Security APIs

System Security (CM0625, CM0631) 2025-26
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Security APIs

Host machine

Trusted device

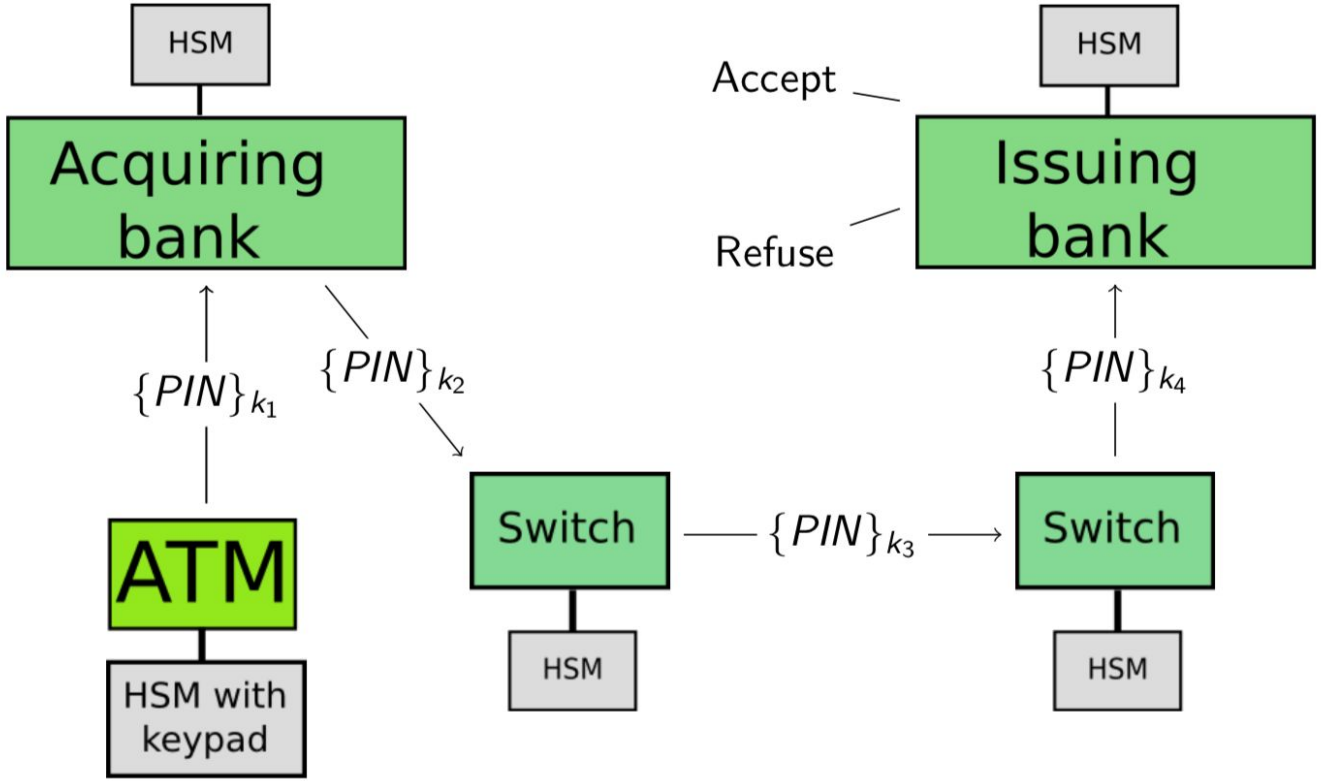
Security API

Case study 1: PIN verification

Hardware Security Modules (HSM)

- Used in the ATM Bank network
- Tamper resistant
- Offer APIs for:
  - Managing cryptographic keys
  - Decrypting/re-encrypting the PIN
  - Checking the validity of the PIN

PIN verification infrastructure (old protocol)



PIN verification

Encrypted PIN Block : contains the PIN at the ATM

PIN_V(EPB , vdata,len,dectab,offset)

Data for computing the user PIN

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 0123456789012345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   0472 ⊕ 4732 mod 10 = 4104
3. The two values coincide: PIN_V returns 'true'

