Type-based Analysis of Financial APIs
(Extended Abstract)

Matteo Centenaro\textsuperscript{1}  Riccardo Focardi\textsuperscript{1}  Flaminia Luccio\textsuperscript{1}
Graham Steel\textsuperscript{2}

\textsuperscript{1}Università Ca’ Foscari di Venezia, Italy
{mcentena,focardi,luccio}@dsi.unive.it

\textsuperscript{2}LSV, ENS Cachan & CNRS & INRIA, France
graham.steel@lsv.ens-cachan.fr

ARSPA-WITS’09
March 28-29 2009, York
PIN processing APIs

Case study: PIN verification API

Acquiring bank

Switch

\{ PIN \}_{k_1}

Switch + keypad

ATM

Issuing bank

Switch

\{ PIN \}_{k_4}

\{ PIN \}_{k_2}

Switch

\{ PIN \}_{k_3}

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

Type-based Analysis of Financial APIs
Hardware Security Modules (HSMs)

- Protects the PIN encryption key
- Protects the PIN derivation key (PDK)
- Have a carefully designed API providing function for the verification and translation of the PIN

Goal

Use the technique of Language-Based information flow to analyze these APIs
The PIN verification API

PIN_V (PAN, EPB, len, offset, vdata, dectab) {
    // deriving user PIN (IBM 3624 method)
    x_1 := enc_{pdk}(vdata);   // encrypts vdata with pdk
    x_2 := left(len, x_1);     // takes len leftmost digits
    x_3 := decimalize(dectab, x_2); // decimalizes
    x_4 := sum_mod10(x_3, offset); // sums the offset

    // recovering the trial PIN from ISO-1 block
    x_5 := dec_k(EPB);         // decrypts the EPB with k
    x_6 := fcheck(x_5);        // extracts formatted PIN
    if (x_6 == "FAIL") then return("format error");

    // checks the trial versus the actual PIN
    if (x_4 == x_6) then return("PIN is correct");
    else return("PIN is wrong");
}
The PIN verification API - example

- \( \text{len}=4 \), \( \text{offset}=4732 \), \( \text{dectab} = 9753108642543210 \), encoding
  
  
  \[
  \begin{array}{cccccccccccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F \\
  9 & 7 & 5 & 3 & 1 & 0 & 8 & 6 & 4 & 2 & 5 & 4 & 3 & 2 & 1 & 0
  \end{array}
  \]

- \( x_1 = \text{enc}_{pdk}(vdata) = A47295FDE32A48B1 \)

\[
\begin{array}{lcl}
  x_2 = \text{left}(4, A47295FDE32A48B1) & = A472 \\
  x_3 = \text{decimalize}(\text{dectab}, A472) & = 5165 \\
  x_4 = \text{sum_mod10}(5165, 4732) & = 9897
\end{array}
\]

- \( EPB = \{9897, r\}_k \)

\[
\begin{array}{lcl}
  x_5 = \text{dec}_k(\{9897, r\}_k) & = (9897, r) \\
  x_6 = \text{fcheck}(9897, r) & = 9897
\end{array}
\]

- Since \( x_4 = x_6 \) the API returns "PIN is correct".
The PIN verification API - information flow challenges

- Noninterference cannot be used: the PIN_V API function is meant to release a one bit information about the user PIN.

- Declassification policy: certain flows are permitted:
  - the attacker should not be able to tamper with the declassification mechanism
  - Robust declassification [MSZ-JCS06]

- PIN_V result depends on low-integrity data:
  - *endorsement* admits some integrity flows
  - there are known attacks based on this approach
What’s new?

- Extend the language-based security framework for robust declassification to allow the integrity of inputs to be assured cryptographically by using MACs
- Model deterministic encryption
- Semantics and Type system to formally analyze PIN processing APIs
A basic imperative language

Syntax

\[ e ::= x \mid e_1 \text{ op } e_2 \]
\[ c ::= \text{skip} \mid x := e \mid c_1 ; c_2 \]
\[ \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]

Semantics

- Memories \( M \) are finite maps from variables to values
- \( e \downarrow^M \nu \) denotes the evaluation of expression \( e \) in a memory \( M \) giving value \( \nu \) as a result
- \( \langle M, c \rangle \Rightarrow M' \) denotes the execution of a command \( c \) in a memory \( M \), resulting in a new memory \( M' \) (big-step semantics)
The security environment $\Gamma$ maps each variable to a level of confidentiality and integrity. (We limit to $H$ and $L$).

