

Sécurité des implémentations pour la cryptographie

Partie 4 : Résistance aux attaques distantes

Benoît Gérard
9 janvier 2017



Plan du cours

Étape 1

Définition du besoin et de l'architecture au niveau système.

Étape 2

Définition de l'interface carte/terminal : API exposée par la carte.

Étape 3

Implémentation d'une version résistante aux attaques non-crypto.

Étape 4

Implémentation d'algo. crypto. résistante aux attaques distantes.

Étape 5

Implémentation d'algo. crypto. résistante aux attaques locales.

Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

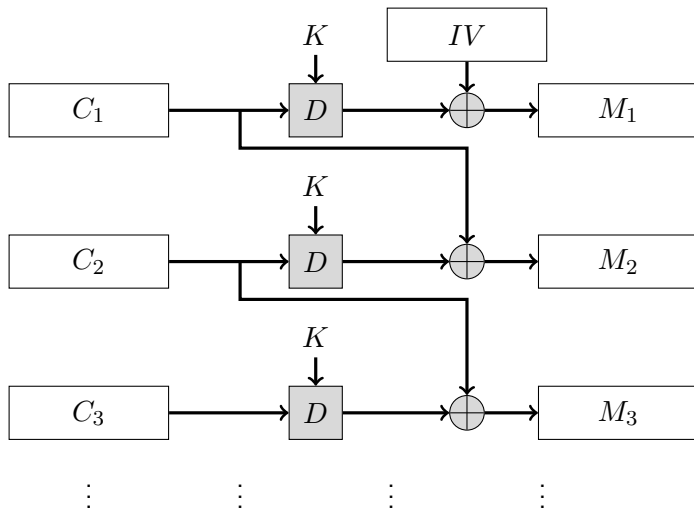
- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

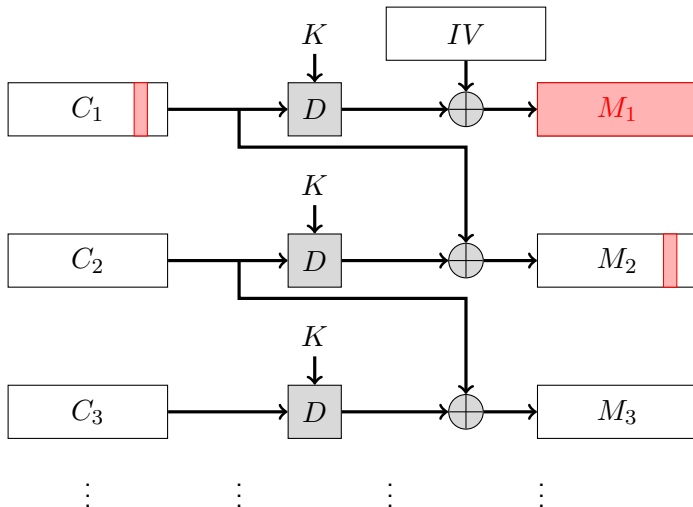
Malléabilité du mode CBC

Principe



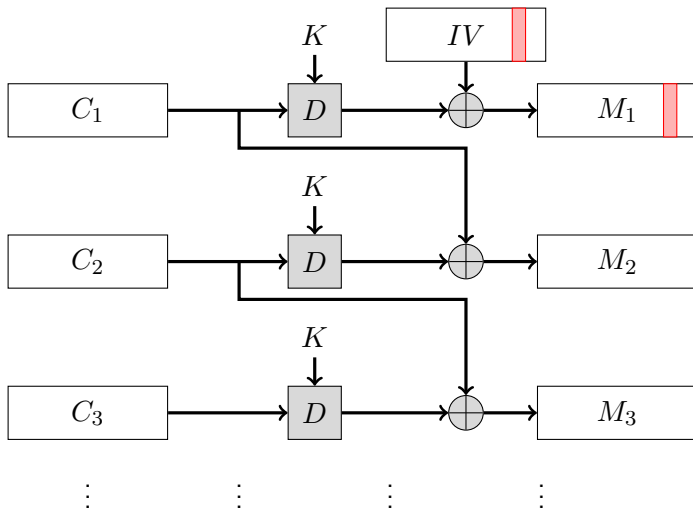
Malléabilité du mode CBC

Principe



Malléabilité du mode CBC

Principe



Cryptographie dans IPsec

- ▶ AH (authentification et intégrité),
- ▶ ESP (chiffrement + intégrité),
- ▶ IKE (échange de clefs).

Intégrité :

- ▶ Mac-then-Encrypt,
- ▶ Encrypt-then-MAC.

Chiffrement avec ESP

- ▶ mode CBC
- ▶ padding

L'attaquant peut modifier l'IV \Rightarrow modification de l'en-tête

- ▶ change l'adresse d'origine,
- ▶ modifie le champ protocole (invalide).

La gateway traite le paquet (déchiffre l'en-tête)

- ▶ protocole invalide \rightarrow message ICMP,
- ▶ message ICMP envoyé à l'adresse d'origine (donc à l'attaquant maintenant),
- ▶ l'attaquant reçoit le message ICMP si la checksum est valide.

Malléabilité du mode CBC

Attaque sur ESP

L'attaquant peut modifier l'IV \Rightarrow modification de l'en-tête

- ▶ change l'adresse d'origine,
- ▶ modifie le champ protocole (invalide).

La gateway traite le paquet (déchiffre l'en-tête)

- ▶ protocole invalide \rightarrow message ICMP,
- ▶ message ICMP envoyé à l'adresse d'origine (donc à l'attaquant maintenant),
- ▶ l'attaquant reçoit le message ICMP si la checksum est valide.

Pour debug une partie du message clair est transmis!!!

Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

Padding Oracle : IPSEC en mode ESP

Format d'une trame ESP

Trame ESP en mode tunnel :

texte à chiffrer	1	2	...	taille	4
------------------	---	---	-----	--------	---

Padding effectué en suffixant par

- ▶ des octets 1, 2, 3, ...
- ▶ un octet contenant le nombre d'octets ajoutés,
- ▶ un dernier octet (à 4 pour le mode tunnel).

