UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

« Sébanjila Kevin BUKASA »

« Analyse de vulnérabilité des systèmes embarqués face aux attaques physiques »

Thèse présentée et soutenue à RENNES , le 8 juillet 2019
Unité de recherche : Institut National de Recherche en Informatique et Automatique (INRIA), Rennes – Équipe CIDRE

Rapporteurs avant soutenance :
Giorgio Di Natale, TIMA – Université de Grenoble
Karine Heydemann, Maître de conférences, LIP6 – Université Pierre et Marie Curie

Composition du jury :

Président :	Pierre-Alain FOUQUE, Professeur des Universités, Université Rennes 1
Examinateurs :	Guillaume BOUFFARD, Ingénieur de recherche, ANSSI
	Giorgio DI NATALE, Directeur de recherches, TIMA – Université de Grenoble
	Ronan LASHERMES, Ingénieur de recherche, INRIA
Dir. de thèse :	Jean-Louis LANET, Chercheur, INRIA
Invité(s) :	Hélène LE BOUDER, Maître de conférence, IMT-ATLANTIQUE

"*Our deepest fear is not that we are inadequate. Our deepest fear is that we are powerful beyond measure. It is our light, not our darkness, that most frightens us.*"

Timo Cruz dans Coach Carter
(extrait de "A Return to Love" de Marianne Williamson)

Aux équipes enseignantes qui m'ont accompagné au long de ma scolarité et qui n'ont eu de cesse de m'apprendre à croire en moi.
À mes encadrants qui ont su m'inspirer et me motiver.
À mes frères et sœurs qui m'avez observés me forçant à me dépasser.
À mes parents qui ont été tours à tours des modèles, des exemples, et des sout des soutiens mais également des exemples
Ce manuscrit vous est dédié.

Remerciements

Merci. Ce simple mot permet d'exprimer sa gratitude envers des personnes qui ont, par leur présence ou par leurs actes, affecté une situation particulière. Dans mon cas, la situation a été parfois routinière et parfois très difficile mais a duré 3 années, durant lesquelles j'ai eu l'occasion de rencontrer un grand nombre de personnes, je souhaite ici exprimer ma gratitude auprès d'un certain nombre d'entre elles. Cependant, n'étant pas disruptif dans l'univers du remerciement, je m'excuse par avance pour "celleux" (*sic*) que j'aurais oublié.

Tout d'abord, je souhaite remercier mon directeur de thèse Jean-Louis Lanet pour son soutien sans failles, sa disponibilité et surtout ses nombreux conseils grammaticaux et orthographiques ... Ensuite, je remercie également Ronan Lashermes, encadrant de son état, qui a été d'une aide inestimable dans ma formation de jeune chercheur, malgré l'évidente déception de ne pas m'avoir vu me rendre en Roumanie.

Ces travaux n'auraient pas été possible sans le soutien d'IDEMIA. Je remercie donc les différents interlocuteurs que j'ai eus au sein de cette entreprise, notamment Francis pour sa sympathie, son aide et sa disponibilité.

Je tiens à remercier tous les membres du personnel de l'INRIA et de l'Université, en particulier ceux avec lesquels j'ai pu échanger une discussion technique, une blague sur l'actualité ou un chocolat. Ils sont trop nombreux pour les citer mais ils ont eu un effet positif certain sur mon passage par ce laboratoire. Un remerciement spécial à Valérie, qui a accepté que je rejoigne son équipe.

J'adresse mes remerciements à tous mes (anciens) collègues doctorants ou déjà docteurs. D'abord ceux avec qui j'ai dû partager mon espace de travail, de ma première co-bureau, plutôt discrète, à mes derniers co-bureau, Babouche et Ricard. Ça a été un plaisir de pouvoir échanger sur nos lectures respectives, sur les citations des plus grands hommes ou sur le football. Sans oublier les passages de celui qui "ne parle pas français correctement", l'Américain, Greg et sa 205 ou encore Benji'. Ensuite Cogito et sa maitresse/maman Alix qui ont été des soutiens psychologiques de poids en plus d'avoir été des aides précieuses durant les premiers mois d'acclimatation. Aurélien qui a toujours eu le bon mot, la petite histoire ou le bon acte pour redonner le sourire (ou lancer un débat). Nisrine et Delphine les membres du bureau du bonheur, où tous les soucis des doctorants trouvaient une solution. Alexandre et Dr Richmond toujours à l'affût en cas de Mathsmergency. Ludo et Léo avec qui j'ai pu partager des expérimentations et des observations. Jésus, Farah, Rahaf et Routa des camarades venus d'autres équipes ou encore Raj, Mouad, Vasile, Pierre, Cédric, Charles, Ronny ou Mounir les collègues "de la dernière heure". Et enfin Laurent pour ses gadgets, ses idées et ses conseils toujours très avisés.

Pour terminer, je souhaite adresser mes remerciements à mes proches : Alex, Arthur, Caro, Cécile, Clémentine, Christine, Chloé, David, Freddy, Jessy, Justine, Mathieu, Mathilde, Marie, Mehdi, Melody, Melissa, Nicole, Pierre-Yves, Ronald, Sandra, Sarah, Stéphanie, Thomas, Valdy, Willy et enfin mes parents, il m'est difficile d'exprimer toute ma gratitude en seulement quelques mots. Par votre présence et votre patience dans les moments difficiles, vous m'avez permis de conserver une atmosphère saine, et vous m'avez permis de me concentrer sur ma thèse de la manière la plus sereine possible, tout ceci malgré mon manque de disponibilité durant cette période.

J'adresse également des remerciements spéciaux à Didier Chabenet, Yves Le Coent, Laura Monceaux, Stéphane Loret, Christian Grothoff et Stéphane Fay, qui m'ont permis tour à tour de découvrir l'informatique, ses métiers et la recherche au travers de discussions ou de collaborations. Sans ces rencontres ce manuscrit n'aurait probablement jamais vu le jour.

Encore une fois à vous tous et à tous les autres, je vous dis **Merci**.

Table des matières

Table des figures

Chapitre 1

Spécificité des appareils mobiles face à la sécurité

Résumé de la partie :

Cette partie présente l'objet d'étude de cette thèse, et les motivations de nos travaux. Ainsi dans un premier temps, un état des lieux des systèmes mobiles sera fait. Ensuite, nous aborderons la manière dont la sécurité de ces systèmes peut-être compromise. Nous évaluerons le profil des attaquants de ces systèmes, et nous verrons quels sont leurs moyens d'attaque. Un volet sur les défenses déjà mises en place sera effectué. Puis dans un troisième temps, nous verrons quel problème scientifique posent ses défenses. Et enfin nous terminerons par la motivation de nos moyens d'attaque et une présentation de nos dispositifs de test.

Sommaire

1.1 Contexte

1.1.1 Les systèmes embarqués

Les systèmes embarqués mobiles sont des systèmes électroniques et informatiques autonomes. Ils sont conçus pour traiter de l'information et réaliser des calculs tout en étant mobiles. Cela leur permet d'exécuter des applications en tout genre.

Ils représentent un large éventail d'objets du quotidien. On peut ainsi citer, les balances connectées qui stockent des courbes de poids de ses utilisateurs sur un cloud. Les smartphones qui permettent de communiquer avec le monde entier. Les télévisions qui offrent du contenu à la demande. Les télécommandes qui commandent tous les appareils d'une maison. Les enceintes connectées grâces auxquelles il est désormais possible de commander un taxi, etc.

Décrits dès le début des années 90 [GVNG94] ces systèmes comprennent des capteurs, des composants et au moins une interface pour communiquer avec l'utilisateur (ou une autre machine) afin de traiter les informations et d'exécuter les services attendus.

1.1.2 Les appareils mobiles

La loi de Moore [1], énoncée dans les années 70 par *Gordon E. Moore*, était une loi empirique du monde de l'électronique et de l'informatique. Elle semble aujourd'hui arriver à sa limite physique. Elle posait le postulat que le nombre de transistors dans les circuits intégrés doublait tous les deux ans (voir Figure 1.1). Par conséquent tous les deux ans, deux phénomènes pouvaient se produire :

— Les circuits intégrés devenaient plus performant pour une surface de silicium identique, le nombre et la densité des transistors serait multipliés par deux.

— Les circuits intégrés conservaient les mêmes performances mais devenaient moins imposants, en conservant la même densité, il devenait possible de diviser la taille de chaque transistor par deux.

Ceci a conduit à une rapide miniaturisation de ces circuits intégrés, et donc en conséquence des équipements les intégrant.

On constate dès lors, l'apparition de nouveaux types de systèmes embarqués (voir Figure 1.2).

En particulier, avec les téléphones mobiles qui commencent à embarquer des capteurs et des processeurs plus performant pour devenir les «smartphones» ou téléphones intelligents, que nous connaissons.

1.1.3 Enjeux

Autour des smartphones et avec l'avènement d'internet sont apparus un certain nombre d'autres systèmes. On peut ainsi citer, les périphériques IoT, ayant la capacité de se connecter de manière autonome à internet pour y échanger des données.

De nombreux écosystèmes se sont créés autour de ces périphériques. On constate ainsi l'apparition des maisons intelligentes, qui disposent de plusieurs IoT, pouvant communiquer entre eux et dont le smartphone sera le point de connexion avec l'utilisateur.

Les systèmes embarqués représentent aujourd'hui la plus large part des systèmes électroniques vendus [2] et leur nombre ne cesse d'augmenter [3] .

Ils sont employés dans tous les secteurs, et de plus en plus dans des domaines où les informations qu'ils traitent sont sensibles.

1. http://www.mooreslaw.org/
2. https://www.forbes.com/sites/louiscolumbus/2018/12/13/2018-roundup-of-internet-of-things-forecasts-and-market-estimates/
3. https://www.forbes.com/sites/louiscolumbus/2018/08/16/iot-market-predicted-to-double-by-2021-reaching-520b/

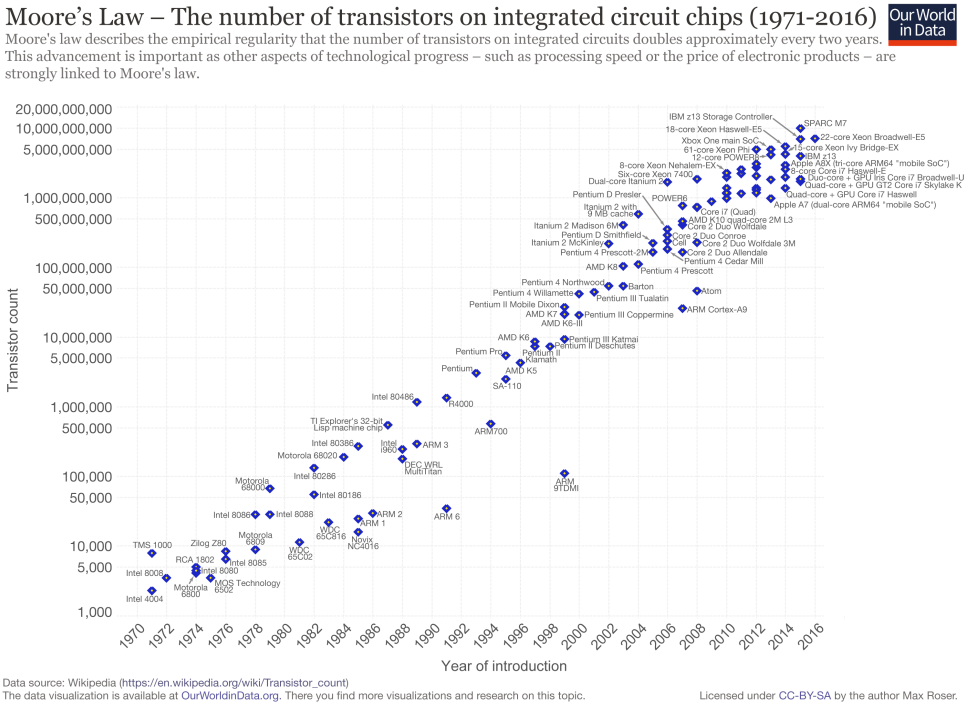

FIGURE 1.1 – Graphique montrant l'évolution du nombre de transistors années après années. Après avoir suivi la loi de Moore, jusqu'au début des années 2000, la croissance est aujourd'hui plus lente, la loi n'est plus valable.

Le smartphone ou la montre connectée servent aujourd'hui de terminal de paiement sans contact [Bet16, Ngu16]. Ils contiennent par conséquent toutes les informations sur nos cartes bancaires. Ils servent de gestionnaires de mots de passe [Cor17], et parfois même de vecteurs d'authentification à double facteur (2FA). La balance connait l'évolution de notre santé et peut donc disposer d'informations sensible pour des assurances. Les portails de nos maisons possèdent désormais des systèmes avancés pour authentifier les personnes à l'entrée, avec identification faciale ou historique des visites. Il existe encore de nombreux autres usages où les informations manipulées peuvent être détournées dans le but de nuire.

Ces systèmes aujourd'hui largement répandus, deviennent incontournables, ils renferment des informations à la sensibilité croissante. Du point de vue des industriels, le constat est le même. Ces appareils laissés à la disposition des utilisateurs renferment de la propriété intellectuelle, comme du code ou des clefs cryptographiques. Ces éléments, sont les atouts de ces systèmes. Les utilisateurs et les industriels veulent garder la main sur ceux-là.

Un nouvel usage tendant à se démocratiser mais qui pose problème aux employeurs est le Bring Your Own Device (BYOD), traduisible en français par «Amenez votre propre appareil». Cet usage encourage les employés à utiliser le même matériel pour la vie professionnelle et leur vie personnelle.

Les ordinateurs et smartphones combinent les deux usages et renferment des informations sensibles sur les deux usages qu'en fait l'utilisateur. Les isoler est donc une condition importante, à la fois pour l'employé qui ne veut pas que son employeur ait accès aux informations de sa vie privée mais aussi pour les employeurs qui ne souhaitent pas livrer des potentiels secrets professionnels à des personnes extérieures. L'ordinateur ou le smartphone pouvant être utilisé par un autre membre de la famille etc.

FIGURE 1.2 – Exemples de systèmes embarqués mobiles, une montre connectée et des smartphones. Ils ont bénéficié de la miniaturisation des circuits intégrés.

1.2 Problématiques de sécurité

La généralisation de ces systèmes et leur utilisation pour des tâches sensibles a rendu nécessaire l'utilisation de moyens de protections.

Au cours de ces travaux, nous avons souhaité évaluer ces systèmes et donc les protections qui y sont généralement appliquées. Pour cela, il a fallu dans un premier temps fixer les vecteurs d'attaques de ces appareil, ensuite nous avons dû identifier les réponses proposées par les constructeurs et les concepteurs de ces systèmes. Enfin il nous a fallu déterminer si ces protections étaient adaptées aux contextes dans lesquels évoluent ces systèmes.

Le but est donc d'évaluer la protection des atouts des systèmes embarqués mobiles. Il est donc nécessaire pour cela de fixer les limites de notre étude. Dans un premier temps, la menace que représente un attaquant par les pouvoirs qu'il possède dans le contexte d'utilisation de notre étude. Ensuite les différents vecteurs d'attaques que possède cet attaquant.

1.2.1 Modèle d'attaquant

L'attaquant est la personne ou l'entité qui aura un intérêt à rechercher des vulnérabilités dans un système pour profiter des informations que celui-ci pourrait contenir. Cet attaquant est caractérisé principalement par ses capacités (compétences et moyens à disposition) et par le but qu'il souhaite atteindre en brisant la sécurité du système.

Akhunzada *et al.* [ASA+15] décrivent un modèle d'attaquant, le Man-At-The-End (MATE). Il est ici question d'un attaquant qui a un accès physique au système qu'il souhaite attaquer. Il dispose d'une bonne connaissance du système mais ignore les informations non publiques à son sujet. L'attaquant possède un accès complet au système, dans la limite fixée par le concepteur. La présence de ports à haut niveau de permission comme les ports de débogages sont des éléments qui peuvent être utilisés. Cependant, en cas d'absence de ceux-ci, l'usage qu'il est possible de faire du système est un usage normal.

Pour illustrer un attaquant potentiel, un exemple récent s'est déroulé au cours de l'année 2016 [4]. Le FBI a retrouvé un smartphone appartenant à un terroriste et n'a pas pu le déverrouiller sans le code PIN, les agents ne disposant en effet, que d'un accès normal à l'appareil. Autre exemple fréquent, les cas d'informatique légale où des enquêteurs veulent accéder à un contenu verrouillé et sans possibilité de contacter le propriétaire. Dans un sens moins vertueux, l'attaquant peut aussi être une personne qui vise sa propre box TV afin d'obtenir l'accès à toutes les chaines de télévision.

La taxonomie proposée du MATE [ASA+15], précise cependant que quatre caractéristiques doivent être remplies pour considérer qu'il s'agit bien d'une attaquant de ce type :

4. https ://www.nextinpact.com/news/98646-apple-contre-fbi-chiffrement-sous-haute-tension.htm

— L'élément attaqué doit être du matériel, du logiciel ou tout type d'atouts, qui sera directement utilisable par l'attaquant.

— L'attaquant est humain et effectue l'attaque dans un cadre individuel, il ne s'agit pas d'une attaque à grande échelle ni perpétrée par ou contre une organisation.

— L'attaque est perpétrée intentionnellement.

— L'attaque a pour conséquence une brèche de sécurité ou une fuite des atouts.

Le MATE se caractérise donc par un usage normal du système. C'est le modèle que nous avons choisi d'incarner durant nos travaux. Cependant, nous avons parfois fait preuve de davantage de contrôle sur notre cible, afin de mettre en lumière d'autres vulnérabilités. Celles-ci, seront détaillées par la suite.

En suivant ce modèle, nous avons donc considéré que nous étions uniquement capable d'installer des applications de la manière prévue par le constructeur et que nous n'avions aucun accès à des informations qu'il aurait souhaité masquer. Nous avons considéré être en mesure d'utiliser toutes les fonctions accessibles au public, mais également de pouvoir profiter de toutes les brèches de sécurité existantes.

Le but des attaques selon ce modèle peut être de différente nature, ainsi il peut s'agir de vouloir un contrôle complet du système (Control-oriented attack (COA)) c'est le cas si l'attaquant souhaite débloquer des fonctions cachées par le constructeur, avoir accès à des chaines gratuites sur sa box TV par exemple. Il peut s'agir d'obtenir des informations secrètes contenues dans un système (Data-Oriented Attack (DOA)), c'est le cas lorsque les autorités cherchent à obtenir le mot de passe permettant d'accéder au téléphone d'un criminel. Enfin il peut s'agir de récupérer des informations sur la conception d'un système (Information-oriented attack (IOA)), ce qui peut être le cas d'un adepte du «Faites le vous-même» (DIY) qui souhaite réparer un système défectueux.

Les attaques vues vont donc se concentrer sur ces possibilités.

1.2.2 Attaques logicielles

Les attaques logicielles sont une classe d'attaques très connue dans le monde l'informatique. Le but de ces attaques est de retrouver des informations contenues dans le système ou de prendre le contrôle de celui-ci en utilisant des failles dans les logiciels qu'ils exécutent.

Cette classe d'attaques étant très largement connue dans le monde des systèmes informatiques, il est donc normal que l'on les retrouve appliquées au monde des systèmes embarqués.

Dans cette classe, il est possible de distinguer deux vecteurs d'attaques :

— **Remote :** l'accès au système est possible uniquement à distance en utilisant des moyens de connexion, comme internet, une connexion Bluetooth, un périphérique externe etc. Il n'est donc pas nécessaire de disposer du matériel directement en main, on peut attaquer depuis une certaine distance, voir à l'autre bout du monde

— **Local :** l'accès direct au système est indispensable que ce soit pour utiliser des périphériques internes ou un clavier etc.

Afin de respecter notre modèle d'attaquant, nous nous sommes donc concentrés sur les attaques de type «Local». Dans ce cas, l'attaquant doit se trouver face au système qu'il souhaite attaquer.

Cette partie sera donc dédiée à la présentation d'un certain nombre d'attaques de ce type et qui ont été réalisées sur des systèmes embarqués mobiles. Toutefois, cette partie ne constitue pas un examen exhaustif de toutes les attaques réalisées, mais permet de mettre en lumière les moyens d'attaque généralement mis en œuvre contre ces systèmes.

Dans les différents cas, l'attaquant a parfois dû installer des logiciels sur le système pour réaliser ses attaques. Cela peut s'apparenter à la mise en place de logiciels malveillants locaux sur le système, et cela est compatible avec le modèle d'attaquant que nous avons sélectionné.

Attaque par chronométrage de cache

L'attaque par chronométrage de cache (cache timing attack), décrite dans [MDR+16] consiste en un chronométrage des accès aux données dans les différents niveaux de mémoire, ceci afin d'en déduire des informations. Pour réaliser cette attaque, l'attaquant doit être capable d'installer une application malicieuse qui lui permet de chronométrer les différents accès. Dans leurs travaux, il est également précisé qu'aucune permission spécifique n'est requise (pas de nécessité d'avoir des accès privilégiés ou d'en être administrateur), et que les cibles principales sont les Central Processing Unit (CPU) Cortex-A53 et Cortex-A57 d'Advanced RISC Machine (ARM) ou les Krait 400 de Qualcomm.

Cette technique leur a permis, à partir des chronométrages, de retrouver les valeurs exactes de clefs cryptographiques, ainsi que les données du comportement d'un utilisateur comme les appuis sur l'écran, la cadence de glissement sur celui-ci, etc.

Nous reviendrons par la suite sur cette attaque spécifique.

Attaques par réutilisation de code

Les attaques par réutilisation de code sont un type d'attaques permettant de détourner de leur usage prévu, des portions de code présentes dans un système. Il en existe diverses formes chacune basée sur une méthode permettant d'atteindre la zone de code désiré et de l'exécuter.

Le code contenu dans un système, dispose d'un ordre d'exécution prévu par le programmeur, c'est le flot d'exécution. En modifiant ce flot, comme visible sur la Figure 1.3, sans modifier le contenu des exécutions, il est possible de prendre le contrôle du système.

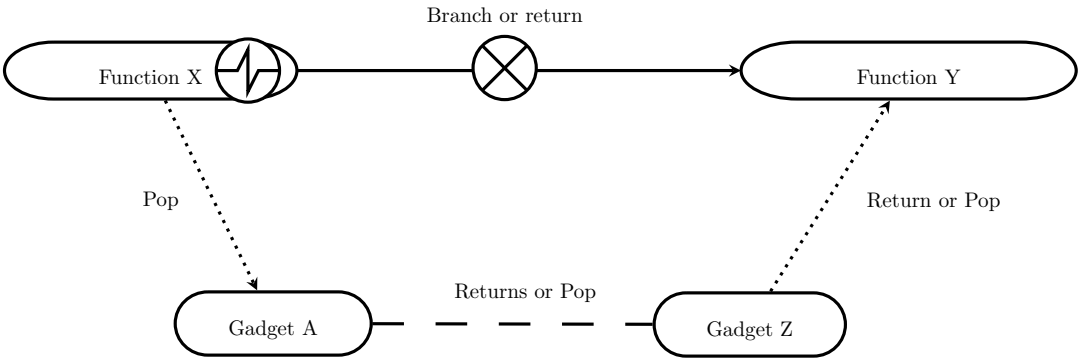

FIGURE 1.3 – Le flot normal est détourné, on ne passe plus de la fonction X à la fonction Y. Des portions de code, les gadgets, sont exécutées puis l'on retourne vers la fonction Y, retrouvant ainsi le flot d'exécution normal.

Contrairement aux autres attaques, celles-ci nécessitent un point d'entrée afin d'être mises en œuvre. Il n'est ainsi pas possible de les utiliser telles quelles.

Mettre en œuvre une de ces attaques nécessite donc un temps d'analyse par l'attaquant. Celui-ci devra alors utiliser ses connaissances sur l'architecture logicielle pour trouver les morceaux de codes utiles, et un moyen d'accéder à ces morceaux de code.

Dans cette configuration, une attaque par réutilisation devient alors une véritable menace. En premier lieu, parce que ces attaques sont basées sur le principe des «Weird Turing Machine». Selon ce principe, en exécutant des instructions correctement choisies, il est possible de recréer une machine de Turing universelle par-dessus le programme existant afin de réaliser toutes les fonctions voulues.

On note également, que ces attaques vont conserver le même niveau de privilège que le code original. Ce qui dans certains cas peut autoriser des comportements préjudiciables pour la sécurité du système.

Ainsi, un fragment de code pouvant accéder à toutes les zones mémoires, durant son exécution légitime, transférera cette capacité lorsque ce fragment sera réutilisé. Pouvant donc conduire à un endommagement du système.

Pour réutiliser du code, il est nécessaire de disposer d'un point d'entrée. Par la suite, la réutilisation du code se base sur trois éléments :

— Identifier les portions de code utiles pour le comportement que l'on souhaite mettre en place. Ces portions sont appelées les «gadgets». Ces gadgets sont des suites d'instructions qui effectuent le comportement voulu.

— Une instruction permettant de naviguer dans le code. Généralement des instructions permettant d'effectuer des sauts indirects dans celui-ci.