- $\ell_1 \sqsubseteq_C \ell_2$ (or $\ell_1 \sqsubseteq_I \ell_2$) denotes that $\ell_1$ is less restrictive than $\ell_2$.

- We have $L \sqsubseteq_C H$ and $H \sqsubseteq_I L$ and take the product lattice.
Noninterference

Definition (Noninterference)

A command $c$ satisfies noninterference if

$$\forall \ell, M_1, M_2. M_1 =_{\ell} M_2 \implies \langle M_1, c \rangle =_{\ell} \langle M_2, c \rangle$$

where

- $M|_{\ell}$ is the restriction of memory $M$ to variables at or below $\ell$.
- $M_1 =_{\ell} M_2$ iff $M_1|_{\ell} = M_2|_{\ell}$
- $\langle M_1, c \rangle =_{\ell} \langle M_2, c \rangle$ iff whenever $\langle M_1, c \rangle \Rightarrow M'_1$ and $\langle M_2, c \rangle \Rightarrow M'_2$ then $M'_1 =_{\ell} M'_2$
- NI captures both confidentiality and integrity leakage
Robustness

Leakages are admitted but they should be independent of the attacker activity

**Definition (Robustness)**

A command $c$ is robust if $\forall M_1, M_2, M'_1, M'_2$ such that

\[
\begin{align*}
M_1 \equiv_{LL} & M_2 \\
H \equiv & H \\
M'_1 \equiv_{LL} & M'_2
\end{align*}
\]

it holds

\[
\langle M_1, c \rangle =_{LL} \langle M_2, c \rangle \text{ iff } \langle M'_1, c \rangle =_{LL} \langle M'_2, c \rangle
\]
Consider program $z_{LL} := (x_{LL} = y_{HH})$ and memories

\begin{array}{l|l|l|l|l}
  M_1 & M_2 & M'_1 & M'_2 \\
  \hline
  y_{HH} & 1234 & y_{HH} & 5678 \\
  x_{LL} & 1234 & x_{LL} & 1234 \\
  z_{LL} & \bullet & z_{LL} & \bullet \\
\end{array}

1. $M_1 =_{LL} M_2$ but $\langle M_1, P \rangle \neq_{LL} \langle M_2, P \rangle$ (interferent)

2. $M_1 =_{HH} M'_1$, $M_2 =_{HH} M'_2$ and $M_1 =_{LL} M_2$ but $\langle M_1, P \rangle \neq_{LL} \langle M_2, P \rangle$ and $\langle M'_1, P \rangle =_{LL} \langle M'_2, P \rangle$ (non-robust)

3. program $z_{LL} := (x_{LH} = y_{HH})$ is robust
Modelling Cryptography

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

Special expressions for
1. confounder generation,
2. (symmetric) cryptography,
3. Message Authentication Codes (MACs),
4. pairing and projection

working, as expected, on values \( \perp \mid n \mid k \mid \{v\}_k \mid \langle v \rangle_k \mid (v_1, v_2) \)

- Keys \( k \in \mathcal{K} \) are partitioned into \( \mathcal{K}_{HH} \) and \( \mathcal{K}_{LL} \)
\(\ell\)-equivalence

**Definition (Patterns)**

\[
\begin{align*}
\varphi_\ell(n) &= n \\
\varphi_\ell(r) &= \square r \\
\varphi_\ell((v_1, v_2)) &= (\varphi_\ell(v_1), \varphi_\ell(v_2)) \\
\varphi_\ell(\langle v \rangle_k) &= \langle \varphi_\ell(v) \rangle_k \\
\varphi_\ell(\{ v \}_k) &= \begin{cases} 
\square \{ v \}_k & \text{if } k \in K_\ell, \ell' \not\supseteq \ell \\
\{ \varphi_\ell(v) \}_k & \text{otherwise}
\end{cases}
\end{align*}
\]