Le dernier bloc de chiffré contiendra donc un des padding en fin de bloc :

- | | |
|-----------------|----------------------------------|
| ▶ 0 4 | ▶ 1 1 4 |
| ▶ 1 2 2 4 | ▶ 1 2 3 3 4 |
| ▶ ... | ▶ 1 2 ... 13 14 14 4 |

Padding Oracle : IPSEC en mode ESP

Attaque 1/3

On a reçu un paquet que l'on souhaite déchiffrer :

en-tête	C_0^*	C_1^*	C_2^*	\dots	C_n^*
---------	---------	---------	---------	---------	---------

On intercepte un autre paquet chiffré avec la même clef :

en-tête	C_0	C_1	C_2	\dots	C_n
---------	-------	-------	-------	---------	-------

Pour déchiffrer C_i^* on va envoyer des paquets de la forme :

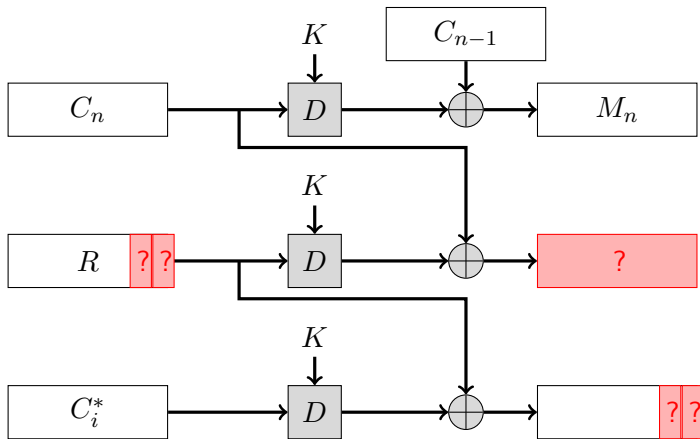
en-tête	C_0	C_1	C_2	\dots	C_n	R	C_i^*
---------	-------	-------	-------	---------	-------	-----	---------

L'attaquant va jouer sur R pour déchiffrer C_i^* .

Padding Oracle : IPSEC en mode ESP

Attaque 2/3

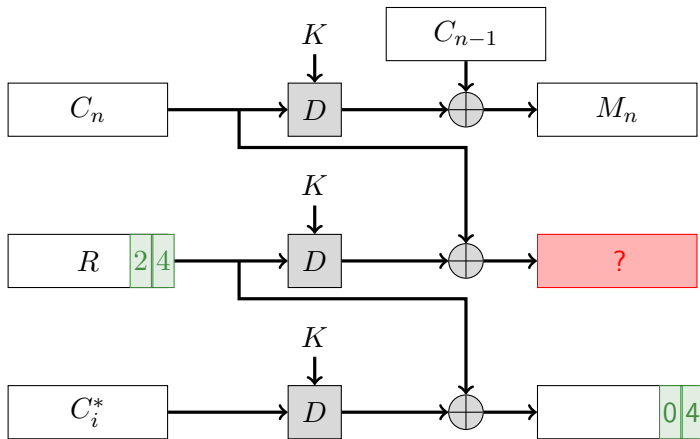
Padding
0—4



Padding Oracle : IPSEC en mode ESP

Attaque 2/3

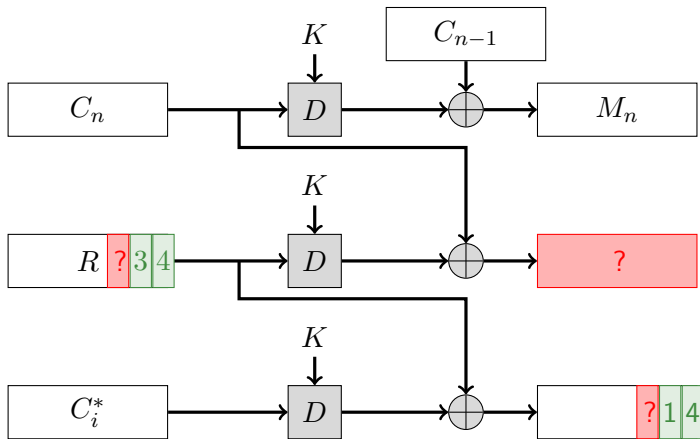
Padding
0—4



Padding Oracle : IPSEC en mode ESP

Attaque 2/3

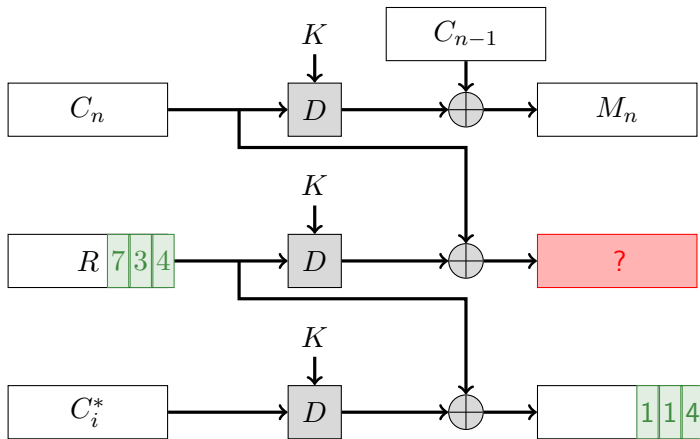
Padding
1—1—4



Padding Oracle : IPSEC en mode ESP

Attaque 2/3

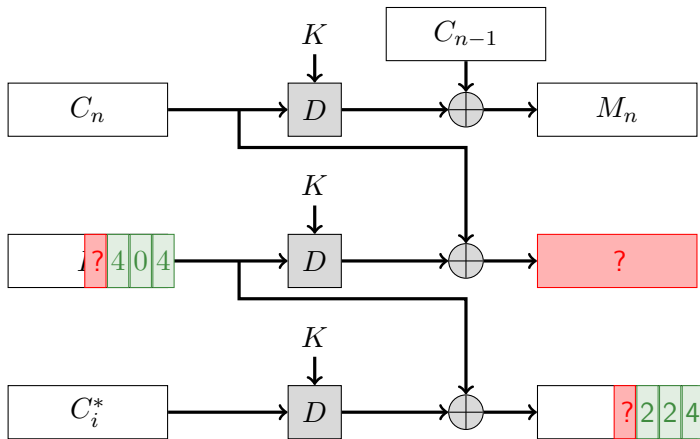
Padding
1—1—4



Padding Oracle : IPSEC en mode ESP

Attaque 2/3

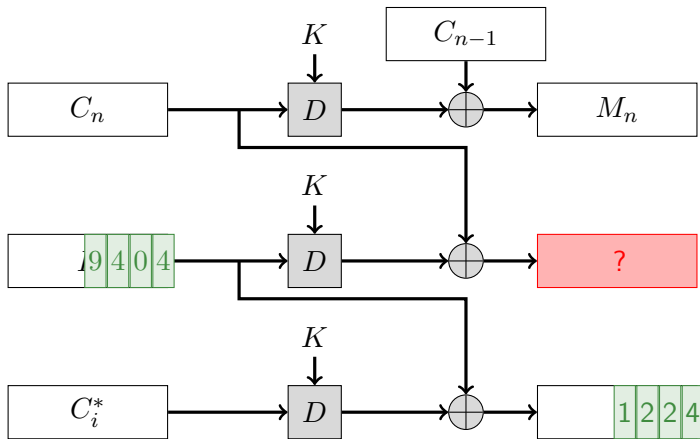
Padding
1—2—2—4



Padding Oracle : IPSEC en mode ESP

Attaque 2/3

Padding
1—2—2—4



Ainsi on termine avec $R = (r_0, r_1, \dots, r_{15})$ tel que :

$$D_K(C_i^*) \oplus R = (1, 2, 3, 4, \dots, 14, 14, 4)$$

On obtient donc

$$P_i^* = C_{i-1}^* \oplus D_K(C_i^*) = C_{i-1}^* \oplus (r_0 \oplus 1, r_1 \oplus 2, \dots, r_{15} \oplus 4)$$

