Type-based Analysis of Security APIs

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

Workshoplet on Formal Methods for Security  
March 12-13 2009, Padova
1 Introduction
   - PIN processing APIs
   - Case study: PIN verification API

2 Formal Model
   - Basic imperative language
   - Noninterference and Robustness
   - Cryptographic extension
   - Formal analysis: the “decimalization” attack

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

4 Conclusion
Hardware Security Modules

(image taken from ‘API Security — Attacks and Defences’ by Ross Anderson and Mike Bond)
The ATM network

(image inspired from ‘API Security — Attacks and Defences’ by Ross Anderson and Mike Bond)
The PIN verification API

\[
\text{PIN}_V(PAN, EPB, \text{len}, \text{offset}, \text{vdata}, \text{dectab}) \{
\begin{align*}
&\text{// deriving user PIN (IBM 3624 method)} \\
&x_1 := \text{enc}_{pdk}(\text{vdata}); \quad \text{// encrypts vdata with pdk} \\
&x_2 := \text{left}(\text{len}, x_1); \quad \text{// takes len leftmost digits} \\
&x_3 := \text{decimalize}(\text{dectab}, x_2); \quad \text{// decimalizes} \\
&x_4 := \text{sum}_\text{mod10}(x_3, \text{offset}); \quad \text{// sums the offset} \\
&\text{// recovering the trial PIN from ISO-1 block} \\
&x_5 := \text{dec}_k(EPB); \quad \text{// decrypts the EPB with k} \\
&x_6 := \text{fcheck}(x_5); \quad \text{// extracts formatted PIN} \\
&\text{if } (x_6 == "FAIL") \text{ then return("format error");} \\
&\text{// checks the trial versus the actual PIN} \\
&\text{if } (x_4 == x_6) \text{ then return("PIN is correct");} \\
&\text{else return("PIN is wrong");}
\}
\]
The PIN verification API: Example

- \( \text{len}=4, \ offset=4732, \ \text{dectab} = 9753108642543210, \ \text{encoding} \)

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F \\
9 & 7 & 5 & 3 & 1 & 0 & 8 & 6 & 4 & 2 & 5 & 4 & 3 & 2 & 1 & 0
\end{array}
\]

- \( x_1 = \text{enc}_{pdk}(vdata) = A47295FDE32A48B1 \)
  \[
  \begin{align*}
  x_2 &= \text{left}(4, A47295FDE32A48B1) = A472 \\
  x_3 &= \text{decimalize}(\text{dectab}, A472) = 5165 \\
  x_4 &= \text{sum}_\mod10(5165, 4732) = 9897
  \end{align*}
  \]
- \( \text{EPB} = \{9897, r\}_k \)
  \[
  \begin{align*}
  x_5 &= \text{dec}_k(\{9897, r\}_k) = (9897, r) \\
  x_6 &= \text{fcheck}(9897, r) = 9897
  \end{align*}
  \]
- Since \( x_4 = x_6 \) the API returns "PIN is correct".
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
- \( e \downarrow^M v \) denotes the evaluation of expression \( e \) in a memory M giving value \( v \) 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'
Semantic rules

\[\text{[skip]} \quad \langle M, \text{skip} \rangle \Rightarrow M\]
\[\text{[assign]} \quad e \downarrow^M v \quad \frac{\langle M, x := e \rangle}{\Rightarrow M[x \mapsto v]}\]
\[\text{[seq]} \quad \langle M, c_1 \rangle \Rightarrow M' \quad \langle M', c_2 \rangle \Rightarrow M'' \quad \frac{}{\langle M, c_1; c_2 \rangle \Rightarrow M''}\]
\[\text{[ift]} \quad e \downarrow^M \text{true} \quad \frac{\langle M, c_1 \rangle \Rightarrow M' \quad \langle M, \text{if } e \text{ then } c_1 \text{ else } c_2 \rangle \Rightarrow M'}{\Rightarrow M'}\]
\[\text{[iff]} \quad e \downarrow^M \text{false} \quad \frac{\langle M, c_2 \rangle \Rightarrow M' \quad \langle M, \text{if } e \text{ then } c_1 \text{ else } c_2 \rangle \Rightarrow M'}{\Rightarrow M'}\]
\[\text{[while]} \quad e \downarrow^M \text{true} \quad \frac{\langle M, c \rangle \Rightarrow M' \quad \langle M', \text{while } e \text{ do } c \rangle \Rightarrow M'' \quad \langle M, \text{while } e \text{ do } c \rangle \Rightarrow M''}{\Rightarrow M''}\]
\[\text{[whilef]} \quad e \downarrow^M \text{false} \quad \frac{}{\langle M, \text{while } e \text{ do } c \rangle \Rightarrow M}\]
The security environment $\Gamma$ maps each variable to a level of confidentiality and integrity. (We limit to $H$ and $L$).