PIN verification pseudo-code

```
PIN V(EPB, vdata, len, dectab, offset) {
    x1 := dec(k,EPB);  // decrypt the typed PIN
    t_pin:=f check(x1); // check format, remove random
    if (t_pin == ⊥) then return(''format wrong'');

    x2 := encpdk(vdata); // encrypt vdata
    x3 := left(len,x2);  // take left 4-5 hex digits
    x4 := decimalize(dectab,x3);     // decimalize digits
    u_pin := sum_mod10(x4, offset); // sum offset

    if (t_pin == u_pin) then return(''PIN is correct'');
}
```

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 0123456789012345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   0472 ⊕ 4732 mod 10 = 4104
3. PIN_V returns ‘true’

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 0123456789012345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   0472 ⊕ 4732 mod 10 = 4104
3. PIN_V returns ‘true’

1. Change one digit of dectab

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   0472 ⊕ 4732 mod 10 = 4104
3. PIN_V returns ‘true’

1. Change one digit of dectab

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   0472 ⊕ 4732 mod 10 = 4104
3. PIN_V returns ‘true’

1. Change one digit of dectab
2. This propagates ...

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 4732 mod 10 = 4104
3. PIN_V returns ‘true’

1. Change one digit of dectab
2. This propagates ...

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 4732 mod 10 = 4104
3. PIN_V returns ‘true’

1. Change one digit of dectab
2. This propagates ...

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 4732 mod 10 = 5104
3. PIN_V returns ‘true’

1. Change one digit of dectab
2. This propagates ...

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 4732 mod 10 = 5104
3. PIN_V returns ‘true’

1. Change one digit of dectab
2. This propagates ...

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 4732 mod 10 = 5104
3. PIN_V returns ‘false’

1. Change one digit of dectab
2. This propagates …
3. … and eventually changes the result!

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 4732 mod 10 = 5104
3. PIN_V returns ‘false’

1. Change one digit of dectab
2. This propagates …
3. … and eventually changes the result!

⇒ We know that 0 appeared in the PIN computation

Decimalization attack [Bond, Zielinski '03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 4732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 4732 mod 10 = 5104
3. PIN_V returns 'false'

1. Change one digit of dectab
2. This propagates …
3. … and eventually changes the result!

⇒ We know that 0 appeared in the PIN computation

1. We "compensate" on the offset to find the position

Decimalization attack [Bond, Zielinski '03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 3732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 4732 mod 10 = 5104
3. PIN_V returns 'false'

1. Change one digit of dectab
2. This propagates …
3. … and eventually changes the result!

⇒ We know that 0 appeared in the PIN computation

1. We "compensate" on the offset to find the position

Decimalization attack [Bond, Zielinski '03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 3732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 4732 mod 10 = 5104
3. PIN_V returns 'false'

1. Change one digit of dectab
2. This propagates …
3. … and eventually changes the result!

⇒ We know that 0 appeared in the PIN computation

1. We "compensate" on the offset to find the position

Decimalization attack [Bond, Zielinski '03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 3732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 3732 mod 10 = 5104
3. PIN_V returns 'false'

1. Change one digit of dectab
2. This propagates …
3. … and eventually changes the result!

⇒ We know that 0 appeared in the PIN computation

1. We "compensate" on the offset to find the position

Decimalization attack [Bond, Zielinski '03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 3732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 3732 mod 10 = 5104
3. PIN_V returns 'false'

1. Change one digit of dectab
2. This propagates …
3. … and eventually changes the result!

⇒ We know that 0 appeared in the PIN computation

1. We "compensate" on the offset to find the position

Decimalization attack [Bond, Zielinski '03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 3732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 3732 mod 10 = 4104
3. PIN_V returns 'false'

1. Change one digit of dectab
2. This propagates …
3. … and eventually changes the result!

⇒ We know that 0 appeared in the PIN computation

1. We "compensate" on the offset to find the position
2. … and we see if this fixes the result!

Decimalization attack [Bond, Zielinski ’03]

Example: PIN_V($\{4104,r\}_k$, vdata, 4, 1123456789112345, 3732)

1. $dec_k(\{4104,r\}_k)$ = 4104,r
   4104
2. enc_{pdk}(vdata) = A47295FDE32A48B1
   1472 ⊕ 3732 mod 10 = 4104
3. PIN_V returns ‘true’

1. Change one digit of dectab
2. This propagates …
3. … and eventually changes the result!

⇒ We know that 0 appeared in the PIN computation

1. We “compensate” on the offset to find the position
2. … and we see if this fixes the result!

⇒ If so we discover value and position!