Complexité de l'attaque

En moyenne on a

- ▶ 2^{15} requêtes pour la première étape,
- ▶ 2^7 requêtes pour les 14 suivantes.

Soit un total de $34560 = 2^{15.08}$ requêtes.

Padding Oracle : IPSEC en mode ESP

Instanciation de l'oracle pour IPsec

Malheureusement ... l'oracle existe :

- ▶ block valide : réponse au paquet.
- ▶ block invalide : pas de réponse.

Plusieurs difficultés techniques potentielles résolues.

Par exemple :

- ▶ Mécanisme d'authentification "en dessous"
 - ▶ annulé grâce à la fragmentation de paquets IP.

Solution

Si on fait de l'*Encrypt-then-MAC* alors le MAC sera invalide et on n'aura pas d'oracle de format.

Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

RSA Chiffrement

$$c = m^e \mod N$$

RSA Déchiffrement

$$m = c^d \mod N$$

Attaque : retrouver m à partir de c

1. s aléatoire $\rightarrow c' = c s^e \mod N$,
2. envoi de c' à l'oracle de déchiffrement,
3. $m' = (c')^d = c^d s^{ed} = m s \mod N$,
4. calcul de $m = m' s^{-1} \mod N$.

Padding oracle sur RSA PKCS#1

Principe

0x00	0x02	<i>non-zero bytes</i>	0x00	<i>data</i>
------	------	-----------------------	------	-------------

TABLE: Padding défini dans PKCS#1.

Modèle d'attaquant

- ▶ Pas d'oracle de déchiffrement.
- ▶ Oracle sur la validité du padding.

Information obtenue

Si m' a un padding correct alors

$$2B \leq m s \bmod N \leq 3B,$$

avec $B = 2^{\text{len}(N)-16}$.

Padding oracle sur RSA PKCS#1

Attaque

Objectif

Retrouver $m = c^d \bmod N$.

Principe

Trouver des messages $c_i = c s_i^e \bmod N$ dont les valeurs déchiffrées ont un padding valide.

- ▶ Utiliser l'information obtenue sur $m s_i \bmod N$ pour réduire les valeurs possibles de m .
- ▶ $s_0 = 1$ car c est conforme vu que c'est un chiffré valide.
- ▶ À partir de s_i , trouver un s_{i+1} (de plus en plus facile).

Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

- Temps d'exécution d'une comparaison

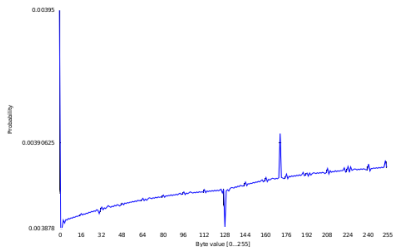
- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

Chiffrement à flots de Ron Rivest :

- ▶ facile à implémenter,
- ▶ rapide,
- ▶ **biaisé !!!**

Keystream distribution at position 2

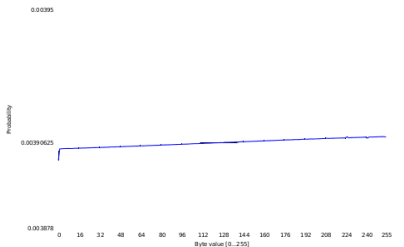


AlFardan et al. (RHUL & UIC)

Biases in RC4

2 / 256

Keystream distribution at position 256



AlFardan et al. (RHUL & UIC)

Biases in RC4

256 / 256

Attaque de type “broadcast” (Usenix 2013)

- ▶ clair fixé,
- ▶ différentes clefs.

Idée de l'attaque

On observe plusieurs c_i correspondant au même p :

$$c_i = p \oplus s_i$$

Comme s_i est biaisé (ici vaut souvent 0x00) alors

$$p = \operatorname{argmax}_{x \in [[0,255]]} \# \{i, c_i = x\}$$

Faiblesses de RC4 impliquées dans des attaques sur

- ▶ WEP,
- ▶ WPA,
- ▶ HIVE (déjà évoqué dans le cours précédent).

Chiffrement à flot sans authentification \Rightarrow malléabilité

- ▶ attaque sur SSH,
- ▶ attaque sur BitTorrent.

Bonus : même suite chiffrente utilisée plusieurs fois (Microsoft Office).

Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

Carte à puce personnelle,

- ▶ fournie par l'administration,
- ▶ permet d'effectuer des démarches administratives (état, services),
- ▶ certificats avec muldo RSA (1024/2048 bits).

Modulo RSA

$N = p \times q$ avec

- ▶ p grand premier secret,
- ▶ q grand premier secret,
- ▶ N grand nombre composé publique.

Remarque

$$\left. \begin{array}{l} N = p \times q \\ N' = p \times q' \end{array} \right\} \implies \text{pgcd}(N, N') = p$$

- ▶ 3 millions de certificats récupérés,
 - ▶ 2.3 millions de moduli RSA 1024 bits
 - ▶ 0.7 millions de moduli RSA 2048 bits
- ▶ pgcd sur toutes les clefs
 - × 103 clefs factorisées !

Exemple de facteurs trouvés

```
c000 0000 0000 0000 0000 0000 0000 [...] 0000 0000 0000 02f9 c924 2492
2492 9249 9249 4924 4924 [...] 9924 9492 4492 424e5
```

Divisions des moduli avec des motifs similaires :

- × 22 nouvelles clefs !

- ▶ de nouveaux facteurs très creux (cf. exemple précédent).

Utilisation de LLL :

- × 39 nouvelles clefs !

- ▶ de nouveaux des facteurs très creux.

Remarque

Un élève de troisième peut usurper l'identité de plus de 100 taïwanais ...

Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

Génération de nonce biaisé dans ECDSA

Présentation d'ECDSA

Paramètres :

- ▶ courbe E d'ordre n ,
- ▶ G point générateur,
- ▶ fonction de hachage H ,
- ▶ message à signer m ,
- ▶ clef privée $s \in [[1; n - 1]]$,
- ▶ clef publique $Q = [s]G$.

