

UNIVERSITÀ CA' FOSCARI DI VENEZIA  
Dipartimento di Informatica  
Technical Report Series in Computer Science

Rapporto di Ricerca CS-2009-6

Giugno 2009

M. Centenaro, R. Focardi, F.L. Luccio, G. Steel

Type-based Analysis of PIN Processing APIs  
(full version)

Dipartimento di Informatica, Università Ca' Foscari di Venezia  
Via Torino 155, 30172 Mestre-Venezia, Italy

# Type-based Analysis of PIN Processing APIs (full version)

Matteo Centenaro<sup>1</sup>, Riccardo Focardi<sup>1</sup>, Flaminia L. Luccio<sup>1</sup> and Graham Steel<sup>2</sup>

<sup>1</sup> Dipartimento di Informatica, Università Ca' Foscari Venezia

<sup>2</sup> LSV, ENS Cachan & CNRS & INRIA, France

**Abstract.** We examine some known attacks on the PIN verification framework, based on weaknesses of the underlying security API for the tamper-resistant Hardware Security Modules used in the network. We specify this API in an imperative language with cryptographic primitives, and we show how its flaws are captured by a notion of robustness that extends the one of Myers, Sabelfeld and Zdancewic to our cryptographic setting. We propose an improved API, give an extended type system for assuring integrity and for preserving confidentiality via randomized and non-randomized encryptions, and show our new API to be type-checkable.

**Keywords.** Security APIs, Language-based security, PIN processing, Hardware Security Modules.

## 1 Introduction

In the international ATM (cash machine) network, users' personal identification numbers (PINs) have to be sent encrypted from the *PIN Entry Device* (PED) on the terminal to the issuing bank for checking. The PIN is encrypted in the PED under a key shared with the server or *switch* to which the ATM is connected. The PIN is then decrypted and re-encrypted under the key for an adjacent switch, to which it is forwarded. Eventually, the PIN reaches the issuing bank, by which time it may have been decrypted and re-encrypted several times. The issuing bank has no direct control over what happens in the intermediate switches, so to establish trust, the international standard ISO 9564 (ANSI X9.8) stipulates the use of tamper proof cryptographic *Hardware Security Modules* (HSMs). These HSMs protect the PIN encryption keys, and in the issuing banks, they also protect the *PIN Derivation Keys* (PDKs) used to derive the customer's PIN from non-secret validation data such as their *Personal Account Number* (PAN). All encryption, decryption and checking of PINs is carried out inside the HSMs, which have a carefully designed API providing functions for *translation* (i.e., decryption under one key and encryption under another one) and *verification* (i.e., PIN correctness checking). The API has to be designed so that even if an attacker obtains access to the host machine connected to the HSM, he cannot abuse the API to obtain PINs.

In the last few years, several attacks have been published on the APIs in use in these systems [8–10]. Very few of these attacks directly reveal the PIN. Instead, they involve the attacker calling the API commands repeatedly with slightly different parameter values, and using the results (which may be error codes) to deduce the value of the PIN. High-profile instances of many PINs being stolen from hacked switches has increased interest in the problem [1, 2]. PIN recovery attacks have been formally analysed, but previously the approach was to take a particular API configuration and measure its vulnerability to combinations of known attacks [26]. Other researchers have proposed improvements to the system to blunt the attacks, but these suggestions address only some attacks, and are “intended to stimulate further research” [21]. We take a step in that direction, using the techniques of language-based security [24].

Looking at the code for the current PIN processing APIs, one can immediately see that the current API functions allow an ‘information flow’ from the high security PIN to the low security result. However, it is a necessary feature of the verification function to reveal whether the encrypted PIN is correct or not, so some flow is inevitable. The language-based security literature has a technique for dealing with this: a ‘declassification policy’ [25], whereby we decide in advance that a certain flow is

permitted. The problem is that an intruder can often manipulate input data in order to declassify data in a way we did not intend. Again there is a technique for this: ‘robust declassification’ [22], whereby we disallow ‘low integrity’ data, which might have been manipulated by the attacker, to affect what can be declassified. However, the functionality of our PIN verification function requires the result to depend on low-integrity data. The solution in the literature is ‘endorsement’ [23], where we accept that certain low integrity data is allowed to affect the result. However, in our examples, endorsing the low integrity data permits several known attacks.

From this starting point, we propose in this paper an extension to the language-based security framework for robust declassification to allow the integrity of inputs to be assured cryptographically by using *Message Authentication Codes* (MACs). We present semantics and a type system for our model, and show how it allows us to formally analyse possible improvements to PIN processing APIs. We believe our modelling of cryptographically assured integrity to be a novel contribution to language based security theory. In addition, we give new proposals for improving the PIN processing system.

There is not room here to describe the operation of the ATM network in detail. Interested readers are referred to existing literature [10, 21, 26]. In this paper, we first introduce our main case study, the PIN verification command (section 2). We review some notions of language based security (section 3). We describe our modelling of cryptographic primitives, and in particular MACs for assuring integrity, and we show why PIN verification fails to be robust (section 4). Our type system is presented in section 5. The MAC-based improved API is type-checked in section 6. We conclude in section 7.

## 2 The Case Study

In the introduction we have observed how PINs travelling along the network have to be decrypted and re-encrypted under a different key, using a *translation* API. Then, when the PIN reaches the issuing bank, its correspondence with the *validation data*<sup>3</sup> is checked via a *verification* API. We focus on this latter API, which we call PIN\_V: it checks the equality of the actual *user* PIN and the *trial* PIN inserted at the ATM and returns the result of the verification or an error code. The former PIN is derived through the PIN derivation key *pdk*, from the public data *offset*, *vdata*, *dectab* (see below), while the latter comes encrypted under key *k* as *EPB* (Encrypted PIN block). Note that the two keys are pre-loaded in the HSM and are never exposed to the untrusted external environment. In this example we will assume only one key of each type (*k* and *pdk*) is used. The API, specified in Table 1, behaves as follows:

-The user PIN of length *len* is obtained by encrypting validation data *vdata* with the PIN derivation key *pdk* ( $x_1$ ), taking the first *len* hexadecimal digits ( $x_2$ ), decimalising through *dectab* ( $x_3$ ), and digit-wise summing modulo 10 the *offset* ( $x_4$ ). More precisely, the outcome of the encryption  $x_1$  is a 16 hexadecimal digit string and decimalize is a function that associates to each possible hexadecimal digit (of its second input) a decimal one as specified by its first parameter (*dectab*). The obtained decimalised value  $x_3$  is the ‘natural’ PIN assigned by the issuing bank to the user. If the user wants to choose her own PIN, an *offset* is calculated by digit-wise subtracting (modulo 10) the natural PIN from the user-selected one. Thus, to get the user PIN the *offset* is summed to the natural PIN ( $x_4$ ).

**Table 1** The PIN verification API.

---

```

PIN_V (PAN, EPB, len, offset, vdata, dectab) {
  x1 := encpdk(vdata);
  x2 := left(len, x1);
  x3 := decimalize(dectab, x2);
  x4 := sum_mod10(x3, offset);
  x5 := deck(EPB);
  x6 := fcheck(x5);
  if (x6 == "FAIL") then return("format error");
  if (x4 == x6) then return("PIN is correct");
  else return("PIN is wrong");
}

```

---

<sup>3</sup> The value of this parameter is up to the issuing bank. It is typically an encoding of the user PAN and possibly other ‘public’ data, such as the card expiration date or the customer name.

-The trial PIN is recovered by decrypting  $EPB$  with key  $k(x_5)$ , and extracting the PIN by removing the random padding and checking the PIN is correctly formatted ( $x_6$ ). For some PIN block formats the PAN is required for the extraction. However we give here the algorithm for extracting the PIN from an ISO1 block, where encryption with random padding is used. In our model this amounts to projecting the first element of a pair consisting of a PIN and random padding. We will discuss other block formats and how to model them later.

-Finally, the equality of the user PIN ( $x_4$ ) and the trial PIN ( $x_6$ ) is returned.

*Example 1.* Let  $len=4$ ,  $offset=4732$ ,  $dectab = 9753108642543210$ , this last parameter encoding this mapping:  $0 \rightarrow 9, 1 \rightarrow 7, 2 \rightarrow 5, 3 \rightarrow 3, 4 \rightarrow 1, 5 \rightarrow 0, 6 \rightarrow 8, 7 \rightarrow 6, 8 \rightarrow 4, 9 \rightarrow 2, A \rightarrow 5, B \rightarrow 4, C \rightarrow 3, D \rightarrow 2, E \rightarrow 1, F \rightarrow 0$ . Let also  $x_1 = \text{enc}_{pdk}(vdata) = A47295FDE32A48B1$ .

Then,  $x_2 = \text{left}(4, A47295FDE32A48B1) = A472$ ,  $x_3 = \text{decimalize}(dectab, A472) = 5165$ , and  $x_4 = \text{sum\_mod10}(5165, 4732) = 9897$ . This completes the user PIN recovery part. Let now  $(9897, r)$  denote PIN 9897 correctly formatted and padded with a random  $r$ , as required by ISO1 (recall that we are omitting details about other PIN formats for the moment) and let us assume that  $EPB = \{9897, r\}_k$ . We thus have:  $x_5 = \text{dec}_k(\{9897, r\}_k) = (9897, r)$ , and  $x_6 = \text{fcheck}(9897, r) = 9897$ . Finally, since  $x_6$  is different from "FAIL" and  $x_4 = x_6$  the API returns "PIN is correct".  $\square$

The given specification is an abstraction and a simplification of real PIN verification code, i.e., `PIN_V` corresponds to `Encrypted_PIN_Verify` of [17] simplified by omitting some parameters for alternative PIN extraction methods. We only model the IBM 3624 PIN calculation method with offset, but this is not limiting as the other PIN calculation methods can be similarly specified and analysed.

### 3 Basic Language and Security

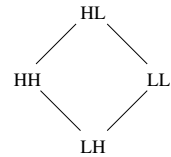
In this section, we recall a standard imperative language core and some basic security notions.

The syntax of *expressions* is:  $e ::= x \mid e_1 \text{ op } e_2$ , where  $x$  ranges over variables  $Var$ , and  $\text{op}$  ranges over arithmetic and Boolean (noted  $b$ ) operations on expressions. The syntax of *commands* is:  $c ::= \text{skip} \mid x := e \mid c_1; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c$ .

Given that we are interested in analysing security APIs, which we assume to be executed on trusted hardware with no multi-threading, we adopt a standard big-step semantics similar to that of Volpano et al. [28]. Memories  $M$  are finite maps from variables to values and we write  $M(x)$  to denote the value associated to  $x$  in  $M$ . Moreover,  $e \downarrow^M v$  denotes the evaluation of expression  $e$  in a memory  $M$  giving value  $v$  as a result: for example,  $x \downarrow^M M(x)$  and  $x + x' \downarrow^M M(x) + M(x')$ .

$\langle M, c \rangle \Rightarrow M'$  denotes the execution of a command  $c$  in a memory  $M$ , resulting in a new memory  $M'$ . Finally,  $M[x \mapsto v]$  denotes the update of variable  $x$  to the new value  $v$ . For example,  $\langle M, x := e \rangle \Rightarrow M[x \mapsto v]$  if  $e \downarrow^M v$ . The complete semantics is reported in Table 6 of the Appendix.

**Security** A *security environment*  $\Gamma$  maps each variable to a level of *confidentiality* and *integrity*. To keep the setting simple, we limit our attention to two possible levels: *high* ( $H$ ) and *low* ( $L$ ). For any given confidentiality (integrity) levels  $\ell_1, \ell_2$ , we write  $\ell_1 \sqsubseteq_C \ell_2$  ( $\ell_1 \sqsubseteq_I \ell_2$ ) to denote that  $\ell_1$  is as restrictive or less restrictive than  $\ell_2$ . In particular, low-confidentiality data may be used more liberally than high-confidentiality ones, thus in this case  $L \sqsubseteq_C H$ ; dually low-integrity data must be treated more carefully than high-integrity ones, giving the counter-variant relation  $H \sqsubseteq_I L$ . We consider the product of the above confidentiality and integrity lattices, and we denote with  $\sqsubseteq$  the component-wise application of  $\sqsubseteq_C$  and  $\sqsubseteq_I$  (on the right).



**Definition 1 (Indistinguishability).** Let  $M|_\ell$  denote the restriction of memory  $M$  to variables whose security level is at or below level  $\ell$ . We say that two memories  $M_1, M_2$  are indistinguishable at level  $\ell$ , written  $M_1 =_\ell M_2$ , if they are the same when observed at level  $\ell$  or below, i.e.,  $M_1|_\ell = M_2|_\ell$ . Two configurations are indistinguishable, written  $\langle M_1, c \rangle =_\ell \langle M_2, c \rangle$ , if whenever  $\langle M_1, c \rangle \Rightarrow M'_1$  and  $\langle M_2, c \rangle \Rightarrow M'_2$  then  $M'_1 =_\ell M'_2$ .

Noninterference requires that data from one level should never interfere with lower levels. Intuitively, command  $c$  satisfies noninterference if, fixed a level  $\ell$ , two indistinguishable memories remain indistinguishable even after  $c$  has been executed.

**Definition 2 (Noninterference).** A command  $c$  satisfies noninterference if  $\forall \ell, M_1, M_2$  we have that  $M_1 =_\ell M_2$  implies  $\langle M_1, c \rangle =_\ell \langle M_2, c \rangle$ .

To see how this captures confidentiality/integrity leakages, consider the cases  $\ell = LL$  and  $\ell = HH$ . The former says that high-confidentiality data cannot be leaked to low-confidentiality levels  $LL, LH$ . Dually, the latter case states that low-integrity data cannot corrupt high-integrity ones  $HH, LH$ .

Noninterference formalizes full security, with no leakage of confidential information or corruption of high-integrity data. The property proposed by Myers, Sabelfeld and Zdancewic (MSZ) in [23], called *robustness*, admits some form of *declassification* (or downgrading) of confidential data, but requires that attackers cannot influence the secret information declassified by a program  $c$ . In our case study of section 2, PIN\_V returns the correctness of the typed PIN which is a one-bit leak of information about a secret datum. Thus, the API is intended to declassify some secret information. Robustness will allow us to check that attackers cannot abuse such a declassification and gain more information than intended.

Consider a pair of memories  $M_1, M_2$  which are not distinguishable by an intruder, i.e.,  $M_1 =_{LL} M_2$ . The execution of  $c$  on these memories may leak confidential information violating noninterference, i.e.,  $\langle M_1, c \rangle \neq_{LL} \langle M_2, c \rangle$ . Robustness states that if the behaviour of the command  $c$  is not distinguishable on  $M_1$  and  $M_2$  then the same must happen for every pair of memories  $M'_1, M'_2$  the attacker may obtain starting from  $M_1, M_2$ . To characterize these memories note that: (i) they are still indistinguishable by the intruder, i.e.,  $M'_1 =_{LL} M'_2$ , as he is deterministic and starts from indistinguishable memories; (ii) they only differ from the initial ones in the low-integrity part, i.e.,  $M_1 =_{HH} M'_1, M_2 =_{HH} M'_2$ , given that only low-integrity variables can be modified by intruders.

As done by MSZ, we require that attackers start from terminating configurations to avoid they ‘incompetently’ self-corrupt their observations. This is done via a notion of *strongly indistinguishability*, written  $\langle M_1, c \rangle \cong_\ell \langle M_2, c \rangle$ , requiring that both configurations terminate, i.e.,  $\langle M_1, c \rangle \Rightarrow M'_1, \langle M_2, c \rangle \Rightarrow M'_2$ , and resulting memories are indistinguishable, i.e.,  $M'_1 =_\ell M'_2$ .

**Definition 3 (Robustness).** Command  $c$  is robust if  $\forall M_1, M_2, M'_1, M'_2$  s.t.  $M_1 =_{LL} M_2, M'_1 =_{LL} M'_2, M_1 =_{HH} M'_1, M_2 =_{HH} M'_2$ , it holds  $\langle M_1, c \rangle \cong_{LL} \langle M_2, c \rangle$  implies  $\langle M'_1, c \rangle =_{LL} \langle M'_2, c \rangle$ .

This notion is a novel simplification of the one of MSZ, who allowed a malicious user to insert untrusted code at given points in the trusted code. In security APIs this is not permitted: an attacker can call a security API any number of times with different parameters but he can never inject code inside it, moreover, no intermediate result will be made public by the API. This leads to a simpler model where attackers can only act before and after each security API invocation and, thus, there is no need to make their code explicit. Memory manipulations performed by attackers are covered by considering all the pairs of indistinguishable memories which only differ in the low-integrity part with respect to the initial ones.

*Example 2.* We will write  $x_\ell$  to denote a variable of level  $\ell$ . Consider a program  $P$  in which variable  $x_{LL}$  stores the user entered PIN,  $y_{HH}$  contains the real one, and  $z_{LL} := (x_{LL} = y_{HH})$ , i.e.,  $z_{LL}$  says if

the entered PIN is the correct one or not. This program does not obey to noninterference since the result of the equality test depends on secret data and it is assigned to a public variable, moreover it is not robust. To see this latter fact, consider memories  $M_1$  and  $M_2$  such that  $M_1(x_{LL}) = M_2(x_{LL}) = 1111$ , thus  $M_1 =_{LL} M_2$ , and  $M_1(y_{HH}) = 1234$  while  $M_2(y_{HH}) = 5678$ . Now assume the attacker generates two memories  $M'_1$  and  $M'_2$  where  $M'_1(y_{HH}) = M'_1(x_{LL}) = M'_2(x_{LL}) = 1234$ , thus  $M'_1 =_{LL} M'_2$ , and  $M'_2(y_{HH}) = 5678$ . It clearly holds that  $M_1 =_{HH} M'_1$  and  $M_2 =_{HH} M'_2$  but the execution of  $P$  in the first two memories leads to indistinguishable results in  $z_{LL}$ , false/false, thus  $\langle M_1, P \rangle \cong_{LL} \langle M_2, P \rangle$ , while for the second ones we get true/false, and so  $\langle M'_1, P \rangle \not\cong_{LL} \langle M'_2, P \rangle$ . Intuitively, the attacker has ‘guessed’ one of the secret PINs and the program is revealing that his guess is correct: the attacker can tamper with the declassification mechanism via variable  $x_{LL}$ .  $\square$

## 4 Cryptographic primitives

In order to model our API case-study, we now extend the language given in section 3 with confounder generation, (symmetric) cryptography, Message Authentication Codes (MACs), pairing and projection. These are introduced as special expressions, ranged over by  $e$ :

$$e ::= \dots \mid \text{new}() \mid \text{enc}_x(e) \mid \text{dec}_x(e) \mid \text{mac}_x(e) \mid \text{pair}(e_1, e_2) \mid \text{fst}(e) \mid \text{snd}(e)$$

working on values  $v ::= n \mid \{v\}_k \mid \langle v \rangle_k \mid (v_1, v_2)$  where  $n ::= \perp \mid r \mid k \mid \dots$  is an atomic name which can be the special value  $\perp$  representing failure, a confounder  $r \in C$ , a cryptographic key  $k \in \mathcal{K}$ , or any other value used in a program, e.g., Booleans and integers. On these atomic values we build cryptographic values and pairs ranged over by  $v$ : more specifically,  $\{v\}_k$  and  $\langle v \rangle_k$  respectively represent the encryption and the MAC of  $v$  using  $k$  as key, and  $(v_1, v_2)$  is a pair of values. We will often omit the brackets to simplify the notation, e.g., we will write  $\{v_1, v_2\}_k$  to indicate  $\{(v_1, v_2)\}_k$ .

Based on this set of values we can easily give the semantics of the special expressions mentioned above. For example, we have  $\text{enc}_x(e) \downarrow^M \{v\}_k$  whenever  $e \downarrow^M v$  and  $x \downarrow^M k$ . Moreover,  $\text{dec}_x(e') \downarrow^M v$  if  $e' \downarrow^M \{v\}_k$  and  $x \downarrow^M k$ ; otherwise  $\text{dec}_x(e') \downarrow^M \perp$ , representing failure, and analogously for the other expressions. Confounder generation  $\text{new}() \downarrow^M r$  extracts a ‘random’ value, noted  $r \leftarrow C$ , from a set of values  $C$ . In real cryptosystems, the probability of extracting the same random confounder is assumed to be negligible, if the set is suitably large, so we symbolically model random extraction by requiring that extracted values are always different. Thus,  $r \leftarrow C$  can be thought as extracting the first element of an infinite stream of confounders and removing it from the list so that it cannot be reused. More formally, we assume that two extractions  $r, r' \leftarrow C$  are such that  $r \neq r'$ . Moreover, similarly to [3, 4], we assume  $C$  to be disjoint from the set of atomic names used in programs. Semantics of expressions is summarized in Table 7 of Appendix.

To guarantee a safe use of cryptography we also assume that every expression  $e$  different from  $\text{enc}$ ,  $\text{dec}$ ,  $\text{mac}$ ,  $\text{pair}$  and every Boolean expression except the equality test: (i) always fails when applied to special values such as confounders, keys, ciphertexts, and MACs (even when occurring in pairs), producing a  $\perp$  and (ii) never produces those values. This is important to avoid “magic” expressions which encrypt/decrypt/MAC messages without knowing the key like, e.g.,  $\text{magicdecrypt}(e) \downarrow^M v$  when  $e \downarrow^M \{v\}_n$ . However, we permit equality checks as they allow the intruder to track if the same encryption appears twice, as occurs in traffic analysis.

### 4.1 Security with cryptography

We now rephrase the notions of noninterference and robustness in order to accommodate cryptographic primitives. In doing so, we extend [13] in a non-trivial way by (i) accounting for integrity primitives such as MACs and (ii) removing the assumption that cryptography is always randomized

via confounders. This latter extension is motivated by the fact that our case study does not always adopt randomization in cryptographic messages, so we need to leave the programmer decide whether or not to insert confounders in encryptions. Notice that non-randomized encrypted messages are subject to traffic analysis, thus confidentiality of those messages cannot be guaranteed except in special cases that we will discuss in detail.

In order to extend the indistinguishability notion of definition 1 to cryptographic primitives we assume that the level of keys is known a-priori. We believe this is a fair assumption, since in practice it is fundamental to have information about a key's security before using it. Since we have only defined symmetric key cryptography we only need *trusted* (of level  $HH$ ) and *untrusted* keys (of level  $LL$ ). The former are only known by the APIs while the latter can be used by the attackers. This is achieved by partitioning the set of keys  $\mathcal{K}$  into  $\mathcal{K}_{HH}$  and  $\mathcal{K}_{LL}$ .

**Patterns and indistinguishability** As the intruder cannot access (or generate, in case of MACs) cryptographic values protected by  $HH$  keys, one might state that such values are indistinguishable. However, an attacker might detect occurrences of the same cryptographic values in different parts of the memory, as occurs in some traffic analysis attacks.

*Example 3.* Consider the program  $z_{LL} := (x_{LL} = y_{LL})$  which writes the result of the equality test between  $x_{LL}$  and  $y_{LL}$  into  $z_{LL}$ . Given that it only works on  $LL$  variables it can be considered as an intruder-controlled program.

$M_1$	$M_2$
$x_{LL} : \{1234\}_k$	$x_{LL} : \{9999\}_k$
$y_{LL} : \{1234\}_k$	$y_{LL} : \{5678\}_k$

Consider the memories  $M_1$  and  $M_2$  (on the right) where the initial value of  $z_{LL}$  is not of interest, and where  $k \in \mathcal{K}_{HH}$ . At first sight, one may conclude the two memories are indistinguishable as an attacker cannot distinguish  $\{1234\}_k$  from  $\{9999\}_k$  and  $\{1234\}_k$  from  $\{5678\}_k$ . However, running the above intruder-program on these memories, we obtain two new memories  $M'_1$  and  $M'_2$  in which  $x_{LL}$  and  $y_{LL}$  remain unchanged, but  $z_{LL} = \text{true}$  in the first one and  $z_{LL} = \text{false}$  in the other one, i.e.,  $M'_1$  and  $M'_2$  differ. The intruder has in fact detected the presence of two equal ciphertexts in the first memory which allows him to distinguish the initial memories  $M_1$  and  $M_2$ .  $\square$

This ability of the attacker to find equal cryptographic values in the memories is formalized through the notion of *pattern* inspired by Abadi et al. [4, 5] and already adopted for modelling noninterference [13, 19]. Note that we adopt patterns to obtain a realistic notion of distinguishability of ciphertexts in a symbolic model, and not to address computational soundness as is done in other works [4–6].

Patterns, ranged over by  $p$ , extend values as follows:  $p ::= v \mid \square_v$  the new symbol  $\square_v$  representing messages encrypted with a key not available at the observation level  $\ell$ . More precisely, we define a function  $p_\ell(v)$  which takes a value and produces the corresponding pattern by replacing all the encrypted values  $v$  protected by keys of level  $\ell' \not\sqsubseteq \ell$  with  $\square_v$ , and leaving all the other values unchanged. For example, for  $\{1234\}_k$  in the example above we have  $p_{LL}(\{1234\}_k) = \square_{\{1234\}_k}$  while  $p_{HH}(\{1234\}_k) = \{1234\}_k$ . Notice that, in  $\square_v$ ,  $v$  is the whole (inaccessible) encrypted value, instead of just a counfounder as used in previous works [4, 5, 13, 19]. In these works, each new encryption includes a fresh confounder which can be used as a ‘representative’ of the whole encrypted value. Here we cannot adopt this solution since our confounders are optional. To disregard the values of counfounders, once the corresponding ciphertext has been accessed (i.e., when knowing the key), we abstract them as the constant  $\perp$ .

Given a bijection  $\rho : \square_v \mapsto \square_v$ , that we call *hidden values substitution*, we write  $p\rho$  to denote the result of applying  $\rho$  to the pattern  $p$ , and we write  $M\rho$  to denote the memory in which  $\rho$  has been applied to all the patterns of  $M$ , i.e.,  $M\rho(x) = M(x)\rho$ . On hidden values substitutions we always require that keys are correctly mapped. Formally  $\rho(\square_{\{v\}_k}) = \square_{\{v'\}_k}$ . We write  $\mathcal{K}_{\sqsubseteq \ell}$  to denote the set of keys at or below  $\ell$ , i.e.,  $\cup_{\ell' \sqsubseteq \ell} \mathcal{K}_{\ell'}$ .

