Sécurité du logiciel, quatrième cours

Tempus fugit : attaques par observation du temps ou du cache

Xavier Leroy

2022-03-31

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Le temps d’exécution comme canal d’information

Expérience utilisateur avec une ancienne version d'Unix

```
login: dmr
password: *****
```

(deux secondes plus tard…)

```
Login incorrect
```

Expérience utilisateur avec une ancienne version d'Unix

```
login: dmr
password: *****
```

(deux secondes plus tard…)

```
Login incorrect
```

Deuxième tentative :

```
login: toto
password: *****
```

(instantanément)

```
Login incorrect
```

Pourquoi cette différence de temps de réponse ?

```
int check_login(char * username, char * password)
{
    struct passwd * userinfo = getpwnam(username);
    if (userinfo == NULL) return 0; // pas d'utilisateur de ce  nom
    char * hash = crypt(password); // prend 2 secondes
    return (strcmp(hash, userinfo->pw_password) == 0);
}
```

La fonction termine plus rapidement s'il n'y a pas de compte utilisateur appelé `username` que s'il y en a un.

$\Longrightarrow$ Permet à un attaquant de vérifier facilement si un compte donné existe.

Vérification de code secret (PIN)

```
for (int i = 0; i < N; i++) {
    if (input[i] != pin[i]) return false;
}
return true;
```

Le temps d'exécution de cette boucle est proportionnel au nombre de chiffres au début de `input` qui sont corrects.

$\Longrightarrow$ Permet à un attaquant de trouver un code secret à N chiffres en temps $10\,N$ au lieu de 10^N.

Vérification de code secret

Une autre implémentation, où la boucle fait toujours N itérations :

```
valid = true;
for (int i = 0; i < N; i++) {
    if (input[i] != pin[i]) valid = false;
}
return valid;
```

Maintenant, le temps d'exécution est $a + bn$, où n est le nombre de chiffres incorrects (= nombre d'affectations `valid = false`).

$\Longrightarrow$ Une attaque efficace est encore possible.

Vérification de code secret

Essayons de symétriser en comptant le nombre de chiffres corrects et le nombre de chiffres incorrects.

```
valid = 0; invalid = 0;
for (int i = 0; i < N; i++) {
    if (input[i] != pin[i]) ++invalid; else ++valid;
}
return (invalid == 0);
```

La prédiction de branchements du processeur peut encore introduire des variations du temps d'exécution, mais de plus en plus difficiles à exploiter.

Vérification de code secret en temps constant

La bonne manière d'écrire ce code est de n'utiliser que des opérations ≪en temps constant≫, c.à.d. dont le temps d'exécution ne dépend pas de la valeur des arguments :

```
d = 0;
for (int i = 0; i < N; i++) {
    d = d | (input[i] ^ pin[i]);
}
return (d == 0);
```

(La variable `d` accumule les bits qui diffèrent entre `input` et `password`; elle reste à 0 si et seulement s'il n'y a aucune différence.)

Chiffrement et signature RSA

À base d'exponentiation modulaire :

$$M \xrightarrow{\text{chiffrement}} C = M^e \bmod N \xrightarrow{\text{déchiffrement}} C^d \bmod N$$

$$M \xrightarrow{\text{signature}} S = M^d \bmod N \xrightarrow{\text{vérification}} S^e \bmod N$$

Un module $N = pq$ produit de deux nombres premiers p, q.

Un exposant secret d et un exposant public e tels que $de = 1 \pmod{(p-1)(q-1)}$.

La clé publique est (N, e).

La clé secrète est d ou éventuellement (p, q, d).

Calcul de l'exponentiation modulaire

Algorithme d'exponentiation rapide dit ≪du paysan russe≫ :

Décomposer d en bits $d_n, \ldots, d_0$ $\qquad (d = \sum_{i=0}^{n} d_i 2^i)$

$C := 1$; $z := M$;

`for` $i = 0$ `to` n `do`

  `if` d_i `then` $C := C \cdot z \bmod N$

  $z := z^2 \bmod N$

`done`

z prend successivement les valeurs $M, M^2, M^4, \ldots, M^{2^n}$.

À la fin on a $C = \prod\{M^{2^i} \mid d_i = 1\} = M^{\sum\{2^i | d_i = 1\}} = M^d$

Temps de calcul de l'exponentation modulaire

```
for i = 0 to n do
    if d_i then C := C · z mod N
    z := z^2 mod N
done
```

Le temps d'exécution de la boucle dépend évidemment des d_i :
on fait $w + n + 1$ multiplications modulaires,
où w est le poids de Hamming (nombre de bits à 1) du secret d.

Cependant, connaître w n'aide pas à deviner d.

De plus, on peut facilement supprimer cette dépendance sur w :

```
if d_i then C := C · z mod N else tmp := C · z mod N
```

Temps de calcul de l'exponentiation modulaire

```
for i = 0 to n do
    if d_i then C := C · z mod N else tmp := C · z mod N
    z := z^2 mod N
done
```

Le temps de calcul de $C \cdot z \bmod N$ dépend significativement de la valeur de C, d'autant plus si des algorithmes ≪rusés≫ de multiplication modulaire sont utilisés.

Cela suffit à monter des attaques par observation du temps…

L'attaque de P. Kocher (1996)

Prendre k messages aléatoires $M_1, \ldots, M_k$.

Les faire signer : $S_i = M_i^d \bmod N$, et mesurer le temps T_i pris.

Deviner successivement les bits de d :

- $d_0 = 1$ toujours.
- Supposons $d_1 = 1$. Alors le calcul de S_i commencerait par calculer $M_i \cdot M_i^2 \bmod N$.
  - Mesurer les temps t_i pour calculer $M_i \cdot M_i^2 \bmod N$.
  - Si les t_i sont corrélés aux T_i : on a bien $d_1 = 1$.
  - S'il n'y a pas de corrélation : on a $d_1 = 0$.
- Recommencer avec les bits suivants.

L'attaque de D. Brumley et D. Boneh sur OpenSSL (2003)

