Trumping The Elephant

Fast Side-Channel Key-Recovery on Dumbo

Louis Vialar

École Polytechnique Fédérale de Lausanne

Kudelski Security Recherche

SSTIC - Juin 2022

Le concours de cryptographie légère du NIST

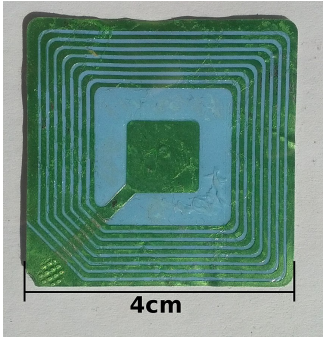


Figure 1: Balise RFID

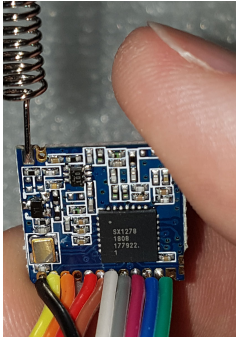


Figure 2: Module LoRa

Certains appareils ont moins de puissance de calcul que des appareils d'usage général, et ont des contraintes/menaces différentes.

La cryptographie légère conçoit et étudie des algorithmes cryptographiques pour ces appareils.

Trois objectifs (contradictoires) :

- **Faible coût** : consommer moins d'énergie, utiliser moins de portes logiques (en surface)
- **Performance** : augmenter le débit
- **Sécurité** : résister aux attaques algorithmiques *et* physiques (canaux auxiliaires)

Compétitions Cryptographiques

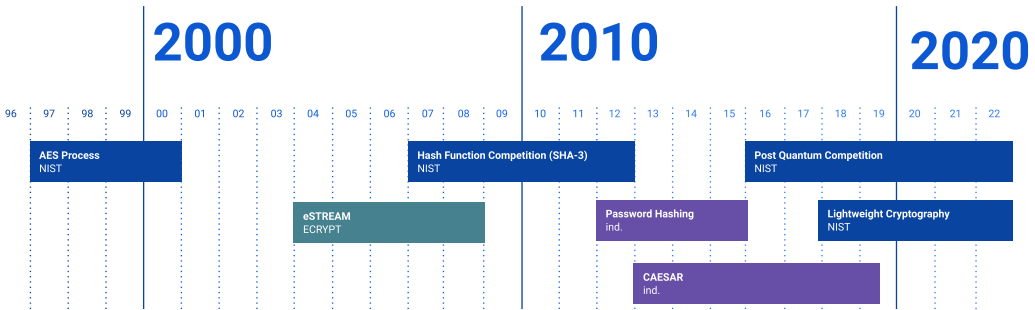


Figure 3: Petit historique de quelques compétitions cryptographiques

La compétition du NIST (NIST Lightweight Cryptography)

- Pas de standard spécifique pour la crypto légère
  - Enjeu : FIPS
- AES-GCM est assez bon dans la majorité des cas, mais la protection contre les canaux auxiliaires coûte cher
- 2 décennies de recherche en crypto légère depuis AES : peut-être qu'il existe des solutions plus efficaces

La compétition du NIST (NIST Lightweight Cryptography)

- Pas de standard spécifique pour la crypto légère
  - Enjeu : FIPS
- AES-GCM est assez bon dans la majorité des cas, mais la protection contre les canaux auxiliaires coûte cher
- 2 décennies de recherche en crypto légère depuis AES : peut-être qu'il existe des solutions plus efficaces
- 2018, le NIST annonce une compétition pour standardiser un nouvel algorithme de chiffrement AEAD léger.

La compétition du NIST (NIST Lightweight Cryptography)

- Pas de standard spécifique pour la crypto légère
  - Enjeu : FIPS
- AES-GCM est assez bon dans la majorité des cas, mais la protection contre les canaux auxiliaires coûte cher
- 2 décennies de recherche en crypto légère depuis AES : peut-être qu'il existe des solutions plus efficaces
- 2018, le NIST annonce une compétition pour standardiser un nouvel algorithme de chiffrement AEAD léger.
- 2021, 10 finalistes sont annoncés, dont Éléphant

Présentation d'Elephant-160 (Dumbo)

Qu'est-ce qu'Elephant ?

- Algorithme de chiffrement symmétrique authentifié avec données associées
- *Encrypt-then-MAC* : on chiffre les données *puis* on les authentifie

Variante	Primitive	Clé	Nonce	État
Dumbo	Spongent	16 o	12 o	20 o
Jumbo	Spongent	16 o	12 o	22 o
Delirium	Keccak	16 o	12 o	25 o

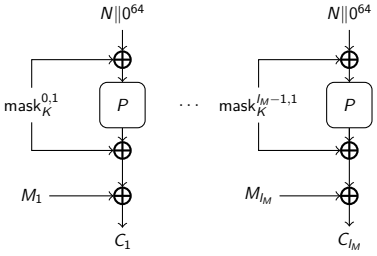
Qu'est-ce qu'Elephant ?