**Definition 4.** Let  $\mathbf{p}_\ell(v) : v \mapsto p$  be recursively defined as follows:

$$\begin{aligned} \mathbf{p}_\ell(n) &= n && n \text{ not a confounder} \\ \mathbf{p}_\ell(r) &= \perp \\ \mathbf{p}_\ell((v_1, v_2)) &= (\mathbf{p}_\ell(v_1), \mathbf{p}_\ell(v_2)) \\ \mathbf{p}_\ell(\langle v \rangle_k) &= \langle \mathbf{p}_\ell(v) \rangle_k \\ \mathbf{p}_\ell(\{v\}_k) &= \begin{cases} \square_{\{v\}_k} & \text{if } k \notin \mathcal{K}_{\square\ell} \\ \{\mathbf{p}_\ell(v)\}_k & \text{otherwise} \end{cases} \end{aligned}$$

Let  $\mathbf{p}_\ell(M)$  denote the restriction of memory  $M$  to level  $\ell$  in which all of the values  $v$  have been substituted by  $\mathbf{p}_\ell$ , formally  $\mathbf{p}_\ell(M) = \mathbf{p}_\ell \circ M|_\ell$ . Two memories  $M_1$  and  $M_2$  are indistinguishable at level  $\ell$ , written  $M_1 \approx_\ell M_2$ , if there exists a hidden values substitution  $\rho$  such that  $\mathbf{p}_\ell(M_1) = \mathbf{p}_\ell(M_2) \rho$ .

*Example 4.* Consider again the two memories  $M_1$  and  $M_2$  of example 3. We observed that they differ at level  $LL$  because of the presence of two equal ciphertexts in  $M_1$ . Since  $k \in \mathcal{K}_{HH}$  we obtain the values of  $x_{LL}$  and  $y_{LL}$  below. Now it is impossible to find a hidden values substitution  $\rho$  mapping the first memory to the second, as  $\square_{\{1234\}_k}$  cannot be mapped both to  $\square_{\{9999\}_k}$  and  $\square_{\{5678\}_k}$ . Thus we conclude that  $M_1 \not\approx_{LL} M_2$ . If, instead,  $M_1(y_{LL})$  were, e.g.,  $\{2222\}_k$  we might use  $\rho = [\square_{\{1234\}_k} \mapsto \square_{\{9999\}_k}, \square_{\{2222\}_k} \mapsto \square_{\{5678\}_k}]$  obtaining  $\mathbf{p}_\ell(M_1) = \mathbf{p}_\ell(M_2) \rho$  and thus  $M_1 \approx_{LL} M_2$ .  $\square$

$\mathbf{p}_{LL}(M_1)$	$\mathbf{p}_{LL}(M_2)$
$x_{LL} : \square_{\{1234\}_k}$	$x_{LL} : \square_{\{9999\}_k}$
$y_{LL} : \square_{\{1234\}_k}$	$y_{LL} : \square_{\{5678\}_k}$

Notice that we do not extend our notion of hidden values to MACs, i.e., we assume that all messages inside MACs are public, and hence equivalence between MACs can be defined on these messages. We will use encryption only for secrecy, and MACs only for authentication.

**Noninterference and robustness** We now reconsider the security notions of section 3 in the new cryptographic setting. The idea is to keep the same definitions and substitute  $=_\ell$  with  $\approx_\ell$  everywhere. For example, the notion of (weak) indistinguishability of executions (definition 1), denoted  $\langle M, c \rangle \approx_\ell \langle M', c \rangle$ , is just rephrased by stating that whenever  $\langle M_1, c \rangle \Rightarrow M'_1$  and  $\langle M_2, c \rangle \Rightarrow M'_2$  then  $M'_1 \approx_\ell M'_2$ .

We need to be careful that memories do not leak cryptographic keys, i.e., that keys disclosed at level  $\ell$  are all of that level or below. We should also discipline the quantification over all possible memories so that variables intended to contain keys really do contain keys. We prefer to postpone these aspects to section 5, after types have been defined. In fact, they are easily achieved by requiring that variables of key-types are only populated by key-values, as the intuition suggests, and that memories are well-formed with respect to types.

In the next section we illustrate a known practical attack on the PIN verification API and we then show that it can be formally captured by the nonrobustness of the API.

## 4.2 Formal analysis of a PIN\_V API attack

In this section we show how the lack of integrity of the API parameters can be exploited to mount a real attack leaking the PIN.

Let us consider the case study of section 2 and in particular the code of the PIN verification API shown in Table 1. Let us now concentrate on two specific parameters, the *dectab* and the *offset*, which are used to calculate the values of  $x_3$  and  $x_4$ , respectively. A possible attack on the system works by iterating the following two steps, until the whole PIN is recovered [9]:

1. The intruder picks a decimal digit  $d$ , changes the *dectab* function so that values previously mapped to  $d$  now map to  $d+1 \bmod 10$ , and then checks whether the system still returns "PIN is correct".

Depending on this, the intruder discovers whether or not digit  $d$  is present in the user ‘natural’ PIN contained in  $x_3$ , thus extracting information on the user PIN digits;

2. when a certain digit is discovered in the previous step by a "*PIN is wrong*" output, the intruder also changes the *offset* until the API returns again that the PIN is correct. This allows the intruder to locate the position of the deduced PIN digit.

*Example 5.* Recall that in Example 1 we assumed  $len = 4$ ,  $dectab = 9753108642543210$ ,  $offset = 4732$ ,  $x_1 = A47295FDE32A48B1$ ,  $EPB = \{9897, r\}_k$ . As we have shown, with these parameters the API returns "*PIN is correct*".

Assume now the attacker chooses the new  $dectab' = 9753118642543211$ , where the two 0's have been replaced by 1's. The aim is to discover whether or not 0 appears in  $x_3$ . If we invoke the API with  $dectab'$  we obtain the same intermediate and final values, as  $decimalize(dectab', A472) = decimalize(dectab, A472) = 5165$ . This means that 0 does not appear in  $x_3$ .

The attacker now proceeds by removing from the  $dectab$  the next decimal digit until the API fails: with  $dectab'' = 9753208642543220$ , i.e., by replacing digit 1 with digit 2, we obtain that  $decimalize(dectab'', A472) = 5265 \neq decimalize(dectab, A472) = 5165$ , reflecting the presence of 1 in the original value of  $x_3$ . Then,  $x_4 = \text{sum\_mod10}(5265, 4732) = 9997$  instead of 9897 thus returning "*PIN is wrong*".

The intruder now knows that digit 1 occurs in  $x_3$ . To discover its position and multiplicity, he now tries variations of the offset so to ‘compensate’ for the modification of the  $dectab$ . In particular, he tries to decrement each offset digit by 1. For example, testing the position of one occurrence of one digit amounts to trying the following offset variations:  $3732, 4632, 4722, 4731$ . Notice that, in this specific case, offset value 4632 makes the API return again "*PIN is correct*". In fact  $x_3 = decimalize(dectab, A472) = 5165$  and  $x_4 = \text{sum\_mod10}(5165, 4732) = 9897$  with  $dectab$ , and  $x_3 = decimalize(dectab'', A472) = 5265$  and  $x_4 = \text{sum\_mod10}(5265, 4632) = 9897$  with  $dectab''$ . Notice, in particular, that the value of  $x_4$  is the same. The attacker now knows that the second digit of  $x_3$  is 1. Given that the *offset* is public, he also calculates the second digit of the user PIN as  $1 + 7 \bmod 10 = 8$ .

The above attack is based on the lack of integrity of the input data, which allows an attacker to influence the declassification mechanism as shown below.

In order to model the PIN derivation encryption of  $x_1$ , we now adopt a small trick: we write  $vdata = \{A47295FDE32A48B1\}_{pdk}$  and the encryption can now be modelled as a decryption  $x_1 := \text{dec}_{pdk}(vdata)$ . The reason for this is that we have a symbolic model for encryption that does not produce any low level bit-string encrypted data. Notice also that this model is reasonable, as the high-confidentiality of the encrypted value is ‘naturally’ protected by the *HH* PIN derivation key.

Consider now two memories,  $M$  and  $M_1$  that store the correct parameters of Example 1 for the PIN verification API call (parameters which are all at level *LL*), but contain  $dectab''$  instead of  $dectab$  and have different encryption values, i.e.,  $EPB$  and a different  $EPB_1$ , respectively. Note that  $M$  and  $M_1$  could be built by an attacker sniffing all encryptions arriving at the verification facility.

It holds that  $M \approx_{LL} M_1$  since the only unequal values are randomised and so can be re-named to make their patterns equal. If we execute *PIN\_V* in  $M$  and  $M_1$  we obtain "*PIN is wrong*" in both cases as for memory  $M_1$ , the encrypted PIN is wrong, and for memory  $M$ , the encrypted PIN is correct but the  $dectab''$  will change the value of derived PIN. It follows that  $\langle M, \text{PIN\_V} \rangle \approx_{LL} \langle M_1, \text{PIN\_V} \rangle$ .

Now suppose that the intruder replaces  $dectab''$  with the original value  $dectab$ , and plugs this into the above memories obtaining memories  $M_2$  and  $M_3$ . Note that  $M_2 \approx_{LL} M_3$ , however  $M_2$  will return "*PIN is correct*" since all the data correspond, whereas memory  $M_3$  will return "*PIN is wrong*" because the encrypted PIN is incorrect. Thus,  $\langle M_2, \text{PIN\_V} \rangle \not\approx_{LL} \langle M_3, \text{PIN\_V} \rangle$ , and hence robustness does not hold. To overcome this problem, integrity of the input must be established.  $\square$

## 5 Type system

In this section we give a type system to statically check that a program with cryptographic primitives satisfies noninterference and/or robustness. We will then use it to type-check a MAC-based variant of the PIN verification API and a MAC-based variant of a translation API we will briefly introduce later.

We refine integrity levels by introducing the notion of *dependent domains* used to track integrity dependencies among variables. Dependent domains are denoted  $D : \tilde{D}$  where  $D \in \mathcal{D}$  is a domain name. Intuitively, the values of domain  $D : \tilde{D}$  are determined by the values in the set of domains  $\tilde{D}$ . For example,  $\text{PIN} : \text{PAN}$  can be read as ‘the value of PIN is fixed relative to the account number PAN’: when the PAN is fixed, the value of the PIN is also fixed. A domain  $D : \emptyset$ , also written  $D$ , whose integrity does not depend on other domains is called an *integrity representative* and it can be used as a reference for checking the integrity of other domains. In fact, integrity representatives cannot be modified by programs and their values remain constant at run-time.

The integrity level associated to a dependent domain  $D : \tilde{D}$  is written  $[D : \tilde{D}]$ , and is at a higher integrity level than  $H$ , i.e.,  $[D : \tilde{D}] \sqsubseteq_I H$ . In some cases, e.g., in arithmetic operations, we necessarily lose information about the precise result domain  $D : \tilde{D}$  and we only record the fact the value is determined by domains  $\tilde{D}$ , written  $[\bullet : \tilde{D}]$ . In this case we know the value is determined by at most variables of domains  $\tilde{D}$ , but we have no precise information on its domain. The obtained integrity levels are thus:  $\delta_I ::= L \mid H \mid [D : \tilde{D}] \mid [\bullet : \tilde{D}]$ , while confidentiality levels are still  $\delta_C ::= L \mid H$ , as before. The preorder of integrity levels is extended as:

$$[D : \tilde{D}_1] \sqsubseteq [\bullet : \tilde{D}_1] \sqsubseteq_I [\bullet : \tilde{D}_2] \sqsubseteq_I H \sqsubseteq_I L$$

with  $\tilde{D}_1 \subseteq \tilde{D}_2$ . We will write  $C$  in place of  $[\bullet]$ , to denote a constant value with no specific domain. Based on these levels, we give the type syntax:

$$\tau ::= \delta \mid \text{cK}_\delta^\mu(\tau) \ \kappa \mid \text{enc}_\delta \ \kappa \mid \text{mK}_\delta(\tau) \mid (\tau_1, \tau_2)$$

A variable of type  $\delta$  contains generic data at level  $\delta$ ; types  $\text{cK}_\delta^\mu(\tau) \ \kappa$  and  $\text{mK}_\delta(\tau)$  respectively refer to encryption and MAC keys of level  $\delta$ , working on data of type  $\tau$ ;  $\kappa$  is a label that uniquely identifies one key and is used to reconstruct types when decrypting high integrity ciphertexts; we write  $K(\kappa) = \text{cK}_\delta^\mu(\tau) \ \kappa$  to refer to such a unique type; label  $\mu$  indicates whether the ciphertext is ‘randomized’ via confounders ( $\mu = R$ ) or not ( $\mu$  missing); we only consider untrusted keys of level  $LL$  and trusted ones of level  $HC$ : trusted keys are not modifiable thanks to the constant integrity type  $C$ ;  $\text{enc}_\delta \ \kappa$  is the type for ciphertexts at level  $\delta$ , obtained using a key labelled  $\kappa$ , thus we will always assume that two key types with the same  $\kappa$  are exactly the same type; pairs are typed as  $(\tau_1, \tau_2)$  and the two types are required to have the same confidentiality level and, for integrity levels at or above  $H$ , even the same integrity level. Intuitively, this requirement is to avoid information leakage when projecting elements from the pairs. Dependent domains will never be observed from ‘inside’, since our observation level will always be in the four-point lattice, making them appear as a unique higher level.

A *security type environment*  $\Delta : x \mapsto \tau$  maps variables to security types. The security environment  $\Gamma$ , suitably extended to the new integrity levels, can be derived from the type environment  $\Delta$  by just ‘extracting’ from the types their security level. This is done via the following level function  $\mathcal{L} : \tau \mapsto \delta$  defined as  $\mathcal{L}(\delta) = \mathcal{L}(K_\delta(\tau) \ \kappa) = \mathcal{L}(\text{enc}_\delta \ \kappa) = \delta$  and  $\mathcal{L}((\tau_1, \tau_2)) = \mathcal{L}(\tau_1) \sqcup \mathcal{L}(\tau_2)$ . Notice that we write  $K_\delta(\tau) \ \kappa$  to indifferently denote encryption and MAC key types. We will also write  $\mathcal{L}_C(\tau)$  and  $\mathcal{L}_I(\tau)$  to respectively extract the confidentiality and integrity level of type  $\tau$ . Formally, if  $\mathcal{L}(\tau) = \delta_C \delta_I$  then  $\mathcal{L}_C(\tau) = \delta_C$  and  $\mathcal{L}_I(\tau) = \delta_I$ . From now on we will assume  $\Gamma = \mathcal{L} \circ \Delta$ .

The subtype preorder  $\leq$  extends the security level preorder  $\sqsubseteq$  on levels  $\delta$  with  $\text{enc}_{\delta_C \delta_I} \ \kappa \leq \delta_C L$ . Moreover, from now on, we will implicitly identify low-integrity types at the same security level, i.e.,

**Table 2** Security Type System - Expressions

---

(var) $\frac{\Delta(x) = \tau}{\Delta \vdash x : \tau}$	(op) $\frac{\Delta \vdash e_1 : \delta \quad \Delta \vdash e_2 : \delta \quad \mathcal{L}_I(\delta) \neq [D : \tilde{D}]}{\Delta \vdash e_1 \text{ op } e_2 : \delta}$	(sub) $\frac{\Delta \vdash e : \tau' \quad \tau' \leq \tau}{\Delta \vdash e : \tau}$
(pair) $\frac{\Delta \vdash e_1 : \tau_1 \quad \Delta \vdash e_2 : \tau_2}{\Delta \vdash \text{pair}(e_1, e_2) : (\tau_1, \tau_2)}$	(fst/snd) $\frac{\Delta \vdash e : (\tau_1, \tau_2)}{\Delta \vdash \text{fst}(e) : \tau_1 \quad \Delta \vdash \text{snd}(e) : \tau_2}$	
(enc) $\frac{\Delta(x) = \text{cK}_\delta(\tau) \kappa \quad \Delta \vdash e : \tau}{\Delta \vdash \text{enc}_x(e) : \text{enc}_{\delta \sqcup \mathcal{L}(\tau)} \kappa}$	(dec) $\frac{\Delta(x) = \text{cK}_\delta(\tau) \kappa \quad \Delta \vdash e : \text{enc}_{\delta'} \kappa}{\Delta \vdash \text{dec}_x(e) : \delta \sqcup \delta'}$	
(enc-r) $\frac{\Delta(x) = \text{cK}_{HC}^R(\tau) \kappa \quad \Delta \vdash e : \tau}{\Delta \vdash \text{enc}_x^R(e) : \text{enc}_{LC \sqcup \mathcal{L}_I(\tau)} \kappa}$	(dec- $\mu$ ) $\frac{\Delta(x) = \text{cK}_{HC}^\mu(\tau) \kappa \quad \Delta \vdash e : \text{enc}_{\delta_C \sqcup \mathcal{L}_I(\tau)} \kappa \quad \mathcal{L}_C(\tau) = H}{\Delta \vdash \text{dec}_x^\mu(e) : \tau}$	
(enc-d) $\frac{\Delta(x) = \text{cK}_{HC}(\tau) \kappa \quad \Delta \vdash e : \tau \quad \text{CloseDD}^{\text{det}}(\tau)}{\Delta \vdash \text{enc}_x(e) : \text{enc}_{LC \sqcup \mathcal{L}_I(\tau)} \kappa}$	(mac) $\frac{\Delta(x) = \text{mK}_{\delta_C \delta_I}(\tau) \quad \Delta \vdash e : \tau}{\Delta \vdash \text{mac}_x(e) : LL \sqcup \mathcal{L}(\tau)}$	

---

we will not distinguish  $\tau$  and  $\tau'$  whenever  $\mathcal{L}(\tau) = \mathcal{L}(\tau') = \delta_C L$ , written  $\tau \equiv \tau'$ . This reflects the intuitions that we do not make any assumption on what is stored into a low-integrity variable. We do not include high keys in the subtyping and we also disallow the encryption (and the MAC) of such keys: formally, in  $\text{K}_\delta(\tau) \kappa$  and  $(\tau_1, \tau_2)$  types  $\tau, \tau_1, \tau_2 \neq \text{K}_{HC}(\tau) \kappa$ . We believe that transmission of high keys can be easily accounted for but we leave this extension as future work.

**Closed key types** In some typing rules we will require that types transported by cryptographic keys are ‘closed’, meaning that they are all dependent domains and all the dependencies are satisfied, i.e., all the required representatives are present. As an example, consider  $\text{cK}_{HC}^\mu(\tau) \kappa$  with  $\tau = (H[D], H[D' : D])$ . Types transported by the key are all dependent domains, noted  $\text{DD}(\tau)$ , and are closed: the set of dependencies, noted  $\text{Dep}(\tau)$ , is  $\{D\}$ , since  $[D' : D]$  depends on  $D$ , and the set of representatives, noted  $\text{IRs}(\tau)$ , is  $\{D\}$ , because of the presence of the representative  $[D]$ . If we instead consider  $\tau' = (H[D], H[D' : D], H[D' : D''])$  we have that the set of dependencies is  $\{D, D''\}$  and the set of representatives is  $\{D\}$ , meaning that the type is not closed: not all the dependencies can be found in the type. We will write  $\text{CloseDD}(\tau)$  to denote that  $\tau$  is closed, formally expressed as  $\text{Dep}(\tau) \subseteq \text{IRs}(\tau)$ , and only contains dependent domains, written  $\text{DD}(\tau)$ . When it additionally does not transport nested randomized ciphertexts we write  $\text{CloseDD}^{\text{det}}(\tau)$ . We will describe the importance of this closure conditions when describing the typing rules. In section A of appendix we report the formal definition of the above predicates.

**Expression typing rules** Expressions are typed with judgment of the form  $\Delta \vdash e : \tau$ , derived from the rules in Table 2. The first five rules are pretty standard. The only unusual requirement is  $\mathcal{L}_I(\delta) \neq [D : \tilde{D}]$  in rule (op). This forbids the typing of an operation with the dependent domain of the operands. This is because for an arbitrary operation, we cannot predict the precise value of the result. However, the dependency on other domains  $\tilde{D}$ , noted  $[\bullet : \tilde{D}]$ , is correctly preserved meaning that the result of the operation depends ‘in some way’ from  $\tilde{D}$ . Recall, in fact, that  $[D : \tilde{D}] \leq [\bullet : \tilde{D}]$ .

The first two rules for cryptography (enc) and (dec) correspond to the (op) rule described above: they allow one to encrypt and decrypt at a level which is the least upper bound of the levels of the key and the plaintext/ciphertext. Recall that we only consider untrusted  $LL$  keys and trusted  $HC$  ones.

The remaining rules are more interesting: rule (enc-r) is for randomized encryption: We let  $\text{enc}_x^R(e)$  and  $\text{dec}_x^R(e)$  denote, respectively,  $\text{enc}_x(e, \text{new}())$  and  $\text{fst}(\text{dec}_x(e))$ , i.e., an encryption randomized via a fresh confounder and the corresponding decryption. The typing rule requires a trusted key  $HC$ . The

---

**Table 3** Security Type System - Commands

---

$$\begin{array}{c}
\text{(skip)} \quad \Delta, pc \vdash \text{skip} \quad \text{(seq)} \quad \frac{\Delta, pc \vdash c_1 \quad \Delta, pc \vdash c_2}{\Delta, pc \vdash c_1; c_2} \quad \text{(while)} \quad \frac{\Delta \vdash b : \tau \quad \Delta, \mathcal{L}(\tau) \sqcup pc \vdash c}{\Delta, pc \vdash \text{while } b \text{ do } c} \\
\text{(if)} \quad \frac{\Delta \vdash b : \tau \quad \Delta, \mathcal{L}(\tau) \sqcup pc \vdash c_1 \quad \Delta, \mathcal{L}(\tau) \sqcup pc \vdash c_2}{\Delta, pc \vdash \text{if } b \text{ then } c_1 \text{ else } c_2} \quad \text{(assign)} \quad \frac{\Delta(x) = \tau \quad \Delta \vdash e : \tau \quad pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH}{\Delta, pc \vdash x := e} \\
\text{(declassify)} \quad \frac{\Delta(x) = \delta_C H \quad \Delta \vdash e : \delta'_C H \quad pc \sqsubseteq \delta_C H}{\Delta, pc \vdash x := \text{declassify}(e)} \\
\text{(if-MAC)} \quad \frac{\Delta(x) = \text{mK}_{HC}(L[D], \tau) \quad \Delta \vdash z : L[D] \quad \Delta \vdash e : LL \quad \Delta \vdash e' : LL \quad \Delta(y) = \tau}{\text{IRs}(L[D], \tau) = \{D\} \quad \text{CloseDD}(L[D], \tau) \quad \Delta, pc \vdash c_1 \quad \Delta, pc \vdash c_2 \quad pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH} \\
\Delta, pc \vdash \text{if } \text{mac}_x(z, e) = e' \text{ then } (y := e; c_1) \text{ else } c_2; \perp_{\text{MAC}}
\end{array}$$


---

integrity level is handled as before (notice that  $C$  has the effect of removing all the information on domains, as for operations), while confidentiality of the ciphertext is  $L$ , meaning that it can be assigned to low-confidentiality variables and thus accessed by an attacker. The rule intuitively states that whenever a fresh confounder is encrypted with the plaintext, the resulting ciphertext preserves secrecy, even if sent on an public/untrusted part of the memory.

Rule (dec- $\mu$ ), when  $\mu = R$  is the dual of (enc-r). Notice that the rule gives the correct type  $\tau$  to the obtained plaintext. This can be done only if the confidentiality of the plaintext is at least as restrictive as the one of the key used, i.e., equal to  $H$ , since  $H$  is the highest possible.

Rule (enc-d) is the most original one. It encodes a way to guarantee secrecy even without confounders, i.e., with no randomization. The idea comes from format ISO0 for the EPB, which intuitively combines the PIN with the PAN before encrypting it in order to prevent codebook-attacks. Consider, for example the ciphertext  $\{\text{PAN}, \text{PIN}\}_k$ . Since every account, identified by the PAN, has its own PIN, the PIN can be thought of as at level  $[\text{PIN} : \text{PAN}]$  ('the PIN is fixed relative to the PAN'). Thus equal PANs will determine equal PINs, which implies that different PINs will always be encrypted together with different PANs, producing different EPBs. This avoids, for example, allowing an attacker to build up a codebook of all the PINs. Intuitively, the PAN is a sort of confounder that is 'reused' only when its own PIN is encrypted. The rule requires  $\text{CloseDD}^{\text{det}}(\tau)$ , i.e., that each sub-expression is at integrity level  $[D_i : D'_i]$ , and that dependent domains are closed, i.e., all the required integrity representatives are in  $\tau$ . This is important as they play the role of confounders, as explained above. Moreover it is also required that no random ciphertexts are included in the message to encrypt. As in (enc-r) integrity is propagated and confidentiality of the ciphertext is  $L$ , meaning that it is safe to assign it to low confidentiality variables.

Rule (mac) is for the generation of MACs: it is similar to (op). The only interesting difference is that the confidentiality level of the key does not contribute to the confidentiality level of the MAC, which just takes the one of  $e$ . This reflects the fact that we only use MACs for integrity and we always assume the attacker knows the content of MACs, as formalized in definition 4. The reason why we force integrity to be low is related to the fact we want to forbid declassification of cryptographic values, which would greatly complicate the proof of robustness. By the way, this is not limiting as there are no good reasons to declassify what has been created to be low-confidentiality.

**Typing rules for commands** As in existing approaches [23] we introduce in the language a special expression  $\text{declassify}(e)$  for explicitly declassifying the confidentiality level of an expression  $e$  to  $L$ . This new expression has no operational import, i.e.,  $\text{declassify}(e) \downarrow^M v$  iff  $e \downarrow^M v$ . Declassification is

thus only useful in the type-system to isolate program points where downgrading of security happens, in order to control robustness.

Judgements for commands have the form  $\Delta, pc \vdash c$  where  $pc$  is the program counter level. It is a standard way to track what information has affected control flow up to the current program point [23]. For example, when entering a while loop, the  $pc$  is raised to be higher or equal to the level of the loop guard expression. This prevents such an expression to allow flows to lower levels. The  $pc$  is a level on the four point lattice, notice that, however, when its integrity level is high we let assignment to integrity levels below  $H$ , e.g., dependent domains, to take place: this makes sense since, as mentioned before, we never move our observation level below  $LH$ .