Signature :

1. tirer k dans $[[1; n - 1]]$,
2. $(x, \cdot) = [k]G$,
3. si $x = 0$ retourner en 1.
4. $y = k^{-1}(H(m) + s \cdot x) \bmod n$
5. si $y = 0$ retourner en 1.
6. renvoyer (x, y) .

k est appelé *nonce* ou clef éphémère et est sensible !

$$x^{-1}(y \cdot k - H(m)) = s$$

Génération de nonce biaisé dans ECDSA

Impact d'un aléa biaisé

Formule de biais

$$B_n = \frac{1}{L} \sum_{j=1}^L e^{2\pi i k_j / n}$$

Or,

$$k = y^{-1}(H(m) + s \cdot x) \mod n$$

Attaque

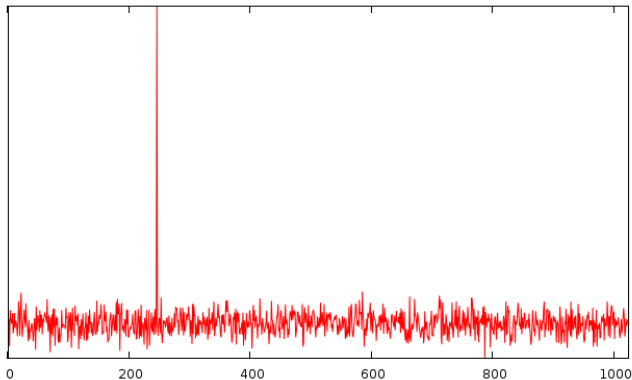
Si la génération des k_j est biaisée :

1. intercepter L signatures “biaisées”,
2. pour chaque s possible calculer le biais,
3. prendre la valeur maximisant le biais.

Génération de nonce biaisé dans ECDSA

Précisions sur l'attaque

Tester chaque valeur de $s \iff$ recherche exhaustive !



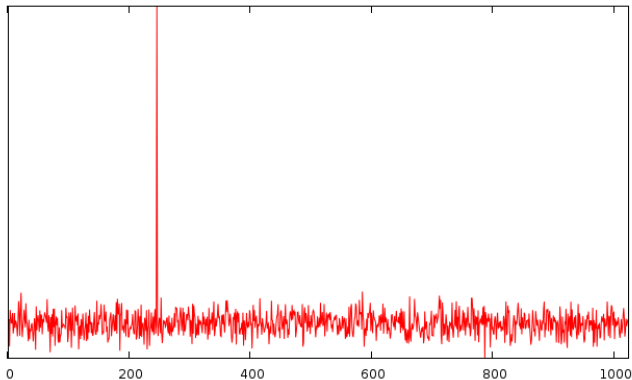
Génération de nonce biaisé dans ECDSA

Précisions sur l'attaque

Tester chaque valeur de $s \iff$ recherche exhaustive !

Solution

Étaler le pic et ne regarder que quelques valeurs puis raffiner.



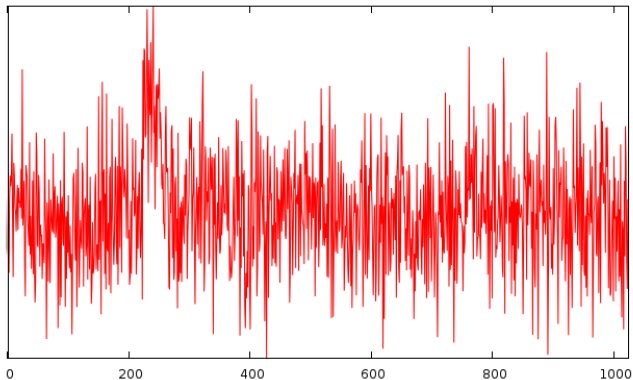
Génération de nonce biaisé dans ECDSA

Précisions sur l'attaque

Tester chaque valeur de $s \iff$ recherche exhaustive !

Solution

Étaler le pic et ne regarder que quelques valeurs puis raffiner.



Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

Problématique

- ▶ Besoin : générer k uniforme dans $[[0; n - 1]]$.
- ▶ À disposition : générateur de bits aléatoire.


On note $\ell = \lceil \log_2(n) \rceil$ (ie. nombre de bits de n). On peut

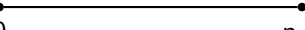
1. générer k uniforme dans $[[0; 2^{\ell-1} - 1]]$,
2. générer k uniforme dans $[[0; 2^\ell - 1]]$ et
 - 2.1 si $k \geq n$ on recommence,
 - 2.2 si $k \geq n$ alors on retourne $k - n$,
3. générer k uniforme dans $[[0; 2^{\ell+\lambda} - 1]]$ et retourner $k \bmod n$.

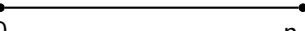
Utiliser 2.1 ou 3 !

