In Proceedings of *14th European Symposium on Research in Computer Security* (ESORICS'09). Springer LNCS 5789.

Type-based Analysis of PIN Processing APIs *

Matteo Centenaro[1], Riccardo Focardi[1], Flaminia L. Luccio[1] and Graham Steel[2]

[1] Dipartimento di Informatica, Università Ca' Foscari Venezia, Italy
[2] LSV, ENS Cachan & CNRS & INRIA, France

Abstract. We examine some known attacks on the PIN verification framework, based on weaknesses of the 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 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.

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 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 must be designed so that should an attacker gain access to a host machine connected to an 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

* Work partially supported by Miur'07 Project SOFT.

from hacked switches have 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" [22]. We take a step in that direction, using the techniques of language-based security [24].

One can immediately see that the current API functions allow an 'information flow' from the high security PIN to the low security result. However, the function must 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] permitting certain flows. The problem is that an intruder can often manipulate input data in order to declassify data in an unintended way. Again there is a technique for this: 'robust declassification' [23], whereby we disallow 'low integrity' data, which might have been manipulated by the attacker, to affect what can be declassified. However, the functionality of the 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, 22, 26]. In this paper, we first introduce our main case study, the PIN verification command (§1). We review some notions of language based security (§2). We describe our modelling of cryptographic primitives, and in particular MACs for assuring integrity, and we show why PIN verification fails to be robust (§3). Our type system is presented (§4), the MAC-based improved API is type-checked (§5), and finally we conclude (§6). For lack of space we omit all the proofs (see [14]).

The Case Study 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*[3] 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

[3] This value 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.

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 below, behaves as follows:

```
PIN_V(PAN, EPB, len, offset, vdata, dectab) {
    x1 := enc_pdk(vdata);
    x2 := left(len, x1);
    x3 := decimalize(dectab, x2);
    x4 := sum_mod10(x3, offset);
    x5 := dec_k(EPB);
    x6 := fcheck(x5);
    if (x6 =⊥) then return("format wrong");
    if (x4 = x6) then return("PIN correct");
                 else return("PIN wrong")}
```

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

The given code specifies a strict subset of the real PIN verification function named `Encrypted_PIN_Verify` [18].

Example 1. Let *len*=4, *offset*=4732, *dectab*=9753108642543210, this last parameter encoding this mapping: $0 \mapsto 9, 1 \mapsto 7, \ldots, F \mapsto 0$. Let also $x_1 = \mathsf{enc}_{pdk}(vdata) =$ A47295FDE32A48B1. Then, $x_2 = \mathsf{left}(4, A47295FDE32A48B1)$ = A472 , $x_3 = \mathsf{decimalize}(dectab, A472) = 5165$, and $x_4 = \mathsf{sum_mod10}(5165, 4732)$ = 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 and let us assume that $EPB = \{|9897, r|\}_k$. We thus have: $x_5 = \mathsf{dec}_k(\{|9897, r|\}_k)$ $= (9897, r)$, and $x_6 = \mathsf{fcheck}(9897, \mathsf{r}) = 9897$. Finally, since x_6 is different from $\perp$ (failure) and $x_4 = x_6$ the API returns $''PIN\ correct''$.

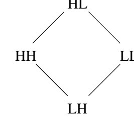
2 Basic Language and Security

In this section, we recall a standard imperative language core and some basic security notions. An expression e is either a variable x or an arithmetic/Boolean operation on expressions e_1 op e_2. Denoting Boolean expressions by b, the syntax of *commands* is $\mathsf{c} ::= \mathsf{skip} \mid x := e \mid \mathsf{c}_1; \mathsf{c}_2 \mid \mathsf{if}\ b\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 \mid \mathsf{while}\ b\ \mathsf{do}\ \mathsf{c}$.

Memories M are finite maps from variables to values and we write $\mathsf{M}(x)$ to denote the value associated to x in M. Moreover, $e \downarrow^{\mathsf{M}} v$ denotes the evaluation of expression e in a memory M giving value v as a result: for example, $x \downarrow^{\mathsf{M}} \mathsf{M}(x)$ and $x + x' \downarrow^{\mathsf{M}} \mathsf{M}(x) + \mathsf{M}(x')$. Moreover, $\langle \mathsf{M}, \mathsf{c} \rangle \Rightarrow \mathsf{M}'$ denotes the execution of a command c in a memory M, resulting in a new memory M'. Finally, $\mathsf{M}[x \mapsto v]$ denotes the update of variable x to the new value v. For example, $\langle \mathsf{M}, x := e \rangle \Rightarrow$

$\mathsf{M}[x \mapsto v]$ if $e \downarrow^{\mathsf{M}} v$. Security APIs are executed on trusted hardware with no multi-threading, we thus adopt a standard big-step semantics similar to that of Volpano et al. [28] which can be found in [14].