Typing rules for commands are depicted in Table 3. The first six rules are largely standard [23]. The  $pc$  is raised on entering if branches and while loops. Notice also that assignments are only possible at or above the  $pc$  level and, as mentioned above, at lower integrity levels if  $\mathcal{L}_I(pc) = H$ . Rule (declassify) lets a high integrity expression to be declassified, i.e., assigned to some high-integrity variable independent of its confidentiality level, when also the program counter is at high-integrity and the assignment to the variable is legal ( $pc \sqsubseteq \delta_C H$ ). The high-integrity requirement is for guaranteeing robustness: no attacker will be able to influence declassification.

The (if-MAC) rule is peculiar of our approach: it allows the checking of a MAC with respect to an integrity representative  $z$ . The rule requires that the first parameter  $z$  is typed at level  $L[D]$ ; the second parameter  $e$  and the MAC value  $e'$  are typed  $LL$ . If the MAC succeeds, variable  $y$  of type  $\tau$  is bound to the result of  $e$  through an explicit assignment in the if-branch. Notice that such an assignment would be forbidden by the type-system, as it is promoting the integrity of an  $LL$  expression to an unrestricted type  $\tau$  (as far as  $pc$  is high integrity). This can however be proved safe since the value returned by the  $LL$  expression matches an existing MAC, giving us enough guarantees about the integrity of the data, and allowing us to ‘reconstruct’ their type from the type of the MAC key.

Side conditions  $\text{IRs}(L[D], \tau) = \{D\}$  and  $\text{CloseDD}(L[D], \tau)$  ensure that the MAC contains only values which directly depends on the unique integrity representative given by variable  $z$ . The ‘then’ branch is typed without any particular restriction, while the ‘else’ one is required to end with a special failure command  $\perp_{\text{MAC}}$  which just aims at non-terminating the program (it may be equivalently though as a command with no semantics, which never reduces, or a diverging program as, e.g., while true do skip). This is needed to avoid the attacker breaks integrity and robustness by just calling an API with wrong MACs. In fact, we can assume the attacker knows which MACs pass the tests and which MACs do not (unless he is trying some brute-force/cryptanalysis attack on the MAC algorithm, that we do not account for here) and by letting the else branch fail we just disregard those obvious, uninteresting, information flows.

**Security results** We give an overview of the security results. Notice, however, that for lack of space we will necessarily omit all of the proofs and also some technical detail, in order to simplify the presentation. All the proofs can be found in the Appendix.

As mentioned above, our type-system aims at guaranteeing a form of noninterference and robustness, in the presence of cryptography and MAC-based integrity. Our results hold under some reasonable well-formedness/integrity assumptions on the memories: (i) variables of high level key-type really contain keys of the appropriate level, and such keys never appears elsewhere in the memory; (ii) values of variables or encrypted messages at integrity  $H$ , or below, must adhere to the expected type; for example, the value of a variable typed as high integrity pair is expected to be a pair; (iii) values for dependent domains  $[D : \tilde{D}]$  are uniquely determined by the values of the integrity representatives  $\tilde{D}$ , e.g., when they appear together in an encrypted message or a MAC or when they have been checked in an if-MAC statement; (iv) confounders are used once: there cannot be two different encrypted messages with the same confounder.

Condition (iii) states, for example, that if a MAC is expected (from the type of its key) to contain the PAN, of level  $[\text{PAN}]$  and the relative PIN, of level  $[\text{PIN} : \text{PAN}]$ , encrypted with another key, all of the possible MACs with that key will respect a function  $f_{[\text{PIN}:\text{PAN}]}$ , pre-established for each memory. Thus, equal PANs will imply equal encrypted PINs. Intuitively, even if the attacker does not know the actual PIN, he knows that what is encrypted is the unique PIN corresponding to the PAN. The value of integrity representatives, like the PAN above, is instead instantiated ‘on-the-fly’, during well-formedness check, via a function  $g$ . For example, let us assume  $f_{[\text{PIN}:\text{PAN}]}(\text{pan}_i) = \text{pin}_i$ . We have that all of these MACs are well-formed:  $\langle \text{pan}_1, \{\text{pin}_1\}_k \rangle_{k'}$ ,  $\langle \text{pan}_2, \{\text{pin}_2\}_k \rangle_{k'}$ ,  $\dots$ ,  $\langle \text{pan}_m, \{\text{pin}_m\}_k \rangle_{k'}$ , as they all respect  $f_{[\text{PIN}:\text{PAN}]}$ . Intuitively, when we check the well-formedness of these MACs we fix a function  $g(\text{PAN})$  the exact moment we check the value in the MAC, i.e., for the first MAC we choose  $g(\text{PAN}) = \text{pan}_1$ , and so on. A well-formed PIN value is thus the one that matches  $f_{[\text{PIN}:\text{PAN}]}(g(\text{PAN}))$ . Memories well-formedness, noted  $\Delta \vdash_g^f M$ , is fully described and formalized in section B of appendix.

Program that are run on well-formed memories always return well-formed memories:

**Proposition 1** *If  $\Delta, pc \vdash c$ ,  $\Delta \vdash_g^f M$  and  $\langle M, c \rangle \Rightarrow M'$  then  $\Delta \vdash_g^f M'$ .*

From now on, we will implicitly assume that memories are well-formed. The next result states that when no declassification occurs in a program, then noninterference holds. This might appear surprising as MAC checks seem to potentially break integrity: an attacker might manipulate one of the MAC parameters to gain control over the MAC check. In this way he can force the execution of one branch or the other, however recall that by inserting  $\perp_{\text{MAC}}$  at the end of the else branch we force that part of the program not to terminate. Weak indistinguishability will thus consider such an execution equivalent to any other, which means it will disregard that situation.

Next lemmas are used to prove the main results. The first one is peculiar of our extension with cryptography: if an expression is typed below the observation level  $\ell$ , we can safely assign it to two equivalent memories and still get equivalent memories. We cannot just check the obtained values in isolation as, by traffic analysis (modelled via patterns), two apparently indistinguishable ciphertext might be distinguished once compared via equality with other ciphertexts existing in the memories.

**Lemma 1 (Expression  $\ell$ -equivalence)** *Let  $M_1 \approx_\ell M_2$  and let  $\Delta \vdash e : \tau$  and  $e \downarrow^{M_i} v_i$ . If  $\mathcal{L}(\tau) \sqsubseteq \ell$  or  $\mathcal{L}(\Delta(x)) \not\sqsubseteq \ell$  then  $M_1[x \mapsto v_i] \approx_\ell M_2[x \mapsto v_i]$ .*

**Lemma 2 (Confinement)** *If  $\Delta, pc \vdash c$  then for every variable  $x$  assigned to in  $c$  and such that  $\Delta(x) = \tau$  it holds that  $pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH$ .*

**Theorem 1 (Noninterference)** *Let  $c$  be a program which does not contain any declassification statement. If  $\Delta, pc \vdash c$  then  $c$  satisfies noninterference, i.e.,  $\forall \ell \sqsupset LH, M_1, M_2. M_1 \approx_\ell M_2$  implies  $\langle M_1, c \rangle \approx_\ell \langle M_2, c \rangle$ .*

We can now state our final results on robustness. We will consider programs that assign declassified data to special variables assigned only once. This can be easily achieved syntactically, e.g., by using one different variable for each declassification statement (which we label for clarity), i.e.,  $x_1 := \text{declassify}_1(e_1), \dots, x_m := \text{declassify}_m(e_m)$ , and avoiding to place declassifications inside while loops. These special variables are nowhere else assigned. We call this class of programs *Clearly Declassifying* (CD). We do this to avoid, one more time, that attackers ‘incompetently’ hide flows by resetting variables after declassification has happened and, since our semantics is big-step, we would not detect these intermediate critical states. CD-programs, instead, keep declassified data unchanged in fixed variables up to their termination.

**Theorem 2 (Robustness)** *If a CD-program  $c$  is such that  $\Delta, pc \vdash c$  then  $c$  satisfies robustness, i.e.,  $\forall M_1, M_2, M'_1, M'_2$  such that  $M_1 \approx_{LL} M_2$ ,  $M'_1 \approx_{LL} M'_2$  and  $M_i \approx_{HH} M'_i$  it holds*

$$\langle M_1, c \rangle \approx_{LL} \langle M_2, c \rangle \text{ implies } \langle M'_1, c \rangle \approx_{LL} \langle M'_2, c \rangle$$

## 6 A type-checkable MAC-based API

We now discuss `PIN_V_M` a MAC-based improvement of `PIN_V`, which prevents the attack of section 4.2, and several others from the literature. We show `PIN_V_M` is type-checkable using our type system, and we also show where the original API fails to type-check.

We assume the intruder starts with only one EPB which contains the correct

PIN. An attacker equipped with several EPBs for the same PAN, only one of which contains the correct PIN, can always violate robustness: he can try all the EPBs until he identifies the one containing the correct PIN, influencing the declassification of data. We feel our assumption is justified because an attacker with a stolen or fake card can only make three guesses at the terminal before the account is blocked. If the attacker were able to circumvent this restriction, given sufficient patience, he could make a ‘brute force’ attack on the PIN from the terminal which we do not expect to be able to prevent with our API.

Our fix is reported in Table 4. The new API checks a MAC of all the parameters at the very beginning. Intuitively, the MAC check guarantees that the parameters have not been manipulated. Some form of ‘legal’ manipulation is always possible: an intruder can get a different set of parameters, e.g., eavesdropped in a previous PIN verification and referring to a *different* PAN, and can call the API with these parameters. Those parameters will have a correct MAC validating their integrity. This is actually captured by our notion of dependent domains by typing all the MAC checked variables as dependent from the PAN.

We need to refine the model given in Table 1 for the verification API (called by our MAC checking version). As discussed in section 4.2, we model PIN derivation as a *decryption*, obtaining the expected hexadecimal (secret) number from which the PIN is then derived. The only change needed in Table 1 is substituting  $x_1 := \text{enc}_{pdk}(vdata)$  with  $x_1 := \text{dec}_{pdk}(vdata)$ . We model the extraction of the PIN from an ISO1 block as randomized decryption. The resulting function is in Table 9 of the appendix.

We show typing in detail: All the parameters  $PAN, EPB, len, offset, vdata, dectab, MAC$  are of type  $LL$ , since we assume the attacker can read and modify them. The important element is the mac key  $ak$  which is given type  $\text{mK}_{HC}(\tau)$  with  $\tau = L[PAN]$ ,  $\text{enc}_{L[\bullet: PAN]} \kappa_{ek}$ ,  $L[LEN : PAN]$ ,  $L[OFFS : PAN]$ ,  $\text{enc}_{L[\bullet: PAN]} \kappa_{pdk}$ ,  $L[DECTAB : PAN]$ . Note that  $\text{IRs}(\tau) = \{PAN\}$ ,  $\text{Dep}(\tau) = \{PAN\}$ , thus we have  $\text{Closed}(L[PAN], \tau)$  and also  $\text{DD}(\tau)$  as  $\tau$  only contains types of the form  $[D : PAN]$ . All the checked variables are typed according to the above tuple, e.g.,  $PAN'$  with  $L[PAN]$ ,  $EPB'$  with  $\text{enc}_{L[\bullet: PAN]} \kappa_{ek}$  and so on. In the code, to make it readable, we directly assign MAC checked expressions to variables. To type-check, this assignment should be done in one variable, and than decomposed by projecting the first and second elements of pairs. The result of the API will be stored in the  $ret$  variable whose type is  $LL$ .

The key  $ek$  is typed as  $\text{cK}_{HC}^R(H[PIN : PAN]) \kappa_{ek}$  and key  $pdk$  as  $\text{cK}_{HC}^R(H[HEX : PAN]) \kappa_{pdk}$ . To complete the typing of the MAC we need to type the two branches. The else branch is trivial: the assignment to  $ret$  is legal and then it is followed by the MAC-fail command. The other one

**Table 4** The new `PIN_V_M` API with MAC-based integrity.

---

```

PIN_V_M(PAN, EPB, len, offset, vdata, dectab, MAC) {
  if (macak(PAN, EPB, len, offset, vdata, dectab) == MAC)
    then EPB' := EPB; len' := len; offset' := offset;
       vdata' := vdata; dectab' := dectab;
       PIN_V(PAN, EPB', len', offset', vdata', dectab');
  else ret := "integrity violation";  $\perp_{MAC}$ 
}

```

---

amounts to checking the original API with the new high integrity types. What happens is that  $x_1$  is typed  $H[\text{HEX} : \text{PAN}]$  by rule (dec- $\mu$ ) and  $x_2, \dots, x_4$  are typed  $H[\bullet : \text{PAN}]$  by rule (op).  $x_6$  is typed  $H[\text{PIN} : \text{PAN}]$  by rule (dec- $\mu$ ). Thus the declassification to  $x_7$ , which is typed  $LH$ , of the result of the comparison, is type-checkable as the expression types  $H[\bullet : \text{PAN}]$  and by subtype leads to  $HH$ . By theorem 2 we are guaranteed PIN\_V\_M is robust. In the original version of the API, without the MAC check,  $x_4$  and  $x_6$  would only be typeable with low integrity, and hence the declassification would violate robustness.

**PIN translation API** We conclude our results with a brief discussion of the *translation* API that we call PIN\_T\_M, used to decrypt and re-encrypt a PIN under a different key. Switches may not be able to support all known PIN formats, so the translation function might need to reformat messages under different block formats. In this paper we consider only the ISO-0 and ISO-1 formats (specified in table 8 of the appendix). ISO-0 pads the PIN with data derived from the PAN.

We specify the code of PIN\_T\_M for translating specifically from ISO-1 to ISO-0 in Table 10 of the appendix. The API takes a PIN block  $EPB_I$  and key  $k$ . It extracts the PIN, reformats it and re-encrypts it with key  $k'$ . Decryption is as in the previous section and gives the PIN in variable  $x_1$  with type  $H[\text{PIN} : \text{PAN}]$ . The PIN is now padded with the PAN and sent encrypted with the new key, which has type  $cK_{HC}(H[\text{PIN} : \text{PAN}], H[\text{PAN}]) \kappa_{k'}$ . Encryption is thus typed via (enc-d) giving  $EPB_O$  of type  $enc_{L[\bullet : \text{PAN}]} \kappa_{k'}$ . Recall that this is safe since the PAN is playing the role of a confounder. MAC creation is not problematic. The API type-checks and, given it does not contain any declassification, it satisfies noninterference (theorem 1).

## 7 Conclusions

We have presented our extensions to information flow security types to model deterministic encryption and cryptographic assurance of integrity for robust declassification. We have shown how to apply this to PIN processing APIs. Most previous approaches to formalising cryptographic operations in information flow analysis have aimed to show how a program that is noninterfering when executed in a secure environment can be guaranteed secure when executed over an insecure network by using cryptography, see e.g., [7, 13, 15, 19, 27]. They typically use custom cryptographic schemes with strong assumptions, e.g. randomised cryptography and/or signing of all messages. This means they are not immediately applicable to the analysis of PIN processing APIs, which have weaker assumptions on cryptography. [11] presents what seems to be the only information flow model for deterministic encryption, that shows soundness of noninterference with respect to the concrete cryptography model. However, it does not treat integrity. Gordon and Jeffrey's type system for authenticity in security protocols could be used to check correspondence assertions between the data sent from the ATM and the data checked at the API [16]. However, this would not address the problem of declassification, robustness or otherwise. Keighren et al. have outlined a framework for information flow analysis specifically for security APIs [18], though this also currently models confidentiality only. The formal analysis of security APIs has usually been carried out by Dolev-Yao style analysis of reachability properties in an abstract model of the API, e.g., [12, 20, 29]. This typically covers only confidentiality properties.

We plan in future to refine our framework on further examples from the PIN processing world and elsewhere, and to model other cryptographic primitives which can be used to assure integrity such as (unkeyed) hash functions and asymmetric key digital signatures. We have also begun to investigate practical ways to implement our scheme in cost-effective way (see [14]).

## References

1. Hackers crack cash machine PIN codes to steal millions. The Times online. [http://www.timesonline.co.uk/tol/money/consumer\\_affairs/article4259009.ece](http://www.timesonline.co.uk/tol/money/consumer_affairs/article4259009.ece).

2. PIN Crackers Nab Holy Grail of Bank Card Security. Wired Magazine Blog 'Threat Level'. <http://blog.wired.com/27bstroke6/2009/04/pins.html>.
3. M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM (JACM)*, 46(5):749–786, 1999.
4. M. Abadi and J. Jurjens. Formal eavesdropping and its computational interpretation. In *TACS: 4th International Conference on Theoretical Aspects of Computer Software*, volume 2215 of *LNCS*, pages 82–94, Tohoku University, Sendai, Japan, October 29-31 2001. Springer.
5. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *JCRYPTOL: Journal of Cryptology*, 15(2):103–127, 2002.
6. P. Adão, G. Bana, J. Herzog, and A. Scedrov. Soundness of formal encryption in the presence of key-cycles. In *10th European Symposium on Research in Computer Security (ESORICS)*, volume 3679 of *LNCS*, pages 374–396. Springer-Verlag, 2005.
7. A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. *Theoretical Computer Science*, 402(2-3):82–101, August 2008.
8. O. Berkman and O.M. Ostrovsky. The unbearable lightness of PIN cracking. In *11th International Conference, Financial Cryptography and Data Security (FC 2007)*, volume 4886 of *LNCS*, pages 224–238, Scarborough, Trinidad and Tobago, February 12-16 2007. Springer.
9. M. Bond and P. Zielinski. Decimalization table attacks for PIN cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory, 2003. <http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-560.pdf>.
10. J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.
11. J. Courant, C. Ene, and Y. Lakhnech. Computationally sound typing for non-interference: The case of deterministic encryption. In *27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, volume 4855 of *LNCS*, pages 364–375, New Delhi, India, December 12-14 2007. Springer.
12. S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.
13. R. Focardi and M. Centenaro. Information flow security of multi-threaded distributed programs. In Úlfar Erlingsson and Marco Pistoia, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'08)*, pages 113–124, Tucson, AZ, USA, June 8 2008. ACM Press.
14. R. Focardi, F.L. Luccio, and G. Steel. Improving pin processing api security. In *Workshop on Analysis of Security APIs*, to appear, 2009.
15. C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In George C. Necula and Philip Wadler, editors, *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 323–335, San Francisco, Ca, USA, January 10-12 2008. ACM Press.
16. A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. Technical Report MSR-2001-49, Microsoft Research, 2001.
17. IBM Inc. CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors. Technical report, 2006. Releases 2.53–3.27. Available at <http://www-03.ibm.com/security/cryptocards/pcicc/library.shtml>.
18. G. Keighren, A. Aspinall, and G. Steel. Towards a type system for security APIs. In *Proceedings of Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, pages 173–192, York, UK, March 28-29 2009.
19. P. Laud. On the computational soundness of cryptographically masked flows. In George C. Necula and Philip Wadler, editors, *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 337–348, San Francisco, Ca, USA, January 10-12 2008. ACM Press.
20. D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.
21. M. Mannan and P. van Oorschot. Reducing threats from flawed security APIs: The banking PIN case. *Computers & Security*, 28(6):410–420, September 2009.
22. A.C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proceedings of 17th Computer Security Foundations Workshop (CSFW)*, pages 172–186. IEEE Computer Society, 2004.
23. A.C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, May 2006.
24. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
25. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, to appear.
26. G. Steel. Formal Analysis of PIN Block Attacks. *Theoretical Computer Science*, 367(1-2):257–270, November 2006.
27. J.A. Vaughan and S. Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, pages 192–206. IEEE Computer Society, 2007.
28. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–187, December 1996.

29. P. Youn, B. Adida, M. Bond, J. Clulow, J. Herzog, A. Lin, R. Rivest, and R. Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, August 2005.

## Appendix

### A Closed key types

In some typing rules we need to recursively collect the level of the types of an encryption or MAC key, by descending into pairs and into high level encryptions. This is achieved by the following function  $LT_{\Delta}(\delta)$

$$\begin{aligned} LT_{\Delta}(\delta) &= \{\delta\} \\ LT_{\Delta}((\tau_1, \tau_2)) &= LT_{\Delta}(\tau_1) \cup LT_{\Delta}(\tau_2) \\ LT_{\Delta}(\text{enc}_{\delta} \kappa) &= \begin{cases} LT_{\Delta}(\tau) & \text{if } K(\kappa) = \text{cK}_{HC}^{\mu}(\tau) \kappa \\ \{LL\} & \text{otherwise} \end{cases} \end{aligned}$$

Notice that, if there is a cycle of encryption labels the computation of  $LT_{\Delta}(\delta)$  does not terminate. From now on we will always assume key types are acyclic. We also need to check when a type does not contains randomized encryptions.

$$\begin{aligned} \text{Det}(\delta) &= \text{true} \\ \text{Det}((\tau_1, \tau_2)) &= \text{Det}(\tau_1) \wedge \text{Det}(\tau_2) \\ \text{Det}(\text{enc}_{\delta} \kappa) &= \text{Det}(\tau) \wedge (K(\kappa) = \text{cK}_{\delta}(\tau) \kappa) \end{aligned}$$

Based on this function, we define some useful sets and predicates:

$$\begin{aligned} \text{IRs}(\tau) &= \{D \mid \delta_C[D] \in LT_{\Delta}(\tau)\} \\ \text{Dep}(\tau) &= \{D \mid D \in \tilde{D}, \delta_C[D' : \tilde{D}] \in LT_{\Delta}(\tau)\} \\ \text{DD}(\tau) &\text{ iff } \delta_C \delta_I \in LT_{\Delta}(\tau) \text{ implies } C \not\sqsubseteq \delta_I \\ \text{Closed}(\tau) &\text{ iff } \text{Dep}(\tau) \subseteq \text{IRs}(\tau) \\ \text{CloseDD}(\tau) &\text{ iff } \text{Closed}(\tau) \wedge \text{DD}(\tau) \\ \text{CloseDD}^{\text{det}}(\tau) &\text{ iff } \text{Closed}(\tau) \wedge \text{DD}(\tau) \wedge \text{Det}(\tau) \end{aligned}$$

respectively, the set  $\text{IRs}(\tau)$  of the integrity representatives in  $\tau$ ; the domains  $\text{Dep}(\tau)$  on which types of  $\tau$  depend; the fact, written  $\text{DD}(\tau)$ , that all the integrity levels are below  $C$ , i.e., are all of the form  $[D : \tilde{D}]$ ; a predicate  $\text{Closed}(\tau)$  checking the closure of dependent domains of  $\tau$  with respect to integrity representatives. For example,  $([D], [D' : D])$  is closed while  $([D], [D' : D''])$  is not;  $\text{CloseDD}(\tau)$  and  $\text{CloseDD}^{\text{det}}(\tau)$ , additionally requiring  $\text{DD}(\tau)$  and  $\text{Det}(\tau)$ .

### B Memory well-formedness

Well-formedness for values is given in Table 5.  $\Theta$  is an injective mapping from key values to their types. An integrity representative assumes the constant value determined by function  $g(D)$  where  $D$  is the domain of the representative (v-ir). A value which depends on a set of domains  $\{D_1, \dots, D_m\} = \tilde{D}'$  and belongs to the dependent domain  $D : \tilde{D}'$  is mapped by a function  $f_{D:D_1, \dots, D_m}$  taking the values of each of the integrity representative of the domains contained in  $\tilde{D}'$  and returning the value of the resulting domain (v-dd). The ordering of the  $D_i$  is important here to keep the parameters in the correct sequence. While  $g$  can change depending on the integrity context, we assume  $f$  is fixed a-priori for a memory and never changes while checking well-formedness. This amounts to fix the dependency among dependent domains without however fixing a-priori their values, which is instead determined once  $g$  is also fixed. We will always assume that  $f$  and  $g$  never returns keys or confounders. Finally, while  $g$  is partial, we always assume  $f$  is defined for each dependent domain  $D : \tilde{D}'$ .

MACs and values encrypted with trusted keys that should always contain ‘high-integrity’ values of type  $\tau$  (v-enc) and (v-mac), and a confounder in case of randomized encryption (v-enc-R). In fact,

**Table 5** Values well-formedness

Note: In rules (v-enc), (v-enc-r), (v-mac),  $g = g'$  when  $g \neq \epsilon$  or  $\mathcal{L}_I(\delta) = L$

$$\begin{array}{c}
\text{(v-name)} \frac{n \notin \mathcal{K} \quad \delta_I \neq [D : \tilde{D}']}{\Theta \vdash_g^f n : \delta_C \delta_I} \quad \text{(v-key)} \frac{\Theta(k) = \tau}{\Theta \vdash_g^f k : \tau} \quad \text{(v-sub)} \frac{\Theta \vdash_g^f v : \tau' \quad \tau' \leq \tau}{\Theta \vdash_g^f v : \tau} \quad \text{(v-pair)} \frac{\Theta \vdash_g^f v_1 : \tau_1 \quad \Theta \vdash_g^f v_2 : \tau_2}{\Theta \vdash_g^f (v_1, v_2) : (\tau_1, \tau_2)} \\
\\
\text{(v-ir)} \frac{g(D) \downarrow \quad n = g(D)}{\Theta \vdash_g^f n : \delta_C [D]} \quad \text{(v-dd)} \frac{g(D_i) \downarrow \quad n = f_{D: D_1, \dots, D_m}(g(D_1), \dots, g(D_m))}{\Theta \vdash_g^f n : \delta_C [D : \{D_1, \dots, D_m\}]} \quad \text{(v-mac)} \frac{\Theta(k) = \text{mK}_\delta(\tau) \quad \Theta \vdash_{g'}^f v : \tau}{\Theta \vdash_g^f \langle v \rangle_k : \delta_C L} \\
\\
\text{(v-enc)} \frac{\Theta(k) = \text{cK}_\delta(\tau) \quad \kappa \quad \Theta \vdash_{g'}^f v : \tau}{\Theta \vdash_g^f \{v\}_k : \text{enc}_{\delta_C \mathcal{L}_I(\delta) \sqcup \mathcal{L}_I(\tau)} \kappa} \quad \text{(v-enc-r)} \frac{\Theta(k) = \text{cK}_{HC}^R(\tau) \quad \kappa \quad \Theta \vdash_{g'}^f v : \tau}{\Theta \vdash_g^f \{v, r\}_k : \text{enc}_{\delta_C C \sqcup \mathcal{L}_I(\tau)} \kappa}
\end{array}$$

those values can only be generated by trusted, well-typed, programs. These rules also instantiate all the integrity representatives corresponding to domains appearing in  $\tau$  via a function  $g$ ; this is to avoid that the same representative assumes different values in the same high integrity context.

