Attaques Side Channel sur des Hardware Wallets open source

5 juin 2019

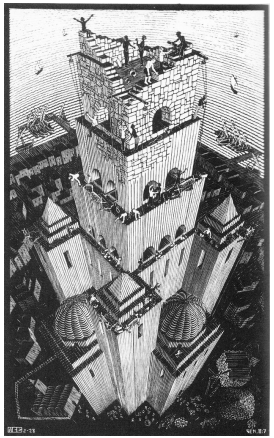
Ledger SAS
1, rue du Mail
75002 Paris - France

Ledger SAS
Parc Technologique de Sologne
Allée Georges Charpak
18100 Vierzon - France

Ledger Technologies Inc.
121 2nd Street
94105 San Francisco - USA

Ledger Donjon

- Ledger “Red” Team - **Indépendant**
  - Amélioration de la sécurité (HSM, Vault, Nano S/X)
  - Evaluation continue de nos produits
  - Services de consulting/évaluation sécuritaire



- Analyse de la sécurité de l’écosystème

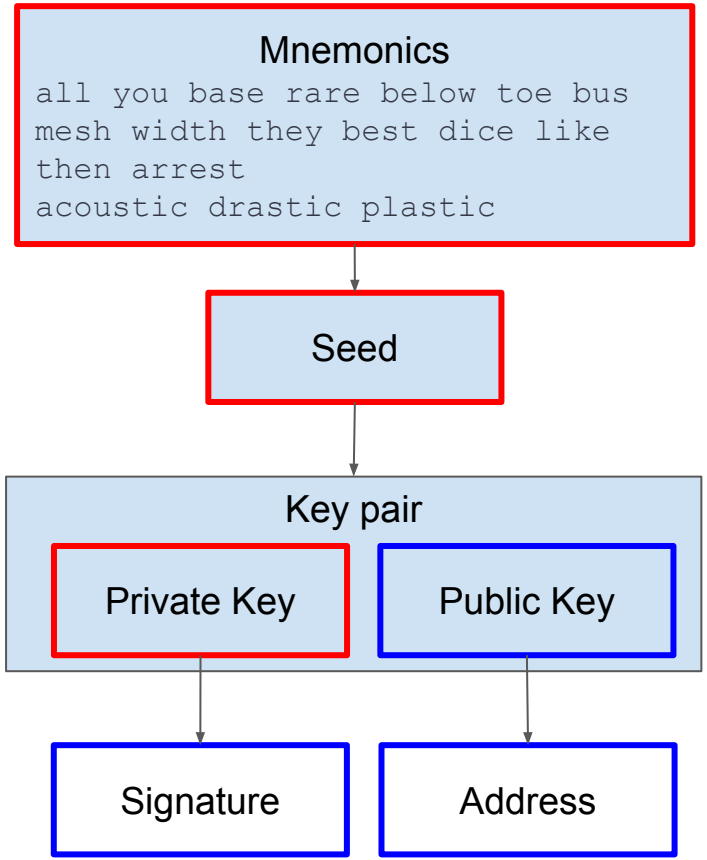
- Compétences/Expertise
  - Analyse Side-Channel
  - Attaques en Perturbation
  - Attaques logicielles
  - Cryptographie

- Nos outils d’attaques sont Open Source: https://github.com/Ledger-Donjon/

Programme

- Hardware Wallets - Les bases, la cible
- Attaque Side-Channel sur la vérification de PIN
- Attaque Side-Channel sur une multiplication scalaire
- Conclusion
- Outils Open Source

Hardware wallets - Les Bases



Different types de Wallets

Modèle de sécurité

- Stocke les données privées (elles ne quittent jamais le device)
- Accés grâce aux secrets avec PIN.
- Signe les transactions (Calculs ECDSA)
- Les Mnemonics servent de backup.

Hardware wallets - les bases - Faire une transaction

1. Authentification de l'utilisateur grâce au PIN

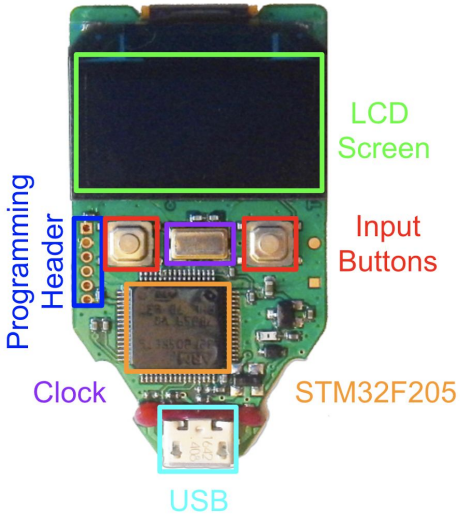
2. Préparation de la transaction
Addresse / Montant

3. Demander confirmation à l'utilisateur avec Trusted display / boutons - Le wallet signe la transaction

4. Envoie la transaction signée à la blockchain.

Contexte: Trezor Hardware wallet

- Hardware wallet à base de STM32
  Composant *General purpose*
- Le firmware est Open-source
  Etude en boîte blanche
- Cryptographie implémentée au sein de trezor-crypto
  (Quasiment) pas de contre-mesures side-channel
- Deux attaques sur deux fonctions présentées:
  (celles avec les contre-mesures)
  - Authentification par PIN (firmware)
  - Signature ECDSA (crypto lib)