— Enfin, il faut se prémunir contre les effets de bord. Il s'agit d'effets indésirables sur le reste de l'exécution, ou sur le moment où le programme principal sera de nouveau exécuté. La réutilisation se faisant au détriment du flot d'exécution prévu par le concepteur, ces effets peuvent apparaitre. En choisissant des gadgets assez petits et correctement ciblés, il est possible d'éviter ou au moins de limiter ce phénomène.

```
add.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <main>:
    8000:       e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
    8004:       e28db000        add     fp, sp, #0
    8008:       e24dd014        sub     sp, sp, #20
    800c:       e3a03064        mov     r3, #100        ; 0x64
    8010:       e50b3008        str     r3, [fp, #-8]
    8014:       e3a03002        mov     r3, #2
    8018:       e50b3010        str     r3, [fp, #-16]
    801c:       e3a03000        mov     r3, #0
    8020:       e50b300c        str     r3, [fp, #-12]
    8024:       ea000007        b       8048 <main+0x48>
    8028:       e51b3010        ldr     r3, [fp, #-16]
    802c:       e51b200c        ldr     r2, [fp, #-12]
    8030:       e0823003        add     r3, r2, r3
    8034:       e50b300c        str     r3, [fp, #-12]
    8038:       e51b3010        ldr     r3, [fp, #-16]
    803c:       e51b2008        ldr     r2, [fp, #-8]
    8040:       e0633002        rsb     r3, r3, r2
    8044:       e50b3008        str     r3, [fp, #-8]
    8048:       e51b200c        ldr     r2, [fp, #-12]
    804c:       e51b3008        ldr     r3, [fp, #-8]
    8050:       e1520003        cmp     r2, r3
    8054:       daffffb3        ble     8028 <main+0x28>
    8058:       e3a03000        mov     r3, #0
    805c:       e1a00003        mov     r0, r3
    8060:       e24bd000        sub     sp, fp, #0
    8064:       e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
    8068:       e12fff1e        bx      lr
```

(Fonction d'addition et d'affichage.)

```
25|     8044:       e50b3008        str     r3, [fp, #-8]
26|     8048:       e51b200c        ldr     r2, [fp, #-12]
27|     804c:       e51b3008        ldr     r3, [fp, #-8]
28|     8050:       e1520003        cmp     r2, r3
29|     8054:       daffffb3        ble     8028 <main+0x28>
30|     8058:       e3a03000        mov     r3, #0
31|     805c:       e1a00003        mov     r0, r3
32|     8060:       e24bd000        sub     sp, fp, #0
33|     8064:       e49db004        pop     {fp}
34|     8068:       e12fff1e        bx      lr
35|
```

(Gadget)

FIGURE 1.4 – À gauche une fonction quelconque. À droite, fin de la fonction isolée où se trouve le gadget. En rouge, le gadget que l'on peut y trouver. Il permet de copier 0 dans le registre $r0$. En exécutant uniquement cette partie, ce sera le comportement obtenu.

Ainsi en disposant de gadgets et de moyens de naviguer dans le code, il devient possible de chainer les gadgets pour créer tous les comportements voulus à partir d'un point d'entrée que l'on aura obtenu au préalable.

Il existe divers types d'attaques par réutilisation de code, on les identifie généralement par leur type de gadget et par les instructions de navigation qu'elles utilisent :

— Return-into-libc (RILC) [Pes] : il s'agit de la première itération d'une attaque en réutilisation de code, elle était basée sur des sauts vers des fonctions de la bibliothèque de fonctions C.

— Return-Oriented Programming (ROP) [DDSW10] : il s'agit de la généralisation de l'attaque précédente, qui se base désormais sur n'importe quel type de gadget et plus uniquement des fonctions

de la bibliothèque C. Les gadgets ont ici la particularité de débuter par le comportement voulu (ou une petite partie dans le cas de gadgets chainés) par l'attaquant, et se termine par un appel à l'instruction `return`. C'est par ce biais qu'il devient possible de chainer les gadgets. Ainsi, on peut considérer que le pointeur de pile devient le pointeur du programme.

— Jump-Oriented Programming (JOP) [BJFL11] : basé sur le même modèle que le ROP, cette attaque étend le fonctionnement des attaques en réutilisation à d'autres instructions que le `return`. Ici, toutes les instructions de sauts deviennent des moyens de naviguer dans le code et est destiné aux systèmes ne disposant pas de cette instruction.

— Call Oriented Programming (COP) [CW14] : cette attaque est une réponse aux contremesures qui ont été mises en place contre les précédentes attaques en réutilisation. Il s'agit d'une adaptation de l'attaque JOP à des gadgets plus larges rendant la détection plus complexe.

Attaque par dépassement de tampon : «Buffer overflow»

L'attaque par dépassement de tampon est connue dans le monde de l'informatique depuis les débuts de cette discipline [Jel68].

Le buffer overflow est le résultat de la tentative de copie d'une donnée dans un espace mémoire trop petit pour pouvoir la recevoir. Cette donnée va donc venir écraser une partie des données se trouvant dans l'espace mémoire contiguë. La donnée va déborder du tampon qui lui est accordé.

Du fait de l'age de cette vulnérabilité, de nombreuses contremesures ont été mises en place. Cependant, certains programmes y sont toujours vulnérables.

Récemment plusieurs attaques de ce type ont été mise en œuvre sur des téléphones mobiles. On peut en citer deux, la première apparue sur le blog de laginimaineb [5], l'auteur de cette attaque. Elle permet une montée en privilège amenant à prendre le contrôle du système même dans le cas où il serait chiffré. Une chaine de modifications est nécessaire au préalable. L'auteur a ainsi dû examiner en profondeur le code source pour trouver des zones où le modifier. Cette vulnérabilité provient d'une faiblesse dans l'implémentation des options de sécurité sur certains processeurs produits par Qualcomm.

Le second cas, issu des travaux de Hay *et al.* [HD14] permet également de prendre le contrôle du système, mais cette fois-ci la vulnérabilité provient d'un élément vulnérable du système d'exploitation.

Attaque par porté dérobée : «Backdoor»

La «backdoor» ou porte dérobée consiste à placer dans le code un élément caché. Il peut s'agir d'un fragment de code n'ayant pas d'effet particulier lorsque le système s'exécute correctement mais qui en a un dans des conditions particulières connues uniquement de l'attaquant. Il peut également s'agir d'un fragment de code masqué dans des variables, donc non-exécuté en temps normal, mais qui le sera pour mener à bien l'attaque.

Cette attaque est différente des précédentes, le modèle d'attaquant s'éloigne ici du MATE. Le code malveillant étant déjà présent dans le système et placé lors de sa conception.

Récapitulatif

En récapitulant les attaques types déjà expérimentées sur les systèmes embarqués :

Attaque	Cible
Chronométrage des temps d'accès au cache	**IOA et DOA**
Réutilisation de code	**COA**
Dépassement de tampon	**COA**

5. http ://bits-please.blogspot.com/2015/08/exploring-qualcomms-trustzone.html

Ces types d'attaques respectent notre modèle d'attaquant et concernent donc des problèmes de sécurité sur lesquels nous avons voulu nous concentrer.

Nous avons donc recherché la présence éventuelle de solutions de sécurité permettant de traiter les problèmes identifiés. Nombre d'entre elles sont mises en place par les concepteurs de ces systèmes embarqués. Nous allons par la suite les présenter.

1.2.3 Implémentation des protections

Pour répondre aux attaques déjà testées sur des systèmes embarqués mobiles, les concepteurs ont mis en place diverses méthodes de protections. En particulier, ces méthodes cherchent à assurer de la protection contre les attaques orientées information (IOA), données (DOA) et prise de contrôle du système (COA). Nous allons ici faire une revue d'un certain nombre de ces techniques. Compte-tenu de l'évolution constante des systèmes et des attaques, cette revue ne sera pas exhaustive, mais nous permet d'identifier les vecteurs de protections préférés par les concepteurs.

L'intégrité de contrôle de flot :

Le Control Flow Integrity (CFI) introduit par Abadi *et al.* [ABEL09], est un mécanisme de défense contre les attaques en réutilisation de code, et les injections de code qui tentent de modifier le flot d'exécution normal d'un programme.

Le CFI fonctionne en deux phases :

— La première consiste à construire le Control Flow Graph (CFG), le flot d'exécution normal du programme. Il est obtenu à partir du code du programme ou du binaire de celui-ci.

— La seconde phase, qui a lieu durant l'exécution, va chercher à imposer de suivre le chemin d'exécution légitime contenu dans le CFG. En cas d'attaque et donc de détournement du CFG, comme c'est le cas lors des attaques par réutilisation de code, le CFI va détecter une violation du chemin légitime et une décision pourra être prise.

La précision du CFI dépend cependant de la précision de la première phase. Ainsi la stratégie de construction du graphe, influera sur la taille du graphe et donc sur le temps nécessaire au parcours de celui-ci. Un graphe précis et qui a donc une faible granularité, proche des instructions, est plus lent à parcourir mais offre une bonne réactivité de détection. À l'inverse, un graphe moins précis, avec une granularité plus proche des fonctions ou des sauts dans le code, est plus rapide à parcourir, mais peut ne pas détecter des modifications du flot dans certains cas, ou nécessite un certain temps avant d'être détecté.

Différentes implémentations de CFI ont été utilisées dans la littérature. Nous pouvons distinguer trois types :

— les sensibles au flux («flow-sensitive»), utilisent les informations de contrôle de flux d'un programme pour déterminer les valeurs possibles d'un pointeur.

— les insensibles au flux («flow-insensitive»), calculent un ensemble de valeurs valables pour toutes les entrées du programme.

— les sensibles au contexte («context-sensitive»), prennent en compte le contexte lors de l'analyse d'une fonction, empêchant les valeurs de se propager vers des chemins impraticables et garantissant ainsi que le contexte d'un appel reste indépendant des autres contextes d'appel.

La distribution aléatoire de l'espace d'adressage :

On peut également citer une autre technique de protection, la distribution aléatoire de l'espace d'adressage (ou Address Space Layout Randomization (ASLR)) [BDS03]. Cette technique permet de complexifier les attaques en réutilisation de code, ou en dépassement de tampon.

Le principe est de disposer de manière aléatoire en mémoire différentes portions de code sensibles. Plus particulièrement, certaines bibliothèques de fonctions systèmes vont se retrouver à des adresses différentes, de même que certaines variables, etc. Ceci retire à un attaquant la possibilité d'obtenir ces informations.

Dans le cas de la réutilisation de code, il est par exemple obligatoire pour un attaquant de savoir précisément où se situe la portion de code qu'il souhaite utiliser, car comme on l'a vu le but est d'obtenir l'effet escompté mais aussi de limiter les effets de bords.

L'ASLR modifie à chaque démarrage de la machine l'adresse à laquelle se trouve certaines bibliothèques du système, rendant difficile leur localisation et donc leur étude en vue d'une utilisation pour de la réutilisation.

De la même manière, le dépassement de tampon est rendu inefficace, car le tampon est disposé à des adresses différentes.

Prévention de l'exécution des données :

La technique de la prévention de l'exécution des données (ou Data Execution Prevention (DEP)) [AA04] est présente sur de nombreux systèmes d'exploitation, elle consiste à empêcher l'exécution de code lorsque celui-ci se trouve dans certaines pages mémoires. Implémenté de manière logicielle ou parfois de manière matérielle (c'est le cas du no-execute «NX» bit dans certaines machines), la protection peut s'appliquer à des zones vulnérables de l'espace adressable. Ainsi lors qu'il est correctement configuré le mécanisme de DEP permet d'empêcher l'exécution de code injecté ou présent dans la pile.

Surveillance du flot d'information :

Viennent ensuite les techniques d'Information Flow Tracking (IFT) qui permettent de suivre au cours de l'exécution du programme les informations qu'il manipule et celles qu'il génère.

L'intégrité du flot de données : La Data Flow Integrity (DFI) [SST07] est une technique de protection contre les DOA, elle permet d'assurer que le flux des données manipulées ne peut dévier d'un graphe de flux de données généré avant l'exécution du programme. C'est le pendant orienté données du CFI. Par exemple, une donnée provenant d'une chaine de caractères, ne doit pas pouvoir se retrouver en tant que valeur de retour contenue dans la pile d'exécution.

L'analyse dynamique de teinte : L'analyse dynamique de teinte (Dynamic Taint Analysis (DTA)), est une technique semblable à la DFI utilisée pour se prémunir des attaques de type DOA [NS05]. La principale différence ici, c'est que les données sont décrites avec un haut niveau d'abstraction, il peut s'agir de fichier, de paquets réseau, etc. Ceci rend la DTA habituellement plus gourmande en ressources que la DFI.

On retrouve aujourd'hui cette technique appliquée aux systèmes embarqués, tels que les téléphones mobiles fonctionnant avec le système d'exploitation Android [EGH+14].

L'intégrité du pointeur de code :

La Code Pointer Integrity (CPI) est une alternative au CFI proposée par Szekeres *et al.* [SPWS13], le but est de se prémunir contre la première phase du CFI, qui consiste à réécrire la valeur du pointeur de code, et ainsi exécuter différentes portions de code. Après chaque réécriture du pointeur de code, celui-ci est vérifié par le CFI, tandis que le CPI protège de la prise de contrôle de certaines données telles que les valeurs des pointeurs de code, en les séparant des données non-sensibles.

Il en résulte une séparation de l'espace adressable en une zone dite protégée et une autre normale.

La zone protégée contient ainsi toutes les données de gestion du code et de contrôle de fonctionnement du programme. Toute modification affectant des données dans cette zone est soumise à des vérifications utilisant des métadonnées, elles-mêmes sauvegardées dans cette zone.

Ainsi en limitant l'accès à des informations dangereuses par un éloignement spatial, le CPI offre une protection aux pointeurs de code.

L'intégrité du noyau et des applications :

La Kernel Integrity (KI) est une technique de protection du code du noyau, code principal du système d'exploitation d'un système embarqué. Cette technique compare lors de l'allumage du système l'empreinte (ou hash) du code du noyau [AFS97] avec une valeur de hash présente en mémoire. Cette comparaison assure qu'aucune modification n'a été effectuée au niveau du code de ce programme essentiel au bon fonctionnement.

La même technique peut être appliquée aux autres applications (on parle alors de Software Integrity (SoftI)). Il s'agit de signer des portions de code jugées sensibles, une vérification de la signature est à effectuer à chaque démarrage du programme (cas de la SoftI) ou du système d'exploitation (cas de la KI) en fonction de la politique voulue.

1.2.4 Implémentations

Ces différentes techniques ont été implémentées dans divers outils et systèmes d'exploitations (ou Operating System -OS-). On les retrouve dans de nombreux systèmes embarqués dès lors qu'ils utilisent ces systèmes d'exploitation. Du fait de l'utilisation massive d'Android dans la téléphonie mobile (plus de 70% des parts de marché des Operating System (OS) de smartphones), et d'ARM (plus de 90% des parts de marché des smartphones), nous nous sommes concentrés sur les solutions testées et ou compatibles avec ces éléments.

C-FLAT : Control-Flow Attestation for Embedded Systems Software [AAD+16]

Cet outil permet d'assurer l'exécution de programmes sur microcontrôleurs, tel qu'attendu par le concepteur, il assure ainsi les fonctions de CFI. Un programme appelé «Measurement Engine» est chargé de calculer un accumulateur d'authentification assurant que le programme suit bien son flot normal. À la fin, la valeur de l'accumulateur est signée puis transmise à un programme de vérification. La vérification est possible car l'authentification est accumulée dans une valeur utilisant une fonction bijective :

$$H_{courant} = H(H_{précédent}, \text{nœud}_{courant}) \quad . \tag{1.1}$$

Dans le cas où il y aurait plusieurs successeurs à une instruction, le précédent H est sauvegardé et $H_{précédent}$ prend la valeur 0. En outre une protection DEP est également disponible.

MoCFI [DDE+12]

MoCFI est un outil offrant une protection contre le CFI sur les systèmes utilisant le système d'exploitation `iOS`[6] (baladeurs iPod, smartphones iPhone et tablettes iPad). Cet outil répond à un problème spécifique de ces systèmes qui utilisent des processeurs ARM. Cet outil permet de générer un CFG et un fichier patch contenant les métadonnées des branches indirectes et des appels de fonctions de l'application. Deux programmes sont alors exécutés en parallèle. L'un constitue le programme à protéger et le second programme assure la CFI. Lors des appels de fonction, les adresses de retour des deux programmes, qui s'exécutent en parallèle sont comparés, s'il y a concordance alors le saut est validé sinon l'anomalie peut être détectée.

SOFIA [dCDKC+16]

Software and cOntrol Flow Integrity Architecture (SOFIA) est un outil de protection matérielle, conçu par De Clercq *et al.*, il permet d'assurer les fonctions de CFI et SoftI au travers de deux mécanismes.

Dans un premier temps, l'Instruction Set Randomization (ISR) permet d'assurer la CFI. Un CFG reprenant les chemins d'exécution possible est généré. Il permet de chiffrer les instructions en utilisant la formule suivante :

$$i' = E_k(PC_{précédent}||PC) \oplus i, \tag{1.2}$$

Avec E_k l'algorithme de chiffrement utilisé prenant k comme clef secrète en entrée, PC l'adresse de l'instruction courante exécutée, i la valeur de cette instruction (non encore chiffrée) et enfin i' la valeur de l'instruction une fois chiffrée.

La principale différence entre cet outil et les précédents est qu'il nécessite une modification au niveau de l'architecture matérielle, afin qu'une partie du processeur puisse déchiffrer les instructions, ainsi le processeur s'exécute par la suite de la même manière. Toutes les instructions pour s'exécuter ont besoin de connaitre leur prédécesseur, en effet seules deux valeurs de PC compatibles rendent possible le déchiffrement ($i = i' \oplus E_k(PC_{précédant}||PC)$).

Ensuite, l'intégrité du code est assuré par un second mécanisme. Les instructions sont divisées en groupe et lors de la compilation chaque groupe se retrouve accompagné d'un MAC (Message Authentication Code). Celui-ci sera vérifié afin d'assurer qu'il n'y a pas eu de modification du code.

Knox [Mob16a, Mob16b]

Knox est une solution conçue par Samsung et mise en place dans ses smartphones. Il s'agit d'une solution de protection logicielle assistée par le matériel. Il s'agit en réalité d'un système de gestion d'appareils mobiles (Mobile Device Management (MDM)) qui permet à une entité globale d'avoir un accès privilégié à des appareils inscrits sur son registre. Ce type d'outil a été mis en place dans le cadre de la politique du BYOD qui a été introduit plus tôt. L'entité principale est ici l'entreprise qui gère une flotte de divers appareils pouvant être utilisés par les employés en dehors du seul cadre de l'entreprise.

Du côté des appareils à administrer, de la même manière que la précédente proposition SOFIA, l'outil s'appuie sur des ajouts matériels, ici au nombre de deux, le premier est la présence de la technologie TrustZone d'ARM, le second est la présence d'un fusible électronique (*eFuse*). Par la suite, un logiciel utilisant ces deux ajouts est nécessaire et permet de communiquer avec l'entité centrale.

D'un côté, la technologie TrustZone permet la séparation du système en deux «mondes», l'un dit «sécurisé» et l'autre dit «normal», toute la partie logicielle qui communique avec l'entité est isolée du reste du système dans cette zone.

6. https://www.apple.com/fr/ios/

De l'autre côté l'*eFuse* fonctionne comme une variable unique, qui une fois qu'elle a été activée ne peut plus être désactivée.

Durant le processus de démarrage un mécanisme de KI est utilisé pour vérifier l'intégrité du code du système. En cas d'erreur lors de cette phase, l'*eFuse* va être activé. Et le programme de communication contenu dans la partie «sécurisée», la TrustZone based Integrity Management (TIMA) va envoyer un rapport d'anomalie à l'entité d'administration.

Lorsque l'anomalie est détectée, la politique de l'entité s'applique. Dans certains cas, il peut s'agir d'une simple remontée d'information. Dans d'autres cas, il peut s'agir de désactiver des fonctions sensibles comme la connexion sans fil non restreinte dans l'enceinte de l'entreprise, ou la connexion à certains services. De cette façon, un appareil vérolé ou pour lequel le propriétaire aura tenté de gagner des privilèges sans y être autorisé, se verra refuser l'accès à des ressources sensibles de l'entreprise.

Systèmes d'exploitation

De nombreux systèmes d'exploitation (OS) intègrent les différents mécanismes présentés ci-dessus. Ainsi ils n'ont pas tous été vus séparément, en particulier les deux leaders du marché des smartphones en 2018, Android et iOS qui exploitent les techniques historiques de DEP et d'ASLR. Dans le cas d'Android, la possibilité d'utiliser du KI est laissé à chaque constructeur tandis que pour iOS c'est une option mise en place dès les premiers modèles.

Cependant, les deux OS utilisent de la SoftI, avec une politique qui reste différente. Dans les deux cas, les applications doivent provenir du magasin d'application qu'offrent les OS, les applications se trouvant sur ses magasins, sont testées et les gestionnaires des OS s'assurent ainsi qu'elles ne sont pas dangereuses pour les utilisateurs. La différence vient du fait qu'Android laisse la possibilité aux utilisateurs d'installer des applications provenant de magasins d'applications tiers, dans ce cas aucune vérification n'est faite sur le comportement de l'application.

1.2.5 Récapitulatif

On peut récapituler les implémentations des techniques de protection, dans le tableau 1.1.

1.3 Problématique scientifique

De nombreuses attaques logicielles sont apparues, nous en avons vu quelques-unes. Elles ont obtenu depuis des réponses de la part des concepteurs. Les solutions vues plus tôt sont toutes logicielles, parfois assistées par des modifications matérielles.

Nous nous sommes donc intéressé à la manière dont ce logiciel était exécuté et si cela pouvait créer des vulnérabilités. En nous penchant sur les différents systèmes, nous en sommes arrivé à la décomposer l'architecture des systèmes embarqué.

Cette architecture fonctionne suivant plusieurs couches successives. Ce sont les niveaux d'abstraction. En partant du haut le plus visible pour l'utilisateur en allant vers le bas, visible généralement uniquement par le concepteur du système. En prenant bien en compte que les concepteurs des différents niveaux, ont une vue restreinte à leur niveau et à ceux limitrophes.

Chaque couche pour fonctionner utilise des services simplifiés fournis par la couche inférieure. C'est l'abstraction.

Ainsi, un concepteur de niveau `X` recevra une description des interfaces de communication avec le niveau `X-1`, il n'en connaitra pas souvent le fonctionnement exact, mais en aura une idée par les spécifications fournies, le concepteur de niveau `X-1`, s'engage à lui fournir le service tel que précisé dans les

Table 1.1

Outil	Implémentation	Type d'attaque visé	Limites
Android & iOS	Système d'exploitation	DEP, ASLR	-
C-FLAT [AAD+16]	Implémentation matérielle (uniquement appareil possédant la TrustZone), décisions au niveau logiciel	CFI, DEP	-
SOFIA [dCDKC+16]	Hardware	CFI, KI	— Instructions à plusieurs prédécesseurs. — Gestion des clefs cryptographiques — Authentification des sonnées de manière dynamique.
Samsung Knox [Mob16a, Mob16b]	Protection logicielle assistée par le matériel	Intégrité du noyau CFI (par le biais de logs)	-

spécifications. Les niveaux d'abstraction apportent donc une chaine de confiance entre les éléments de plus haut niveau et ceux de plus bas niveau.

Dans la Figure 1.5 on commence ainsi par les programmes qui sont les moyens d'interagir directement avec le système, dans le cas de smartphones, il s'agit des applications, ces programmes pour fonctionner utilisent des services fournis pas l'OS ou par des pilotes, ceux-ci utilisent un jeu d'instruction pour fonctionner.

Pour fonctionner un logiciel nécessite donc un matériel sur lequel il s'appuie et qui lui rend divers services, ici visible dans la partie microarchitecture.

Sur la Figure 1.6, il s'agit du schéma de fonctionnement générique d'un processeur.

Dans la Figure 1.6, on voit que l'information part des registres pour rejoindre l'unité arithmétique et logique (Arithmetic-Logic Unit (ALU)), c'est cette partie qui est chargée des calculs, dans certains cas comme dans les cas de chargement de la mémoire, ou de lecture de celle-ci, d'autres fonctions peuvent être exécutées par cette partie.

Une fois le travail de cette partie terminée, la donnée résultante retourne dans un registre pour être utilisée une autre fois ou bien est envoyée dans la mémoire pour être utilisée plus tard.

Le schéma de transfert des informations est dépendant de l'instruction à exécuter. Celle-ci, spécifie le type d'opération à effectuer, de registre, si l'ALU sera ou non exploité, etc.

Les informations transitent en binaire dans les bus reliant tous les éléments du processeur entre eux et alimentant ainsi le processeur pour lui permettre d'exécuter les applications. Les bus et les registres partagent généralement une même largeur, fixée par le nombre de bits qu'ils peuvent recevoir à la fois, cette valeur varie généralement de 8 à 64, selon le mode opératoire du processeur et sa puissance de calcul. Cette partie est figée à la conception du système et tout le fonctionnement de celui-ci y compris les fonctions de sécurité reposent sur lui. C'est dans cette partie que l'information est réellement manipulée, toutes les strates supérieures, interprètent ces mouvements pour les rendre compréhensibles pour les autres machines ou pour les utilisateurs. L'information voyage ici sous forme de signaux électrique, traversant des portes logiques, des transistors, etc.