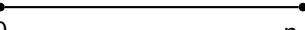
Génération d'aléa uniforme $\bmod n$

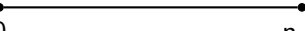
En dessin

But 
0 n-1

1. 
0 n-1


2.1 
0 n-1

2.2 
0 n-1


3. 
0 n-1


Génération d'aléa uniforme mod n


En dessin

But 
0 $n-1$

1. 
0 $2^{\ell}-1$ $n-1$

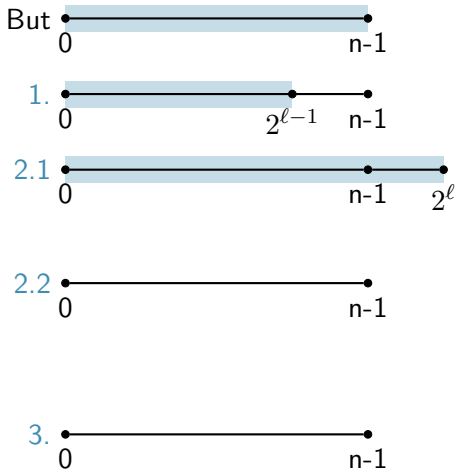
2.1 
0 $n-1$

2.2 
0 $n-1$

3. 
0 $n-1$

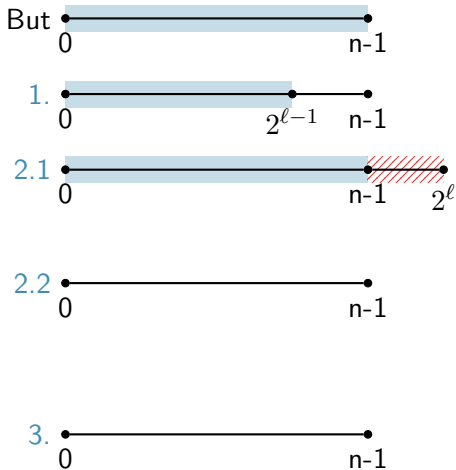
Génération d'aléa uniforme mod n

En dessin



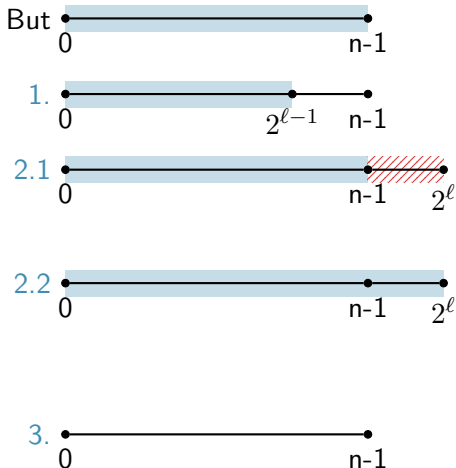
Génération d'aléa uniforme mod n

En dessin



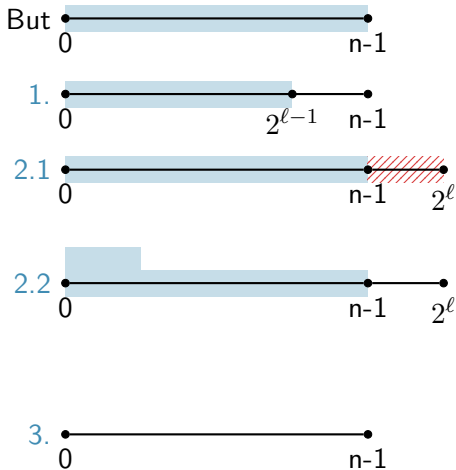
Génération d'aléa uniforme mod n

En dessin



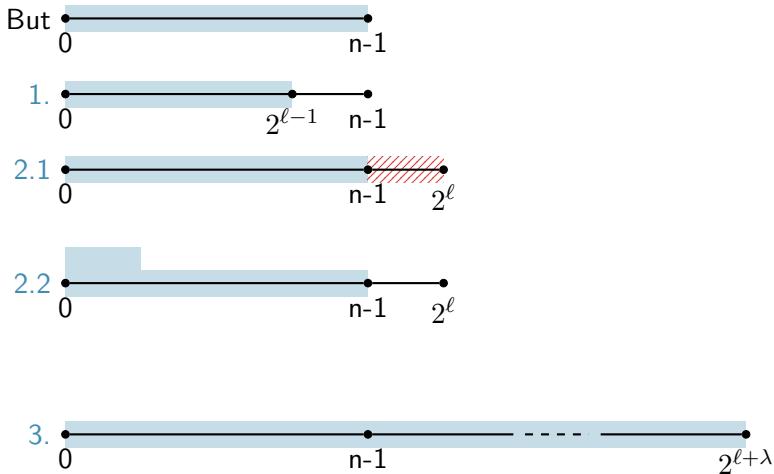
Génération d'aléa uniforme mod n

En dessin



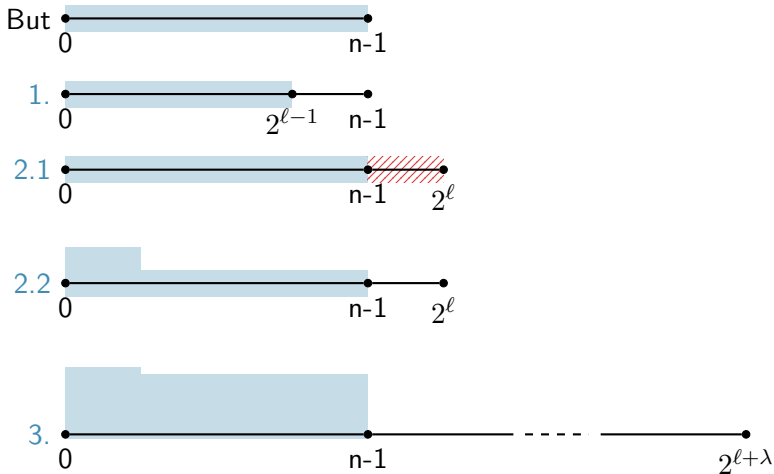
Génération d'aléa uniforme mod n

En dessin



Génération d'aléa uniforme mod n

En dessin



Règle

Le temps d'exécution d'un code ne doit pas dépendre des données sensibles manipulées.

- ▶ parfois très difficile à mettre en oeuvre,
- ▶ en logiciel : dépendant du langage,
- ▶ plus facile si on implémente certaines fonctions en matériel.

Sources de dépendance

Essentiellement (mais pas uniquement) :

- ▶ branchements conditionnels,
- ▶ phénomènes liés au cache.

Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \implies en moyenne 5000 essais.

Si mal implanté \implies au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



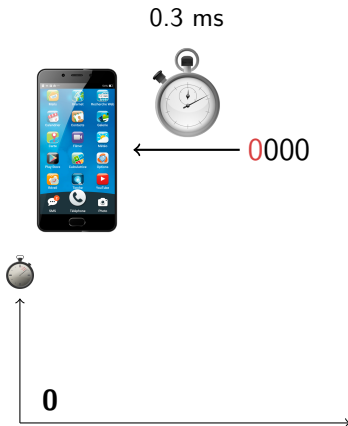
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



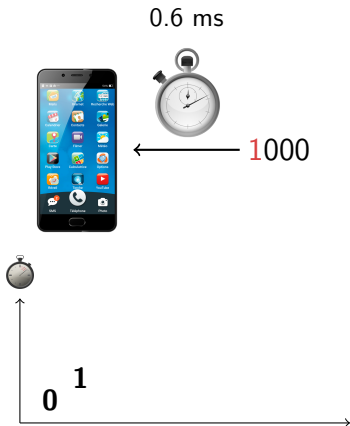
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



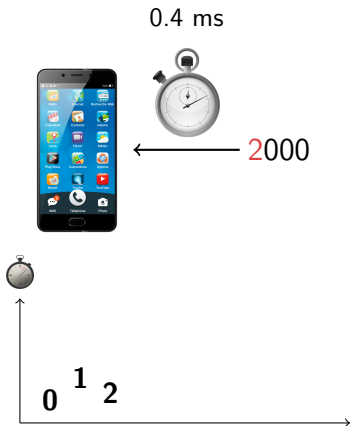
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



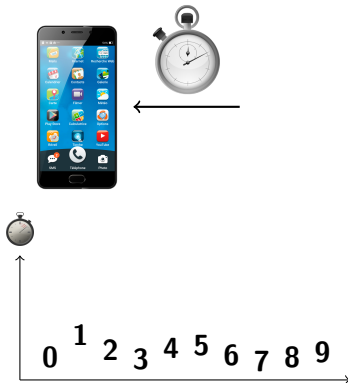
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \implies en moyenne 5000 essais.