Decimalization attack [Bond, Zielinski ’03]

This attack has been shown on real devices

- An insider sniffs ATM card data, launches the attack and infers the PIN
- **How many invocations** on average?
  - Four digit PINs: 14.47
  - Five digit PINs: 19.3
  - Strategies found automatically in [Focardi, Luccio ’10]
- Once the PIN is found (old) cards can be cloned

⇒ Thousand of PINs leaked in a lunch break!

NOTE: in countries where the chip cards are not yet widely used the attack would still work

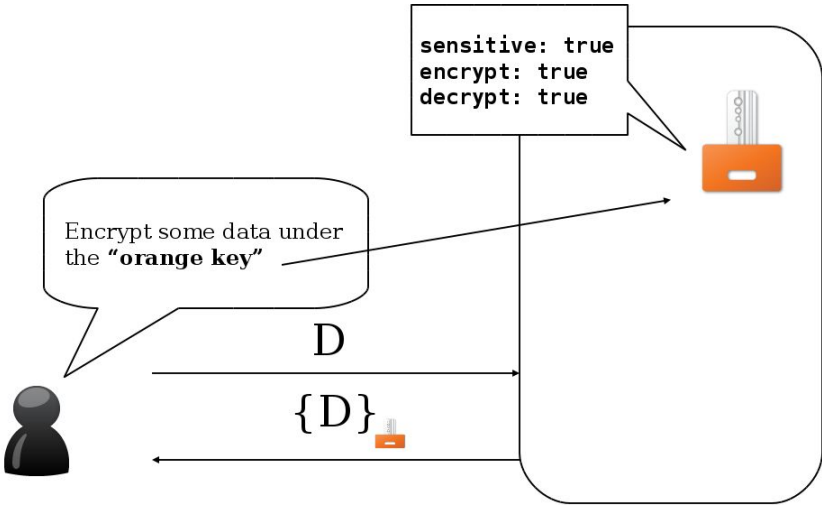
Case study 2: PKCS#11

PKCS#11 cryptographic operations

PKCS#11 is a standard API to cryptographic devices

Keys have **attributes** and are referenced via **handles** (that we represent with colors)

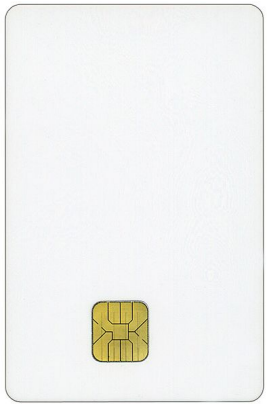
Example: orange key is sensitive and can be used to encrypt/decrypt data