Security A *security environment* Γ 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 ℓ_1, ℓ_2, we write $\ell_1 \sqsubseteq_C \ell_2$ ($\ell_1 \sqsubseteq_I \ell_2$) to denote that ℓ_1 is as restrictive or less restrictive than ℓ_2. In particular, low-confidentiality data may be used more liberally than high-confidentiality ones, thus in this case $L \sqsubseteq_C H$; dually, $H \sqsubseteq_I L$. 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* $\mathsf{M}|_\ell$ *denote the restriction of memory* M *to variables whose security level is at or below level* ℓ. M_1 *and* M_2 *are* indistinguishable *at level* ℓ*, written* $\mathsf{M}_1 =_\ell \mathsf{M}_2$*, if* $\mathsf{M}_1|_\ell = \mathsf{M}_2|_\ell$. *Two configurations are* indistinguishable*, written* $\langle \mathsf{M}_1, \mathsf{c}\rangle =_\ell \langle \mathsf{M}_2, \mathsf{c}\rangle$*, if whenever* $\langle \mathsf{M}_1, \mathsf{c}\rangle \Rightarrow \mathsf{M}_1'$ *and* $\langle \mathsf{M}_2, \mathsf{c}\rangle \Rightarrow \mathsf{M}_2'$ *then* $\mathsf{M}_1' =_\ell \mathsf{M}_2'$. *They are* strongly indistinguishable*, written* $\langle \mathsf{M}_1, \mathsf{c}\rangle \cong_\ell \langle \mathsf{M}_2, \mathsf{c}\rangle$*, if* $\langle \mathsf{M}_1, \mathsf{c}\rangle =_\ell \langle \mathsf{M}_2, \mathsf{c}\rangle$ *and* $\langle \mathsf{M}_1, \mathsf{c}\rangle \Rightarrow \mathsf{M}_1'$, $\langle \mathsf{M}_2, \mathsf{c}\rangle \Rightarrow \mathsf{M}_2'$.

Noninterference requires that data from one level should never interfere with lower levels. Intuitively, command c satisfies noninterference if, fixed a level ℓ, two indistinguishable memories remain indistinguishable even after executing c.

Definition 2 (Noninterference). *A command* c *satisfies* noninterference *if* $\forall\ \ell, \mathsf{M}_1, \mathsf{M}_2$ *we have that* $\mathsf{M}_1 =_\ell \mathsf{M}_2$ *implies* $\langle \mathsf{M}_1, \mathsf{c}\rangle =_\ell \langle \mathsf{M}_2, \mathsf{c}\rangle$.

Noninterference formalizes full security, with no leakage of confidential information ($\ell = LL$) or corruption of high-integrity data ($\ell = HH$). 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 1, PIN_V returns the correctness of the typed PIN which is a one-bit leak of information about a secret datum. Robustness will allow us to check that attackers cannot abuse such a declassification and gain more information than intended.

Consider a pair of memories $\mathsf{M}_1, \mathsf{M}_2$ which are not distinguishable by an intruder, i.e., $\mathsf{M}_1 =_{LL} \mathsf{M}_2$. The execution of c on these memories may leak confidential information violating noninterference, i.e., $\langle \mathsf{M}_1, \mathsf{c}\rangle \neq_{LL} \langle \mathsf{M}_2, \mathsf{c}\rangle$. 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., $\mathsf{M}_1' =_{LL} \mathsf{M}_2'$, as he is deterministic and starts from indistinguishable memories; (ii) they only differ from the initial ones in the low-integrity part, i.e., $\mathsf{M}_1 =_{HH} \mathsf{M}_1'$, $\mathsf{M}_2 =_{HH} \mathsf{M}_2'$, given that only low-integrity variables can be modified by intruders. Following MSZ, we require that attackers start from strongly indistinguishable, terminating configurations to avoid they 'incompetently' self-corrupt their observations.

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

This notion is a novel simplification of that of MSZ, who allow 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, with no need of making their code explicit. Memory manipulations and multiple runs performed by attackers are covered by considering all $=_{HH}$ memories.

Example 2. We write x_ℓ to denote a variable of level ℓ. 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 is neither noninterferent nor robust.

M_1	M_2
y_{HH} : 1234	y_{HH} : 5678
x_{LL} : 1111	x_{LL} : 1111

M'_1	M'_2
y_{HH} : 1234	y_{HH} : 5678
x_{LL} : 1234	x_{LL} : 1234

To see this latter fact, consider the memories on the right. It clearly holds that $\mathsf{M}_1 =_{HH} \mathsf{M}'_1$, $\mathsf{M}_2 =_{HH} \mathsf{M}'_2$ and $\mathsf{M}_1 =_{LL} \mathsf{M}_2$, $\mathsf{M}'_1 =_{LL} \mathsf{M}'_2$, but the execution of P in the first two memories leads to indistinguishable results in z_{LL}, false/false, thus $\langle \mathsf{M}_1, P\rangle \cong_{LL} \langle \mathsf{M}_2, P\rangle$, while for the second ones we get true/false, and so $\langle \mathsf{M}'_1, P\rangle \not\cong_{LL} \langle \mathsf{M}'_2, P\rangle$. 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 x_{LL}.

