SeseLab: une plateforme logicielle pour faciliter l'enseignement des attaques physiques

Pablo Rauzy
Université Paris 8 / LIASD
Saint-Denis, France
Email: pablo.rauzy@univ–paris8.fr

***Abstract*—L'actualité de la sécurité informatique ne manque pas de nous rappeler la nécessité de faire part à nos étudiant·e·s de la puissance des attaques par canaux auxiliaires, y compris dans les formations qui ne sont pas concentrées sur la sécurité. Dans le cadre d'un cours d'introduction à la sécurité des systèmes embarqués, j'ai voulu faire expérimenter ces attaques aux étudiant·e·s. Le matériel nécessaire à la mise en œuvre d'une attaque physique, en plus d'être coûteux et encombrant, nécessiterait un apprentissage spécifique et donc un encadrement conséquent. Pour pallier ces soucis, je présente ici SeseLab, une plateforme entièrement logicielle utilisable dans des salles de TP classiques, permettant tout de même aux étudiant·e·s de simuler des attaques physiques sur des implémentations cryptographiques.**

I. Introduction

Il n'y a évidemment pas besoin de convaincre le public de RESSI de l'importance de l'enseignement de la sécurité des systèmes d'information. De plus, l'actualité de la sécurité informatique ne manque pas de nous rappeler la nécessité de faire part à nos étudiant·e·s de la puissance des attaques par canaux auxiliaires [LSG+18], [KGG+18], y compris dans les formations qui ne sont pas concentrées sur la sécurité.

Dans certaines formations spécialisées dans la sécurité, il est possible de consacrer un module d'enseignement complet à la sécurité matérielle et donc aux attaques physiques. Cela peut notamment permettre d'avoir le temps de faire prendre en main du matériel spécifique aux étudiant·e·s (la figure 1 représente par exemple une plateforme permettant de réaliser des attaques par injection de fautes).

Dans des formations plus généralistes, ou spécialisées dans d'autres domaines que la sécurité, l'ensemble des aspects de notre domaine sont souvent concentrés en un seul module d'enseignement. En plus de cela, le matériel coûteux et encombrant nécessaire à la mise en œuvre d'attaques physiques n'est tout simplement pas présent. Dans ces cas là, il est utile de disposer de solutions clé-en-main permettant aux étudiant·e·s d'appréhender les attaques physiques plus rapidement, dans une salle de TP ordinaire (c'est à dire simplement équipée de postes informatiques).

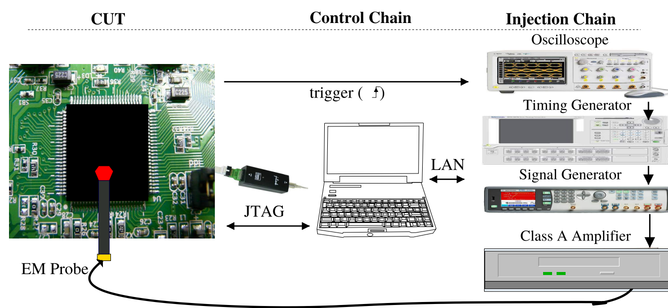

Fig. 1. Plateforme d'injection électromagnétique de fautes.

Une autre motivation pour ce type de solution est l'augmentation du nombre d'étudiant·e·s dans nos formations alors même que nos moyens matériels et humains diminuent. S'il est encore à peu près possible d'encadrer correctement seul·e une séance de TP avec 50 étudiant·e·s sur machine, ça devient plus compliqué quand chacun·e de ces étudiant·e·s manipule du matériel qu'illes n'ont pas l'habitude d'utiliser.

Ce sont ces raisons qui m'ont amené à développer la plateforme SeseLab[1], une plateforme entièrement logicielle utilisable dans des salles de TP classiques, permettant de donner aux étudiant·e·s la possibilité de simuler des attaques physiques simples, aussi bien passives (via une analyse de consommation de courant) qu'actives (par des injections de fautes), typiquement sur des implémentations cryptographiques.

La prochaine section introduit la plateforme SeseLab, et la suivante son utilisation à travers deux séances de TP d'introduction aux attaques physiques.