- Algorithme de chiffrement symmétrique authentifié avec données associées
- *Encrypt-then-MAC* : on chiffre les données *puis* on les authentifie

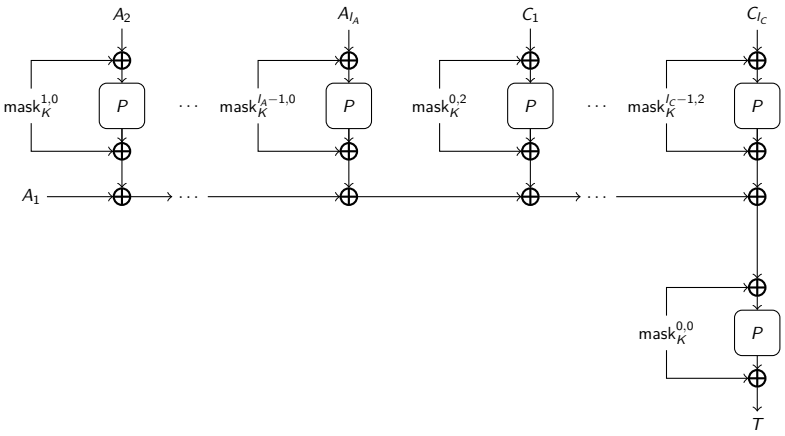
Variante	Primitive	Clé	Nonce	État
Dumbo	Spongent	16 o	12 o	20 o
Jumbo	Spongent	16 o	12 o	22 o
Delirium	Keccak	16 o	12 o	25 o

- Aujourd'hui : **Dumbo**, la variante principale d'Elephant

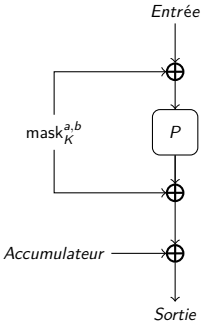
Chiffrement



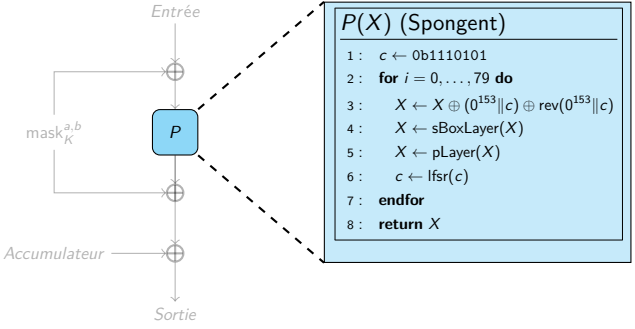
Authentification



Briques de construction



Briques de construction



Briques de construction

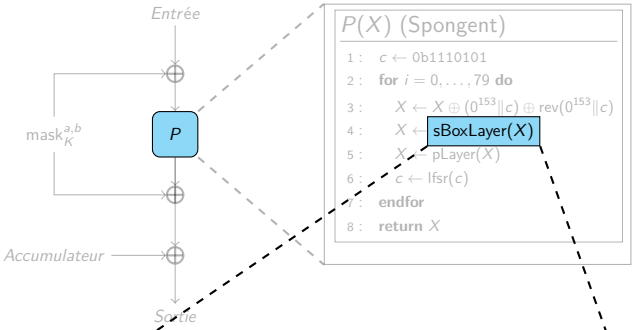
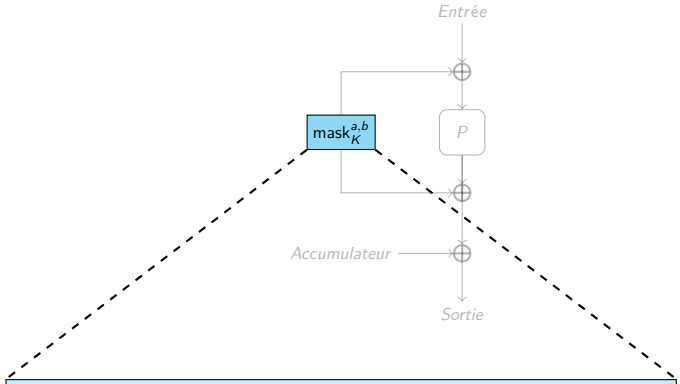