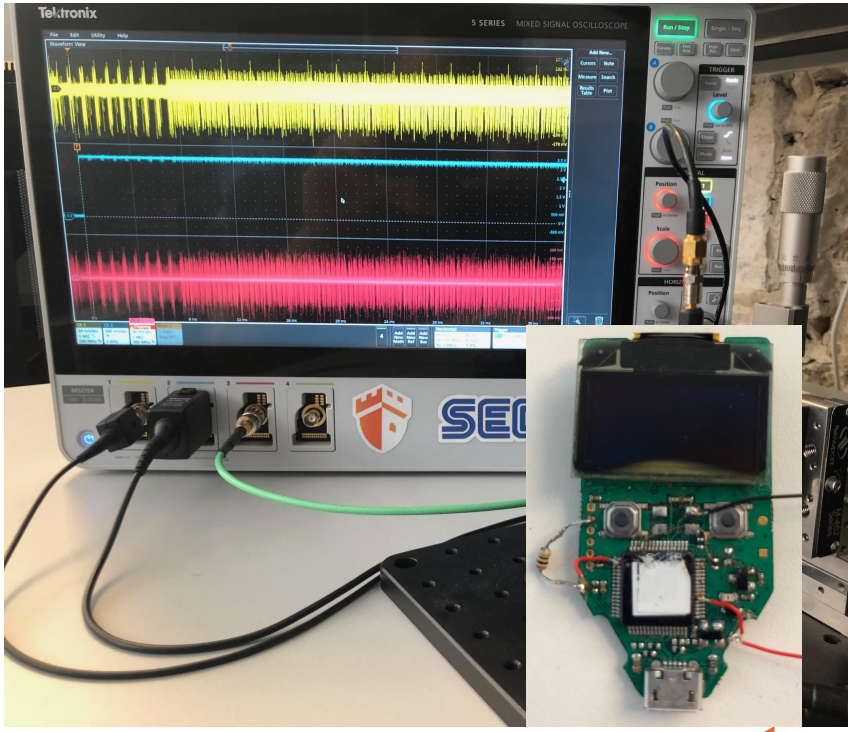


Side-Channel/Setup

- **Observer** et **exploiter** une dépendance entre:
  - Des données physiques mesurées (temps de calcul, conso de courant, EM, ...)
  - Des données sensibles/secrètes (un PIN, une clé privée, ...)

- Banc side-channel classique:
  - Cible avec une résistance entre VCC/GND
  - Oscilloscope

- Lascar tout au long de l'étude:
  - Acquisition
  - Synchronisation
  - Analyse en Machine Learning
  - Attaque
  - "Trace Decoder"



Vérification de PIN

Attaque 1: Vérification du PIN

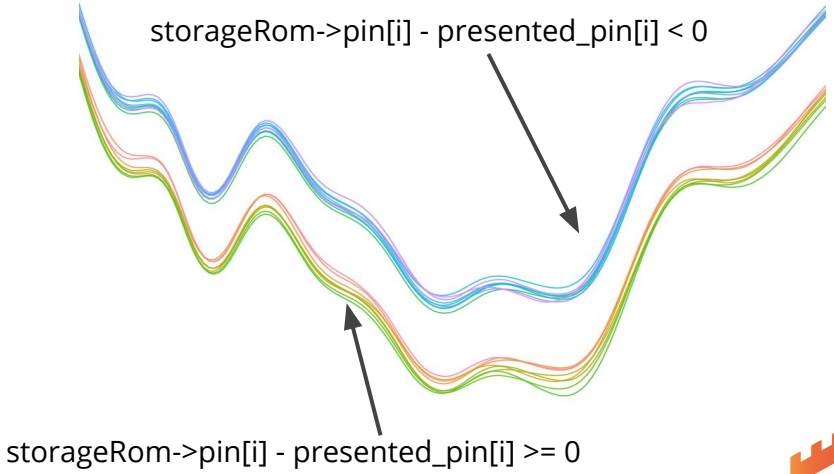
- Fonction critique: le PIN donne accès aux comptes
- Stratégie *Divide & Conquer*: chaque PIN-digit attaqué indépendamment (chaque étape de la boucle *while*)
- Code développé en temps constant

```
bool storage_containsPin(const char *presented_pin)
{
      /* The execution time of the following code only
depends on the
       * (public) input.  This avoids timing attacks.
       */
      char diff = 0;
      uint32_t i = 0;
      while (presented_pin[i]) {
            diff |= storageRom->pin[i] - presented_pin[i];
            i++;
      }
      diff |= storageRom->pin[i];
      return diff == 0;
}
```

Attaque 1: Phase d’apprentissage

- L'accès à un device ouvert nous permet de caractériser la fuite en courant observée, en fonction d’un secret choisi (PIN-digit)
- Contexte propice au Machine Learning
- Construction d’un dictionnaire (base de donnée)
  - PIN digit -> conso de courant