3 Cryptographic primitives

In order to model our API case-study, we now extend arithmetic and Boolean expressions with confounder generation $\mathsf{new}()$, symmetric cryptography $\mathsf{enc}_x(e)$, $\mathsf{dec}_x(e)$, Message Authentication Codes (MACs) $\mathsf{mac}_x(e)$, pairing $\mathsf{pair}(e_1, e_2)$ and projection $\mathsf{fst}(e), \mathsf{snd}(e)$. We extend standard values as, e.g., Booleans and integers, with confounders $r \in C$ and cryptographic keys $k \in \mathcal{K}$. 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 $\mathsf{enc}_x(e) \downarrow^{\mathsf{M}} \{\!|v|\!\}_k$ whenever $e \downarrow^{\mathsf{M}} v$ and $x \downarrow^{\mathsf{M}} k$. Moreover, $\mathsf{dec}_x(e') \downarrow^{\mathsf{M}} v$ if $e' \downarrow^{\mathsf{M}} \{\!|v|\!\}_k$ and $x \downarrow^{\mathsf{M}} k$; otherwise $\mathsf{dec}_x(e') \downarrow^{\mathsf{M}} \perp$, representing failure, and analogously for the other expressions. Confounder generation $\mathsf{new}() \downarrow^{\mathsf{M}} r$ 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. Formally, if $r, r' \leftarrow C$ then $r \neq r'$. Moreover, similarly to [3, 4], we assume C to be disjoint from the set of atomic names used in programs. Full semantics of expressions can be found in [14].

To guarantee a safe use of cryptography we also assume that every expression e different from enc, dec, mac, pair, and that 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 $\bot$; (ii) never produces those values. This is important to avoid "magic" expressions which encrypt/decrypt/MAC messages without knowing the key like, e.g., $\mathsf{magicdecrypt}(e) \downarrow^{\mathsf{M}} v$ when $e \downarrow^{\mathsf{M}} \{\!|v|\!\}_n$. However, we permit equality checks as they allow the intruder to track equal encryptions, as occurs in traffic analysis.

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; (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. Notice, however, 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 HSMs while the latter can be used by the attackers. This is achieved by partitioning the set $\mathcal{K}$ into $\mathcal{K}_{HH}$ and $\mathcal{K}_{LL}$.

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. Consider the memories M_1 and M_2, with $k \in \mathcal{K}_{HH}$. An attacker cannot distinguish $\{\!|1234|\!\}_k$ from $\{\!|9999|\!\}_k$ and $\{\!|1234|\!\}_k$ from $\{\!|5678|\!\}_k$. However, running the above intruder-program on these memories, he respectively obtains $z_{LL} = \mathsf{true}$ and $z_{LL} = \mathsf{false}$, i.e., the resulting memories clearly differ. The intruder has in fact detected the presence of two equal ciphertexts in the first memory which allows him to distinguish M_1 and M_2.

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

Patterns and indistinguishability 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, 20]. 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, e.g., in [4–6].

Patterns p extend values with the new symbol $\square_v$ representing messages encrypted with a key not available at the observation level ℓ. More precisely, we define a function $\mathsf{p}_\ell(v)$ 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 $\mathsf{p}_{LL}(\{\!|1234|\!\}_k) = \square_{\{\!|1234|\!\}_k}$ while $\mathsf{p}_{HH}(\{\!|1234|\!\}_k) = \{\!|1234|\!\}_k$. Function $\mathsf{p}_\ell(v)$ descends recursively into subvalues. For example, if $k' \in \mathcal{K}_{LL}$ we have $\mathsf{p}_{LL}(\{\!|\{\!|10|\!\}_k, 20|\!\}_{k'}) = \{\!|\square_{\{\!|10|\!\}_k}, 20|\!\}_{k'}$. In case of MACs, we just descend into subvalues, i.e., $\mathsf{p}_\ell(\langle v\rangle_k) = \langle \mathsf{p}_\ell(v)\rangle_k$, i.e., we assume that all messages inside MACs are public.

Notice that, in $\square_v$, v is the whole (inaccessible) encrypted value, instead of just a confounder as used in previous works [4, 5, 13, 20]. 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 ρ to the pattern p, and we write $\mathsf{M}\rho$ to denote the memory in which ρ has been applied to all the patterns of M. On hidden values substitutions we always require that keys are correctly mapped. Formally $\rho(\square_{\{\!|v|\!\}_k}) = \square_{\{\!|v'|\!\}_k}$.

Definition 4 (Crypto-indistinguishability). *Let* $\mathsf{p}_\ell(\mathsf{M})$ *denote* $\mathsf{M}|_\ell$ *in which all of the values* v *have been substituted by* p_ℓ. M_1 *and* M_2 *are indistinguishable at* ℓ*, written* $\mathsf{M}_1 \approx_\ell \mathsf{M}_2$*, if there exists* ρ *such that* $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\ \rho$ *.*

Example 4. Consider again 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. It is impossible to find a hidden values substitution ρ 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 $\mathsf{M}_1 \not\approx_{LL} \mathsf{M}_2$. If, instead, $\mathsf{M}_1(y_{LL})$ were, e.g., $\{\!|2222|\!\}_k$ we might use $\rho = [\square_{\{\!|9999|\!\}_k} \mapsto \square_{\{\!|1234|\!\}_k}, \square_{\{\!|5678|\!\}_k} \mapsto \square_{\{\!|2222|\!\}_k}]$ obtaining $\mathsf{p}_{LL}(\mathsf{M}_1) = \mathsf{p}_{LL}(\mathsf{M}_2)\rho$ and thus $\mathsf{M}_1 \approx_{LL} \mathsf{M}_2$.