All the other rules work as expected. Notice that ciphertexts obtained through high level keys can be given confidentiality level  $L$  independently of the confidentiality level of the plaintext, reflecting the possibility of generating low-confidential ciphertexts containing high-confidentiality plaintexts. As well-formedness only aims at checking the integrity of values, we do not constraint the value of  $\delta_C$  as done, instead, when typing expressions.

It can be easily proved that any value in which only low-level keys appear is well-formed and can be given any possible low-integrity type, meaning that we do not restrict attacker ability to manipulate low-integrity memories apart from subvalues generated via high level keys. (Recall that types at level  $\delta_C L$  are all implicitly considered the same.)

Intuitively, a memory  $M$  is well-formed if all its variables stores well-formed values. For dependent domains we fix a-priori all the representatives via a specific  $g$ . This is to account for run-time situations in which a MAC check has been performed. Let  $g_\tau$  denote  $\epsilon$  when  $H \sqsubseteq \mathcal{L}_I(\tau)$  and  $g$  otherwise.

**Definition 5 (Memory well-formedness).** Let  $\Theta : k \mapsto \text{K}_\delta(\tau) \kappa$  be a mapping from keys to key-types such that  $k \in \mathcal{K}_{\mathcal{L}(\Theta(k))}$ , and injective for high level keys, i.e.,  $k, k' \in \mathcal{K}_{HC}$  implies  $\Theta(k) \neq \Theta(k')$ . A memory  $M$  is well-formed with respect to  $\Delta$  and  $g$  ( $\neq \epsilon$ ), written  $\Delta \vdash_g^f M$ , if the followings hold

- (i)  $\Delta(x) = \tau$  and  $M(x) = v$  imply  $\Theta \vdash_{g_\tau}^f v : \tau$
- (ii) If values  $\{v, r\}_k, \{v', r'\}_k$  occur in  $M$ , then  $\Theta(k) = \text{cK}_{HC}^R(\tau) \kappa$  and  $v \neq v'$  imply  $r \neq r'$ .

**Table 6** Command Semantics
$$\begin{array}{l}
[skip] \langle M, skip \rangle \Rightarrow M \quad [assign] \frac{e \downarrow^M v}{\langle M, x := e \rangle \Rightarrow M[x \mapsto v]} \quad [seq] \frac{\langle M, c_1 \rangle \Rightarrow M' \quad \langle M', c_2 \rangle \Rightarrow M''}{\langle M, c_1; c_2 \rangle \Rightarrow M''} \\
[if] \frac{b \downarrow^M \text{true} \quad \langle M, c_1 \rangle \Rightarrow M'}{\langle M, \text{if } b \text{ then } c_1 \text{ else } c_2 \rangle \Rightarrow M'} \quad [iff] \frac{b \downarrow^M \text{false}, \perp \quad \langle M, c_2 \rangle \Rightarrow M'}{\langle M, \text{if } b \text{ then } c_1 \text{ else } c_2 \rangle \Rightarrow M'} \\
[whilet] \frac{b \downarrow^M \text{true} \quad \langle M, c \rangle \Rightarrow M' \quad \langle M', \text{while } b \text{ do } c \rangle \Rightarrow M''}{\langle M, \text{while } b \text{ do } c \rangle \Rightarrow M''} \quad [whilef] \frac{b \downarrow^M \text{false}, \perp}{\langle M, \text{while } b \text{ do } c \rangle \Rightarrow M}
\end{array}$$
**Table 7** Expression Semantics

NOTE: We let  $e \downarrow^M v$ ,  $e_1 \downarrow^M v_1$ ,  $e_2 \downarrow^M v_2$  and  $x \downarrow^M k$ .

$$\begin{array}{l}
\text{new}() \downarrow^M r \quad r \leftarrow C \\
\text{enc}_x(e) \downarrow^M \{v\}_k \\
\text{dec}_x(e') \downarrow^M v \quad \text{if } e' \downarrow^M \{v\}_k \\
\text{mac}_x(e) \downarrow^M \langle v \rangle_k \\
\text{pair}(e_1, e_2) \downarrow^M (v_1, v_2) \\
\text{fst}(e') \downarrow^M v_1 \quad \text{if } e' \downarrow^M (v_1, v_2) \\
\text{snd}(e') \downarrow^M v_2 \quad \text{if } e' \downarrow^M (v_1, v_2)
\end{array}$$

All remaining cases, e.g., decryption with an incorrect key, evaluate to the special value  $\perp$  making the expressions total.

**Table 8** PIN-block formats

We report some of the standard PIN block formats. All of the formats below are 16 hexadecimal digits, i.e., 64 bits long, and support PINs from 4 to 12 digits in length.

**ISO-0, ANSI X9.8, VISA-1, ECI-1**

0	L	P	P	P	P	P	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	F	F
---	---	---	---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	---	---

0 identifies the format, L is the length of the PIN, P are the PIN digits while P/F is either e PIN digit or the pad value F, depending on the PIN length. Then the rightmost 12 digits of the PAN are written as follows:

0	0	0	0	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN
---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

and we now just xor the two numbers:

0	L	P	P	P	P	P	P	P	P	P	P	P	P	P	F	F
				xor	xor	xor	xor	xor	xor	xor	xor	xor	xor	xor	xor	xor
				PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN

**ISO-1**

1	L	P	P	P	P	P	P/R	P/R	P/R	P/R	P/R	P/R	P/R	P/R	R	R
---	---	---	---	---	---	---	-----	-----	-----	-----	-----	-----	-----	-----	---	---

1 identifies the format, L is the length of the PIN, P are the PIN digits while P/R is either e PIN digit or the pad random value R, depending on the PIN length.

---

**Table 9** Revised model of the PIN verification API with types.

---

```
PIN_V (PAN, EPB, len, offset, vdata, dectab) {  
  
    // deriving user PIN with IBM 3624 PIN calculation method:  
    x1 := decpdk(vdata);           // *decrypt* vdata with pdk - x1:H[HEX: PAN]  
    x2 := left(len, x1);           // takes len leftmost digits - x2:H[•: PAN]  
    x3 := decimalize(dectab, x2);  // decimalizes - x3:H[•: PAN]  
    x4 := sum_mod10(x3, offset);    // sums the offset - x4:H[•: PAN]  
  
    // recovering the trial PIN from ISO-1 block  
    x6 := deckR(EPB);               // decrypts the EPB with k - x6:H[PIN: PAN]  
  
    // checks trial PIN versus actual user PIN and declassifies the result  
    x7 := declassify(x4 = x6)      // declassify the result - x7:LH  
    if (x7) then ret := "PIN is correct"; // ret:LL  
        else ret := "PIN is wrong";  
}  


---


```

---

**Table 10** The new PIN\_T\_M API with MAC-based integrity, with types.

---

```
PIN_T_M (PAN, EPBI, EPBO, MACI, MACO) {  
  
    // checking the MAC of PIN decryption  
    if (macak(EPBI, PAN) = MACI) then  
    {  
        // recovering the trial PIN from ISO-1 block  
        x1 := deckR(EPBI);           // decrypts the EPB with k - x1:H[PIN: PAN]  
  
        EPB'O := enck'R(x1, PAN);      // PIN encryption ISO-0, PAN in padding  
                                           // - EPBO: encL[•: PAN] κk'  
        MACO := macak'(EPB'O, PAN);    // generates the next MAC - MACO:LL  
        EPBO := EPB'O;                 // rises to LL - EPBO:LL  
        ret := "success";               // ret:LL  
    }  
    else  
        ret := "Integrity Check Failed";  
}  


---


```

## Proofs

This appendix contains all the proofs of the results. In some cases we also add a few technical details that have been skipped in the paper for lack of space.

**Preliminaries** We first formalize various assumption that have been given along the paper (a few of them omitted for the sake of simplicity).

**Assumption 1 (Types)** *We always assume that:*

1. In  $(\tau_1, \tau_2)$  we have  $\mathcal{L}_C(\tau_1) = \mathcal{L}_C(\tau_2)$ . If  $H \sqsubseteq \mathcal{L}_I((\tau_1, \tau_2))$  then also  $\mathcal{L}_I(\tau_1) = \mathcal{L}_I(\tau_2)$ ;
2. In  $\text{cK}_\delta^\mu(\tau) \kappa$  and  $\text{cK}_{\delta'}^{\mu'}(\tau') \kappa'$ ,  $\kappa = \kappa'$  implies  $\text{cK}_\delta^\mu(\tau) \kappa = \text{cK}_{\delta'}^{\mu'}(\tau') \kappa'$ .
3. In  $\text{K}_\delta(\tau) \kappa$  and  $(\tau_1, \tau_2)$  types  $\tau, \tau_1, \tau_2 \neq \text{K}_{HC}(\tau) \kappa$
4. For types referring to the same key label as  $\text{enc}_\delta \kappa$  and  $\text{cK}_{\delta'}^\mu(\tau) \kappa$ , we always assume  $\mathcal{L}_I(\delta) = \mathcal{L}_I(\delta') \sqcup \mathcal{L}_I(\tau)$

Intuitively, (1) states that, in pairs, the two types are required to have the same confidentiality level and, for integrity levels at or above  $H$ , even the same integrity level; (2) formalizes that  $\kappa$  is a label that uniquely identifies one key type; (3) states that we disallow the encryption (and the MAC) of high level keys; (4) requires that the integrity level of encryption is coherent with the one of the key (notice that this is not limiting, as if the integrity level is not the right one it is impossible to type encryptions and decryptions).

**Assumption 2 (Expressions)** *For generic expressions it always holds*

1.  $e_1 \text{ op } e_2 \downarrow^M n$  with  $n \notin \mathcal{K} \cup \mathcal{C}$
2.  $e_1 \text{ op } e_2 \not\downarrow^M \perp$  then either  $\text{op}$  is '=' (i.e., the equality check) or  $e_i \downarrow^M n_i$  with  $n_i \notin \mathcal{K} \cup \mathcal{C}$
3. when the observation level  $\ell$  is  $HL$  (top) the equality check is defined as  $e_1 = e_2 \downarrow^M \text{true}$  iff  $e_i \downarrow^M v_i$  and  $\text{p}_{HL}(v_1) = \text{p}_{HL}(v_2)$  (instead of  $v_1 = v_2$ );  $e_1 = e_2 \downarrow^M \text{false}$  otherwise.

Intuitively, (1) generic expressions never generate keys or confounders; (2) they fail when applied to names which are not atomic and different from keys/confounders, unless the operation is the equality check; (3) equality check  $e_1 = e_2$  is evaluated as expected (i.e., syntactic equality of the resulting values) apart when the observation level is  $HL$ ; in this particular case we require it is consistent with equality of patterns at the actual observation level  $HL$ ; this assumption is needed to prove noninterference when  $\ell = HL$ . We might even remove this assumption and restrict noninterference result (theorem 4) to  $LL$  and  $HH$  which are anyway the levels of interest since they respectively represent the attacker and the trusted users. Robustness, in fact, is based on these two levels and does not depend on assumption 2(3). Technically, under this assumption, expression evaluation, and consequently program execution, depend on  $\ell$ ; to keep the notation as much simple as possible, we however prefer to omit to decorate with  $\ell$  program semantics. When not explicitly specified, we will always intend that programs are executed below  $HL$ , i.e., with the expected semantics for equality check ( $e_1 = e_2 \downarrow^M \text{true}$  iff  $e_i \downarrow^M v_i$  and  $v_1 = v_2$ , and  $e_1 = e_2 \downarrow^M \text{false}$  otherwise.)

**Assumption 3 (Confounders)** *Confounders  $r$  occur in memories only as  $\{v, r\}_k$  with  $\Theta(k) = \text{cK}_{HC}^R(\tau) \kappa$ .*

This assumption comes from the fact that (typed) programs only use confounders in the expected way, i.e., inside randomized ciphertexts. It is trivial to show that the assumption is preserved when running (typed) programs: it is enough to observe that, by assumption 2(1) above, the only expression which returns confounders is `new` and it is only (implicitly) typed by rule (enc-r), while randomized decryption always ‘throws away’ the confounder.

**Memory well-formedness** We now give some results on value well-formedness. To do so, we first need to formalize what is a subvalue. In particular, we formalize a notion of subvalues at level  $\delta$ , i.e., subvalues observable by only knowing keys at or below  $\delta$ . To simplify a bit the notation, we will often write  $v$  to denote the set  $\{v\}$  only containing value  $v$ . Recall that we write  $\mathcal{K}_{\sqsubseteq\delta}$  to denote the set of keys at or below  $\delta$ , i.e.,  $\cup_{\delta' \sqsubseteq \delta} \mathcal{K}_{\delta'}$ .

**Definition 6.** *The set of  $\delta$ -subvalues of a value  $v$ , written  $\text{Sub}_\delta(v)$  is recursively defined as:*

$$\begin{aligned} \text{Sub}_\delta(n) &= n \\ \text{Sub}_\delta((v_1, v_2)) &= (v_1, v_2) \cup \text{Sub}_\delta(v_1) \cup \text{Sub}_\delta(v_2) \\ \text{Sub}_\delta(\langle v \rangle_k) &= \langle v \rangle_k \\ \text{Sub}_\delta(\{\!\{v\}\!\}_k) &= \begin{cases} \{\!\{v\}\!\}_k & \text{if } k \notin \mathcal{K}_{\sqsubseteq\delta} \\ \{\!\{v\}\!\}_k \cup \text{Sub}_\delta(v) & \text{otherwise} \end{cases} \end{aligned}$$

We write  $\text{Sub}(v)$  to denote  $\text{Sub}_{HL}(v)$ . i.e., the set of all the subvalues of  $v$ .

Notice that keys used to create ciphertexts and MACs are not considered as subvalues. We actually assume it is never possible to deduce a key from a ciphertext or MAC. Those keys, when needed, can be extracted from a value by just looking at all the ciphertexts/MACs appearing as a subvalue. Formally

$$\mathcal{K}(v) = \{k \mid \{\!\{v'\}\!\}_k \in \text{Sub}(v) \vee \langle v' \rangle_k \in \text{Sub}(v)\}$$

Next lemma states that any value in which only low-level keys appear is well-formed and can be given any possible low-integrity type, meaning that we do not restrict attacker ability to manipulate low-integrity memories apart from subvalues generated via high level keys.

**Lemma 3** *If  $\mathcal{K}(v) \subseteq \mathcal{K}_{LL}$  and  $\text{Sub}(v) \cap \mathcal{K}_{HC} = \emptyset$  then  $\Theta \vdash^f v : LL$ .*

*Proof.* It is sufficient to consider  $\Theta$  such that  $\Theta(k) = LL$  for all  $k \in \mathcal{K}(v)$ . We now proceed by induction on the structure of  $v$ .

Base case:

$n$

By (v-name) we have that all names  $n$ , except keys, are such that  $\Theta \vdash^f n : LL$ . Since  $\text{Sub}(v) \cap \mathcal{K}_{HC} = \emptyset$  we have that  $n \notin \mathcal{K}_{HC}$ . When  $n \in \mathcal{K}_{LL}$ , by  $\Theta(k) = LL$  for all  $k \in \mathcal{K}(v)$  and (v-key) we obtain  $\Theta \vdash^f k : LL$ .

Inductive case:

$(v_1, v_2)$

Since  $\text{Sub}(v_1), \text{Sub}(v_2) \subseteq \text{Sub}(v)$  we also have  $\mathcal{K}(v_i) \subseteq \mathcal{K}(v) \subseteq \mathcal{K}_{LL}$  and  $\text{Sub}(v_i) \cap \mathcal{K}_{HC} \subseteq \text{Sub}(v) \cap \mathcal{K}_{HC} = \emptyset$ . Thus by inductive hypothesis we know that  $\Theta \vdash^f v_i : LL$ . By applying (v-pair) we directly obtain the thesis since  $(LL, LL) \equiv LL$ .

$\{\!\{v'\}\!\}_k$

As above, by induction we know that  $\Theta \vdash^f v' : LL$  and by  $\mathcal{K}(v) \subseteq \mathcal{K}_{LL}$  we know that  $\Theta(k) = LL$ . Recall we identify  $LL$  with every low-integrity/confidentiality type as, in particular,  $\text{cK}_{LL}(LL) \kappa \equiv LL$ . By (v-enc) we thus get  $\Theta \vdash^f \{\!\{v'\}\!\}_k : \text{enc}_{LL} \kappa \equiv LL$ .

$\langle v' \rangle_k$

We proceed exactly as in the previous case.  $\square$

We now prove that in any well-formed value, which is not itself a high level key, high level keys  $k$  never appear as subvalues, even if they can appear as keys as, e.g., in  $\{\!\{v\}\!\}_k$ .

**Lemma 4** *Let  $\Theta \vdash_g^f v : \tau$ . Then  $v \notin \mathcal{K}_{HC}$  implies  $\text{Sub}(v) \cap \mathcal{K}_{HC} = \emptyset$ .*

*Proof.* By induction on the structure of  $v$ :

Base case:

$n, \langle v \rangle_k$

Trivial since  $\text{Sub}(v) = \{v\}$ . By  $v \notin \mathcal{K}_{HC}$  we directly have that  $\text{Sub}(v) \cap \mathcal{K}_{HC} = \emptyset$ .

Inductive cases:

$(v_1, v_2)$

Let  $\Theta \vdash_g^f (v_1, v_2) : \tau$ . Since pairs are not in the subtyping relation, the judgement comes from (v-pair) implying that  $\tau = (\tau_1, \tau_2)$ ,  $\Theta \vdash_g^f v_1 : \tau_1$  and  $\Theta \vdash_g^f v_2 : \tau_2$ . By assumption 1(3), we have that  $\tau_i \neq \text{cK}_{HC}^\mu(\tau) \kappa$ . Notice that if  $v_i \in \mathcal{K}_{HC}$  it would be necessarily be typed as  $\text{cK}_{HC}^\mu(\tau) \kappa$  since the only possible rule for high keys is (v-key) and high level key-type is not in the subtype relation. So  $v_i \notin \mathcal{K}_{HC}$  and, by induction,  $\text{Sub}(v_i) \cap \mathcal{K}_{HC} = \emptyset$ . Thus,

$$\begin{aligned} \text{Sub}((v_1, v_2)) \cap \mathcal{K}_{HC} &= \\ &= (\text{Sub}(v_1) \cup \text{Sub}(v_2) \cup (v_1, v_2)) \cap \mathcal{K}_{HC} = \emptyset \end{aligned}$$

$\{\!\{v'\}\!\}_k$

Let  $\Theta \vdash_g^f \{\!\{v'\}\!\}_k : \tau$ . The judgement might derive from (v-sub) and one among (v-enc) and (v-enc-r), meaning  $\Theta(k) = \text{cK}_\delta^\mu(\tau') \kappa$  and  $\Theta \vdash_{g'}^f v' : \tau'$ . By assumption 1(3), we have that  $\tau' \neq \text{cK}_{HC}^\mu(\tau) \kappa$ . As for the above case, this means that  $v' \notin \mathcal{K}_{HC}$ . By induction we thus get  $\text{Sub}(v_i) \cap \mathcal{K}_{HC} = \emptyset$ . Thus,

$$\begin{aligned} \text{Sub}(\{\!\{v'\}\!\}_k) \cap \mathcal{K}_{HC} &\subseteq \\ &\subseteq (\text{Sub}(v') \cup \{\!\{v'\}\!\}_k) \cap \mathcal{K}_{HC} = \emptyset \end{aligned}$$

$\square$

Intuitively, judgments  $\Theta \vdash^f v : LL$ ,  $\Theta \vdash^f v : \tau$  and  $\Theta \vdash_g^f v : \tau$  have increasing discriminating power, as they are used to check the well-formedness of values at increasing integrity levels. We formally show that  $\Theta \vdash_g^f v : \tau$  always implies  $\Theta \vdash^f v : LL$ . Moreover, when  $\mathcal{L}_I(\tau) = H$ , it also implies  $\Theta \vdash^f v : \tau$ . This does not hold for high level key values whose integrity cannot be reduced by subtyping. For example, a high level key  $k$  can be typed as  $\Theta \vdash_g^f k : \tau$  but it can never be typed  $\Theta \vdash^f k : \delta_C L$ .

**Lemma 5** *Let  $v \notin \mathcal{K}_{HC}$ . Then*

(i)  $\Theta \vdash_g^f v : \tau$  implies  $\Theta \vdash^f v : LL$ ;

(ii)  $\Theta \vdash_g^f v : \tau$  and  $\mathcal{L}_I(\tau) = H$  imply  $\Theta \vdash^f v : \tau$ .

*Proof.* By induction on the structure of  $v$ .

Base case:

$n$

We have that all names  $n$ , except keys, are such that  $\Theta \vdash^f n : LL$ , thanks to (v-name). Moreover, the condition  $k \in \mathcal{K}_{\mathcal{L}(\Theta(k))}$  on  $\Theta$  ensures that low keys  $k \in \mathcal{K}_{LL}$  are such that  $\Theta(k) = \tau' \equiv LL$  and by (v-key)  $\Theta \vdash^f k : LL$ . By hypothesis  $n \notin \mathcal{K}_{HC}$ .

Inductive cases:

$(v_1, v_2)$

Let  $\Theta \vdash_g^f (v_1, v_2) : \tau$ . Since pairs are not in the subtyping relation, the judgement comes from (v-pair) implying that  $\tau = (\tau_1, \tau_2)$ ,  $\Theta \vdash_g^f v_i : \tau_i$ . By lemma 4 we know that  $v_i \notin \mathcal{K}_{HC}$ . Thus, inductive hypothesis applies and we obtain  $\Theta \vdash^f v_i : LL$  and, by (v-pair),  $\Theta \vdash^f (v_1, v_2) : (LL, LL) \equiv LL$ .

$\{\{v'\}\}_k$

Let  $\Theta \vdash_g^f \{\{v'\}\}_k : \tau$ . The judgment might derive from (v-sub) and one among (v-enc) and (v-enc-r), meaning  $\Theta(k) = \text{cK}_\delta^\mu(\tau') \kappa$  and  $\Theta \vdash_{g'}^f v' : \tau'$ . By lemma 4 we know that  $v' \notin \mathcal{K}_{HC}$ . Thus, inductive hypothesis applies and we obtain  $\Theta \vdash^f v' : LL$ . By reapplying the very same rule (v-enc) or (v-enc-r) with empty  $g$  and  $\delta_C = L$  we obtain  $\Theta \vdash^f \{\{v'\}\}_k : \text{enc}_{LL} \kappa \equiv LL$ .

$\langle v' \rangle_k$

This case is identical to the previous one.

(ii) Again, by induction on the structure of  $v$ :

Base case:

$n$

By rule (v-name), we have that all names  $n$ , except keys, are such that  $\Theta \vdash^f n : \tau'$ , with  $\mathcal{L}_I(\tau') = H$ . When  $\mathcal{L}_I(\tau') = H$  we also obtain  $\Theta \vdash^f n : \tau$ , by taking  $\tau' = \tau$ . The condition  $k \in \mathcal{K}_{\mathcal{L}(\Theta(k))}$  on  $\Theta$  ensures that low keys  $k \in \mathcal{K}_{LL}$  are such that  $\Theta(k) = \tau' \equiv LL$  and by (v-key)  $\Theta \vdash^f k : LL$ . Thus, low keys can never be typed at level  $\mathcal{L}_I(\tau) \sqsubseteq_I H$ , meaning the  $n \notin \mathcal{K}_{LL}$ . Moreover, by hypothesis  $n \notin \mathcal{K}_{HC}$ . Thus  $n$  cannot be a key.

Inductive cases:

$(v_1, v_2)$

Let  $\Theta \vdash_g^f (v_1, v_2) : \tau$  with  $\mathcal{L}_I(\tau) = H$ . Since pairs are not in the subtyping relation, the judgement comes from (v-pair) implying that  $\tau = (\tau_1, \tau_2)$ ,  $\Theta \vdash_g^f v_i : \tau_i$ . By assumption 1(1), we also have  $\mathcal{L}_I(\tau_1), \mathcal{L}_I(\tau_2) = H$ . Thus, induction gives  $\Theta \vdash^f v_i : \tau_i$ , and from (v-pair) with empty  $g$  we directly obtain  $\Theta \vdash^f (v_1, v_2) : (\tau_1, \tau_2)$ .

$\{\{v_1\}\}_k$

Let  $\Theta \vdash_g^f \{\{v_1\}\}_k : \tau$  with  $\mathcal{L}_I(\tau) = H$ . The judgment might derive from (v-sub) and one among (v-enc) and (v-enc-r), meaning  $\Theta(k) = \text{cK}_\delta^\mu(\tau_1) \kappa$  and  $\Theta \vdash_{g'}^f v_1 : \tau_1$ . By reapplying the very same rules with  $g = \epsilon$  we thus obtain  $\Theta \vdash^f \{\{v_1\}\}_k : \tau$ .

$\langle v \rangle_k$

MACs are never typed at high integrity. □

**Corollary 1** *Let  $H \sqsubseteq_I \mathcal{L}_I(\tau)$ . Then,  $\Theta \vdash_g^f v : \tau$  if and only if  $\Theta \vdash^f v : \tau$ .*

*Proof.* Notice that  $\Theta \vdash^f v : \tau$  denotes  $\Theta \vdash_g^f v : \tau$  with an empty  $g$ . So we only need to prove that  $\Theta \vdash_g^f v : \tau$ , with  $H \sqsubseteq_I \mathcal{L}_I(\tau)$  and non-empty  $g$ , implies  $\Theta \vdash^f v : \tau$ . By  $H \sqsubseteq_I \mathcal{L}_I(\tau)$  we have two easy cases: when  $\mathcal{L}_I(\tau) = L$ , by lemma 5(i) we obtain that  $\Theta \vdash^f v : LL \equiv \tau$ . When  $\mathcal{L}_I(\tau) = H$ , by lemma 5(ii) we directly obtain  $\Theta \vdash^f v : \tau$ .  $\square$

Next lemma states that value subtyping works as expected even in the presence of  $g_\tau$ . Recall that  $g_\tau$  denotes  $\epsilon$  when  $H \sqsubseteq \mathcal{L}_I(\tau)$  and  $g$  otherwise.