II. La plateforme SeseLab

La plateforme SeseLab est composée de deux parties. La première et la plus importante, écrite en Python, simule le matériel. Elle est composée d'un microcontrôleur, d'une sonde permettant de mesurer son activité électrique, et d'un banc d'essais permettant de lancer l'exécution des programmes écrit en assembleur SeseASM et optionnellement d'injecter des fautes pendant l'exécution. La seconde partie de SeseLab est écrite en assembleur SeseASM, et implémente une bibliothèque de manipulation de grands nombres afin de faciliter l'implémentation d'algorithmes cryptographiques.

A. Le "matériel"

Le code de SeseLab simule un microcontrôleur simple (on pourrait dire "idéalisé"). Il est essentiellement composé d'un module qui joue le rôle de processeur (`cpu.py`) qui utilise lui

[1] https://pablo.rauzy.name/software.html#seselab

même un module qui gère la mémoire (`memory.py`). Le rôle du processeur est d'exécuter une à une les instructions assembleurs (qui, si on veut garder l'analogie avec une plateforme physique seraient chargées dans une ROM).

Le module qui gère la mémoire, aussi bien les registres que la RAM, est équipé pour simuler la consommation électrique du système en utilisant le modèle répandu des poids et distances de Hamming [KJJ99]. Un module joue le rôle de sonde (`probe.py`) et sert simplement à enregistrer au fur et à mesure de l'exécution l'activité électrique de chaque cycle d'exécution du CPU. L'activité électrique est calculée comme étant la distance de Hamming des mises à jour de valeurs dans les registres et en RAM à laquelle on ajoute, avec un coefficient de moindre importance, le poids de Hamming du système (c'est à dire des registres, de la RAM, mais aussi des adresses présentes sur les bus à la fin du cycle).

Un autre module nommé `compiler.py` est en charge de la transformation de programme écrit dans notre assembleur simplifié SeseASM (voir section II-B) en la représentation interne des instructions utilisée par le module `cpu.py`. L'assembleur est volontairement réduit pour une prise en main aussi rapide que possible.

Enfin, le module `bench.py` simule le banc d'essais : c'est lui qui fait appel à `compiler.py` puis instancie un microcontrôleur avec des registres et une RAM à zéro, ainsi qu'une ROM contenant seulement les instructions à exécuter.

C'est aussi ce module qui permet de simuler des injections de fautes. Les injections sont possibles n'importe quand pendant l'exécution en interrompant le programme avec `^C`. À ce moment là, SeseLab propose de choisir entre 1) une injection de fautes dans un registre, 2) un saut d'instruction, ou 3) abandonner l'exécution. Dans le premier cas, on doit alors choisir dans quel registre faire la faute puis entre une faute aléatoire et une mise à zéro du registre choisi. Dans le second cas, on choisit combien d'instruction sauter. Après ces perturbations, l'exécution reprend normalement (si possible) son cours jusqu'à sa fin ou une prochaine interruption.

Un programme *`foo.asm`* se lance avec la commande `python3 bench.py` *`foo.asm`* `conso.txt`, où `conso.txt` est le fichier qui contiendra à chaque ligne l'activité électrique simulée du cycle d'exécution correspondant. L'option `-i` après `bench.py` active la fonctionnalité permettant de simuler des injections de fautes pendant l'exécution.

B. L'assembleur SeseASM

Le processeur simulé dispose par défaut de 32 registres nommés `r0` à `r31`, mais ne dispose pas de mémoire cache (à la fois dans un soucis de simplicité et de proximité avec la réalité de certaines puces embarquées). Le registre `r31` est utilisé comme adresse de retour (cf. les instructions `cal` et `ret` plus bas). Par convention le registre `r30` est utilisé comme pointeur de pile.

Chaque cycle d'exécution consiste en 1) récupérer l'instruction courante et décoder cette instruction, 2) lire dans les registres et/ou dans les cases mémoire nécessaires, 3) faire le calcul correspondant à l'instruction, 4) écrire dans le registre ou dans la case mémoire destination, et enfin 5) écrire l'activité électrique simulée du cycle dans le fichier choisi par l'utilisateur.