Security of keys

Confidentiality of sensitive keys

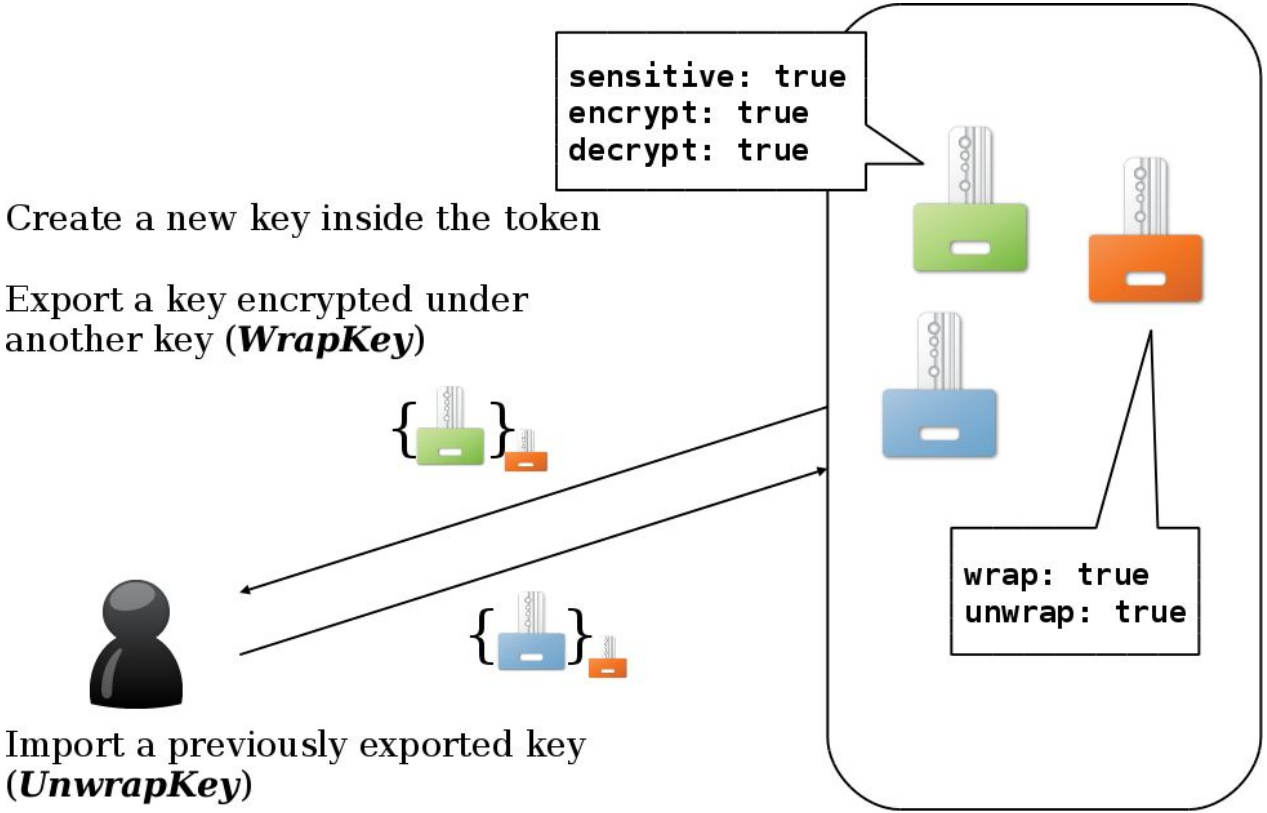
- `sensitive` keys should never be **accessible as plaintext** outside the device
- all crypto operations happen inside the device