**Lemma 6**  *$\Theta \vdash_{g_{\tau'}}^f v : \tau'$  and  $\tau' \leq \tau$  then  $\Theta \vdash_{g_\tau}^f v : \tau$ .*

*Proof.* By (v-sub) we have that  $\Theta \vdash_{g_{\tau'}}^f v : \tau'$  and  $\tau' \leq \tau$  imply  $\Theta \vdash_{g_{\tau'}}^f v : \tau$ . Now we have three cases. If  $H \not\sqsubseteq \mathcal{L}_I(\tau)$  then also  $H \not\sqsubseteq \mathcal{L}_I(\tau')$  and  $g_\tau = g_{\tau'} = g$  which concludes the proof. If  $H \sqsubseteq \mathcal{L}_I(\tau)$  and  $H \sqsubseteq \mathcal{L}_I(\tau')$  then  $g_\tau = g_{\tau'} = \epsilon$  and even this case is concluded. The only interesting case is when  $H \sqsubseteq \mathcal{L}_I(\tau)$  and  $H \not\sqsubseteq \mathcal{L}_I(\tau')$  which implies  $g_\tau = \epsilon$  and  $g_{\tau'} = g$ . Intuitively, this happens when subtyping ‘crosses’ the  $H$  integrity boundary. By corollary 1(ii) we obtain that  $\Theta \vdash_g^f v : \tau$  implies  $\Theta \vdash^f v : \tau$  giving the thesis.  $\square$

**Key-safety of well-formed memories** We now show that memory well-formedness implies key-safety: when observing a memory at  $\delta$  we can only learn keys of level  $\delta$  or below. We first need to formalize the set of values deducible from a set of values  $V$  by only using deducible keys, written  $\text{knows}(V)$ . It is defined as the least set such that

- (1)  $V \subseteq \text{knows}(V)$ ;
- (2) if  $(v_1, v_2) \in \text{knows}(V)$  then  $v_1, v_2 \in \text{knows}(V)$ ;
- (3) if  $k, \{v\}_k \in \text{knows}(V)$  then  $v \in \text{knows}(V)$ .

What is deducible at  $\delta$  from a memory  $M$ , written  $\text{knows}_\delta(M)$ , can be now expressed as:

$$\text{knows}_\delta(M) = \text{knows}(\text{img}(M|_\delta))$$

where  $\text{img}(M|_\delta) = \{M|_\delta(x) \mid x \in \text{dom}(M|_\delta)\}$ . We can now define memory key-safety as follows:

**Definition 7 (Key-safety of memories).** *A memory  $M$  is key-safe iff  $\text{knows}_\delta(M) \cap \mathcal{K} \subseteq \mathcal{K}_{\sqsubseteq \delta}$ .*

In order to prove that well-formed memories are key-safe we need a few lemmas. First, it is trivial to show that  $\text{knows}_\delta(M)$  only contains subvalues of  $\text{img}(M|_\delta)$ , written  $\text{Sub}(M|_\delta) = \cup_{v \in \text{img}(M|_\delta)} \text{Sub}(v)$ .

**Lemma 7**  $\text{knows}_\delta(M) \subseteq \text{Sub}(M|_\delta)$ .

*Proof.* Let  $V = \text{img}(M|_\delta)$ . We proceed by induction on  $|\text{knows}_\delta(M)|$ .

Base case:  $|\text{knows}_\delta(M)| = |V|$ . We have  $\text{knows}_\delta(M) = V \subseteq \cup_{v \in V} \text{Sub}(v) = \text{Sub}(M|_\delta)$ .

Inductive case:  $|\text{knows}_\delta(M)| > |V|$ . We pick  $v \in \text{knows}_\delta(M)$ . If  $v \in V$  we are done as  $V \subseteq \cup_{v \in V} \text{Sub}(v) = \text{Sub}(M|_\delta)$ . If, instead,  $v \notin V$  we remove it from  $\text{knows}_\delta(M)$ , obtaining  $V' = \text{knows}_\delta(M) \setminus \{v\}$ . Doing so we necessarily break condition (2) or (3), otherwise we would have  $\text{knows}_\delta(M)$  is not the least set satisfying those constraints (since  $V'$  is included in it). We now have different subcases depending on which condition we break:

- (2) We have either  $(v, v') \in V'$  or  $(v', v) \in V'$ . Without loss of generality, assume  $(v, v') \in V'$ . By inductive hypothesis we have that  $(v, v') \in \text{Sub}(v'')$  for at least one  $v'' \in V$  which implies  $v \in \text{Sub}(v'') \subseteq \text{Sub}(M|_\delta)$  giving the thesis.

(3) We have that  $k, \{v\}_k$  belong to  $V'$ . By inductive hypothesis we have that  $\{v\}_k \in \text{Sub}(v'')$  for at least one  $v'' \in V$  which implies  $v \in \text{Sub}(v'') \subseteq \text{Sub}(M|_\delta)$  giving the thesis.  $\square$

Next lemma states that values typed at a certain integrity level contains subvalues which are typed at the same integrity level or below (i.e., higher since integrity levels are countervariant).

**Lemma 8** *Let  $\Theta \vdash_g^f v : \tau$  and let  $v' \in \text{Sub}(v)$ . Then,  $\Theta \vdash_{g'}^f v' : \tau'$  and  $\mathcal{L}_I(\tau') \sqsubseteq_I \mathcal{L}_I(\tau)$ .*

*Proof.* Observe that if  $\Theta \vdash_g^f v : \tau$  then  $\Theta \vdash_{g'}^f v' : \tau'$  occurs in the proof of the former judgement since all subvalues are recursively judged.

Thus the only fact we need to prove is that  $\mathcal{L}_I(\tau') \sqsubseteq_I \mathcal{L}_I(\tau)$ . It is sufficient to proceed by induction on the length of the derivation from  $\Theta \vdash_{g'}^f v' : \tau'$  to  $\Theta \vdash_g^f v : \tau$ . For length 0 we have that  $v = v'$  which trivially gives the thesis. For length  $i > 0$ , by an inspection of the rules, we can see that, when  $v$  is not atomic, judgement  $\Theta \vdash_g^f v : \tau$  always depends on  $\Theta \vdash_{g''}^f v'' : \tau''$  such that  $v''$  is a subvalue of  $v$  and, in all cases, we have  $\mathcal{L}_I(\tau'') \sqsubseteq_I \mathcal{L}_I(\tau)$ . Since we also have that  $v' \in \text{Sub}_\delta(v'')$  (in case of pairs we will choose the appropriate subvalue), by inductive hypothesis we obtain  $\mathcal{L}_I(\tau') \sqsubseteq_I \mathcal{L}_I(\tau'')$  from which the thesis.  $\square$

We can now prove that low keys never occurs as subvalues of values which have integrity level at or below  $H$ .

**Lemma 9** *Let  $\Theta \vdash_g^f v : \tau$  and  $\mathcal{L}_I(\tau) \sqsubseteq_I H$ . Then  $\text{Sub}(v) \cap \mathcal{K}_{LL} = \emptyset$ .*

*Proof.* By lemma 8 we know that  $v' \in \text{Sub}(v)$  is such that  $\Theta \vdash_{g'}^f v' : \tau'$  with  $\mathcal{L}_I(\tau') \sqsubseteq_I \mathcal{L}_I(\tau) \sqsubseteq_I H$ . Since low keys can only be typed via (v-key) and (v-sub) as  $\delta_C L$ , we conclude that  $v' \notin \mathcal{K}_{LL}$  from which the thesis.

**Proposition 2 (Key-safety)** *Let  $\Delta \vdash_g^f M$ , then  $M$  is key-safe.*

*Proof.* We want to prove that  $\text{knows}_\delta(M) \cap \mathcal{K} \subseteq \mathcal{K}_{\sqsubseteq \delta}$ . First recall that  $\text{knows}_\delta(M) = \text{knows}(\text{img}(M|_\delta))$  and, by definition,  $M|_\delta$  is the submemory of  $M$  only containing variables whose level is less then or equal  $\delta$ . Let  $v_1, \dots, v_n$  be all the values in  $\text{img}(M|_\delta)$ . From  $\Delta \vdash_g^f M$  we know that  $\Theta \vdash_{g_{\tau_i}}^f v_i : \tau_i$  with  $\mathcal{L}(\tau_i) \sqsubseteq \delta$ , for all  $i = 1 \dots n$ .

If  $LL \not\sqsubseteq \delta$  we necessarily have that  $\mathcal{L}_I(\delta) = H$ . By lemma 9 we know that  $\text{Sub}(v_i) \cap \mathcal{K}_{LL} = \emptyset$ . By lemma 7 we obtain that  $\text{knows}_\delta(M) \cap \mathcal{K}_{LL} \subseteq \bigcup_{i=1}^n \text{Sub}(v_i) \cap \mathcal{K}_{LL} = \emptyset$ .

If  $HC \not\sqsubseteq \delta$  we know that  $v_i \notin \mathcal{K}_{HC}$  since high keys can only be typed at level  $HC$ . By lemma 4 we obtain that  $\text{Sub}(v_i) \cap \mathcal{K}_{HC} = \emptyset$ . Again by lemma 7 we obtain that  $\text{knows}_\delta(M) \cap \mathcal{K}_{HC} \subseteq \bigcup_{i=1}^n \text{Sub}(v_i) \cap \mathcal{K}_{HC} = \emptyset$ .

We have thus proved that  $\text{knows}_\delta(M) \cap \mathcal{K}_{\not\sqsubseteq \delta} = \emptyset$ , where  $\mathcal{K}_{\not\sqsubseteq \delta} = \mathcal{K} \setminus \mathcal{K}_{\sqsubseteq \delta}$ . Now we can write

$$\begin{aligned} \text{knows}_\delta(M) \cap \mathcal{K} &= \\ &= \text{knows}_\delta(M) \cap (\mathcal{K}_{\sqsubseteq \delta} \cup \mathcal{K}_{\not\sqsubseteq \delta}) \\ &= (\text{knows}_\delta(M) \cap \mathcal{K}_{\sqsubseteq \delta}) \cup (\text{knows}_\delta(M) \cap \mathcal{K}_{\not\sqsubseteq \delta}) \\ &\subseteq \mathcal{K}_{\sqsubseteq \delta} \cup \emptyset \\ &= \mathcal{K}_{\sqsubseteq \delta} \end{aligned}$$

giving the thesis.  $\square$

**Evaluation of well-typed expressions** We now prove that well-typed expressions of type  $\tau$  evaluated on well-formed memories, always give a well-formed value of type  $\tau$ . Moreover, randomized ciphertexts are guaranteed to be either identical copies of already existing ones or completely new ones, i.e., with a fresh confounder. Recall, we write  $\text{Sub}(M)$  to denote the set of subvalues of the whole memory  $M$ , i.e.,  $\cup_{v \in \text{img}(M)} \text{Sub}(v)$ .

**Proposition 3** *Let  $\Delta \vdash_g^f M$ ,  $\Delta \vdash e : \tau$  and  $e \downarrow^M v$ . Then*

- (i)  $\Theta \vdash_{g_\tau}^f v : \tau$
- (ii) if  $\{\{v', r\}\}_k \in \text{Sub}(v)$ , with  $k \in \mathcal{K}_{HC}^R$ , then either  $\{\{v', r\}\}_k \in \text{Sub}(M)$  or  $r$  has been extracted fresh, noted  $r \leftarrow C$ , during the evaluation of  $e$ .

*Proof.* By induction on the structure of  $e$ .

$x$

By  $\Delta \vdash x : \tau$  we have that  $\Delta(x) = \tau'$  with  $\tau' \leq \tau$ . Since  $x \downarrow^M M(x)$  by (i) of definition 5 we directly obtain that  $\Theta \vdash_{g_{\tau'}}^f v : \tau'$  and by lemma 6 we obtain thesis (i). Thesis (ii) is easily obtained by observing that  $\text{Sub}(v) \subseteq \text{Sub}(M)$ .

$e_1 \text{ op } e_2$

By  $\Delta \vdash e_1 \text{ op } e_2 : \delta$  we have  $\mathcal{L}_I(\delta) \neq [D : \tilde{D}]$ . Here we do not even need induction: by assumption 2(1) we have that  $e_1 \text{ op } e_2 \downarrow^M n$  with  $n \notin \mathcal{K}$ . Thesis (i) is a direct consequence of (v-name), thesis (ii) holds since  $\text{Sub}(v) = \{n\}$ .

$\text{pair}(e_1, e_2)$

By  $\Delta \vdash \text{pair}(e_1, e_2) : (\tau_1, \tau_2)$  we have  $\Delta \vdash e_i : \tau'_i$  with  $\tau'_i \leq \tau_i$ . Let  $e_i \downarrow^M v_i$  and notice that  $\text{pair}(e_1, e_2) \downarrow^M (v_1, v_2)$ . By induction and lemma 6 we directly obtain that  $\Theta \vdash_{g_{\tau_i}}^f v_i : \tau_i$ . We have two cases: if  $H \sqsubseteq_I \mathcal{L}_I((\tau_1, \tau_2))$ , by assumption 1, we know that  $H \sqsubseteq_I \mathcal{L}_I(\tau_1) = \mathcal{L}_I(\tau_2)$  thus  $g_{\tau_1} = g_{\tau_2} = g_\tau = \epsilon$  and we obtain the thesis by (v-pair). If  $H \not\sqsubseteq_I \mathcal{L}_I((\tau_1, \tau_2))$  we also have  $H \not\sqsubseteq_I \mathcal{L}_I(\tau_1), \mathcal{L}_I(\tau_2)$ , since, by definition,  $\mathcal{L}((\tau_1, \tau_2)) = \mathcal{L}(\tau_1) \sqcup \mathcal{L}(\tau_2)$ . Thus  $g_{\tau_1} = g_{\tau_2} = g_\tau = g$  and, again, by (v-pair) we obtain thesis (i). Now observe that  $\text{Sub}(v_1, v_2) = (v_1, v_2) \cup \text{Sub}(v_1) \cup \text{Sub}(v_2)$ , thus every value of the form  $\{\{v', r\}\}_k \in \text{Sub}(v_1, v_2)$  occurs either in  $\text{Sub}(v_1)$  or  $\text{Sub}(v_2)$ . Since, by induction, (ii) holds on  $v_1$  and  $v_2$ , it also holds for  $(v_1, v_2)$ .

$\text{fst}(e_1), \text{snd}(e_1)$

We show the proof for  $\text{fst}(e_1)$  as the one for  $\text{snd}(e_1)$  is identical. By  $\Delta \vdash \text{fst}(e_1) : \tau$  we have  $\Delta \vdash e_1 : \tau'' = (\tau', \tau'_2)$  with  $\tau' \leq \tau$ . Let  $e_1 \downarrow^M v_1$ . By induction we know that  $\Theta \vdash_{g_{\tau''}}^f v_1 : \tau''$ . Now we have three cases: if  $v_1 = (v, v_2)$  and  $H \sqsubseteq_I \mathcal{L}_I(\tau'')$ , as for the above case, we have that  $H \sqsubseteq_I \mathcal{L}_I(\tau')$ , thus  $g_{\tau''} = g_{\tau'} = \epsilon$ , implying  $\Theta \vdash_{g_{\tau'}}^f v : \tau'$  and, by lemma 6 we obtain  $\Theta \vdash_{g_\tau}^f v : \tau$ . The case  $v_1 = (v, v_2)$  and  $H \not\sqsubseteq_I \mathcal{L}_I(\tau'')$  implies also  $H \not\sqsubseteq_I \mathcal{L}_I(\tau')$ , giving  $g_{\tau''} = g_{\tau'} = g$ , and we proceed as above. The last case is when  $v$  is not a pair. Here we have that  $\text{fst}(e)$  returns  $\perp$ . The only way a non-pair can be typed as a pair is when  $\tau'' \equiv LL \equiv (LL, LL)$ . By (v-name) we obtain  $\Theta \vdash^f \perp : \tau \equiv LL$ . Thesis (ii) is trivial in the case the result is  $\perp$ . If, instead, the result is  $v$  it is sufficient to observe that  $\text{Sub}(v) \subseteq \text{Sub}(v_1)$  and that, by induction, (ii) holds for  $v_1$ .

$\text{enc}_x^\mu(e_1)$

In this case  $\Delta \vdash e : \text{enc}_\delta \kappa$ . Let  $e_1 \downarrow^M v_1$  and  $x \downarrow^M v_2$ . The proof follows by cases on the key type.

If  $\Delta(x) = \text{cK}_{LL}(\tau) \kappa$  then the expression has been typed by (enc). It follows that  $\delta = \delta_C L$  and  $\Delta \vdash e_1 : \tau'$  with  $\tau' \leq \tau$ . We have two cases: if  $v_2 \notin \mathcal{K}$  we have  $e \downarrow^M \perp$  and by (v-name) we directly obtain  $\Theta \vdash^f \perp : \delta_C L \equiv \text{enc}_\delta \kappa$ . If, instead,  $v_2 = k \in \mathcal{K}$ , we have  $e \downarrow^M \{\{v_1\}\}_k$ . By induction hypothesis,  $\Theta \vdash_{g_{\tau'}}^f v_1 : \tau'$  and  $\Theta(v_2) = \text{cK}_{LL}(\tau) \kappa \equiv \text{cK}_{LL}(LL) \kappa$ ; by lemma 5(ii) we obtain  $\Theta \vdash^f v_1 : LL$  and by (v-enc) we directly obtain that  $\Theta \vdash^f \{\{v_1\}\}_k : \delta_C L \equiv \text{enc}_\delta \kappa$ .

If  $\Delta(x) = \text{cK}_{HC}(\tau) \kappa$  then the expression can be typed either by (enc) or (enc-d). In both cases  $\Delta \vdash e_1 : \tau'$  with  $\tau' \leq \tau$  and  $\mathcal{L}_I(\delta) = C \sqcup \mathcal{L}_I(\tau)$ . So by induction and lemma 6,  $\Theta \vdash_{g_\tau}^f v_1 : \tau$  and  $\Theta(k) = \text{cK}_{HC}(\tau) \kappa$ . Notice that, since  $\mathcal{L}_I(\delta) = C \sqcup \mathcal{L}_I(\tau)$ , we have  $g_{\text{enc}_\delta} \kappa = g_\tau$ . We can now apply (v-enc) to get  $\Theta \vdash_{g_\tau}^f \{v_1\}_k : \text{enc}_\delta \kappa$ . Notice that (v-enc) does not constraint the confidentiality level of  $\delta$ , which is useful to fulfill both (enc) and (enc-d) requirements.

The case  $\Delta(x) = \text{cK}_{HC}^R(\tau) \kappa$  is exactly as the above one.

Thesis (ii) is trivial in the case the result is  $\perp$ . If, instead, the result is  $\{v_1\}_k$ , with  $\Theta(k) \neq \text{cK}_{HC}^R(\tau) \kappa$ , we observe that  $\text{Sub}(\{v_1\}_k) = \{v_1\}_k \cup \text{Sub}(v_1)$  and, since by induction we know that (ii) holds on  $\text{Sub}(v_1)$ , we obtain the thesis. The only interesting case is when  $\Theta(k) = \text{cK}_{HC}^R(\tau) \kappa$  which gives  $\{v_1, r\}_k$  and consequently  $\text{Sub}(\{v_1, r\}_k) = \{v_1, r\}_k \cup \text{Sub}(v_1) \cup r$ . Since  $r \leftarrow C$  during the evaluation of this encryption and by induction we know that (ii) holds on  $v_1$ , we obtain it also holds for  $\{v_1, r\}_k$ .

$\text{dec}_x^\mu(e_1)$

We have  $\Delta \vdash \text{dec}_x(e_1) : \tau$  and  $\Delta \vdash e_1 : \text{enc}_\delta \kappa$ . Let  $e_1 \downarrow^M v_1$  and  $x \downarrow^M v_2$ . By induction hypothesis,  $\Theta \vdash_{g_{\tau''}}^f v_1 : \tau''$  with  $\tau'' = \text{enc}_\delta \kappa$ . The proof follows by cases on the key type.

If  $\Delta(x) = \text{cK}_{LL}(\tau') \kappa$  then the expression has been typed with  $\tau = LL \sqcup \delta$  by (dec). It follows that  $\tau = \delta_C L$ . We have two cases: if  $v_1 \neq \{v\}_{v_2}$  or  $v_2 \notin \mathcal{K}$  we have  $e \downarrow^M \perp$  and by (v-name) we directly obtain  $\Theta \vdash^f \perp : \delta_C L \equiv \tau$ . If, instead,  $v_1 = \{v\}_k$  and  $v_2 = k \in \mathcal{K}$ , we have  $e \downarrow^M v$ . By induction hypothesis,  $\Theta \vdash^f v_1 : \text{enc}_\delta \kappa \equiv \delta_C L$ . This implies, by (v-enc), that  $\Theta \vdash^f v : \tau'$  and, by lemma 5(i) we obtain  $\Theta \vdash^f v : LL \leq \delta_C L$ .

If  $\Delta(x) = \text{cK}_{HC}(\tau') \kappa$ , since high keys can only be typed via (v-key), we know that  $v_2 = k$  and  $\Theta(k) = \text{cK}_{HC}(\tau') \kappa$ . The expression can be typed either by (dec) or (dec- $\mu$ ).

If the expression has been typed with  $\tau = HC \sqcup \delta$  by (dec) we have two subcases: if  $v_1 \neq \{v\}_k$  we have  $e \downarrow^M \perp$ . The only way  $v_1$  can have type  $\text{enc}_\delta \kappa$  without being a ciphertext is when  $\mathcal{L}_I(\delta) = L$  thus  $\mathcal{L}_I(HC \sqcup \delta) = L$ . By (v-name) we directly obtain  $\Theta \vdash^f \perp : \tau$ . If, instead,  $v_1 = \{v\}_k$ , we have  $e \downarrow^M v$ . By induction hypothesis,  $\Theta \vdash_{g_{\tau''}}^f v_1 : \tau''$  with  $\tau'' = \text{enc}_\delta \kappa$  and by (v-enc) we know that  $\Theta \vdash_{g'}^f v : \tau'$ , with  $C \sqcup \mathcal{L}_I(\tau') = \mathcal{L}_I(\tau'')$ . This implies  $H \sqsubseteq_I \mathcal{L}_I(\tau'')$  iff  $H \sqsubseteq_I \mathcal{L}_I(\tau')$  which means  $g_{\tau''} = g_{\tau'}$ . Notice also that  $g'$  can be different from  $g_{\tau''}$  only when  $g_{\tau''} = \epsilon$  but this happens when  $H \sqsubseteq_I \mathcal{L}_I(\tau')$  and by corollary 1 we obtain that  $\Theta \vdash_{g'}^f v : \tau'$  implies  $\Theta \vdash^f v : \tau'$ . Thus  $\Theta \vdash_{g_{\tau''}}^f v : \tau'$  and since  $g_{\tau''} = g_{\tau'}$ ,  $\Theta \vdash_{g_{\tau'}}^f v : \tau'$ . Now it can be easily shown that  $\tau = H \delta_I$  with  $\delta_I = C \sqcup \mathcal{L}_I(\tau')$ . In fact,  $\tau = HC \sqcup \delta = HC \sqcup \mathcal{L}_I(\delta) = HC \sqcup \mathcal{L}_I(\tau'') = HC \sqcup \mathcal{L}_I(\tau')$ . Thus  $\tau' \leq \tau$  and by lemma 6 we obtain  $\Theta \vdash_{g_\tau}^f v : \tau$ .

If the expression has been typed with  $\tau = \tau'$  by (dec- $\mu$ ) two subcases must be considered: if  $v_1 \neq \{v\}_k$  we have  $e \downarrow^M \perp$ . The only way  $v_1$  can have type  $\text{enc}_\delta \kappa$  without being a ciphertext is when  $\mathcal{L}_I(\delta) = \mathcal{L}_I(HC \sqcup \mathcal{L}_I(\tau')) = L$ , i.e.,  $\mathcal{L}_I(\tau') = L$ . By (v-name) we directly obtain  $\Theta \vdash^f \perp : \delta_C L \equiv \tau'$ . Otherwise  $v_1 = \{v\}_k$  and we have  $e \downarrow^M v$ . If  $\mathcal{L}_I(\tau') = L$  the proof follows directly by (v-name). If, instead,  $\mathcal{L}_I(\tau') \sqsubseteq_I H$  it follows  $\mathcal{L}_I(\delta) \sqsubseteq_I H$  and  $v_1$  must have been typed via (v-enc). The fact  $\Theta \vdash_{g_{\tau'}}^f v : \tau'$  is obtained exactly as done for the previous case.

The case  $\Delta(x) = \text{cK}_{HC}^R(\tau') \kappa$  is exactly as the above one except for  $v_1 = \{v, r\}_k$  which is typed via (v-enc-r) in place of (v-enc).

Thesis (ii) is trivial in the case the result is  $\perp$ . If, instead, the result is  $v$  it is sufficient to observe that  $\text{Sub}(v) \subseteq \text{Sub}(\{v\}_k)$  and that, by induction, (ii) holds for  $\{v\}_k$ .

$\text{mac}_k(e_1)$

Proof is analogous to the one for encryption: The expression has been typed by rule (mac) with

$\Delta(x) = \text{mK}_{\delta_C \delta_I}(\tau)$ . It follows that  $\Delta \vdash e_1 : \tau'$  with  $\tau' \leq \tau$ . We have two cases: if  $v_2 \notin \mathcal{K}$  we have  $\delta_I = L$  and  $e \downarrow^M \perp$  and by (l-name) we directly obtain  $\Theta \vdash^f \perp : LL$ . If, instead,  $v_2 = k \in \mathcal{K}$ , we have  $e \downarrow^M \langle v_1 \rangle_k$ . If the key is of level  $LL$ , by induction hypothesis,  $\Theta \vdash_{g_{\tau'}}^f v_1 : \tau'$  and  $\Theta(v_2) = \text{mK}_{LL}(\tau) \equiv \text{mK}_{LL}(LL)$  and by lemma 5(i) we obtain  $\Theta \vdash^f v_1 : LL$  which by (v-mac) gives  $\Theta \vdash^f \langle v_1 \rangle_k : LL$ . If instead the key is of level  $HC$ , by induction and lemma 6,  $\Theta \vdash_{g_{\tau}}^f v_1 : \tau$  and  $\Theta(k) = \text{mK}_{HC}(\tau)$ . Thus, by (v-mac)  $\Theta \vdash_{g_{\tau}}^f \langle v_1 \rangle_k : \delta_C L$  and by lemma 5(i)  $\Theta \vdash^f \langle v_1 \rangle_k : LL \sqsubseteq \delta_C L$ .