Si mal implanté \implies au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



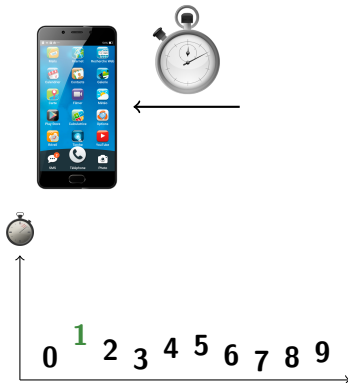
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



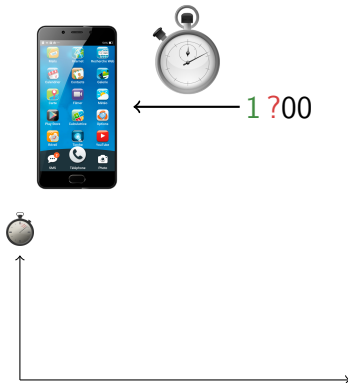
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



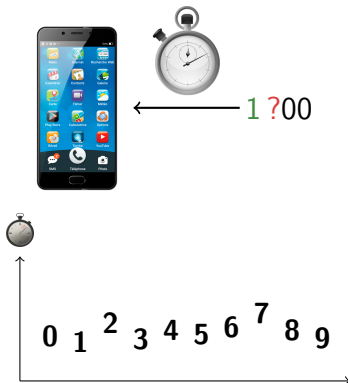
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



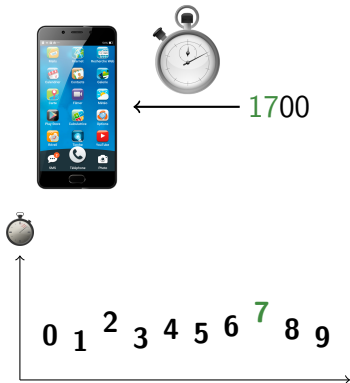
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



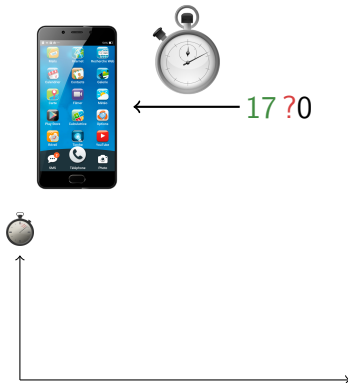
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



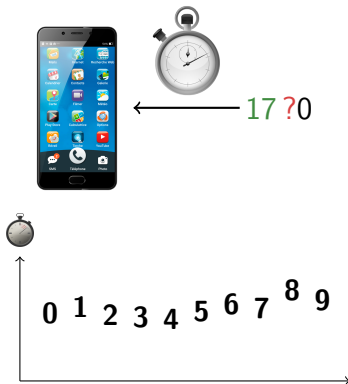
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



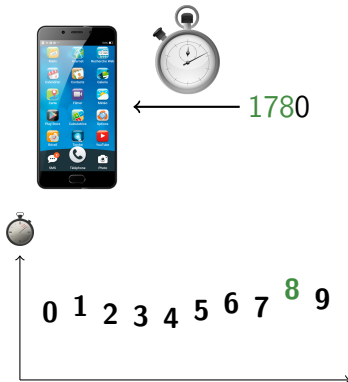
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



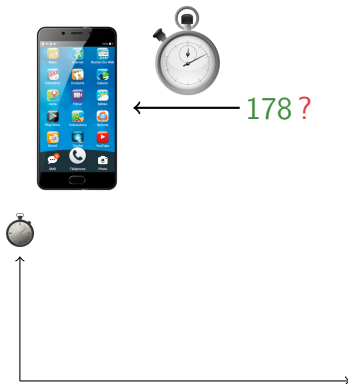
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \Rightarrow en moyenne 5000 essais.

Si mal implanté \Rightarrow au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



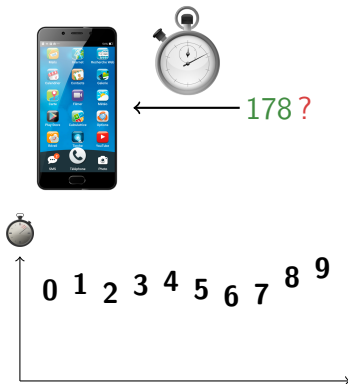
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \implies en moyenne 5000 essais.

Si mal implanté \implies au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



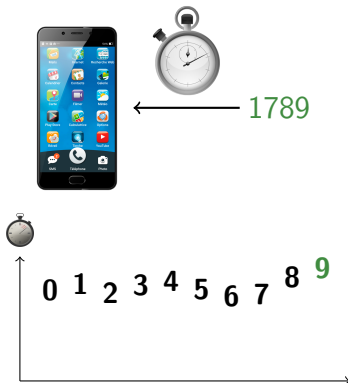
Temp d'exécution

(Mauvais) Exemple du code PIN

Code PIN de 4 chiffres \implies en moyenne 5000 essais.

Si mal implanté \implies au plus 37 essais.

```
bool testPIN(int code[4])
{
    for (int i=0 ; i<4 ; i++)
    {
        if (code[i]!=code_ref[i])
            return false;
    }
    return true;
}
```



Temp d'exécution

Exemple du code PIN corrigé

- ▶ Effectuer la boucle jusqu'au bout.
- ▶ Pas de branchement sur les valeurs sensibles.

```
bool testPIN(int code[4]) {  
    uint32_t diff = 0;  
    for ( int i = 0 ; i < 4 ; i++ ) {  
        diff |= code[i] ^ code_ref[i];  
    }  
    return (diff == 0);  
}
```

Temp d'exécution

Exemple du code PIN corrigé

- ▶ Effectuer la boucle jusqu'au bout.
- ▶ Pas de branchement sur les valeurs sensibles.

```
bool testPIN(int code[4]) {  
    uint32_t diff = 0;  
    for ( int i = 0 ; i < 4 ; i++ ) {  
        diff |= code[i] ^ code_ref[i];  
    }  
    return (diff == 0);  
}
```

Quid d'une comparaison de deux grands entiers ?

