Type-based Analysis of Financial APIs [⋆] (extended abstract)

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

[1] Università Ca' Foscari Venezia,
{mcentena,focardi,luccio}@dsi.unive.it
[2] LSV, ENS Cachan & CNRS & INRIA, France
graham.steel@lsv.ens-cachan.fr

Abstract. We revise a known attack on the PIN verification framework, based on a weakness of the underlying security API. We specify this flawed API in an imperative language with cryptographic primitives and we show why its type-based verification fails in the type system of Myers, Sabelfeld and Zdancewic. We propose an improved API, extend the type system with cryptographic primitives for assuring integrity, and show our new API to be type-checkable. (for presentation only)

Keywords: Language-based Security, Security APIs, Financial Cryptography, PIN Verification.

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. Issuing banks cannot expect to securely share secret keys with every cash machine, and so the PIN is encrypted under various different keys as it passes through the network. Typically, it will first be 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 a switch adjacent to 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 internationally agreed standards ANSI X9.8 and ISO 9564 stipulate the

[⋆] Work partially supported by Miur'07 Project SOFT: "*Security Oriented Formal Techniques*". The full version of this extended abstract is available at [5].

use of tamper proof cryptographic hardware security modules (HSMs). In the switches, these HSMs protect the PIN encryption keys, while in the issuing banks, they also protect the PIN derivation keys (PDKs) used to derive the customers PIN from non-secret validation data such as their personal account number (PANs). All encryption, decryption and checking of PINs is carried out inside the HSMs. To this aim, the HSMs have a carefully designed API providing functions for *translation* (i.e., decryption under one key and encryption under another) 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 customer PINs. More specifically, we assume the attacker can program the host machine, running software and controlling API calls, but he cannot physically access the device to do, e.g., power analysis.

In the last few years, several attacks have been published on the APIs in use in these systems [2, 3, 6, 9]. 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 a hacked switch has increased interest in the problem [1]. 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 [12]. Other researchers have proposed improvements to the system to blunt the attacks [10], but these suggestions address only some attacks, and are "intended to stimulate further research" [10, p. 5]. We take a step in that direction here, proposing a language-based security framework whereby possible improvements to PIN processing APIs may be formally analysed, and suggesting some improvements of our own.

We do not describe the operation of the ATM network in detail. Interested readers are referred to existing literature [6, 10, 12]. In this paper, we will go straight into a case study on the PIN verification command (section 2), showing how it can be attacked, how it could be improved, and how these improvements could be formally analysed. We conclude with a discussion of next steps in section 3.

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

Table 1 The PIN verification API.

```
PIN_V(pan,epb,len,offset,vdata,dectab) {
  // deriving user PIN with IBM 3624 PIN calculation method
  x1 = encrypt(pdk, vdata);       // encrypts vdata with PDK
  x2 = left(len, x1);             // takes len leftmost digits
  x3 = dectab(x2);                // decimalizes
  x4 = sum_mod10(x3,offset);      // sums the offset

  // recovering the trial PIN
  x5 = decrypt(k, epb);           // decrypts the EPB with k
  x6 = f_check(x5,pan);           // extracts formatted PIN
  if (x6 == FAIL)
    return("format error");       // format was wrong

  // if passes the check, x6 contains the trial PIN

  // checks the trial PIN versus the actual user PIN
  if (x4 == x6)
    return("PIN is correct");
  else
    return("PIN is wrong");
}
```

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 that we call `PIN_V` and specify as in Table 1.

`PIN_V` 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` (described below), while the latter comes encrypted under key `k` as *Encrypted PIN Block* (EPB), passed as parameter `epb`. Note that the two keys are pre-loaded in the HSM and are never exposed to the untrusted external environment. In general, the HSM may have to manage several encryptions and PIN derivation keys, but for the purposes of this example we will assume only one key of each type is used. More specifically, the API behaves as follows:

– the user PIN of length `len` is obtained by encrypting validation data `vdata` with key `pdk` (`x1`), getting the first `len` hexadecimal digits (`x2`), decimalising through `dectab` (`x3`), and digit-wise summing modulo 10

[3] The value of this parameter is up to the issuing bank. It is usually a combination of the user PAN with other public data, such as the card expiration date or the customer name.

the `offset` (x4). More precisely, the outcome of the encryption `x1` is a 16 hexadecimal digit string and `dectab` is a function that associates to each possible hexadecimal digit a decimal one. The obtained decimalized value `x3` is the 'natural' PIN assigned by the issuing bank to the user. Whenever 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 user natural PIN, giving `x4`.