Le jeu d'instructions est réduit aux 24 suivantes :

- **`nop`** : ne fait rien
- **`mov`** *`dst val`* : copie la valeur de *val* dans *dst*
- **`not`** *`dst val`* : écrit dans *dst* la valeur de la négation bit à bit de *val*
- **`and`** *`dst val1 val2`* : écrit dans *dst* le et logique bit à bit de *val1* et *val2*
- **`orr`** *`dst val1 val2`* : écrit dans *dst* le ou logique bit à bit de *val1* et *val2*
- **`xor`** *`dst val1 val2`* : écrit dans *dst* le ou exclusif bit à bit de *val1* et *val2*
- **`lsl`** *`dst val1 val2`* : écrit dans *dst val1* décalée de *val2* bits à gauche
- **`lsr`** *`dst val1 val2`* : écrit dans *dst val1* décalée de *val2* bits à droite
- **`min`** *`dst val1 val2`* : écrit dans *dst* le min de *val1* et *val2*
- **`max`** *`dst val1 val2`* : écrit dans *dst* le max de *val1* et *val2*
- **`add`** *`dst val1 val2`* : écrit dans *dst* la somme de *val1* et *val2*
- **`sub`** *`dst val1 val2`* : écrit dans *dst* la différence de *val1* et *val2*
- **`mul`** *`dst val1 val2`* : écrit dans *dst* le produit de *val1* et *val2*
- **`div`** *`dst val1 val2`* : écrit dans *dst* le quotient entier de *val1* et *val2*
- **`mod`** *`dst val1 val2`* : écrit dans *dst* le modulo de *val1* et *val2*
- **`jmp`** *`addr`* : saute à l'adresse *addr*
- **`beq`** *`addr val1 val2`* : saute à l'adresse *addr* si *val1* = *val2*
- **`bne`** *`addr val1 val2`* : saute à l'adresse *addr* si *val1* ≠ *val2*
- **`ret`** : saute à l'adresse contenu dans le registre `r31`
- **`cal`** *`addr`* : écrit dans `r31` l'adresse de l'instruction suivante et saute à *addr*
- **`cmp`** *`dst val1 val2`* : écrit dans *dst* 1 si *val1*<*val2*, -1 si *val1*>*val2*, 0 sinon
- **`prn`** *`val`* : affiche la valeur de *val* en base 10
- **`prx`** *`val`* : affiche la valeur de *val* en base 16 sur 2 chiffres (un octet)
- **`prc`** *`val`* : affiche le caractère ASCII correspondant à la valeur de *val*

Les valeurs (`val`, `val1`, et `val2` dans la liste d'instructions ci-dessus) peuvent être soit une valeur immédiate, notée `#`*`N`* (e.g. `#13`, `#42`) ; soit un registre, noté `r`*`N`* (e.g. `r2`, `r17`) ; soit une case mémoire, notée `@`*`N`* (e.g. `@12345`, `@67890`) ; soit une référence (adresse mémoire avec indirection), notée `!`*v* (e.g. `!r2`, `!@100`, ...), et dans ce cas il est aussi possible de spécifier un décalage, donné après une virgule (e.g. `!r12,#-3`, `!r12,r2`).

Les destinations (`dst` dans la liste d'instructions ci-dessus) valides sont les valeurs modifiables, c'est-à-dire toutes sauf les valeurs immédiates.

Les adresses (`addr`) sont données soit sous forme de valeur, auquel cas elles correspondent à l'index de l'instruction dans le code, soit via un *label*. Les labels peuvent être défini n'importe où dans le code avec *`label`*`:` et auront pour valeur l'index de l'instruction qui les suit. L'exécution démarre au label `main`.

Une directive `.include` permet d'inclure un autre fichier.

Des exemples de code sont présentés dans la section suivante qui présente la bibliothèque de grands nombres de la plateforme SeseLab.

C. La bibliothèque `bignum.asm`

Pour pouvoir implémenter des algorithmes cryptographiques il est nécessaire de pouvoir manipuler des grands nombres. Comme la manipulation de grands nombres n'est pas l'objet de cet enseignement, une bibliothèque qui se veut simple d'utilisation est fournie avec la la plateforme.

Les nombres y sont représentés en base 256 dans la mémoire (un chiffre = un octet), du chiffre de poids fort vers celui de poids faible, suivi de la taille (en chiffre) du nombre. Les fonctions de la bibliothèque manipulent des pointeurs vers cette dernière case.