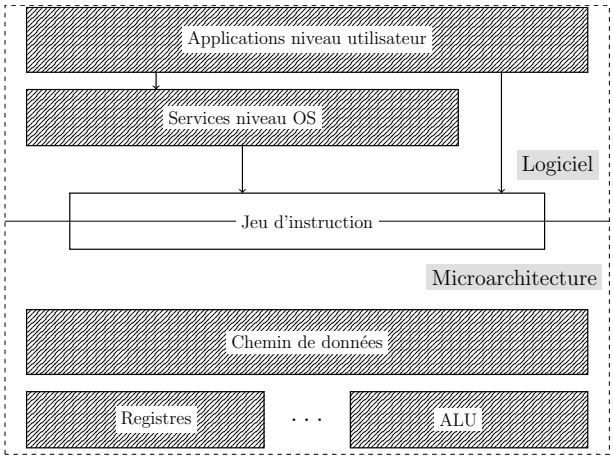

FIGURE 1.5 – Niveaux d'abstraction. La partie logicielle est celle visible par l'utilisateur.

Compte-tenue de l'importance des couches les plus basses du système dans la chaine de confiance, nous nous sommes donc concentrés dessus. Les protections présentes dans la littérature, étant restreintes au logiciel et parfois à quelques modifications matérielles, nous avons voulu savoir si cette base, pouvait être la cause d'une vulnérabilité.

Nous nous sommes donc interrogé sur l'effet de la microarchitecture sur la sécurité des systèmes et les techniques de sécurité mises en place. Et savoir, s'il était possible d'exploiter cette microarchitecture pour mettre en place des attaques de type COA, DOA ou IOA.

1.4 Attaques physiques

Avec l'idée d'évaluer la microarchitecture, nous nous sommes donc tournés vers les attaques physiques.

Les circuits électroniques sont composés de transistors. Lorsqu'ils manipulent des informations, plusieurs phénomènes physiques vont apparaitre en particulier de la chaleur et des émissions électromagnétiques. Ainsi, les interactions avec ces phénomènes physiques, deviennent des interactions avec le système lui-même. Ce sont les attaques physiques.

Il en existe deux familles principales, les attaques par observation (en écoute) et les attaques par perturbation (en faute). En anglais, elles sont regroupées sous le terme Side-Channel Attack (SCA). Dans la suite de nos travaux, nous nous efforcerons d'utiliser le terme SCA pour les attaques par observation et Fault Injection (FI) pour les attaques par perturbation.

1.4.1 Attaques par observation

Les attaques en écoute (par observation ou SCA) sont des attaques où les interactions du système avec différentes valeurs physiques sont observées. Ceci, dans le but d'inférer des informations sur le comportement interne du système.

L'observation se fait par différents canaux physiques (mesure de la consommation de courant, des émissions électromagnétiques, etc.).

On peut distinguer deux types de SCA :

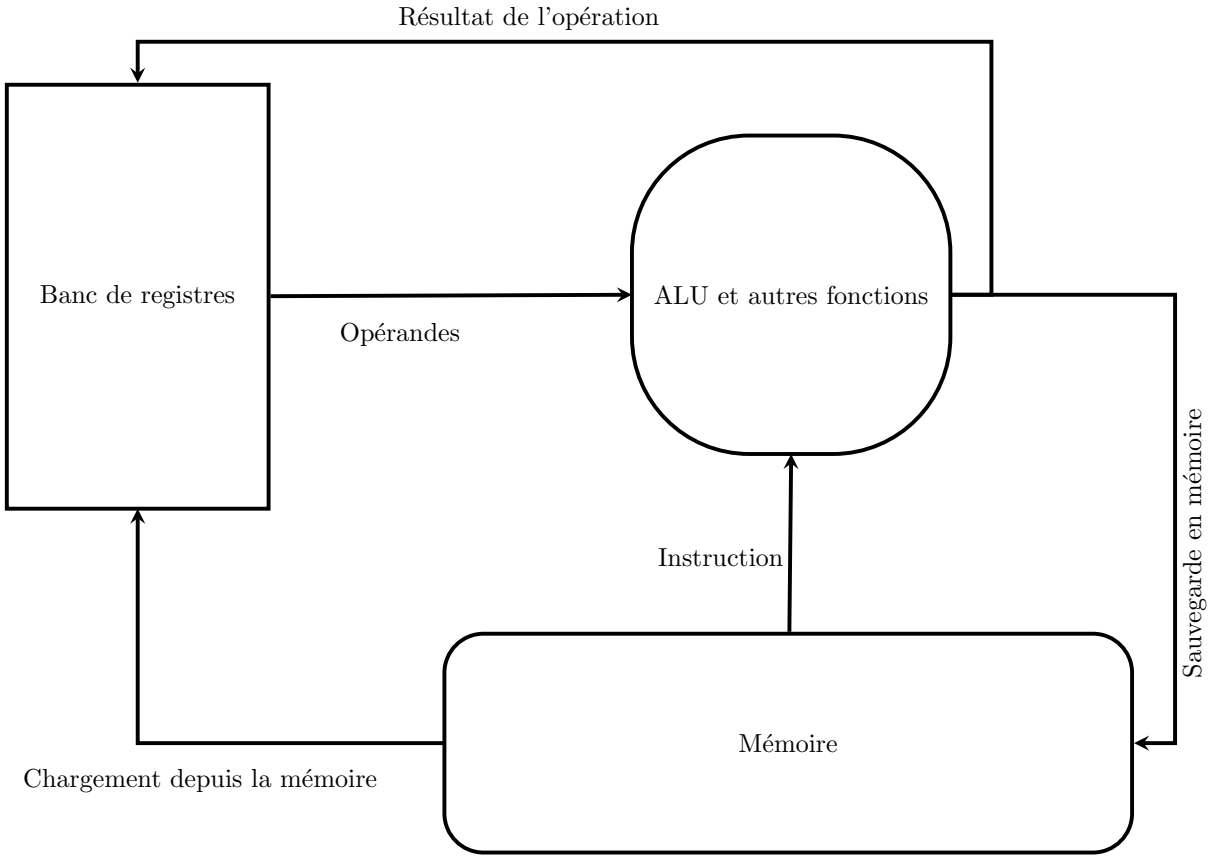

FIGURE 1.6 – Schéma de fonctionnement simplifié d'un processeur

Observation de canaux auxiliaires :

Dans cette classe d'attaque, il est possible de retrouver de l'information par des canaux qui ne sont, normalement, pas prévu à cet effet. Ce qui est ici exploité est la dépendance entre certaines valeurs physiques et l'état dans lequel se trouve le système à un moment donné. Par exemple, l'analyse de consommation de courant nous renseigne sur les périodes de temps où le système a davantage besoin d'énergie et donc où l'on suppose qu'il traite une grande quantité d'informations. On utilise donc un canal auxiliaire au canal de traitement de l'information pour connaitre le fonctionnement de celui-ci. Différents types de canaux existent parmi lesquels on retrouve le temps d'exécution [FKK10, Koc96], la consommation d'énergie du système [MOP08] ou encore les émissions électromagnétiques [QS01]. L'information extraite au travers de ces canaux est appelée la «fuite».

Ces différents canaux permettent des fuites et il existe plusieurs méthodes pour en extraire de l'information : c'est le modèle de fuite. On retrouve donc :

— La Simple Power Analysis (SPA) [Man02] où la fuite donne immédiatement l'information recherchée.

— L'utilisation de modèles mathématiques pour modéliser la fuite et des outils statistiques afin d'en extraire l'information. C'est le cas par exemple pour la Differential Power Analysis (DPA) [Pau99] ou la Correlation Power Analysis (CPA) [BCO04]. Dans ces méthodes, la différence vient du choix de l'outil statistique qui permet de retrouver l'information et du modèle mathématique utilisé pour caractériser la fuite.

— L'utilisation du profilage, que l'on retrouve dans les attaques dites «Templates». Basé sur de la

classification statistique à la place des modèles mathématiques des techniques précédentes. Cette méthode comporte néanmoins certaines limitations puisqu'une phase de caractérisation est nécessaire. Celle-ci requiert un contrôle complet sur un système du même type que celui qui sera attaqué, afin de classifier le maximum de caractéristiques de la fuite. Cependant, cette méthode est reconnue comme étant la plus efficace du point de vue de la théorie de l'information [CRR03] et diverses attaques l'ont déjà utilisée [AEPSQ, LBBC+16].

L'efficacité des attaques par canaux auxiliaires a été démontrée contre de nombreux algorithmes cryptographiques tel que Data Encryption Standard (DES) [BCO04, Pau99], Advanced Encryption Standard (AES) [GBTP08, Man02], le chiffrement RSA [GPPT15], Elliptic Curve Cryptography (ECC) [Osw02, GPP+16] mais aussi d'autres algorithmes [EFGT17]. De plus, certaines applications comme la vérification de Personal Identify Number (PIN) [LBBC+16] ont montré une vulnérabilité face à ces attaques, leur praticité est aussi démontrée pour des applications de rétro-ingénierie [Cla07].

Observation de canaux cachés :

Les attaques par canaux cachés quant à elles se rapportent à l'utilisation de canaux non répertoriés. Dans ce cas, le canal est prévu pour laisser passer des informations, mais il n'est normalement pas accessible à l'utilisateur final.

Dans les canaux cachés habituels, on retrouve généralement les ports de diagnostic comme le Test Access Port (TAP) ou les portes dérobées «Backdoors».

1.4.2 Attaques par perturbation

Le domaine des injections de faute (FI en anglais) est étudié depuis de nombreuses années, les premières recherches datant des années 60-70 [HS67, Arm72, UBW72, MC78].

Ces premiers travaux, proviennent de la nécessité de concevoir des systèmes soumis à des environnements particuliers (domaine aérospatial, nucléaire, ou militaire).

Les systèmes en question devaient pouvoir résister ou être tolérant à la présence de fautes pour assurer une sûreté de fonctionnement, même dans des environnements hostiles.

Les FI étaient donc un moyen de tester les systèmes avant leur utilisation en les soumettant à des environnements comparables afin de vérifier la compatibilité. Pour cela, lors de l'apparition des fautes dans le système, celui-ci était analysé afin de valider ou non le fonctionnement.

Plus récemment, des systèmes sécurisé ont eu à faire face à des attaques de même nature, visant à modifier leurs paramètres physiques afin de perturber leur fonctionnement. Le but étant d'affecter le bon fonctionnement des primitives de sécurité que ces systèmes emploient. Ceci, afin de rendre les primitives de sécurité inopérantes ou d'obtenir des informations sur leur fonctionnement.

Les FI sont aujourd'hui testées contre des applications de sécurité pour assurer de leur fonctionnement ou de l'absence de fuite, même dans les conditions d'apparition d'une faute. En particulier, les fautes sont utilisées dans le domaine de la cryptanalyse pour tester les nouveaux algorithmes cryptographiques [JT12, LFG13]. Les FI sont passés d'outils de test de la sûreté de fonctionnement («dependability» [ALR01]), qui se caractérise par à outil de test de sécurité.

Il existe diverses techniques d'injection de faute [KDN14], qui seront détaillées par la suite, mais toutes tendent à résoudre les mêmes défis :

— Prouver la possibilité ou non d'injecter des fautes dans le système.

— Observer les effets de la faute induite sur le système.

— Mesurer le niveau d'intrusion nécessaire à l'injection.

— Explorer un maximum de possibilités de fautes.

L'analyse d'une injection de faute passe par la détermination du modèle de faute, celui-ci permet d'expliquer quelle perturbation est générée par une faute. Chaque comportement étant spécifique à un système en particulier, il est difficile de savoir à l'avance quel modèle s'appliquera pour quelle situation. Parmi les modèles génériques on retrouve :

— Bit-flip : modification de la valeur d'un (ou plusieurs) bit. Celui-ci peut alors prendre la valeur 0 ou la valeur 1. Cette modification est temporaire.

— Stuck-at : la valeur d'un (ou plusieurs) bit dans le système se retrouve fixée à 0 ou à 1 pendant tout le fonctionnement du système.

— Bridging : création d'un pont entre plusieurs composants du circuit, ainsi des valeurs ne devant pas être échangées vont l'être. Un (ou plusieurs) bit peut alors passer d'un point à un autre.

Ces modèles permettent de modéliser diverses modifications à plus haut niveau, modification d'une variable manipulée, modification d'une instruction exécutée etc.

De nombreuses techniques d'injections ont été développées. Elles permettent de tester de manière plus fine les systèmes et de tenter de répondre aux défis énoncés plus haut, plus particulièrement, lorsque les FI sont utilisées dans le domaine de la sécurité de l'information. En effet, dans cette configuration, l'attaquant sait quelle information il recherche, qu'il s'agisse de rendre inopérant un algorithme cryptographique ou de récupérer des données sensibles (clefs, identifiants, etc.). Des protections existent contre ces types d'attaques [BBKN12], mais ne sont opérantes en grande majorité qu'au niveau logiciel. Par conséquent, ces solutions ont confiance dans le matériel pour assurer un fonctionnement légitime tel que prévu par le programmeur.

Nous allons ici faire une présentation des méthodes d'injections et une revue de quelques outils qui les utilisent. Cette revue n'entend pas être exhaustive, le nombre d'outils étant en constante augmentation. Le but ici est de montrer les avantages et inconvénients conceptuels des différentes méthodes.

Injection matérielle :

L'injection matérielle, Hardware-Based Fault Injection (HBFI), est la méthode qui consiste à modifier les paramètres physiques et, ou environnementaux du système directement en utilisant des perturbations électromagnétiques (Electromagnetic Fault Injection (EMFI)) [MDH+14], des lasers [BWK+87], ou en provoquant des fluctuations de tension électrique dans le système [TSW16, TMA11], etc. Le principal avantage de cette technique est que l'on attaque un vrai système dans sa version définitive, mais c'est aussi un de ces principaux inconvénients. Effectivement, pour que l'attaque soit réellement efficace, il faut déjà avoir en sa possession le système finalisé. Ce qui implique que des concepts ou prototypes ne peuvent pas être évalués.

Les outils HBFI, ont comme contrainte principale, la répétabilité et la difficulté de contrôle sur les caractéristiques exactes des tests. C'est plus particulièrement vrai lorsque l'on prend en compte les outils basés sur des conditions d'environnement précises (température, pression, humidité, ondes électromagnétiques par exemple). Dans ce cas, on note également un surcoût lié au matériel prérequis.

Certains outils HBFI vont donc tenter d'émuler les effets des modifications physiques sur le système, cela étant plus stable, et reproductible. Dans [AAA+90], les fautes sont induites en modifiant ainsi certains pins d'entrées pour les relier entre eux.

Fault Injection system for Study of Transient fault effects (FIST) [KLD+94], un autre outil utilise quant à lui, des émissions de photons et des modifications de l'intensité de courant délivrée au système en vue de le perturber. Les fautes ainsi crées peuvent causer un ou plusieurs bit-flips, les valeurs binaires se retrouvent modifiées, dans des régions aléatoires sur la puce. Des fautes non reproductibles en modifiant les pins d'entrée apparaissent alors.

Évidemment, les outils HBFI sont dépendant de la configuration utilisée et du système testé. En outre, ces injecteurs peuvent être plutôt complexes à mettre en place.

Simulation d'injection :

Les techniques de simulations d'injection (Simulation-Based Fault Injection (SiBFI)) permettent de simuler l'effet réel d'une faute sur la description matérielle du système attaqué. Ces techniques utilisent généralement des modélisations haut niveau des circuits, dans des langages Hardware Description Language (HDL). Ces langages modélisent le comportement et la structure des circuits, on y retrouve donc en particulier les transferts au niveau des registres (Register Transfer Level (RTL)). Les fautes sont injectées directement dans ces modèles simulés. Ainsi, on observe plus finement l'effet de la faute. Certaines techniques modifient les circuits directement, d'autres modifient les variables qui transitent dans ces circuits. Ainsi, il est possible de simuler différents comportements en fonction du modèle de faute choisi.

Le principal désavantage des techniques SiBFI est leur lenteur d'exécution, la simulation des signaux et des transferts de données (RTL) d'un circuit est plus lente que la vitesse de fonctionnement du circuit en lui-même. Il est à noter également que les SiBFI, même lorsqu'il s'agit de circuits simples ne sont pas capable de simuler l'effet de fautes à des niveaux beaucoup plus élevés. En effet, la simulation est par nature plus lente que l'exécution immédiate, et suivre la propagation d'une faute va nécessiter une simulation d'autant plus longue que l'on souhaite obtenir une information à plus haut niveau d'abstraction.

Cependant, ces techniques ont pour avantage d'être peu couteuses. D'une part, le système peut ne pas être finalisé. Ensuite, il ne sera pas endommagé, s'il est déjà en cours d'utilisation. Et enfin, même si le temps d'analyse est plus long, la mise en place de l'expérimentation est moins chronophage et requiert moins de compétences spécifiques en comparaison avec les méthodes matérielles.

VERIFY [STB97] (VHDL-based Evaluation of Reliability by Injection Faults Efficiently) est un outil utilisant une extension d'un langage HDL le Very high speed integrated circuit Hardware Description Language (VHDL), afin de décrire les fautes sur des composants. Avec cet outil, les concepteurs de matériel peuvent décrire les modèles de fautes à tester, ainsi ils obtiennent le comportement de leur système face à ces fautes. Le modèle d'analyse est la comparaison entre une exécution sur un modèle «gold» qui représente le fonctionnement optimal ou normal du système, en spécifiant des emplacements critiques : les «checkpoints». Les concepteurs ont ainsi des moyens de comparer et de mieux comprendre l'effet d'une faute.

FAUMachine[7] est un outil de simulation de système. Il a été utilisé dans divers travaux afin de simuler des injections de fautes dans des systèmes [PSC07, SPS09]. Sa particularité est qu'il permet de simuler divers types de fautes et dans divers périphériques connectés au système, tout en rendant possible l'observation des effets sur le fonctionnement total du système et ce même sur le long terme.

LIFTING [BN08] se rapproche de VERIFY, en rendant possible la simulation de circuit modélisé en Verilog, un autre langage HDL.

Émulation d'injection :

L'émulation de faute Emulation-based Fault Injection (EBFI), exploite la capacité des Field-Programmable Gate Array (FPGA) à pouvoir émuler la description RTL d'un grand nombre de circuits. Le prototypage matériel ainsi réalisé permet d'être une alternative nécessitant moins de temps d'exécution en comparaison aux techniques de SiBFI [CMR+01, Lev00].

Un avantage de cette technique est qu'elle permet de tester le système dans un environnement complet d'utilisation, avec des interactions en temps réel avec d'autres systèmes par exemple etc. Cependant, contrairement à la SiBFI le système doit être finalisé et fonctionnel, le circuit complet étant émulé. De

7. http://www3.informatik.uni-erlangen.de/EN/Research/FAUmachine/description.shtml

plus des mécanismes supplémentaires sont à prévoir afin de spécifier le moment et l'endroit de l'injection de la faute. En cas de fautes multiples, celles-ci peuvent apparaitre à différentes localisations, la complexité des mécanismes va donc s'accroitre.

Antoni *et al.* [ALF02] ont proposé un outil permettant d'injecter une faute dans des cellules particulières d'un FPGA en utilisant la capacité de ces derniers à être reconfigurés durant leur fonctionnement. Ceci permet d'éviter de recourir à un circuit de contrôle complexe pour l'injection. Cependant, le temps de reconfiguration doit être également pris en compte et il peut s'avérer important en comparaison de la durée d'exécution du test d'injection, limitant ainsi le bénéfice de l'émulation par rapport à la simulation.

Civera *et al.* [CMR+01] ont quant à eux proposé un outil permettant d'assurer un contrôle plus flexible de l'injection, pour cela c'est la description HDL du circuit qui est modifié, en utilisant directement des portions du circuit contenant le modèle de faute. Une variable de contrôle permet alors d'activer ou non ces portions, générant ainsi le modèle de faute.

Injections implémentées dans le logiciel :

L'objectif des outils d'injections implémentées dans le logiciel, Software implemented fault injection (SWIFI), est de reproduire au niveau logiciel les modifications se déroulant au niveau matériel. Leur usage principal est la détection de vulnérabilités en accord avec les fautes au niveau matériel. Les outils SWIFI utilisent donc un niveau d'abstraction logiciel du modèle de faute lorsque le logiciel est exécuté ou modifie le programme avant l'exécution de celui-ci. Ainsi il est possible de tester des systèmes complets y compris le système d'exploitation ce qui rend cette technique populaire puisque simple à mettre en œuvre.

Les modèles de faute généralement pris en comptent par les techniques SWIFI sont des projections de l'effet de la faute au niveau du jeu d'instruction et se traduisent généralement par :

— le saut d'instruction (une ou plusieurs instructions ne sont pas exécutées),

— la modification d'instruction (une ou plusieurs instructions sont modifiées).

Suivant ces modèles, on peut séparer les outils de SWIFI en deux catégories, les «pre-runtime» où les fautes sont injectées dans le logiciel avant qu'il s'exécute [HSR95], et les «runtime» où les fautes sont injectées durant l'exécution [CMS+98, KKA95].

Dans les deux cas, des modifications par rapport au logiciel, lorsqu'il s'exécutera de manière nominale, sont notable et peuvent ajouter des biais à l'analyse. On note ainsi la modification du flux d'exécution, la modification du comportement de certains composants, ou enfin les variations temporelles qu'induisent ces techniques. C'est le cas par exemple pour FERRARI [KKA95] ou EFS [RBLC15] qui nécessitent d'exécuter deux programmes, l'outil d'injection et le programme à tester. Certaines des caractéristiques du fonctionnement, en particulier les temps d'exécution et donc les moments d'injection varient par rapport à une attaque dans le monde réel, où seul le programme a attaqué sera exécuté.

Une autre des limitations de ces outils concerne la représentation et l'analyse des états possibles. Ils offrent généralement plus de possibilités de faute que les autres techniques, cependant nombre d'entre elles ne sont pas traduisibles par des fautes réelles. D'un autre côté, les modèles logiciels pourront ne pas être en mesure de tenir compte de tous les comportements n'ayant pas directement un effet visible sur le jeu d'instruction ou le programme testé. De cette manière, la modification d'une instruction peut ne pas avoir d'effet sur le programme en cours de test, mais peut se répercuter sur un autre programme, en ne testant que le dit programme, il n'est pas possible de constater cet effet.

Le principal défi consiste soit à ne générer qu'un ensemble minimal de fautes (celles qui peuvent conduire à une corruption silencieuse des données), soit à les éliminer pendant qu'elles sont générées. Ceci conduit à plusieurs phases d'optimisation au cours de la simulation et reste un défi difficile à relever.

1.5 Déroulement des travaux

Nous avons vu que de nombreuses attaques avaient déjà été effectuées sur des systèmes embarqués, cependant la plupart de ces attaques ont aujourd'hui eu des réponses. Nous nous sommes interrogé sur le périmètre que ces réponses couvraient. Plus précisément, nous nous sommes concentrés sur la matérialisation physique du système, la microarchitecture.

En effet, il est possible d'affecter et d'observer des changements d'états de la microarchitecture par le biais d'attaques physiques. Par conséquent, la partie en rouge sur l'image 1.7 sera affectée ou pourra exhiber son comportement.

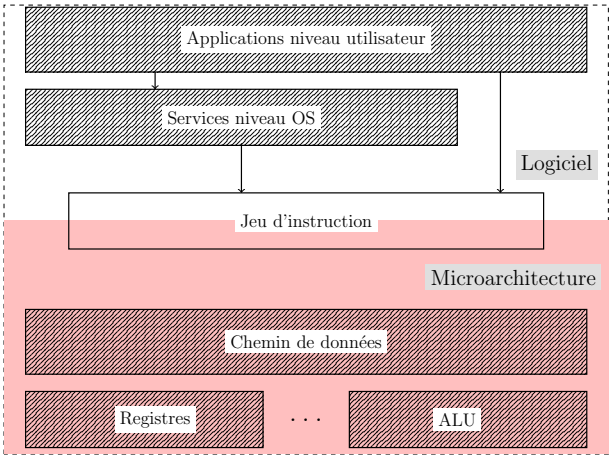

FIGURE 1.7 – Niveaux d'abstraction, en rouge la microarchitecture, affectée directement par les attaques physiques. Les services qu'elle rend aux couches supérieures sont affectés. Nous nous sommes demandé dans quelle mesure et comment cela pouvait-il se traduire sur les couches supérieures.

L'essentiel de nos travaux se concentre sur deux axes :

1. La manière dont le reste du système va se comporter en cas de perturbation, en nous interrogeant sur la manière dont les sécurités telles qu'elles existent aujourd'hui permettent d'assurer la chaine de confiance. En résumé quel est l'effet d'une perturbation sur la microarchitecture ?