- the trial PIN is recovered by decrypting `epb` with `k` (x5), and by checking the format (x6). This last operation is more elaborated than a simple check: once the format is recognised, the PIN is 'extracted' from the formatted 16-digit hexadecimal number; for some formats the *Personal Account Number* (PAN), passed as parameter `pan`, is required for the extraction. For the analysis we develop here, we do not require any more detail on formats. Interested readers are referred to the literature [6, 8].
- the equality of the user PIN (`x4`) and the trial PIN (`x6`) is returned.

We now present a numerical example to illustrate the `PIN_V` execution.

Example 1. Let `len=4`, `offset=4732`, `dectab=9753108642543210`, this last parameter encoding the following mapping:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
9 7 5 3 1 0 8 6 4 2 5 4 3 2 1 0
```

Let also `x1 = encrypt(pdk,vdata) = A47295FDE32A48B1`. Then,

```
|x2 = left(4, A47295FDE32A48B1)|= A472|
|x3 = dectab(A472)             |= 5165|
|x4 = sum_mod10(5165,4732)     |= 9897|
```

This completes the user PIN recovery part. Let now `f_9897` denote PIN `9897` correctly formatted (recall that we are omitting details about PIN formats) and let us assume that `epb = E`$_\mathtt{k}$`(f_9897)`. We thus have:

```
|x5 = decrypt(k, Ek(f_9897)) |= f_9897|
|x6 = f_check(f_9897, pan)   |= 9897  |
```

Since `x6` is different from `FAIL` and `x4=x6` the API returns `"PIN is correct"` and this completes the example.

The given specification is an abstraction and a simplification of real ones. In particular, `PIN_V` corresponds to `Encrypted_PIN_Verify` of [8] simplified by omitting some parameters for alternative PIN extraction methods

$$\begin{tikzcd}
 & \mathrm{HL} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathrm{HH} \arrow[dr, dash] & & \mathrm{LL} \arrow[dl, dash] \\
 & \mathrm{LH} &
\end{tikzcd}$$

Fig. 1. Confidentiality/Integrity security lattice

and, as mentioned above, some details on formats and key management. Notice that we only model the IBM 3624 PIN calculation method with offset; however this is not limiting as the other PIN calculation methods can be similarly specified and analysed.

2.1 Typing: a first unsuccessful attempt

In order to analyse the security of the previously described API we try to adopt the type-based approach proposed by Myers, Sabelfeld and Zdancewic (MSZ) [11]. The idea is to have two distinct security levels ℓ_C and ℓ_I, the former for *confidentiality* and the latter for *integrity*. For our case study we can limit our attention to two 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 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 low-integrity data must be treated more carefully than high-integrity ones, giving the counter-variant relation $H \sqsubseteq_I L$. The lattice corresponding to the product of the above described confidentiality and integrity lattices is depicted in Figure 1.

The property proposed by MSZ, called *robust declassification*, aims at verifying that attackers cannot influence the secret information downgraded or *declassified* by a program C. In our case, `PIN_V` returns the correctness of the PIN typed which is a one-bit information about a secret datum. Thus, the API is intended to declassify some secret information, but we would like to check that attackers cannot abuse such a declassification and gain more information than intended. The core idea of the MSZ type-system is to check that declassification only occurs on high integrity variables and in high integrity contexts, so to ensure that the attacker cannot manipulate *what* is declassified and *when* declassification happens. In such a case, declassification is proved to be *robust*.

We now assign each parameter and variable a security level. Intuitively, what happens 'outside' the API is considered untrusted, thus each parameter can be modeled as a global variable[4] of level LL; internal variables are in general highly-confidential: a security API may be thought as a black-box taking some parameters and returning some results, without revealing intermediate calculations. In particular, notice that all variables `x1`, …, `x6` contain sensitive information that should never be leaked.

As far as integrity is concerned, we have that variables `x1`, …, `x4` are all low-integrity, since they are calculated starting from the parameters and through operations that do not provide any integrity check. Variables `x5` and `x6` are low-integrity, too: the decryption might, in principle, perform an integrity check, guaranteeing that the obtained value was indeed encrypted with the expected, trusted, key and thus not manipulated by the attacker, but in this application there is no such a feature.

The format check performed for `x6` might increase the confidence in the integrity of the value in `x5`: if the format is recognized this should give evidence that the decryption was successful. Unfortunately, formats have not been designed to give integrity guarantees and there are well documented attacks based on format confusion [6, §3.5.4]. Moreover, declassifying information about the correctness of the format of `x5`, which is a highly confidential variable, should be robust but this is not the case, given the low integrity of the same variable and of the other parameter (`pan`): there are, in fact, known attacks based on PAN manipulation that allow an attacker to deduce information about the PIN digits [6, §3.5.3].

Finally, given that x4 and x6 are high-confidentiality but low-integrity, the declassification of their equality is not robust. In the next section, we show a concrete example of attack based on the above mentioned lack of integrity.

2.2 Attacking the `PIN_V` API

In this section we show how the lack of integrity presented in section 2.1 is exploited to mount a real attack on the `PIN_V` API.

As we have previously mentioned, all the parameters of the `PIN_V` API do not provide any integrity check. Let us now concentrate on two specific ones, the `dectab` and the `offset`, which are used to calculate the values of `x3` and `x4`, respectively. A possible attack on the system works by iterating the following two steps, until the whole PIN is recovered [3]:

1. The attacker picks a decimal digit d, changes the `dectab` function so that values previously mapped to d now map to $d + 1 \bmod 10$,

[4] This makes sense as the sematics adopted here and by MSZ is single-threaded, so global variables are not going to be changed in between API computations.

and then checks whether the system still returns `"PIN is correct"`. Depending on this, the attacker discovers whether or not digit d is present in the user 'natural' PIN contained in `x3`, 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 attacker also changes the `offset` until the API returns again that the PIN is correct. This allows the attacker to locate the position of the deduced PIN digit.

We illustrate the attack through a simple example.

Example 2. Recall that in Example 1 we assumed `len=4`, `offset=4732`, `dectab=9753108642543210`, `x1 = A47295FDE32A48B1`, `EPB = E`$_k$`(f_9897)`. With these parameters the API returns `"PIN is correct"`.

Assume the attacker chooses the new `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 `x3`. If we invoke the API with `dectab'` we obtain the same intermediate and final values, since `dectab'(A472) = dectab(A472) = 5165`. This means that 0 does not appear in `x3`.

The attacker now proceeds by removing from the `dectab` the next decimal digit until the API fails: with `dectab''=9753`$\underline{2}$`086425432`$\underline{2}$`0`, i.e., by replacing digit 1 with digit 2, we obtain that `dectab(A472) = 5165` $\neq$ `5265 = dectab''(A472)`, reflecting the presence of 1 in the original value of `x3`. Then, `x4=9997` instead of `9897` thus returning `"PIN is wrong"`.

The attacker now knows that digit 1 occurs is in `x3`. To discover its position and multiplicity, he now tries variations of the offset so to 'compensate' 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 is correct"`. In fact:

`x3 = dectab(A472)`	`= 5165`	`x3 =` $\underline{\texttt{dectab''}}$`(A472)`	`= 5`$\underline{2}$`65`
`x4 = sum_mod10(5165,4732)`	`= 9897`	`x4 = sum_mod10(5`$\underline{2}$`65,4`$\underline{6}$`32)`	`= 9897`

Notice, in particular, that the value of `x4` is the same. The attacker now knows that the second digit of `x3` is 1. Given that the `offset` is public, he also calculates the second digit of the user PIN as $1 + 7 \bmod 10 = 8$.

2.3 MAC-based integrity: towards a type-checkable API

In this section we discuss a MAC-based improvement of `PIN_V`, called `PIN_V_M`, which prevents the previously discussed attack, and several oth-

Table 2 The new API with MAC-based integrity.