Par exemple le nombre `0x12ab34cd` sera représenté en mémoire à l'adresse 100 de la façon suivante :

Adresse :	96	97	98	99	100
Valeur :	0x12	0xab	0x34	0xcd	4

Les fonctions de la bibliothèque attendent leurs arguments dans les registres r20 à r24 :

- `bignum_print` : affiche la valeur du bignum sur lequel pointe `r20`.
- `bignum_init` : initialise (à zéro) un bignum de taille `r21` à l'adresse `r20`.
- `bignum_cleanup` : nettoie le bignum pointé par `r20` de ses chiffres de poids forts à zéro (et réduit sa taille en conséquence).
- `bignum_zero` : met la taille du bignum pointé par `r20` à zéro si celui-ci vaut zéro, ne fait rien sinon.
- `bignum_clear` : met à zéro la valeur du bignum pointé par `r20` (sans réduire sa taille).
- `bignum_copy` : copie le bignum pointé par `r21` à l'adresse pointée par `r20`.
- `bignum_cmp` : même chose que l'instruction `cmp` de l'assembleur mais avec `r20` comme résultat et la comparaison des bignums pointés par `r21` et `r24`.
- `bignum_sub` : soustrait au bignum pointé par `r20` celui pointé par `r24` (le résultat doit être positif).
- `bignum_add` : écrit à l'adresse pointée par `r20` le bignum résultat de l'addition de ceux pointés par `r21` et `r22`, modulo celui pointé par `r24`.
- `bignum_rshift` : décale le bignum pointé par `r20` de `r21` bits vers la droite.
- `bignum_mul` : écrit à l'adresse pointée par `r20` le bignum résultat de la multiplication de ceux pointés par `r21` et `r22`, modulo celui pointé par `r24`.

Le code suivant est un programme complet qui initialise et affiche le grand nombre donné en exemple précédemment :

```
 1.  .include bignum.asm
 2.  main:
 3.    sub r30 r30 #1         ; decrement sp
 4.    mov !r30 r31           ; push ra
 5.    mov r20 #100           ; address
 6.    mov r21 #4             ; size
 7.    cal bignum_init
 8.    mov !r20,#-4 #18       ; 0x12
 9.    mov !r20,#-3 #171      ; 0xab
10.    mov !r20,#-2 #52       ; 0x34
11.    mov !r20,#-1 #205      ; 0xcd
12.    cal bignum_print
13.    mov r31 !r30           ; pop ra
14.    add r30 r30 #1         ; increment sp
15.    ret
```

Le code suivant montre l'implémentation de `bignum_cmp`, qui compare les grands nombres pointés par `r21` et `r24` et écrit le résultat dans `r20` :

```
 1.  bignum_cmp:
 2.    beq _bn_cmp_same r21 r24
 3.    cmp r20 !r21 !r24
 4.    bne _bn_cmp_done r20 #0
 5.    sub r25 #0 !r21
 6.  _bn_cmp_loop:
 7.    beq _bn_cmp_same r25 #0
 8.    cmp r20 !r21,r25 !r24,r25
 9.    bne _bn_cmp_done r20 #0
10.    add r25 r25 #1
11.    jmp _bn_cmp_loop
12.  _bn_cmp_same:
13.    mov r20 #0
14.  _bn_cmp_done:
15.    ret
```

III. Séances de TP

L'objectif des deux séances de TP basées sur SeseLab que je présente ici est purement introductif. Il s'agit avant tout de donner aux étudiant·e·s un aperçu du domaine des attaques physiques. Dans ce cadre, les deux séances tournent autour du cryptosystème RSA et d'un même algorithme simple à appréhender, nommément l'exponentiation modulaire "square-and-multiply", ici implémenté en Python :

```
1.  def modexp (base, exp, mod):
2.    result = 1
3.    base = base % mod
4.    while exp != 0:
5.      if exp & 1 == 1:
6.        result = (result * base) % mod
7.      base = (base * base) % mod
8.      exp >>= 1
9.    return result
```

A. *Séance SPA*

Le but de la première séance est de réaliser une attaque par analyse simple de consommation de courant [KJJ99].

La séance commence par une prise en main de la plateforme et du langage SeseASM via l'écriture de divers petits programmes. Elle se poursuit ensuite par l'utilisation de la bibliothèque `bignum.asm` pour une implémentation (guidée) de l'algorithme d'exponentiation modulaire "square-and-multiply".

