad \dfrac{\dfrac{\dfrac{\begin{array}{c}[\Rightarrow l=\top]\\ \mathsf{k}_1:\top\,\mathsf{key}\in\Gamma\end{array}}{\Gamma\vdash \mathsf{k}_1:l\,\mathsf{key}} \quad \dfrac{\begin{array}{c}[\Rightarrow \mathsf{E}'\equiv\top\,\mathsf{key}]\\ \mathsf{k}_2:\top\,\mathsf{key}\in\Gamma\end{array}}{\Gamma\vdash \mathsf{k}_2:\mathsf{E}'} \quad \begin{array}{c}[\,\text{\textsc{Holds}}\,]\\ lvl(\mathsf{E}')=l\end{array}}{\Gamma\vdash\mathsf{senc}(\mathsf{k}_1,\mathsf{k}_2):\mathsf{enc}(\mathsf{E}')} \quad \begin{array}{c}[\,\text{\textsc{Holds}}\,]\\ \mathsf{enc}(\mathsf{E}')<:\mathsf{E}\end{array}}{\Gamma\vdash\mathsf{senc}(\mathsf{k}_1,\mathsf{k}_2):\mathsf{E}}}{\Gamma\vdash x:=\mathsf{senc}(\mathsf{k}_1,\mathsf{k}_2):\mathsf{cmd}}$$

Fig. 5. Successful typing derivation for the wrap command

$$\dfrac{\dfrac{\begin{array}{c}[\Rightarrow \mathsf{E}\equiv\bot\,\mathsf{data}]\\ y:\bot\,\mathsf{data}\,\mathsf{loc}\in\Gamma\end{array}}{\Gamma\vdash y:\mathsf{E}\,\mathsf{loc}} \qquad \dfrac{\dfrac{\dfrac{\begin{array}{c}[\Rightarrow l=\top]\\ \mathsf{k}_1:\top\,\mathsf{key}\in\Gamma\end{array}}{\Gamma\vdash \mathsf{k}_1:l\,\mathsf{key}} \quad \dfrac{\dfrac{\begin{array}{c}[\Rightarrow \mathsf{E}'\equiv\top\,\mathsf{key}]\\ x:\mathsf{enc}(\top\,\mathsf{key})\,\mathsf{loc}\in\Gamma\end{array}}{\Gamma\vdash x:\mathsf{enc}(\top\,\mathsf{key})\,\mathsf{loc}}}{\Gamma\vdash\,!x:\mathsf{enc}(\mathsf{E}')} \quad \begin{array}{c}[\,\text{\textsc{Holds}}\,]\\ lvl(\mathsf{E}')=l\end{array}}{\Gamma\vdash\mathsf{sdec}(\mathsf{k}_1,!x):\mathsf{E}'} \quad \begin{array}{c}\left[\begin{array}{c}\text{\textsc{Does}}\\ \text{\textsc{Not}}\\ \text{\textsc{Hold}}\end{array}\right]\\ \mathsf{E}'<:\mathsf{E}\end{array}}{\Gamma\vdash\mathsf{sdec}(\mathsf{k}_1,!x):\mathsf{E}}}{\Gamma\vdash y:=\mathsf{sdec}(\mathsf{k}_1,!x):\mathsf{cmd}}$$

Fig. 6. Failed typing derivation for the decrypt command

Our type system can be applied to these commands as follows:

$$\Gamma=\left\{\begin{array}{c}\mathsf{k}_1:\top\,\mathsf{key},\ \mathsf{k}_2:\top\,\mathsf{key},\\ x:\mathsf{enc}(\top\,\mathsf{key})\,\mathsf{loc},\ y:\bot\,\mathsf{data}\,\mathsf{loc}\end{array}\right\} \qquad \begin{array}{l}x:=\mathsf{senc}(\mathsf{k}_1,\mathsf{k}_2):\mathsf{cmd}\\ y:=\mathsf{sdec}(\mathsf{k}_1,!x):\mathsf{cmd}\end{array}$$

Figure 5 shows the typing derivation for the wrap command, and Fig. 6 shows the typing derivation for the decrypt command (unnecessary instances of the T-SUB rule have been omitted in both cases).

The first command type-checks, since $lvl(\mathsf{E}')=l$ and $\mathsf{enc}(\mathsf{E}')<:\mathsf{E}$ both hold, but the second command does not, since $\mathsf{E}'<:\mathsf{E}$ does not hold. The flaw is that a sensitive piece of data is written to a public location — the failed subtype condition indicates that the security level of the data is greater than that of the location. Note that using the definition $x:\mathsf{enc}(\bot\,\mathsf{data})\,\mathsf{loc}$ instead of $x:\mathsf{enc}(\top\,\mathsf{key})\,\mathsf{loc}$ in the above example makes the second command type-check, but it prevents the first command from type-checking, since $\mathsf{enc}(\mathsf{E}')<:\mathsf{E}$ no longer holds.