Table 2: S-Box de Spongent

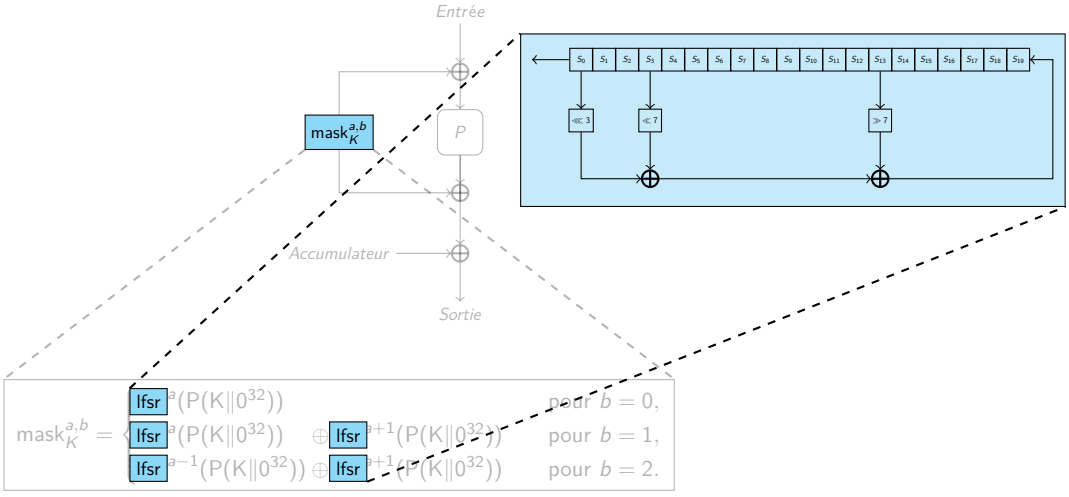
X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(X)	E	D	B	0	2	1	4	F	7	A	8	5	9	C	3	6

Briques de construction



$$\text{mask}_K^{a,b} = \begin{cases} \text{lfsr}^a(\text{P}(\text{K}\|0^{32})) & \text{pour } b = 0, \\ \text{lfsr}^a(\text{P}(\text{K}\|0^{32})) \oplus \text{lfsr}^{a+1}(\text{P}(\text{K}\|0^{32})) & \text{pour } b = 1, \\ \text{lfsr}^{a-1}(\text{P}(\text{K}\|0^{32})) \oplus \text{lfsr}^{a+1}(\text{P}(\text{K}\|0^{32})) & \text{pour } b = 2. \end{cases}$$

Briques de construction



Récapitulons

- Les données associées sont authentifiées en utilisant

$$\mathsf{mask}_K^{a,0} = \mathsf{lfsr}^a(\mathsf{P}(\mathsf{K} \| 0^{32}))$$

Récapitulons

- Les données associées sont authentifiées en utilisant

$$\text{mask}_K^{a,0} = \text{lfsr}^a(\text{P}(\text{K} \| 0^{32}))$$

- lfsr est inversable : on peut retrouver $\text{mask}_K^{a-1,0}$ depuis $\text{mask}_K^{a,0}$.

Récapitulons

- Les données associées sont authentifiées en utilisant

$$\mathrm{mask}_K^{a,0} = \mathrm{lfsr}^a(\mathrm{P}(\mathrm{K}\|0^{32}))$$

- lfsr est inversable : on peut retrouver $\mathrm{mask}_K^{a-1,0}$ depuis $\mathrm{mask}_K^{a,0}$.
- P est inversable : on peut retrouver $K\|0^{32}$ depuis $\mathrm{P}(\mathrm{K}\|0^{32})$.

Idée d'attaque

Retrouver $\mathrm{mask}_K^{1,0}$ **durant l'authenticiation des données associées. Inverser** *lfsr* **puis** P **pour retrouver** K**.**