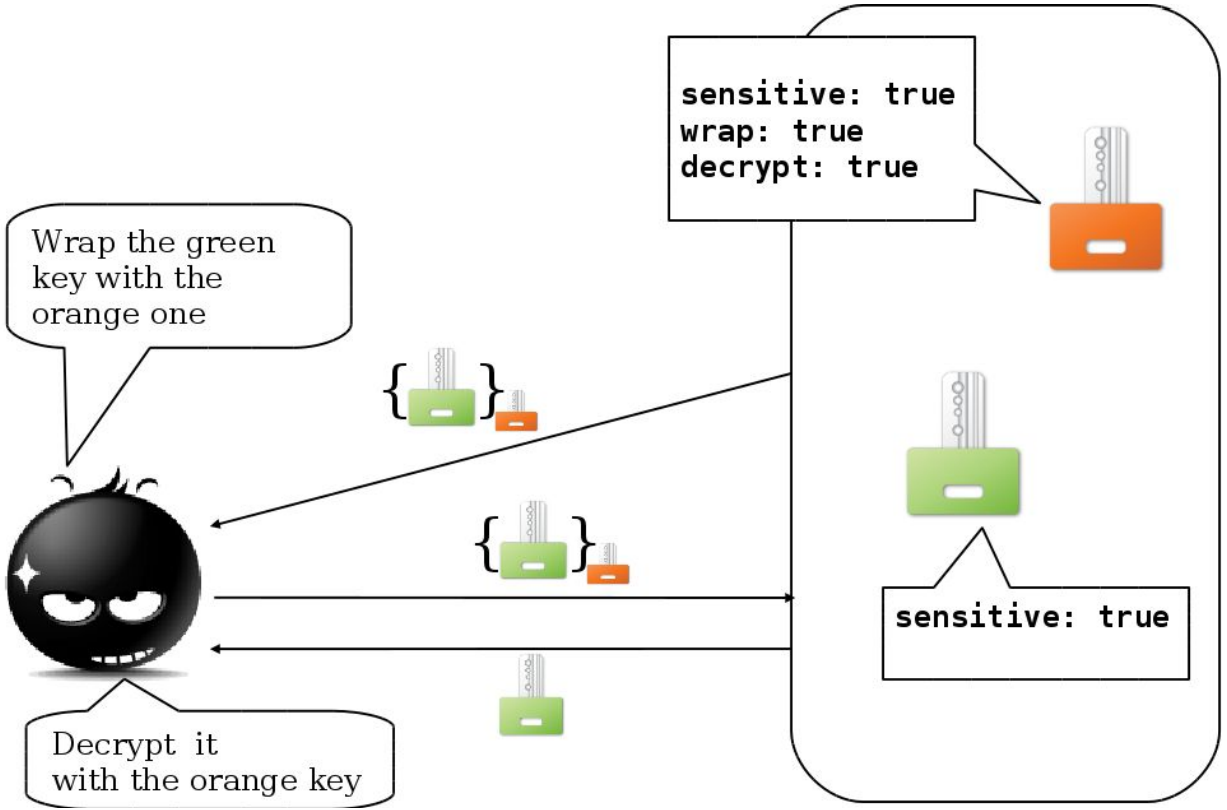
Attack scenario

1. the device is used on compromised host
2. the attacker extracts `sensitive` keys
3. the attacker **clones the device**

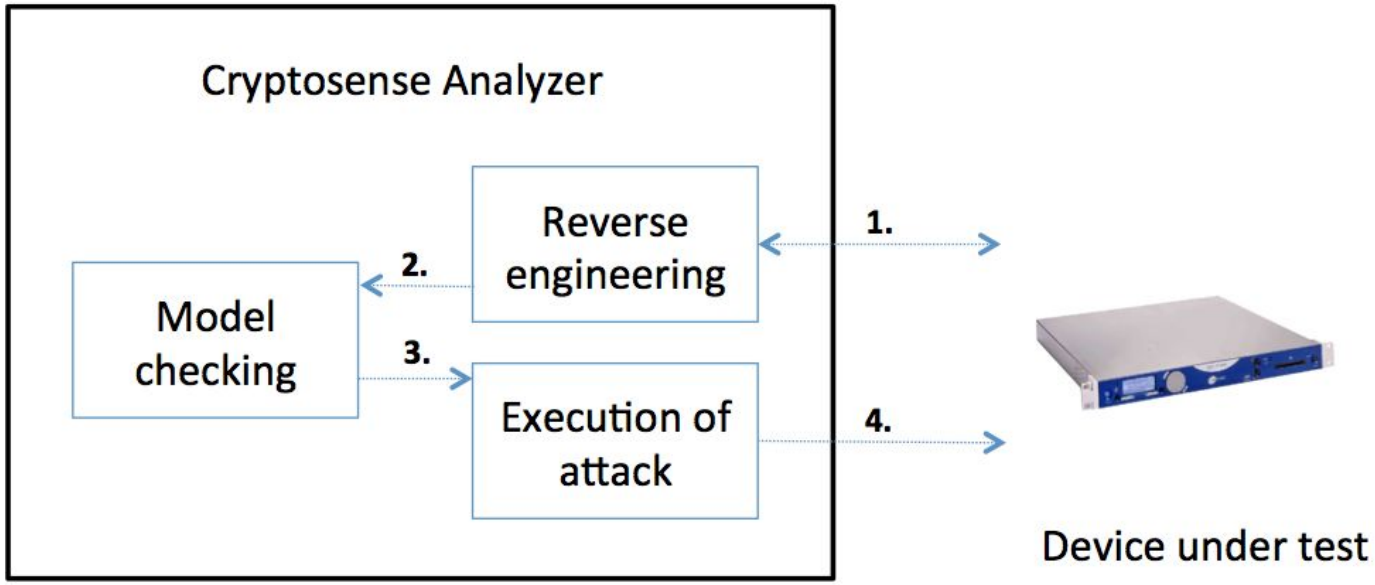
Key management example



The *wrap-then-decrypt* attack [CHES'03]