- $\ell_1 \sqsubseteq_C \ell_2$ ($\ell_1 \sqsubseteq_I \ell_2$) denotes that $\ell_1$ is less restrictive than $\ell_2$.
- We have $L \sqsubseteq_C H$ and $H \sqsubseteq_I L$ giving
Noninterference

Definition (Noninterference)

A command $c$ satisfies \textit{noninterference} if

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

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 $M'_1 =\ell M'_2$
Robustness admits leakages but ...

1. the leakage should be independent of the attacker activity
2. the attacker can only manipulate low integrity variables, thus he will always preserve $=_{HH}$ on memories

**Definition (Robustness)**

A command $c$ is robust if $\forall M_1, M_2, M'_1, M'_2$ such that $M_1 =_{LL} M_2, M'_1 =_{LL} M'_2, M_1 =_{HH} M'_1, M_2 =_{HH} M'_2$, it holds

$$\langle M_1, c \rangle =_{LL} \langle M_2, c \rangle \text{ iff } \langle M'_1, c \rangle =_{LL} \langle M'_2, c \rangle$$

**NOTE:** This is a simplification of [MSZ’06]
### Noninterference & Robustness: a simple example

Consider program $z_{LL} := (x_{LL} = y_{HH})$ and memories

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M'_1$</th>
<th>$M'_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_{HH}$: 1234</td>
<td>$y_{HH}$: 5678</td>
<td>$y_{HH}$: 1234</td>
<td>$y_{HH}$: 5678</td>
</tr>
<tr>
<td>$x_{LL}$: 1234</td>
<td>$x_{LL}$: 1234</td>
<td>$x_{LL}$: 1111</td>
<td>$x_{LL}$: 1111</td>
</tr>
<tr>
<td>$z_{LL}$: •</td>
<td>$z_{LL}$: •</td>
<td>$z_{LL}$: •</td>
<td>$z_{LL}$: •</td>
</tr>
</tbody>
</table>

1. $M_1 =_{LL} M_2$ but $\langle M_1, P \rangle \not=_{LL} \langle M_2, P \rangle$ (interferent)
2. $M_1 =_{HH} M'_1$, $M_2 =_{HH} M'_2$ and $M_1 =_{LL} M_2$ but $\langle M_1, P \rangle \not=_{LL} \langle M_2, P \rangle$ 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 \( \bot \mid n \mid k \mid \{v\}_k \mid \langle v \rangle_k \mid (v_1, v_2) \)

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

**Definition (Patterns)**

\[
\begin{align*}
p_\ell(n) &= n \\
p_\ell(r) &= \emptyset_r \\
p_\ell((v_1, v_2)) &= (p_\ell(v_1), p_\ell(v_2)) \\
p_\ell(\langle v \rangle_k) &= \langle p_\ell(v) \rangle_k \\
p_\ell(\{v\}_k) &= \begin{cases} \\
\emptyset\{v\}_k & \text{if } k \in \mathcal{K}_\ell, \ell' \not\subseteq \ell \\
\left\{p_\ell(v)\right\}_k & \text{otherwise}
\end{cases}
\end{align*}
\]

**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 \( \rho : \emptyset_v \mapsto \emptyset_v \) such that \( p_\ell(M_1) = p_\ell(M_2) \rho \).
$\ell$-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 $k, k' \in K_{HH}$.