Point d'attaque

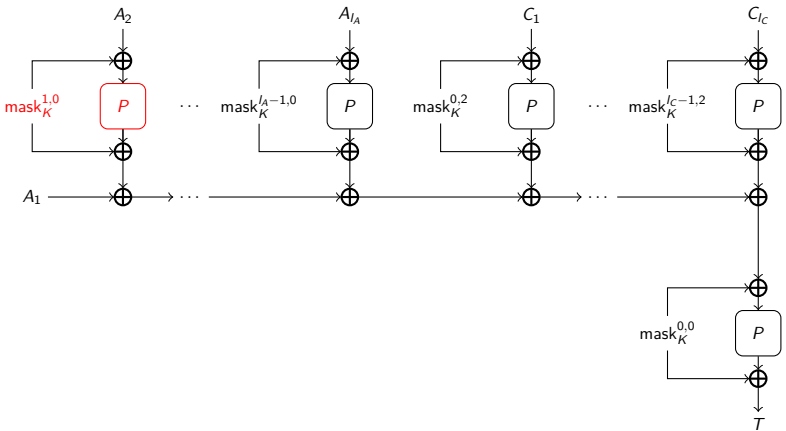


Figure 4: Authentification dans Dumbo, avec le point d'attaque

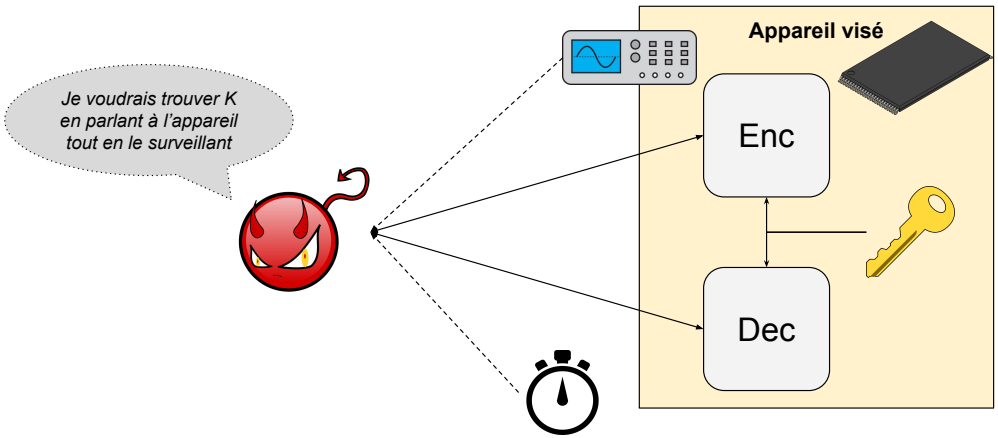
$P(X)$ (Spongent-π[160])

1 : $c \leftarrow$ 0b1110101
2 : **for** $i = 0, \ldots, 79$ **do**
3 : $\quad X \leftarrow X \oplus (0^{153} \| c) \oplus \mathrm{rev}(0^{153} \| c)$
4 : $\quad X \leftarrow \mathrm{sBoxLayer}(X)$
5 : $\quad X \leftarrow \mathrm{pLayer}(X)$
6 : $\quad c \leftarrow \mathrm{lfsr}(c)$
7 : **endfor**
8 : **return** X

Plus précisément, notre point d'attaque sera la SBox de Spongent.

Notre attaque par CPA

Attaque par canaux auxiliaires



Analyse de consommation

Une catégorie d'analyse en canaux auxiliaires basée sur l'analyse de la consommation de courant.

Intuition : changer la valeur d'un bit ne consomme pas la même énergie suivant sa valeur

$\Longrightarrow$ la consommation énergétique d'une puce dépend des données qu'elle traite (en l'occurrence, du poids de Hamming)

