Parcours École Ingénieur – 2ème année

PRJ 1401 – Rapport final

Attaque par cannaux auxiliaires : Cryptanalyse du système RSA

15 mai 2024

Par Etienne RICHARD, Matthieu CIZDZIEL et Mathis MOGUEDET

Table des matières

Introduction **2**

1 Définitions et notations **2**
1.1 Définitions 2
1.2 Notations 2

2 Contexte du projet **3**
2.1 Introduction 3
2.2 Généralités sur le RSA 3
2.3 Attaques par canaux auxiliaires 4
2.4 Matériel utilisé 4
2.5 Enjeux et objectifs du projet 5
2.6 Conclusion 5

3 Implémentation du RSA **6**
3.1 Introduction 6
3.2 Implémentation sur 32 bits 6
3.3 Rapprocher notre implémentation d'un réel 9
3.4 Implémentation du RSA avec des grands nombres ($>$ 32 bits) 9
3.5 Conclusion 10

4 Attaque **11**
4.1 Introduction 11
4.2 Installation de la ChipWhisperer (CW) 11
4.3 Fonctionnement de la ChipWhisperer (CW) 11
4.4 Premiers tests avec un projet développé par un membre de la communauté . 12
4.5 Conception de notre firmware 13
4.6 Cryptanalyse 14
4.7 Automatisation 16
4.8 Validation de l'automatisation 16

Conclusion **16**

Introduction

Durant notre 2ème année de PEI (Parcours Élève Ingénieur) nous avons été amené à entreprendre un projet pluridisciplinaire, c'est à dire un projet traitant d'un sujet lié à plusieurs UE (matière). Sachant que nous allons nous orienter vers un cycle ingénieur Cyberdéfense, nous avons décider de traiter d'un sujet qui allie à la fois Mathématiques, Informatique et Électronique. Le sujet de ce projet est : "Attaque par canaux auxiliaires : Cryptanalyse du système RSA", l'UE de ce projet sera encadré par Mme. Fatima Zahraa Kachkouch et notre tuteur sera M. Guy Gogniat. En effet, l'essor des technologies de chiffrement est indissociable de la sécurisation des communications et des données dans un monde de plus en plus connecté. Parmi les algorithmes de chiffrement les plus utilisés, le RSA (Rivest-Shamir-Adleman) occupe une place prépondérante. Son utilisation repose sur des principes mathématiques solides, mais sa sécurité dépend également de la robustesse de son implémentation. Dans le cadre de ce rapport, nous nous penchons sur l'implémentation du crypto-système RSA sur un firmware customisé de la ChipWhisperer. Nous détaillons les différentes étapes de cette implémentation, mettant en lumière les défis techniques rencontrés et les solutions adoptées pour les surmonter. En outre, nous examinons les implications de notre travail pour la sécurisation des systèmes embarqués et les possibilités d'attaques par canaux auxiliaires sur le RSA.

1 Définitions et notations

1.1 Définitions

BigInt : Grand entier en français, généralement représenté sur plusieurs mots mémoire.

Chunk : Une partie d'une entité, dans notre cas, un groupe de bits qui est utilisé pour stocker une partie d'un grand entier.

Débordement : Un événement se produisant lorsqu'un calcul dépasse les capacités de stockage d'un type de données, nécessitant une gestion spéciale.

ANSSI : Agence Nationale de la Sécurité des Systèmes d'Information

1.2 Notations

n : désigne le modulo utilisé pour les opérations modulaires

p *et* q : désigne deux nombres premiers tel que $n = p \times q$

e : désigne l'exposant public

d : désigne l'exposant privé

m : désigne un message sous forme de nombre, tel que $m \in \{0, 1, \ldots, n-1\}$

c : désigne le chiffré sous forme de nombre, tel que $c \in \{0, 1, \ldots, n-1\}$

$\phi(n)$: désigne l'indicatrice d'Euler

2 Contexte du projet

2.1 Introduction

En terme de sécurité informatique les méthodes de chiffrement sont très importantes notamment dans le domaine des télécommunication. La cryptographie est donc une notion très importante dans la cybersécurité actuelle. Les 3 principes de la cryptographie sont la confidentialité, l'authenticité et l'intégrité. La confidentialité assure que le contenu d'un message chiffré ne peut être lu que par son destinataire. L'authenticité assure l'origine du message, c'est à dire l'identité du messager. Enfin l'intégrité assure la non-modification d'un message. Dans notre cas nous mettrons en oeuvre des moyens pour s'attaquer à la confidentialité d'un protocole cryptographique. Ainsi nous allons vous présenter un algorithme connu pour sa robustesse et utilisé dans le monde entier, le protocole RSA.

2.2 Généralités sur le RSA

Nous ne pouvons pas commencer la présentation du projet sans présenter le protocole RSA. Le protocole RSA utilise une clé publique pour chiffrer le message et une clé privé pour le déchiffrer. Les clés publique et privé n'étant pas les mêmes, on parle de chiffrement asymétrique. La robustesse du RSA repose dans la difficulté de factoriser un produit de deux grands nombres premiers. Sans rentrer dans le détail de sa robustesse nous allons vous expliquer son fonctionnement :

Génération des clés
Soit p et q deux nombres premiers distincts. On pose alors $n = pq$ et $\phi(n) = (p-1)(q-1)$. On choisit ensuite un nombre e premier avec $\phi(n)$ et strictement inférieur à $\phi(n)$ On peut enfin calculer d l'inverse modulaire de e modulo $\phi(n)$. Ainsi le couple (n,e) constitue la clé publique et d la clé privée. Nous pouvons rapidement nous rendre compte qu'il n'est pas possible d'obtenir d sans connaitre la factorisation de n, c'est pourquoi on dit que le RSA repose sur un problème de factorisation.