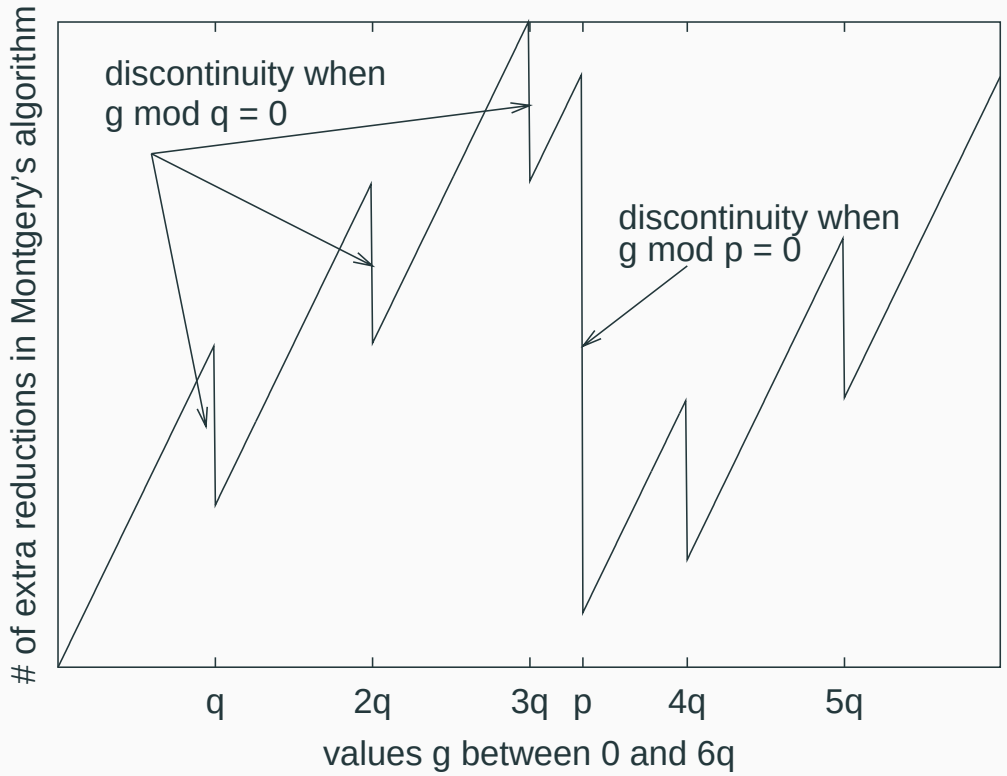
Une implémentation plus efficace de RSA :

- Utilisation du théorème des restes chinois : on calcule $M^d \bmod p$ et $M^d \bmod q$, et on combine les résultats pour obtenir $M^d \bmod N$ (avec $N = pq$).
- Utilisation de la représentation de Montgomery pour accélérer les multiplications modulaires $C \cdot z \bmod q$.
- Plusieurs algorithmes de multiplication, selon les tailles des arguments.

Chacun de ces points contribue à «fuiter» davantage d'information dans le temps d'exécution…

Temps d'exécution de la multiplication de Montgomery

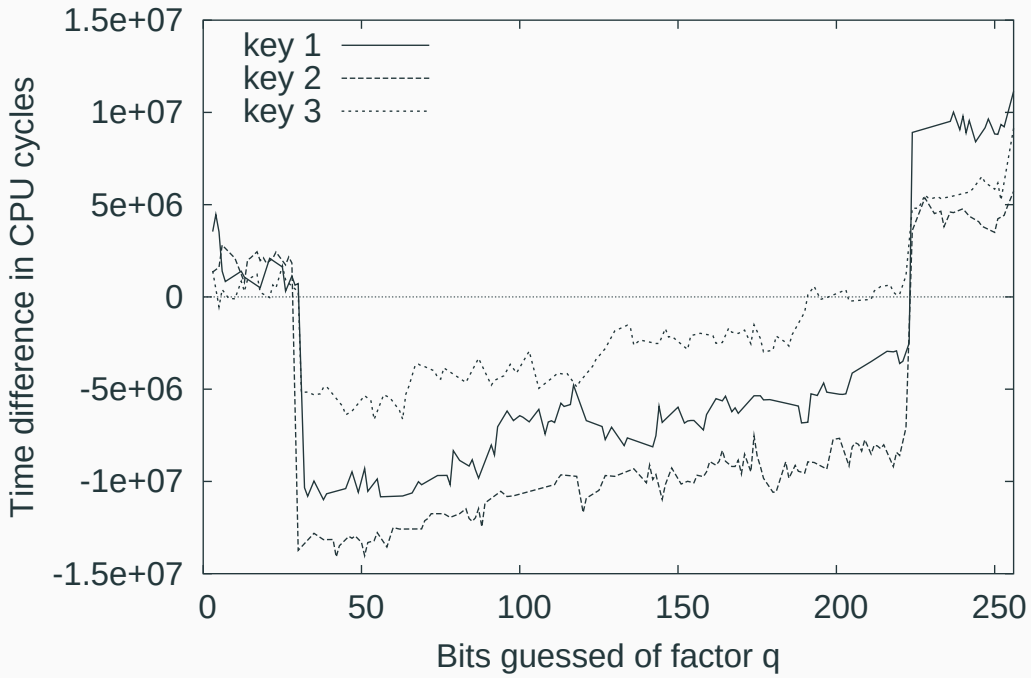
L'algorithme de Montgomery effectue des étapes de réduction supplémentaires lorsque le produit g s'approche du module p, q.



(Brumley & Boneh, 2003)

L’attaque de D. Brumley et D. Boneh

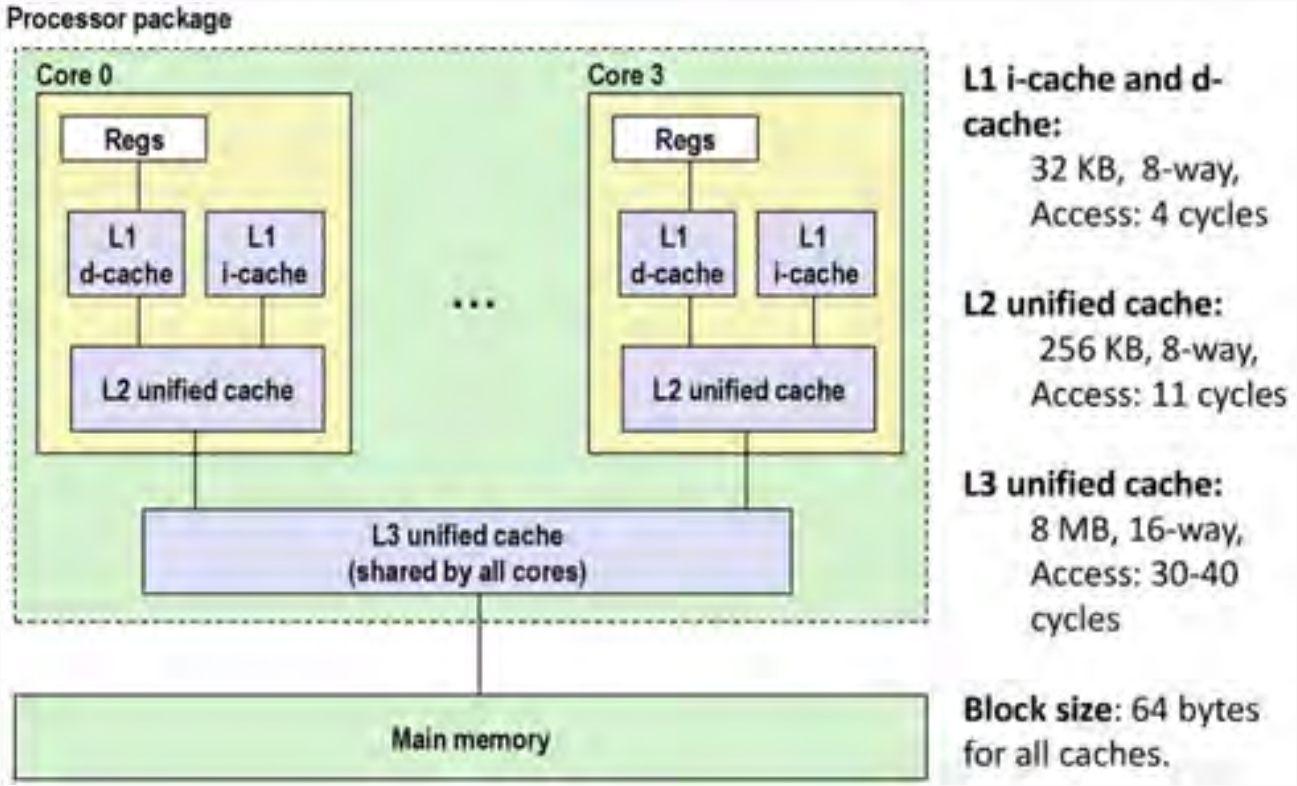
Une recherche dichotomique qui identifie les bits de poids forts du facteur q. Une fois la moitié des bits identifiés, l’algorithme de Coppersmith trouve toute la clé secrète.