```
PIN_V_M(pan,epb,len,offset,vdata,dectab,MAC) {
  // checking the MAC
  if ( mac(ak,pan,epb,len,offset,vdata,dectab) == MAC )
    // if integrity test is passed, invokes the original API
    return(PIN_V(pan,epb,len,offset,vdata,dectab));
  else
    return("integrity violation"); // MACs do not correspond
}
```

ers from the literature. We claim PIN_V_M is type-checkable using an extension with cryptography of the MSZ type-system. However, although type-checkable, the proposed API is not completely satisfactory for different reasons that will be discussed in the next section.

To simplify the analysis, we do not consider the cases in which users, by mistake, type a wrong PIN. This is because 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, meaning that he has influenced the declassification of data. We do not try to capture this attacker behaviour in our first attempt at defining a type system. Instead, we assume the attacker has only one EPB containing the correct PIN. This is of course strictly more secure than the real situation, but we will show how security in this model eliminates a wide class of attacks.

We consider a new *authentication key* ak. Our fix contains a unique MAC of all the parameters which is checked at the very beginning. The code is reported in Table 2. Intuitively, the MAC check gives guarantees about the fact that the parameters have not been arbitrarily manipulated. However, care is needed since some form of 'legal' manipulation is always possible: an attacker 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. This cannot be prevented, as those parameters will have a correct MAC validating their integrity. So, what the MAC prevents is changing a *subset* of the parameters.

Dependent integrity types 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 $\mathsf{D} \in \mathit{Dom}$ 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

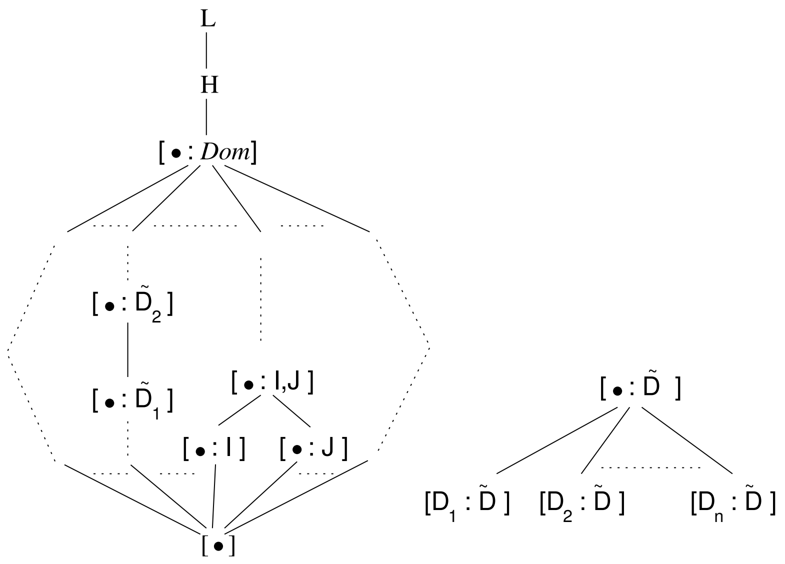

Fig. 2. New integrity levels: $\tilde{\mathsf{D}}_1 \subseteq \tilde{\mathsf{D}}_2$ and $Dom = \{\mathsf{D}_1, \ldots, \mathsf{D}_n\}$.

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 $\mathsf{D} : \emptyset$, 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. We assume only one integrity representative for each domain.

For example, a natural integrity representative candidate is the PAN, given that it uniquely identifies the user account and, as a consequence, all the user data `len, offset, vdata, dectab`. We can say that, after the MAC has been checked, the values of the former parameters are promoted to the four dependent domains LEN : PAN, OFF : PAN, VD : PAN and DEC : PAN, meaning that their integrity depend on PAN integrity.

The integrity level associated to a dependent domain $\mathsf{D} : \tilde{\mathsf{D}}$ is written $[\mathsf{D} : \tilde{\mathsf{D}}]$, and is at a higher integrity level 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 resulting domain and we only record the fact the value is determined by domains $\tilde{\mathsf{D}}$, written $[\bullet : \tilde{\mathsf{D}}]$. In this case we know the value is determined by at most variables of domains $\tilde{\mathsf{D}}$, but we have no precise information on its domain. For example, if we perform an arithmetic operation on `len` and `offset` the result would be of level $[\bullet : \mathsf{PAN}]$. We have the following ordering of integrity levels:

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