Chiffrement
Soit un message représenté par un entier naturel m strictement inférieur à n et c le message chiffré. Nous avons la relation suivante :

$$c \equiv m^e[n] \tag{1}$$

Ainsi pour chiffrer un message m on utilise une fonction d'exponentiation modulaire utilisant la clé publique.

Déchiffrement
Nous pouvons de la même manière déchiffrer le message chiffré à l'aide de la clé privée. Nous avons ainsi la relation suivante :

$$m \equiv c^d[n] \tag{2}$$

Ainsi nous remarquons que la base du chiffrement et du déchiffrement repose sur une exponentiation modulaire. Le problème de l'exponentiation utilisé par le RSA est qu'elle ne peut pas être naïve, c'est à dire qu'on ne peut pas faire $x^n = x * x * x ... * x$ n fois. En effet, le RSA utilise des nombres de 2048 bits ce qui rend impossible l'utilisation d'un algorithme naïf. Nous verrons par la suite lors de l'implémentation du RSA qu'on peut utiliser l'algorithme d'exponentiation rapide pour des nombres de 2048 bits.

2.3 Attaques par canaux auxiliaires

Il est désormais temps de vous introduire au concept d'attaque par canaux auxiliaires. Selon Wikipedia [5] une attaque par canaux auxiliaires est une : "Attaque informatique qui, sans remettre en cause la robustesse théorique des méthodes et procédures de sécurité, recherche et exploite des failles dans leur implémentation, logicielle ou matérielle." En reprenant ce que nous vous avons présenter précédemment nous ne remettons pas en compte la robustesse du protocole RSA mais son implémentation utilisant l'algorithme d'exponentiation rapide. Ainsi il existe une multitude d'attaques par canaux auxiliaires comme les attaques temporelles basées sur le temps mis par l'algorithme pour effectuer certaines opération, les attaques pas sondage qui consiste à analyser un circuit en y posant une sonde ou encore les attaques par consommation de courant. Dans notre cas nous allons nous intéresser aux attaques par consommation de courant. Ces attaques sont basées sur le fait que chaque type d'opération que peut effectuer le processeur utilise plus ou moins de courant. On peut également généraliser ce principes pour les différentes fonctions d'un programme qui n'ont pas la même consommation électrique.

2.4 Matériel utilisé

Nous pouvons désormais présenter le matériel utilisé dans ce projet. Nous utiliserons un kit ChipWhisperer de la marque NewAE prêtée par notre tuteur M. Guy Goniat. Ce kit est composé de trois cartes différentes :

- Une carte cible qui fera tourner le firmware cible, c'est à dire le firmware implémentant le RSA ;
- Une carte mère CW308 sur laquelle nous branchons la carte cible, cette carte permet d'alimenter la carte cible ;
- Une carte d'acquisition servant d'interface entre l'ordinateur et la carte mère.Cette carte permet d'uploader le firmware sur la carte mère, de communiquer avec la carte cible et de récupérer la consommation de courant de la carte cible sur l'ordinateur

Nous pouvons nous aider de la figure ci-dessous pour mieux comprendre l'organisation des différents composants :

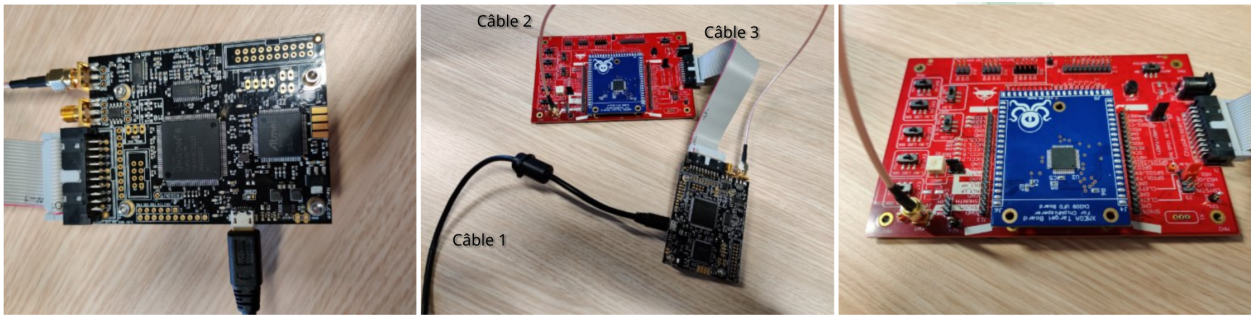

FIGURE 1 – Matériel utilisé pour le projet

Nous pouvons voir sur cette image 3 câbles différents. Le câble 1 permet de communiquer entre l'ordinateur et la carte d'acquisition. Il permet également d'alimenter l'ensemble du kit ChipWhisperer. Le câble 2 permet de récupérer la consommation du courant de la carte cible. Enfin le câble 3 permet de transmettre les informations entre la carte mère et la carte d'acquisition. Nous utiliserons également la librairie ChipWhisperer pour mener à bien ce projet.