**Definition (Cryptographic \(\ell\)-equivalence)**

Two memories \(M_1\) and \(M_2\) are indistinguishable at level \(\ell\), written \(M_1 \approx_\ell M_2\), if there exists a bijection \(\rho: \square v \leftrightarrow \square v\) such that \(\varphi_\ell(M_1) = \varphi_\ell(M_2) \rho\).
**ℓ-equivalence — an example**

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{LL}$ : ${1234}_k$</td>
<td>$x_{LL}$ : ${9999}_{k'}$</td>
</tr>
<tr>
<td>$y_{LL}$ : ${1234}_k$</td>
<td>$y_{LL}$ : ${5678}_{k'}$</td>
</tr>
</tbody>
</table>

where $k, k' \in \mathcal{K}_{HH}$.

<table>
<thead>
<tr>
<th>$p_{LL}(M_1)$</th>
<th>$p_{LL}(M_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{LL}$ : $\Box {1234}_k$</td>
<td>$x_{LL}$ : $\Box {9999}_{k'}$</td>
</tr>
<tr>
<td>$y_{LL}$ : $\Box {1234}_k$</td>
<td>$y_{LL}$ : $\Box {5678}_{k'}$</td>
</tr>
</tbody>
</table>

- There exists no bijection mapping $p_{LL}(M_1)$ to $p_{LL}(M_2)$
- Thus, $M_1 \not\approx_{LL} M_2$
- In fact, program $z_{LL} := (x_{LL} = y_{LL})$ distinguishes $M_1$ and $M_2$
PIN verification API is not robust

Suppose to run $PIN_v$ on these memories

\[
\begin{array}{cc}
M_1 & M_2 \\
\hline
dectab_{LL} & 9753208642543220 \\
EPB_{LL} & \{9897, r\}_k \\
\end{array}
\begin{array}{cc}
dectab_{LL} & 9753208642543220 \\
EPB_{LL} & \{1111, r\}_k \\
\end{array}
\]

It gets “PIN is wrong” on both: $\langle M_1, PIN_V \rangle \approx_{LL} \langle M_2, PIN_V \rangle$.

\[
\begin{array}{cc}
M'_1 & M'_2 \\
\hline
dectab_{LL} & 9753108642543210 \\
EPB_{LL} & \{9897, r\}_k \\
\end{array}
\begin{array}{cc}
dectab_{LL} & 9753108642543210 \\
EPB_{LL} & \{1111, r\}_k \\
\end{array}
\]

It gets “PIN is wrong” only on $M'_2$: $\langle M'_1, PIN_V \rangle \not\approx_{LL} \langle M'_2, PIN_V \rangle$

- By changing the dectab the attacker can control what is leaked
New integrity levels - Dependent domains

- Dependent domains, noted $D : \tilde{D}$, track integrity dependencies among variables
  - values of domain $D : \tilde{D}$ are determined by the values in the set of domains $\tilde{D}$.
  - For example, PIN : PAN states that when the PAN is fixed, the value of the PIN is also fixed.
  - Domains $D : \emptyset$, also written $D$, are called integrity representatives as they do not depend on other domains
- The integrity level associated to a dependent domain $D : \tilde{D}$ is written $[D : \tilde{D}]$, and is such that $[D : \tilde{D}] \sqsubseteq I_H$
  - In some cases we lose information about the precise result domain $D : \tilde{D}$ and we only record the dependency via $[\bullet : \tilde{D}]$
  - we write C (constant) in place of $[\bullet]$
New integrity levels - lattice

The new levels give rise to a new integrity lattice

\[ [D : \tilde{D}_1] \sqsubseteq [\bullet : \tilde{D}_1] \sqsubseteq_l [\bullet : \tilde{D}_2] \sqsubseteq_l H \sqsubseteq_l L \] with \( \tilde{D}_1 \subseteq \tilde{D}_2 \)

NOTE: There are no two levels such that they are of the form \([D : \tilde{D}]\) with the same \(D\) and a different \(\tilde{D}\).
The personal account number (PAN) is a natural candidate for an integrity representative.