with $\tilde{\mathsf{D}}_1 \subseteq \tilde{\mathsf{D}}_2$. Intuitively, $[\bullet : \tilde{\mathsf{D}}_1]$ is less restrictive than $[\bullet : \tilde{\mathsf{D}}_2]$ given that the fewer are the dependencies, the fewer are the integrity representatives needed inside the MAC to check the integrity of the corresponding value. For example, $[\bullet : \mathsf{I}, \mathsf{J}]$ requires that both integrity representatives of level $[\mathsf{I}]$ and $[\mathsf{J}]$ are in the MAC to check the integrity of the corresponding value. We obtain the lattice of Figure 2.

Promotion to these high integrity types is achieved by adding new assignments, guarded by the MAC check, to fresh variables whose types $\tau_1, \ldots, \tau_n$ are derived from the MAC key type $\mathsf{mK}([\mathsf{D}], \tau_1, \ldots, \tau_n)$. In particular, $[\mathsf{D}]$ is the integrity representative and $\tau_1, \ldots, \tau_n$ are required to depend at most on $[\mathsf{D}]$ like, e.g., in $\mathsf{mK}([\mathsf{PAN}], [\mathsf{OFF} : \mathsf{PAN}])$ (notice that from now on we only show integrity types and omit confidentiality ones). More formally, our typing rule for the MAC has the following shape:

$$\frac{\begin{array}{c}\Gamma \vdash k : \mathsf{mK}([\mathsf{D}], \tau_1, \ldots, \tau_n) \qquad \Gamma \vdash \mathtt{v} : [\mathsf{D}] \\ \Gamma \vdash \mathtt{e}_1, \ldots, \mathtt{e}_n, \mathtt{e} : \mathrm{L} \qquad \Gamma, \mathtt{z}_1 : \tau_1, \ldots, \mathtt{z}_n : \tau_n \vdash \mathtt{C}_1 \qquad \Gamma \vdash \mathtt{C}_2\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{if}\ (\ \mathtt{mac}(\mathtt{k}, \mathtt{v}, \mathtt{e}_1, \ldots, \mathtt{e}_n) == \mathtt{e}\ ) \\ \qquad \{\mathtt{z}_1 = \mathtt{e}_1; \ldots \mathtt{z}_n = \mathtt{e}_n; \mathtt{C}_1\}\ \ \textbf{else}\ \ \mathtt{C}_2\end{array}}$$

Intuitively, checking MAC e, with integrity representative $\mathtt{v}$, allows a programmer to assign low integrity expressions $e_1 \ldots e_n$ to high integrity fresh variables $\mathtt{z}_1 \ldots \mathtt{z}_n$ of type $\tau_1, \ldots, \tau_n$, as specified by the MAC key type. Notice that L is the top integrity type (see Figure 2), thus every well-typed expression can be given such a type via standard subtyping.

Table 3 reports the code of the new API with type declarations and high integrity variables. More specifically, we have:

$\Gamma \vdash \mathtt{epb,len,offset,vdata,dectab,MAC} : \mathrm{L}$
$\Gamma \vdash \mathtt{pan} : [\mathsf{PAN}]$
$\Gamma \vdash \mathtt{ak} : \mathsf{mK}([\mathsf{PAN}], \mathsf{enc}^1_{[\bullet:\mathsf{PAN}]}, [\mathsf{LEN{:}PAN}], [\mathsf{OFF{:}PAN}], \mathsf{enc}^2_{[\bullet:\mathsf{PAN}]}, [\mathsf{DEC{:}PAN}])$

Type $\mathsf{enc}^1_{[\bullet:\mathsf{PAN}]}$ represents a ciphertext of level $[\bullet{:}\mathsf{PAN}]$ encrypted using a key of type $\mathsf{cK}^1([\mathsf{PIN{:}PAN}], ...)$, i.e., the encryption key $\mathtt{k}$. Notice that the superscript 1 links the types enc and cK and that the type of the key is not completely specified as we are omitting details about EPB formats. The other encrypted term is `vdata`. In fact, in order to model PIN derivation we adopt a small trick: we write $\mathtt{vdata} = \{n\}_{pdk}$, where n is the expected results of the encryption of `vdata`, which can now be modelled as a decryption `x1 = decrypt(pdk, vdata)`. The reason

Table 3 The new API with type declarations and high integrity variables.