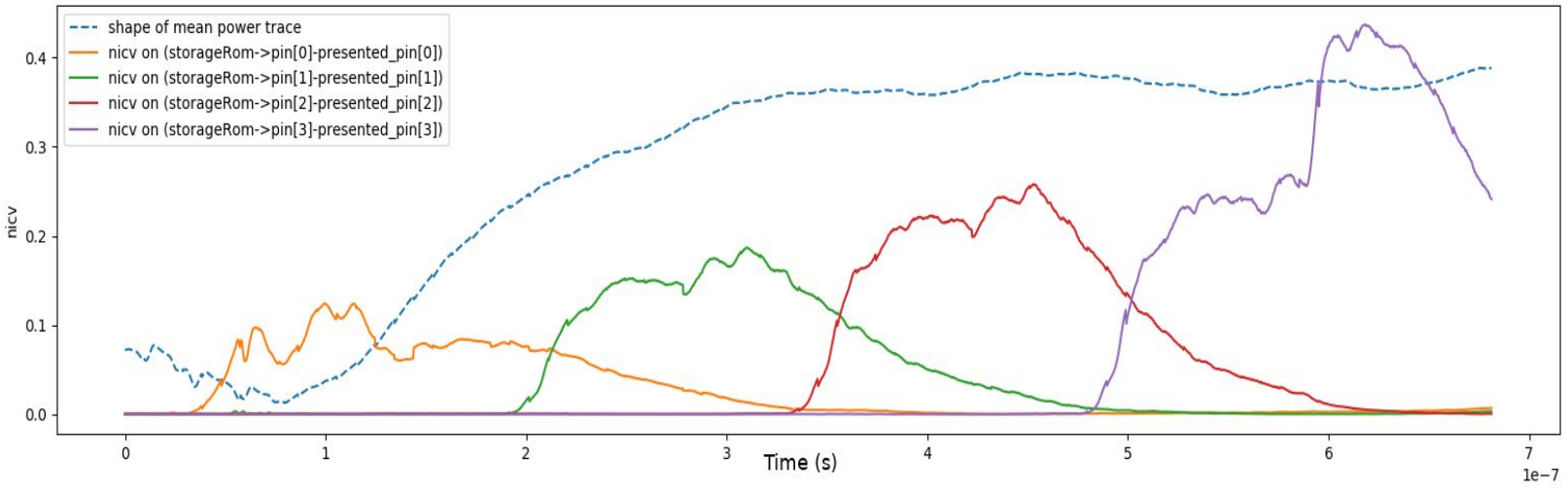
Device A



Caracterisation de la fuite

- Acquisition de traces de consommation de courant.
- Recherche de zones d'intérêt grâce à une analyse de variance

Valeur sensible: storageRom->pin[i] - presented_pin[i] pour 0<=i<4



Phase de Matching

- Sur un nouveau device, acquisition en courant de vérification de PIN
- On utilise le *maximum de vraisemblance* pour faire correspondre cette trace avec une des entrées de nos dictionnaires.
- Notre base de donnée retourne le digit le plus probable, selon elle.

Device B

‘digit 4, proba=0.76’

$$d_{LOG}(k \mid \mathbf{x}_i) = -\frac{1}{2}\log|\mathbf{S}_k| - \frac{1}{2}(\mathbf{x}_i - \bar{\mathbf{x}}_k)'\mathbf{S}_k^{-1}(\mathbf{x}_i - \bar{\mathbf{x}}_k)$$

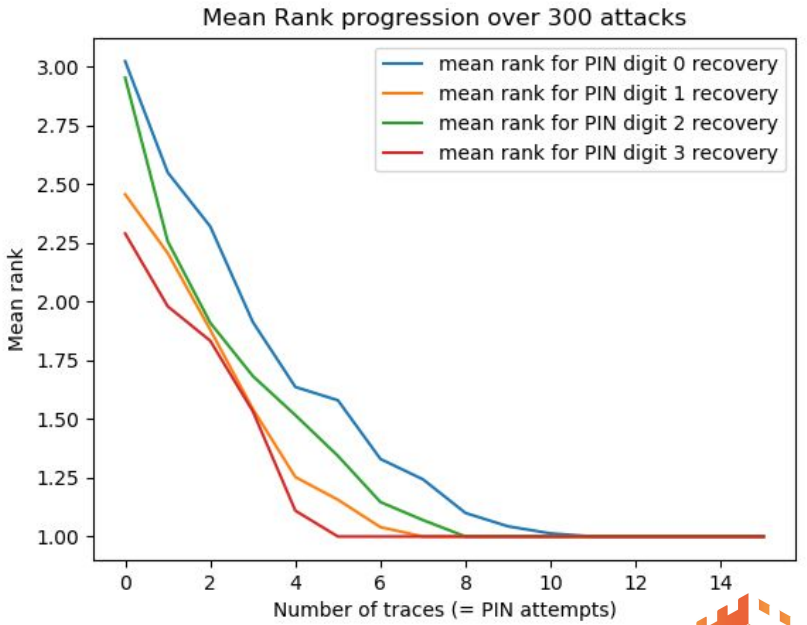
MATHS

Phase de matching: Résultats

- Sur notre nouveau device B:
  - 16 tentatives de PIN dispo avant effacement.
  - On acquiert ces 16 traces en courant
  - … et on les donne à nos bases de données

- Résultats sur 300 attaques (avec PIN random):
  Moins de 11 traces permettent de reconstruire tout le PIN

- Pistes d'amélioration:
  - Travail en EM
  - Davantage de données pour le profilage
  - Utiliser d'autres variables sensibles
  - Réseaux de neurones
  - Stratégie d'input adaptative.