Modèle de consommation pour la i-ème SBox

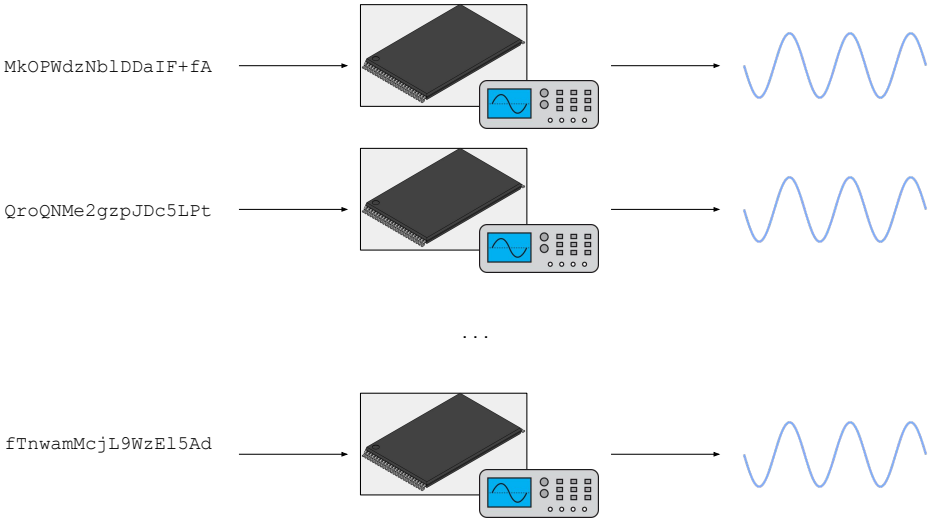
$Model_i(k, a)$

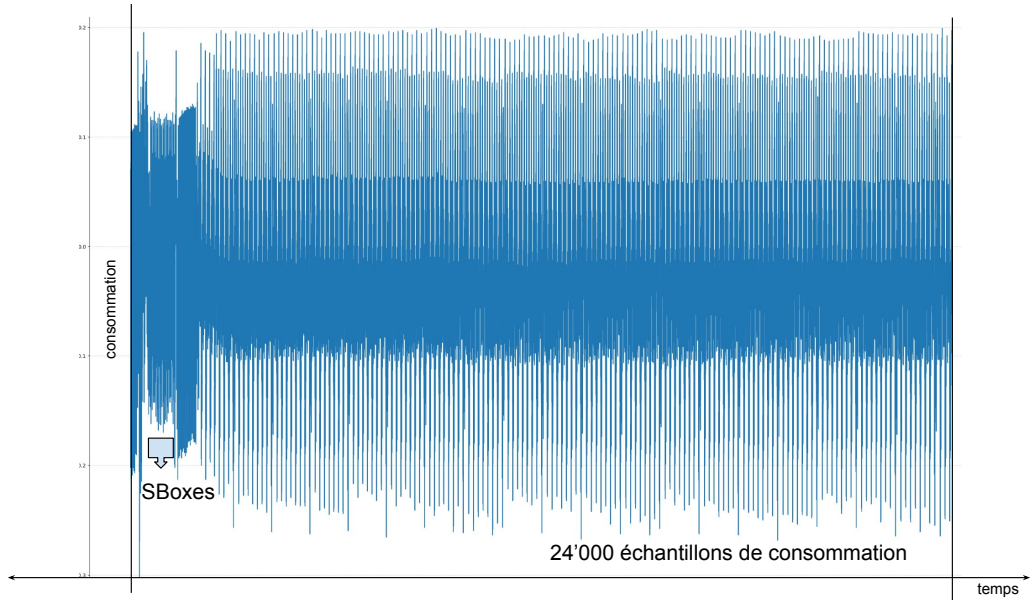
1 : // Ajouter l'octet du masque à l'octet de données avant d'entrer dans la permutation
2 : $S \leftarrow a \oplus k$
3 : // Première opération du premier tour: ajouter c aux premier et dernier octets
4 : **if** $i = 0$ **then**
5 : $S \leftarrow S \oplus 0x75$ // `0b1110101`
6 : **elseif** $i = 19$ **then**
7 : $S \leftarrow S \oplus 0xae$ // `0b1110101` à l'envers
8 : **endif**
9 : // Seconde opération du premier tour: sBoxLayer
10 : $S \leftarrow \text{SBox}(S)$
11 : **return** $\text{HammingWeight}(S)$

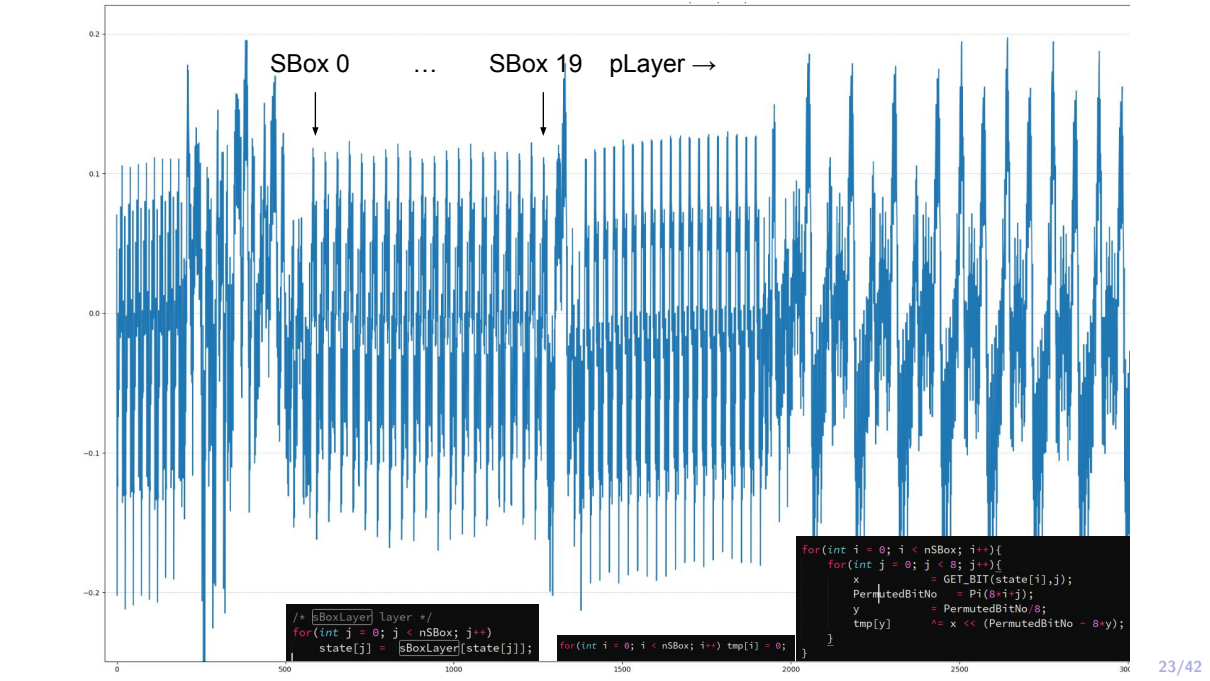
Idée

Si on peut déterminer le poids de hamming exact de l'opération ciblée pour quelques valeurs de a, on peut calculer le modèle avec les 256 valeurs possibles pour k jusqu'à ce qu'on trouve celle qui donne toujours la bonne valeur.