```
PIN_V_M(pan,epb,len,offset,vdata,dectab,MAC) {
  // types of parameters
  L   epb, len, offset, vdata, dectab, MAC;
  // the integrity representative is of type [PAN]
  [PAN] pan;

  // authentication keys type
  mK([PAN], enc^1_[*:PAN], [LEN:PAN], [OFF:PAN],
      enc^2_[*:PAN], [DEC:PAN]) ak;

  // encryption keys
  cK_1([PIN:PAN], ... )   k;
  cK_2([HEX:PAN])         pdk;

  // checking the MAC
  if ( mac(ak,pan,epb,len,offset,vdata,dectab) == MAC ) {
    epb'=epb; l'=len; offset'=offset; vdata'=vdata;
    dectab'=dectab;
    // invokes the original API with high integrity variables
    return(PIN_V(pan,epb',l',offset',vdata',dectab'));
  } else
    return("integrity violation"); // MACs do not correspond
}
```

for this is that we have a symbolic model for encryption that does not produce any low level bit-string encrypted data.

Now, if

$$\begin{array}{rl} & \Gamma, \\ \texttt{epb'} : & \mathsf{enc}^1_{[\bullet:\mathsf{PAN}]}, \\ \texttt{l'} : & [\mathsf{LEN}:\mathsf{PAN}], \\ \texttt{offset'} : & [\mathsf{OFF}:\mathsf{PAN}] \\ \texttt{vdata'} : & \mathsf{enc}^2_{[\bullet:\mathsf{PAN}]}, \\ \texttt{dectab'} : & [\mathsf{DEC}:\mathsf{PAN}], \\ & \vdash \texttt{return(PIN\_V(pan,epb',l',offset',vdata',dectab'));} \\ & \Gamma \vdash \texttt{return("integrity violation");} \end{array}$$

Then