2.5 Enjeux et objectifs du projet

Nous connaissons désormais tous les éléments nécessaires à la compréhension des enjeux et des objectifs de notre projet. L'objectif principal de ce projet est de récupérer la clé privée d'un utilisateur sans la connaitre, tout en restant dans un cadre légal. De ce fait nous attaquerons notre propre plateforme dédiée aux attaques par canaux auxiliaires. Les enjeux liés à ce projet sont nombreux, il faudra réussir à s'organiser autour d'un projet en équipe, comprendre le fonctionnement du kit ChipWhisperer et de la librairie associée et respecter les deadlines du planning établit lors de l'étude projet.

2.6 Conclusion

Ce projet vise ainsi à attaquer un protocole de cryptographie, le RSA, connu et utilisé pour sa robustesse. Cela sera possible grâce à l'implémentation du RSA et grâce à l'utilisation de matériels dédié à ce genre d'attaque. Un travail en équipe et une forte rigueur seront important dans ce projet puisque ce dernier est conséquent. Nous pouvons désormais nous intéresser un peu plus en détail à l'implémentation du RSA dans ce projet.

3 Implémentation du RSA

Après vous avoir introduit le contexte du projet, nous allons maintenant vous expliquer comment nous avons fait pour implémenter le crypto-système RSA sur un firmware customisé de la ChipWhisperer.

3.1 Introduction

Dans cette section, nous abordons l'implémentation du crypto-système RSA sur un firmware personnalisé de la ChipWhisperer. Après avoir établi le contexte du projet, nous décrivons en détail notre démarche pour mettre en œuvre le RSA sur des nombres de 32 bits, en nous appuyant sur les capacités natives du crypto-processeur de la plateforme. Nous explorons les différentes étapes nécessaires à cette implémentation, de la génération des clés à l'exécution des opérations de chiffrement et de déchiffrement, en passant par les algorithmes essentiels tels que l'exponentiation modulaire et le calcul de l'inverse modulaire. Par la suite, nous discutons des défis rencontrés lors de la transition vers une implémentation plus réaliste du RSA, conformément aux normes de sécurité établies par l'ANSSI. Enfin, nous examinons l'approche adoptée pour représenter et manipuler des nombres de grande taille (> 32 bits) dans un environnement embarqué limité, ainsi que les difficultés persistantes rencontrées dans l'implémentation des opérations de réduction modulaire.

3.2 Implémentation sur 32 bits

Dans un premier temps, nous avons décidé d'implémenter le RSA sur des petits nombres de taille supportés nativement par le crypto-processeur. Ce dernier utilise une architecture de 32 bits, cela signifie que lorsqu'on stock un nombre dessus, ce nombre ne peut être que de 32 bits maximum (de 0 à $2^{32} - 1$, pour des entiers positifs). Nous allons donc commencer par implémenter le RSA sur des nombres de 32 bits afin de pouvoir utiliser les opérations usuelles : addition, multiplication, réduction modulaire, ...

Comme nous l'avons vu dans la section précédente, pour implémenter le RSA, nous avons besoin de savoir trois choses différentes :

1. Générer des clés de chiffrement/déchiffrement ;
2. Chiffrer un message m, sur 32 bits $m < 2^{32}$;
3. Déchiffrer un chiffré c, sur 32 bits $c < 2^{32}$.

Les algorithmes nécessaire pour effectuer ces opérations sont : la génération de nombres premiers aléatoire afin de déterminer p et q, le calcul d'inverse modulaire pour déterminer la clé privée d tel que $ed \equiv 1[n]$ et l'exponentiation modulaire afin de chiffrer et déchiffrer, cf. équation (1) et (2).

Nous avons écrit le pseudo-code des ces deux algorithmes essentiels (cf. Algorithme 1 et 2), pour ce qui est de leur implémentation en C dans le firmware de la ChipWhisperer, vous pouvez les retrouver dans l'archive finale du projet.

Afin d'illustrer la différence de complexité entre les deux algorithmes d'exponentiation : le naïf et le rapide, vous pouvez retrouver sur la Figure 2 le graphe du nombres d'opération par rapport à l'exposant utilisé. On se rends compte que l'exponentiation rapide est bien plus efficace !