2. La manière dont il est possible d'obtenir des informations sur un système en observant sa microarchitecture. Nous nous sommes interrogés cette fois-ci sur les capacités de détection et de préservation contre les fuites d'informations qu proposent ces protections. En résumé, quel est l'effet de la couche logicielle sur la microarchitecture ?

Nous avons cependant vu que des travaux sur les attaques physiques existaient déjà, néanmoins ces techniques étaient concentrées sur les algorithmes de cryptographie. Or comme nous le montre le schéma 1.7, tout le logiciel repose sur la microarchitecture. Nous avons voulu étendre le domaine d'application de ces attaques physiques à n'importe quel type de logiciel. Par conséquent, nous avons d'une part testé des algorithmes de chiffrement tel que l'AES, mais également mis à l'épreuve divers types de programme en nous concentrons sur l'effet global de la microarchitecture.

Enfin, les précédents travaux se concentraient sur des architectures relativement simples, les microcontrôleurs. Afin de couvrir un plus large échantillon de systèmes embarqués, nous nous sommes également intéressé à leur pendant plus puissant, les SoC.

1.5.1 Choix des techniques et défis

Nous nous sommes concentrés sur les attaques physiques. De par leur nature non-intrusive, les attaques par observation permettent uniquement de se concentrer sur les attaques de type DOA ou IOA. À l'inverse, les attaques en perturbation sont quant à elles adaptables à tous les types d'attaques puisqu'il est possible d'en observer l'effet pour en déduire de l'information et, ou prendre le contrôle du système.

Attaques par observation

On a vu qu'il existait deux classes d'attaques par observation, afin de respecter le modèle d'attaquant introduit plus tôt, les canaux cachés ont été ignorés. En effet, il s'agit de canaux masqué par les concepteurs ou les constructeurs, ainsi pour y accéder des modifications matérielles intrusives peuvent être nécessaires.

Sur les systèmes disposant de pin de connexions (utilisant généralement le protocole Join Test Action Group (JTAG)) pour des sondes de diagnostic, un attaquant va pouvoir les utiliser pour obtenir des accès privilégiés. C'est le cas de nombreuses attaques [RK10, VL18, GBBF15]. Cependant, l'accès à ces ports nécessite parfois une modification du système. On peut également ajouter, que du fait de la miniaturisation et comme contre-mesure, les fabricants rendent ces ports de moins en moins accessibles voir inutilisables.

Pour ces raisons, nous avons voulu prendre une orientation différente. Nous nous sommes concentrés sur les attaques par canaux auxiliaires ne nécessitant pas de modification au préalable. Avec la volonté également de rester proche d'une attaque reproductible en dehors de notre configuration de test en laboratoire.

Pour cela, nous nous sommes tournés vers l'observation des émissions électromagnétiques. Ce comportement est présent sur tous les systèmes électroniques et résulte du courant électrique qui traverse le circuit.

Attaques par perturbation

De la même manière, nous avons eu à faire des choix concernant les attaques en perturbation.

TABLE 1.2 – Récapitulatif des techniques d'injections

Technique	Compatibilité avec modèle d'attaquant	Modèle de faute	Avantage principal	Contrainte principale
HBFI	+++	Inconnu	Attaque représentant un cas réel	Difficulté expérimentale
SiBFI	—	Connu	Proche du réel, au niveau du circuit	Modèle HDL fonctionnel requis
EBFI	—	Connu	Idem que pour la SiBFI, mais avec des temps d'exécution raccourcis	Nécessité de disposer du modèle HDL complet et totalement fonctionnel.
SWIFI	+++	Connu	Mise en place rapide des expérimentations	Couverture réduite

D'une part le choix de la méthode de test s'est portée sur les injections matérielles.

Comme reporté sur le tableau 1.2, deux des techniques d'injections requièrent des informations sensibles inaccessibles pour les utilisateurs finaux. Ainsi, notre modèle d'attaquant est considéré comme ne pouvant pas avoir accès à ces informations, nous avons donc mis de côté ces techniques, le SiBFI et le EBFI.

Ensuite pour départager les deux autres techniques, nous nous sommes interrogés sur les avantages et inconvénients de chaque technique. Le propos qui suit, dans cette sous-section, est issu de la publication de ces tests sous le titre «When fault injections collides with hardware security» [BCLL18]. Le but de nos travaux est de mettre en lumière les potentiels problèmes de sécurité que l'on retrouve dans les systèmes embarqués.

En premier lieu, nous avons pris en compte les caractéristiques nécessaires pour une technique d'injection de fautes nous semblant satisfaisante :

— Représenter de la manière la plus précise possible une faute se déroulant sur un système embarqué. Celle qu'un attaquant ferait sur le système.

— L'injection de faute se fait du point de vue de l'attaquant, en respectant le modèle MATE que nous avons choisi.

— Le modèle de faute n'est pas connu à l'avance.

D'une part les méthodes SWIFI permettant d'abstraire la faute au niveau logiciel, la faute devient un comportement modifiant le fonctionnement du jeu d'instruction. Cela en fait une technique pertinente pour les concepteurs de logiciels. Néanmoins, notre positionnement au niveau de la microarchitecture, nous empêche de prendre uniquement en compte cette abstraction.

D'autre part, les techniques SWIFI se basent sur des modèles de faute connus et admis dans la littérature. Notre propos est que ces modèles sont en constantes évolution et correspondent à des systèmes particuliers. En effet, nous avons vu que les systèmes ne cessent de se complexifier, que ce soit par l'augmentation du nombre de transistors ou de sous-systèmes dédiés à de nouvelles fonctionnalités. Nous arguons donc que l'abstraction faite actuellement au niveau du jeu d'instructions selon des modèles de faute connus n'est plus suffisante pour évaluer un système matériel.

Elle reste cependant pertinente pour l'évaluation d'un logiciel qui s'exécutera sur un système pour lequel son modèle de faute aura au préalablement été testé avec des techniques HBFI.

Pour ces raisons, nous nous sommes donc tournés vers des injections HBFI électromagnétiques.

Défis

Après avoir fixé les techniques d'attaques que nous avons souhaité utiliser, nous nous sommes intéressé aux différents défis auxquels nous avons eu à faire face.

Compréhension du modèle de fuite : Tout d'abord, identifier le modèle de fuite lors de l'écoute électromagnétique est un défi que nous avons à relever. Afin de comprendre ce qu'il est possible de retrouver comme information par le biais de cette attaque.

Compréhension du modèle de faute : Sans connaissance sur l'architecture interne, et avec l'augmentation de la complexité, la compréhension du modèle de faute s'annonce être un des défis les plus complexes auquel nous devrons faire face.

Synchronisation avec la cible : Un autre défi est celui de la synchronisation avec la cible. Les attaques physiques que nous avons sélectionnées sont effectives durant l'exécution du système. Un attaquant doit donc pouvoir se synchroniser avec lui pour deux raisons. La première, il doit être capable de «suivre» la vitesse de fonctionnement du système. Ensuite, il doit être capable de savoir à quel moment attaquer.

1.5.2 Équipements du laboratoire

Ces travaux ont été menés au Laboratoire de Haute Sécurité (LHS) de Rennes (voir Figure 1.8), qui est situé dans les locaux de l'INRIA (Institut National de Recherche en Informatique et Automatique).

Le LHS dispose de divers équipements permettant de mener à bien plusieurs types d'attaques physiques.

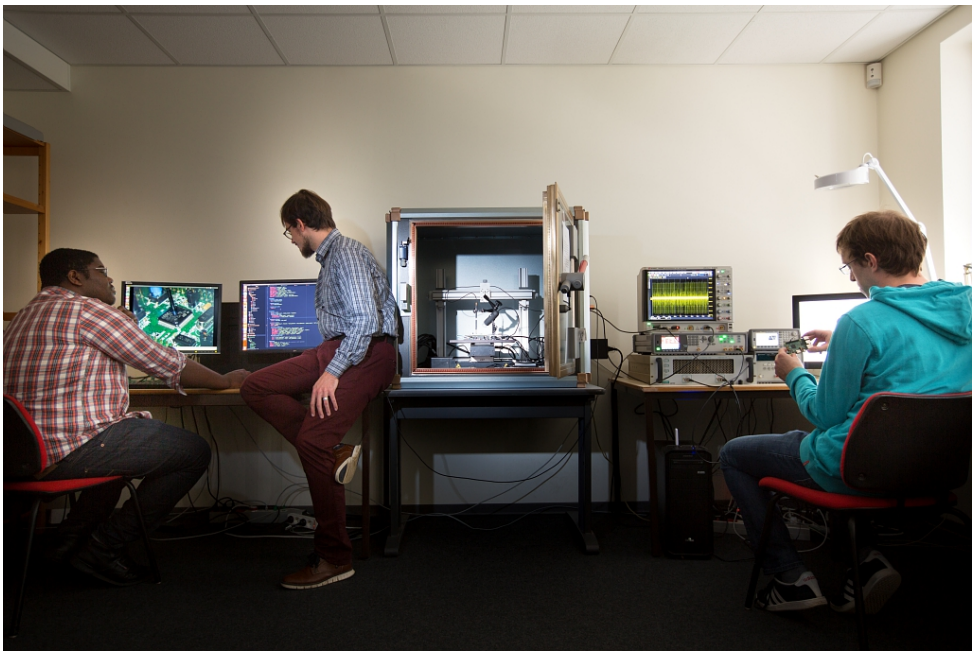

FIGURE 1.8 – Plateformes de test EMA à gauche et Faustine à droite. Au milieu la cage de Faraday permettant d'isoler le système à tester de l'environnement électromagnétique extérieur.

Ces équipements sont connus sous les noms de «EMA» pour réaliser les attaques en écoute électromagnétiques et «Faustine» pour les attaques en injection.

On compte ainsi en élément partagé :

— Un oscilloscope Keysight Infiniium disposant d'une bande passante de 4 GHz permettant l'échantillonnage de 20 M de points par secondes.

— Un ordinateur de contrôle fonctionnant grâce à un processeur Intel Xeon E5-1603v3 cadencé à 2.80 GHz disposant de 4 cœurs de calcul et de 40 GB de mémoire RAM.

— Diverses sondes électromagnétiques de marque Langer.

— Une communication entre le système attaqué et l'ordinateur de contrôle.

L'écoute nécessite uniquement l'ajout d'un pré-amplificateur (3 GHz).

Pour les injections un certain nombre d'équipements supplémentaires sont nécessaires (voir Figure 1.9) :

— Keysight 33509B : permettant de générer le délai avant l'envoi de l'impulsion électromagnétique.

— Keysight 81160A : permettant de générer la forme et le nombre d'impulsions envoyées.

— Milmega 80RF1000-175 : permettant d'amplifier le signal créé par les deux autres équipements et l'envoyer au travers d'une sonde sur le système à attaquer.

— OpenOCD [8], installé sur le PC de contrôle. Logiciel permettant de faire du débogage sur le système attaqué. Dans l'unique but de faire du diagnostic après injection, comprendre l'état dans lequel il se trouve et si l'injection s'est correctement déroulée.

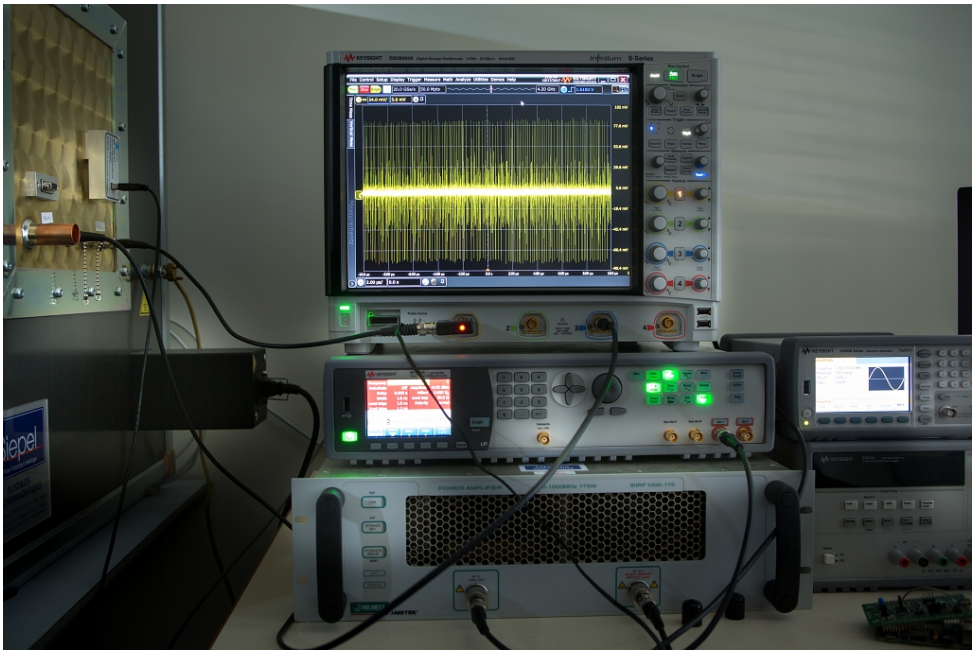

FIGURE 1.9 – De haut en bas et de gauche à droite : l'oscilloscope, le générateur de signal, l'amplificateur de l'impulsion, le générateur de délai, une alimentation électrique.

1.5.3 Résultats attendus

Des attaques physiques ayant d'ores et déjà été réalisées sur des μc, qu'il s'agisse d'écoute ou de perturbation, nos attentes sont de pouvoir les reproduire, au moins dans les conditions, des travaux réalisés au préalable.

S'agissant des μc, nous nous attendons donc à être capable d'effectuer des attaques en perturbation sur tout type de code s'exécutant sur notre cible. Néanmoins, compte-tenu de l'augmentation de la surface d'attaque, nous nous attendons à expérimenter des difficultés pour précisément cibler des comportements intéressant.

S'agissant des SoC, le flou est plus important. À notre connaissance, au début de nos travaux sur les attaques physiques utilisant des émissions électromagnétiques, il n'y a pas de travaux dans la littérature. Ainsi, nous nous attendons à ouvrir un domaine d'application pour ces attaques. Cependant, les différences de base entre les μc et les SoC étant liées à la microarchitecture, nous nous attendons à obtenir des perturbations qui seront plus complexes à maitriser. Pour l'écoute, nous nous attendons à obtenir des résultats comparable à ceux obtenus dans le cas des μc, cependant nous pensons que la différence de vitesse de fonctionnement pourra affecter principalement le matériel nécessaire pour les mesures et les outils d'analyse de celles-ci.

8. http ://openocd.org/

Chapitre 2

Que retrouve-t-on dans les appareils mobiles ?

Résumé de la partie :

Dans cette partie, nous allons nous concentrer sur la microarchitecture. Il s'agit du cœur de ces travaux, nous allons donc présenter les composants qui la caractérisent et présenter certaines notions nécessaires pour appréhender les tests que nous allons mener par la suite. Par conséquent, dans un premier temps, nous allons faire une revue des différentes architectures matérielles existantes dans le domaine des systèmes embarqués. Ensuite nous allons nous restreindre aux architectures que nous avons voulus évaluer spécifiquement en motivant nos choix. On détaillera les concepts et composants génériques de ces systèmes. Nous passerons également un moment pour détailler les protections qui existent sur ces systèmes. Et enfin, nous discuterons de l'effet de ces protections sur la manière dont doit être analysée la microarchitecture face aux attaques physiques.

Sommaire

2.1 Le cœur des systèmes embarqués

Tous les systèmes embarqués manipulent de l'information, qu'il s'agisse de valeurs provenant de capteurs qui seront traitées plus tard, où de données provenant du réseau qui doivent être transmises à l'utilisateur.

On observe alors la présence de systèmes ayant des niveaux d'exigence différents, que ce soit en matière de rapidité d'accès aux informations concernant le volume d'informations traitées voir ou encore le type de logiciel.

Les processeurs étant le cœur de ces systèmes, l'architecture de ces derniers traduit cette différence d'exigence.

Nous allons ici faire une revue succincte des différents types de processeurs que l'on retrouve dans les systèmes embarqués. Compte-tenu de leur grand nombre, cette revue va se concentrer sur les architectures que l'on retrouve le plus souvent.

2.1.1 RISC

Dans les années 1980, la complexité des systèmes et des processeurs était la contrainte principale à leur amélioration. Ainsi deux philosophies se sont affrontées, se rapprocher des concepteurs de logiciel et rendre les processeurs capables de mieux les comprendre. Ou au contraire rendre les processeurs plus simples mais plus rapides quitte à complexifier le travail des développeurs de logiciel.

La différence de philosophie s'exprime dans les différentes architectures (Instruction Set Architecture (ISA)) utilisées. Celles-ci déterminent l'ensemble des actions que peut exécuter un processeur et la manière dont il traite l'information. L'ISA est utile au concepteur de logiciel, elle lui permet de connaitre l'éventail des possibilités d'un processeur.

La microarchitecture est quant à elle, l'implémentation qui va être faite de cette ISA.

Si l'on considère la Figure 1.6, l'ISA permet d'organiser l'enchainement des échanges. Dans ce domaine, on distingue deux grandes catégories historiques.

Dans le cas des PC et serveurs, on utilise le Complex Instruction Set Computing (CISC) (et ses hybrides). On pourrait le résumer par «plus d'opérations dans une seule instruction», les instructions exécutées par le processeur sont complexes et font un travail plus important. À l'inverse, les systèmes embarqués utilisent très majoritairement le Reduced Instruction Set Computing (RISC) que l'on pourrait résumer par «une seule opération dans une seule instruction», où les instructions sont plus simples et leur travail est allégé.

De nos jours, l'écart de performance qui existait au début de la guerre CISC vs RISC est largement négligeable [BMS13]. Le RISC est le modèle de conception matériel le plus utilisé aujourd'hui, pour ces

systèmes embarqués. ARM étant son plus grand ambassadeur avec plus de 90% des parts de marché dans les systèmes embarqués mobiles en 2011.

2.1.2 ARM

Actuellement le leader mondial dans la conception des processeurs pour systèmes embarqués est ARM [1]. Le monde de l'embarqué se divise en 3 usages principaux, afin de répondre aux besoins du marché, ARM propose 3 variantes principales qui nous permettent d'établir 3 catégories de systèmes embarqués (2.1), chacune avec son ISA :

— L'ISA *ARM-A*, «A» pour «Advanced» qui concerne tous les systèmes nécessitant une capacité de calcul se rapprochant des systèmes à architecture CISC. On les retrouve dans des smartphones par exemple.

— L'ISA *ARM-R*, «R» pour «Real-Time» qui concerne tous les systèmes nécessitant des calculs temps réels, pour des applications critiques. Que l'on retrouve dans les assistants de conduite pour véhicules.

— L'ISA *ARM-M*, «M» pour «Microcontroller» qui concerne des systèmes embarqués plus léger tels que les montres connectées.

FIGURE 2.1 – Différentes architectures pour différents usages. Un smartphone utilisera l'ISA ARM-A, la montre plutôt ARM-M et la voiture autonome implémentera un système temps-réel ARM-R.

1. https://www.arm.com/

Toutes ces architectures restent cependant des RISC à part entière. L'ARM-R est réservé à des systèmes aux performances spécifiques, (des applications médicales, en tant que base de systèmes d'avioniques, pour la gestion de systèmes industriels, des antennes relais mobiles etc.) et ces systèmes sont incompatibles avec notre modèle d'attaquant, ils seront ignorés dans le reste des travaux.

Les différentes architectures étant des RISC, on retrouve des caractéristiques communes.

Mais chez ARM, divers jeux d'instructions cohabitent. Apparus au cours du temps et des évolutions de l'architecture on compte principalement :

— le jeu d'instruction ARM original de 32 bits connu aussi sous le nom de AArch32

— le jeu d'instruction Thumb et son évolution Thumb2

— le jeu d'instruction ARM64 ou AArch64

À leurs côtés, on retrouve des jeux d'instructions plus spécifiques, réservés généralement à des composants spécifiques comme Digital signal processing (DSP), Floating-point (FP), NEON, Jazelle, VFP, etc.

Ainsi, un même ISA pourra être compatible avec plusieurs jeux d'instruction. Un natif, pour les calculs génériques, et d'autres sous la forme d'options pour des raisons de compatibilité ou de performances meilleures dans certains domaines comme le traitement du signal.

AArch32

Il s'agit du jeu d'instruction original il a posé les bases pour les suivants. Il repose sur des données de 32 bits qui sont échangées dans le système. On le retrouve principalement supporté par les ISA ARMv7-A.

Les calculs s'effectuent en utilisant 15 registres de 32 bits. Auxquels il faut ajouter un registre de compteur de programme (PC).

Ce jeu d'instruction est désormais de moins en moins utilisé, il a été remplacé par 2 déclinaisons, Aarch64 majoritairement utilisé pour les ISA ARM-A (à partir de v8) et Thumb pour les ISA ARM-M.

Thumb et Thumb-2

Apparu en 1994 Thumb est un jeu d'instruction compact, sur 16 bits. Toutes les instructions sont codées sur 16 bits, mais les valeurs manipulées peuvent être sur 32 bits. Il permet une meilleure optimisation dans le cas où les contraintes de taille et de communication entre les éléments sont fortes (programmes petits, taille des données restreintes, exécution des instructions très séquentielle, etc.). Il permet également d'optimiser au maximum l'utilisation des ressources matérielles.

Les opcodes des instructions sont plus restreints, il n'est donc pas possible de faire des branchements trop lointains dans le code, des instructions vont être limitées à certains registres uniquement etc.

Son évolution Thumb-2, apparue en 2003, apporte un pont entre AArch32 et Thumb. Des instructions sur 32 bits sont de nouveaux intégrées, de nouvelles instructions sont apportées afin d'atteindre les performances et les capacités du AArch32.

AArch64

Il s'agit d'une évolution directe du AArch32. On retrouve aujourd'hui ce jeu d'instruction principalement supporté par les ISA ARMv8-A.

Le nombre de registres augmente et passe à 33, dont un registre PC. Les instructions restent sur 32 bits, mais les registres passent à 64 bits. Pour conserver la compatibilité avec AArch32, les registres sont tous accessibles en mode 32 bits, seuls les 32 derniers bits sont alors lus.

2.1.3 Les architectures concurrentes

D'autres architectures à l'usage plus spécifique, sont aussi présentes dans le monde de l'embarqué. Parmi elles, on retrouve :

— Programmable Intelligent Computer (PIC)

— Microprocessor without Interlocked Pipeline Stages (MIPS)

— x86

— les architectures libres telles que celles basées sur l'ISA RISC-V [Wat16, PW17]

— et bien d'autres qui sont parfois unique à un système en particulier (comme Cell utilisé principalement dans la Playstation 3 de Sony) ...

Compte tenu de la très grande emprise d'ARM sur le marché des systèmes embarqués, les processeurs basés sur ses ISA seront donc le centre de nos travaux. On a ainsi vu que la gamme d'ARM se compose, pour l'embarqué grand public, des ARM-A et ARM-R. Nous allons par conséquent nous concentrer sur les ISA utilisant ces jeux d'instructions.

2.2 Du transistor à la microarchitecture

Afin de voir de manière plus concrète ce qui se trouve au sein d'un système embarqué, nous allons ici faire un récapitulatif des éléments qui le composent et des interactions entre eux qui constituent la microarchitecture.

Comme on l'a dit, les systèmes s'articulent autour d'un centre chargé de gérer les calculs.

2.2.1 Transistors

A la base des systèmes électroniques on retrouve des transistors. Il s'agit des briques élémentaires qui permettent de mettre en place toute la logique des systèmes.

D'un point de vue schématique, un transistor peut être vu comme un interrupteur contrôlé par un courant électrique.

FIGURE 2.2 – **(E)** Entrée qui commande l'ouverture ou la fermeture de la connexion entre (C_1) et (C_2). À gauche, un transistor classique, à droite un transistor inverse.

La technologie aujourd'hui la plus couramment utilisée est celle des «transistors à effet de champ à structure métal-oxyde-semi-conducteur» (MOSFET). La tension électrique appliquée en entrée génère un champ électrique, qui permet d'établir une connexion entre les deux points (C_1) et (C_2) sur la Figure 2.2.

Un circuit binaire reconnait deux valeurs 0 et 1, elles sont représentées dans les circuits par les rails d'alimentation. Ces rails sont appelés «V_{DD}» et «*Ground*» (ou parfois V_{SS}). La tension exacte sur chaque rail dépend de l'implémentation du système et permet de représenter au niveau logique les valeurs binaires 0 (*Ground*) ou 1 (V_DD).

Ainsi, avec un transistor connecté par (C_1) au rail V_{DD}, l'application de la tension de seuil sur l'entrée (E) permet de connecter (C_1) et (C_2). En considérant (C_2) comme la sortie de notre circuit, en appliquant la tension de seuil en entrée, il est possible de propager en sortie la tension correspondant à V_{DD}.

De la même manière, il est possible d'obtenir la tension du *Ground* par la sortie, qui devient alors (C_1), en connectant le (C_2) à ce dernier et en appliquant la bonne tension sur l'entrée.

Au niveau logique, dans le premier cas (C_2) est la sortie et propage un 1, dans le second (C_1) est la sortie et propage un 0 binaire.