```
Γ ⊢ if ( mac(ak,pan,epb,len,offset,vdata,dectab) == MAC )
        epb'=epb; l'=len; offset'=offset; vdata'=vdata;
        dectab'=dectab;
        return(PIN_V(pan,epb',l',offset',vdata',dectab'));
    } else
        return("integrity violation");
```

In order to type the two `return` commands we observe the following:

1. `return("integrity violation")` is easily typed as it outputs a low confidentiality value and the guard of the if statement is low-confidentiality too. Low confidentiality outputs are, in fact, critical if they are performed inside if statements with high-confidentiality guards as, e.g.,

   ```
   if (h==h') then
     return(true);
   else
     return(false);
   ```

   Here, the MAC values are low-confidentiality, thus there is no security restriction on low-level outputs.
2. `return(PIN_V(pan,epb',l',offset',vdata',dectab'))` is more interesting, as it calls the original API in which the integrity level of the parameters has been rised as explained above. Given that all of the calculations of variables `x1` ... `x6` are done starting from [D : PAN] variables, by subtyping (Figure 2) they can all be typed at integrity level [•: PAN]. As a conclusion, the equality of `x4` and `x6` is calculated between [• : PAN] variables and we can thus say its declassification is *robust with respect to the* PAN.

Robustness with respect to integrity representatives We finally discuss the role of integrity representatives in the robustness notion and, thus, in the security property guaranteed by the typed API. We have explained that integrity representatives never change their values at runtime. This is fundamental to avoid that their value is changed after a MAC-based integrity check. Consider, for example, the case in which the PAN is changed after the MAC has been checked. If this happens, all of the variables at level [•: PAN] would loose their high integrity nature with respect to the integrity representative, which has been modified. More formally, the original notion of robustness of MSZ can be expressed as follows: Program C has robustness if $\forall M_1, M_2, A, A'$ we have

$$\langle M_1, A[C]\rangle \simeq \langle M_2, A[C]\rangle \Rightarrow \langle M_1, A'[C]\rangle \simeq \langle M_2, A'[C]\rangle$$

where M_1 and M_2 are two memories, A and A' are two *fair* attackers, i.e., attackers that do not violate integrity and confidentiality directly,[5] $\langle M, A[C]\rangle$ denotes the execution of program C under attack A starting from memory M. The property expresses the fact that 'what cannot be leaked to one attacker, cannot be leaked to any other attacker'. In other words, attackers cannot influence what is being declassified by program

[5] These attackers should not read high-confidentiality or write high-integrity variables.

C and when declassification happens. This is a slight simplification of the original notion where, for technical reasons that we do not discuss here, two different behavioural equivalences are adopted. Moreover, MSZ admit an attacker A to be placed in between the trusted code C, noted $C[A]$. In our setting, the attacker can only act before and after the API execution, which we think is better represented by the 'dual' notion $A[C]$.

More importantly, here we require that integrity representatives have the same values in the two memories M_1 and M_2:

$$\forall \text{ variable } \mathtt{v}, \Gamma \vdash \mathtt{v} : [\mathsf{D}] \Rightarrow M_1[\mathtt{v}] = M_2[\mathtt{v}].$$

Moreover, we require that programs and attackers cannot change values of those variables. This is already included in the notion of fair attackers which requires that all the assignments are done to low integrity variables. As far as APIs are concerned, well-typed ones will only perform assignments to high-integrity variables, apart from the special case of MAC checking discussed above. So, typing will automatically avoid MAC-unguarded assignments to variables with an integrity level lower than H, such as $[\mathsf{D}]$ or $[\mathsf{D} : \tilde{\mathsf{D}}]$ variables. Finally, memories should respect integrity types. For example, if we have a MAC key `k` of type $\mathsf{mK}([\mathsf{D}'], [\mathsf{D} : \mathsf{D}'])$, a well-formed memory should never contain two MACs such as `mac(k,m,n)` and `mac(k,m,n')`, with $\mathtt{n} \neq \mathtt{n'}$, since once `m` (of level $[\mathsf{D}']$) is fixed also the last message, of level $[\mathsf{D} : \mathsf{D}']$, should be fixed.

To summarize, our robustness with respect to integrity representatives corresponds to the original notion of robustness in which integrity representatives have the same initial, and consequently run-time, values. Attacks in which those variables are changed are thus not captured by our property. Note however that the attack described in section 2.2 is not based on changing the `pan` and is thus revealed by our robustness notion. It is sufficient to consider a memory M_1 with the values of all the parameters as in Example 2 (recall that we model parameters as global variables) and another memory M_2 which differs from M_1 for the value of the PIN encrypted in the `epb`. Now, consider an attacker A that changes the `dectab` so to make verifications on both memories fail. We have that $\langle M_1, A[C] \rangle \simeq \langle M_2, A[C] \rangle$. Now, if we consider an 'empty' attacker A' which does nothing, we have that verification on M_1 succeeds while verification on M_2 fails, given the `epb` contains a different PIN. Thus, $\langle M_1, A'[C] \rangle \not\simeq \langle M_2, A'[C] \rangle$. Notice that this happens with the same constant value for `pan` in the two memories: in fact the attack is not based on changing such a value. On the other hand, the attack is provably not present in the `PIN_V_M` API which we have (informally) shown to be robust with respect to `pan`. In fact, the change in the `epb` is immediately detected by the MAC check thus making $\langle M_1, A[C]\rangle \not\approx \langle M_2, A[C]\rangle$.