L’attaque peut être menée à travers une connexion réseau !

Le cache mémoire comme canal d'information

Le cache mémoire



Accélère les accès à une case mémoire récemment accédée (localité temporelle), ou proche d'une case mémoire récemment accédée (localité spatiale).

Attaques par observation du cache

Le temps que prend une lecture mémoire varie beaucoup selon qu'une donnée à une adresse proche a été accédée récemment.

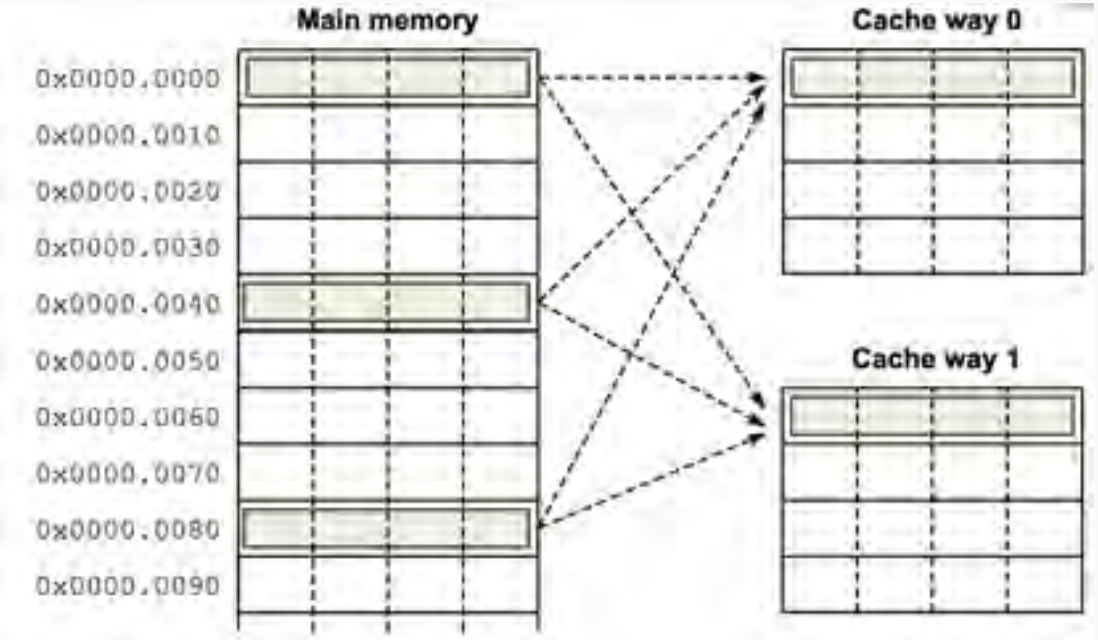
Schéma général d'une attaque :

1. Vider le cache (L1 ou plus) (instruction `clflush`, etc)
2. Exécuter du code protégé qui manipule une donnée secrète
3. Mesurer les temps d'accès à différentes cases mémoire
4. En déduire les cases mémoires accédées par le code protégé
5. En déduire des informations sur la donnée secrète.

(Au lieu de vider le cache en 1-, on peut aussi le préremplir avec des adresses en conflit avec les adresses à observer.)

(2- et 3- peuvent se faire en parallèle.)

Attaques par observation du cache



Remarque : il n'est pas nécessaire de pouvoir lire et écrire dans la zone mémoire que l'on veut observer. Toute zone mémoire qui partage les mêmes entrées de cache fait l'affaire.

Exemple : normalisation d'une chaîne de caractères

Passer les lettres en majuscules et normaliser les caractères non imprimables.

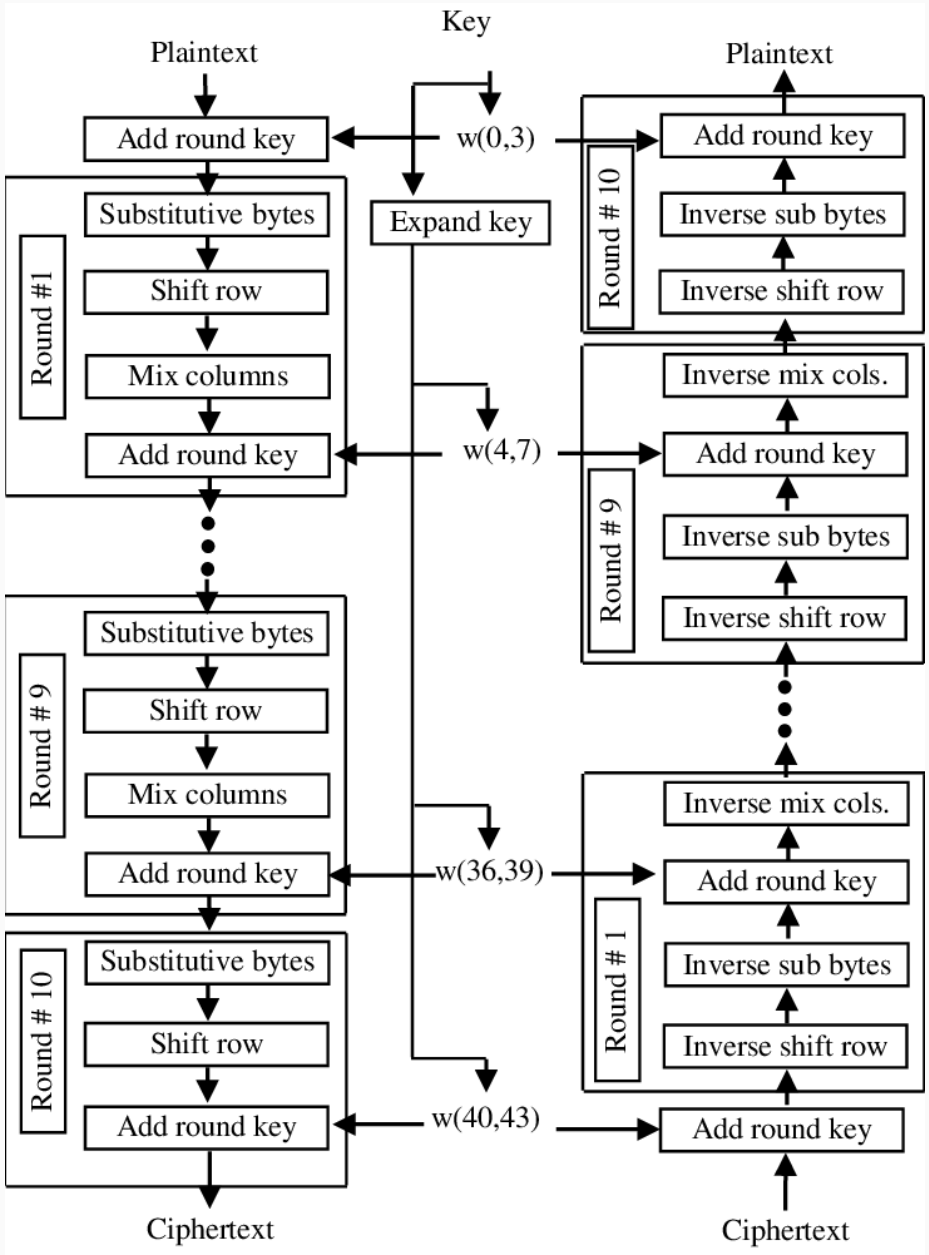
```
void normalize(unsigned char * s, size_t len)
{
    static const unsigned char tbl[256] = "\
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\
 !\"#$%&'()*+,-./0123456789:;<=>?\
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_\
`ABCDEFGHIJKLMNOPQRSTUVWXYZ{|}~";

    for (size_t i = 0; i < len; i++) s[i] = tbl[s[i]];
}
```