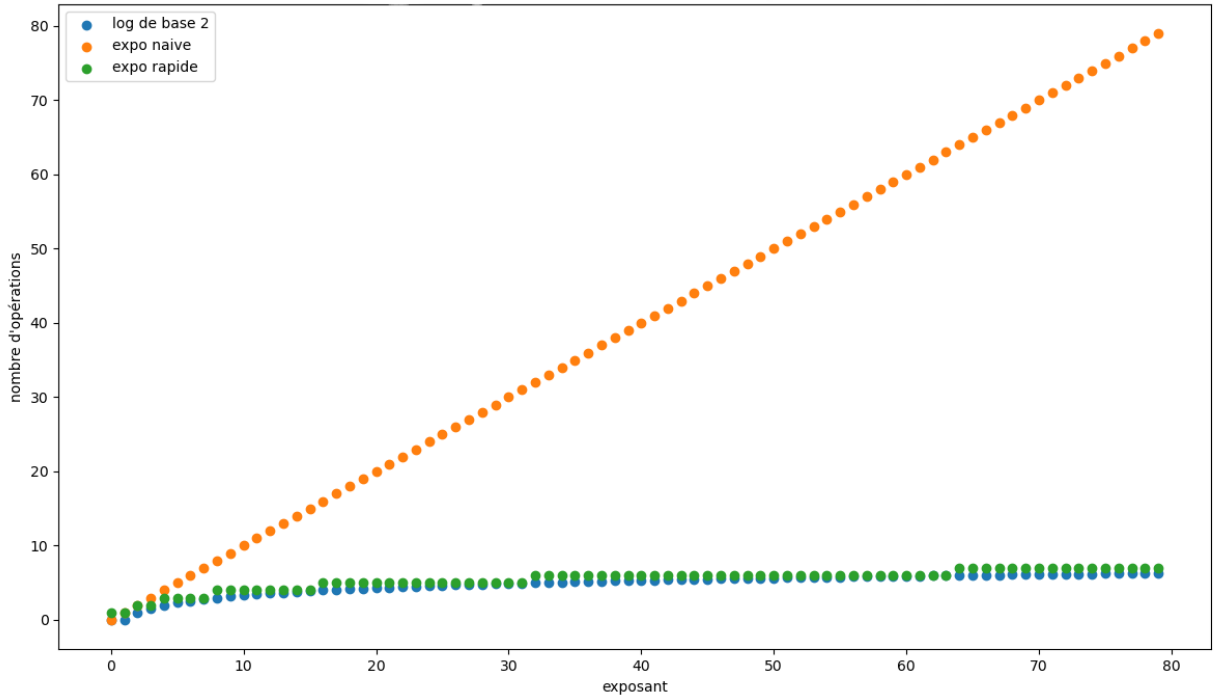

FIGURE 2 – Comparaison des complexités des deux algorithmes d'exponentiation

Il est aussi important de relever que la présence de l'algorithme d'exponentiation rapide n'est pas un hasard, c'est grâce à lui que le système est faillible. En effet, si il rencontre un 1 dans la représentation binaire du nombre il fait 2 calculs, sinon, s'il rencontre un 0, il en fait un seul. Grâce à ça, il a une fuite de la clé qui s'opèrera sur la trace de consommation électrique. On gagne en rapidité, mais attention, on fait fuir des données liée à la clé privée ...

Une fois ces primitives implémentées, il a été facile d'implémenter la génération de clé, le chiffrement et le déchiffrement, vous pouvez également retrouver ces implémentations en C dans l'archive final du projet.

Algorithm 1: Pseudo-code de l'exponentiation modulaire

Input: base, exponent, modulus
Output: result
$result \leftarrow 1$;
$base \leftarrow base \mod modulus$;
while $exponent > 0$ **do**
    if *exponent est impair* **then**
        $result \leftarrow (result \times base) \mod modulus$;
    end
    $exponent \leftarrow exponent \div 2$;
    $base \leftarrow (base \times base) \mod modulus$;
end
return $result$;

Algorithm 2: Pseudo-code de l'inverse modulaire

Input: a, m
Output: inverse de a modulo m
$m0 \leftarrow m$;
$y \leftarrow 0$;
$x \leftarrow 1$;
if $m = 1$ **then**
    return 0;
end
while $a > 1$ **do**
    $q \leftarrow a \div m$;
    $t \leftarrow m$;
    $m \leftarrow a \mod m$;
    $a \leftarrow t$;
    $t \leftarrow y$;
    $y \leftarrow x - q \times y$;
    $x \leftarrow t$;
end
if $x < 0$ **then**
    $x \leftarrow x + m0$;
end
return x;

3.3 Rapprocher notre implémentation d'un réel

A cette étape de l'implémentation du RSA, nous avons un RSA qui fonctionne sur 32 bits, le problème est que cette implémentation est très loin de la réalité.
En effet, d'après les recommandation de l'ANSSI [2], un RSA implémenté avec les types de bases du C (32 bits), n'est pas du tout sécurisé. Effectivement, nous pouvons retrouver page 19 les 4 règles suivantes :

1. La taille minimale du module est de 2048 bits, pour une utilisation ne devant pas dépasser la fin de l'année 2030.
2. Pour une utilisation à partir de l'année 2031, la taille minimale du module est de 3072 bits.
3. Les exposants secrets doivent être de même taille que le module.
4. Pour les applications de chiffrement, les exposants publics doivent être strictement supérieurs à $2^{16} = 65536$.

Afin d'attaquer une version du RSA qui pourrait être utilisé dans un cadre réel, nous allons donc devoir l'implémenter avec un module, notée n précédemment, de 2048 bits ! Or, comme vu précédemment nous allons utiliser une machine en 32 bits, ce qui signifie que les types de bases sur lesquels les opérations élémentaires sont implémentées, ne dépasse pas 32 bits. Nous allons devoir créer et implémenter des opérations sur les grands nombres, nous construirons un tableau de plusieurs nombres classique de 32 bits jusqu'à atteindre un nombre de 2048 bits.

3.4 Implémentation du RSA avec des grands nombres ($>$ 32 bits)

Pour représenter un grand entier dans un système embarqué 32 bits, nous utilisons un découpage en chunks de 16 bits. Imaginons que nous souhaitons un nombre de 64 bits. Nous divisons cet entier en mots de 16 bits. Ainsi, chaque chunk de 16 bits contient une partie du grand entier.