The values of the former parameters are labelled with the four integrity levels [LEN : PAN], [OFF : PAN], [VD : PAN] and [DEC : PAN].

Summing `len` and `offset` the result would be of level [● : PAN].
Types

New security levels

\[
\begin{align*}
\delta_C & ::= \ L \mid H \\
\delta_I & ::= \ L \mid H \mid [D : \tilde{D}] \mid [\bullet : \tilde{D}] \\
\delta & ::= \delta_C \delta_I
\end{align*}
\]

Type syntax

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

- \(\kappa\) is a key label, related to a unique key;
- \(\mu\) indicates whether the ciphertext is ‘randomized’ via confounders \((\mu = R)\) or not \((\mu\) missing);
The proposed type system can deal with

- randomized encryption (in the usual way)

- non-randomized encryption requiring that the message content has an high-integrity level dependent on some integrity representatives *included* in the message

- MAC-check block in order to establish trust on some given data; this will be used to treat the API inputs as untainted subject to a successful MAC verification

- when decrypting an high-integrity message, rebuild the exact payload type
Typing non-randomized ciphertexts

\[ \Delta(x) = cK_{HC}(\tau) \kappa \quad \Delta \vdash e : \tau \quad \delta_I = C \sqcup L_I(\tau) \]

\[(\text{enc-d}) \quad \text{Closed}(\tau) \quad \text{DD}(\tau) \]

\[ \Delta \vdash \text{enc}_x(e) : \text{enc}_{L\delta_I} \kappa \]

- We can type, e.g., \([\text{PAN, PIN}]_k\)
- PIN is at level \([\text{PIN} : \text{PAN}]\)
- equal PANs will determine equal PINs,
- thus different PINs will always be encrypted together with different PANs, producing different EPBs.
- the PAN is a sort of confounder that is ‘reused’ only when its own PIN is encrypted
- this avoids building a codebook of all the PINs
Typing MAC-checks (simplified)

\[
\Delta(x) = mK_{HC}(L[D], \tau) \quad \Delta \vdash z : L[D] \\
\Delta \vdash e : LL \\
\Delta \vdash y : \tau \\
\Delta \vdash e' : LL \\
\Delta, LL \sqcup pc, [\bullet : D] \sqcup ir \vdash c_1 \\
\Delta, LL \sqcup pc, ir \vdash c_2 \\
\Delta, pc, ir \vdash \text{if } mac_x(z, e) = e' \text{ then } (y := e; c_1) \text{ else } c_2
\]

- $z$ is typed at level $L[D]$ (integrity representative)
- $e$ and $e'$ are typed $LL$
- If the MAC succeeds, variable $y$ of type $\tau$ is bound to the result of $e$
  - Here is where the integrity level can increase
- In the if branch we raise $ir$ so as to record the checked integrity representative
  - This allows for special assignments and declassifications
MAC-based PIN verification

```plaintext
PIN_V_M(PAN, EPB, len, offset, vdata, dectab, MAC) {
    // checking the MAC
    if (mac_k(PAN, EPB, len, offset, vdata, dectab) == MAC) then
        EPB' := EPB; len' := len; offset' := offset; vdata' := vdata;
        dectab' := dectab;
        // invokes the original API
        return(PIN_V(PAN, EPB', len', offset', vdata', dectab'));
    else
        return("integrity violation"); // MACs do not correspond
}
```
Security results

**Proposition**

*If $\Delta, pc, C \vdash c$ and $c$ does not contain any declassification outside MAC-check blocks, then $c$ satisfies noninterference.*

**Theorem**

*If $\Delta, pc, C \vdash c$ then $c$ satisfies robustness.*
1. extended information flow security types to deal with
   - deterministic encryption
   - cryptographic assurance of integrity

2. used the model to check the PIN processing API

3. proposed a solution to make the PIN verification type-checkable
1 Introduction
   - PIN processing APIs
   - Case study: PIN verification API

2 Formal Model
   - Basic imperative language
   - Noninterference and Robustness
   - Cryptographic extension

3 Types
   - New integrity types
   - Typing rules (briefly)
   - A type-checkable API
   - Results

4 Conclusion