Attaque 1 - Vérification de PIN: En résumé

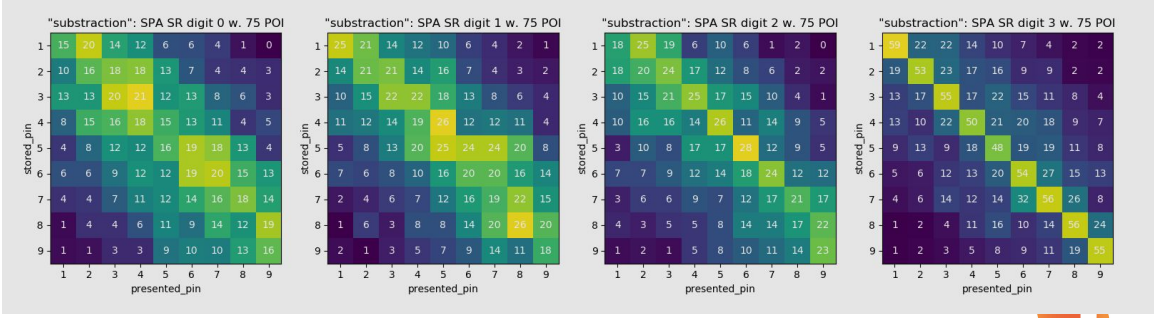
Device A

Device B

- Phase 1. Apprentissage sur device A - 1 DB par PIN-digit
- Phase 2. Sur un device B *volé:*
  - Entrer 11 PINs aléatoire, tout en mesurant le courant.
  - Passer les phases de matching: le PIN est reconstruit
  - Entrer ce PIN sur le device (il reste 4 essais) - Accédez au comptes crypto

Peut-on faire mieux?

- La performance de la phase de matching dépend du PIN stocké **et** du PIN présenté.
- Un apprentissage dépendant du PIN présenté améliore les résultats.



Attaque 1 - Vérification de PIN: En résumé

Device A

Device B

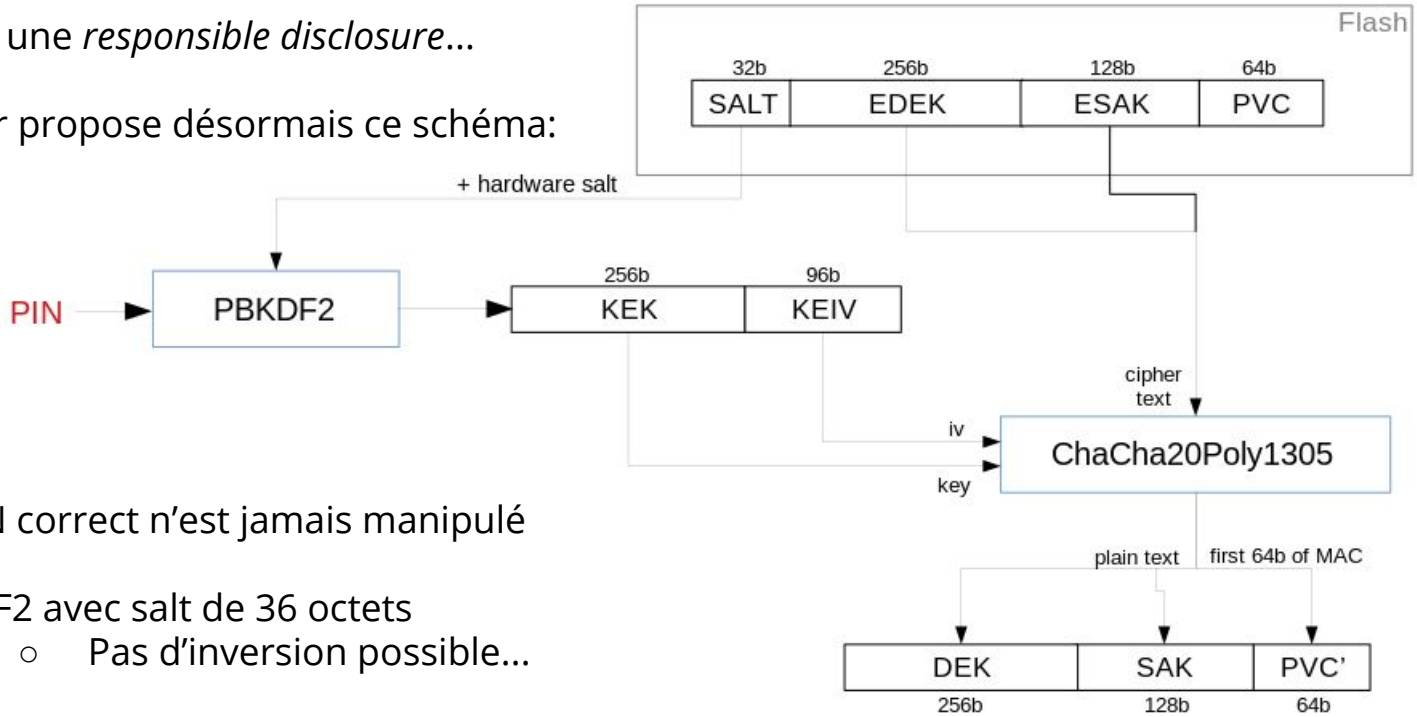
- Phase 1. Apprentissage sur device A- 9 Base de donnée pour chaque chiffre
- Phase 2. Sur un device B
  - Entrer le PIN “5555” - Mesure de la consommation de courant - Phase de Matching
  - Matching - On trouve la valeur de PIN la plus probable “cand“
  - Entrer le PIN le plus probable “cand“ - Mesure de la consommation de courant
  - Profiter

```
cand = “5555”
measuredTraces = []
while !logged:
    measuredTraces += testLogin(cand)
    cand = matching(measuredTraces)
enjoy()
```

=> 4.8 essais pour s’authentifier en moyenne

Attaque 1: Solution

- Après une *responsible disclosure*...
- Trezor propose désormais ce schéma:



- Le PIN correct n'est jamais manipulé
- PBKDF2 avec salt de 36 octets
    - Pas d'inversion possible...