Prenons un exemple concret : supposons que notre grand entier soit `0xABCD1234567890EF`. Pour le découper en chunks de 16 bits, nous aurons :

- Chunk 1 : `0xABCD`
- Chunk 2 : `0x1234`
- Chunk 3 : `0x5678`
- Chunk 4 : `0x90EF`

Chaque chunk peut être représenté par un entier de 16 bits dans la mémoire du système embarqué. En utilisant cette méthode de découpage, nous pouvons stocker et manipuler des grands entiers même dans des systèmes avec des limitations de taille de données.
Si nous utilisons des sous-entiers de 16 bits et pas de 32 bits c'est parce que le 32 bits servira de type d'overflow, en effet lorsqu'on aura une sous opération qui dépassera la taille d'un chunk alors un entier de 32 bits permettra de résoudre ce problème.

Sachant que chaque chunk est sur 16 bits c'est comme si nous avons une représentation du nombre en base 2^{16}, nous n'avons "plus" qu'a considéré un chunk comme un symbole de la base 2^{16} et implémenté les opérations usuelles dessus.

Cette méthode fonctionne pour l'addition et la soustraction, mais pour les opérations d'inverse modulaire, de multiplication modulaire et de division, cela ne fonctionne plus et il faut utiliser des algorithmes plus poussés, nous nous sommes basé sur ceux du chapitre 14 du livre "Handbook of Applied Cryptography" [1]. Nous avons réussi à implémenter les fonctions suivante :

```
1 uint32_t bi_add(bigint_t* out, bigint_t* a, bigint_t* b);
2 uint32_t bi_mul(bigint_t* out, bigint_t* a, bigint_t* b);
3 uint32_t bi_sub(bigint_t *res, bigint_t *a, bigint_t *b);
4 void bi_exp_naif(bigint_t* res, bigint_t* base, uint32_t exp);
5 void bi_setzero(bigint_t* a);
6 void bi_mov(bigint_t* src, bigint_t* dst);
7 void print_bigint(bigint_t *a);
8 int bi_compare(bigint_t *a, bigint_t *b);
```

FIGURE 3 – Prototypes des fonctions implémentées sur les grands nombres

Le code source des ces fonctions se trouve dans l'archive finale du projet, malheureusement, comme vous pouvez le remarquer, nous n'avons pas réussi à implémenter les opérations de réductions modulaire et sans ça nous ne pouvons pas calculer modulo n, mais que faire des calculs sur $\mathbb{Z}$. Nous avons donc été incapable de faire un RSA grandeur nature, mais ce n'est pas non plus très grave car le principe de l'attaque reste le même et toutes les choses que nous avons faites par la suite seront adaptable aisément pour 2048 bits.

3.5 Conclusion

Cette section a mis en lumière les différentes étapes de notre implémentation du RSA sur un firmware customisé de la ChipWhisperer. Nous avons débuté en explorant les fondements du RSA sur des nombres de 32 bits, en utilisant les opérations élémentaires disponibles sur la plateforme. Cependant, nous avons rapidement confronté les limites de cette approche lorsque nous avons cherché à nous conformer aux standards de sécurité actuels, nécessitant l'utilisation de clés et de modules de taille beaucoup plus grande. Nous avons donc entrepris d'étendre notre implémentation pour manipuler des nombres de grande taille, en découpant ces nombres en chunks de 16 bits pour surmonter les contraintes de la plateforme. Malgré les progrès réalisés, des défis persistent, notamment dans l'implémentation des opérations de réduction modulaire, qui sont essentielles pour le fonctionnement du RSA à grande échelle. Néanmoins, ces efforts nous ont permis de mieux comprendre les complexités de

l'implémentation du RSA dans des environnements embarqués et de poser les bases pour des développements futurs dans ce domaine.

4 Attaque

4.1 Introduction

Dans cette partie nous allons voir comment l'attaque par analyse de consommation a été mise en place, c'est à dire comment fonctionne notre kit ChipWhisperer et utiliser pleinement ses fonctionnalités. Ceci comprend l'explication des firmwares, l'obtention et l'analyses des résultats obtenus.

4.2 Installation de la ChipWhisperer (CW)

Tout d'abord, nous avons pris en main le matériel, nous avons donc découvert les différentes cartes qui étaient à notre disposition dans le kit fournit. Deuxièmement, nous avons branché physiquement la carte d'acquisition à un ordinateur via un port USB, c'est cette carte qui nous enverra la consommation de notre cible.
Nous nous sommes ensuite penché sur la connexion avec un ordinateur, afin de pouvoir communiquer avec la ChipWhisperer. Nous avons du chercher longuement afin de trouver comment nous connecter à cette carte via l'ordinateur. Pour ce faire, nous avons principalement consulté le wiki ChipWhisperer [4] et d'autres forums afin de trouver une solution. Nous avons donc commencé par installer les différents packages nécessaires pour utiliser la carte. Cette partie fut compliqué car le guide d'installation de la ChipWhisperer n'expliquait pas la marche à suivre pour nos OS respectifs. Cependant il était écrit qu'il était tout de même possible d'installer les paquets nécessaires au fonctionnement de la carte. Il a ainsi fallut installer chaque paquets un par un en trouvant le bon paquet disponible sur nos distributions.
Ensuite nous avons installé JupyterNotebook, qui est un outil simple et accessible pour visualiser et organiser notre connexion et notre communication avec la carte. Il nous permettais de lancer chaque commande séparément et d'avoir le retour de la carte en conséquence. Nous devions aussi utiliser un environnement python spécifique à l'aide de "pyenv" pour que notre JupyterNotebook fonctionne correctement.