3 Discussion and Conclusions

We have presented an extension of the language based security framework with cryptographic primitives for ensuring integrity of data. We have shown how such an analysis could be applied to the improvements we suggest to the PIN processing API used in the ATM network. We briefly discuss in a number of open issues.

We have presented the case study of the verification API. There are other critical APIs to be analysed in the PIN management protocol, such as the one used to translate the EPBs into different formats when passing through the switches. The integrity types presented here can be also used to check the translate API, since once the MAC for the incoming values has been checked, our types give enough information to create the new MAC for the outgoing, reformatted, message [5]. This MAC check would prevent some known translate attacks [6, §3.5.3].

The fix proposed here has some limitations: (*i*) it is based on a unique MAC which is not implementable in practice, since part of the data needs to be authenticated by the switch forwarding the PIN via the translation API discussed above. We are studying a solution with two MACs that we claim to be implementable with not much overhead with respect to the existing protocols [7]; (*ii*) we have abstracted away from the PIN format and the presence, in the HSMs, of different encryption keys, one for each incoming switch. We are studying how the extend types so to deal with these aspects. Our idea is to use tagged unions so to have the actual format being used as a tag for choosing among different types for the encryption keys; this would allows for different formats circulating under the same encryption key.

We have assumed that the user will never insert the wrong PIN at the ATM, as this would by itself break robustness. It would be important to remove this assumption and we are investigating the possibility of picking the `EPB` as integrity representative, instead of `PAN`. This makes sense only if collisions on EPBs happen with negligible probability, as having equals EPBs referring to different data would make it impossible to type those data as $[\mathsf{D} : \mathsf{EPB}]$. Such a type requires that data are fixed once `EPB` is fixed, which would not be the case if collisions are considered. This solution requires a careful analysis of PIN formats: there are PIN formats with no randomized padding which produce colliding EPBs with high probability and, in some cases, just when the PIN is the same. The standards forbid these formats for online PIN verification, but they are often still supported by HSM APIs for local use, and so could in theory be exploited by an attacker. For lack of space, in this paper we have mainly focussed on integrity issues. However, it is worth noticing, that the above mentioned analysis of PIN formats is also important for being able to type `EPB`s as low-confidentiality. In fact, poor formats might by themselves compromise the confidentiality of encrypted PINs.

In the full version of this paper [5], we have extended in a non-trivial way the semantic model of MSZ with cryptography by adopting some known techniques [4]. In particular, we have extended the notion of low-equivalent memories in the presence of MACs and, in general, any cryptographic operation which is not randomized, e.g., via confounders. In fact, when cryptography is assumed to be randomized, the underlying notion of low-equivalence is simplified via the assumption that a new encryption is different from any other previous encryption. Finally, we have formalized the new dependent integrity types described in this paper. To the best of our knowledge, in the literature there are no similar type-based analyses of integrity, confidentiality and declassification in the same setting.

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. O. Berkman and O. M. Ostrovsky. The unbearable lightness of PIN cracking. In Springer LNCS vol.4886/2008, editor, *11th International Conference, Financial Cryptography and Data Security (FC 2007), Scarborough, Trinidad and Tobago*, pages 224–238, February 12-16 2007.
3. M. Bond and P. Zielinski. Decimalization table attacks for PIN cracking, 2003. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-560.pdf.
4. M. Centenaro and R. Focardi. Information flow security of multi-threaded distributed programs. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2008.
5. M. Centenaro, R. Focardi, F. L. Luccio, and G. Steel. Type-based Analysis of Financial APIs (Full Paper). Submitted for publication. Available at http://www.dsi.unive.it/∼focardi/Articoli/CFLS-PIN09-full.pdf.
6. J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.
7. R. Focardi, F. L. Luccio, and G. Steel. Improving PIN Security. Forthcoming.
8. IBM Inc. CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors, 2006. Releases 2.53–3.27.
9. D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.
10. M. Mannan and P. van Oorschot. Weighing Down The Unbearable Lightness of PIN Cracking. In Springer LNCS vol.5143/2008, editor, *12th International*

Conference, Financial Cryptography and Data Security (FC 2008), Cozumel, Mexico, pages 176–181, January 28-31 2008. Extended version available at http://www.scs.carleton.ca/research/tech reports/.

11. A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.
12. G. Steel. Formal Analysis of PIN Block Attacks. *Theoretical Computer Science*, 367(1-2):257–270, November 2006.