$\mathsf{p}_{LL}(\mathsf{M}_1)$	$\mathsf{p}_{LL}(\mathsf{M}_2)$
$x_{LL} : \square_{\{\!|1234|\!\}_k}$	$x_{LL} : \square_{\{\!|9999|\!\}_k}$
$y_{LL} : \square_{\{\!|1234|\!\}_k}$	$y_{LL} : \square_{\{\!|5678|\!\}_k}$

Noninterference and robustness Security notions of section 2 naturally extend to the new cryptographic setting by substituting $=_\ell$ with $\approx_\ell$ everywhere. We need to be careful that memories do not leak cryptographic keys, i.e., that keys disclosed at level ℓ are all of that level or below, and that variables intended to contain keys really do contain keys. This will be achieved in section 4 via a notion of memory well-formedness.

Formal analysis of an API attack on $\mathsf{PIN_V}$ We now illustrate how the lack of integrity of the API parameters can be exploited to mount a real attack leaking the PIN, and we show how this is formally captured as a violation of robustness. We consider the case study of section 1 and concentrate on two specific parameters, the *dectab* and the *offset*, which are used to respectively calculate the values of x_3 and x_4. 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 $d+1$ mod 10, and then checks whether the system still returns "*PIN correct*". Depending on this, the intruder discovers whether or not digit d is present in the user 'natural' PIN contained in x_3;
2. when a certain digit is discovered in the previous step by a "*PIN 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 digit.

Example 5. In example 1 we let *len*=4, *dectab*=9753108642543210, *offset*=4732, *x1*= A47295FDE32A48B1, *EPB*=$\{\!|9897, r|\!\}_k$. With these parameters the API returns "*PIN correct*". The attacker first chooses $dectab'$=$97531\underline{1}864254321\underline{1}$, where the two 0's have been replaced by 1's. The aim is to discover whether or not 0 appears in x_3. Invoking the API with $dectab'$ we obtain the same intermediate and final values, as $\mathsf{decimalize}(dectab', A472) = \mathsf{decimalize}(dectab, A472) = 5165$. This means that 0 does not appear in x_3. The attacker proceeds by replacing the 1's of *dectab* by 2's: with $dectab''$=$9753\underline{2}086425432\underline{2}0$ he obtains that $\mathsf{decimalize}(dectab'', A472)$=5265 $\neq$ $\mathsf{decimalize}(dectab, A472)$=5165, reflecting the presence of 1's in the original value of x_3. Then, x_4=$\mathsf{sum_mod10}(5265, 4732)$ =9997 instead of 9897 returning "*PIN wrong*".

The intruder now knows that digit 1 occurs is 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: $\underline{3}732, 4\underline{6}32, 47\underline{2}2, 473\underline{1}$. Notice that, in this specific case, offset value 4632 makes the API return again "*PIN correct*". 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. We now show that this is formally captured as a violation of robustness. We adopt a small trick to model the PIN derivation encryption of x_1: we write *vdata* as a ciphertext, e.g., $\{\!|\mathrm{A47295FDE32A48B1}|\!\}_{pdk}$, and we model the first encryption as a decryption $x_1 := \mathsf{dec}_{pdk}(vdata)$. 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 the four memories below, that only differ in the value of *EPB* and *dectab*. It clearly holds that $\mathsf{M}_1 \approx_{HH} \mathsf{M}_1'$, $\mathsf{M}_2 \approx_{HH} \mathsf{M}_2'$ and $\mathsf{M}_1 \approx_{LL} \mathsf{M}_2$,

$\mathsf{M}_1' \approx_{LL} \mathsf{M}_2'$, the last two using $\rho = [\Box_{\{\!|1234,r'|\!\}_k} \mapsto \Box_{\{\!|9897,r|\!\}_k}]$. Since parameters are all at level LL, these memories could be built by an attacker sniffing all encryptions arriving at the verification facility. If we execute $\mathsf{PIN_V}$ in M_1 and M_2 we obtain $''PIN\ wrong''$ in both cases as for memory M_2, the encrypted PIN is wrong, and for memory M_1, the encrypted PIN is correct but the *dectab''* will change the value of derived PIN. It follows $\langle \mathsf{M}_1, \mathsf{PIN_V}\rangle \approx_{LL} \langle \mathsf{M}_2, \mathsf{PIN_V}\rangle$. In M_1' and M_2' the *dectab* is the correct one. Thus, executing $\mathsf{PIN_V}$ gives, respectively, $''PIN\ correct''$ and $''PIN\ wrong''$ and so $\langle \mathsf{M}_1', \mathsf{PIN_V}\rangle \not\approx_{LL} \langle \mathsf{M}_2', \mathsf{PIN_V}\rangle$, breaking robustness. To overcome this problem, integrity of the input must be established.