Multiplication scalaire (ECDSA)

Attaque 2: Multiplication scalaire sur courbe elliptique

- Pour la plupart des cryptomonnaies, une transaction nécessite un ECDSA (~ une mult scalaire)
- Le scalaire utilisé pour ECDSA est un nonce, et sa connaissance permet de reconstruire la clé privée.
- 4-bit NAF (Non-Adjacent Form), avec recodage du scalaire
- Le scalaire *k* est transformé puis éclaté en 64 quartets:

  a = k + 2^256 (mod curve->order)
  a = [a[0], ... a[63]]

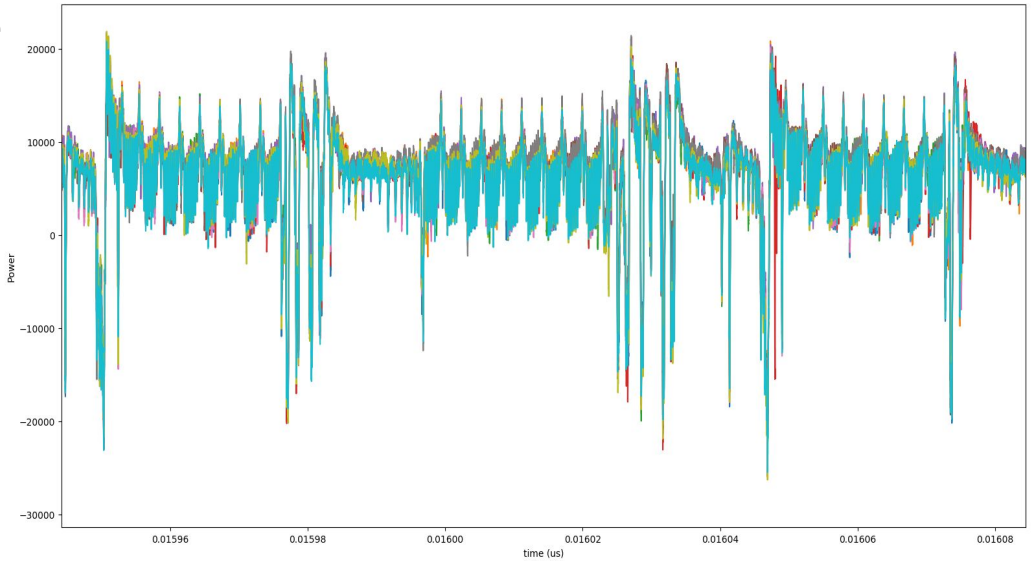
- Stratégie *Divide & Conquer* - A chaque étape de la boucle *for*:
  - On récupère *(sign^nsign)* (1 bit)
  - On récupère *(bits >>1)* (3 bits)

Ce qui permet de reconstruire *a* puis *k*.

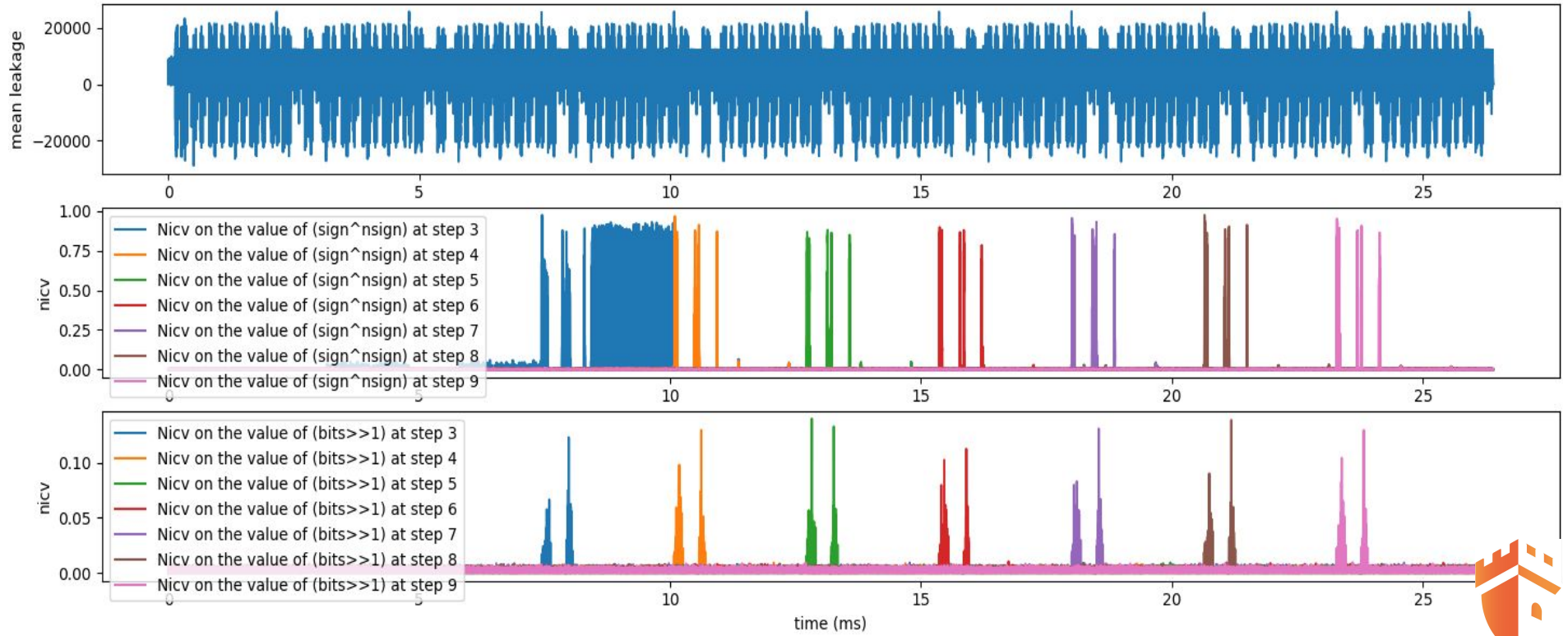
```
// simplified pseudo-code of point_multiply():
// this implem is not correct but is used to give
// an idea of what the code is actually doing.
point_multiply(bignum256 k, curve_point P)
{
      // scalar transformation:
      a = k + 2^256 (mod curve->order)

     //pmult precomputation...
      pmult[8] = [ P, 3P, 5P, ...,  15P];

      Q = P;
     //main loop:
      for (i = 62; i >= 0; i--) {
         nsign, sign, bits = process(a[i], a[i-1])
         Q = [16]Q;
         conditional_negate(sign ^ nsign, Q.z);
         point_jacobian_add(pmult[bits >> 1], Q);
      }
      return Q;
}
```