La suite du TP consiste en l'utilisation du logiciel gnuplot pour tracer graphiquement les enregistrements de l'activité électrique simulée par SeseLab lors de l'exécution de l'algorithme. Les étudiant·e·s essayent ensuite à partir de plusieurs exemples générés avec leur code de trouver des corrélations entre la valeur de la clé secrète (l'exposant) et les traces. Par exemple, une confirmation que que le temps d'exécution dépend linéairement de la clé [Koc96].

Enfin, on va jusqu'à retrouver complètement la clé secrète dans la trace de consommation grâce à l'utilisation d'une instruction spéciale `dbg` de l'assembleur[2] qui permet de marquer précisément les tours de boucle de l'algorithme "square-and-multiply" (cf. la figure 2). L'obtention de cette information est possible aussi sur du vrai matériel, mais demande l'aide d'une sonde JTAG et d'un débogueur.

[2]Cette instruction génère un pic dans un second signal émis par SeseLab qui est normalement toujours à zéro.

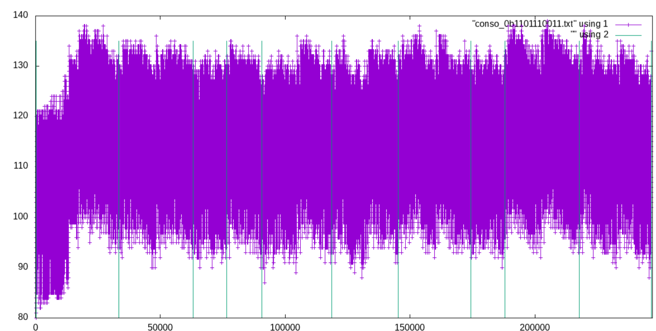

Fig. 2. Exemple de trace de consommation pour $e = 883$ (0b1101110011).

B. *Séance BellCoRe*

L'objectif de cette séance est de réaliser une attaque par injection de faute [BDL97]. L'attaque que l'on va réaliser est connue sous le nom de BellCoRe[3].

La séance commence par une meilleure familiarisation avec le cryptosystème RSA et son optimisation avec le théorème des restes chinois (CRT).

Soient p et q deux grands nombres premiers et $N = p \cdot q$. Soient e et d tels que $e \cdot d \equiv 1 \bmod \varphi(N)$, où φ est l'indicatrice d'Euler donc $\varphi(N) = (p-1) \cdot (q-1)$.

La paire (e, N) est la clé publique et (d, N) la clé privée. Pour retrouver d à partir de (e, N) il suffit de faire une inversion modulaire modulo $\varphi(N)$, mais calculer cette valeur demande de connaître p et q, c'est pour ça que la sécurité de RSA repose sur la factorisation de N.

La signature s d'un message m se calcule $s = m^e \bmod N$.

L'optimisation CRT consiste à faire deux calculs, un modulo p et un modulo q, plutôt qu'un seul modulo N, ainsi qu'une recombinaison modulo N. Les "petits" calculs sont $8\times$ plus rapide que l'original, et la recombinaison est très peu coûteuse. Pour cela la clé secrète devient (p, q, d_p, d_q, i_q), dans laquelle $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$, et $i_q = q^{-1} \bmod p$.

Cette fois-ci, on a $s = s_q + q \cdot (i_q \cdot (s_p - s_q) \bmod p)$. où $s_p = m^{d_p} \bmod p$, et $s_q = m^{d_q} \bmod q$.

Après ces rappels par la pratique qui permettent de constater que l'optimisation CRT de RSA permet de gagner un facteur presque 4 en temps d'exécution, commence la leçon sur le compromis entre optimisation pour la vitesse et optimisation pour la sécurité…

L'attaque BellCoRe consiste à injecter une faute à peu près n'importe où dans le calcul, de sorte à fauter soit le calcul de s_p soit celui de s_q. Si on réussit à injecter une telle faute, alors on obtient un résultat final faux $\widehat{s}$, et on peut retrouver p ou q très facilement en calculant $\text{pgcd}(s - \widehat{s}, N)$. En effet, $\forall x, \text{pgcd}(x, N)$ ne peut être égal qu'à N (x multiple de N), p resp. q (x multiple de p resp. q), ou 1 (x premier avec N).