M_1	M_2
dectab''	*dectab''*
$\{\!\vert 9897, r\vert\!\}_k$	$\{\!\vert 1234, r'\vert\!\}_k$

M_1'	M_2'
dectab	*dectab*
$\{\!\vert 9897, r\vert\!\}_k$	$\{\!\vert 1234, r'\vert\!\}_k$

4 Type system

We now give a new type system to statically check that a program with cryptographic primitives satisfies robustness and, if it does not declassify any information, noninterference. We will then use it to type-check a MAC-based variant of the PIN verification and PIN translation API.

We refine integrity levels by introducing the notion of *dependent domains* used to track integrity dependencies among variables. Dependent domains are denoted $\mathsf{D} : \tilde{\mathsf{D}}$ where $\mathsf{D} \in \mathcal{D}$ is a domain name. Intuitively, the values of domain $\mathsf{D} : \tilde{\mathsf{D}}$ are determined by the values in the set of domains $\tilde{\mathsf{D}}$. For example, $\mathsf{PIN} : \mathsf{PAN}$ can be read as 'the PIN value relative to the account number PAN': when the PAN is fixed, the value of the PIN is also fixed. A domain $\mathsf{D} : \emptyset$, also written D, is called *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 $\mathsf{D} : \tilde{\mathsf{D}}$, written $[\mathsf{D} : \tilde{\mathsf{D}}]$, is higher than H, i.e., $[\mathsf{D} : \tilde{\mathsf{D}}] \sqsubseteq_I H$. In some cases, e.g., in arithmetic operations, we necessarily loose information about the precise result domain $\mathsf{D} : \tilde{\mathsf{D}}$ and we only record the fact the value is determined by domains $\tilde{\mathsf{D}}$, written $\bullet : \tilde{\mathsf{D}}$. The resulting integrity preorder is $[\mathsf{D} : \tilde{\mathsf{D}}_1] \sqsubseteq [\bullet : \tilde{\mathsf{D}}_1] \sqsubseteq_I [\bullet : \tilde{\mathsf{D}}_2] \sqsubseteq_I H \sqsubseteq_I L$ with $\tilde{\mathsf{D}}_1 \subseteq \tilde{\mathsf{D}}_2$. We write δ_I to note the new integrity levels $L, H, [\mathsf{D} : \tilde{\mathsf{D}}], [\bullet : \tilde{\mathsf{D}}]$, and δ_C to note the usual confidentiality levels L, H. We also write C in place of $[\bullet]$, to denote a constant value with no specific domain. Based on new levels $\delta = \delta_C\delta_I$, we can give the type syntax:

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

Type δ is for generic data at level δ; types $\mathsf{cK}^{\mu}_{\delta}(\tau)\ \kappa$ and $\mathsf{mK}_{\delta}(\tau)$ respectively refer to encryption and MAC keys of level δ, working on data of type τ; κ is a label that uniquely identifies one key type and label μ indicates whether the ciphertext is 'randomized' via confounders ($\mu = R$) or not (μ missing); we only consider untrusted and trusted (constant) keys, respectively of level LL and

$$\text{(enc-r)}\ \frac{\Delta(x) = \mathsf{cK}^R_{H\mathsf{C}}(\tau)\ \kappa \quad \Delta \vdash e : \tau}{\Delta \vdash \mathsf{enc}^R_x(e) : \mathsf{enc}_{L\mathsf{C}\sqcup\mathcal{L}_I(\tau)}\ \kappa} \qquad \text{(mac)}\ \frac{\Delta(x) = \mathsf{mK}_\delta(\tau) \quad \Delta \vdash e : \tau}{\Delta \vdash \mathsf{mac}_x(e) : LL \sqcup \mathcal{L}(\tau)}$$

$$\text{(dec-}\mu\text{)}\ \frac{\Delta(x) = \mathsf{cK}^\mu_{H\mathsf{C}}(\tau)\ \kappa \quad \Delta \vdash e : \mathsf{enc}_{\delta_C\mathsf{C}\sqcup\mathcal{L}_I(\tau)}\ \kappa \quad \mathcal{L}_C(\tau) = H}{\Delta \vdash \mathsf{dec}^\mu_x(e) : \tau}$$

$$\text{(enc-d)}\ \frac{\Delta(x) = \mathsf{cK}_{H\mathsf{C}}(\tau)\ \kappa \quad \Delta \vdash e : \tau \quad \mathsf{CloseDD}^{\mathsf{det}}(\tau)}{\Delta \vdash \mathsf{enc}_x(e) : \mathsf{enc}_{L\mathsf{C}\sqcup\mathcal{L}_I(\tau)}\ \kappa}$$

Table 1. Security Type System - Cryptographic expressions with trusted keys.

$H\mathsf{C}$; $\mathsf{enc}_\delta\ \kappa$ is the type for ciphertexts at level δ, obtained using the unique key labelled κ; pairs are typed as (τ_1, τ_2).