Exemple d'attaque par observation du cache

`get_hashed_password` : une fonction protégée qui lit un mot de passe au clavier, le normalise avec `normalize`, et le hache.

1. Vider le cache.
2. Appeler `get_hash_password`.
3. Mesurer le temps pris par `normalize` sur les entrées `"a"`, `"b"`, `"c"`, etc.
4. En déduire les cases du tableau `tbl` accédées par `normalize` appelée depuis `get_hash_password`.
   (On suppose que la taille des lignes de cache est 1.)
5. En déduire quelles lettres `a`, `b`, `c`, etc, figurent dans le mot de passe.

Le chiffrement symétrique AES-128



(A. G. Wadday et al, 2018)

Le chiffrement symétrique AES-128

Dans une implémentation d'AES en logiciel, les étapes ≪subst/shift/mix≫ sont généralement tabulées.

T_0, T_1, T_2, T_3 : tables de 256 constantes de 32 bits.

$x_0, \ldots, x_{15}$: les 16 octets de l'état courant.

$$
\begin{aligned}
(x'_0, x'_1, x'_2, x'_3) &= T_0[x_0] \oplus T_1[x_5] \oplus T_2[x_{10}] \oplus T_3[x_{15}] \oplus K_0 \\
(x'_4, x'_5, x'_6, x'_7) &= T_0[x_4] \oplus T_1[x_9] \oplus T_2[x_{14}] \oplus T_3[x_3] \oplus K_1 \\
(x'_8, x'_9, x'_{10}, x'_{11}) &= T_0[x_8] \oplus T_1[x_{13}] \oplus T_2[x_2] \oplus T_3[x_7] \oplus K_2 \\
(x'_{12}, x'_{13}, x'_{14}, x'_{15}) &= T_0[x_{12}] \oplus T_1[x_1] \oplus T_2[x_6] \oplus T_3[x_{11}] \oplus K_3
\end{aligned}
$$

Attaquer AES via le cache

(Osvik, Shamir, Tromer, *Cache attacks and countermeasures : the case of AES*, 2005. Ashokummar, Giri, Menezes, *Highly efficient algorithms for AES key retrieval in cache access attacks*, 2016.)

$$
\begin{aligned}
(x'_0, x'_1, x'_2, x'_3) &= T_0[x_0] \oplus T_1[x_5] \oplus T_2[x_{10}] \oplus T_3[x_{15}] \oplus K_0 \\
(x'_4, x'_5, x'_6, x'_7) &= T_0[x_4] \oplus T_1[x_9] \oplus T_2[x_{14}] \oplus T_3[x_3] \oplus K_1 \\
(x'_8, x'_9, x'_{10}, x'_{11}) &= T_0[x_8] \oplus T_1[x_{13}] \oplus T_2[x_2] \oplus T_3[x_7] \oplus K_2 \\
(x'_{12}, x'_{13}, x'_{14}, x'_{15}) &= T_0[x_{12}] \oplus T_1[x_1] \oplus T_2[x_6] \oplus T_3[x_{11}] \oplus K_3
\end{aligned}
$$

Avec des lignes de cache de 4 mots de 32 bits, chaque accès $T_i[x_j]$ fait ≪fuiter≫ les 4 bits de poids fort de x_j.

Au premier round, $x =$ texte choisi $\oplus$ clé, donc on retrouve assez facilement les 4 bits de poids fort de chaque octet de la clé.

Attaquer AES via le cache

(Osvik, Shamir, Tromer, *Cache attacks and countermeasures : the case of AES*, 2005. Ashokummar, Giri, Menezes, *Highly efficient algorithms for AES key retrieval in cache access attacks*, 2016.)

$$
\begin{aligned}
(x'_0, x'_1, x'_2, x'_3) &= T_0[x_0] \oplus T_1[x_5] \oplus T_2[x_{10}] \oplus T_3[x_{15}] \oplus K_0 \\
(x'_4, x'_5, x'_6, x'_7) &= T_0[x_4] \oplus T_1[x_9] \oplus T_2[x_{14}] \oplus T_3[x_3] \oplus K_1 \\
(x'_8, x'_9, x'_{10}, x'_{11}) &= T_0[x_8] \oplus T_1[x_{13}] \oplus T_2[x_2] \oplus T_3[x_7] \oplus K_2 \\
(x'_{12}, x'_{13}, x'_{14}, x'_{15}) &= T_0[x_{12}] \oplus T_1[x_1] \oplus T_2[x_6] \oplus T_3[x_{11}] \oplus K_3
\end{aligned}
$$

Avec des lignes de cache de 4 mots de 32 bits, chaque accès $T_i[x_j]$ fait ≪fuiter≫ les 4 bits de poids fort de x_j.

En analysant finement le 2^e round, on retrouve toute la clé avec un nombre raisonnable de chiffrements (10 à 1000).

Protections contre les attaques temporelles

Protections contre les attaques temporelles

Comment éviter la fuite d'information via le temps d'exécution ?

Quelques approches :

- Programmer ≪en temps constant≫ (*constant-time*)
  (plus précisément : en temps indépendant des secrets).
- Empêcher la mesure précise du temps.
- ≪Quantiser≫ le temps d'exécution ou de communication.
- Masquer les secrets avec de l'aléa. (*blinding*)

Mesure précise du temps d'exécution

Bien des des attaques par observation du temps ne peuvent pas se faire à distance : l'attaquant doit ≪tourner≫ sur la même machine que le code attaqué.