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

1. The intruder picks a decimal digit $d$,
2. changes the $dectab$ function so that values previously mapped to $d$ now map to $d + 1 \mod 10$,
3. if the system still returns "$PIN is correct" the digit is not in the intermediate PIN (goto 1)
4. otherwise, the digit is in the intermediate PIN
5. the intruder now changes the $offset$ until the API returns again that the PIN is correct. This allows the intruder to locate the position of the deduced PIN digit.
6. the algorithm is iterated until all the PIN digits are found.
The “decimalization” attack — an example

We consider again $len = 4$, $offset = 4732$, $EPB = \{9897, r\}_k$, $dectab = 9753108642543210$, $x1 = A47295FDE32A48B1$,

1. We pick 0
2. $dectab' = 9753118642543211$
3. $dectab'(A472) = dectab(A472) = 5165$ we still get “PIN is correct”. 0 is not in the intermediate PIN (goto 1)

1. We pick 1
2. $dectab'' = 9753208642543220$
3. $dectab''(A472) = 5265 \neq dectab(A472) = 5165$ and we get “PIN is wrong”. 1 is in the intermediate PIN

4. offset 4632 makes the API return again ”PIN is correct” meaning that 1 is in the second position and $1 + 7 = 8$ is the second PIN digit.
PIN verification API is not robust

Previous attack is related to the absence of robustness

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$M_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dectab_{LL}$ : $9753208642543220$</td>
<td>$dectab_{LL}$ : $9753208642543220$</td>
</tr>
<tr>
<td>$EPB_{LL}$ : ${9897, r}_k$</td>
<td>$EPB_{LL}$ : ${1111, r}_k$</td>
</tr>
</tbody>
</table>

We get “PIN is wrong” on both, i.e., $\langle M_1, PIN_V \rangle \approx_{LL} \langle M_2, PIN_V \rangle$.

<table>
<thead>
<tr>
<th>$M'_1$</th>
<th>$M'_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dectab_{LL}$ : $9753108642543210$</td>
<td>$dectab_{LL}$ : $9753108642543210$</td>
</tr>
<tr>
<td>$EPB_{LL}$ : ${9897, r}_k$</td>
<td>$EPB_{LL}$ : ${1111, r}_k$</td>
</tr>
</tbody>
</table>

We get “PIN is wrong” only on $M'_2$, i.e., $\langle M'_1, PIN_V \rangle \not\approx_{LL} \langle M'_2, PIN_V \rangle$

- 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 loose 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 security levels

$$\delta_C ::= L \mid H$$

$$\delta_I ::= L \mid H \mid [D : \tilde{D}] \mid [\bullet : \tilde{D}]$$

$$\delta ::= \delta_C \delta_I$$

Extended integrity lattice

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

with $$\tilde{D}_1 \subseteq \tilde{D}_2$$

Type syntax

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

- $$\kappa$$ is a key label, related to a unique key;
- $$\mu$$ indicates whether the ciphertext is ‘randomized’ via confounders ($$\mu = R$$) or not ($$\mu$$ missing);
Typing non-randomized ciphertexts

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

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

- We can type, e.g., \(\{\text{PAN}, \text{PIN}\}_k\)
- 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)

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

- \( z \) 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
Security results

Proposition

If $\Delta, pc, C \vdash c$ and $c$ does not contain any declassification, then $c$ satisfies noninterference.

Proposition

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

Theorem

If $\Delta, pc, C \vdash c$ then $c$ satisfies robustness.
MAC-based PIN verification

PIN_V_M(PAN, EPB, len, offset, vdata, dectab, MAC) {

    // checking the MAC
    if (mac_ak(PAN, EPB, len, offset, vdata, dectab) == MAC)
        EPB' := EPB; len' := len; offset' := offset; vdata' := vdata;
        dectab' := dectab;
        // invokes the original API
        return(PINV(PAN, EPB', len', offset', vdata', dectab'));
    else
        return("integrity violation"); // MACs do not correspond
}
we have extended information flow security types to model
deterministic encryption and cryptographic assurance of
integrity for robust declassification
we have applied the new model to PIN processing APIs
the model appears to be suitable for ‘certifying’ real APIs and
we are referring to it while developing a more realistic version
of our proposed fix
many interesting related work, but I had no time to write a
‘related work’ slide :(