A *security type environment* $\Delta : x \mapsto \tau$ maps variables to security types. The security environment Γ can be derived from Δ by just 'extracting' the level of the types as follows: $\mathcal{L}(\delta) = \mathcal{L}(\mathsf{K}_\delta(\tau)\ \kappa) = \mathcal{L}(\mathsf{enc}_\delta\ \kappa) = \delta$ and $\mathcal{L}((\tau_1, \tau_2)) = \mathcal{L}(\tau_1) \sqcup \mathcal{L}(\tau_2)$. Notice that we write $\mathsf{K}_\delta(\tau)\ \kappa$ to indifferently denote encryption and MAC key types. We also write $\mathcal{L}_C(\tau)$ and $\mathcal{L}_I(\tau)$ to respectively extract the confidentiality and integrity level of type τ.

The subtype preorder $\leq$ extends the security level preorder $\sqsubseteq$ on levels δ with $\mathsf{enc}_{\delta_C\delta_I}\ \kappa \leq \delta_C L$. Moreover, from now on, we will implicitly identify low-integrity types at the same security level, i.e., we will not distinguish τ and τ' 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 $\mathsf{K}_\delta(\tau)\ \kappa$ and (τ_1, τ_2) types $\tau, \tau_1, \tau_2 \neq \mathsf{K}_{H\mathsf{C}}(\tau)\ \kappa$. 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 $\mathsf{cK}^\mu_{H\mathsf{C}}(\tau)\ \kappa$ with $\tau = (H[\mathsf{D}], H[\mathsf{D}' : \mathsf{D}])$. Types transported by the key are all dependent domains and are closed: the set of dependencies is $\{\mathsf{D}\}$, since $[\mathsf{D}' : \mathsf{D}]$ depends on D, and the set of representatives is $\{\mathsf{D}\}$, because of the presence of the representative $[\mathsf{D}]$. If we instead consider $\tau' = (H[\mathsf{D}], H[\mathsf{D}' : \mathsf{D}], H[\mathsf{D}' : \mathsf{D}''])$ we have that the set of dependencies is $\{\mathsf{D}, \mathsf{D}''\}$ and the set of representatives is $\{\mathsf{D}\}$, meaning that the type is not closed: not all the dependencies can be found in the type. We write $\mathsf{CloseDD}(\tau)$ to denote that τ is closed and only contains dependent domains. When it additionally does not transport randomized ciphertexts we write $\mathsf{CloseDD}^{\mathsf{det}}(\tau)$. We will describe the importance of this closure conditions when describing the typing rules.

Typing cryptography and MACs Expressions are typed with judgment $\Delta \vdash e : \tau$, derived from the rules in Table 1. For lack of space we only report rules for trusted cryptographic operations; full type-system can be found in [14].

$$\frac{\Delta(x) = \delta_C H \quad \Delta \vdash e : \delta'_C H \quad pc \sqsubseteq \delta_C H}{\Delta, pc \vdash x := \mathsf{declassify}(e)} \qquad \frac{\Delta(x) = \tau \quad \Delta \vdash e : \tau \quad pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH}{\Delta, pc \vdash x := e}$$

$$\frac{\begin{array}{c}\Delta(x) = \mathsf{mK}_{H\mathsf{C}}(L[\mathsf{D}], \tau) \quad \Delta \vdash z : L[\mathsf{D}] \quad \Delta \vdash e : LL \quad \Delta \vdash e' : LL \quad \Delta(y) = \tau \\ \mathsf{IRs}(L[\mathsf{D}], \tau) = \{\mathsf{D}\} \quad \mathsf{CloseDD}(\mathsf{L}[\mathsf{D}], \tau) \quad \Delta, pc \vdash \mathsf{c}_1 \quad \Delta, pc \vdash \mathsf{c}_2 \quad pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH\end{array}}{\Delta, pc \vdash \mathsf{if}\ \mathsf{mac}_x(z, e) = e'\ \mathsf{then}\ (y := e; \mathsf{c}_1)\ \mathsf{else}\ \mathsf{c}_2; \bot_{\mathsf{MAC}}}$$

Table 2. Security Type System - Commands

Rule (enc-r) is for randomized encryption: We let $\mathsf{enc}_x^R(e)$ and $\mathsf{dec}_x^R(e)$ denote, respectively, $\mathsf{enc}_x(e, \mathsf{new}())$ and $\mathsf{fst}(\mathsf{dec}_x(e))$, i.e., an encryption randomized via a fresh confounder and the corresponding decryption. The typing rule requires a trusted key $H\mathsf{C}$. The integrity level of the ciphertext is simply the least upper bound of the levels of the key and the plaintext; the confidentiality level, instead, is L, meaning that the resulting ciphertext preserves secrecy even when written on an public/untrusted part of the memory.

Rule (dec-μ) is for (trusted) decryption and gives the correct type τ to the obtained plaintext, if the confidentiality of the plaintext is at least H. This is to avoid that indistinguishable ciphertexts are decrypted and then written on low variables, breaking noninterference in a trivial way.

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 $\{\!|\mathrm{PAN}, \mathrm{PIN}|\!\}_k$. Since every account, identified by the PAN, has its own PIN, the PIN can be thought of as at level $[\mathsf{PIN} : \mathsf{PAN}]$ ('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 $\mathsf{CloseDD}^{\mathsf{det}}(\tau)$ which intuitively ensures that the ciphertext is completely determined by the included integrity representative (e.g., the PAN), playing the role of confounder. As in (enc-r) integrity is propagated and confidentiality of the ciphertext is L.