The wrap/decrypt attack is one of a number of attacks which initially require the type of a key to be altered, therefore our type system should be able to identify when an API command may allow this to occur. One such command is 'unwrap,' which takes an existing key and ciphertext corresponding to a second key encrypted under the first one, and then decrypts the ciphertext before storing the result. Figure 7 shows that our type system is indeed able to identify that the following instantiation of that command is insecure:

$$\Gamma=\left\{\begin{array}{c}key:\top\,\mathsf{key},\\ wkey:\mathsf{enc}(l\,\mathsf{key}),\ res:l'\,\mathsf{key}\,\mathsf{loc}\end{array}\right\} \qquad res:=\mathsf{sdec}(key,wkey):\mathsf{cmd}$$

$$\dfrac{\dfrac{\overset{[\Rightarrow \mathsf{E}' \equiv l'\,\mathsf{key}]}{res : l'\,\mathsf{key}\,\mathsf{loc} \in \Gamma}}{\Gamma \vdash res : \mathsf{E}'\,\mathsf{loc}} \qquad \dfrac{\dfrac{\dfrac{\overset{[\Rightarrow l'' = \top]}{key : \top\,\mathsf{key} \in \Gamma}}{\Gamma \vdash key : l''\,\mathsf{key}} \quad \dfrac{\overset{[\Rightarrow \mathsf{E} \equiv l\,\mathsf{key}]}{wkey : \mathsf{enc}(l\,\mathsf{key}) \in \Gamma}}{\Gamma \vdash wkey : \mathsf{enc}(\mathsf{E})} \quad \overset{[\,\textsc{Holds}\,]}{lvl(\mathsf{E}) = l''}}{\Gamma \vdash \mathsf{sdec}(key, wkey) : \mathsf{E}} \quad \overset{\left[\begin{array}{c}\textsc{May}\\ \textsc{Not}\\ \textsc{Hold}\end{array}\right]}{\mathsf{E} <: \mathsf{E}'}}{\Gamma \vdash \mathsf{sdec}(key, wkey) : \mathsf{E}'}}{\Gamma \vdash res := \mathsf{sdec}(key, wkey) : \mathsf{cmd}}$$

Fig. 7. The unwrap command is only secure when l key $<:$ l' key (i.e., when $l = l'$)

Since the security level associated with the type of a key restricts what that key can be used to encrypt and decrypt, and the instantiation of the 'unwrap' command given above allows this level to be changed (i.e., when $l \neq l'$), then it is clearly insecure. This particular flaw can be prevented in practice by including usage information for the key within the ciphertext, thereby making it possible to carry out a check which is equivalent to ensuring that l and l' are equal. However, it is then necessary to ensure that no API command allows this usage information to be modified unintentionally.

8 Related Work

Vaughan and Zdancewic [12] give a security typed language in which valid programs are guaranteed to be non-interfering; a result which is achieved via a combination of static and dynamic checks. However, they require that encrypted messages adhere to a strict format which prevents their system from being used to analyse many existing security APIs.

Laud [13] presents a weakened variant of non-interference termed 'computational independence,' using static analysis to track dependencies between variables. Security is guaranteed when the public outputs are computationally independent from all of the sensitive inputs. Encryption is probabilistic and assumed to be secure with respect to a polynomially-bounded adversary. Key cycles are permitted, as the rules will identify the resulting cyclic dependencies.

Focardi and Centenaro [14] give a type system for enforcing non-interference in multi-threaded distributed programs which share common memory locations. They use *confounders* (unique values associated with each new ciphertext) as an abstraction of probabilistic encryption, and give a definition of equivalence for low security values based on the notion of *patterns* [10]. If the confounder is uniquely determined by the message, then their definition of indistinguishability for ciphertexts is equivalent to the one given in this paper. Their definition for memories is stronger than our one since we do not distinguish between copies of the same ciphertext and different ciphertexts created from the same key and message (doing so is only necessary when considering conditionals). However, because they deal with distributed systems where restrictions on key usage cannot be enforced, they do not associate a secondary security level with cryptographic keys which means that if a high security key is used to encrypt some low security data, the result of the subsequent decryption is forced to be high.

Bengtson et al. [15] have developed an extended typechecker for F# code that is annotated with *refinement types*. A refinement type includes a logical formula which places restrictions upon the associated term. They consider an active adversary and use a generalised version of the symbolic cryptography model. The focus of their research is on authentication and authorisation properties for security protocols, but the flexibility afforded by refinement types means that the technique may be applicable to related domains such as security API analysis. However, due to the different target domain, the underlying type system that Bengtson et al. employ is quite different from the one which we give in this paper.