Thesis (ii) is trivial in the case the result is  $\perp$ . If, instead, the result is  $\langle v_1 \rangle_k$  we observe that  $\text{Sub}(\langle v_1 \rangle_k) = \langle v_1 \rangle_k \cup \text{Sub}(v_1)$  and, since by induction we know that (ii) holds on  $\text{Sub}(v_1)$ , we obtain the thesis.  $\square$

It is useful to prove that well-formedness of values of type  $\delta_C \delta_I$  does not constrain in any way the confidentiality level.

**Lemma 10** *If  $\Theta \vdash_g^f v : H\delta_I$  then  $\Theta \vdash_g^f v : L\delta_I$ .*

*Proof.* By induction on the structure of  $v$ .

Base case:

$n$

We know that  $\Theta \vdash_g^f n : \delta'_C \delta'_I$  with  $\delta'_C \delta'_I \leq H\delta_I$ . Notice that  $n \notin \mathcal{K}_{HC}$  as high keys can never be typed as  $H\delta_I$ . Low keys are only typed  $LL$  and  $HL$  which directly gives the thesis. For  $n \in \mathcal{K}$  we observe that rules (v-name), (v-ir) and (v-dd) do not restrict the confidentiality level. By reapplying the same rule as above with  $\delta'_C = L$  we thus obtain  $\Theta \vdash_g^f n : L\delta'_I$ . From  $L\delta'_I \leq L\delta_I$  we obtain the thesis.

Inductive cases:

$(v_1, v_2)$

The only case a pair can be typed  $H\delta_I$  is when  $\delta_I = L$ , since  $HL \equiv (HL, HL)$ . Thus, by assumption 1(1) and (v-pair), we know that  $\Theta \vdash_g^f v_i : HL$ . By induction we get  $\Theta \vdash_g^f v_i : LL$  and by (v-pair) directly  $\Theta \vdash_g^f v : (LL, LL) \equiv LL$ .

$\{v\}_k$

We know that  $\Theta \vdash_g^f \{v\}_k : \text{enc}_{\delta'_C \delta'_I} \kappa$  with  $\text{enc}_{\delta'_C \delta'_I} \kappa \leq \delta'_C L \leq HL$ . The first judgment is either (v-enc) or (v-enc-r): we reapply it with  $\delta'_C = L$  getting  $\Theta \vdash_g^f \{v\}_k : \text{enc}_{L\delta'_I} \kappa \leq LL$ .

$\langle v \rangle_k$

This case proved exactly as above.  $\square$

**Theorem 3** *If  $\Delta, pc \vdash c$ ,  $\Delta \vdash_g^f M$  and  $\langle M, c \rangle \Rightarrow M'$  then  $\Delta \vdash_g^f M'$*

*Proof.* By induction on the derivation length of  $\langle M, c \rangle \Rightarrow M'$ .

Base case, length 1:

[skip], [whilef]

Since  $\langle M, \text{skip} \rangle \Rightarrow M$  and  $\Delta \vdash_g^f M$  we have nothing to prove. The same holds for while loops when the guard is false, since memory is untouched.

**[assign]**

We have  $\langle M, x := e \rangle \Rightarrow M[x \mapsto v] = M'$  given that  $e \downarrow^M v$ . We now prove condition (i) of definition 5. Since the only change from  $M$  to  $M'$  is the value of  $x$ , which is set to  $v$ , we just need to check that  $\Delta(x) = \tau$  implies  $\Theta \vdash_{g\tau}^f v : \tau$ . For (assign) this is directly achieved via proposition 3(i), since the rule requests  $\Delta \vdash e : \tau$ . For (declassify) we know that  $\Delta(x) = \delta_C H$  and  $\Delta \vdash e : \delta'_C H$  and, by proposition 3(i),  $\Theta \vdash^f v : \delta'_C H$ . By lemma 10 we have that  $\Theta \vdash^f v : LH \leq \delta_C H$ .

Notice that  $\text{Sub}(M') \subseteq \text{Sub}(M) \cup \text{Sub}(v)$ . Thus if we prove condition (ii) of definition 5 on  $\text{Sub}(M) \cup \text{Sub}(v)$  we obtain that it holds even for  $M'$ . We consider three different cases: Let  $\Theta(k) = \text{cK}_{HC}^R(\tau) \kappa$  and  $\{v, r\}_k, \{v', r'\}_k \in \text{Sub}(M)$ . Since  $\Delta \vdash_g^f M$  we directly know that  $r \neq r'$ . If, instead,  $\{v, r\}_k, \{v', r'\}_k \in \text{Sub}(v)$ , by proposition 3(ii) we know that either  $\{v', r'\}_k \in \text{Sub}(M)$ , which leads to the previous case, or  $r$  has been extracted fresh, noted  $r \leftarrow C$ , during the evaluation of  $e$ , which directly gives  $r \neq r'$ . Finally, if  $\{v, r\}_k, \{v', r'\}_k \in \text{Sub}(v)$  we have that either both values also appear in  $M$ , which one more time leads to the first case, or one of the counfounder has been extracted fresh during the evaluation of  $e$  which directly gives  $r \neq r'$ .

Inductive case, length  $n$ . We consider the last rule applied:

**[seq]**

We have  $\langle M, c_1; c_2 \rangle \Rightarrow M'$  since  $\langle M, c_1 \rangle \Rightarrow M''$  and also  $\langle M'', c_2 \rangle \Rightarrow M'$ . By rule (seq) we have that  $\Delta, pc \vdash c_i$ . By induction on first command we get  $\Delta \vdash_g^f M''$ , then by induction on the second one  $\Delta \vdash_g^f M'$ .

**[ift], [iff]**

If the command is typed with rule (if) we proceed by induction on the executed branch and we directly get  $\Delta \vdash_g^f M'$ . The interesting case is when the command is typed via (if-MAC). If the mac check is false the command do not terminate and we have nothing to prove. We analyze the case when  $\text{mac}_x(z, e) = e'$  is true, i.e., we have:

$$\langle M, \text{if } \text{mac}_x(z, e) = e' \text{ then } (y := e; c_1) \text{ else } c_2; \perp_{\text{MAC}} \rangle \Rightarrow M'$$

because of  $\langle M, y := e \rangle \Rightarrow M''$  and  $\langle M'', c_1 \rangle \Rightarrow M'$ . By rule (if-MAC) we know that  $\Delta(x) = \text{mK}_{HC}(L[D], \tau)$ . By  $\Delta \vdash_g^f M$  we know that  $M(x)$  has type  $\text{mK}_{HC}(L[D], \tau)$ , i.e.,  $M(x) = k$  and  $\Theta(k) = \text{mK}_{HC}(L[D], \tau)$  thus  $\text{mac}_x(z, e) \downarrow^M \langle v, v' \rangle_k$  and  $e' \downarrow^M \langle v, v' \rangle_k$ . We know that  $\Delta \vdash e' : LL$ , thus by proposition 3 we know that  $\Theta \vdash_{g\tau}^f \langle v, v' \rangle_k : LL$ . By (v-mac) and (v-pair) we are guaranteed that  $\Theta \vdash_{g'}^f v : L[D]$  and  $\Theta \vdash_{g'}^f v' : \tau$ . By rule (if-MAC) we also know that  $\Delta \vdash z : L[D]$  and from  $\Delta \vdash_g^f M$  we have  $\Theta \vdash_g^f v : L[D]$ , with the expected  $g$  since the type integrity is lower than  $H$ . We obtain that  $g(D) = g'(D)$ . From  $\text{IRs}(L[D], \tau) = \{D\}$  and  $\text{Closed}(L[D], \tau)$  we easily obtain that  $\mathcal{L}(\tau) = \delta_C[\bullet : D]$  and also that  $\Theta \vdash_{g'}^f v' : \tau$  implies  $\Theta \vdash_g^f v' : \tau$ , since by lemma 8 nothing above  $[\bullet : D]$  will ever appear as subvalue of  $v'$  and  $g$  and  $g'$  are the same on domain  $D$ . This is what we need to prove to check the well-formedness of the memory obtained after the assignment (item (ii) of well-formedness is dealt with as in the case [assign]). Thus  $\Delta \vdash_g^f M'$ . By induction on the derivation for  $c_1$  we have that also  $\Delta \vdash_g^f M'$ .

**[while]**

This case is analogous to [seq]. Execution  $\langle M, \text{while } e \text{ do } c \rangle \Rightarrow M'$  derives from  $\langle M, c \rangle \Rightarrow M''$  and  $\langle M'', \text{while } e \text{ do } c \rangle \Rightarrow M'$ . Rule (while) ensures that  $\Delta, pc' \vdash c$ . Thus, by induction, we get  $\Delta \vdash_g^f M''$  and consequently  $\Delta \vdash_g^f M'$ . Notice that we can apply induction on  $\langle M'', \text{while } e \text{ do } c \rangle \Rightarrow M'$  as, even if the command is the same as the one we are analysing, the derivation length is  $n - 1$ .  $\square$

**Indistinguishability of well-formed memories** We now extend the equivalence notion on memories, so to respect dependent domains. Notice that observation point is placed in the four-point lattice (i.e., we write  $\ell$  and not  $\delta$ ), i.e., variables below integrity  $H$  are always observed as a whole, at the appropriate confidentiality level.

We say that a substitution  $\rho$  *respects*  $\Theta$  if  $\rho(\square_{\{v_1\}_k}) = \square_{\{v_2\}_k}$  and  $\Theta(k) = K_{\delta'}(\tau) \kappa$  imply  $\exists g'$  such that  $\Theta \vdash_{g'}^{f_i} v_i : \tau$ . Moreover,  $g' = \epsilon$  iff  $H \sqsubseteq_I \mathcal{L}_I(\tau)$ .

**Definition 8 (WF-Indistinguishability).**  $M_1$  and  $M_2$  are WF-indistinguishable at level  $\ell$ , written  $M_1 \approx_{\ell}^{\Delta} M_2$ , if

1.  $\Delta \vdash_{g'}^{f_i} M_i$ ;
2.  $\text{p}_{\ell}(M_1) = \text{p}_{\ell}(M_2) \rho$  (meaning that  $M_1 \approx_{\ell} M_2$ );
3.  $\rho$  *respects*  $\Theta$ .

Intuitively, two memories are WF-indistinguishable if (1) they are well-formed; (2) they are indistinguishable; (3) encrypted hidden values mapped by  $\rho$  can be typed with the same  $g'$  (at the expected type  $\tau$ ).

We now need a few lemmas that characterize  $\text{CloseDD}(\tau)$  types, defined in section A.

**Lemma 11**  $\text{DD}(\tau)$ , with  $\tau \neq K_{HC}(\tau') \kappa$ , implies  $\mathcal{L}_I(\tau) \sqsubseteq_I H$ .

*Proof.* Notice that stating  $\mathcal{L}_I(\tau) \sqsubseteq_I H$  is equivalent to  $H \not\sqsubseteq_I \mathcal{L}_I(\tau)$  since every integrity level is comparable with  $H$ . We proceed by induction on the structure of  $\tau$ :

Base case:

$\delta_C \delta_I$

Trivial since  $\text{DD}(\delta_C \delta_I)$  directly requires that  $C \not\sqsubseteq_I \delta_I$  and  $C \sqsubseteq_I H \sqsubseteq L$ . Thus  $\delta_I \neq L, H$ , i.e.,  $\mathcal{L}_I(\tau) \sqsubseteq_I H$ .

Inductive case:

$K_{\delta'}(\tau') \kappa$

We have assumed  $\tau \neq K_{HC}(\tau) \kappa$  thus  $\delta' = LL$ , which is anyway forbidden as  $C \sqsubseteq_I L$ . Thus this case never occurs.

$\text{enc}_{\delta'} \kappa$

Notice that if  $\kappa$  refers to a low key we have  $LL$  in  $\text{LT}_{\Delta}(\text{enc}_{\delta'} \kappa)$ , making  $\text{DD}(\tau)$  false. Thus we only consider the case in which  $\kappa$  refers to a high key  $\text{cK}_{HC}^{\mu}(\tau') \kappa$ . By assumption 1(4) we have  $\mathcal{L}_I(\delta') = C \sqcup \mathcal{L}_I(\tau')$ . We also have  $\text{DD}(\text{enc}_{\delta'} \kappa)$  iff  $\text{DD}(\tau')$  and  $\tau'$  cannot be a high key by assumption 1(3). Thus, by induction we get  $\mathcal{L}_I(\tau') \sqsubseteq_I H$  which implies  $\mathcal{L}_I(\delta') = \mathcal{L}_I(\text{enc}_{\delta'} \kappa) \sqsubseteq_I H$ .

$(\tau_1, \tau_2)$

We have  $\text{DD}(\tau)$  iff  $\text{DD}(\tau_1)$  and  $\text{DD}(\tau_2)$  and by assumption 1(3)  $\tau_i$  cannot be high level key types. Thus by induction  $\mathcal{L}_I(\tau_i) \sqsubseteq_I H$  and so  $\mathcal{L}_I(\tau) \sqsubseteq_I H$ .  $\square$

We let  $\text{gcore}_{\tau} = (\text{IRs}(\tau) \cup \text{Dep}(\tau)) \times \text{Val}$ . When we intersect a function  $g$  with  $\text{gcore}_{\tau}$  we get its restriction on the set of domains  $\text{IRs}(\tau) \cup \text{Dep}(\tau)$ . Intuitively, these are the domains which are needed to type a value as  $\tau$ , since they are all the integrity representatives plus all the domains which type  $\tau$  depends on. Next lemma formalizes this fact. Moreover it proves that any other  $g'$  which includes the above mentioned restriction of  $g$ , can be used in place of  $g$  to type the value.

**Lemma 12** Let  $\Theta \vdash_g^f v : \tau$ ,  $g \neq \epsilon$ ,  $\text{DD}(\tau)$ . Then

- (i)  $\text{dom}(g) \supseteq \text{IRs}(\tau) \cup \text{Dep}(\tau)$
- (ii)  $g' \supseteq g \cap \text{gcore}_\tau$  implies  $\Theta \vdash_{g'}^f v : \tau$ .

*Proof.* By induction on the structure of  $v$ :

Base case:

$n$

First notice that types  $\delta_C[D]$  and  $\delta_C[D : \{D_1, \dots, D_m\}]$  have no subtypes. The only rules where the value of  $g$  matters are (v-ir) and (v-dd) which respectively gives judgements  $\Theta \vdash_g^f n : \delta_C[D]$  and  $\Theta \vdash_g^f n : \delta_C[D : \{D_1, \dots, D_m\}]$ . In the former case we have  $D \in \text{IRs}(\delta_C[D]) = \{D\}$ . We know that  $g(D) \downarrow$  and  $n = g(D)$ . Since  $\text{Dep}(\delta_C[D]) = \emptyset$  we directly have (i). In order to prove (ii), by lemma hypothesis we get  $g'(D) \downarrow$ ,  $g'(D) = g(D)$  and thus  $n = g'(D)$  which implies  $\Theta \vdash_{g'}^f n : \delta_C[D]$ . In the latter case we have that  $D_1, \dots, D_m \in \text{Dep}(\delta_C[D : \{D_1, \dots, D_m\}]) = \{D_1, \dots, D_m\}$ . We additionally know that  $g(D_i) \downarrow$  and  $n = f_{D:D_1, \dots, D_m}(g(D_1), \dots, g(D_m))$ . By lemma hypothesis we thus get  $g'(D_i) \downarrow$ ,  $g'(D_i) = g(D_i)$  thus  $n = f_{D:D_1, \dots, D_m}(g'(D_1), \dots, g'(D_m))$  which implies  $\Theta \vdash_{g'}^f n : \delta_C[D : \{D_1, \dots, D_m\}]$ , since  $f$  is the same function.

Inductive case:

$(v_1, v_2)$

We have  $\Theta \vdash_g^f v : \tau = (\tau_1, \tau_2)$  because of  $\Theta \vdash_g^f v_i : \tau_i$  (pairs are not in the subtype relation). We have that  $\text{DD}(\tau)$  iff  $\text{DD}(\tau_1)$  and  $\text{DD}(\tau_2)$ ; moreover,  $\text{IRs}(\tau) \cup \text{Dep}(\tau) = \cup_i \text{IRs}(\tau_i) \cup \text{Dep}(\tau_i)$ . Thus,  $g' \supseteq g \cap (\text{IRs}(\tau_i) \cup \text{Dep}(\tau_i)) \times \text{Val}$  meaning that, by induction, we get  $\text{dom}(g) \supseteq \text{IRs}(\tau_i) \cup \text{Dep}(\tau_i)$  thus  $\text{dom}(g) \supseteq \text{IRs}(\tau) \cup \text{Dep}(\tau)$ , and  $\Theta \vdash_{g'}^f v_i : \tau_i$  which, from (v-pair), trivially gives thesis (ii).

$\{v'\}_k$

We have  $\Theta(k) = \text{K}_\delta(\tau') \kappa$  and  $\Theta \vdash_g^f \{v'\}_k : \tau$  because of one of the (v-enc) rules. Moreover notice that type  $\text{enc}_\delta \kappa$  derived from those rules is always such that  $C \sqsubseteq \mathcal{L}_I(\delta)$ . Thus by  $\text{DD}(\tau)$  we know that  $\tau = \text{enc}_\delta \kappa$ ; in fact subtyping would give  $\text{enc}_\delta \kappa \leq \delta'$  with  $C \sqsubseteq \mathcal{L}_I(\delta') = L$  forbidden by  $\text{DD}(\tau)$ . We consider (v-enc): since by hypothesis  $g \neq \epsilon$  we have that  $\Theta \vdash_g^f v' : \tau'$ . Now we have  $\text{DD}(\tau)$  iff  $\text{DD}(\tau')$ . Notice, in fact, that  $\text{LT}_\Delta(\text{enc}_\delta \kappa) = \text{LT}_\Delta(\tau')$  where  $\tau'$  is the type transported by the unique key relative to label  $\kappa$ . For the same reason,  $\text{IRs}(\tau) \cup \text{Dep}(\tau) = \text{IRs}(\tau') \cup \text{Dep}(\tau')$ . By induction, we thus get  $\text{dom}(g) \supseteq \text{IRs}(\tau') \cup \text{Dep}(\tau') = \text{IRs}(\tau) \cup \text{Dep}(\tau)$  and  $\Theta \vdash_{g'}^f v' : \tau'$  and, again by (v-enc), we get  $\Theta \vdash_{g'}^f \{v'\}_k : \tau$ . Rule (v-enc-r) is analyzed in the very same way apart from the fact  $v' = (v'', r)$ .

$\langle v' \rangle_k$

This case is not possible as MACs can never be typed at a level  $\delta_C \delta_I$  such that  $C \not\sqsubseteq_I \delta_I$ , as required by  $\text{DD}(\tau)$ .  $\square$

We now prove that when  $v$  is typed  $\tau$  under  $g$  and  $g'$ , then  $g$  and  $g'$  are the same when restricted to the set of integrity representatives of  $\tau$ . We let  $\text{gcoreIR}_\tau = \text{IRs}(\tau) \times \text{Val}$ .

**Lemma 13** *Let  $\Theta \vdash_g^f v : \tau$ ,  $\Theta \vdash_{g'}^f v : \tau$ ,  $g, g' \neq \epsilon$ ,  $\text{DD}(\tau)$ , with  $\tau \neq \text{K}_{HC}(\tau') \kappa$ . Then  $g \cap \text{gcoreIR}_\tau = g' \cap \text{gcoreIR}_\tau$ .*

*Proof.* We proceed by induction on the structure of  $\tau$ :

Base case:

$\delta_C \delta_I$

Since  $DD(\delta_C \delta_I)$  requires that  $C \not\sqsubseteq_I \delta_I$  the only possible types are  $\delta_C[D]$  and  $\delta_C[D : \{D_1, \dots, D_m\}]$ . The only one which has a non empty  $IRs(\tau)$  is the former. In fact,  $IRs(\delta_C[D]) = \{D\}$ , and it can only type atomic names  $n$ . Thus  $v = n$  with  $n = g(D) = g'(D)$ , from which the thesis.

Inductive case:

$K_{\delta'}(\tau') \kappa$

We have assumed  $\tau \neq K_{HC}(\tau) \kappa$  thus  $\delta' = LL$ , which is anyway forbidden as  $C \sqsubseteq_I L$ . Thus this case never occurs.

$enc_{\delta'} \kappa$

Notice that if  $\kappa$  refers to a low key we have  $LL$  in  $LT_{\Delta}(enc_{\delta'} \kappa)$ , making  $DD(\tau)$  false. Thus we only consider the case in which  $\kappa$  refers to a high key  $cK_{HC}^{\mu}(\tau') \kappa$ . We also have  $IRs(enc_{\delta'} \kappa) = IRs(\tau')$  and  $DD(\tau)$  iff  $DD(\tau')$ . Since, by (v-enc),  $v = \{v'\}_k$  and  $\Theta \vdash_g^f v' : \tau'$ ,  $\Theta \vdash_{g'}^f v' : \tau'$ , by induction we get  $g \cap gcorelR_{\tau} = g' \cap gcorelR_{\tau'}$ , which gives the thesis. The case for (v-enc-r) is done analogously.

$(\tau_1, \tau_2)$

We have  $IRs((\tau_1, \tau_2)) = \cup_i IRs(\tau_i)$ . We have that  $DD(\tau)$  iff  $DD(\tau_1)$  and  $DD(\tau_2)$ . By rule (v-pair) we have that  $\Theta \vdash_g^f v_i : \tau'$ ,  $\Theta \vdash_{g'}^f v_i : \tau'$  and, by induction, we get  $g \cap gcorelR_{\tau_i} = g' \cap gcorelR_{\tau_i}$ . Since  $g \cap gcorelR_{\tau} = \cup_i g \cap gcorelR_{\tau_i}$  and  $g' \cap gcorelR_{\tau} = \cup_i g' \cap gcorelR_{\tau_i}$  we obtain that  $g \cap gcorelR_{\tau} = g' \cap gcorelR_{\tau}$ .  $\square$

If we type  $v_1$  and  $v_2$  as  $\tau$  under the same  $g$  and  $f$ , and the  $DD(\tau)$ ,  $Det(\tau)$ , then we can conclude that the two values are the same:

**Lemma 14** *Let  $\Theta \vdash_g^f v_1 : \tau$ ,  $\Theta \vdash_{g'}^f v_2 : \tau$  and  $g \neq \epsilon$ ,  $DD(\tau)$ ,  $Det(\tau)$ , with  $\tau \neq K_{HC}(\tau') \kappa$ . Then  $v_1 = v_2$ .*

*Proof.* We proceed by induction on the structure of  $\tau$ :

Base case:

$\delta_C \delta_I$

Since  $DD(\delta_C \delta_I)$  requires that  $C \not\sqsubseteq_I \delta_I$  the only possible types are  $\delta_C[D]$  and  $\delta_C[D : \{D_1, \dots, D_m\}]$ . In the former case  $v_1, v_2 = g(D)$ , in the latter case  $v_1, v_2 = f_{D:D_1, \dots, D_m}(g(D_1), \dots, g(D_m))$ .

Inductive case:

$K_{\delta'}(\tau') \kappa$

We have assumed  $\tau \neq K_{HC}(\tau) \kappa$  thus  $\delta' = LL$ , which is anyway forbidden as  $C \sqsubseteq_I L$ . Thus this case never occurs.

$enc_{\delta'} \kappa$

Notice that if  $\kappa$  refers to a low key we have  $LL$  in  $LT_{\Delta}(enc_{\delta'} \kappa)$ , making  $DD(\tau)$  false. Thus we only consider the case in which  $\kappa$  refers to a high deterministic key  $cK_{HC}(\tau') \kappa$ , given that randomized keys are excluded by condition  $Det(\tau)$ . Now we have  $DD(\tau)$  iff  $DD(\tau')$  and  $Det(\tau)$  iff  $Det(\tau')$ . Since, by (v-enc),  $v_i = \{v'_i\}_k$  and  $\Theta \vdash_g^f v'_1 : \tau'$ ,  $\Theta \vdash_{g'}^f v'_2 : \tau'$ , by induction we get  $v'_1 = v'_2$ , which gives the thesis.