Rule (mac) is for the generation of MACs. Here, 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. The reason why we force integrity to be low is technical and, more specifically, is 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 $\mathsf{declassify}(e)$ for explicitly declassifying the confidentiality level of an expression e to L. This new expression has no operational import, i.e., $\mathsf{declassify}(e) \downarrow^{\mathsf{M}} v$ iff $e \downarrow^{\mathsf{M}} v$. Declassification is thus only

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

Judgments for commands have the form $\Delta, pc \vdash \mathsf{c}$ 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. In Table 2 we report the rule for declassification plus the only two that differ from [23].

The first rule 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. Assignments (second rule) are only possible at or above the pc level and at lower integrity levels (dependent domains) if $\mathcal{L}_I(pc) = H$. This makes sense since we never move our observation level below LH and is achieved by requiring $pc \sqsubseteq \mathcal{L}(\tau) \sqcup LH$.

The third 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 $L[\mathsf{D}]$; the second parameter e and the MAC value e' are typed LL. If the MAC succeeds, variable y of type τ 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 τ (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, guaranteeing data integrity and allowing us to 'reconstruct' their type from the type of the MAC key.

Side conditions $\mathsf{IRs}(L[\mathsf{D}], \tau) = \{\mathsf{D}\}$ and $\mathsf{CloseDD}(\mathsf{L}[\mathsf{D}], \tau)$ ensure that the MAC contains only values which directly depend 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 $\perp_{\mathsf{MAC}}$ which just aims at causing non-termination of the program (it may be equivalently thought of as a command with no semantics, which never reduces, or a diverging program as, e.g., while true do skip). This is needed to prevent the attacker from breaking integrity and robustness by just calling an API with incorrect MACs. In fact, we can assume the attacker knows which MACs pass the tests and which do not (unless he is trying brute-force/cryptanalysis attacks 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 now prove that well-typed programs satisfy robustness and, in case they do not declassify any information, noninterference. 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 appear 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 $[\mathsf{D} : \tilde{\mathsf{D}}]$ are uniquely determined by the values of the integrity representatives $\tilde{\mathsf{D}}$, 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 $[\mathsf{PAN}]$ and the relative PIN, of level $[\mathsf{PIN} : \mathsf{PAN}]$, encrypted with another key, all of the possible MACs with that key will respect a function $f_{[\mathsf{PIN:PAN}]}$, pre-established for each memory. For example, let us assume $f_{[\mathsf{PIN:PAN}]}(pan_i) = pin_i$. We have that all of these MACs are well-formed: $\langle pan_1, \{\!|pin_1|\!\}_k \rangle_{k'}$, $\langle pan_2, \{\!|pin_2|\!\}_k \rangle_{k'}$, $\ldots$, $\langle pan_m, \{\!|pin_m|\!\}_k \rangle_{k'}$, as they all respect $f_{[\mathsf{PIN:PAN}]}$.

Our first result states that a well-typed program run on well-formed memory, noted $\Delta \vdash \mathsf{M}$, always returns a well-formed memory:

Proposition 1 *If* $\Delta, pc \vdash \mathsf{c}$, $\Delta \vdash \mathsf{M}$ *and* $\langle \mathsf{M}, \mathsf{c} \rangle \Rightarrow \mathsf{M}'$ *then* $\Delta \vdash \mathsf{M}'$.

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 $\perp_{\mathsf{MAC}}$ 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 (uninteresting) situation.

The next lemmas are used to prove the main results. The first one is peculiar to our extension with cryptography: if an expression is typed below the observation level ℓ, 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 ciphertexts might be distinguished once compared with others.

Lemma 1 (Expression ℓ-equivalence) *Let* $\mathsf{M}_1 \approx_\ell \mathsf{M}_2$ *and let* $\Delta \vdash e : \tau$ *and* $e \downarrow^{\mathsf{M}_i} v_i$. *If* $\mathcal{L}(\tau) \sqsubseteq \ell$ *or* $\mathcal{L}(\Delta(x)) \not\sqsubseteq \ell$ *then* $\mathsf{M}_1[x \mapsto v_i] \approx_\ell \mathsf{M}_2[x \mapsto v_i]$.

Lemma 2 (**Confinement**) *If* $\Delta, pc \vdash \mathsf{c}$ *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* $\Delta, pc \vdash \mathsf{c}$ *then* c *satisfies noninterference.*[4]

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, i.e., $x_1 := \mathsf{declassify_1}(e_1), \ldots, x_m := \mathsf{declassify_m}(e_m)$, and

[4] For technical reasons this results does not hold for level LH (see [14] for details).

avoiding placing declassifications inside while loops. These special variables are only assigned here. 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.