Formal verification



Real attacks [ACM CCS'10]

Brand	Model	Supported Functionality s	as	cobj	chan	w	ws	Attacks found wd	rs	ru	su	Tk
Aladdin	eToken PRO	✓	✓	✓	✓	✓	✓	✓				wd
Athena	ASEKey	✓	✓	✓								
Bull	Trustway RCI	✓	✓	✓	✓	✓	✓	✓				wd
Eutron	Crypto Id. ITSEC		✓	✓								
Feitian	StorePass2000	✓	✓	✓	✓	✓	✓	✓	✓	✓		rs
Feitian	ePass2000	✓	✓	✓	✓	✓	✓	✓	✓	✓		rs
Feitian	ePass3003Auto	✓	✓	✓	✓	✓	✓	✓	✓	✓		rs
Gemalto	SEG		✓		✓							
MXI	Stealth MXP Bio	✓	✓		✓							
RSA	SecurID 800	✓	✓	✓	✓				✓	✓	✓	rs
SafeNet	iKey 2032	✓	✓	✓		✓						
Sata	DKey	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	rs
ACS	ACOS5	✓	✓	✓	✓							
Athena	ASE Smartcard	✓	✓	✓								
Gemalto	Cyberflex V2	✓	✓	✓		✓	✓	✓				wd
Gemalto	SafeSite V1		✓		✓							
Gemalto	SafeSite V2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	rs
Siemens	CardOS V4.3 B	✓	✓	✓		✓				✓		ru

PKCS#11 is still flawed after 20+ years !?!

Fixes?

Fixes: Various proposals in the literature to modify the API, but never included in PKCS#11

⇒ Proprietary fixes exist but break compliance

Example: offline key management and **no key wrapping** in production

Mitigations: monitor/filter API calls **locally**

`wrap_with_trusted` *attribute* requires that keys are only wrapped under **`trusted`** keys (flagged by the Security Officer)

👍 Secure key wrapping, *in principle*

⚠️ No guidance in the docs

⚠️ How should **`trusted`** keys be generated/managed?

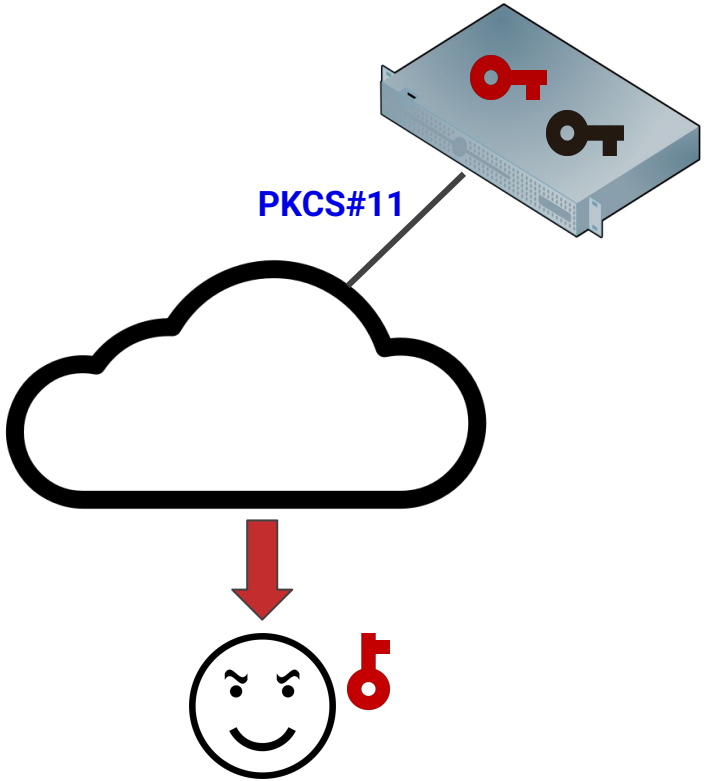
⚠️ What if a **`trusted`** key is flagged **`wrap+decrypt`**?