Caractérisation de la fuite

- Limitation due à la profondeur mémoire de l'oscilloscope.
  (premières étapes de la boucle)
- Toutes les étapes de l'algo sont identifiables sur la trace en courant
- Un peu de post-processing pour resynchroniser les traces
  (pic-à-pic, alignement élastique)

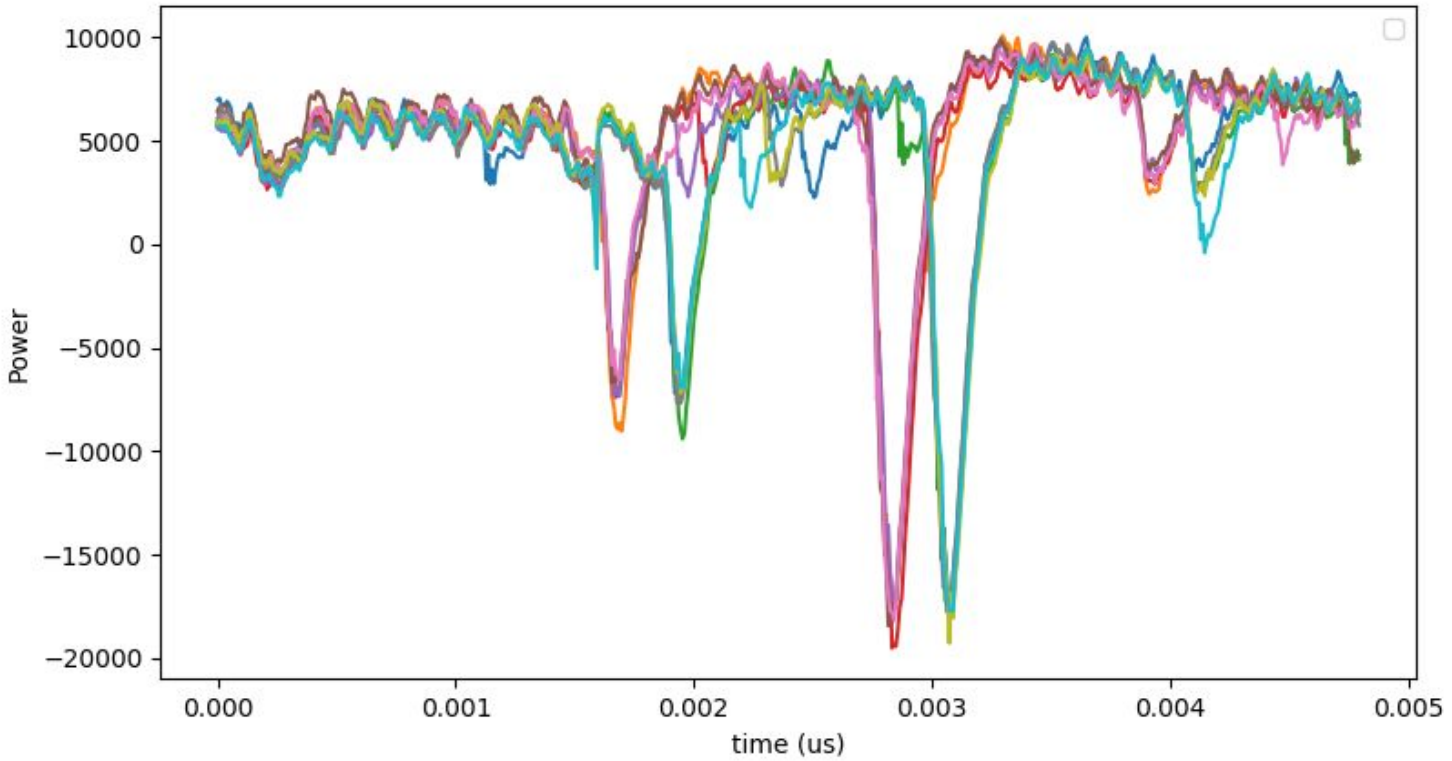


Caractérisation de la fuite



Partie 1 - Retrouver *(sign^nsign)* avec une fuite en timing

(0.98 en valeur pic de NICV semble louche…)