Theorem 2 (Robustness) $\mathsf{c} \in \mathrm{CD}$ *and* $\Delta, pc \vdash \mathsf{c}$ *imply* c *satisfies robustness.*

5 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 3, 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. The new API initially checks a MAC of all the parameters. 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 and the correct MAC validating their integrity. This is actually captured by our notion of dependent domains by typing all the MAC checked variables as dependent on the PAN.

```
PIN_V_M(PAN,EPB,len,offset,vdata,dectab,MAC){
 if (mac_ak(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"; ⊥_MAC}
```

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 has type $\mathsf{mK}_{H\mathsf{C}}(L[\mathsf{PAN}], \tau)$ with type $\tau = \mathsf{enc}_{L[\bullet:\mathsf{PAN}]}\ \kappa_{ek},\ L[\mathsf{LEN}:\mathsf{PAN}], L[\mathsf{OFFS}:\mathsf{PAN}],\ \mathsf{enc}_{L[\bullet:\mathsf{PAN}]}\ \kappa_{pdk},$ $L[\mathsf{DECTAB}:\mathsf{PAN}]$. Note that $\mathsf{IRs}(L[\mathsf{PAN}], \tau) = \{\mathsf{PAN}\}$ and $\mathsf{CloseDD}(L[\mathsf{PAN}], \tau)$, meaning that $L[\mathsf{PAN}]$ and τ are all domains which only depends on representative PAN. All the checked variables are typed according to the above tuple, e.g., PAN' with $L[\mathsf{PAN}]$, EPB' with $\mathsf{enc}_{L[\bullet:\mathsf{PAN}]}\ \kappa_{ek}$ and so on. Key ek is typed as $\mathsf{cK}^R_{H\mathsf{C}}(H[\mathsf{PIN}:\mathsf{PAN}])\ \kappa_{ek}$ and key pdk as $\mathsf{cK}^R_{H\mathsf{C}}(H[\mathsf{HEX}:\mathsf{PAN}])\ \kappa_{pdk}$. The result of the API will be stored in the ret variable whose type is LL.

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 amounts to checking the original API with the new high integrity types. What happens is that x_1 is typed $H[\mathsf{HEX}:\mathsf{PAN}]$ by rule (dec-μ) and $x_2, \ldots, x_4$ are typed $H[\bullet:\mathsf{PAN}]$ as results of arithmetic operations. x_6 (which is modelled as $\mathsf{dec}^R_k(EPB)$) is typed $H[\mathsf{PIN}:\mathsf{PAN}]$ by rule (dec-μ). Thus, $x_7 := \mathsf{declassify}(x_4 = x_6)$, which we explicitly add to the code, can be typed LH as $x_4 = x_6$ types $H[\bullet:\mathsf{PAN}] \leq HH$. Theorem 2 guarantees that 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 This API is used to decrypt and re-encrypt a PIN under a different key and, possibly, a different format. In [14] we specify a MAC-based

extension of the API for specifically translate from ISO-1 to ISO-0 and we type-check it. ISO-0 is not randomized and pads the PIN with data derived from the PAN. We thus use our (enc-d) typing rule to prove its security.

6 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, 16, 20, 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 Jeffreys' 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 [17]. 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 [19], 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, 21, 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 [15].

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`.
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. *JACM*, 46(5):749–786, 1999.
4. M. Abadi and J. Jurjens. Formal eavesdropping and its computational interpretation. In *TACS'01*, volume 2215 of *LNCS*, pages 82–94, October 29-31 2001.
5. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *JCRYPTOL*, 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 *ESORICS'05,* vol. 3679, LNCS, pages 374–396, 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. Ostrovsky. The unbearable lightness of PIN cracking. In *Financial Cryptography and Data Security* vol. 4886, LNCS, pages 224–238, 2007.
9. M. Bond and P. Zielinski. Decimalization table attacks for PIN cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory, 2003.
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 noninterference: The case of deterministic encryption. In *Found. of Software Technology and Theoretical Computer Science* vol. 4855, pages 364–375. Springer, 2007.
12. S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *IEEE Computer Security Foundations Symposium*, pages 331–344, June 23-25 2008.
13. R. Focardi and M. Centenaro. Information flow security of multi-threaded distributed programs. In *ACM SIGPLAN PLAS'08*, pages 113–124, June 8 2008.
14. R. Focardi, M. Centenaro, F. Luccio, and G. Steel. Type-based analysis of PIN processing APIs (full version). Technical Report CS-2009-6, Università Ca' Foscari, Venezia, Italy, 2009. `http://www.unive.it/nqcontent.cfm?a_id=5144`.
15. R. Focardi, F. Luccio, and G. Steel. Improving pin processing api security. In *Workshop on Analysis of Security APIs*, July 10-11 2009, to appear.
16. C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *POPL'08*, pages 323–335. ACM Press, 2008.
17. A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. Technical Report MSR-2001-49, Microsoft Research, 2001.
18. I. Inc. CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors. Technical report, 2006. Rel. 2.53–3.27.
19. G. Keighren, A. Aspinall, and G. Steel. Towards a type system for security APIs. In *ARSPA-WITS'09*, pages 173–192, York, UK, March 28-29 2009.
20. P. Laud. On the computational soundness of cryptographically masked flows. In *POPL'08*, pages 337–348. ACM Press, January 10-12 2008.
21. D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.
22. M. Mannan and P. van Oorschot. Reducing threats from flawed security APIs: The banking PIN case. *Computers & Security*, 28(6):410–420, September 2009.
23. A. 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. 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. *TCS*, 367(1-2):257–270, 2006.
27. J. 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.