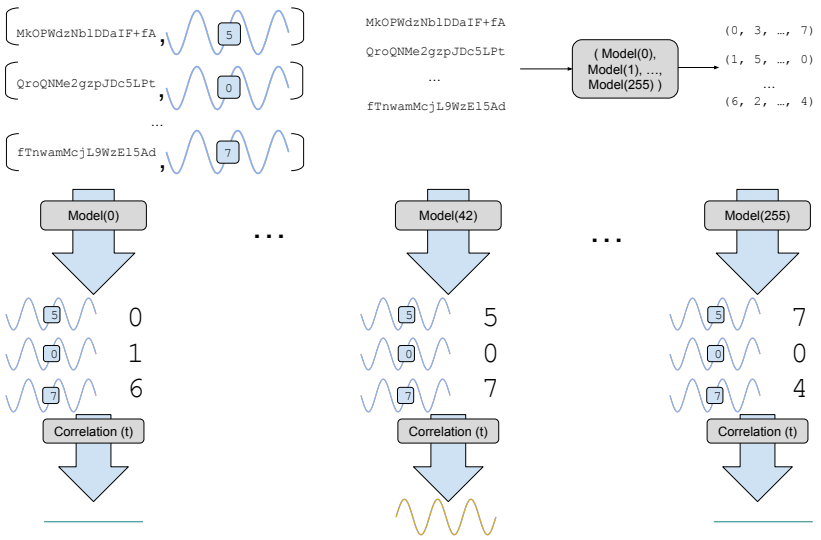
Étape 1 : capturer des traces

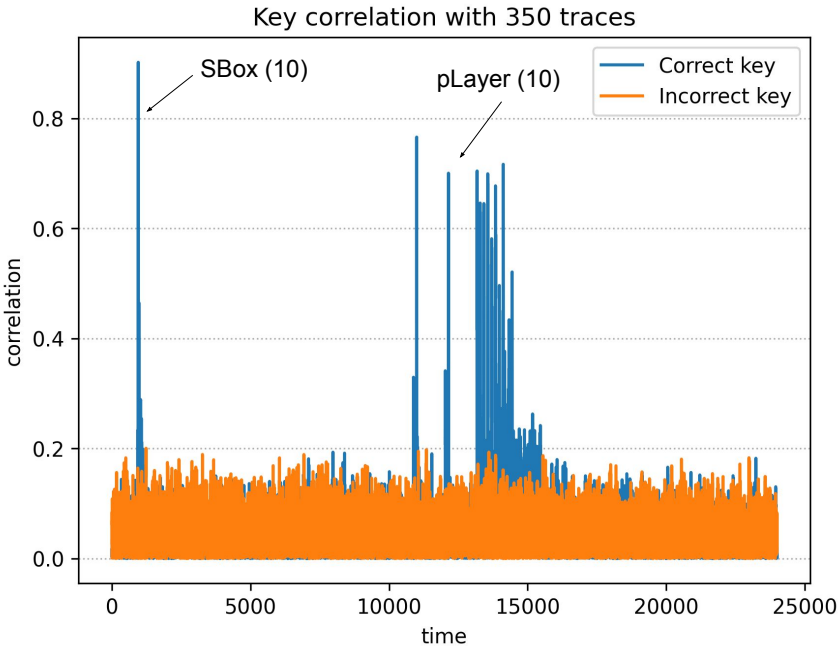



consommation
SBoxes
24’000 échantillons de consommation
temps

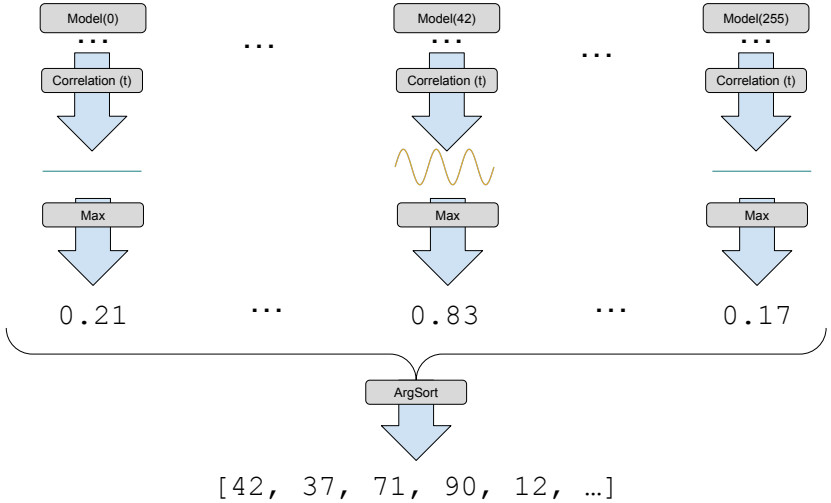

SBox 0
…
SBox 19
pLayer →
0.2
0.1
0.0
−0.1
−0.2
0
500
1000
1500
2000
2500
/* sBoxLayer layer */
for(int j = 0; j < nSBox; j++)
state[j] = sBoxLayer[state[j]];
for(int i = 0; i < nSBox; i++) tmp[i] = 0;
for(int i = 0; i < nSBox; i++){
for(int j = 0; j < 8; j++){
x = GET_BIT(state[i],j);
PermutedBitNo = Pi(8*i+j);
y = PermutedBitNo/8;
tmp[y] ^= x << (PermutedBitNo - 8*y);
}
}