D'un côté, si on faute le calcul de s_p en $\widehat{s_p} \neq s_p$, on a $s - \widehat{s} = q \cdot \big((i_q \cdot (s_p - s_q) \bmod p) - (i_q \cdot (\widehat{s_p} - s_q) \bmod p)\big)$.

De l'autre, si on faute le calcul de s_q en $\widehat{s_q} \neq s_q$, on a $s - \widehat{s} \equiv (s_q - \widehat{s_q}) - (q \bmod p) \cdot i_q \cdot (s_q - \widehat{s_q}) \equiv 0 \bmod p$.

Les étudiant·e·s ont à leur disposition une implémentation "obfusquée" de CRT-RSA en SeseASM. Illes sont alors amené·e·s à inspecter le code de `bignum.asm` pour trouver par exemple quel registre il est le plus intéressant de fauter pendant l'exécution pour avoir un effet sur le résultat final (ou à essayer au hasard pendant très longtemps…). Une fois un résultat faux obtenu, illes peuvent mettre en œuvre l'attaque BellCoRe et retrouver la clé secrète utilisée dans le code.

La figure 3 montre un exemple d'attaque dans le cas où la clé publique est $(e, N) = (17, 47775493107113604137)$. On retrouve p (ou q, peu importe) $= 8548494751$ avec le calcul de pgcd et directement derrière q (ou p) $= 5588760887$ en divisant N par le premier. Cela permet de retrouver la clé secrète $d = 22482584984930046353$ via une simple inversion modulaire modulo $\varphi(N)$.

[3]Comme *Bell Communication Research*, le laboratoire où les attaques par injection de fautes ont été découverte par l'équipe de Dan Boneh.

```
$ python3 bench.py -i crtrsa.asm /dev/null
s = 3f010be37eb5eca9
$ python3 bench.py -i crtrsa.asm /dev/null
^C
> Inject fault? (r = fault register, s = skip instruction, q = quit) r
> Which register? (0-32) 25
> Zero or random? (z = zero, r = random) z
! r25 zeroized
> Resuming...
s = 424a4dfd3625451d
$ python3
>>> # import de gcd et modinv
>>> gcd(47775493107113604137, 0x3f010be37eb5eca9 - 0x424a4dfd3625451d)
8548494751
>>> 47775493107113604137 // 8548494751
5588760887
>>> modinv(17, (8548494751 - 1) * (5588760887 - 1))
22482584984930046353
```

Fig. 3. Attaque BellCoRe avec SeseLab (en gras les entrées utilisateur).

IV. Conclusions et perspectives

Les retours des étudiant·e·s sur ces TP ont été globalement positifs, bien que peu nombreux étant donné que je n'ai pu donner ces TP que deux fois chacun. Il semble tout de même ressortir qu'illes préfèrent généralement le second TP, où réussir l'attaque leur semble moins "flou".

L'objectif de la présentation de ce matériel pédagogique à RESSI est double. D'une part, le partager avec la communauté, car je pense que les considérations évoquées en introduction et qui m'ont poussé à sa création sont partagées. D'autre part, avoir des retours de cette communauté, ce qui permettrait d'améliorer SeseLab plus rapidement que mes seuls retours d'expérience limité à une fois par an et par séance.

Au niveau pédagogique, une perspective évidente serait de produire des TP d'implémentation et d'étude de contre-mesures contre les attaques mise en œuvre dans les deux TP existants. Ce n'est pas faisable dans le cadre de la maquette du cours qui a donné naissance à la plateforme SeseLab, mais ça m'intéresserait de tout de même produire ce matériel à l'avenir, que ce soit dans un autre établissement ou à la faveur d'un changement de maquette chez nous.

Concernant la plateforme SeseLab, une évolution souhaitable serait l'implémentation d'une interface graphique permettant une utilisation plus simple et surtout une inspection des registres et de la mémoire en temps réel pendant l'exécution.

Peut-être qu'il serait intéressant de réduire les fonctionnalités des instructions de l'assembleur SeseASM pour les contraindre à travailler dans les registres, et de lui ajouter des instructions "load" et "store" pour en faire quelque chose qui ressemble de plus près à un processeur RISC réel (type MIPS).

References

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Proceedings of Eurocrypt'97*, 1997.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints*, 2018.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of CRYPTO'99*, 1999.

[Koc96] Paul Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of CRYPTO'96*, 1996.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, 2018.