Pour mesurer précisément le temps écoulé, l'attaquant a besoin

- d'un accès à une horloge matérielle de haute résolution (p.ex. le registre Time Stamp Counter des processeurs x86);
- ou d'une exécution en parallèle de plusieurs *threads*.

$$\begin{array}{l} \texttt{while true do} \\ \quad \texttt{time} := \texttt{time} + 1 \\ \texttt{done} \end{array} \quad \Big\| \quad \begin{array}{l} T_0 := \texttt{time}; \\ \text{calcul à mesurer}; \\ T_1 := \texttt{time} \end{array}$$

Empêcher la mesure précise du temps d'exécution

Le système ou l'environnement d'exécution peuvent :

- Interdire l'accès à l'horloge matérielle de haute résolution (p.ex. bloquer l'instruction `rdtsc` des processeurs x86).
- Forcer les *threads* de l'attaquant à s'exécuter sur le même cœur de processeur que l'attaqué, par entrelacement.
- Ordonnancer les *threads* de manière indépendante du temps d'exécution.

Ordonnancement par comptage d'instructions

(Stefan et al, *Eliminating cache-based timing attacks with instruction-based scheduling*, 2013.)

$$\texttt{fillArray}(L);$$

$$\begin{array}{l|l|l}
\texttt{if } \textit{secret} \texttt{ then} & \texttt{for } i = 1 \texttt{ to } n & \texttt{for } i = 1 \texttt{ to } n + m \\
\quad \texttt{fillArray}(H) & \texttt{do skip done;} & \texttt{do skip done;} \\
\texttt{else} & \texttt{readArray}(L); & x := 0 \\
\quad \texttt{skip} & x := 1 &
\end{array}$$

Avec un ordonnancement basé sur des tranches de temps (préemption au bout d'un temps T fixé), on termine avec $x = 0$ ou $x = 1$ suivant la durée d'exécution de $\texttt{readArray}(L)$, qui dépend de l'état du cache.

Ordonnancement par comptage d'instructions

(Stefan et al, *Eliminating cache-based timing attacks with instruction-based scheduling*, 2013.)

```
                                fillArray(L);
if secret then          ||  for i = 1 to n      ||  for i = 1 to n + m
  fillArray(H)          ||  do skip done;       ||  do skip done;
else                    ||  readArray(L);       ||  x := 0
  skip                  ||  x := 1              ||
```

Avec un ordonnancement basé sur des comptes d'instructions (préemption après N instructions exécutées), la valeur finale de x est indépendante de l'état du cache.

Quantisation du temps

Ajouter une attente à la fin du calcul pour garantir la durée D_{max}.

$T_0 := \texttt{now};$
$\texttt{for}\ i = 0\ \texttt{to}\ n\ \texttt{do}$
$\quad \texttt{if}\ d_i\ \texttt{then}\ C := C \cdot z \bmod N\ \texttt{else}\ tmp := C \cdot z \bmod N$
$\quad z := z^2 \bmod N$
$\texttt{done}$
$D = \texttt{now} - T_0;$
$\texttt{sleep}(D_{max} - D)$

Plus de ≪fuites≫ temporelles !

… au prix d'un ralentissement de tous les calculs.

D_{max} peut être difficile à déterminer *a priori*.

Quantisation du temps

Variante : ajuster le temps à un multiple entier de Δ.
(P.ex. $\Delta = 10^7$ cycles dans le cas de l'attaque de Brumley-Boney.)

$T_0 := \texttt{now};$

...

$D = \texttt{now} - T_0;$

$\texttt{sleep}(\texttt{ceil}(D/\Delta) \times \Delta)$

Quantisation du temps

(Askarov, Zhang, Myers, *Predictive black-box mitigation of timing channels*, 2010.)

Variante : ajuster D_{max} au vol, suivant une loi exponentielle.

$T_0 := \texttt{now};$

$\ldots$

$D = \texttt{now} - T_0;$

$\texttt{if } D > D_{max} \texttt{ then } D_{max} := D_{max} \times (1+\varepsilon)$

$\texttt{else sleep}(D_{max} - D)$

L'attaquant obtient un bit d'information à chaque fois que $D > D_{max}$. Cela arrive au plus une fois par tranche de temps de durée $(1+\varepsilon)^k$. D'où une fuite d'information en $O(\log^2 t)$.

D'autres lois plus subtiles sont possibles, voir Askarov *et al.*

Masquage (*blinding*)

Injecter de l'aléa dans le calcul afin de décorréler le temps d'exécution de la valeur du secret.

Exemple (artificiel) :
vérification de code PIN avec permutation aléatoire.

```
// tirer au hasard une permutation S de {0, ..., n − 1}
for (int i = 0; i < N; i++) {
    if (input[S[i]] != pin[S[i]]) return false;
}
return true;
```

Le temps d'une exécution donne seulement une borne inférieure sur le nombre de chiffres communs à `input` et `pin`.

RSA avec masquage du message

Si M est le message à signer, on peut le masquer à l'aide d'un nombre aléatoire R avant l'exponentiation modulaire.

$$C \overset{def}{=} (R^e \cdot M)^d = (R^e)^d \cdot M^d = R^{ed} \cdot M^d = R \cdot M^d \pmod{N}$$

car $ed \bmod \varphi(N) = 1$ et $R^{\varphi(N)} = 1 \pmod{N}$ (Euler).

Ensuite, on peut démasquer pour obtenir le bon résultat :

$$S \overset{def}{=} R^{-1} \cdot C \pmod{N}$$

Le temps de calcul de $(R^e \cdot M)^d$ ne révèle rien à l'attaquant, puisque celui-ci choisit M mais pas R.

Programmation «en temps constant»

La programmation ≪en temps constant≫

Une discipline de programmation pour produire des programmes dont le temps d'exécution ne dépend pas de données secrètes.

S'appuie sur une classification des opérations de base du langage / des instructions du processeur :

- Opérations en temps constant : même temps d'exécution quelles que soient les valeurs des arguments de l'opération et l'état du processeur.
- Opérations en temps variable suivant les valeurs des arguments ou l'état du processeur (caches mémoire, prédicteur de branchements, etc).

Une classification typique

	Temps constant	Temps non constant
Arithmétique entière ([1])	`+ - * &	^` décalages comparaisons	division, modulo
Lecture/écriture en mémoire ([2])		`x[i]` `*p`
Branchements conditionnels		`if` `while` `&&` `||`

(1) Certains processeurs font la multiplication en temps variable.

(2) Pour `x[i] = v` et `*p = v`, le temps d'exécution dépend de `x`, `i`, `p` (l'adresse accédée) mais pas de `v` (la valeur écrite).

Le critère ≪temps constant≫

Une propriété de flux d'information :

> Une donnée de niveau H (secret) ne doit pas être utilisée comme argument d'une opération en temps non constant.