Temp d'exécution

Comparaison de deux grands entiers (1/4)

```
typedef enum
{ INF = -1, EQU = 0, SUP = 1 } num_order_t;
num_order_t compareBIGINT(int a[256], int b[256]) {
    bool a_inf_b = false;
    bool a_sup_b = false;
    for ( int i = 255 ; i >= 0 ; i -= 1 ) {
        if ( a_inf_b || a_sup_b )
            ; // do nothing
        else {
            if ( a[i] < b[i] )
                a_inf_b = true;
            if ( a[i] > b[i] )
                a_sup_b = true;
        }
    }
    if ( a_inf_b ) return INF;
    if ( a_sup_b ) return SUP;
    return EQU;
}
```

Temp d'exécution

Comparaison de deux grands entiers (2/4)

```
if ( a_inf_b || a_sup_b )  
    ; // do nothing  
else {  
    if ( a[i] < b[i] )  
        a_inf_b = true;  
    if ( a[i] > b[i] )  
        a_sup_b = true;  
}
```

```
if ( a_inf_b || a_sup_b )  
    ; // do nothing  
else {  
    a_inf_b = (a[i] < b[i]);  
    a_sup_b = (a[i] > b[i]);  
}
```

```
a_inf_b = (a_inf_b || (a[i] < b[i])) & !a_sup_b;  
a_sup_b = (a_sup_b || (a[i] > b[i])) & !a_inf_b;
```

Temp d'exécution

Comparaison de deux grands entiers (3/4)

```
typedef enum
{ INF = -1, EQU = 0, SUP = 1 } num_order_t;
num_order_t compareBIGINT(int a[256], int b[256]) {
    bool a_inf_b = false;
    bool a_sup_b = false;
    int res;
    for ( int i = 255 ; i >= 0 ; i -= 1 ) {
        a_inf_b = (a_inf_b || (a[i] < b[i])) & !a_sup_b;
        a_sup_b = (a_sup_b || (a[i] > b[i])) & !a_inf_b;
    }
    res = EQU * (!a_inf_b & !a_sup_b);
    res += INF * a_inf_b;
    res += SUP * a_sup_b;
    return (num_order_t) res;
}
```


Temp d'exécution

Comparaison de deux grands entiers (4/4)

```
...  
    movl    (%rax), %eax  
    cmpl    %eax, %edx  
    jge     .L4  
.L3:  
    movl    $1, %eax  
    jmp     .L5  
.L4:  
    movl    $0, %eax  
.L5:  
    andl    $1, %eax  
    cmpl    $0, -12(%rbp)  
    sete    %dl  
    andl    %edx, %eax  
...
```

Extrait du code compilé de la fonction de comparaison.

Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

RSA

Déchiffrement/signature : $c^d \bmod N$ avec d secret.

Square & Multiply :

Calcul de c^{37}

$$\bar{0}^2 \rightarrow t^2$$

$$\bar{1}^2 \rightarrow t^2 \times c$$

$$37 = \overline{100101}^2$$

RSA

Déchiffrement/signature : $c^d \bmod N$ avec d secret.

Square & Multiply :

Calcul de c^{37}

$$\bar{0}^2 \rightarrow t^2$$

$$\bar{1}^2 \rightarrow t^2 \times c$$

$$37 = \overline{100101}^2$$

$$t = c^{\bar{1}^2} = c = c^1$$

RSA

Déchiffrement/signature : $c^d \bmod N$ avec d secret.

Square & Multiply :

Calcul de c^{37}

$$\bar{0}^2 \rightarrow t^2$$

$$\bar{1}^2 \rightarrow t^2 \times c$$

$$37 = \overline{100101}^2$$

$$t = c^{\bar{1}^2} = c = c^1$$

$$t = c^{\bar{10}^2} = c^2 = c^2$$

RSA

Déchiffrement/signature : $c^d \bmod N$ avec d secret.

Square & Multiply :

Calcul de c^{37}

$$\overline{0}^2 \rightarrow t^2$$

$$\overline{1}^2 \rightarrow t^2 \times c$$

$$37 = \overline{100101}^2$$

$$t = c^{\overline{1}^2} = c = c^1$$

$$t = c^{\overline{10}^2} = c^2 = c^2$$

$$t = c^{\overline{100}^2} = (c^2)^2 = c^4$$

RSA

Déchiffrement/signature : $c^d \bmod N$ avec d secret.

Square & Multiply :

Calcul de c^{37}

$$\overline{0}^2 \rightarrow t^2$$

$$\overline{1}^2 \rightarrow t^2 \times c$$

$$37 = \overline{100101}^2$$

$$t = c^{\overline{1}^2} = c = c^1$$

$$t = c^{\overline{10}^2} = c^2 = c^2$$

$$t = c^{\overline{100}^2} = (c^2)^2 = c^4$$

$$t = c^{\overline{1001}^2} = (c^4)^2 \times c = c^9$$

RSA

Déchiffrement/signature : $c^d \bmod N$ avec d secret.

Square & Multiply :

Calcul de c^{37}

$$\overline{0}^2 \rightarrow t^2$$

$$\overline{1}^2 \rightarrow t^2 \times c$$

$$37 = \overline{100101}^2$$

$$t = c^{\overline{1}^2} = c = c^1$$

$$t = c^{\overline{10}^2} = c^2 = c^2$$

$$t = c^{\overline{100}^2} = (c^2)^2 = c^4$$

$$t = c^{\overline{1001}^2} = (c^4)^2 \times \mathbf{c} = c^9$$

$$t = c^{\overline{10010}^2} = (c^9)^2 = c^{18}$$

RSA

Déchiffrement/signature : $c^d \bmod N$ avec d secret.

Square & Multiply :

Calcul de c^{37}

$$\overline{0}^2 \rightarrow t^2$$

$$\overline{1}^2 \rightarrow t^2 \times c$$

$$37 = \overline{100101}^2$$

$$t = c^{\overline{1}^2} = c = c^1$$

$$t = c^{\overline{10}^2} = c^2 = c^2$$

$$t = c^{\overline{100}^2} = (c^2)^2 = c^4$$

$$t = c^{\overline{1001}^2} = (c^4)^2 \times \mathbf{c} = c^9$$

$$t = c^{\overline{10010}^2} = (c^9)^2 = c^{18}$$

$$t = c^{\overline{100101}^2} = (c^{18})^2 \times \mathbf{c} = c^{37}$$

Prédiction de branches

Sur RSA

```
BIGINT modExp(BIGINT m, bool d[]) {  
    BIGINT t = m;  
    for ( int i = 1 ; i >= 0 ; i — ) {  
        t = t * t;  
        if ( d[i] == 1 )  
            t = t * m;  
    }  
}
```

- ▶ attaquant faible : processus espion et analyse du temps d'exécution,
- ▶ attaquant fort : utilisation de `pfmon`.