Partie 2 - Retrouver le *(bits>>1)* avec une attaque profilée

- Attaque profilée identique à celle sur le PIN.
  - 64 attaques indépendantes (une par étape de la boucle)
  - Récupération des 64 valeurs ‘(bits>>1)’

- **99% de taux de succès en utilisant une seule trace**

- Conclusion sur l’attaque de la multiplication scalaire
  - Attaque en timing pour retrouver les 64 valeurs de (sign^nsign)
  - Attaque profilée pour retrouver les 64 valeurs de (bits>>1)

Une seule trace est nécessaire pour reconstruire le scalaire
(donc pour reconstruire le nonce)
(donc pour reconstruire la clé privée)

Conclusion

Conclusion

- Deux attaques Side Channel Profilées sur un Hardware Wallet
- Le PIN de l'utilisateur est retrouvé en ~5 essais
- La multiplication scalaire révèle une fuite permettant de retrouver le secret en une seule exécution
    - Cette vulnérabilité n'est pas critique dans ce cas car le PIN est demandé en premier lieu
    - **Mais** dangereux car c'est une librairie Open Source et revendiquée comme résistante aux SCA
        - Le code pourrait être réutilisé dans d'autres projets comme brique sécurisée (authentication, key exchange…)

Tools

Outils

- **Lascar**

  https://github.com/Ledger-Donjon/lascar (LGPL v3.0)

  - Boîte à outils d'analyses par canaux auxiliaires

- **Rainbow** 🌈

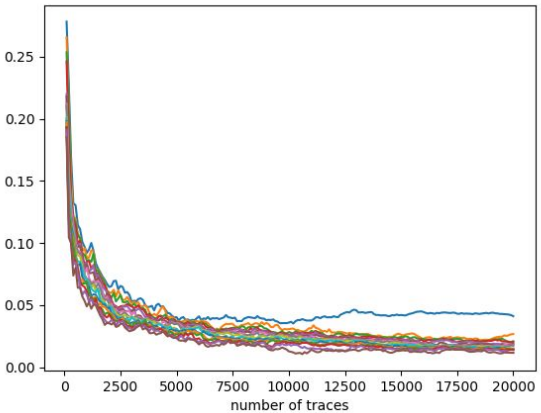
  https://github.com/Ledger-Donjon/rainbow (LGPL v3.0)

  - Simulateur de fuite par canal auxiliaire

Outil: Lascar

- Cadriciel pour les attaques Side-Channel
  - DPA, CPA, Template, Neural Networks, ...
  - Caracterisation : NICV, T-test, ...
  - Trace pre-processing
- Tout en Python, et conçu pour être facilement étendu
- PLusieurs exemples + vérification de PIN Trezor disponible sur Rainbow

```python
from lascar import *

container = Hdf5Container("my-data.h5")

selection_function = lambda value, secret: aes.sbox[value[3] ^ secret]
engine = CpaEngine("cpa_byte_3", selection_function, secret_range=range(256))

output_method = ScoreProgressionOutputMethod(engine)

session = Session(container, engine, output_method).run()
```



Outil: Rainbow

- Simulation de fuite de courant basé sur Unicorn
- Utilisés pour analyser des parties de firmware/morceaux de code sans contexte

- Rejouer l'analyse de variance sur le PIN en simulation :