Le transistor inverse a un fonctionnement inversé par rapport au transistor classique. Il se ferme (et donc connecte C_1 et C_2) lorsque la tension appliquée en entrée est inférieure à la tension de seuil.

En chainant plusieurs transistors il devient alors possible de créer des fonctions logiques plus complexes. Ces fonctions permettent de manipuler l'information et de réaliser des calculs.

2.2.2 Logique combinatoire

Dans un premier temps, on considère la logique combinatoire. Elle permet en associant des transistors de manipuler l'information pour réaliser une opération. Le résultat va ici dépendre uniquement des entrées.

En connectant plusieurs transistors, il est possible de créer des opérations logiques comme celle visible sur la Figure 2.3. Dans ce cas, il y a deux entrées qui vont commander l'activation de quatre transistors.

La fonction est un NON-ET et les entrées x et y sont fixées à 1. Dans ce cas, on considère que la tension de seuil est atteinte pour les deux entrées et donc la valeur logique est un 1.

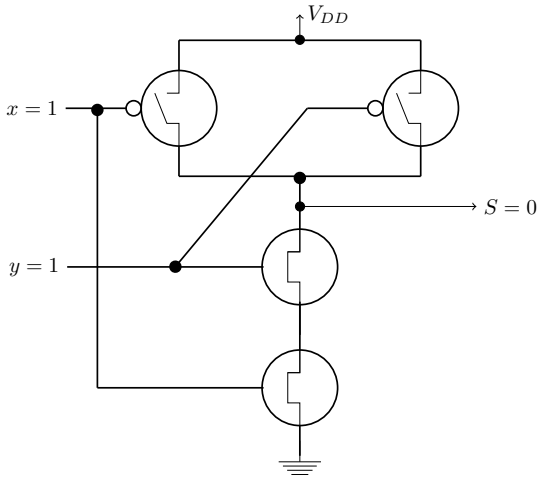

FIGURE 2.3 – Schéma d'une porte NAND. Deux transistors sont ici ouverts et connectés au VDD, les deux autres sont fermés et connectés au Ground. C'est donc sa tension qui sera transmise en sortie.

Dans ce cas, la sortie S est l'équivalent d'un 0 logique. On peut cependant évaluer les différents états possibles à l'aide de la table de vérité qui caractérise cette porte logique :

x	**y**	**S**
0	0	1
0	1	1
1	0	1
1	1	0

Il existe de nombreuses autres opérations combinatoires possibles. L'ensemble des opérations logiques est regroupé dans l'unité arithmétique et logique (ALU).

2.2.3 Logique séquentielle

Dans un second temps, la logique séquentielle vise quant à elle à associer les transistors pour créer des cellules dont la valeur dépend d'un état précédent.

Ces éléments de mémorisation (les bascules), permettent de conserver une information équivalente à la valeur d'un bit logique.

Si l'on considère 32 de ces bascules, il devient possible de créer des registres de 32 bits.

2.2.4 Horloge

En reliant des portes logiques à des bascules, il devient possible de réaliser des calculs et de sauvegarder les valeurs de ces calculs pour les réutiliser plus tard.

L'horloge permet de réaliser plusieurs calculs de manière séquentielle, en fixant un point de rafraichissement des valeurs d'entrées des différentes portes et bascules. Un cristal de quartz oscillant à une fréquence constante est généralement l'élément de base pour générer le signal d'horloge.

La fréquence d'oscillation du cristal va être multipliée ou divisée au moyen d'une Phase Locked Loop (PLL), afin de générer la fréquence d'horloge du système et donc générer des «tic» réguliers.

Ainsi, le «tic» d'horloge signale un point à partir duquel les bascules doivent prendre en compte leur nouvelle entrée, les transistors doivent commencer à devenir (ou non) traversant.

2.2.5 Chemin critique

Entre deux tics d'horloge, pour qu'une opération soit correctement prise en compte, il faut que le circuit soit dans un état stable. Il faut donc que chaque point de sortie aie obtenu la valeur correspondant à l'entrée courante.

On peut alors caractériser une porte logique par le nombre de transistors consécutifs qu'il faut traverser pour passer de la valeur en entrée à la valeur en sortie.

Sur la Figure 2.2, la profondeur du circuit est d'un seul transistor. La transmission entre le moment où l'entrée a une tension à appliquer et le changement d'état dans la connexion C_1-C_2 est donc d'une unité de temps.

Si l'on considère le circuit sur la Figure 2.3, il y a cette fois quatre transistors impliqués, et seuls deux doivent être traversés consécutivement pour obtenir le résultat. La profondeur de ce circuit est donc de deux. Les entrées sont chacune connectées à deux transistors, la propagation s'effectue sur plusieurs d'entre eux, de plus ils sont également connectés entre eux. Ces interconnexions augmentent d'autant plus le temps de propagation qui est alors plus long qu'une unité de temps.

Plus la profondeur du circuit est importante plus le temps de propagation des valeurs en entrée est importante. Le chemin critique est donc d'autant plus long.

La durée minimale entre deux tics est donc fixée sur le temps de propagation pour réaliser l'opération la plus complexe, c'est le chemin critique.

2.2.6 Processeur

Le processeur se compose d'éléments de logique séquentielle et combinatoire permettant de reconnaitre les instructions machines qui vont lui être envoyées et effectuer les traitements en conséquence.

Il s'agit principalement des unités de calcul et de leurs registres associés. Il est possible de disposer de plusieurs unités de calculs, les cœurs, dans un même processeur.

2.2.7 Étagement

L'exécution d'une instruction est la suite de plusieurs circuits combinatoires. Ces opérations sont découpées en séquences plus courtes. C'est l'étagement du processeur.

Ainsi, une instruction passe au travers d'un pipeline où son exécution est séparée en plusieurs étages.

Lorsqu'une instruction a terminé un étage, elle passe au suivant et l'instruction suivante à exécuter vient prendre sa place.

C'est le signal d'horloge qui annonce la fin d'un étage et le passage des instructions au suivant.

2.2.8 Chemin de données

Le chemin de données représente la manière dont l'information transite et est traitée à travers le processeur. Il s'agit des différents chemins que cette information va emprunter, en particulier les circuits combinatoires et logiques qui sont utilisés.

2.2.9 Jeu d'instruction

Le jeu d'instruction est l'interface entre la microarchitecture et le logiciel du système. Il permet de représenter au développeur les capacités du système.

Le jeu d'instruction constitue la partie visible par le concepteur, de la logique du processeur.

Multicœurs

Dans le cas des SoC, on peut retrouver, dans le processeur plusieurs unités de calcul. Elles vont être connectées à la même mémoire et fonctionner en parallèle pour exécuter des applications différentes. Ils sont cependant connecté à la même mémoire, ont dit alors qu'il s'agit d'un processeur multicœur.

Dans le cas des μc, il est plus rare de retrouver ce type de mise en place.

Mémoires caches

Le principe de base de la mémoire cache est de permettre de compenser les faibles vitesses de transfert des mémoires de masse. En effet, un processeur est capable d'exécuter plusieurs milliards d'instructions par secondes, mais les composants mémoire offrant des temps d'accès compatible avec ces vitesses sont très coûteux.

Pour palier à cela, les périphériques de mémoire de masse sont généralement déportés (cas général des SoC) ou la mémoire disponible est de faible capacité (cas général des μc).

Le fait de déporter ces organes de mémoire entraine une forte augmentation des temps d'accès, pour palier à cela les mémoires caches sont apparues.

On les appelle L1 (pour Level 1), L2 et L3. Les contraintes indicatives de temps d'accès sont respectivement d'un à deux cycles d'horloge pour L1, jusqu'à une centaine de cycles pour la mémoire Random Access Memory (RAM). C'est autant de cycles durant lesquels le cœur reste en attente. Cependant, ce chiffre est à comparer avec les dizaines de milliers de cycles pour rapatrier une donnée depuis un disque dur par exemple.

Ces mémoires créent la hiérarchie de la mémoire que l'on retrouve dans les systèmes modernes.

Diverses architectures existent pour l'attribution des caches aux cœurs de calcul, celle-ci sera donc présentée lorsque nous aborderons nos choix de cible.

La hiérarchie de la mémoire fonctionne de manière asynchrone avec le processeur, ceci dans le but de lisser les temps d'accès à la donnée par le processeur. Ainsi, les différents niveaux de mémoire caches sont conçus pour permettre de toujours placer la donnée nécessaire le plus proche du cœur.

Parmi les stratégies mises en place on peut citer :

— La séparation des données et des instructions, instructions et données sont séparées dans des fractions de mémoire cache distinct.

— L'associativité, permettant a des zones définies dans la mémoire de masse d'atterrir dans une même ou dans différentes zones de la mémoire cache.

— La localité spatiale, principe selon lequel dans le temps des données ou des instructions proches les unes des autres sont accédées.

— La localité temporelle, principe selon lequel une donnée qui vient d'être accédée a des chances d'être accédée de nouveau dans peu de temps.

Une ligne de cache est la plus petite unité pouvant être transférée en une fois par un cache. Chaque ligne, se compose d'un certain nombre de bits, défini à la conception.

Chaque niveau de mémoire cache ne représentant qu'une fraction de la taille du niveau supérieur, des mécanismes de consistance sont mis en place, c'est la consistance des caches.

Dans le cas d'un processeur multicœur, un ou plusieurs niveaux de cache va être séparé. C'est le cas des architectures où chaque cœur dispose de son propre cache L1. Des données présentes simultanément sur deux caches de même niveau peuvent apparaitre. Dans le but de conserver la cohérence des caches et de s'assurer qu'en cas de modification de la donnée d'un des deux caches, l'autre soit également modifié, différents mécanismes peuvent mis en place. En particulier, la mise en place d'un snooper, en charge d'écouter les différentes transactions pour prévenir ou modifier directement les autres caches.

2.2.10 Mémoire virtuelle

Les cœurs fonctionnent avec un mécanisme de mémoire virtuelle. Ce mécanisme permet d'isoler les différentes applications qui pourraient s'exécuter sur un cœur.

L'adresse virtuelle est celle du point de vue de l'application et du processeur dans la hiérarchie de la mémoire. Une traduction de cette adresse est effectuée pour obtenir l'adresse physique, position réelle en mémoire.

C'est la Memory Management Unit (MMU) qui est en charge de cette traduction.

2.2.11 Circuits spécifiques (périphériques)

Le processeur n'est pas le seul composant, il en existe d'autres chargés de tâches spécifiques, les périphériques.

On y retrouve :

— la mémoire,

— les unités de contrôle du courant,

— les unités de gestion des entrées-sorties,

— les unités de gestion de l'horloge,

— etc.

Dans un SoC on retrouve tous les circuits qui permettent de gérer la hiérarchie de la mémoire, la MMU, le snooper, etc.

On retrouve également des circuits graphiques, des processeurs spécialisés dans le traitement des tâches de gestion de l'affichage. Ce sont des processeurs multicœurs, dans les cas où ils ne disposent pas de leur propre mémoire, des circuits de cohérence sont également ajouté au SoC.

2.2.12 Bus

Le bus sert à interconnecter les composants. Il permet ainsi aux différents composants d'échanger des données.

2.2.13 Isolation

Les cœurs sont multi applicatifs, de la même manière que les μc, plusieurs modes d'opérations sont mis en place.

Sur les SoC on les retrouve en tant que :

— EL0 : mode utilisateur chaque application est dans ce mode.

— EL1 : seules les applications privilégiées et l'OS sont dans ce mode.

2.2.14 Microarchitecture

La microarchitecture est composée de tous les éléments, présentés précédemment, qui interagissent entre eux. Les principes de fonctionnement qui ont été exposés ici sont des principes génériques et ne couvrent pas l'éventail de possibilités dont disposent les concepteurs. En particulier, les caractéristiques physiques de chacun des composants peuvent différer.

À partir de ces éléments génériques, il faut donc être en mesure d'inférer les spécificités en observant le fonctionnement.

2.2.15 Différences entre microcontrôleur et System-on-Chip

Les μc et les SoC tendent à se rapprocher. Ils possèdent les mêmes éléments et les différences entre les deux types sont de moins en moins évidentes à identifier.

En dehors de la mémoire cache qui est relativement absente du monde des μc, ces derniers se rapprochent des SoC sur tous les autres aspects. La principale différence reste les jeux d'instructions que l'on retrouve sur ces deux types de systèmes. La liste des différences notables entre ces deux types de système s'en ressent :

— Vitesse de fonctionnement des cœurs de calcul plus importante dans le cas des SoC.

— Présence d'une hiérarchie de la mémoire complexe.

— Technologies de gravure utilisées (qui se répercute sur la taille totale du circuit).

— Généralement davantage de transistors dans les SoC.

Nous verrons par la suite si ces différences ont un effet sur la tenue des attaques physiques.

2.3 Sécurité matérielle

Nous nous sommes focalisés sur les attaques physiques, le but de celles-ci est d'observer ou de perturber le fonctionnement de la microarchitecture.

Dans cette optique, nous nous sommes focalisé sur la sécurité matérielle afin d'identifier les contre-mesures existantes face à des attaques physiques. Dans cette section, la sécurité matérielle correspond aux ajouts ou modifications au niveau matériel afin d'assurer la sécurité des systèmes embarqués face à des attaques physiques.

Dans un premier temps, nous effectuons une revue des contre-mesures existantes, puis nous abordons le cas spécifique des enclaves sécurisées.

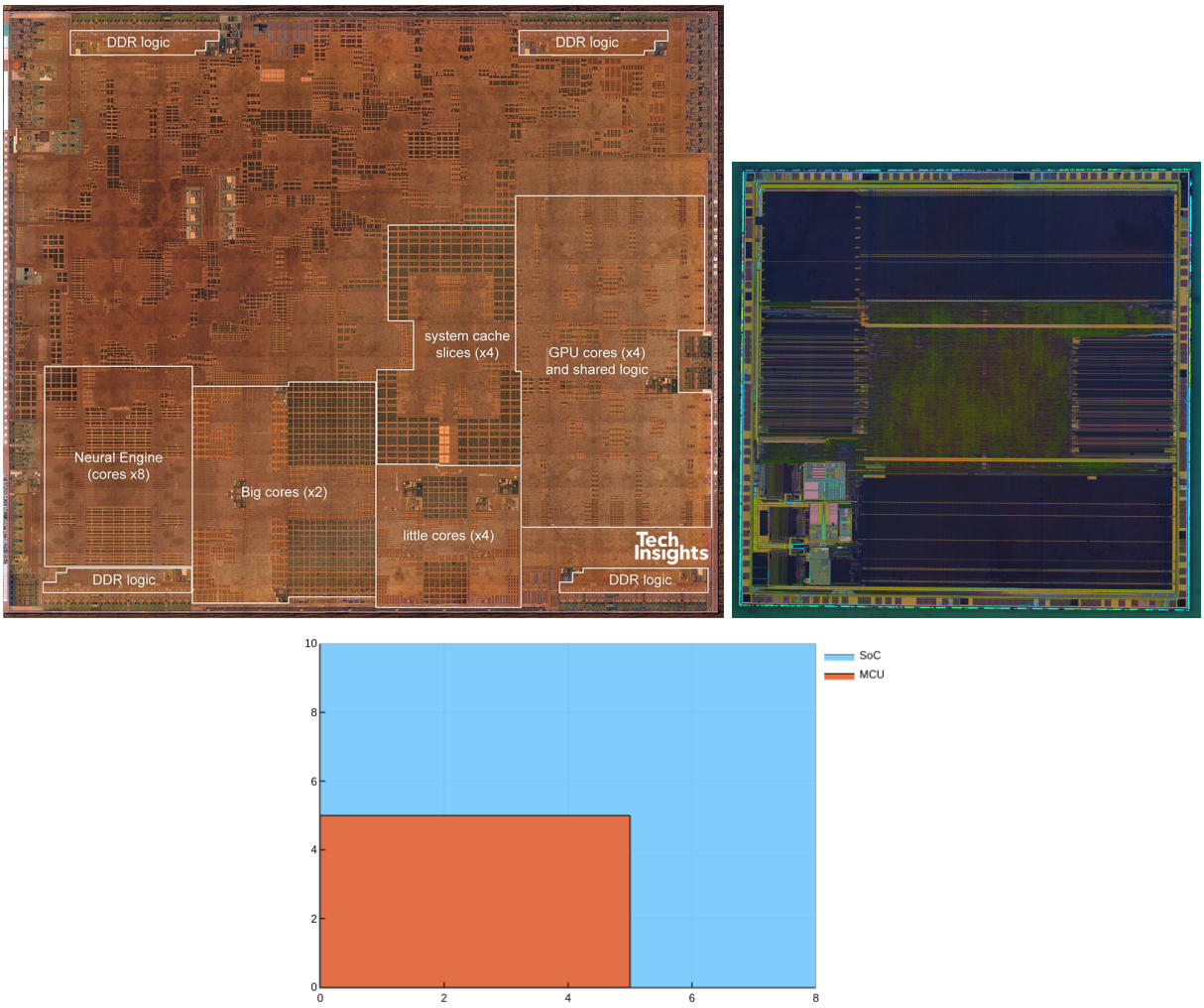

FIGURE 2.4 – Comparaison de taille entre un μc et un SoC. À gauche le SoC Apple A12, à droite le μc STM32F103VGT6-HD.

2.3.1 Contre-mesures contre les attaques physiques

Il existe diverses contre-mesures contre les attaques physiques, on peut les classer selon les deux grandes familles d'attaques.

Dans le cas des attaques par observation, le défi consiste à limiter le signal sortant du circuit ou à le rendre inexploitable. L'utilisation de «shields», des boucliers permettant de contenir les émissions est une solution permettant de limiter le signal sortant. D'autre part, il est également possible de générer du bruit électromagnétique, qui vient diminuer le rapport signal/bruit et permet de complexifier la tenue des attaques par observation. Il est aussi possible de limiter les possibilités d'exploitation pour des cas spécifiques : rendre les signaux plus uniformes dans le cas d'attaques par chronométrage en utilisant du temps constant, ou au contraire le rendre imprévisible en introduisant des gigues.

Dans le cas des attaques par perturbation, il est cette fois-ci question de limiter ou de détecter les signaux entrants. Dans ce cas, des techniques de redondance permettent d'augmenter la difficulté expérimentale. Par exemple, plusieurs sous-circuits effectuent les mêmes opérations en parallèle ou effectuent

plusieurs fois la même opération. Il faut alors perturber plusieurs sous-circuits de la même manière au même moment ou un seul à plusieurs moments différents. Il est aussi possible de détecter un signal entrant en utilisant des sous-circuits spécifiques ([ZDT+14]).

La solution des boucliers électromagnétiques est une solution adaptable aux observations et aux perturbations. Elle constitue donc un premier niveau de protection. Néanmoins, cette protection ne modifie pas la microarchitecture en elle-même et est donc une solution d'appoint uniquement qui devra être complétée par d'autres solutions de protections que nous allons voir par la suite.

2.3.2 Les enclaves sécurisées

Il existe d'autres méthodes de sécurités comme l'ajout de circuits dédiés à des fonctions de sécurité. Ces éléments non nécessaires au bon fonctionnement des systèmes embarqués, sont ajoutés pour assurer une protection de l'utilisateur, ils se chargent des fonctionnalités les plus sensibles. On peut identifier deux types de modifications :

— Les ajouts internes : il s'agit de modifications apportées à la microarchitecture. Les cœurs de calculs vont subir des modifications, comparées à des microarchitectures dites non-sécurisées. Ces modifications sont intégrées directement durant la conception du système.

— Les ajouts externes : cette fois les modifications sont apportées uniquement à la fin de la fabrication. Le ou les éléments ajoutés le sont en dehors de la microarchitecture.

Ces deux techniques répondent à une volonté d'isoler une partie du système de l'influence de l'utilisateur final, pour y réaliser des opérations dites «sensibles», celui-ci n'a alors plus la maitrise totale de son matériel, mais délègue la sécurité aux concepteurs et fabricants.

Certification

L'avantage de ces parties isolées est que leur comportement est connu et plus ou moins figé. Ils peuvent intégrer des opérations supplémentaires par le biais de mises à jour dans certains cas, mais selon les cas celles-ci sont limitées ou entrainent des contraintes que nous verrons par la suite. Ils ne sont donc pas soumis à ce que pourrait en faire l'utilisateur, leur fonctionnement peut donc être certifié.

Cette certification assure à l'utilisateur qu'une entité indépendante du constructeur a réalisé des tests de sécurité et que ceux-ci se sont révélés suffisant pour un certain niveau d'utilisation. En France, l'Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) certifie et se charge d'agréer les centres Centre d'Évaluation de la Sécurité des Technologies de l'Information (CESTI) qui réalisent les tests de certification.

Cette certification (normes ISO 15408-x) connue sous le nom de Critères Communs, suit des critères d'évaluation, selon une norme internationale. Des exemples et pratiques sont proposés et doivent être respectés par les différents CESTI.

Enclaves sécurisées

Ces éléments de sécurité sont du point de vue de l'utilisateur final des enclaves, il s'agit de boites noires. On distingue les enclaves sécurisées internes intégrées à la microarchitecture et les enclaves externes qui sont intégrées à l'architecture mais sont indépendantes de la microarchitecture. L'utilisateur ne dispose d'aucun moyen pour accéder à ces enclaves, il n'est ainsi plus le maitre de son matériel.

Enclaves sécurisées externes

Dans les enclaves sécurisées externes, on retrouve les coprocesseurs. Ceux-ci peuvent être placés dans le SoC ou à l'extérieur (voir Figure 2.5).

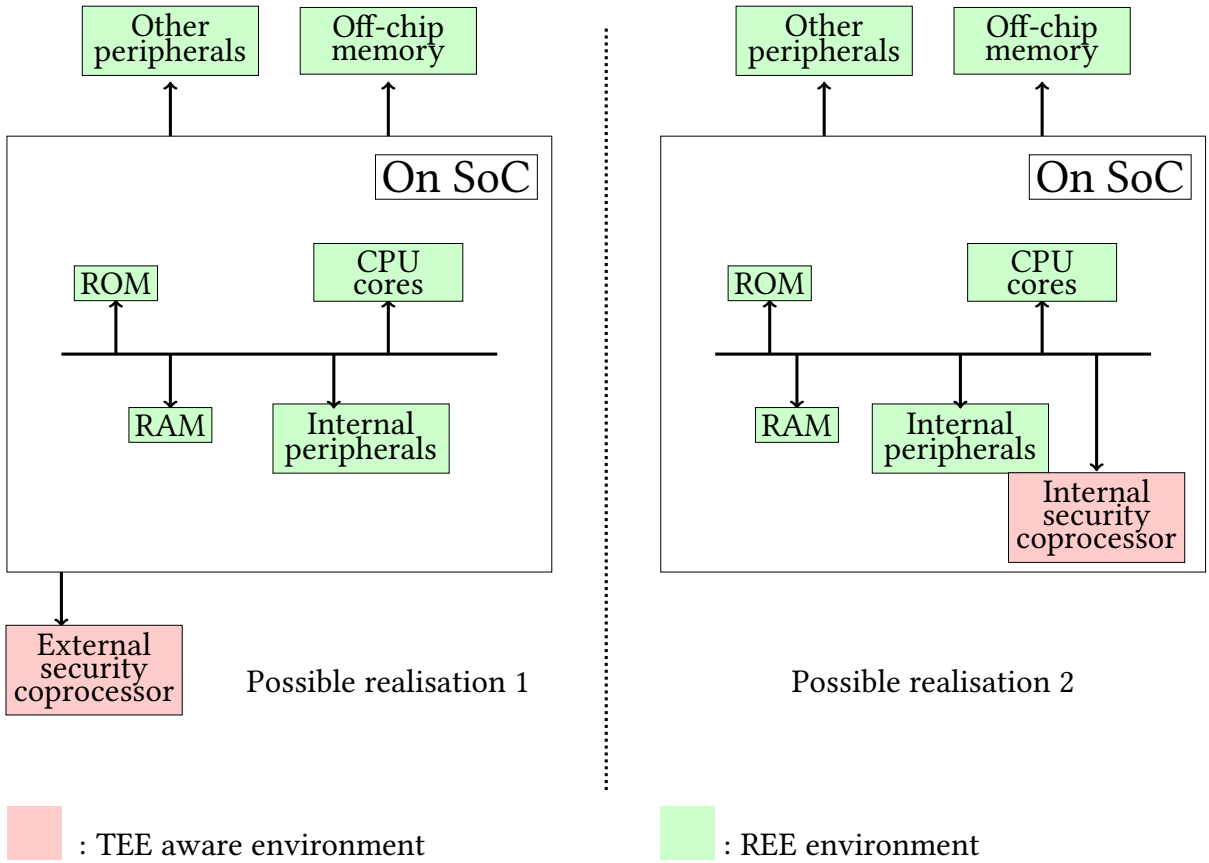

FIGURE 2.5 – Implémentations possibles d'une enclave, à gauche l'implémentation avec un coprocesseur disposé à l'extérieur, à droite à l'intérieur du système, dans ce cas il s'agit forcément d'un System-on-Chip (SoC).