OpenSSL

- ▶ Multiplication de point par Montgomery Ladder
- ▶ Pour optimiser, on passe les bits de poids fort à 0.

Le temps d'exécution dépend du nombre de bits de poids fort à 0 du nonce

...

En sélectionnant les signatures les plus rapides,

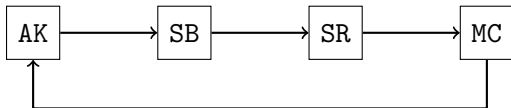
- ▶ on obtient un biais sur les nonces (qui sont petits),
- ▶ on attaque comme si les nonces étaient générés biaisés.

DÉMO avec algorithme LLL

Timing-attack de l'AES en tables

Implémentation tabulée de l'AES

Coeur de l'AES :



AK agit sur les octets (linéaire),

SB agit sur les octets,

SR agit sur les octets (linéaire),

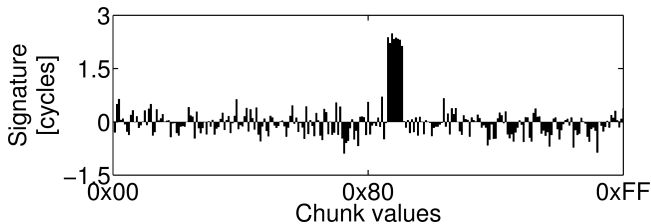
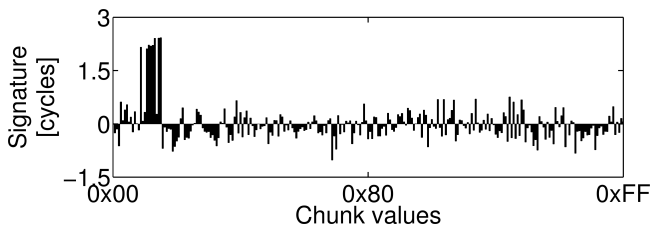
MC agit sur les mots de 32 bits (linéaire).

Tableaux T_0, T_1, T_2, T_3 indexés par un octet (état \oplus clef) et contenant des mots de 32 bits.

Timing-attack de l'AES en tables

Quelques résultats

Sur un Galaxy S2 : entraînement (haut) puis attaque (bas)



Attaques par oracles de format/padding

- Malléabilité du mode CBC

- Padding Oracle : IPSEC en mode ESP

- Padding Oracle sur RSA PKCS#1

Attaques exploitant du biais sur un aléa

- RC4

- Version Taïwanaise de la CCP

- Génération de nonce biaisé dans ECDSA

- Génération d'aléa uniforme mod n

Attaques exploitant le temps d'exécution

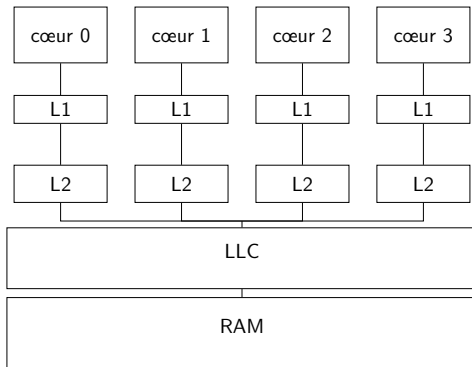
- Temps d'exécution d'une comparaison

- Temps d'exécution et cryptographie

- Micro-architecture et sécurité

Micro-architecture et sécurité

Cache : exemple d'Intel



Un *mapping* donnant l'emplacement du cache où sera stockée la donnée à partir de son adresse mémoire et potentiellement l'état du cache. Si l'emplacement n'est pas libre on procède à une *éviction*.

Attaque du type *Flush and Reload*

1. le processus espion vide le cache,
2. la main est rendue au processus cible,
3. le processus espion récupère la main,
4. le processus espion effectue des accès mémoire,
5. il exploite les temps de lecture pour en déduire les adresses accédées par le processus cible.

- ▶ L'attaque se base sur la propriété d'inclusion du cache.
- ▶ L'attaque est possible dès que de la mémoire est partagée (e.g. librairie).

Attaque du type *Prime and Probe*

1. le processus espion remplit le cache,
2. la main est rendue au processus cible,
3. le processus espion récupère la main,
4. le processus espion effectue des accès mémoire,
5. il exploite les temps de lecture pour en déduire les adresses accédées par le processus cible.

- ▶ L'attaque se base sur la propriété d'inclusion du cache.
- ▶ L'attaque est possible si l'on connaît les adresses du code cible.

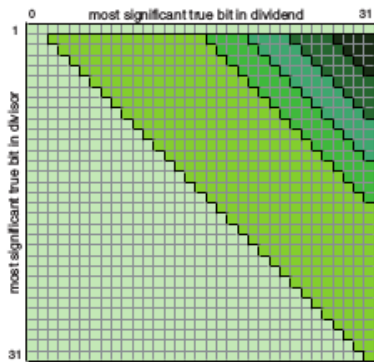
- ▶ Le cache contient des données mais aussi du code.
- ▶ On peut donc potentiellement savoir quelles instructions ont été exécutées.
 - ▶ Même certains algorithmes d'exponentiation réguliers pourraient être attaqués.
- ▶ On peut aussi détecter que deux processus dans le cloud sont co-localisés.
 - ▶ Cela viole la propriété de cloisonnement.
- ▶ On peut même monter un tunnel SSH entre eux (*canal caché*).

Hot topic !

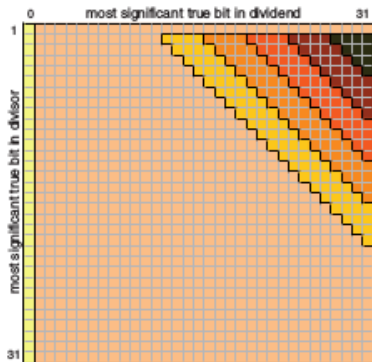
Grâce à un bug lié à une situation de compétition (*race condition*) et à un mécanisme de cache, on peut même lire le contenu d'une mémoire normalement non accessible (CVE-2017-5754).

Micro-architecture et sécurité

Instruction à temps d'exécution variable



(a) Core 2 latency classes



(b) Xeon latency classes

FIGURE: Temps d'exécution de la division d'Intel.

Messages

- ▶ L'attaquant va chercher à exploiter toutes les informations disponibles.
- ▶ Un rien peut suffire à faire tomber la sécurité.



Bonnes pratiques

- ▶ Limiter les sources d'information auxiliaires données à l'utilisateur (et donc à l'attaquant).
- ▶ Ne pas minimiser les faiblesses cryptographiques des primitives.
- ▶ Éviter de faire reposer la sécurité sur la supposée non-exploitable d'une fuite.