4.3 Fonctionnement de la ChipWhisperer (CW)

Maintenant que notre environnement de travail est configuré et prêt à être utilisé, il va falloir communiquer avec la carte ! En utilisant le logiciel JupyterNotebook précédemment installé, nous avons pu suivre des “notebooks” d'introduction. À la fin de ces “notebooks”, nous étions capable de flasher les firmwares d'introduction et de communiquer avec la carte en utilisant le protocole “SimpleSerial”. En sommes, voici l'environnement que nous avons réussi à établir :

1. Nous pouvons flasher un firmware sur la carte cible ;

2. Nous pouvons nous connecter sur la carte d'acquisition en utilisant Python ;
3. Nous pouvons communiquer avec la carte cible ;
4. Nous pouvons déclencher des captures de consommations électrique.

Une fois que nous avons complété les "notebooks tutoriels" fournit par le constructeur, nous allons essayer de faire une attaque très simple avec un firmware développé par un membre de la communauté.

4.4 Premiers tests avec un projet développé par un membre de la communauté

A ce niveau d'avancement du projet, nous sommes donc capable de flasher des firmwares déjà fait et de les utiliser en communiquant avec la carte cible à travers Python. Nous allons utiliser ces compétences pour essayé un projet développé par un membre de la communauté ChipWhisperer [3], ce projet est une petite version du RSA avec les bons "notebooks" fournit et le firmware déjà créer. Nous avions juste à lancer le script python pour voir apparaitre cette trace de consommation électrique, cf Figure 4.

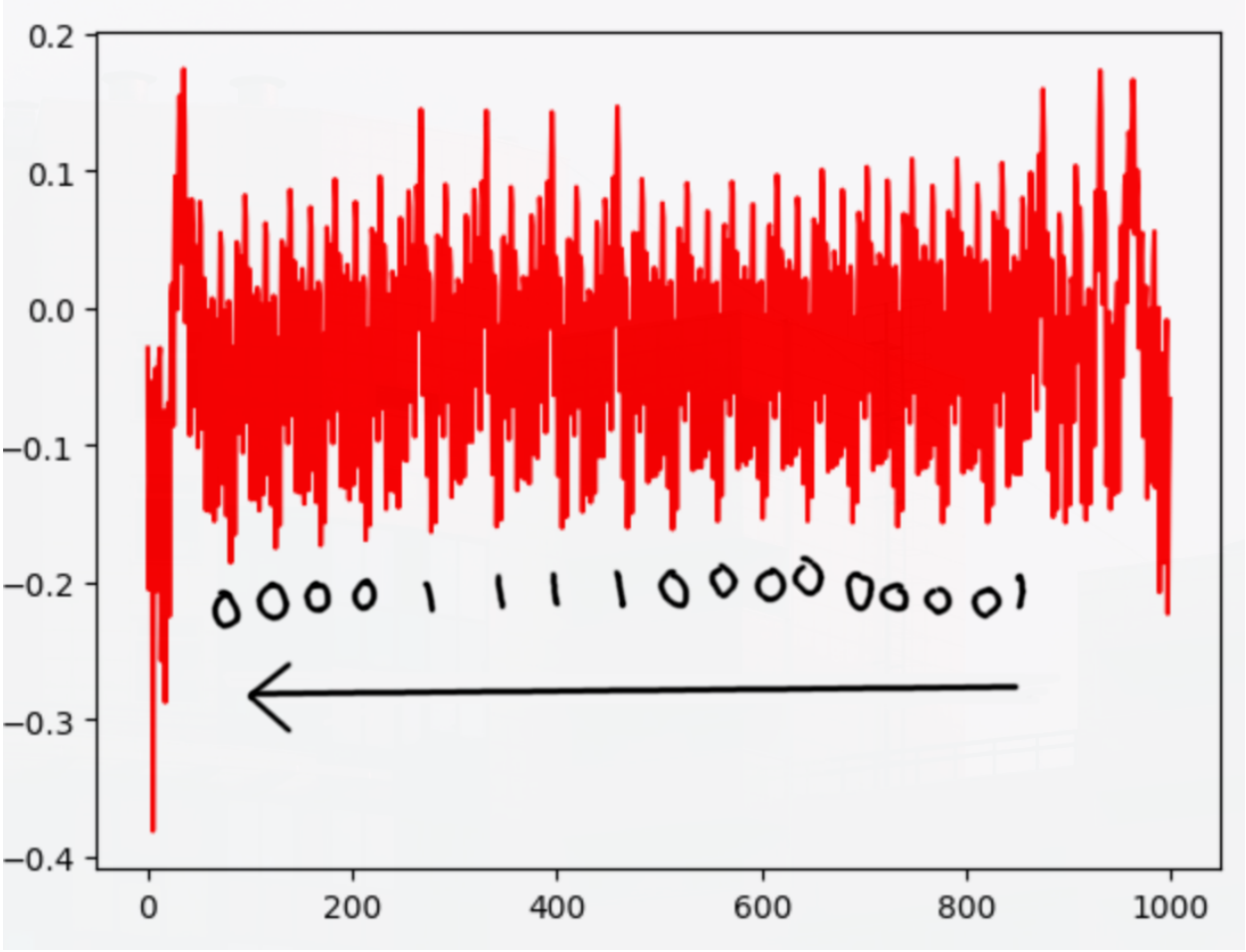

FIGURE 4 – Schéma expliquant la commande de percussion - percussions actives

Cette expérience à permis de nous mettre en confiance car nous nous sommes rendu compte qu'il est possible d'obtenir un résultat qui fonctionne avec notre matériel. En effet, nous pouvons lire aisément la clé sur la trace de consommation électrique, nous vous expliquerons plus en détail dans la suite de ce rapport la procédure à suivre. Nous allons donc pouvoir passer à l'implémentation de notre propre firmware.