Étape 2 : calcul de corrélation




Key correlation with 350 traces
SBox (10)
pLayer (10)
Correct key
Incorrect key
correlation
0.0
0.2
0.4
0.6
0.8
0
5000
10000
15000
20000
25000
time

Étape 3 : tri des hypothèses



Retrouver la clé

```
def recherche_exhaustive(valeurs):
  for nb_erreurs in range(4):
    for position_erreurs in itertools.combinations(range(20), nb_erreurs):
      masques_possibles = itertools.product(*[ \
        valeurs[i] if i in position_erreurs \
        else [valeurs[i][0]] \
        for i in range(20)])

      for masque in masques_possibles:
        perm_k = inverser_lfsr(masque)
        k = inverser_spongent(perm_k)

        if k[16:] == ([0] * 4):
          return bytes(k[:16])
  return None
```

Configuration expérimentale

- Attaque menée sur ChipWhisperer Lite ARM
  - Facile à utiliser et bon pour la reproducibilité
- Ordinateurs classiques (i7-4790K, i7-8565U) pour les calculs.
- Implémentation de référence (en C) fournie par les auteurs au NIST
  - Compilée avec la suite d'outils ChipWhisperer et optimisation -03

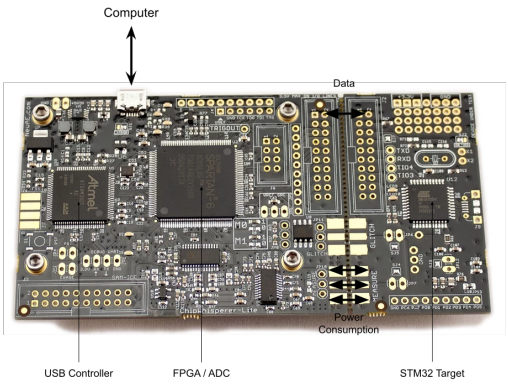


Figure 5: La plateforme CWLite ARM

Résultats

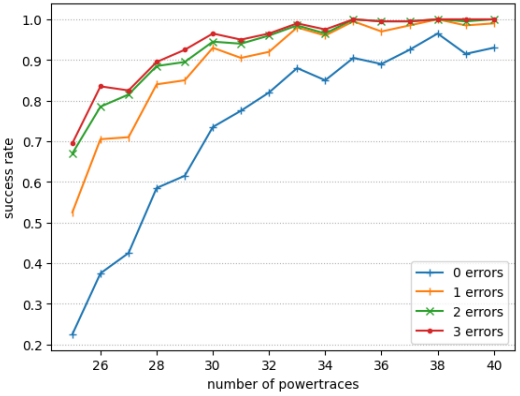


Figure 6: Taux de succès de l'attaque (0 - 1)

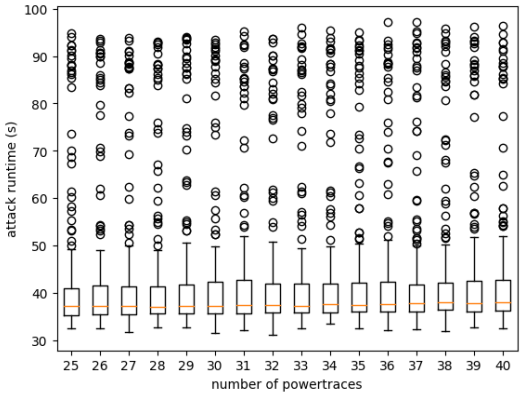


Figure 7: Temps d'exécution de l'attaque (s)

Contremesures

Des contremesures ont été dévelopées à destination des implémentations matérielles par Abdulgadir et al. dans **Side-Channel Resistant Implementations of Three Finalists of the NIST Lightweight Cryptography Standardization Process: Elephant, TinyJAMBU, and Xoodyak**.

Contremesures

Des contremesures ont été dévelopées à destination des implémentations matérielles par Abdulgadir et al. dans **Side-Channel Resistant Implementations of Three Finalists of the NIST Lightweight Cryptography Standardization Process: Elephant, TinyJAMBU, and Xoodyak**.

Coût :

- 4.22 ratio de nombre de portes
- 0.43 ratio de débit
- 1.22 ratio de consommation estimé
- 2.52 ratio de consommation par bit ratio

Conclusion

Conclusion

- CPA efficace ($<$ 40 traces) sur Elephant-Dumbo
  - Fonctionne aussi sur Jumbo avec quelques changements mineurs
- Attaque **l'implémentation logicielle de référence**
  - Pas protégée, pas optimisée pour notre matériel cible
  - Des implémentations plus optimisées demanderont probablement plus de traces mais resteront vulnérables
- Des contremesures existent sur les implémentations matérielles mais ont un impact en performances

Questions?