Ceux-ci prennent en charge des fonctionnalités sensibles. Parmi lesquelles la sauvegarde de clefs de chiffrement, ou des opérations d'authentification. On retrouve dans ces coprocesseurs :

— Les cartes à puce

— Les Secure Elements

— Les Trusted Platform Module (TPM)

Ces coprocesseurs, sont généralement indépendant, ils disposent de leur propre mémoire, notamment. Ils n'ont cependant que des ressources limitées comparativement aux cœurs d'un SoC ou d'un μc.

Les téléphones mobiles sont un cas à part, historiquement, ils intègrent des protections supplémentaires. Plus particulièrement, ils disposent de processeurs de modem (baseband) [CK] qui gèrent les communications qui s'effectuent via le réseau de l'opérateur. On y retrouve alors principalement deux architectures [Wei] :

— Le processeur de modem est maitre du système, devant le processeur du SoC. Ils partagent des ressources comme la mémoire.

— Chacun dispose de ses propres ressources, aucun n'a la main, mais des transactions entre les deux ont lieu, notamment pour l'accès au réseau mobile.

Leur architecture est donc en partie différente des autres plateformes embarquées fonctionnant avec des SoC ARM. Chez Qualcomm on les retrouve séparées sous l'appellation Mobile Station Modem (MSM) lorsque les SoC sans modem sont les Application Processor Qualcomm (APQ).

Enclaves sécurisées internes

Aux côtés des coprocesseurs sécurisés, non standardisés, ce qui signifie que chaque produit doit être certifié séparément. Un groupement d'entreprises privées, GlobalPlatform, a souhaité créer un standard permettant une certification globale. Menant à l'établissement d'un profil de protection certifié selon les Critères Communs, le Trusted Execution Environment Protection Profile (TEE PP), que doivent respecter les différents concepteurs de Trusted Execution Environment (TEE).

Une TEE est donc un environnement d'exécution respectant le TEE PP. Il s'agit d'une zone isolée par des modifications matérielles du reste du système. Les utilisateurs peuvent ainsi y accorder leur confiance pour la protection des informations qu'elle traite.

Le fonctionnement simplifié de ce profil de protection est qu'un système va être séparé en deux environnements distincts :

— Le **Rich Execution Environment (REE)** qui dispose de toutes les applications «normales» du système, toutes celles pour lesquelles l'utilisateur n'est pas protégé matériellement.

— Le **TEE** qui contient lui un Trusted OS, et des Trusted Application (TA). Ces applications et l'OS sont jugés de confiance par les concepteurs puisque l'isolation est assurée.

Le profil de protection prévoit donc une isolation entre ces deux environnements, de telle sorte qu'ils ne puissent pas entrer en collision afin d'assurer la Root–Of–Trust (RoT).

La RoT, consiste à assurer que le fonctionnement d'un système est exactement celui prévu par le concepteur, et qu'aucune modification non prévue n'a pu être effectuée. Ainsi, le démarrage du système est sécurisé, seul un OS validé et signé cryptographiquement par le concepteur, peut être lancé.

Chaque environnement dispose ainsi de son propre OS, de ses différentes couches applicatives, etc.

Ce profil de protection intègre ainsi le fonctionnement des enclaves sécurisées externes telles quelles ont été présentées précédemment. Les coprocesseurs constituent donc le TEE totalement isolé quand le reste est le REE. Cependant, le profil de protection permet également la mise en place d'un autre niveau d'isolation tel que celui proposé par la TrustZone d'ARM.

TrustZone La TrustZone (TZ) est l'implémentation de la TEE proposée par ARM sur ses SoC depuis 2014.

Cette option de sécurité se base sur divers éléments architecturaux ajoutés par rapport aux SoC d'ancienne génération ne supportant par la TZ. Le fonctionnement de la TZ est schématisé sur la Figure 2.6.

Tous les composants dans et hors du SoC peuvent être scindés en deux états. L'un qui sera exclusif au REE et l'autre exclusif au TEE. Pour assurer l'isolation des deux environnements, ARM a mis en place plusieurs modifications architecturales :

— Ajout d'un 33e bit pour toutes les données. C'est le NS bit, il permet d'attester de l'état d'une donnée manipulée, si elle appartient ou non à l'espace sécurisé.

— 4 nouveaux niveaux de privilèges, SEL0 pour les applications sécurisées, SEL1 pour l'OS sécurisé, SEL2 pour l'hyperviseur sécurisé et EL3 pour le mode Moniteur.

— Conception de divers composants afin d'étendre l'isolation hors du SoC. Par exemple le TrustZone Address Space Controller (TZASC) est une MMU supplémentaire sécurisée permettant de partager ou non la mémoire avec des périphériques pouvant manipuler des données sécurisées ou non.

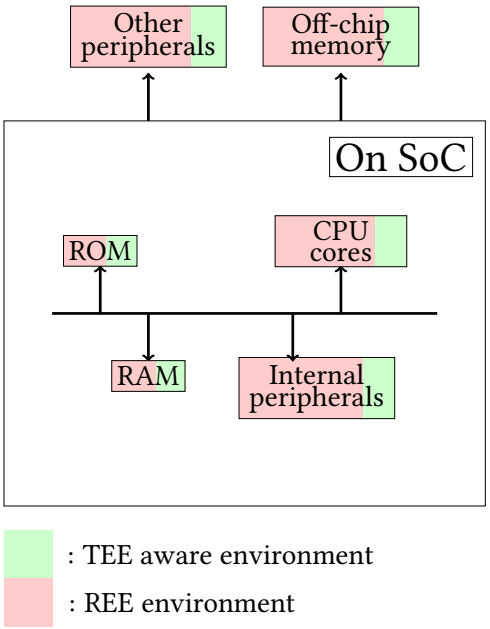

FIGURE 2.6 – Implémentation de la TrustZone, toutes les ressources peuvent être placées en partie sécurisée à la demande de chaque application sécurisée.

Au niveau logiciel, la séparation se produit donc de la manière visible sur la Figure 2.7.

Ainsi, aux deux modes d'opération utilisateur et système, ARM ajoute quatre nouveaux modes.

SEL0, constitue le pendant sécurisé du EL0. Les applications qui s'y exécutent peuvent être approuvées et signées par le concepteur.

SEL1, constitue quant à lui l'OS sécurisé. De la même manière que l'OS.

SEL2 est pour sa part un niveau où va se placer l'hyperviseur permettant l'utilisation de plusieurs OS sécurisés.

Le passage d'une zone à l'autre s'effectue par un passage dans le moniteur qui est le dernier mode ajouté.

Lorsqu'une application non sécurisée demande l'appel à une fonction sécurisée, elle passe par un pilote présent dans le système d'exploitation. Ce pilote va alors rediriger la demande vers le moniteur qui est chargé de sa vérification. Les vérifications effectuées dépendent de l'implémentation choisie. Ensuite le moniteur peut effectuer d'autres traitements comme vider la totalité de la mémoire cache, et va passer la demande à un routeur de fonctions. Celui-ci est présent dans le système sécurisé et c'est lui qui va lancer l'applet correspondant. Le retour éventuel de données emprunte le même chemin.

L'isolation proposée par ARM est donc une extension de l'isolation proposée à ce jour sur les systèmes ayant un OS et des applications. Le passage dans le moniteur étant analogue à un appel système dans le cas des OS. Le principal apport de cette isolation est qu'elle est ici totalement inaccessible à l'utilisateur puisqu'elle repose sur des éléments dont il ne dispose pas. Dans ce cas, la RoT est assurée au moyen de vérifications de signatures qu'un utilisateur ne peut pas effectuer.

On peut ainsi qualifier la TZ de protection logicielle assistée par le matériel. Cela cause principalement deux problèmes :

— D'une part, l'utilisateur n'a plus la main sur son matériel, il doit faire pleine confiance en les concepteurs pour assurer sa protection et ne dispose d'aucun moyen d'évaluer lui-même le comportement de cette enclave.

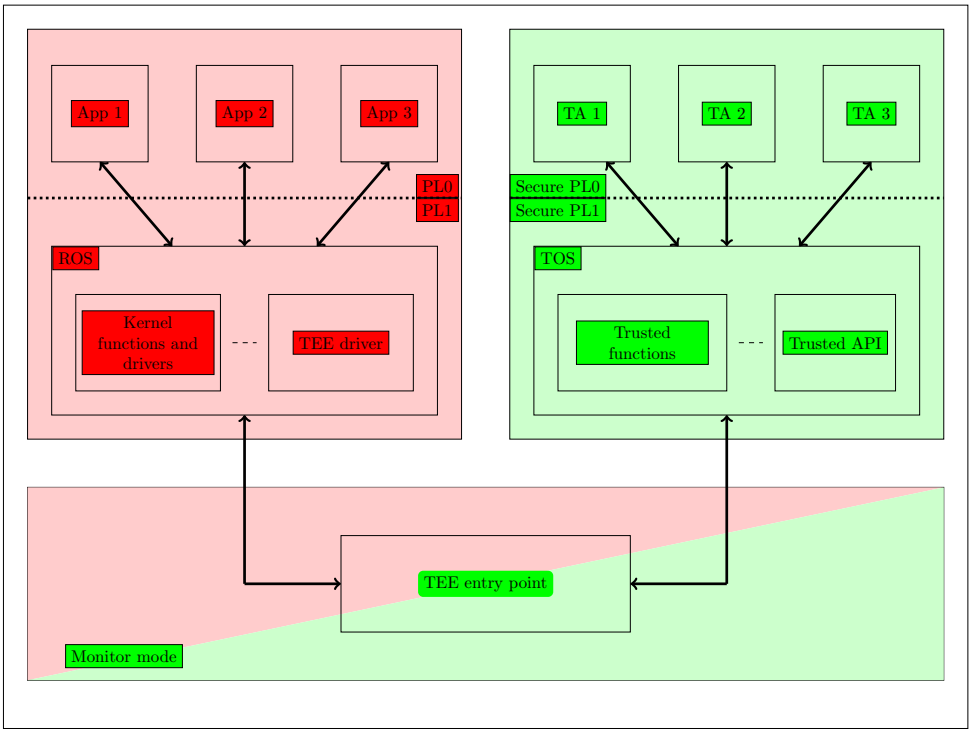

FIGURE 2.7 – Implémentation logicielle de la TrustZone. En vert l'environnement REE avec les 2 niveaux d'exception usuels. À droite la nouvelle partie isolée sécurisée. En bas, le Moniteur, permettant le passage d'une zone à l'autre.

— D'autre part, la sécurité reste logicielle et donc théoriquement sensible à toutes les attaques physiques qui sont possibles sur microarchitecture sans TZ.

Lors de la conduite de nos travaux, le niveau EL3 était connu sous le nom «Monitor mode», néanmoins ARM a modifié sa nomenclature ainsi que les préconisations faites aux concepteurs de logiciel et aux constructeurs. En fournissant un modèle d'appel à la TZ basé sur le nouveau «Arm trusted firmware» (ATF) [ARM19], cet ATF comporte des briques permettant de concevoir des OS sécurisés. Ainsi ARM souhaite assurer une certaine uniformité, ARM fixe des conventions d'appel, préconise et fournit certains pilotes à utiliser etc.

Ces nouvelles préconisations ne seront pas prises en compte au long de ce manuscrit, néanmoins elles n'entrainent pas une modification matérielle, les conclusions de nos recherches sur l'effet de la TZ face aux attaques physiques restent donc valables.

2.4 Conséquences de ces couches matérielles

L'ajout d'éléments matériels conduit à l'augmentation du nombre de sous-circuits (séquentiels et combinatoires) et de transistors. La surface doit être couverte par les lignes d'alimentation et la propagation de l'horloge augmente aussi grandement.

Or ces éléments fonctionnent avec du courant électrique. D'un côté, le passage d'un courant dans le circuit provoque une émission électromagnétique [HSK97]. De l'autre côté, un couplage électromagnétique peut se produire en présence d'un champ électromagnétique suffisamment puissant dans le voisinage du circuit. Ce couplage peut entrainer la modification des valeurs logiques en transit sur le circuit, un 0

logique peut devenir un 1 logique et vice-versa.L'augmentation de la surface possible de couplage, par l'augmentation du nombre de transistors ou de sous-circuits est alors bénéfique pour un attaquant.

Dans le cas d'attaques en injection il peut donc se produire plusieurs phénomènes menant à un effet, permettant d'expliquer les modèles de fautes usuels. D'une part, une violation de la contrainte de temps induite par le chemin critique peut se produire : par couplage, le signal d'horloge peut être modifié.

Cette violation peut amener des circuits combinatoire à voir leur valeur sauvegardée trop tôt amenant ainsi à une situation où la tension de seuil n'a pas pu être atteinte pour prendre en compte un changement de valeur, le circuit en question pourra alors conserver sa valeur passée (stuck-at), ou au contraire prendre une valeur non prévue par rayonnement trop intense (bridging).

D'autre part, les circuits ou transistors peuvent être affectés individuellement amenant à des bit flips.

2.4.1 Moyens d'attaque

De ce point de vue, l'attaque électromagnétique devient donc particulièrement adaptée pour évaluer en attaque physique les systèmes. Nous nous sommes donc concentré sur cette méthode pour l'écoute et pour l'injection de faute dans les systèmes embarqués.

2.4.2 Surface d'attaque

La surface d'attaque représente tous les éléments par lesquels le courant pourrait fuiter et donc amener à une écoute. De la même manière, il peut entrer et donc amener à une perturbation.

Dans nos travaux, il s'agit de tout le circuit avec des éléments plus ou moins avantagés, comme le signal d'horloge ou les lignes d'alimentation qui traversent tout le circuit et proposent donc de grandes surfaces d'émission et de couplage.

Par conséquent, l'augmentation de surface des SoC par rapport aux μc, ainsi que l'augmentation du nombre de fonctions matérielles entrainent une augmentation du nombre transistors présents dans ces systèmes.

En reprenant l'exemple du μc ST, la finesse de gravure est de 130 nm alors que celle du A12 est de 7 nm. On se retrouve donc sur une surface plus grande, des transistors moins encombrants dans le cas du A12. Le nombre total de transistors est d'autant plus grand.

Cependant, dû au manque d'informations précises sur l'architecture à ce sujet, et afin de respecter notre modèle d'attaquant, il n'est pas possible de connaitre avec exactitude le nombre de transistors.

2.4.3 Choix des cibles

Pour ces travaux nous nous intéressons aux systèmes embarqués. Nous avons identifié qu'ils pouvaient être séparés en deux catégories.

Pour cela, notre but était de tester ces deux catégories. Dans un premier temps, concernant les μc, nous nous sommes tournés vers le STM32F100RB. Il embarque un cœur ARM Cortex-M3 qui implémente l'ISA ARMv7-M.

Pour le SoC, nous nous sommes tournés vers les cartes de développement Raspberry avec les Raspberry PI2B et PI3B embarquant respectivement un SoC Broadcom 2836 et 2837 implémentant respectivement les ISA ARMv7-A et ARMv8-A.

2.5 Conclusion

Dans cette partie, nous avons présenté les éléments que l'on retrouve dans les systèmes embarqués, en partant du transistor jusqu'à obtenir la microarchitecture. Nous avons également abordé les différentes

options de sécurité qu'intègrent ces systèmes.

Nous nous sommes attardé sur l'effet de ces éléments sur la tenue des attaques physiques. Et plus particulièrement, nous avons abordé les effets des options de sécurité qui par leur certification assurent que des tests ont été mené en amont, mais qui en contrepartie limitent le contrôle de l'utilisateur final sur son matériel.

À partir des éléments présentés ici, nous avons pu nous concentrer sur les attaques à mener. Nous avons également essayé de mettre en relation les différents éléments abordés ici avec les résultats des attaques que nous verrons par la suite.

Chapitre 3

Cas des attaques physiques sur *Microcontrôleur*

Résumé de la partie :

Dans cette partie, on va s'intéresser aux attaques physiques qui ont été mises en lumière avec pour cible les microcontrôleurs. Le but reste celui de démontrer que la sécurité dépend de la microarchitecture. Nous allons voir quelles ont été les attaques préalablement réalisées dans le cadre d'observations ou de perturbations. Par la suite, nous allons déterminer quels enjeux restent à résoudre dans le domaine des attaques physiques en analysant les axes sur lesquels se sont concentrés les attaques de la littérature, afin d'introduire notre contribution. Enfin nous terminerons, par notre contribution qui s'appuie sur des attaques en injection de faute sur microcontrôleur.

Sommaire

3.1 Attaques par observation

Des attaques par observation ont été étudiées sur les microcontrôleurs et les cartes à puces. Utilisant les canaux qui ont été présentés plus tôt (le temps d'exécution, la consommation d'énergie ou les émissions électromagnétiques notamment). Elles visaient des algorithmes de chiffrement [KS05, JFGEM17, BPT10, TOT+13] ou des applications non cryptographiques [LBBC+16].

Cette dernière attaque, en particulier a mis en lumière les faiblesses de la microarchitecture dans le cas des μc. En utilisant la méthode du profilage pour caractériser le comportement du système, en fonction de ses émissions électromagnétiques, cette attaque a permis de démontrer qu'il était possible de lier une écoute caractérisée au préalable à un comportement. Ainsi, il devient possible de retrouver des informations lorsqu'elles transitent dans le système.

Nous nous sommes donc tournés vers les attaques par perturbation pour ce type d'architecture.

3.2 Attaques en faute

Les μc ont également fait l'objet d'attaques en faute. Ces attaques ont majoritairement pour but la cryptanalyse [DMM+13]. Il s'agit d'attaques de type DOA ou IOA, qui vont cibler au choix un algorithme de chiffrement ou son implémentation.

Les travaux menés par Moro *et al.* [MDH+14] ont cependant porté sur le modèle de faute. Ainsi on retrouve la volonté de s'attaquer à la microarchitecture que nous avons souhaité explorer. Ses travaux, ont permis d'élaborer un modèle de faute sur la cible lors d'une injection de faute électromagnétique. Modèle que nous avons réutilisé dans notre contribution, utilisant la même cible. Ainsi dans ce cas, le modèle de faute décrit principalement deux effets, basés sur des bit-flips, que peuvent avoir une faute sur le système :

— Le premier sur le flot des instructions, une faute entraine une modification aléatoire du code d'une instruction. Cette modification peut avoir deux effets, entrainer l'apparition d'une instruction ayant un effet (avec effet de bord) sur le reste de l'exécution ou une instruction n'ayant aucun effet (sans effet de bord). Les instructions sans effet de bord permettent d'exploiter plus simplement les attaques en faute, puisque seul le modèle de faute du saut d'instruction sera présent.

— Le second apparait quant à lui au niveau du flot de données. Ainsi lorsqu'une instruction de type `load` est victime d'une faute, les valeurs et les adresses que cette instruction manipule peuvent être modifiées. Nous avons également pu constater ce phénomène lors de nos expérimentations.

3.3 Enjeux

Nous avons souhaité nous baser sur les travaux de Moro *et al.*, pour continuer dans la recherche de vulnérabilité. En effet, un modèle de faute a été identifié attestant d'un effet visible sur le système.

L'enjeu principal a donc été ici de réussir, à partir d'un modèle de faute, de mettre en lumière une vulnérabilité. Celle-ci nous permet de réaliser des attaques logicielles.

En parallèle, un autre enjeu a été de pouvoir reproduire les conditions pour l'injection de faute, afin de reproduire le modèle tel qu'il a été énoncé.

Nous avons voulu par cela :

— Démontrer la faisabilité d'une attaque logicielle ayant pour point d'entrée une faute.

— Démontrer que les vulnérabilités matérielles sont un problème à adresser rapidement, si l'on souhaite mettre en place une véritable chaine de confiance (RoT) qui assure que le système est sécurisé. Celle-ci devant reposer sur une microarchitecture dépourvue de vulnérabilité pour être efficace.

— Enfin démontrer que les μc sont vulnérables, rendant les IoT en général vulnérable à leur tour.

3.4 Contribution : mise en place de vulnérabilités au niveau logiciel par une injection de faute

La publication de la contribution présentée ci-après a été faite sous le titre *Let's shock our IoT's heart : ARMv7-M under (fault) attacks* [BLLL18] lors de la conférence «Availability, Reliability and Security» (ARES) en 2018.

3.4.1 Cible et cas testés

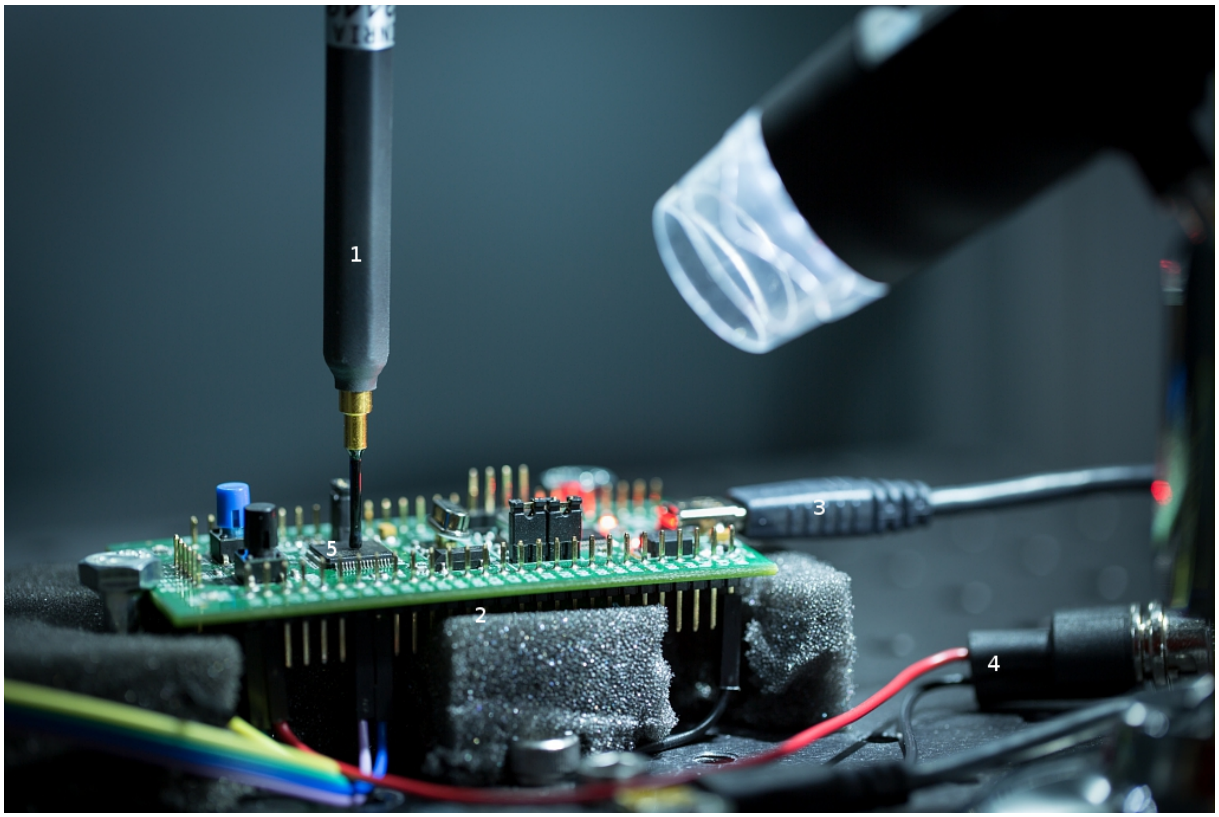

FIGURE 3.1 – Sonde d'injection disposée sur la carte de développement STM32. On note que la sonde n'est pas en contact mais se trouve immédiatement au dessus du cœur. [1] La sonde d'injection. [2] La carte de développement. [3] Alimentation et communication. [4] Signal de synchronisation. [5] Le cœur Cortex-M3.

Nous avons souhaité attaquer les processeurs présents majoritairement dans les systèmes de type IoT, il s'agit du modèle qui a été utilisé par Moro *et al.*.

Pour ce faire, nous nous sommes tournés vers la carte de développement STM32VLDISCOVERY. Elle est équipée d'un μc Cortex M3 conçu par ARM et suivant l'ISA ARMv7-M. La fréquence de fonctionnement de cette puce est de 24 MHz, le jeu d'instruction utilisé est le `Thumb-2`.

La communication avec cette carte se fait selon deux protocoles, le premier Universal Asynchronous receiver-transmitter (UART) nous a permis de communiquer avec le logiciel contenu dans la carte, il est possible ainsi d'afficher un menu, envoyer des commandes et recevoir des réponses au moyen d'un terminal capable de lire une liaison série. Le second protocole est le JTAG. Celui-ci nous a permis d'obtenir des

informations de diagnostic de la carte. Dans nos différentes expérimentations, il nous a permis de poser des points d'arrêt tel que l'exécution du programme s'arrête avant l'exécution d'une instruction spécifique, rendant alors possible l'observation des valeurs dans les registres ou la modification de ceux-ci en utilisant le logiciel OpenOCD.

Le montage expérimental est visible sur la Figure 3.1.

Pour cela, nous avons mis en place différents scénarios d'attaques visibles sur le tableau 3.1. Ces attaques logicielles ayant déjà été réalisées à de nombreuses reprises.

TABLE 3.1 – Scénarios d'attaques mis en place sur μc.