4.5 Conception de notre firmware

À l'aide des algorithmes que nous vous avons expliqué et que nous avons implémenté en C dans la section précédente, il a été très facile pour nous de créer le fimware contenant le système RSA implémenté dedans. De plus avec l'expérience que nous avons acquise plus tôt, nous avons pu créer notre "notebook" facilement également. Nous avons donc tout qui est prêt pour passer à la cryptanalyse !

4.6 Cryptanalyse

Une fois que nous avions les traces de consommations, nous devons les exploiter afin de retrouver la clé privée utilisée et donc retrouver le message en clair. Les traces était sous la forme de tableaux sous python, il nous fallait donc utiliser la librairie matplotlib pour afficher les graphiques correspondants aux traces. Nous obtenions alors des traces comme celle ci-dessous :

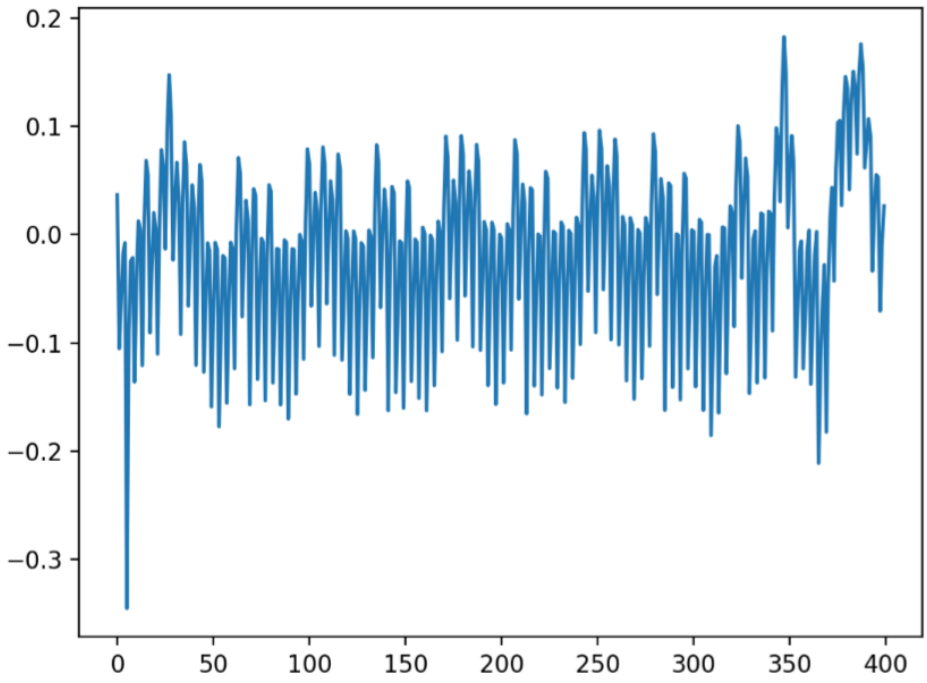

FIGURE 5 – Trace brute de la consommation de courant

Dans ce cas nous avons utilisé la clé privé 11010101. Cette trace n'est pas très lisible et nous avons donc utilisé une transformée de fourrier pour lisser la courbe. Après modification de la trace nous obtenons le graphique suivant :