Graphics Attribution

https://commons.wikimedia.org/wiki/File:Crypto_key_topicon.svg
https://commons.wikimedia.org/wiki/File:Evil_red.svg
https://commons.wikimedia.org/wiki/File:CPT-sound-nyquist-thereom-raw.svg
https://commons.wikimedia.org/wiki/File:TSOP-48_Blank.svg
https://commons.wikimedia.org/wiki/File:Symbol_oscilloscope.svg
https://en.wikipedia.org/wiki/File:
LoRa_Module_with_antenna_and_SPI_wires_attached.jpg
https://commons.wikimedia.org/wiki/File:Simpleicons_Business_stopwatch.svg

Les images de conseillers fédéraux proviennent de la Chancellerie Fédérale de la Confédération Helvétique.

Extraits de code

```c
int main(void)
{
  platform_init();
  init_uart();
  trigger_setup();

  simpleserial_init();
  simpleserial_addcmd('r', 0, reset);
  simpleserial_addcmd('k', 16, set_key);
  simpleserial_addcmd('p', 17, set_plaintext_block);
  simpleserial_addcmd('a', 17, set_ad_block);
  simpleserial_addcmd('n', 16, encrypt_with_nonce);

  while(1)
    simpleserial_get();
}
```

```
void encrypt_with_nonce(uint8_t* nonce, uint8_t len)
{
  uint8_t out[528] = {0}; // 512 + potential 16 bytes tag
  long long unsigned int ct_size = 0;

  trigger_high();
  crypto_aead_encrypt(out, &ct_size, pt_bytes, pt_len, ad_bytes,
    ad_len, 0, nonce, key);
  trigger_low();

  uint8_t len_out[2] = {0};
  len_out[0] = (ct_size >> 8) & 0xff;
  len_out[1] = (ct_size) & 0xff;

  simpleserial_put('r', 2, len_out);
  simpleserial_put('r', ct_size, out);
}
```

```
def capture_traces(n_traces, cap_first_offset):
    block_size = 20
    first_ad_block_size = block_size - 12

    # Générer 28bits de données associées aléatoires
    ad_gen = lambda: np.random.randint(0, 256, first_ad_block_size + \\
       block_size, np.uint8)

    # Le nonce est sans importance, on peut le laisser à 0
    nonce_gen = lambda: np.zeros(12, np.uint8)

    # Paramétrer le ChipWhisperer
    target.read()
    scope.adc.timeout = 2
    scope.adc.samples = 24000
    scope.adc.offset = cap_first_offset
```

```
    traces, values = [], []
    for _ in range(n_traces):
        target.flush() ; wrap.reset()
        nonce, ad = nonce_gen(), ad_gen()

        target.simpleserial_write('a', to_bytes(ad))
        scope.arm()
        target.simpleserial_write('n', to_bytes(nonce, 12) + b'\x00' * 4)

        # Récupère et stoque la trace
        scope.capture()
        traces.append(scope.get_last_trace())
        values.append(ad[first_ad_block_size:first_ad_block_size + \\
           block_size])

    return traces, values
```

```
def classifier_ad(position, ad: np.ndarray, key_byte):
    ad_byte = numba.u1(ad[position])
    state = numba.u1(key_byte) ^ ad_byte
    result = classifier(state, position)

    iv = numba.u1(lfsr_iv)
    state = numba.u1(state)

    if position == 0:
        state ^= iv
    elif position == state_bytes - 1:
        state ^= numba.u1(reverse_bits(iv))

    return hamming(sbox[state])
```

```
def run_cpa(captured):
    eng = [
        lascar.CpaEngine(f'cpa{byte}',
          lambda ad, guess, index=byte: classifier_ad(index, ad, guess),
          range(256))
        for byte in range(20)
    ]

    lascar.Session(captured, engines=eng).run(batch_size="auto")
    results = [engine.finalize() for engine in eng]

    possible_keys = []
    for index, r in enumerate(results):
        possible_keys.append(abs(r).max(1).argsort()[::-1][:8])

    return possible_keys
```

```
def attack(num_traces, load_file, save_file, verbosity, wrap):
    print("Key to attack:\t\t", key.hex(" ", 1).upper())
    key = random.randbytes(16)
    wrap.set_key(key)

    print("[1] Capturing traces on device")
    captured = capture_traces(num_traces, cap_first_offset = 974000)

    print("[2] Run correlation analysis on captured traces")
    masks = run_cpa(captured)

    print("[3] Find the correct result")
    res = exhaustive_search(pt, nonce, ct, masks)

    print("[=>] Got:\t\t\t\t", res.hex())
    print("[=>] Expected:\t\t\t\t", key.hex())
```