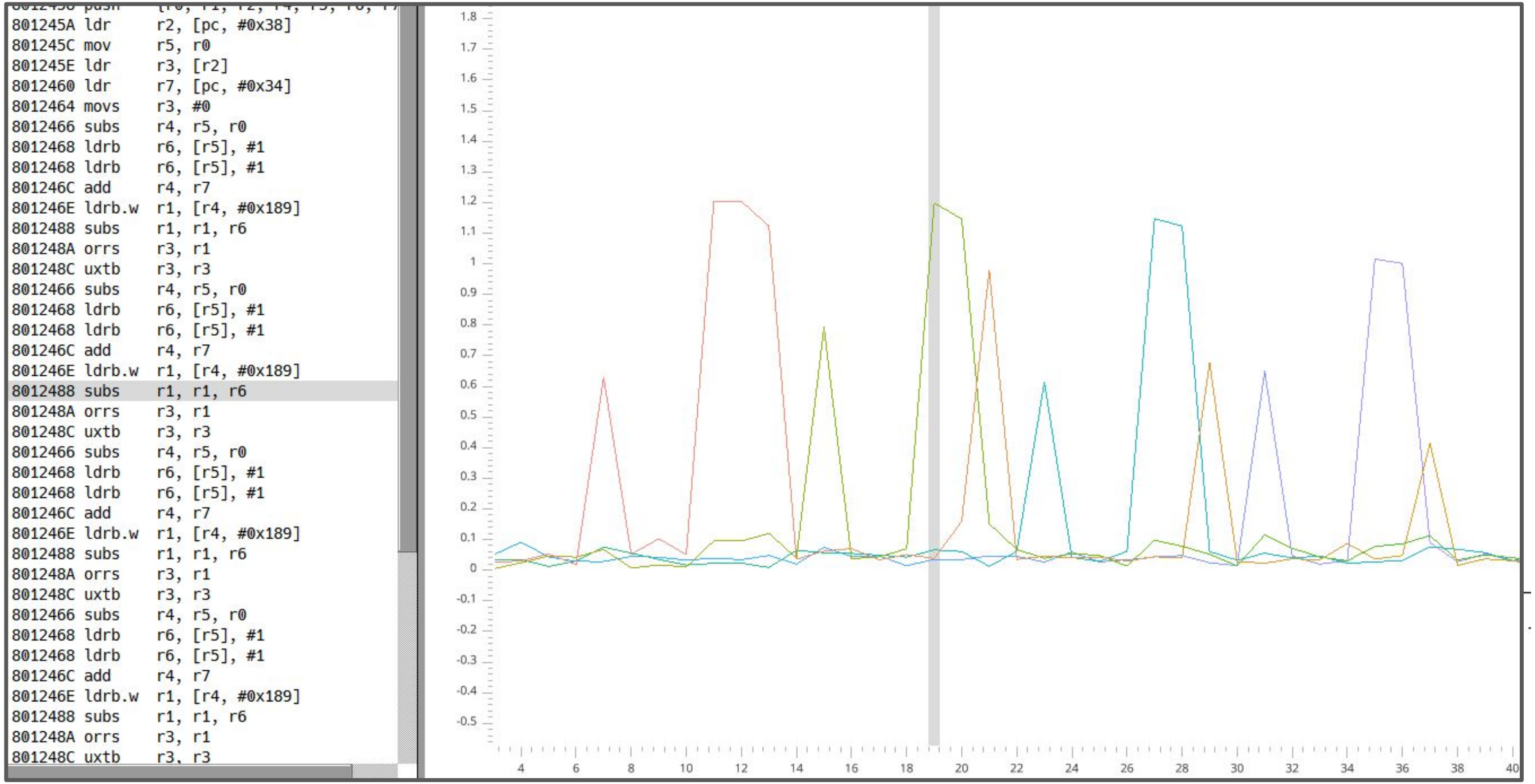
    - Charger dans la mémoire d'Unicorn le ELF du firmware
    - Appeler directement la fonction depuis Python
    - *Hook* des modifications de registre donnent une valeur 32bit par instruction

- Un peu plus facile et rapide que d'utiliser un oscilloscope (500 traces prend moins de 3 secondes)

- On peut prendre le poids de Hamming de ces valeurs, ajouter du bruit, ...

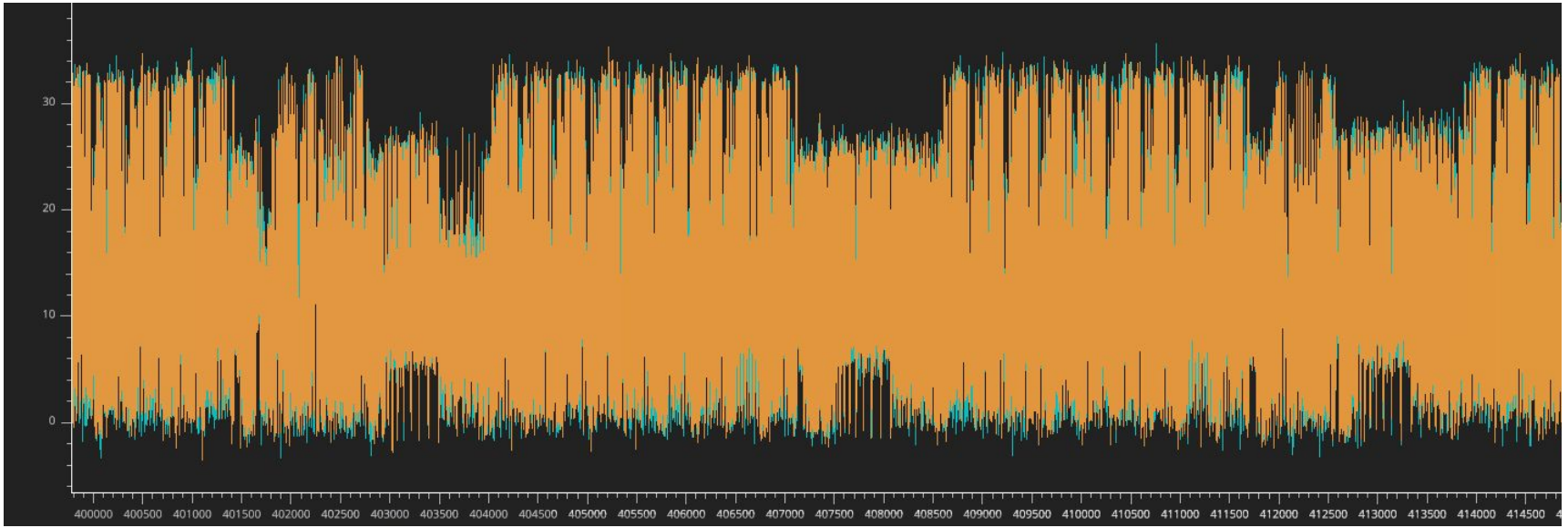
Outil: Rainbow

Ledger



Outil: Rainbow

- Multiplication scalaire : un peu plus lourd



- Embedded Software / Firmware / Crypto-lib

charles@ledger.fr / https://www.ledger.com/join-us/

Questions?

https://github.com/Ledger-Donjon