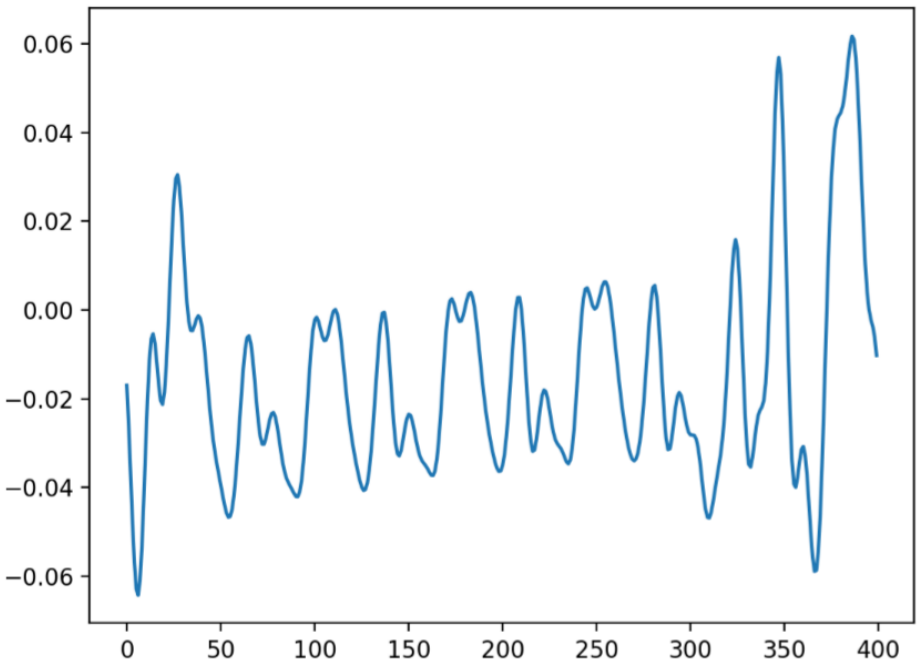

FIGURE 6 – Trace lissée de la consommation de courant

Nous voyons ainsi des patterns apparaître clairement sur le graphique. l'algorithme d'exponentiation rapide se basant sur la représentation binaire de l'exposant les patterns apparaissant sur la trace correspondent aux 0 et 1 constituant l'exposant. La particularité de l'algorithme d'exponentiation rapide est qu'il parcours l'exposant du bit de poids faible au bit de poids fort et en fonction de la parité du bit actuel il va effectuer certaines opérations. Ainsi nous pouvons retrouver la clé utilisée pour déchiffrer le message. Nous devons regarder les patterns présents en partant de la droite vers la gauche pour trouver la clé privé. Ayant au préalable testé l'attaque avec les clés 11111111 et 00000000 nous connaissions les correspondance entre les patterns et les chiffres binaires. Ainsi nous avons les correspondances suivantes :

FIGURE 7 – Pattern pour un 1

FIGURE 8 – Pattern pour un 0

Nous pouvons ainsi retrouver la clé privé qui est 11010101 ce qui correspond bien à la clé utilisée.

4.7 Automatisation

La partie automatisation est assez importante car si nous devions déchiffrer sur 2048 bits, c'est à dire analyser une trace qui comprend 2048 pics de consommations différents, visuellement, la tache s'avérait longue et fastidieuse. L'automatisation d'une recherche de clés est donc essentielle et nous permet de gagner du temps d'analyse. Pour automatiser la recherche de patterns nous voulions dans un premier temps utiliser une IA de classification binaire mais nous nous somme rendu compte que celle ci ne permet pas d'avoir une précision assez élevée. Ainsi sur 2048 bits, si 3 ou 4 bits sont faux sans qu'on sache lesquels nous ne pourront pas déchiffrer le message. Nous avons alors décidé d'utiliser un algorithme spécialisé dans la comparaison de pattern, l'algorithme de Pearson. Cette algorithme prend en entrée deux patterns et renvoie en sortie un coefficient de corrélation compris entre -1 et 1. A savoir que si le coefficient se tend vers -1 alors les deux patterns sont inversé, si le coefficient tend vers 0 alors les patterns n'ont rien en commun et enfin si le coefficient tend vers 1 les patterns sont très similaires. Nous avons ainsi utilisé un seuil de détection de 0,9 pour affirmer que deux patterns sont les mêmes. Nous avons ainsi développé une fonction qui parcours la trace et compare l'échantillon parcouru avec deux patterns correspondant au 1 et au 0. Ainsi en utilisant cette fonction sur les traces vu précédemment, nous obtenons la clé 11010101.

4.8 Validation de l'automatisation

Afin de valider notre fonction d'automatisation, nous avons utilisé différentes clés générées aléatoirement et nous avons comparé ces clés avec le résultat de notre fonction. Le résultat est que sur 50 clés testés nous avons un taux de réussite de 100% ce qui est très appréciable.

Conclusion

La mise en œuvre du crypto-système RSA sur un firmware personnalisé de la ChipWhisperer représente un défi technique significatif, mais aussi une opportunité d'approfondir notre compréhension des mécanismes de chiffrement et des contraintes liées aux environnements embarqués. Notre travail nous a permis de mettre en évidence les différentes étapes de l'implémentation du RSA, en mettant l'accent sur les adaptations nécessaires pour surmonter les limitations matérielles et les exigences de sécurité. Cependant, des défis subsistent, notamment en ce qui concerne la sécurisation des opérations de réduction modulaire et la prévention des attaques par canaux auxiliaires. Néanmoins, notre exploration ouvre la voie à de futures recherches visant à renforcer la sécurité des systèmes embarqués et à améliorer la résilience des algorithmes de chiffrement face aux menaces émergentes.

Références

[1] P. van Oorschot A. Menezes et S. Vanston. *Handbook of Applied Cryptography*. www.cacr.math.uwaterloo.ca/hac. 1996. URL : https://cacr.uwaterloo.ca/hac/about/chap14.pdf.

[2] ANSSI. *Guide des mécanismes cryptographiques*. Version 2.04 - 2020-01-01. Règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques. 2021. URL : https://cyber.gouv.fr/publications/mecanismes-cryptographiques.

[3] Hellman. *ChipWhisperer mini-RSA SPA tutorial*. 2023. URL : https://gist.github.com/hellman/c10a7a0dfe7f70d3615269b020aa3286.

[4] NewAE Technology. *Documentation for the ChipWhisperer software*. 2023. URL : https://chipwhisperer.readthedocs.io/en/latest/index.html.

[5] Wikipédia. *Attaque par canal auxiliaire*. 2023. URL : https://fr.wikipedia.org/wiki/Attaque_par_canal_auxiliaire.