Scénario	**But visé**	**Exemple de cas d'utilisation**
Modification du flot d'exécution	COA	Obtenir un accès lorsqu'une vérification doit être effectuée
Dépassement de tampon	COA ou DOA	Remplacer une donnée inconnue par une donnée connue
Porte dérobée	COA ou DOA	Activation d'une porte dérobée ayant été placée par une personne malveillante dans le système
Réutilisation de code	COA ou DOA	Attaque en réutilisation de code

Pour réaliser les injections ciblées, nous avons dû nous synchroniser avec la cible. Ne s'agissant pas de l'objectif principal à ce moment, nous avons utilisé un signal envoyé par la carte pour nous permettre de localiser temporellement l'exécution du programme. Ceci nous a permis de cibler des points que nous avions identifiés comme sensibles.

Ce prérequis de synchronisation peut être remplacé par diverses méthodes (en utilisant une attaque en écoute par exemple). Cependant, ne s'agissant pas du cœur de ces travaux elles ont donc été ignorées pour la suite.

Second prérequis, une connaissance du code source ou une possibilité de le récupérer. Dans notre cas, nous avons disposé de celui-ci, mais la reproduction de l'attaque dans la vie réelle dépend de la connaissance de celui-ci. Il peut être obtenu par rétro-ingénierie du code machine extrait du système.

Pour le système attaqué, nous avons nous-même écrit le programme s'exécutant. Nous nous sommes placés dans une situation où l'utilisateur dispose d'un nombre réduit de fonctionnalités accessibles directement à savoir :

— Vérifier la validité d'un code PIN pour valider l'accès.

— Entrer un texte à chiffrer ou à déchiffrer.

— Demander le chiffrement ou le déchiffrement de ce texte.

— Lire le contenu du résultat de son chiffrement/déchiffrement.

De nombreuses autres fonctions, afin de mener à bien les services demandés, sont présentes dans le code source. Celui-ci est donc relativement conséquent, de l'ordre de 14000 lignes de code compilé.

3.4.2 Modification du flot d'exécution

Premier cas d'attaque : nous avons comme objectif de prendre un chemin différent du flot d'exécution prévu par le programme.

Description

Si l'on abstrait au niveau logiciel, notre objectif a été d'exécuter, dans une fonction conditionnelle, une condition normalement interdite.

Un exemple est celui d'un code PIN. Lorsque l'on entre un code PIN dans le système, une fonction de vérification va s'exécuter. Cette fonction va prendre en entrée le code que l'utilisateur va taper au clavier ainsi que le code présent en mémoire. Une comparaison des deux valeurs est effectuée, si elles sont identiques, la fonction répond avec une variable spécifiée par le concepteur pour signifier que l'accès est donné, et une autre valeur si les codes sont différents afin de refuser l'accès.

Données : Variable : différents
Résultat : Résultat de la comparaison : accès refusé/accepté
début;
si *différents* **alors**
  résultat_test = 0xFFFFFFFF;
sinon
  résultat_test = 0x55555555;
fin
retour;

Algorithme 1 : Pseudo-code illustrant la fonction attaquée.

Tel que présenté dans l'algorithme 1, en cas de différence, la valeur assignée est dans notre cas fixée à `0xFFFFFFFF` dans l'autre cas la valeur renvoyée est `0x55555555`.

Lorsque que cet algorithme est implémenté sur le système, il est compilé et en résulte le code assembleur visible sur le listing 3.1.

```
1 ldr r3, [pc, #32] ; r3 <- adresse de la variable "différents"
2 ldr r0, [pc, #28] ;
3 ldr r3, [r3, #0] ; r3 <- valeur contenue dans la variable "différents"
4 movs   r1, #128   ; 0x80
5 cmp r3, #1   ; r3 contient le résultat de la comparaison
6 ite eq       ; Instruction "si"
7 mov.w  r4, #4294967295   ; 0xffffffff --> le résultat est incorrect
8 mov.w  r4, #1431655765   ; 0x55555555 --> le résultat est correct
```

Listing 3.1 – Instructions après compilation

Faisabilité

En reprenant l'abstraction du modèle de faute, nous avons donc tenté de supprimer diverses instructions. Dans un premier temps, nous avons voulu valider que le modèle de faute abstrait était suffisant pour générer le comportement voulu. Nous avons donc modifié directement le code assembleur, afin de valider l'accès et donc retourner dans tous les cas la valeur `0x55555555`.

Cette étape facultative, nous a permis de valider immédiatement la faisabilité de notre attaque avant la mise en place effective de celle-ci.

Pour trouver la bonne modification dans le code assembleur, nous avons donc exécuté plusieurs fois la fonction. En testant à chaque fois une instruction différente afin de mettre en lumière laquelle pourrait générer le comportement voulu.

Il est apparu qu'il s'agissait de l'instruction `ite eq`. En effet, en supprimant cette instruction l'exécution continue en séquence et comme le registre de destination du résultat est le même sur les deux instructions ligne 7 et 8 sur le listing 3.2, la seconde valeur écrase la première et génère le résultat voulu. D'autres possibilités sont cependant apparues, suppression des instructions `ldr r3, [r3, #0]` ou `cmp r3, #1`, cependant ces possibilités nécessitent que le drapeau `eq` qui est mis à jour par l'instruction `cmp r3, #1` est nécessaire pour que l'instruction `ite eq` puisse s'exécuter de la manière voulue.

```
1  ldr   r3, [pc, #32]  ; (8000ad0 <test_persistence+0x30>)
2  ldr   r0, [pc, #28]  ; (8000acc <test_persistence+0x2c>)
3  ldr   r3, [r3, #0]
4  movs  r1, #128   ; 0x80
5  cmp   r3, #1   ; Le registre r3 contient le résultat de la comparaison
6  nop → suppression de ite eq
7  mov.w r4, #4294967295   ; 0xffffffff --> le résultat est incorrect
8  mov.w r4, #1431655765   ; 0x55555555 --> le résultat est correct
```

Listing 3.2 – Instruction fauté

Mise en place

Nous avons, mis en place notre système en tentant d'atteindre le modèle de faute. Connaissant à l'avance le comportement voulu, nous avons pu tester différents paramètres pour parvenir à sauter la bonne instruction.

Dans un premier temps, nous avons effectué un scan de la puce. Nous avons injecté à divers endroits des fautes pour observer le comportement de la puce en exécutant notre programme en boucle jusqu'à trouver une zone «sensible», cette zone selon les travaux de Moro *et al.* correspond aux points où des exceptions `Usage Fault` apparaissaient.

Nous avons également exploré différents paramètres qui sont également disponibles sur nos équipements : types d'impulsions, répétition de celles-ci, etc. Ces expérimentations préliminaires, nous ont permis de trouver des paramètres où le taux de faute exploitable nous a semblé suffisant. Il est difficile de trouver des paramètres «optimaux» pour l'injection. Un grand nombre d'entre eux peuvent être explorés de manière plus ou moins aléatoire. On note ainsi, les caractéristiques des équipements, les endroits et moments d'injection, etc. L'expérience de l'expérimentateur est donc un atout dans la recherche et l'exploration des paramètres. Nous ne certifions pas qu'il s'agisse des paramètres optimaux, mais dans notre cas, ils ont été suffisants pour déclencher le comportement souhaité.

À titre indicatif, la localisation qui a été choisie est présente sur la Figure 3.2.

Pour information les paramètres sur nos équipements ont été les suivants :

— Un train de 15 impulsions (avec une période de 3.1 ns et une largeur 1.6 ns).

— Chaque impulsion ayant un front montant de 1.6 ns.

— Une amplitude de -9dBm avant amplification.

— Une amplification de 175W fixée par l'amplificateur.

Ces paramètres ont, sauf mention contraire été conservés sur les autres cas qui ont été testés sur cette architecture.

Résultat

Avec ces paramètres, nous avons été capables de générer des fautes. Parmi ces comportements fauté, nous avons obtenu des comportements non-prévus entrainant des crashs du système et des fautes générant le comportement voulu.

Ainsi, 10% des exécutions totales, ont mené à un saut de l'instruction ciblée et donc à une validation de l'accès alors qu'il n'aurait pas du être validé.

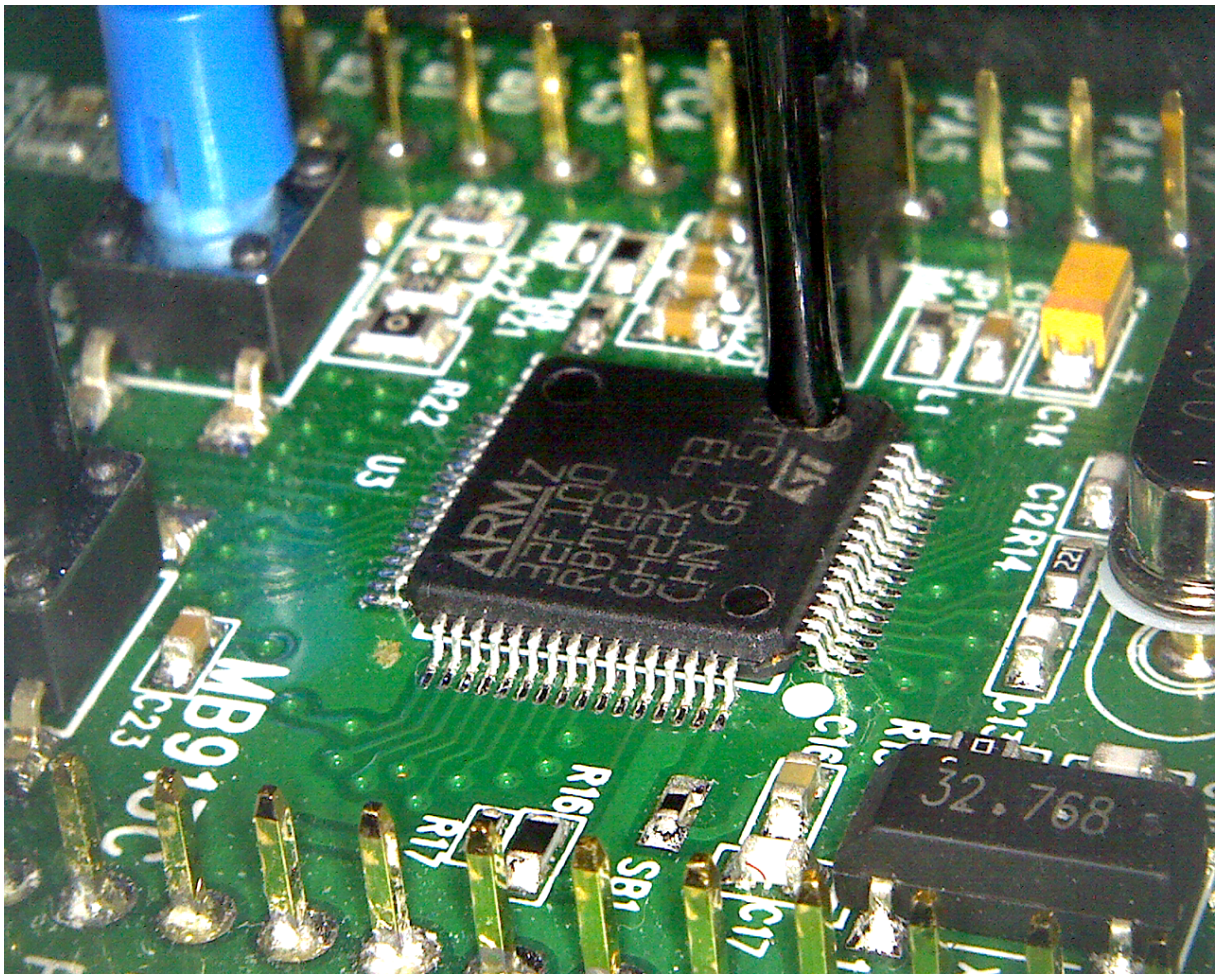

FIGURE 3.2 – Position de la sonde sur la puce.

Conséquences

Ce cas simple nous a permis de valider la faisabilité des attaques en fautes sur μc, démontrant que les fautes peuvent être un point d'entrée pour une attaque de type COA sur ce type de système.

Nous avons été ici capables de modifier le flot de contrôle d'un programme, cependant le cas du code PIN que nous avons pris en compte est déjà connu et les algorithmes de vérification de type VERIFY PIN apportent des contremesures. On retrouve par exemple, la duplication des tests de similarité, dans ce cas au moins deux injections de fautes seront nécessaires. On retrouve également, des algorithmes de vérification qui imbriquent plusieurs tests, dans ce cas il devient très difficile avec une seule faute d'affecter le déroulement normal du programme sans causer d'effets de bord importants.

Notre proposition est cependant d'étendre ces tests à toutes les fonctions avec des branchements conditionnels, en effet tout autre programme utilisant ce type d'instruction est vulnérable face à ce genre d'attaque.

3.4.3 Dépassement de tampon

Description

Dans ce cas, notre but a été de modifier une valeur présente en mémoire sans avoir accès à la zone mémoire en question. Par exemple, une clef cryptographique utilisée dans un système.

En modifiant cette valeur, il nous devient possible de connaitre la clef de chiffrement puisque nous la choisissons nous-même.

Pour reproduire un cas similaire, nous avons mis en place une attaque afin d'écraser une valeur de clef se trouvant sur notre système. Le dépassement de tampon est une attaque connue et un des moyens de protection habituellement mis en place est celui qui consiste à placer un «canari». Il s'agit d'une valeur secrète placée en amont d'une variable que l'on souhaite protéger. En cas de dépassement de tampon, celle-ci sera écrasée. Il suffit alors de vérifier l'état de ce canari pour savoir si une attaque de type dépassement de tampon a été ou non lancée.

L'implémentation de notre application laisse apparaitre deux dispositions de la mémoire : une pour notre premier cas DT1 (voir listing 3.3) avec le tampon de mémoire du texte à manipuler placé avant le tampon de mémoire contenant la clef. On peut ainsi voir l'état de la mémoire sur le listing 3.4.

```
1  char text[128];
2  char key[16];
3  ...
4  char big_buffer[256];
```

Listing 3.3 – DT1, déclaration des variables usuelles.

```
1 0x20000918 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx --> début du tampon de text
2 0x20001fa0 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
3     ...    --> suite du tampon
4 0x20000a18 33221100 77665544 bbaa9988 ffeeddcc --> début du tampon de la clef
5 0x20000a28 yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy --> début du tampon big_buffer
```

Listing 3.4 – État de la mémoire pour DT1, les valeurs x et y ne représentent pas de valeur spécifique.

Une pour le second cas DT2 (voir listing 3.5) où les deux tampons, sont séparés par une zone contenant le canari, comme visible sur le listing 3.6.

```
1   char text[128];
2   char canari[16];
3   char key[16];
4   ...
5   char big_buffer[256];
```

Listing 3.5 – DT2, déclaration des variables avec canari.

La fonction ciblée est ici *strncpy* (voir listing 3.7). Cette fonction copie le nombre d'octets spécifié d'un tampon A vers l'adresse d'un tampon B.

Dans notre cas, la valeur a été fixée à 128 octets, représentant la taille maximale du tampon de texte manipulé par l'utilisateur. Ainsi, il n'est pas censé pouvoir écrire sur une plus grande zone. Par conséquent, les variables déclarées après ne pourront pas être infectées, rendant théoriquement impossible le dépassement de tampon. Et ce peu importe la taille de *big_buffer*.

```
0x20000918 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx --> début du tampon de text
0x20001fa0 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
    ...    --> suite du tampon
0x20000a18 zzzzzzzz zzzzzzzz zzzzzzzz zzzzzzzz --> tampon du canari
0x20000a28 33221100 77665544 bbaa9988 ffeeddcc --> tampon de la clef
0x20000a38 yyyyyyyy yyyyyyyy yyyyyyyy yyyyyyyy --> tampon big_buffer
```

Listing 3.6 – État de la mémoire pour DT2, les valeurs x, y et z ne représentent pas de valeur spécifique.

```
char * strncpy(text, big_buffer, 128);
```

Listing 3.7 – Prototype de la fonction *strncpy*

Faisabilité

De la même manière que précédemment, nous avons vérifié la faisabilité, en tentant de supprimer différentes instructions dans la fonction *strncpy*.

Après nos test, il est apparu que la faiblesse ne se situait pas immédiatement dans la fonction *strncpy* mais dans son appel tel que visible dans l'extrait de code 3.8.

```
mov r2, r5              ; passage de l'argument valeur (dans r2)
mov r0, r6              ; tampon de destination (text) (dans r0)
ldr r1, [pc, #24]       ; tampon de la source (big_buffer) (dans r1)
bl  8004770 <strncpy>   ; call strncpy procedure
```

Listing 3.8 – Appel de la fonction *strncpy*

En effet, les arguments sont passés un-à-un dans les différents registres pour satisfaire la convention d'appel d'ARM, à savoir que les valeurs nécessaires à l'exécution d'une fonction sont remplies par la fonction appelante. En supprimant un de ces passages de paramètres on se retrouve dans un état imprédictible, en effet, la valeur dans le registre $r2$ devient alors dépendante de l'exécution courante du programme et n'est plus contrôlable.

```
nop → suppression du chargement dans le registre r2
mov r0, r6              ; tampon de destination (text) (dans r0)
ldr r1, [pc, #24]       ; tampon de la source (big_buffer) (dans r1)
bl  8004770 <strncpy>   ; appel de la fonction strncpy
```

Listing 3.9 – Appel de la fonction *strncpy*

Dans le cas où cette valeur est plus grande que la valeur de base (128) alors le dépassement de tampon est possible. Dans le cas DT2, un dépassement de tampon est visible en vérifiant l'état du canari.

Mise en place

Pour reproduire la faute, nous avons conservé les mêmes paramètres que pour l'attaque CFH.

Résultats

En premier lieu, nous avons pu confirmer que la protection qui consiste à spécifier une valeur maximale directement dans le code, n'est pas une protection suffisante puisque nous avons pu modifier la valeur dans le registre `r2`. Cependant, un problème s'est posé à nous, la valeur de `r2` dépend de l'exécution du reste du système. Et dans notre cas, `r2` avant son utilisation contenait une adresse donc sous la forme `0x2000xxxx`. Nous nous sommes référés ensuite à l'extrait de documentation suivant :

Extrait de la documentation de la fonction *strncpy*

... If the end of the source C string (which is signaled by a null-character) is found before num characters have been copied, destination is padded with zeros until a total of num characters have been written to it ...
Si la fin du tampon source est atteinte (signalée par un caractère null) avant la valeur du paramètre num, alors le tampon de destination est rempli, avec des zéros, jusqu'à atteindre la valeur spécifiée par ce paramètre ...

Notre système a donc subit un effacement de mémoire, celle-ci étant remplacée pour une très grande partie par des zéros, ce qui a eu pour conséquence de provoquer des crashs. Ces crashs étant impossibles à analyser au vu des accès offerts sur notre puce, nous avons émis l'hypothèse qu'il s'agissait probablement d'écriture dans des zones mémoires interdites qui provoquaient le crash. Les instructions n'ayant pas pu être touchées, le code se situant en amont de l'adresse d'écriture.

Pour l'attaque DT1, nous avons été en mesure de modifier la valeur de la clef pour y placer une valeur qui nous était connue, celle présente dans le tampon *big_buffer*.

Pour l'attaque DT2, dans la majorité des cas, nous avons été en mesure, par le biais de cette protection, de détecter le dépassement de tampon. Néanmoins, quelques cas ont montré une modification partielle de la clef sans que le canari ne soit modifié. Dans ces cas, le modèle de faute correspondait à celui de la modification du flot de données tel que présenté dans les travaux de Moro *et al.* Cependant, notre injection se concentrait sur l'instruction `mov r2, r5` tandis que l'effet observé dans les travaux de Moro *et al.* correspondait à des instructions `ldr`. Dans ce cas, nous avons tout de même été en mesure de modifier la clef ce qui permet de confirmer que le canari n'est pas une protection suffisante. Cependant, une modification partielle a été constatée, elle ne correspond pas au modèle de faute tel qu'il a été présenté plus tôt. La raison exacte nous est inconnue en absence d'expérimentations supplémentaires.

Conséquences

Avec ces deux attaques, nous avons démontré être en capacité d'injecter des fautes et de générer des comportements menant à des vulnérabilités. Cependant, les résultats observés lors de l'expérimentation DT2 ont montrés que le modèle de faute pouvait être élargi à un nouveau comportement.

3.4.4 Activation d'une porte dérobée

Notre troisième cas est une porte dérobée (backdoor en anglais), il s'agit d'une portion de code inaccessible en temps normal et masquée dans le code source.

Description

Dans ce cas, nous avons souhaité activer une porte dérobée qui aurait été placée dans le code source au préalable. Dans ce cas, on sort un peu du cas de l'attaquant MATE comme il a été présenté. Mais des

portes dérobées étant découvertes dans tous types de systèmes [1], nous avons souhaité mettre en lumière ce cas.

Dans notre cas, la fonction de la porte dérobée est de copier la valeur de la clef dans le tampon du texte chiffré. L'utilisateur ayant pour seule fonction accessible celle d'afficher la valeur du chiffré, il peut ainsi obtenir la valeur de la clef.

Dans ce cas, nous avons dû concevoir la porte dérobée et l'intégrer au code. Nous avons également conçu une fonction (voir listing 3.10) qui permet l'entrée dans la porte dérobée. Le but de cette fonction est de faire clignoter une LED, pour ce faire un signal est généré après un certain temps d'attente spécifié par la variable `wait_for`. Nous avons pris le même genre de conception qui pourrait être utilisé par un

```
1  void blink_wait()
2  {
3    unsigned int wait_for = 3758874636;
4    unsigned int counter;
5    for(counter = 0; counter < wait_for; counter += 8000000);
6  }
```

Listing 3.10 – Fonction légitime contenant un appel masquée à la porte dérobée

attaquant. À savoir, placer notre porte dérobée quelque part dans le code source, ne placer à aucun endroit un appel à cette fonction et masquer cet appel. Pour masquer cet appel nous avons utilisé une spécificité

```
1  08000598 <blink_wait>:
2    push   {r7, lr}
3    sub    sp, #8
4    add    r7, sp, #0
5    ldr    r3, [pc, #44]  ; ( = 80005cc)
6    ...
7    adds    r7, #8
8    mov    sp, r7
9    pop    {r7, pc}
10   .word    0xe00be00c   ; 3758874636 (@80005cc)
```

Listing 3.11 – Code assembleur de la fonction légitime contenant l'appel à la porte dérobée

du compilateur ARM. En effet, les variables locales sont sauvegardée à la suite du code assembleur de la fonction, il s'agit du *«literal pool»*. Dans notre cas, y sera placé la valeur d'attente spécifiée précédemment dans la variable `wait\_for`.

```
1    .word    0xe00be00c   ; 3758874636 (@80005cc)
```

Listing 3.12 – Contenu du *literal pool*

Cette valeur est importante car une fois décomposée, elle correspond à deux instructions de saut (voir listing 3.13). Pour simplifier nos tests, nous avons utilisé des valeurs fixes pour la clef et pour le texte à chiffrer. Le texte chiffré résultant est donc le même. Ainsi, en cas d'exécution correcte, la valeur de la clef est `00112233445566778899AABBCCDDEEFF`, le texte à chiffrer est `000102030405060708090a0b0c0d0e0f` et le chiffré correspondant `279fb74a7572135e8f9b8ef6d1eee003`.

1. https ://forum.xda-developers.com/general/security/xiaomi-firmware-multiple-backdoords-t2847069

```
1  b backdoor     ; 0xe00b
2  b backdoor     ; 0xe00c
```

Listing 3.13 – Exécution de la valeur wait_for

Faisabilité

Pour ce cas, on a vu qu'il était possible de masquer des instructions dans des variables. Pour étudier la faisabilité de cette attaque il a fallu valider trois étapes :

— Possibilité d'atteindre les instructions masquées.

— Possibilité d'exécuter ces instructions.

— Possibilité de passer dans la fonction de la porte dérobée.

Toutes les trois, par le biais de l'abstraction du modèle de faute correspondant à un `nop`. Après avoir testé de nouveaux plusieurs instructions nous avons validé le comportement voulu lorsque nous avons pu remplacer l'instruction de retour juste avant le *literal pool*, le résultat est visible sur le listing 3.14. Le remplacement des autres instructions ne nous a pas permis d'atteindre le même comportement.

```
1  08000598 <blink_wait>:
2    ...
3    adds   r7, #8
4    mov    sp, r7
5    nop → suppression de l'instruction de retour
6    b backdoor → 0xe00b
7    b backdoor → 0xe00c
```

Listing 3.14 – Exécution de la valeur wait_for après une faute

Mise en place

Pour reproduire la faute, nous avons conservé les mêmes paramètres que pour l'attaque CFH.

Résultat

Nous avons obtenu deux résultats distincts.

Le premier se situait d'un point de vue temporel à une injection 2.116μs après notre signal de synchronisation. Dans ce cas, c'est bien le *literal pool* qui a été exécuté. En effet, nous y avions placé un point d'arrêt afin de nous assurer que le programme passait bien par là. De plus, nous avons été capables d'observer la valeur de la clef dans le tampon du texte chiffré à savoir `00112233445566778899AABBCCDDEEFF`. Ce résultat était donc celui qui se rapproche de l'abstraction du modèle de faute.