Exemples :

✔ $\texttt{z}^H \texttt{ := x}^H \texttt{ + y}^H$ ✘ $\texttt{z}^H \texttt{ := x}^H \texttt{ / y}^H$

✘ $\texttt{if x}^H \texttt{ < y}^H \texttt{ then z}^H \texttt{ := 1}$

✘ $\texttt{x}^H \texttt{ := t}^L\texttt{[i}^H\texttt{]}$

Règles de typage pour le temps constant

À la manière des systèmes de types pour les flux d'informations du 2e cours

$$\frac{\vdash a_1 : \ell \qquad a_2 : \ell}{\vdash a_1 + a_2 : \ell} \qquad \frac{\vdash a_1 : L \qquad a_2 : L}{\vdash a_1 / a_2 : \ell}$$

$$\frac{\vdash b : L \qquad \vdash c_1 : * \qquad \vdash c_2 : *}{\vdash \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 : *} \qquad \frac{\vdash b : L \qquad \vdash c : *}{\vdash \texttt{while } b \texttt{ do } c}$$

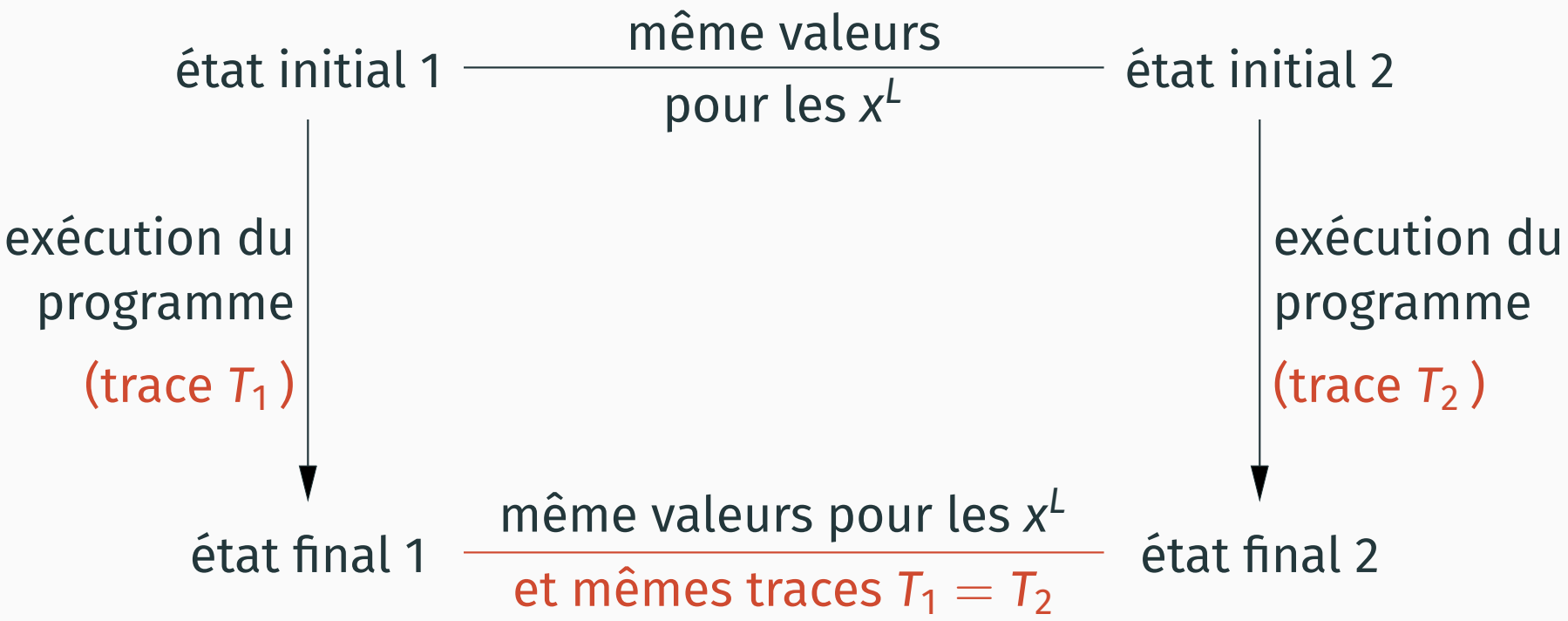
Note : les flux indirects ($\texttt{if } b^H \texttt{ then } x^L := 1 \texttt{ else } x^L := 0$) sont automatiquement exclus ; plus besoin de tracer le niveau du *pc*.

Une sémantique pour les fuites temporelles

On peut matérialiser les fuites d'information provenant des opérations en temps non constant par une transformation de programmes :

$$
\begin{aligned}
[\![z := x + y]\!] &= z := x + y \\
[\![z := x/y]\!] &= \mathtt{out}(x); \mathtt{out}(y); z := x/y \\
[\![\mathtt{if}\ x < y\ \ \mathtt{then}\ c_1\ \mathtt{else}\ c_2]\!] &= \mathtt{out}(x); \mathtt{out}(y); \\
&\quad\ \mathtt{if}\ x < y\ \mathtt{then}\ [\![c_1]\!]\ \mathtt{else}\ [\![c_2]\!] \\
[\![\mathtt{while}\ x < y\ \mathtt{do}\ c\ \mathtt{done}]\!] &= \mathtt{out}(x); \mathtt{out}(y); \\
&\quad\ \mathtt{while}\ x < y\ \mathtt{do} \\
&\quad\ \ [\![c]\!]; \mathtt{out}(x); \mathtt{out}(y) \\
&\quad\ \mathtt{done}
\end{aligned}
$$

Non-interférence en présence de fuites temporelles



Programmer en style ≪temps constant≫

Sur les données secrètes, pas de conditionnelles ni d'indexation de tableaux, juste des opérations arithmétiques et bit-à-bit
≈ des circuits combinatoires.

Exemple (rappel) :

```
d = 0;
for (int i = 0; i < N; i++) {
    if (input[i] != pin[i]) d = 1; ✘
}
return (d == 0);
```

Programmer en style ≪temps constant≫

Sur les données secrètes, pas de conditionnelles ni d'indexation de tableaux, juste des opérations arithmétiques et bit-à-bit
≈ des circuits combinatoires.

Exemple (rappel) :

```
d = 0;
for (int i = 0; i < N; i++) {
    d = d | (input[i] ^ pin[i]); ✔
}
return (d == 0);
```

Éviter les tableaux

Les entiers N bits peuvent remplacer des tableaux de N booléens.

Exemple : une *S-box* DES = une fonction 6 bits $\rightarrow$ 4 bits.

Implémentation tabulée classique :

```
int tbl[64] = { /* 64 entiers de 4 bits */ };
int sbox(int x) { return tbl[x]; }
```

Implémentation tabulée avec 4 entiers 64 bits :

```
uint64_t tbl0 = ..., tbl1 = ..., tbl2 = ..., tbl3 = ...;
int sbox(int x) {
    return (tbl0 >> x & 1) << 0 | (tbl1 >> x & 1) << 1 |
           (tbl2 >> x & 1) << 2 | (tbl3 >> x & 1) << 3;
}
```

(En temps constant… mais beaucoup plus lent !)