$(\tau_1, \tau_2)$

Let  $v_i = (v_i^1, v_i^2)$ . We have that  $DD(\tau)$ ,  $Det(\tau)$  iff  $DD(\tau_1)$ ,  $Det(\tau_1)$  and  $DD(\tau_2)$ ,  $Det(\tau_2)$ ; By rule (v-pair) we have that  $\Theta \vdash_g^f v_i^j : \tau$  and, by induction, we get  $v_1^j = v_2^j$  from which  $v_1 = (v_1^1, v_1^2) = (v_2^1, v_2^2) = v_2$ .  $\square$

When we remove the condition on deterministic cryptography, we can prove  $\rho_{HH}(v_1) = \rho_{HH}(v_2)$  instead of  $v_1 = v_2$ :

**Lemma 15** *Let  $\Theta \vdash_g^f v_1 : \tau$ ,  $\Theta \vdash_g^f v_2 : \tau$  and  $g \neq \epsilon$ ,  $\text{DD}(\tau)$ , with  $\tau \neq \text{K}_{HC}(\tau') \kappa$ . Then  $\rho_{HH}(v_1) = \rho_{HH}(v_2)$ .*

*Proof.* Proof of this lemma is exactly as the one above apart from randomized encryptions. Notice, in fact, that we have removed the condition  $\text{Det}(\tau)$ . The equality only holds above  $HH$  as, intuitively, at that level we can enter randomized encryptions and disregards confounders. Formally, we are in the case  $\text{enc}_{\delta'} \kappa$  with  $\text{cK}_{HC}^R(\tau') \kappa$ . By (v-enc),  $v_i = \{v'_i, r_i\}_k$  and  $\Theta \vdash_g^f v'_1 : \tau'$ ,  $\Theta \vdash_g^f v'_2 : \tau'$ , and by induction we get  $\rho_{HH}(v'_1) = \rho_{HH}(v'_2)$ . Now it is sufficient to notice that  $\rho_{HH}(\{v'_i, r_i\}_k) = \{\rho_{HH}(v'_i), \perp\}_k$ , which gives the thesis.  $\square$

**Proposition 4** *Let  $\Theta \vdash_g^{f_i} v_i : \tau$ ,  $\Theta \vdash_{g'}^{f_i} v'_i : \tau$ ,  $g, g' \neq \epsilon$  and  $\text{CloseDD}^{\text{det}}(\tau)$ , with  $\tau \neq \text{K}_{HC}(\tau') \kappa$ . Then  $v_1 = v'_1$  iff  $v_2 = v'_2$ .*

*Proof.* We prove the  $\Rightarrow$  implication. The other direction is completely analogous. By lemma 12(ii) we get that

$$\Theta \vdash_{g \cap \text{gcore}_{\tau}}^{f_i} v_i : \tau \quad \Theta \vdash_{g' \cap \text{gcore}_{\tau}}^{f_i} v'_i : \tau$$

By  $\text{CloseDD}(\tau)$  we know that  $\text{Dep}(\tau) \subseteq \text{IRs}(\tau)$  which implies  $\text{gcore}_{\tau} = \text{gcorelR}_{\tau}$ . Thus

$$\Theta \vdash_{g \cap \text{gcorelR}_{\tau}}^{f_i} v_i : \tau \quad \Theta \vdash_{g' \cap \text{gcorelR}_{\tau}}^{f_i} v'_i : \tau$$

By lemma 13 we get that  $v_1 = v'_1$  implies  $g \cap \text{gcorelR}_{\tau} = g' \cap \text{gcorelR}_{\tau} = \tilde{g}$ . Thus we get

$$\Theta \vdash_{\tilde{g}}^{f_2} v_2 : \tau \quad \Theta \vdash_{\tilde{g}}^{f_2} v'_2 : \tau$$

From lemma 14 we finally get  $v_2 = v'_2$ .  $\square$

No well typed expression will ever return a confounder.

**Lemma 16** *Let  $\Delta \vdash_g^f M$ . Then  $\Delta \vdash e : \tau$  and  $e \downarrow^M v$  imply  $v$  satisfies assumption 3.*

*Proof.* Easy by assumption 3 and by induction on the structure of expressions. The only interesting cases are (i) randomized encryptions, which generate one confounder  $r$  in the value  $\{v, r\}_k$  thus respecting the assumption, and (ii) randomized decryptions, which always disregard the confounder. Assumption 2(1) ensures that no other expression will ever generate confounders.  $\square$

It is now useful to prove that expressions evaluated at level  $\delta_C H$  never return ciphertext, MACs or pairs.

**Lemma 17** *Let  $\Delta \vdash_g^f M$ . Then  $\Delta \vdash e : \delta_C H$  and  $e \downarrow^M v$  imply  $v = n$ .*

*Proof.* Ciphertexts, MACs and pairs can never be typed as  $\delta_C H$ . Notice, in particular, that MACs always have low-integrity, ciphertexts can be promoted only to low-integrity and pairs are not in the subtype relation. The fact that  $n \notin C$  derives from assumptions 2(1) and 3  $\square$

Notice that when we have  $\rho_{\ell}(M_1) = \rho_{\ell}(M_2)\rho$  it might be the case that some hidden values in  $M_2$  are not in the domain of  $\rho$ . This means that these values  $\square_v$  are untouched by  $\rho$  but we might of course extend  $\rho$  so that  $\rho(\square_v) = \square_v$ . We note  $\bar{\rho}$  this ‘closure’ of  $\rho$  and we obviously have  $\rho_{\ell}(M_1) = \rho_{\ell}(M_2)\bar{\rho}$ . Notice that if item (3) of definition 8 holds for  $\rho$  it trivially holds also for  $\bar{\rho}$ . In fact, by lemma 8, since  $v$  is subvalue of a values stored in  $M_2$ , which is well-formed, we have that  $\Theta \vdash_{g'}^f v : \tau'$ . By corollary 1 we obtain that  $\Theta \vdash^f v : \tau'$  when  $H \sqsubseteq_I \mathcal{L}_I(\tau)$ , as required by item (3) of definition 8. In the next result, when  $\ell = HL$  we adopt the nonstandard semantics of equality check formalized in assumption 2(3).

**Proposition 1 (Expression equivalence).** *Let  $M_1 \approx_\ell^\Delta M_2$  and let  $\Delta \vdash e : \tau$  with  $\mathcal{L}(\tau) \sqsubseteq \ell$ . Then,  $e \downarrow^{M_i} v_i$  and  $\mathfrak{p}_\ell(M_1) = \mathfrak{p}_\ell(M_2)\rho$  and  $\rho$  f-respects  $\Theta$ , imply  $\mathfrak{p}_\ell(v_1) = \mathfrak{p}_\ell(v_2)\rho'$  such that  $\bar{\rho} \subseteq \rho'$  and  $\rho'$  f-respects  $\Theta$ .*

*Proof.* By induction on the structure of the expression  $e$ :

$x$

From  $\Delta \vdash x : \tau$  we know, by (var) and (sub), that  $\Delta(x) = \tau'$  with  $\mathcal{L}(\tau') \sqsubseteq \mathcal{L}(\tau) \sqsubseteq \ell$ . As a consequence the variable is observable, i.e.,  $\mathfrak{p}_\ell(M_1) = \mathfrak{p}_\ell(M_2)\rho$  implies  $\mathfrak{p}_\ell(v_1) = \mathfrak{p}_\ell(v_2)\bar{\rho}$ , which gives the thesis (with  $\bar{\rho} = \rho'$ ).

pair( $e_1, e_2$ )

We have  $\Delta \vdash \text{pair}(e_1, e_2) : (\tau_1, \tau_2)$ . By (pair) and since pairs are not in the subtype relation, we know that  $\Delta \vdash e_1 : \tau_1$  and  $\Delta \vdash e_2 : \tau_2$ . It always holds  $\mathcal{L}(\tau_i) \sqsubseteq \mathcal{L}(\tau_1, \tau_2) \sqsubseteq \ell$ . Let  $e_i \downarrow^{M_j} v_j^i$ . By induction on  $e_1$  we can say  $\mathfrak{p}_\ell(v_1^1) = \mathfrak{p}_\ell(v_2^1)\rho'_1$  and  $\mathfrak{p}_\ell(v_1^1) = \mathfrak{p}_\ell(v_2^1)\bar{\rho}'_1$  with  $\bar{\rho} \subseteq \rho'_1 \subseteq \bar{\rho}'_1$  and  $\rho'_1, \bar{\rho}'_1$  f-respect  $\Theta$ . Since  $\bar{\rho}$  maps all the hidden values in  $M_2$  we also have  $\mathfrak{p}_\ell(M_1) = \mathfrak{p}_\ell(M_2)\bar{\rho}'_1$ . By induction on  $e_2$  we now obtain  $\mathfrak{p}_\ell(v_1^2) = \mathfrak{p}_\ell(v_2^2)\rho'_2$  with  $\bar{\rho}'_1 \subseteq \rho'_2$  and  $\rho'_2$  f-respects  $\Theta$ . Notice that, as above, we also have  $\mathfrak{p}_\ell(v_1^1) = \mathfrak{p}_\ell(v_2^1)\rho'_2$ , thus  $\mathfrak{p}_\ell((v_1^1, v_2^1)) = \mathfrak{p}_\ell((v_1^2, v_2^2))\rho'_2$  and since  $\bar{\rho} \subseteq \rho'_1 \subseteq \bar{\rho}'_1 \subseteq \rho'_2$ , we obtain the thesis with  $\rho' = \rho'_2$ .

fst( $e'$ )

We have  $\Delta \vdash \text{fst}(e') : \tau$  with  $\mathcal{L}(\tau) \sqsubseteq \ell$ . By (pair) and (sub) we know that  $\Delta \vdash e' : (\tau_1, \tau_2)$  and  $\tau_1 \leq \tau$  thus  $\mathcal{L}(\tau_1) \sqsubseteq \mathcal{L}(\tau) \sqsubseteq \ell$ . By assumption 1 we know that either  $\mathcal{L}(\tau_1) = \mathcal{L}(\tau_2)$  (when  $H \sqsubseteq \mathcal{L}_I(\tau_1)$ ) or  $H \not\sqsubseteq \mathcal{L}_I(\tau_1), \mathcal{L}_I(\tau_2)$ . In both cases we obtain that  $\mathcal{L}((\tau_1, \tau_2)) \sqsubseteq \ell$ . Let  $e' \downarrow^{M_j} v_j$ . By induction we get  $\mathfrak{p}_\ell(v_1) = \mathfrak{p}_\ell(v_2)\rho'_1$  such that  $\bar{\rho} \subseteq \rho'_1$  and  $\rho'_1$  f-respects  $\Theta$ . From this we know that either  $v_i = (v_i^1, v_i^2)$  or none of them is a pair, respectively giving  $e \downarrow^{M_j} v_j^1$  or  $e \downarrow^{M_j} \perp$ . In the former case it is sufficient to observe that  $\mathfrak{p}_\ell(v_1^1) = \mathfrak{p}_\ell(v_2^1)\rho'_1$  and we thus obtain the thesis with  $\rho' = \rho'_1$ . The latter case trivially give the thesis with  $\rho' = \bar{\rho}$  since no new mapping is required to match  $\perp$  with  $\perp$ .

snd( $e'$ )

Exactly as above.

enc $_x^\mu(e')$

The expression considered by this case is  $e = \text{enc}_x^\mu(e')$ . From the type system we know that  $\Delta(x) = \text{cK}_{\delta'}^R(\tau') \kappa$ ,  $\Delta \vdash e' : \tau'$  and  $\Delta \vdash e : \text{enc}_\delta \kappa$  with  $\text{enc}_\delta \kappa \leq \tau$  meaning that  $\delta \sqsubseteq \mathcal{L}(\tau) \sqsubseteq \ell$ . Let  $e' \downarrow^{M_j} v_j$  and  $x \downarrow^{M_j} v_j^k$ . We proceed by cases depending on the typing rule used to get  $\Delta \vdash e : \text{enc}_\delta \kappa$ :

(enc) we have that  $\delta = \delta' \sqcup \mathcal{L}(\tau')$ , meaning that  $\delta', \mathcal{L}(\tau') \sqsubseteq \ell$ , i.e., both the key and the subexpression are observable at  $\ell$ . By induction we get  $\mathfrak{p}_\ell(v_1^k) = \mathfrak{p}_\ell(v_2^k)\rho'_k$ , meaning that either  $v_i^k = k \in \mathcal{K}$  or none of them is a key, respectively giving  $e \downarrow^{M_j} \{v_j\}_k$  or  $e \downarrow^{M_j} \perp$ . The latter case trivially give the thesis with  $\rho' = \bar{\rho}$  since no new mapping is required to match  $\perp$  with  $\perp$ . In the former case we just observe that, since the key is below  $\ell$ , we have  $\mathfrak{p}_\ell(\{v_j\}_k) = \{\mathfrak{p}_\ell(v_j)\}_k$ . By induction (since  $\mathcal{L}(\tau') \sqsubseteq \ell$ ) we get  $\mathfrak{p}_\ell(v_1) = \mathfrak{p}_\ell(v_2)\rho'_1$  such that  $\bar{\rho} \subseteq \rho'_1$  and  $\rho'_1$  f-respect  $\Theta$ , from which the thesis with  $\rho' = \rho'_1$ .

(enc-r) we have  $\Delta(x) = \text{cK}_{HC}^R(\tau') \kappa$  and  $e = \text{enc}_x^R(e')$ . By  $\Delta \vdash_{\mathfrak{g}}^{f_i} M_i$  we are guaranteed that  $M_i(x) = k \in \mathcal{K}_{HC}$  where  $k$  is the unique key such that  $\Theta(k) = \text{cK}_{HC}^R(\tau') \kappa$  (recall  $\Theta$  is injective on high keys). Thus  $e \downarrow^{M_j} \{v_j, r_j\}_k$ , with  $r_j$  extracted fresh. We have two cases: if  $HC \sqsubseteq \ell$  the key is observable and we have  $\mathfrak{p}_\ell(\{v_j, r_j\}_k) = \{\mathfrak{p}_\ell(v_j), \perp\}_k$ . By rule (enc-r) we know that  $\delta = LC \sqcup \mathcal{L}_I(\tau')$ , meaning that  $\mathcal{L}_I(\tau') \sqsubseteq_I \mathcal{L}_I(\delta) \sqsubseteq_I \mathcal{L}_I(\ell)$  and, since in this case we have  $HC \sqsubseteq \ell$ , we obtain  $\mathcal{L}(\tau') \sqsubseteq \ell$ . We can thus apply induction and we get  $\mathfrak{p}_\ell(v_1) = \mathfrak{p}_\ell(v_2)\rho'_1$  such that  $\bar{\rho} \subseteq \rho'_1$  and  $\rho'_1$  f-respect  $\Theta$ , from which the thesis with  $\rho' = \rho'_1$ . If  $HC \not\sqsubseteq \ell$  the

key is not observable and we have  $\text{p}_\ell(\{v_j, r_j\}_k) = \square_{\{v_j, r_j\}_k}$ . Thesis is trivially obtained with  $\rho' = \bar{\rho} \cup [\square_{\{v_2, r_2\}_k} \mapsto \square_{\{v_1, r_1\}_k}]$ . Even in this case the fact confounders are fresh trivially guarantees that the extension is possible and by proposition 3 we get  $\Theta \vdash_{g\tau}^{f_i} \{v_i, r_i\}_k : \tau$ , i.e.,  $\rho'$  f-respects  $\Theta$ .

(enc-d) We have  $\Delta(x) = \text{cK}_{HC}(\tau') \kappa$  and  $e = \text{enc}_x(e')$ . As above, by  $\Delta \vdash_g^{f_i} M_i$  we are guaranteed that  $M_i(x) = k \in \mathcal{K}_{HC}$  where  $k$  is the unique key such that  $\Theta(k) = \text{cK}_{HC}(\tau') \kappa$ . Thus  $e \downarrow^{M_j} \{v_j\}_k$ . We have two cases: if  $HC \sqsubseteq \ell$  the key is observable and we have  $\text{p}_\ell(\{v_j\}_k) = \{\text{p}_\ell(v_j)\}_k$ . As for the above case, by rule (enc-d) we know that  $\delta = LC \sqcup \mathcal{L}_I(\tau')$  from which we derive  $\mathcal{L}(\tau') \sqsubseteq \ell$ . Thus, by induction we get  $\text{p}_\ell(v_1) = \text{p}_\ell(v_2)\rho'_1$  such that  $\bar{\rho} \subseteq \rho'_1$  and  $\rho'_1$  f-respect  $\Theta$ , from which the thesis with  $\rho' = \rho'_1$ . if  $HC \not\sqsubseteq \ell$  the key is not observable and we have  $\text{p}_\ell(\{v_j\}_k) = \square_{\{v_j\}_k}$ . If  $\square_{\{v_2\}_k} \notin \text{dom}(\bar{\rho})$  and  $\square_{\{v_1\}_k} \notin \text{img}(\bar{\rho})$  we trivially obtain the thesis with  $\rho' = \bar{\rho} \cup [\square_{\{v_2\}_k} \mapsto \square_{\{v_1\}_k}]$  (notice that by proposition 3 we get  $\Theta \vdash_{g\tau}^{f_i} \{v_i\}_k : \tau$ , i.e.,  $\rho'$  f-respects  $\Theta$ ). If, instead,  $\square_{\{v_2\}_k} \in \text{dom}(\bar{\rho})$  or  $\square_{\{v_1\}_k} \in \text{img}(\bar{\rho})$  we show that  $\bar{\rho}(\square_{\{v_2\}_k}) = \square_{\{v_1\}_k}$ , i.e., the needed mapping is already in  $\bar{\rho}$  and we thus have the thesis by simply taking  $\rho' = \bar{\rho}$ . Assume, then, that  $\square_{\{v_2\}_k} \in \text{dom}(\bar{\rho})$ . By the constraints on hidden value substitutions we know that  $\bar{\rho}(\square_{\{v_2\}_k}) = \square_{\{v'\}_k}$ , since the mapping is required to respect keys. By the fact that  $\bar{\rho}$  f-respects  $\Theta$  we know that  $\exists g'$  such that  $\Theta \vdash_{g'}^{f_i} v_2 : \tau'$  and  $\Theta \vdash_{g'}^{f_i} v' : \tau'$ . By lemma 11 and since  $\text{CloseDD}(\tau')$  and, by assumption 1(3),  $\tau' \neq \mathcal{K}_{HC}(\tau) \kappa$  we get that  $\mathcal{L}_I(\tau') \sqsubset_I H$ , thus  $g' \neq \epsilon$ . Since  $\Delta \vdash e' : \tau'$  and  $\Delta \vdash_g^{f_i} M_i$ , by proposition 3 we get  $\Theta \vdash_g^{f_i} v_i : \tau'$ . By  $\text{CloseDD}^{\text{det}}(\tau')$  and proposition 4 we directly obtain that  $v_1 = v'$ , showing that the new mapping is the same already in  $\bar{\rho}$ . (The other direction is proved similarly.)

$\text{mac}_x(e')$

The expression considered by this case is  $e = \text{mac}_x(e')$ . From the type system we know that  $\Delta(x) = \text{cK}_{\delta'_C \delta'_I}^\mu(\tau') \kappa$ ,  $\Delta \vdash e' : \tau'$  and  $\Delta \vdash e : LL \sqcup \mathcal{L}(\tau')$  with  $LL \sqcup \mathcal{L}(\tau') \leq \tau$  meaning that  $\mathcal{L}(\tau') \sqsubseteq \mathcal{L}(\tau) \sqsubseteq \ell$ , i.e., the subexpression is observable at  $\ell$ . Let  $e' \downarrow^{M_j} v_j$  and  $x \downarrow^{M_j} v_j^k$ . We proceed by cases depending on level  $\delta'_C \delta'_I$  of the key.

If  $\delta'_C \delta'_I = LL$ , we have that also the key is observable at  $\ell$ . By induction we get  $\text{p}_\ell(v_1^k) = \text{p}_\ell(v_2^k)\rho'_k$ , meaning that either  $v_i^k = k \in \mathcal{K}$  or none of them is a key, respectively giving  $e \downarrow^{M_j} \langle v_j \rangle_k$  or  $e \downarrow^{M_j} \perp$ . The latter case trivially give the thesis with  $\rho' = \bar{\rho}$  since no new mapping is required to match  $\perp$  with  $\perp$ . In the former case we just observe that  $\text{p}_\ell(\langle v_j \rangle_k) = \langle \text{p}_\ell(v_j) \rangle_k$ . By induction we get  $\text{p}_\ell(v_1) = \text{p}_\ell(v_2)\rho'_1$  such that  $\bar{\rho} \subseteq \rho'_1$  and  $\rho'_1$  f-respect  $\Theta$ , from which the thesis with  $\rho' = \rho'_1$ .

If  $\delta'_C \delta'_I = HC$ , the key might be not observable at  $\ell$ . However, since the key is high and the memories are well-formed, by  $\Delta \vdash_g^{f_i} M_i$  we are guaranteed that  $M_i(x) = k \in \mathcal{K}_{HC}$  where  $k$  is the unique key such that  $\Theta(k) = \text{cK}_{HC}^R(\tau') \kappa$  (recall  $\Theta$  is injective on high keys). Thus  $e \downarrow^{M_j} \langle v_j \rangle_k$ . Now observe that  $\text{p}_\ell(\langle v_j \rangle_k) = \langle \text{p}_\ell(v_j) \rangle_k$  (this is independent of the level of the key). By induction, as above, we get  $\text{p}_\ell(v_1) = \text{p}_\ell(v_2)\rho'_1$  such that  $\bar{\rho} \subseteq \rho'_1$  and  $\rho'_1$  f-respect  $\Theta$ , from which the thesis with  $\rho' = \rho'_1$ .

$\text{dec}_x(e')$

The expression is  $e = \text{dec}_x(e')$  and  $\Delta \vdash e : \tau$  with  $\mathcal{L}(\tau) \sqsubseteq \ell$ . From the type system we know that  $\Delta(x) = \text{cK}_{\delta'}^\mu(\tau') \kappa$ ,  $\Delta \vdash e' : \text{enc}_{\delta''} \kappa$ . Let  $e' \downarrow^{M_j} v_j'$  and  $x \downarrow^{M_j} v_j^k$ . In case of rule (dec), we know that  $\delta' \sqcup \delta'' \leq \tau$  meaning that  $\delta', \delta'' \sqsubseteq \ell$ , i.e., both the key and the subexpression are observable at  $\ell$ . For (dec- $\mu$ ) we know that  $\tau' \leq \tau$ . By the fact  $\mathcal{L}_C(\tau) = H$  we know that  $\ell = H\ell_I$  thus the key is observable, and by the typing rule we know that  $\mathcal{L}_I(\delta'') = \mathcal{L}_I(\delta') \sqcup \mathcal{L}_I(\tau') = C \sqcup \mathcal{L}_I(\tau')$  which implies  $\mathcal{L}_I(\delta'') \sqsubseteq_I \ell_I$ . Thus  $\delta'' \sqsubseteq \ell$  meaning that also the subexpression is observable. By induction we get  $\text{p}_\ell(v_1^k) = \text{p}_\ell(v_2^k)\rho'_k$  and  $\text{p}_\ell(v_1') = \text{p}_\ell(v_2')\rho'_1$  meaning that either  $v_i^k = k \in \mathcal{K}$  or none of them is a key, and either  $v_i' = \{v_j\}_k$  or none of them is a ciphertext based on key  $k$ .

When  $v_i^k \notin \mathcal{K}$  or  $v_i' \neq \{v_j\}_k$  we obtain  $e \downarrow^{M_j} \perp$ . We easily get the thesis with  $\rho' = \bar{\rho}$  since no new mapping is required to match  $\perp$  with  $\perp$ . Otherwise, we get  $e \downarrow^{M_j} v_j$ . Observe that, since the key is below  $\ell$ , we have  $\mathfrak{p}_\ell(\{v_j\}_k) = \{\mathfrak{p}_\ell(v_j)\}_k$ . By the above induction we had that  $\bar{\rho} \subseteq \rho'_1$  and  $\rho'_1$  f-respect  $\Theta$ , from which the thesis with  $\rho' = \rho'_1$ .

$e_1 \text{ op } e_2$

$\Delta \vdash e_1 \text{ op } e_2 : \delta$ . By (op) we know that  $\Delta \vdash e_1 : \delta'$  and  $\Delta \vdash e_2 : \delta'$  with  $\delta' \sqsubseteq \delta \sqsubseteq \ell$ , thus both subexpressions are observable. Let  $e_i \downarrow^{M_j} v_j^i$ . By induction on  $e_1$  we can say  $\mathfrak{p}_\ell(v_1^1) = \mathfrak{p}_\ell(v_2^1)\rho'_1$  and  $\mathfrak{p}_\ell(v_1^1) = \mathfrak{p}_\ell(v_2^1)\bar{\rho}'_1$  with  $\bar{\rho} \subseteq \rho'_1 \subseteq \bar{\rho}'_1$  and  $\rho'_1, \bar{\rho}'_1$  f-respect  $\Theta$ . Since  $\bar{\rho}$  maps all the hidden values in  $M_2$  we also have  $\mathfrak{p}_\ell(M_1) = \mathfrak{p}_\ell(M_2)\bar{\rho}'_1$ . By induction on  $e_2$  we now obtain  $\mathfrak{p}_\ell(v_1^2) = \mathfrak{p}_\ell(v_2^2)\rho'_2$  with  $\bar{\rho}'_1 \subseteq \rho'_2$  and  $\rho'_2$  f-respects  $\Theta$ . Notice that, as above, we also have  $\mathfrak{p}_\ell(v_1^1) = \mathfrak{p}_\ell(v_2^1)\rho'_2$ . We consider two cases: if  $e_1 \text{ op } e_2$  is the equality test  $e_1 = e_2$  by lemma 16 and 17 we know that for  $\delta \sqsubseteq HH$  we have  $v_j^i = n_j^i \notin C$ . Thus, easily,  $v_j^1 = v_j^2$  iff  $\mathfrak{p}_\ell(v_j^1) = \mathfrak{p}_\ell(v_j^2)$ . By lemma 16 we have that the same holds for  $\delta = LL$ , since no counfounders will be abstracted in the patterns (since high keys are not known) and hidden values are decorated by the whole encrypted message. When  $\delta = HL$  we exploit assumption 2(3). We obtain that  $v_1^1 = v_1^2$  iff  $v_2^1 = v_2^2$ . Thus we get the thesis with  $\rho' = \rho'_2$ . For all the other generic expressions, by assumption 2(2), we know that  $e_1 \text{ op } e_2 \not\downarrow^M \perp$  implies  $e_i \downarrow^M n_i$  with  $n_i \notin \mathcal{K} \cup C$ . By  $\mathfrak{p}_\ell(v_1^i) = \mathfrak{p}_\ell(v_2^i)\rho'_2$  we obtain that  $e_i \downarrow^{M_1} n_i$  iff  $e_i \downarrow^{M_2} n_i$ , ( $n_i \notin \mathcal{K} \cup C$ ) meaning that the expression  $e_1 \text{ op } e_2$  either evaluate to  $\perp$  or to the very same  $n$  in both memories. We thus get the thesis with  $\rho' = \rho'_2$ .  $\square$

**Corollary 2** Let  $M_1 \approx_\ell^\Delta M_2$  and let  $\Delta \vdash e : \tau$  and  $e \downarrow^{M_i} v_i$ . If  $\mathcal{L}(\tau) \sqsubseteq \ell$  or  $\mathcal{L}(\Delta(x)) \not\sqsubseteq \ell$  then  $M_1[x \mapsto v_i] \approx_\ell^\Delta M_2[x \mapsto v_i]$ .