A new scenario: cloud HSMs

HSM hardware accessible as a service in the cloud

- Compliance to standard APIs: no proprietary fixes
- No offline, secure key management procedures
- No API-level monitors/filters

New attacker model: a vulnerability in one application would expose the full (flawed!) PKCS#11 API



A formally verified configuration

Focardi & Luccio
ACM CCS'21

- User **roles** to secure PKCS#11
- First secure PKCS#11 configuration that does not break the API **compliance**
- Implementation in a **real Cloud HSM solution**
- Formal model and automated **proof of security**

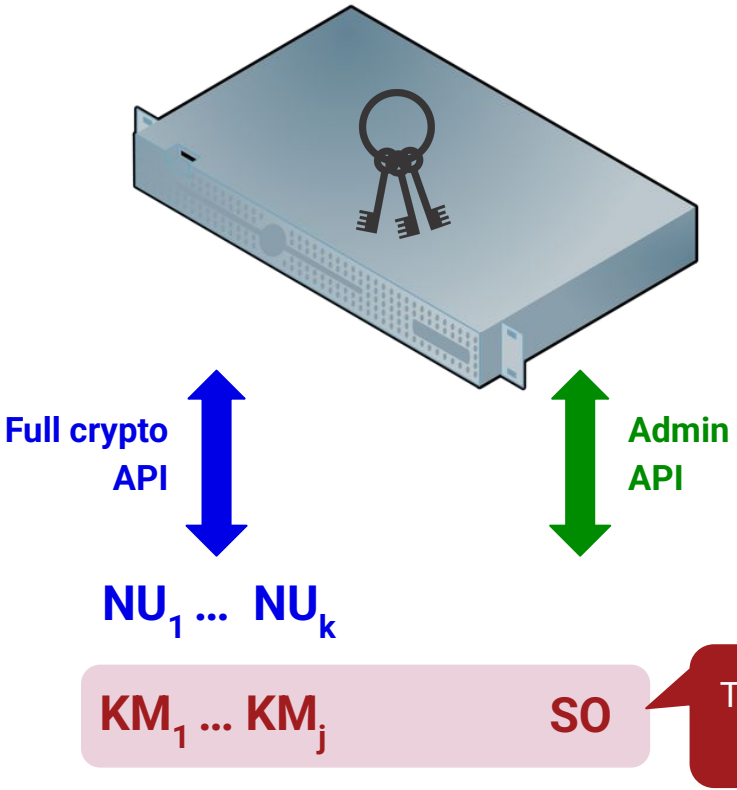
User roles

Normal Users (NU): used in production applications, full API but no attack should be possible

Key Managers (KM): create and manage candidate **`trusted`** keys

Security Officer (SO): admin, cannot do crypto but marks **`trusted`** keys

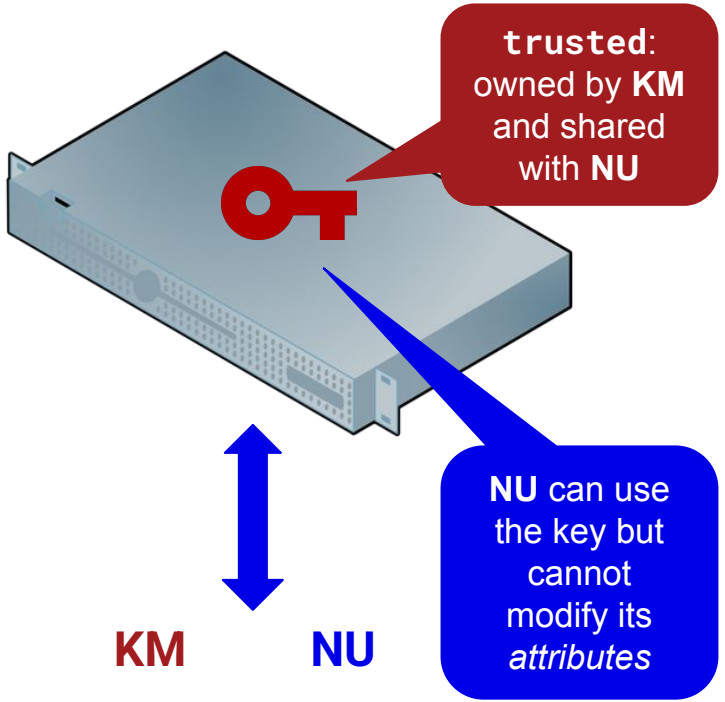
- KMs and SO only accessed by management apps or humans