Transformer les conditionnelles en sélections (*IF-conversion*)

Cas de base :

$$\texttt{if } b \texttt{ then } x := a_1 \texttt{ else } x := a_2 \implies x := \texttt{sel}(b, a_1, a_2)$$

$$\texttt{if } b \texttt{ then } x := a_1 \implies x := \texttt{sel}(b, a_1, x)$$

$\texttt{sel}(b, a_1, a_2)$ est un opérateur qui

- évalue b, a_1 et a_2 ;
- renvoie la valeur de a_1 si b est vrai ;
- renvoie la valeur de a_2 si b est faux

en temps indépendant de la valeur de b (≪en temps constant≫).

Transformer les conditionnelles en sélections (*IF-conversion*)

Plus généralement :

- Exécuter les deux branches `then` et `else` avec renommage des variables affectées ;
- Sélectionner les valeurs correctes avec l'opérateur `sel`.

Exemple :

$$\texttt{if } b \texttt{ then } (x := a_1; y := a_2) \texttt{ else } (y := a_3; z := a_4)$$

$$\begin{aligned} \Longrightarrow \quad & x_1 := a_1; \;\; y_1 := a_2[x \leftarrow x_1]; \\ & y_2 := a_3; \;\; z_2 := a_4[y \leftarrow y_2]; \\ & x := \texttt{sel}(b, x_1, x); \;\; y := \texttt{sel}(b, y_1, y_2); \;\; z := \texttt{sel}(b, z, z_2) \end{aligned}$$

Limites de la *IF-conversion*

Cette transformation ne s'applique que si les branches `then` et `else`

- terminent toujours ;
- ne font jamais d'erreurs à l'exécution ;
- n'ont pas d'effets observables par le reste du programme.

Exemple problématique :

$$\texttt{if } y \neq 0 \texttt{ then } z := x/y \texttt{ else abort()}$$
$$\not\Longrightarrow \quad z_1 := x/y;\ \texttt{abort()};\ z := \texttt{sel}(y \neq 0, z_1, z)$$

Implémenter l'opérateur de sélection

Avec des instructions spéciales du processeur
(*conditional move*, instructions à prédicats, etc).

De manière portable, quand b, a_1 et a_2 sont de type `bool` :

$$\texttt{sel}(b, a_1, a_2) = b \wedge a_1 \vee \neg b \wedge a_2$$

De manière portable, quand a_1 et a_2 sont des entiers et $b = 0$ ou 1 :

$$\texttt{sel}(b, a_1, a_2) = b \times a_1 + (1 - b) \times a_2$$
$$\texttt{sel}(b, a_1, a_2) = a_2 + b \times (a_1 - a_2)$$
$$\texttt{sel}(b, a_1, a_2) = (-b) \wedge a_1 \vee (b - 1) \wedge a_2$$

(Si $b = 0$, on a $b - 1 = 11 \ldots 11$ et $-b = 00 \ldots 00$.
Si $b = 1$, on a $b - 1 = 00 \ldots 00$ et $-b = 11 \ldots 11$.)

Résister aux optimisations du compilateur

Un compilateur optimisant peut implémenter la ≪IF-conversion≫, transformant certaines conditionnelle en code temps constant :

$$\texttt{if}\ \ b\ \ \texttt{then}\ \ x := a_1\ \ \texttt{else}\ \ x := a_2\ \ \rightarrow\ \ x := \texttt{sel}(b, a_1, a_2)$$

Mais il peut aussi introduire des branchements conditionnels pour calculer une expression arithmétique ou booléenne, comme nos implémentations de `sel` !

$$x := b \times a_1 + (1 - b) \times a_2 \rightarrow \texttt{if}\ \ b\ \ \texttt{then}\ \ x := a_1\ \ \texttt{else}\ \ x := a_2$$

Résister aux optimisations du compilateur

(Simon, Chisnall, Anderson, *What you get is what you C : controlling side effects in mainstream C compilers*, 2018).

Expérience : 4 implémentations de `sel` en C portable, compilées pour x86-32 par différentes versions de Clang.

		VERSION_1		VERSION_2		VERSION_3		VERSION_4	
		inlined	library	inlined	library	inlined	library	inlined	library
Clang 3.0	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✓	✓	✗
	-O2	✓	✓	✓	✗	✗	✓	✓	✗
	-O3	✓	✓	✓	✗	✗	✓	✓	✗
Clang 3.3	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✗	✓	✗
	-O2	✓	✓	✗	✗	✗	✗	✗	✗
	-O3	✓	✓	✗	✗	✗	✗	✗	✗
Clang 3.9	-O0	✓	✓	✓	✓	✓	✓	✓	✓
	-O1	✓	✓	✓	✓	✓	✗	✓	✗
	-O2	✓	✓	✗	✗	✗	✗	✗	✗
	-O3	✓	✓	✗	✗	✗	✗	✗	✗

✔ = production de code ≪temps constant≫.

✘ = production d'un branchement conditionnel.

Attaques sur l'exécution spéculative

Exécution spéculative dans les processeurs

Exemple : la prédiction de branchements.

```
        load x0, [x1]
        branch if x0 = 0 to L1
        mul x3, x2, x3
        add x4, x3, x4
        branch to L2
L1:     ...
```

La lecture en mémoire prend beaucoup de temps.

Exécution spéculative dans les processeurs

Exemple : la prédiction de branchements.

```
        load x0, [x1]
        branch if x0 = 0 to L1
        mul x3, x2, x3
        add x4, x3, x4
        branch to L2
L1:     ...
```

Le processeur prédit (sur la base des précédentes exécutions) que `x0` sera non nul et que le branchement ne sera pas pris.

Exécution spéculative dans les processeurs

Exemple : la prédiction de branchements.

```
        load x0, [x1]
        branch if x0 = 0 to L1
        mul x3, x2, x3
        add x4, x3, x4
        branch to L2
L1:     ...
```

Le processeur exécute les instructions qui suivent de manière spéculative, c'est-à-dire réversible si besoin.

(P.ex. les valeurs initiales de `x3` et `x4` sont gardées quelque part.)

Exécution spéculative dans les processeurs

Exemple : la prédiction de branchements.

```
        load x0, [x1]
        branch if x0 = 0 to L1
        mul x3, x2, x3
        add x4, x3, x4
        branch to L2
L1:     ...
```

Quand le `load` termine, le branchement conditionnel est résolu.

Si `x0` vaut 0, la prédiction était fausse, et le processeur revient en arrière sur l'exécution spéculative : les actions des instructions spéculées sont ignorées (p.ex. les registres `x3`, `x4` reprennent leurs valeurs initiales), et l'exécution reprend au point `L1`.

Exécution spéculative dans les processeurs

Exemple : la prédiction de branchements.

```
        load x0, [x1]
        branch if x0 = 0 to L1
        mul x3, x2, x3
        add x4, x3, x4
        branch to L2
L1:     ...
```

Si `x0` est non nul, la prédiction est validée, et le processeur engage (*commits*) les actions des instructions spéculées, puis continue l'exécution.

Exécution spéculative dans les processeurs

De nombreuses instructions peuvent être exécutées de manière spéculative :

- arithmétique, logique
- branchements
- lectures en mémoire
- écritures en mémoire
  (tant que ça reste dans le tampon d'écriture du processeur).