9 Conclusions and Future Work

Using typing rules for analysing the security properties of cryptographic systems is not new (e.g., [16]), but it is common for restrictions to be placed upon the use of encryption and decryption, as well as on any keys involved. Consequently, certain security APIs cannot be analysed using some of these existing systems. For example, the IBM 4758 [3] has one internal master key that is used to encrypt all other keys which are then stored on the attached host, therefore rule sets in which the result of a decryption cannot be used as a key (e.g., [17]) are unable to analyse the security API for that device.

In this paper, we have presented the foundations of a type system that is designed to deal with common features of security APIs such as encrypted keys and nested encryptions. We gave a definition of indistinguishability which captures the potential for an adversary to determine that the keys used in two ciphertexts are different, even though their actual values remain unknown. We then proved that well-typed command sequences are non-interferent with respect to this definition. We also proved the type-safety of our system meaning that the type information can be ignored at run-time.

The next stage of our research is to extend our type system to include additional features present in Volpano, Smith and Irvine's original type system [1] and Volpano and Smith's extension [18] — specifically procedures, primitive operations and conditional statements. This will allow us to analyse more accurate representations of functions in widely used security APIs such as PKCS #11 [20]. Adding conditionals will require a modified definition of the indistinguishability of stores, similar to the one given by Focardi and Centenaro [14]. It should be noted that such a change will not affect our results for the indistinguishability of expressions.

Further ahead, we plan to extend our type system to deal with data integrity, since this is equally as important as data confidentiality for key management APIs, as well as permitting explicit declassification thus allowing our system to analyse an additional class of security APIs.

References

1. Volpano, D.M., Smith, G., Irvine, C.E.: A Sound Type System for Secure Flow Analysis. Journal of Computer Security **4**(3) (1996) 167–187
2. Sumii, E., Pierce, B.C.: Logical Relations for Encryption. In: Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), IEEE Computer Society Press (June 2001) 256–269

3. IBM 4758 PCI Cryptographic Coprocessor.
http://www-03.ibm.com/security/cryptocards/pcicc/overview.shtml.
4. nCipher nShield Hardware Security Module. http://www.ncipher.com/en/Products/Hardware%20Security%20Modules/nShield.aspx.
5. Cortier, V., Keighren, G., Steel, G.: Automatic Analysis of the Security of XOR-Based Key Management Schemes. In: Proceedings of the 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007). Number 4424 in Lecture Notes in Computer Science, Springer-Verlag (March 2007) 538–552
6. Courant, J., Monin, J.F.: Defending the Bank with a Proof Assistant. In: Proceedings of the 6th International Workshop on Issues in the Theory of Security (WITS '06). (March 2006) 87–98
7. Delaune, S., Kremer, S., Steel, G.: Formal Analysis of PKCS #11. [19] 331–344
8. Youn, P.: The Analysis of Cryptographic APIs using the Theorem Prover Otter. Master's thesis, Massachusetts Institute of Technology (May 2004)
9. Youn, P., Adida, B., Bond, M.K., Clulow, J., Herzog, J., Lin, A., Rivest, R.L., Anderson, R.J.: Robbing the Bank with a Theorem Prover. Technical Report 644, University of Cambridge Computer Laboratory (August 2005)
10. Abadi, M., Rogaway, P.: Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In: TCS '00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, Springer-Verlag (2000) 3–22
11. Clulow, J.S.: On the Security of PKCS #11. In: Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003). Volume 2779 of Lecture Notes in Computer Science., Cologne, Germany, Springer-Verlag (September 2003) 411–425
12. Vaughan, J.A., Zdancewic, S.: A Cryptographic Decentralized Label Model. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy, IEEE Computer Society Press (May 2007) 192–206
13. Laud, P.: Handling Encryption in an Analysis for Secure Information Flow. In: Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003. Volume 2618 of Lecture Notes in Computer Science., Springer-Verlag (April 2003) 159–173
14. Focardi, R., Centenaro, M.: Information Flow Security of Multi-threaded Distributed Programs. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '08), ACM Press (June 2008) 113–124
15. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement Types for Secure Implementations. [19] 17–32
16. Abadi, M.: Secrecy by Typing in Security Protocols. In: Proceedings of TACS '97: 3rd International Symposium on Theoretical Aspects of Computer Software. Volume 1281 of Lecture Notes in Computer Science., Springer-Verlag (September 1997) 611–638
17. Laud, P., Vene, V.: A Type System for Computationally Secure Information Flow. In: Fundamentals of Computation Theory. Lecture Notes in Computer Science, Springer-Verlag (September 2005) 365–377
18. Volpano, D.M., Smith, G.: A Type-Based Approach to Program Security. In: Proceedings of TAPSOFT '97, Colloqium on Formal Approaches in Software Engineering. (1997) 607–621
19. Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008), IEEE Computer Society Press (June 2008)
20. PKCS #11 : Cryptographic Token Interface Standard.
http://www.rsa.com/rsalabs/node.asp?id=2133.