Key sharing

1. **KMs** generate (candidate) **`trusted`** wrapping keys
2. **KMs** **share** these keys with **NUs**
3. **NUs** can use wrapping keys but **cannot** modify their *attributes,* e.g., cannot mark them as **`decrypt`** keys

- ***Key sharing*** is not in PKCS#11 but can be added on top, at the cloud/admin layer



Secure configuration

Rule 1 (Sensitive keys). Sensitive keys should be generated with **`wrap_with_trusted`** set or **`extractable`** unset (i.e. will never be wrapped).

Rule 2 (Trusted keys). The SO sets the **`trusted`** attribute only on candidate keys generated by one of the KMs.

Rule 3 (Roles of candidate keys). Candidate keys managed by the KMs should only admit **`wrap`** and **`unwrap`** operations, during their lifetime.

Rule 4 (Management of candidate keys). Candidate keys managed by the KMs should be generated with **`extractable`** unset (i.e. will never be wrapped)

Rule 5 (Freshness of candidate keys). Candidate keys managed by the KMs should be generated in the device.

Implementation in real cloud HSMs

AWS CloudHSM implements the required **key sharing** capability:
*"Users who share the key can use the key in cryptographic operations, but they **cannot change its attributes**"*

- The secure configuration can be implemented straightforwardly

Note: we assume a worst-case scenario in which all keys are shared

Other cloud solutions:

- do not have publicly available documentation
  (e.g. Utimaco, Microsoft)
- do not implement PKCS#11, yet?
  (e.g. Google, in 2021)
- do not seem to implement key sharing in the form we need
  (e.g. IBM)

Formal analysis

We formalize a significant subset of PKCS#11 in the Tamarin prover:

- Symmetric crypto and wrap
- **`wrap_with_trusted`** and **`trusted`** attributes
- User roles + key sharing

We automatically prove security for an **unbounded** number of users, keys and sessions

```
rule Wrap:
  [ !NU(U),
    !Key(U1,ha1,k1),
    !Key(U2,ha2,k2) ]
--[
    Wrap(U,ha1,ha2),
    IsSet(ha1,'wrap_with_trusted'),
    IsSet(ha1,'extractable'),
    IsSet(ha2,'trusted'),
    IsSet(ha2,'wrap')
  ]->
  [ Out(senc(k1,k2)) ]
```

Normal User U

Keys k1, k2 owned by U1, U2 (ha1, ha2 are *handles*)

U wraps ha1 with ha2, i.e., k1 with k2

Appropriate attributes

ciphertext is sent out (**simplified**, see the paper for detail!)

Automated proof

Keys which, at some point, are marked as **`trusted`** are never leaked

```
lemma SecrecyTrusted:
"
All W ha k #i #j #w.
  IsHandle(ha,k)@i &
  SetAttr(W,ha,'trusted')@j &
  KU(k)@w
==> F
"
```

ha is a handle for key k at time i

and ha has **`trusted`** set at time j

and the attacker knows k at time w

implies `false`, i.e., it cannot occur

Similar lemmas for sensitive keys generated with **`wrap_with_trusted`** set or **`extractable`** unset (*cf*. Rule 1)

Complete model with additional helper and sanity lemmas available at github.com/secgroup/CloudHSM-model

The complete model can be proved automatically in about **1m30s** on a MacBook Pro 2018