Le processeur peut revenir en arrière sur ces exécutions en annulant les modifications des registres et les écritures mémoire.

Exécution spéculative dans les processeurs

De nombreuses instructions peuvent être exécutées de manière spéculative :

- arithmétique, logique
- branchements
- lectures en mémoire (avec accès au cache)
- écritures en mémoire
  (tant que ça reste dans le tampon d'écriture du processeur).

Le processeur peut revenir en arrière sur ces exécutions en annulant les modifications des registres et les écritures mémoire. Mais l'état du cache est conservé.

La famille d'attaque Spectre

Principe :

Un code de confiance, exécuté de manière spéculative,
lit en mémoire à une adresse qui dépend d'un secret.

L'attaquant mesure l'état du cache et en déduit une partie du secret.

Spectre v1 : contourner les tests de bornes de tableaux

```
const unsigned int len = ...;
unsigned char buff[len];

int f(unsigned int idx, int table[256 * CACHE_LINE_SIZE])
{
    int i;
    if (idx < len)
        return table[buff[idx] * CACHE_LINE_SIZE];
    else
        return -1;
}
```

La fonction `f` s'exécute en mode privilégié, p.ex. dans le noyau.

Les paramètres `idx` et `table` sont contrôlés par l'attaquant.

Spectre v1 : contourner les tests de bornes de tableaux

```
const unsigned int len = ...;
unsigned char buff[len];

int f(unsigned int idx, int table[256 * CACHE_LINE_SIZE])
{
    int i;
    if (idx < len)
        return table[buff[idx] * CACHE_LINE_SIZE];
    else
        return -1;
}
```

L'attaquant appelle `f` plusieurs fois avec `idx` valide (pour entraîner la prédiction), puis prépare le cache et appelle `f` avec `idx` trop grand.

Spectre v1 : contourner les tests de bornes de tableaux

```
const unsigned int len = ...;
unsigned char buff[len];

int f(unsigned int idx, int table[256 * CACHE_LINE_SIZE])
{
    int i;
    if (idx < len)
        return table[buff[idx] * CACHE_LINE_SIZE];
    else
        return -1;
}
```

La branche `then` du `if` est exécutée spéculativement.

La valeur de l'octet à l'adresse `buff + idx` fuite via le cache.

Cela permet de lire un bon bout de l'espace mémoire du noyau.

Protection contre Spectre v1

Endurcir les tests de bornes de tableaux contre la spéculation (macros `_nospec` dans le noyau Linux).

Accès habituel avec test de bornes :

```
T safe_read(T * tbl, unsigned len, unsigned idx)
{
    if (idx >= len) abort();
    return tbl[idx];
}
```

Protection contre Spectre v1

Endurcir les tests de bornes de tableaux contre la spéculation (macros `_nospec` dans le noyau Linux).

Accès endurci contre la spéculation :

```
T safe_read_nospec(T * tbl, unsigned len, unsigned idx)
{
    if (idx >= len) abort();
    return tbl[sel(idx < len, idx, 0)];
}
```

L'effet de `sel(idx < len, idx, 0)` est d'écrêter `idx` pour qu'il ne soit jamais trop grand pendant une exécution spéculative.

En exécution normale, `idx < len` et l'accès se fait bien à l'index `idx`.

Variante : tromper le vérificateur statique de BPF

(Schlüter, Borkmann, Krysiuk, *BPF and Spectre* PRISC 2022.)

`r1` : pointeur valide vers une variable accessible.
`r2` : entier quelconque contrôlé par l'attaquant

```
1:   if r0 != 0 goto line 3
2:   r1 = r2
3:   if r0 != 1 goto line 5
4:   r2 = load(r1)
5:   // faire fuiter r2
```

Le vérificateur sait que `r0` ne peut être à la fois 0 et 1, et donc l'accès `load(r1)` en ligne 4 est à une adresse valide.

Si les deux branchements conditionnels (lignes 1 et 3) sont prédits non pris, on lit spéculativement à l'adresse `r2`.

Généralisation : attaques par exécutions transitoires

De nombreux types d'états transitoires du processeur peuvent faire fuiter des données via des canaux temporels.

$\rightarrow$ Séminaire de F. Piessens le 21/04.

Spectre v1 Bounds Check Bypass	2017-5753
Spectre v2 Branch Target Injection	2017-5715
SpectreRSB[25]/**ret2spec**[26] Return Mispredict	2018-15572
Meltdown Rogue Data Cache Load	2017-5754
Spectre-NG v3a	2018-3640
Spectre-NG v4 Speculative Store Bypass	2018-3639
Foreshadow L1 Terminal Fault, L1TF	2018-3615
Spectre-NG Lazy FP State Restore	2018-3665
Spectre-NG v1.1 Bounds Check Bypass Store	2018-3693
Spectre-NG v1.2 Read-only Protection Bypass (RPB)	
Foreshadow-OS L1 Terminal Fault (L1TF)	2018-3620
Foreshadow-VMM L1 Terminal Fault (L1TF)	2018-3646
RIDL/ZombieLoad Microarchitectural Fill Buffer Data Sampling (MFBDS)	2018-12130

RIDL Microarchitectural Load Port Data Sampling (MLPDS)	2018-12127
RIDL Microarchitectural Data Sampling Uncacheable Memory (MDSUM)	2019-11091
Fallout Microarchitectural Store Buffer Data Sampling (MSBDS)	2018-12126
Spectre SWAPGS[34][35][36]	2019-1125
RIDL/ZombieLoad v2 Transactional Asynchronous Abort (TAA)[37][38][39]	2019-11135
RIDL/CacheOut L1D Eviction Sampling (L1DES)[41][42][43]	2020-0549
RIDL Vector Register Sampling (VRS)[41][42]	2020-0548
Load Value Injection (LVI)[44][45][46][47]	2020-0551
Take a Way[48][49]	
CROSSTalk Special Register Buffer Data Sampling (SRBDS)[52][53][54]	2020-0543
Blindside[55]	
Branch History Injection (BHI)	CVE-2022-0001 CVE-2022-0002

Point d’étape

Bilan sur les attaques par observation du temps ou du cache

Le temps d'exécution est une source non négligeable de fuites d'information.

Ces fuites sont considérablement amplifiées par les traits des processeurs modernes : caches, spéculation, etc.

Quelques fonctionnalités (cryptographie) peuvent être endurcies contre ces attaques, par programmation en temps constant, ou par masquage, ou par assistance matérielle (coprocesseurs).

Quelques protections (incomplètes ?) dans les systèmes d'exploitation et les navigateurs (moteurs d'exécution JavaScript).

Les enclaves Intel SGX sont en voie d'abandon car trop vulnérables aux attaques par exécution transientes.

Autres attaques reliées

Par observation :

- Consommation électrique instantanée.
- Émissions électromagnétiques.
- Et bien plus $\rightarrow$ chap. 19 de *Security Engineering*, Anderson.

Par perturbation :

- Injection de fautes $\rightarrow$ séminaire de K. Heydemann le 7 avril