*Proof.* If  $\mathcal{L}(\tau) \sqsubseteq \ell$ , by  $M_1 \approx_\ell^\Delta M_2$  we know that there exists  $\rho$  such that  $\mathfrak{p}_\ell(M_1) = \mathfrak{p}_\ell(M_2)\rho$  and  $\rho$  f-respects  $\Theta$ . Now, by proposition 1 we directly have  $\mathfrak{p}_\ell(v_1) = \mathfrak{p}_\ell(v_2)\rho'$  such that  $\bar{\rho} \subseteq \rho'$  and  $\rho'$  f-respects  $\Theta$ . Of course we also have  $\mathfrak{p}_\ell(M_1) = \mathfrak{p}_\ell(M_2)\rho'$  and, consequently,  $\mathfrak{p}_\ell(M_1)[x \mapsto v_1] = \mathfrak{p}_\ell(M_2)[x \mapsto v_2]\rho'$ . If, instead,  $\mathcal{L}(\Delta(x)) \not\sqsubseteq \ell$  then the variable  $x$  is above the observation level and we trivially have  $M_i|_\ell = M_i[x \mapsto v_i]|_\ell$  which gives  $\mathfrak{p}_\ell(M_1)[x \mapsto v_1] = \mathfrak{p}_\ell(M_1) = \mathfrak{p}_\ell(M_2)\rho = \mathfrak{p}_\ell(M_2)[x \mapsto v_2]\rho$ . We still need to prove that  $\Delta \vdash_{g'}^{f_i} M_i[x \mapsto v_i]$ . This is directly achieved by proposition 3:  $\Delta \vdash_{g'}^{f_i} M_i$  and  $\Delta \vdash e : \tau$  imply  $\Theta \vdash_{g\tau}^{f_i} v_i : \tau$  which implies  $\Delta \vdash_{g'}^{f_i} M_i[x \mapsto v_i]$ .  $\square$

**Lemma 18 (Confinement)** If  $\Delta, pc \vdash c$  then for every variable  $x$  assigned to in  $c$  and such that  $\Delta(x) = \tau$  it holds that  $pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH$ .

*Proof.* We proceed by induction on the structure of  $c$ .

skip

The command does not assign to any variable thus the lemma trivially holds.

$x := e$

This command can be typed by two different rules: (assign) and (declassify). The former requires  $pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH$  while the latter  $pc \sqsubseteq \mathcal{L}(\tau)$ , which directly give the thesis.

$c_1; c_2$

Since  $\Delta, pc \vdash c_i$ , the thesis follows directly by induction on  $c_1$  and  $c_2$ .

if  $b$  then  $c_1$  else  $c_2$

Two different rules may be used to type this command.

(if)

The typing rule assures that the expression  $b$  types  $\tau'$  and  $\Delta, \mathcal{L}(\tau') \sqcup pc \vdash c_i$ . By induction on  $c_1$  and  $c_2$  it holds that for every variable  $x$  assigned to in  $c_1, c_2$  such that  $\Delta(x) = \tau$   $\mathcal{L}(\tau') \sqcup pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH$ , thus  $pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH$ .

**(if-MAC)**

For variable  $y$  we directly have  $pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH$ . Since  $\Delta, pc \vdash c_1$  and  $\Delta, pc \vdash c_2$ , by induction on the two commands we get the thesis.

while  $e$  do  $c$

This case is analogous to the former one of the if command.  $\square$

We now prove noninterference on all the level but  $LH$ . In fact, on  $LH$ , the property does not hold since, intuitively, randomized but high-equivalent messages (i.e., messages encrypted with high keys differing only for the values of counfounders) might be distinguished by  $LH$  users, who know no keys, via traffic analysis. Once more, this is like the attacker fouling himself, by ‘incompetently’ sending messages on which he can perform traffic analysis. We disregard this uninteresting behaviour by just requiring  $\ell \sqsupset LH$ . As for expression equivalence, in the next result, when  $\ell = HL$  we adopt the nonstandard semantics of equality check formalized in assumption 2(3).

**Theorem 4 (Noninterference)** *Let  $c$  be a program which does not contain any declassification statement. If  $\Delta, pc \vdash c$  then  $c$  satisfies noninterference, i.e.,  $\forall \ell \sqsupset LH, M_1, M_2. M_1 \approx_\ell^\Delta M_2$  implies  $\langle M_1, c \rangle \approx_\ell^\Delta \langle M_2, c \rangle$ .*

*Proof.* Recall that  $\langle M_1, c \rangle \approx_\ell^\Delta \langle M_2, c \rangle$  denotes weak-indistinguishability, i.e., whenever  $\langle M_i, c \rangle \Rightarrow M'_i$  we have  $M'_1 \approx_\ell^\Delta M'_2$ . We proceed by induction on derivation length of  $\langle M_i, c \rangle \Rightarrow M'_i$ .

Base case, length 1:

**[skip]**

Since  $\langle M_i, \text{skip} \rangle \Rightarrow M_i$  we have nothing to prove.

**[assign]**

We have  $\langle M_i, x := e \rangle \Rightarrow M_i[x \mapsto v_i] = M'_i$  given that  $e \downarrow^{M_i} v_i$ . If rule (assign) have been used then  $\Delta(x) = \tau$ ,  $\Delta \vdash e : \tau$ . Notice that either  $\mathcal{L}(\tau) \sqsubseteq \ell$  or  $\ell \sqsupset \mathcal{L}(\tau) = \mathcal{L}(\Delta(x))$ . Corollary 2 directly gives the thesis. By hypothesis, we do not have declassify commands, thus rule (declassify) cannot have been used to type the assignment.

Inductive case, length  $n$ . We consider the last rule applied:

**[seq]**

We have  $\langle M_i, c_1; c_2 \rangle \Rightarrow M'_i$  since  $\langle M_i, c_1 \rangle \Rightarrow M''_i$  and also  $\langle M''_i, c_2 \rangle \Rightarrow M'_i$ . By rule (seq) we have that  $\Delta, pc \vdash c_i$ . By induction on  $c_1$  we have  $M''_1 \approx_\ell^\Delta M''_2$  and by induction on  $c_2$  we obtain  $M'_1 \approx_\ell^\Delta M'_2$ .

**[ift], [iff]**

Let  $\langle M_i, \text{if } b \text{ then } c_1 \text{ else } c_2 \rangle \Rightarrow M'_i$ . The typing can be made by rules (if) and (if-MAC).

For rule (if) it holds that  $\Delta \vdash b : \tau$ ,  $\Delta, \mathcal{L}(\tau) \sqcup pc \vdash c_1$  and  $\Delta, \mathcal{L}(\tau) \sqcup pc \vdash c_2$ . If  $\mathcal{L}(\tau) \sqsubseteq \ell$  then by proposition 1 the boolean expression  $b$  necessarily evaluates to two identical boolean values (or  $\perp$ ) so the same branch will be followed. Suppose  $e \downarrow^{M_i} \text{true}$ , we have that  $\langle M_i, c_1 \rangle \Rightarrow M'_i$ , thus by induction on  $c_1$ ,  $M'_1 \approx_\ell^\Delta M'_2$  (analogously for false/ $\perp$  and  $c_2$ ). If instead  $\ell \sqsupset \mathcal{L}(\tau)$  by lemma 18 for every variable  $x$  assigned by  $c_1$  and  $c_2$  such that  $\Delta(x) = \tau'$  it holds that  $\mathcal{L}(\tau) \sqcup pc \sqsubseteq \mathcal{L}(\tau') \sqcup LH$ , thus  $\ell \sqsupset \mathcal{L}(\tau') \sqcup LH$  which implies  $\ell \sqsupset \mathcal{L}(\tau')$  (since  $\ell$  is at least  $LH$ ). Intuitively, all the assignments performed by the branches are above the level of observation. By corollary 2 we directly obtain  $M'_1 \approx_\ell^\Delta M'_2$ .

In case (if-MAC) has been applied then the command has the following form:

$$\text{if } \text{mac}_x(z, e) = e' \text{ then } (y := e; c_1) \text{ else } c_2; \perp_{\text{MAC}}$$

and it must be that  $\Delta(x) = \text{mK}_{HC}(L[D], \tau)$ ,  $\Delta \vdash z : L[D]$ ,  $\Delta \vdash e : LL$ ,  $\Delta(y) = \tau$ ,  $\Delta \vdash e' : LL$ . Moreover  $\Delta, pc \vdash c_1$  and  $\Delta, pc \vdash c_2$ . Note that if one of the execution takes the else branch it will not terminate, giving  $\langle M_1, c \rangle \approx_{\ell}^{\Delta} \langle M_2, c \rangle$ . We thus consider the case in which both executions take the if branch. Let  $e \downarrow^{M_i} v_i$ , we consider different cases depending on level  $\ell$ :

Let  $LL \sqsubseteq \ell$ , by corollary 2 we get that  $M_1[y \mapsto v_i] \approx_{\ell}^{\Delta} M_2[y \mapsto v_i]$ .

If  $LL \not\sqsubseteq \ell$  By hypothesis  $\ell \sqsupset LH$  thus the only possibility is  $\ell = HH$ : as in the proof of theorem 3 (case *ift*) we get that  $\text{mac}_x(z, e) \downarrow^{M_i} \langle v_i, v'_i \rangle_k$  and  $e' \downarrow^{M_i} \langle v_i, v'_i \rangle_k$  and  $\Theta \vdash_g^{f_i} v'_i : \tau$ . Since memories are well-formed and, by lemma 11 we have that  $\mathcal{L}_I(\tau) \sqsubset_I H$ , we know that  $\Theta \vdash_g^{f_i} M_i(y) : \tau$ . Now by lemma 15 we have that  $\rho_{HH}(v'_i) = \rho_{HH}(M_i(y))$ , meaning that the assignment does not change in any way the equality of the two memories, i.e.,  $M_1[y \mapsto v_i] \approx_{HH}^{\Delta} M_2[y \mapsto v_i]$ .

We have thus proved that, for all  $\ell \sqsupset LH$ ,  $M_1[y \mapsto v_i] \approx_{\ell}^{\Delta} M_2[y \mapsto v_i]$ . Now, by induction on  $c_1$  we get the thesis.

**[whilet], [whilef]**

For rule (while) it holds that  $\Delta \vdash b : \tau$ ,  $\Delta, \mathcal{L}(\tau) \sqcup pc \vdash c$ . If  $\mathcal{L}(\tau) \sqsubseteq \ell$  then by proposition 1 the boolean expression  $b$  evaluates to two identical boolean values (or  $\perp$ ) so the same branch will be followed. Suppose  $e \downarrow^{M_i} \text{true}$ , we have that  $\langle M_i, c \rangle \Rightarrow M'_i$  and  $\langle M'_i, \text{while } e \text{ do } c \rangle \Rightarrow M'_i$ , thus by induction (on the length of the derivation),  $M'_1 \approx_{\ell}^{\Delta} M'_2$  and then  $M'_1 \approx_{\ell}^{\Delta} M'_2$ . If instead  $\ell \sqsubset \mathcal{L}(\tau)$  by lemma 18 for every variable  $x$  assigned by  $c$  such that  $\Delta(x) = \tau'$  it holds that  $\mathcal{L}(\tau) \sqcup pc \sqsubseteq \mathcal{L}(\tau') \sqcup LH$ , thus  $\ell \sqsubset \mathcal{L}(\tau') \sqcup LH$  which implies  $\ell \sqsubset \mathcal{L}(\tau')$  (since  $\ell$  is at least  $LH$ ). Intuitively, all the assignments performed by the loops are above the level of observation. By corollary 2 we directly obtain  $M'_1 \approx_{\ell}^{\Delta} M'_2$ . When the guard is false the result trivially holds, since memories remain untouched.  $\square$

When we re-introduce declassification statements, noninterference still holds for  $\ell = HH$  as, as expected, integrity is not broken by declassifying information. This is proved in the following theorem:

**Theorem 5** *If  $\Delta, pc \vdash c$  then  $\forall M_1, M_2$  such that  $M_1 \approx_{HH}^{\Delta} M_2$  it holds  $\langle M_1, c \rangle \approx_{HH}^{\Delta} \langle M_2, c \rangle$*

*Proof.* The proof of this theorem is a simple extension of the one of theorem 4, instantiated with  $\ell = HH$ , to programs which contain declassification. The only case that we need to extend is the assignment, since it is where declassification may occur: We have  $\langle M_i, x := \text{declassify}(e') \rangle \Rightarrow M_i[x \mapsto v_i] = M'_i$  given that  $e \downarrow^{M_i} v_i$ . From the type system it follows  $\Delta \vdash e' : \delta'_C H \sqsubseteq HH$ . By corollary 2 we obtain the thesis.  $\square$

We can now state our final results on robustness. We will consider programs that assign declassified data to special variables assigned only once. This can be easily achieved syntactically, e.g., by using one different variable for each declassification statement (which we label for clarity), i.e.,  $x_1 := \text{declassify}_1(e_1), \dots, x_m := \text{declassify}_m(e_m)$ , and avoiding to place declassifications inside while loops. These special variables are nowhere else assigned. We call this class of programs *CD-programs*.

**Lemma 19 (CD-programs)** *In a CD-program  $c$ , if  $\langle M_1, x := \text{declassify}(e') \rangle \Rightarrow M_1[x \mapsto v_1]$  and  $\langle M_2, y := e \rangle \Rightarrow M_2[y \mapsto v_2]$  occur in the derivation of  $\langle M, c \rangle \Rightarrow M'$  then  $x \neq y$ .*

*Proof.* Easy by induction on the structure of commands, by observing that the only command re-executing the same syntactic portion of a program is the while-loop.  $\square$

Let  $\Delta, pc \vdash c$ . We now prove that any value  $v$  declassified during the computation of  $c$  on a well-formed memory is an atomic name.

**Lemma 20** *Let  $\Delta, pc \vdash c$  and  $\Delta \vdash_g^f M$ . Then any  $\langle M_1, x := \text{declassify}(e') \rangle \Rightarrow M_1[x \mapsto v_1]$  occurring in the derivation of  $\langle M, c \rangle \Rightarrow M'$  is such that  $v_1 = n$  and  $n \notin C$ .*

*Proof.* It is sufficient to notice that the declassified value is expected to be of type  $\delta_C H$  and apply lemma 17.  $\square$

Information leakage in clearly declassifying programs can be always observed by inspecting the equality of declassifying variables as proved in the following:

**Lemma 21** *Let  $c$  be a CD-program on variables  $x_1, \dots, x_k$ . If  $\Delta, pc \vdash c$ ,  $M_1 \approx_{LL}^\Delta M_2$ ,  $\langle M_i, c \rangle \Rightarrow M'_i$  and  $M'_1 \not\approx_{LL}^\Delta M'_2$  then  $\exists i$  such that  $M'_1(x_i) = n \neq n' = M'_2(x_i)$ .*

*Proof.* Lemma 20 already proves that declassified values are all atomic names. We now proceed by induction on the number of declassifying variables in  $c$ .

Base case (no declassification): We have  $M'_1 \approx_{LL}^\Delta M'_2$  by theorem 4, thus we have nothing to prove.

Inductive case: Assume we have  $x_i, \dots, x_m$  declassifying variables. Let  $M'_1 \not\approx_{LL}^\Delta M'_2$ . If  $M'_1(x_m) \neq M'_2(x_m)$  we are done. Otherwise it will be  $M'_1(x_m) = M'_2(x_m) = n$ . We replace  $x_m := \text{declassify}_m(e_m)$  with  $x_m := n$  obtaining  $c''$  and re-execute it, i.e.,  $\langle M_i, c'' \rangle \Rightarrow M''_i$ . By lemma 19 we know that the above one was the only assignment done to  $x_m$  during the derivation, thus  $M'_i = M''_i$ , thus  $M'_1 \not\approx_{LL}^\Delta M'_2$ . By induction we now have that  $\exists i \in [1, m-1]$  such that  $M'(x_i) = M'_1(x_i) = n \neq n' = M'_2(x_i) = M'_2(x_i)$ .  $\square$

**Lemma 22** *Let  $c_1; c_2$  be a CD-program such that  $\Delta, pc \vdash c_1; c_2$ . Then  $\langle M_1, c_1; c_2 \rangle \simeq_{LL}^\Delta \langle M_2, c_1; c_2 \rangle$  implies  $\langle M_1, c_1 \rangle \simeq_{LL}^\Delta \langle M_2, c_1 \rangle$ .*

*Proof.*  $\langle M_1, c_1; c_2 \rangle \simeq_{LL}^\Delta \langle M_2, c_1; c_2 \rangle$  means that  $\langle M_i, c_1; c_2 \rangle \Rightarrow M'_i$  and  $M'_1 \approx_{LL}^\Delta M'_2$ . Notice, in fact, that  $\simeq_{LL}^\Delta$  is strong and requires both executions to terminate. We know that  $\langle M_i, c_1 \rangle \Rightarrow M''_i$  and  $\langle M''_i, c_2 \rangle \Rightarrow M'_i$ . Since  $c_1; c_2$  is a CD-program, so are  $c_1$  and  $c_2$ , as the condition is purely syntactic. Assume, by contradiction, that  $M''_1 \not\approx_{LL}^\Delta M''_2$ . By lemma 21 there exist a declassifying variable  $x$  such that  $M''_1(x) = n \neq n' = M''_2(x)$ . By lemma 19 we know that  $x$  will never be assigned again in  $\langle M''_i, c_2 \rangle \Rightarrow M'_i$ , thus giving  $M'_1(x) = n \neq n' = M'_2(x)$  and the contradiction  $M'_1 \not\approx_{LL}^\Delta M'_2$ .  $\square$

Next result finally proves robustness of well-typed CD-programs. Notice that it only adopts the standard semantics of equality check and is thus not based on assumption 2(3).

**Theorem 6 (Robustness)** *If a CD-program  $c$  is such that  $\Delta, pc \vdash c$  then  $c$  satisfies robustness, i.e.,  $\forall M_1, M_2, M'_1, M'_2$  such that  $M_1 \approx_{LL}^\Delta M_2$ ,  $M'_1 \approx_{LL}^\Delta M'_2$  and  $M_i \approx_{HH}^\Delta M'_i$  it holds*

$$\langle M_1, c \rangle \simeq_{LL}^\Delta \langle M_2, c \rangle \text{ implies } \langle M'_1, c \rangle \approx_{LL}^\Delta \langle M'_2, c \rangle$$

*Proof.* The proof follows by induction on the structure of the command  $c$ .

skip

By  $\langle M'_i, \text{skip} \rangle \Rightarrow M'_i$  we directly get  $\langle M'_1, c \rangle \approx_{LL}^\Delta \langle M'_2, c \rangle$ .

$x := e$

If the expression  $e$  is not a declassification then by Theorem 4,  $c$  satisfies noninterference, thus  $\langle M'_1, x := e \rangle \approx_{LL}^\Delta \langle M'_2, x := e \rangle$ .

Suppose  $e = \text{declassify}(e')$ . The type system states that  $\Delta(x) = \delta_C H$ ,  $\Delta \vdash e' : \delta'_C H$  and  $pc \sqsubseteq \delta_C H$ . Let  $e' \downarrow^{M_i} v_i$  and  $e' \downarrow^{M'_i} v'_i$ . By corollary 2, since  $M_i \approx_{HH}^\Delta M'_i$  and  $\delta'_C H \sqsubseteq HH$ ,

we get  $M_i[x \mapsto v_i] \approx_{HH}^{\Delta} M'_i[x \mapsto v'_i]$ . By lemma 20 we know that all these values are atomic and not confounders, thus we get  $v_i = v'_i = n_i$ . Now,  $\langle M_1, x := e \rangle \simeq_{LL}^{\Delta} \langle M_2, x := e \rangle$  implies  $M_1[x \mapsto v_1] \approx_{LL}^{\Delta} M_2[x \mapsto v_2]$ , meaning  $v_1 = v_2$ . We get  $v'_1 = v_1 = v_2 = v'_2$ . Thus  $M'_1[x \mapsto v'_1] \approx_{LL}^{\Delta} M'_2[x \mapsto v'_2]$ , giving the thesis  $\langle M'_1, x := e \rangle \approx_{LL}^{\Delta} \langle M'_2, x := e \rangle$ .

if  $b$  then  $c_1$  else  $c_2$

If rule (if) has been used to type the command, then  $\Delta \vdash b : \tau$  and  $\Delta, \mathcal{L}(\tau) \sqcup pc \vdash c_i$ .

If  $\mathcal{L}_I(\tau) = L$  then  $\mathcal{L}(\tau) \sqcup pc \not\sqsubseteq \delta_C H$  thus, by lemma 18, no declassification may occur in both  $c_1$  and  $c_2$ . By Theorem 4, the whole command is noninterferent giving thesis  $\langle M'_1, c \rangle \approx_{LL}^{\Delta} \langle M'_2, c \rangle$ .

If  $\mathcal{L}(\tau) = HH$  then  $\Delta, HH \sqcup pc \vdash c_i$  and by Confinemnt lemma (Lemma 18) it holds that for every variable  $x$  assigned to by  $c_1$  and  $c_2$  such that  $\Delta(x) = \tau'$  then  $HH \sqcup pc \sqsubseteq \mathcal{L}(\tau') \sqcup LH$ , meaning  $\mathcal{L}(\tau') \not\sqsubseteq LL$ . Thus, by corollary 2,  $\langle M'_1, c \rangle \approx_{LL}^{\Delta} \langle M'_2, c \rangle$ .

Suppose  $\mathcal{L}(\tau) = LH$  then the same branch will be followed by each of the four executions. Let, for example  $b \downarrow^{M_i}$  false and  $b \downarrow^{M'_i}$  false, then it must be that  $\langle M_i, \text{if } e \text{ then } c_1 \text{ else } c_2 \rangle \Rightarrow N_i$  because of  $\langle M_i, c_2 \rangle \Rightarrow N_i$  (by command semantics rule [iff]) thus  $\langle M_1, c_2 \rangle \simeq_{LL}^{\Delta} \langle M_2, c_2 \rangle$  which by induction on  $c_2$  gets  $\langle M'_1, c_2 \rangle \approx_{LL}^{\Delta} \langle M'_2, c_2 \rangle$ . This implies  $\langle M'_1, \text{if } b \text{ then } c_1 \text{ else } c_2 \rangle \approx_{LL}^{\Delta} \langle M'_2, \text{if } b \text{ then } c_1 \text{ else } c_2 \rangle$ . (Analogously for the case  $b \downarrow^{M_i}$  true.)

If rule (if-MAC) has been used to type the command it is

$$\text{if } \text{mac}_x(z, e) = e' \text{ then } (y := e; c_1) \text{ else } c_2; \perp_{\text{MAC}}$$

and it must be that  $\Delta(x) = \text{mK}_{HC}(L[D], \tau)$ ,  $\Delta \vdash z : L[D]$ ,  $\Delta \vdash e : LL$ ,  $\Delta(y) = \tau$ ,  $\Delta \vdash e' : LL$  and also  $\Delta, pc \vdash c_1, \Delta, pc \vdash c_2$ . Suppose  $e \downarrow^{M_i} v_i, z \downarrow^{M_i} v_i^*$  and  $e \downarrow^{M'_i} v'_i, z \downarrow^{M'_i} v'_i^*$ . Since from the hypothesis the program terminates in both  $M_1$  and  $M_2$  it must be that the then-branch has been followed in both configurations giving  $\langle M_1, y := e; c_1 \rangle \simeq_{LL}^{\Delta} \langle M_2, y := e; c_1 \rangle$ .

Notice that if one of the executions  $\langle M'_1, c \rangle$  and  $\langle M'_2, c \rangle$  takes the else branch it will not terminate, directly giving  $\langle M'_1, c \rangle \approx_{LL}^{\Delta} \langle M'_2, c \rangle$ . We thus consider the case in which both executions take the if branch. By corollary 2 we get that  $M_1[y \mapsto v_1] \approx_{LL}^{\Delta} M_2[y \mapsto v_2]$ ,  $M'_1[y \mapsto v'_1] \approx_{LL}^{\Delta} M'_2[y \mapsto v'_2]$ . As in the proof of lemma 3 (case [ift]) and Theorem 4 we get that  $\text{mac}_x(z, e) \downarrow^{M_i} \langle v_i^*, v_i \rangle_k$ ,  $\text{mac}_x(z, e) \downarrow^{M'_i} \langle v_i^*, v_i \rangle_k$  and  $e' \downarrow^{M_i} \langle v_i^*, v_i \rangle_k$ ,  $e' \downarrow^{M'_i} \langle v_i^*, v_i \rangle_k$  and  $\Theta \vdash_g^{f_i} v_i : \tau, \Theta \vdash_g^{f_i} v'_i : \tau$ . Since memories are well-formed and, by lemma 11 we have that  $\mathcal{L}_I(\tau) \sqsubset_I H$ , we know that  $\Theta \vdash_g^{f_i} M_i(y) : \tau, \Theta \vdash_g^{f_i} M'_i(y) : \tau$ . Now by lemma 15 we have that  $\rho_{HH}(v_i) = \rho_{HH}(M_i(y))$  and  $\rho_{HH}(v'_i) = \rho_{HH}(M'_i(y))$ , meaning that the assignment does not change in any way the equality of the two memories, i.e.,  $M_i[y \mapsto v_i] \approx_{HH}^{\Delta} M'_i[y \mapsto v_i]$ . Hence it is possible to apply induction on  $c_1$  to conclude the case.

while  $e$  do  $c_1$

No declassification is allowed inside a while loop, thus the case follows by Theorem 4.

$c_1; c_2$

Rule (seq) states  $\Delta, pc \vdash c_1$  and  $\Delta, pc \vdash c_2$ . By Lemma 22 since  $\langle M_1, c_1; c_2 \rangle \simeq_{LL}^{\Delta} \langle M_2, c_1; c_2 \rangle$  then it must be that  $\langle M_1, c_1 \rangle \simeq_{LL}^{\Delta} \langle M_2, c_1 \rangle$  thus by induction we get  $\langle M'_1, c_1 \rangle \approx_{LL}^{\Delta} \langle M'_2, c_1 \rangle$ . Let  $\langle M_i, c_1 \rangle \Rightarrow N_i^1$ ,  $\langle M'_i, c_1 \rangle \Rightarrow N_i^2$  it holds  $N_1^1 \approx_{LL}^{\Delta} N_2^1$ ,  $N_1^2 \approx_{LL}^{\Delta} N_2^2$  and by theorem 5  $N_i^1 \approx_{HH}^{\Delta} N_i^2$ . The thesis follows by induction on  $c_2$ .  $\square$