Le deuxième résultat fauté se situait quant à lui plus tard temporellement, à 2.268μs. La valeur qui y a été observée est `279FB74A445566778899AABBCCDDEEFF`. On remarque qu'il s'agit d'un mélange de la clef de chiffrement et du texte chiffré. De la même manière que pour le cas précédent, cette modification se rapproche de ce qui a été énoncé concernant le modèle de faute sur le flot de donnée, or dans ce cas nous avons ciblé une instruction différente du `load`.

Conséquences

Ce cas nous a permis de montrer qu'une portion de code malveillante pouvait être masquée dans un code source qui paraitrait sain lors d'une évaluation via le flot d'exécution ou d'autres outils d'analyse statique de code. Nous avons ici, voulu faire un rappel sur le problème de l'exécution de données et montrer que le *literal pool*, spécifique à ARMv7-M pouvait devenir une source de vulnérabilité dans le cas d'attaques par injection de faute. Cependant, le résultat hybride obtenu lors de certaines expérimentations laisse supposer que le modèle de faute énoncé sur le flot de contrôle et sur le flot de données n'est pas le seul comportement.

3.4.5 Réutilisation de code

Dernier cas de test, l'attaque par réutilisation de code. Nous nous sommes ici focalisé sur la variante ROP, cette variante réutilise le code en utilisant des gadgets contenus dans celui-ci. La particularité de ces gadgets est qu'ils se terminent par un retour. L'adresse de retour peut provenir de la pile d'exécution, ou d'un registre spécifique le `lr` selon les cas de gadget.

Description

Le but de cette attaque est de prendre le contrôle du système, en effet l'attaquant va concevoir un flot d'exécution qui va se superposer au flot normal dans le but de réaliser de nouvelles fonctionnalités.

L'utilisateur va utiliser des «gadgets», il s'agit de portions de code de petite taille ayant un comportement qui sera utile pour l'attaquant et se terminant par une instruction de retour.

Dans ARMv7-M, 2 instructions permettent d'effectuer un retour : l'instruction `bx lr` et l'instruction `pop {...,pc}`. Dans le premier cas, le registre `lr` contient l'adresse de retour dans le second celle-ci est récupérée depuis la pile. Un exemple de gadget (voir listing 3.15) laisse apparaitre que pris à part cette suite d'instruction aura pour effet de charger le registre `r0` avec la valeur 0, et de modifier la valeur du registre `r1` avant de charger la première valeur de la pile comme prochaine instruction à exécuter. Dans

```
1 800039c     mov r3, #0
2 800039e     add r1, r1, r3
3 80003a0     mov r0, r3
4 80003a2     pop {pc}
```

Listing 3.15 – Exemple de gadget

ce cas, si l'attaquant souhaite utiliser uniquement l'affectation de valeur au registre `r0` il peut envisager d'utiliser ce gadget, en trouvant un moyen de charger l'adresse de celui-ci (800039c) dans son registre `pc`. Néanmoins, dans ce gadget on constate également la présence d'une instruction `add r1, r1, r3`, celle-ci provoque un effet de bord en modifiant également la valeur dans le registre `r1`, l'attaquant devra donc être minutieux dans son choix de gadget afin de contrôler les effets de bords qui en résultent ou faire en sorte qu'il n'y en ai pas.

Afin de réaliser cette attaque, plusieurs étapes sont nécessaires :

1. Un code source conséquent est recherché pour y trouver les gadgets nécessaires.
2. Ensuite l'attaquant doit s'assurer de la viabilité des gadgets disponibles afin d'éliminer ou de contrôler les effets de bords.
3. Puis, il doit également s'assurer que la pile contient la configuration nécessaire à l'exécution de son ou de ses gadgets.

4. Enfin, l'attaquant doit être capable de prendre le contrôle de la pile en exécutant une adresse se trouvant en son sein pour lancer son attaque.

Le but de cette attaque a été de prendre le contrôle du système, en lui faisant exécuter un code permettant d'avoir un accès à la valeur de la clef. Plus précisément, nous avons voulu copier celle-ci dans le tampon du texte chiffré auquel nous avons un accès libre, par le biais de la commande d'affichage du texte chiffré.

Faisabilité

Dans notre cas, le code assembleur est relativement conséquent. Nous avons pu y trouver suffisamment de gadgets nécessaires. Pour les identifier, nous avons utilisé un outil d'analyse de code `ROPgadget` [2]. Une analyse manuelle nous a permis de les sélectionner afin de mettre en place le comportement voulu. Cependant, un gadget était manquant, nous l'avons donc ajouté nous-même. Il est néanmoins évident que dans des systèmes avec des fonctionnalités plus diverses le gadget aurait pu être trouvé ou remplacé par un enchainement de plusieurs gadgets.

Ce gadget (voir listing 3.16) nous permet de sauvegarder le registre `lr` avec la valeur voulue. Il est obligatoire dans ce cas, car la fonction que nous avions identifiée comme étant sensible utilisait comme fonction de retour le `bx lr`, et que dans le cas du ROP nous ne disposons que d'un contrôle sur la pile. Afin d'assurer un retour correct après l'exécution de notre comportement malveillant, nous y avons donc placé une adresse. Dans un cas réel où un attaquant n'aurait guère d'intérêt à conserver le comportement normal du système après son intrusion cette étape est cependant optionnelle, l'instruction `pop {r0, r1, r2, pc}` étant disponible par ailleurs. Il a fallu relier plusieurs gadgets afin de parvenir au

```
1       ...
2  080003e2 <gadget_supplémentaire>:
3   80003e2:  4c01    ldr r4, [pc, #4] ; (80003e8)
4   80003e4:  46a6    mov lr, r4
5   80003e6:  bd07    pop {r0, r1, r2, pc}
6   80003e8:  0800039d .word 0x0800039d
7       ...
```

Listing 3.16 – Gadget ajouté manuellement

point de copier la clef, en effet vu le comportement recherché, nous avons souhaité exécuter une fonction déjà présente le `memcpy` (listing 3.17).

Le choix de cette fonction a été motivé par le fait qu'elle se terminait par la fonction de retour `bx lr`, ainsi à l'aide de notre gadget supplémentaire il est possible de retourner à l'état normal d'exécution tout en étant transparent du point de vue de l'utilisateur légitime, ou des protections logicielles. Selon la

```
1  void * memcpy(tampon_du_chiffré,tampon_de_la_clef,taille_a_copier);
```

Listing 3.17 – Prototype de la fonction *memcpy*

convention d'appel, les adresses des différents tampons doivent aller dans les registres consécutifs :

— `r0` = adresse du tampon du chiffré.

— `r1` = adresse du tampon de la clef.

2. https://github.com/JonathanSalwan/ROPgadget

— `r2` = taille des données à copier.

Le but va donc être de remplir les différents registres et d'exécuter la fonction `memcpy` afin d'obtenir la clef simplement. La recherche de gadgets s'est donc orientée sur ces éléments.

Après avoir trouvé les gadgets, nous avons dû mettre la pile dans un état permettant le lancement de notre attaque. Il s'agit d'une étape importante puisqu'elle conditionne l'utilisation des gadgets. Cet état de la pile (voir listing 3.18) a été obtenu à la suite de l'exécution du système pendant un certain temps. Plusieurs fonctions ont été identifiées qui ont permis d'atteindre après des lancements successifs, cet état.

```
0x20001f70 xxxxxxxx xxxxxxxx xxxxxxxx 080008bd
0x20001f80 xxxxxxxx xxxxxxxx 20000ac8 000000f3
0x20001f90 xxxxxxxx 080020c1 0000ffff xxxxxxxx
0x20001fa0 20000a1c 08007f0b xxxxxxxx xxxxxxxx
0x20001fb0 0800039d 00000010 080003a3 08005e81
0x20001fc0 40011000 00000000 00000064 00000001
0x20001fd0 20000a74 20000ab4 40011000 20000ab4
0x20001fe0 08008ba9 20000ab4 00000010 00000010
0x20001ff0 00000010 20000ab4 08008c33 57d49195
```

Listing 3.18 – État de la pile rendant possible le ROP, les valeurs en *xxxxxxxx* sont inutiles et n'ont pas d'effet de bord. Elles servent de remplissage et permettent de mettre en lumière les valeurs utiles.

Une fois la pile dans la bonne configuration il faut donc en prendre le contrôle. Dans notre cas, la prise de contrôle passe par l'injection d'une faute. Ainsi, comme dans les cas précédents, nous sommes partis de l'abstraction logicielle du modèle de faute pour trouver notre réponse, ainsi comme visible sur le listing 3.19 notre premier gadget se trouvait immédiatement à la suite d'une fonction. Pour le joindre,

```
       ...
080003dc <fonction quelconque>:
 80003dc:  e006    b.n 80003ec <autre_fonction>
       ...
080003e2 <premier gadget>:
       ...
```

Listing 3.19 – Appel du premier gadget

il nous faut donc selon notre modèle de faute remplacer l'instruction terminale de la fonction quelconque par un `nop`. Nous avons été capables ainsi de manière «logicielle» de recréer le lancement du premier

```
       ...
080003dc <fonction quelconque>:
 80003dc:  nop → suppression de b.n 80003ec
       ...
080003e2 <premier gadget>:
       ...
```

Listing 3.20 – Activation du premier gadget

gadget. Par la suite nous avons tenté de reproduire le même comportement avec une faute, mais nous avons également souhaité finaliser la prise de contrôle qui consiste à être capable de copier la valeur de la clef dans le tampon du texte chiffré.

Mise en place

Pour reproduire la faute, nous avons conservé les mêmes paramètres que pour l'attaque CFH.

Résultat

Pour valider le parcours de notre chaine de gadget, nous avons mis en place un certain nombre de points d'arrêts. Ainsi, nous avons pu suivre le déroulement de l'attaque tel qu'il était prévu. Sur les figures suivantes, on retrouvera sur la partie gauche la pile avec le pointeur de pile au début de l'exécution (`sp\_i`) et le pointeur en fin de gadget (`sp\_f`). Au centre ce sera l'état des registres manipulés par nos différents gadgets. Enfin sur la droite, un rappel de l'état des registres `r0, r1, r2` nécessaire au lancement de la fonction de copie.

```
1       ...
2  080003e2 <premier_gadget>:
3      ldr r4, [pc, #4] ; ( 0x0800039d )
4      mov lr, r4
5      bd07    pop {r0, r1, r2, pc}
6     .word 0x0800039d
7       ...
```

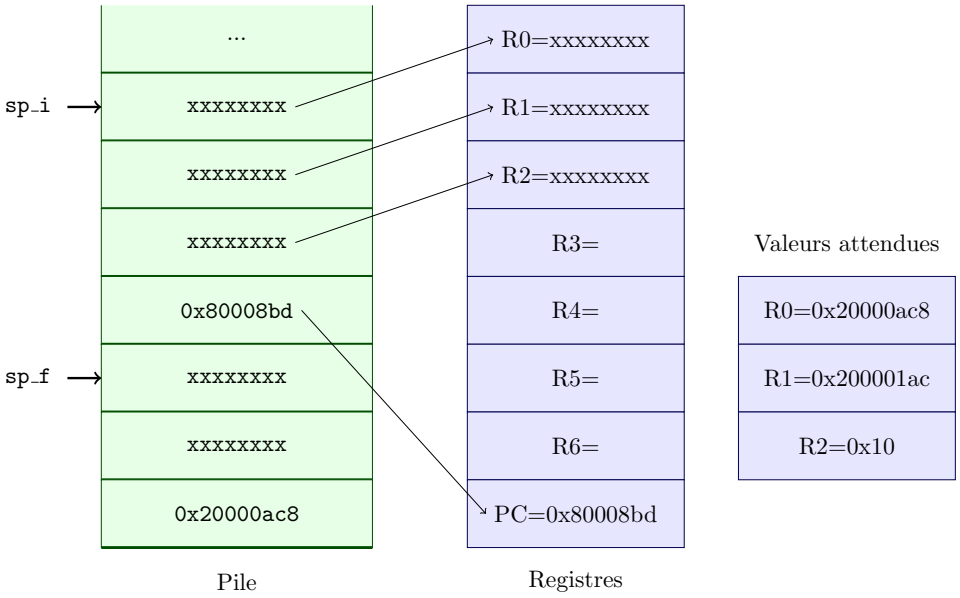

FIGURE 3.3 – 1er gadget et état du système correspondant à son exécution

Le rôle du premier gadget est de sauvegarder le registre `lr`. Après l'exécution de celui-ci on obtient l'état de la pile tel que visible sur la Figure 3.3. Sur cette figure, la pile se situe à gauche, au centre c'est l'état courant des registres. Sur la droite, on retrouve les valeurs que l'on souhaite obtenir dans les registres pour rappel.

Le deuxième gadget (voir listing 3.4), a pour fonction de déplacer le pointeur de pile et charger de nouveau des valeurs depuis la pile vers des registres spécifiés. On constate que la valeur `0x20000ac8` nécessaire pour alimenter le `r0` apparait dans un autre registre (`r4`). Ensuite, il effectue un nouveau saut,

```
080008bc <deuxième_gadget>:
       ...
   add sp, #8
   pop {r4, r5, r6, pc}
       ...
```

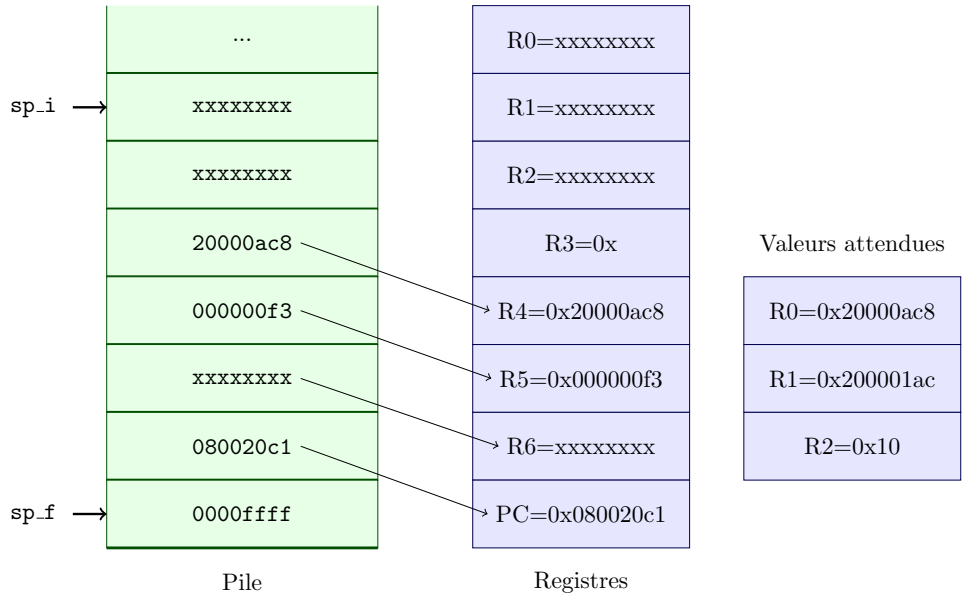

FIGURE 3.4 – 2ème gadget et état du système correspondant à son exécution

en récupérant depuis la pile une valeur qui va alimenter le registre `pc`, on voit l'état de ces différents registres sur le listing 3.4. Cette valeur permet de lancer le troisième gadget.

La fonction de ce troisième gadget 3.5 est de copier la valeur contenue dans le registre `r4` vers le registre `r0`, la valeur `20000ac8` se retrouve donc au bon endroit. On constate également l'apparition de la valeur `0x20000a1c` dans le registre `r5`. Ce gadget, se termine également par un chargement d'autres registres puis par un saut vers une adresse dans la pile permettant de lancer le quatrième gadget.

La fonction de ce quatrième gadget est de copier la valeur du registre `r5` dans le registre `r1`, ainsi la valeur `0x20000a1c` se retrouve au bon endroit. De nouveaux des variables, sont chargées depuis la pile.

La Figure 3.7, montre l'état du dernier gadget que nous avons utilisé. Celui-ci nous permet d'obtenir la copie de la clef par l'appel à la fonction `memcpy`.

Conséquences

Ce cas ne respecte pas totalement le modèle d'attaquant que nous avons sélectionné. En particulier, nous avons dû créer un gadget. Il relève donc davantage de la preuve de concept que d'une attaque transposable telle quelle dans le monde réel. Son rôle était de mettre en lumière que même des attaques complexes comme la réutilisation de code pouvaient être de réelles menaces pour les systèmes embarqués.

Un aspect de la dangerosité de ces attaques qui n'a pas été montré ici est également, que le nouveau flot d'exécution crée hérite de toutes les autorisations d'accès de la première fonction qui l'a appelé, ainsi les niveaux de privilèges sont conservés.

```
080020c0 <troisième_gadget>:
       ...
   mov r0, r4
   pop {r3, r4, r5, pc}
       ...
```

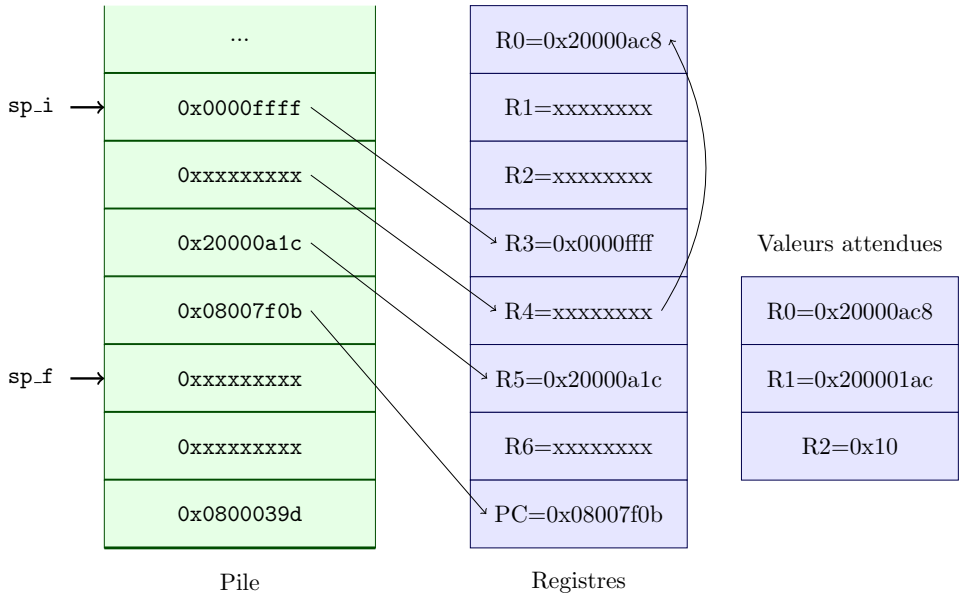

FIGURE 3.5 – 3ème gadget et état du système correspondant à son exécution

Nous avons pu prouver qu'une attaque relevant d'une seule faute bien placée pouvait permettre de prendre totalement le contrôle d'un système, sans qu'il soit possible de mettre en place une quelconque protection logicielle.

3.5 Conclusion

Cette partie a porté sur l'étude des attaques physiques sur μc. Un état de l'art a d'abord été effectué sur les attaques visant ces systèmes, qu'il s'agisse d'attaques par observation ou d'attaques par injections de fautes. Ensuite, une présentation de notre contribution concernant l'exploitation des attaques en fautes a été effectué. Nous avons ainsi été en mesure d'utiliser les injections de fautes comme vecteur d'introduction de vulnérabilités permettant de lancer des attaques logicielles.

Ces vulnérabilités n'épargnent pas les fonctions cryptographiques au même titre que les logiciels généralistes. D'après nos observations, les protections se focalisent sur la partie logicielle des systèmes en accordant une pleine confiance en la microarchitecture, celle-ci par les vulnérabilités qu'elle présente, rend donc ces protections insuffisantes dans le domaine des μc.

Il nous parait donc nécessaire d'envisager des techniques de protection de la microarchitecture dans le cas des μc. Il est important de noter que tous les systèmes utilisant des μc sont concernés. C'est le cas des systèmes de type IoT par exemple qui les utilisent en tant que cœur de calcul, mais c'est également le cas des SoC.

Une illustration est faite par Tang *et al.* dans [TSS17] où ils démontrent qu'il est possible de prendre

```
08007f0a <quatrième_gadget>:
       ...
   mov r1, r5
   pop {r4, r5, pc}
       ...
```

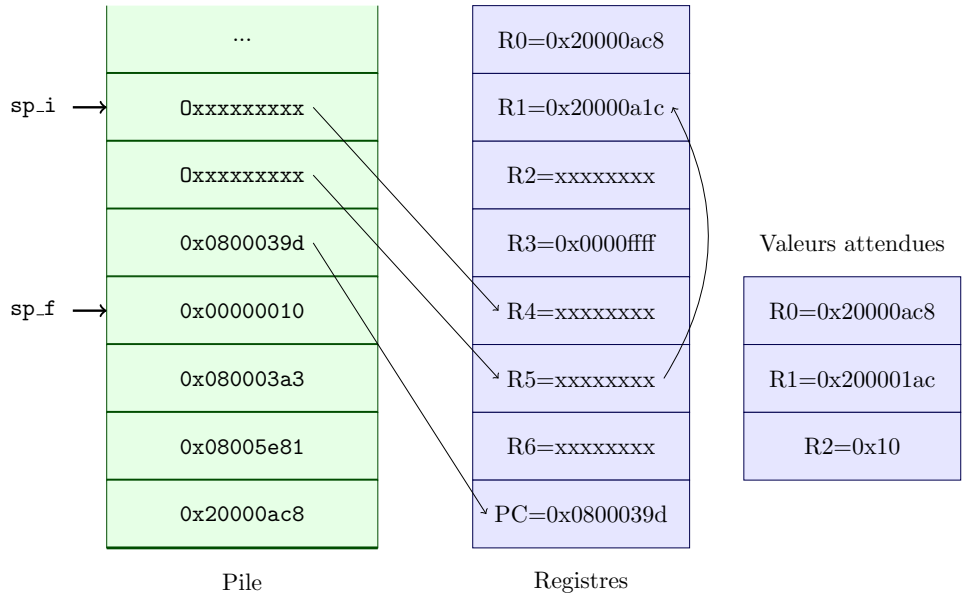

FIGURE 3.6 – 4ème gadget et état du système correspondant à son exécution

le contrôle du processeur principal d'un SoC en prenant le contrôle du μc en charge de la gestion de l'énergie. Cette attaque sera plus détaillée dans la partie suivante. Dans leurs travaux, le μc utilisé dispose des mêmes cœurs Cortex-M3 que celui de nos travaux. Ce même cœur est d'ailleurs présent dans un grand nombre d'appareils mobiles utilisés à ce jour.

Démonstration faite de la vulnérabilité de la microarchitecture dans le cas des μc, nous avons voulu nous concentrer sur les SoC. Ces derniers, comme on vient de le voir, sont sensibles par le biais des μc qu'ils embarquent. Cependant, nous avons voulu tester si la microarchitecture permettait de réaliser autant d'attaques que dans le cas des μc.

```
0800039c <cinquième_gadget>:
       ...
   pop {r2, r5}
   mov lr, r5
   pop {pc}
       ...
```

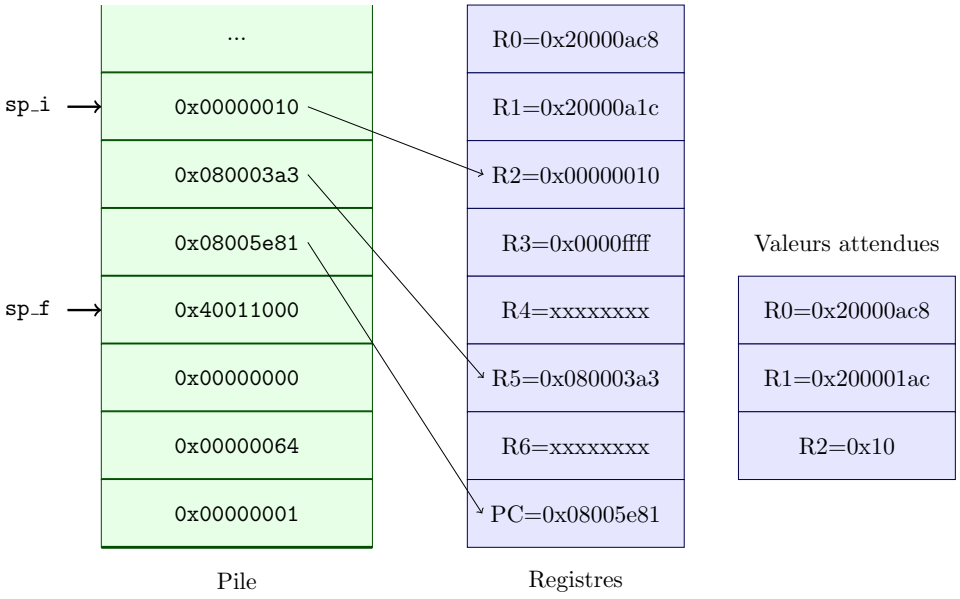

FIGURE 3.7 – 5ème gadget et état du système correspondant à son exécution

Chapitre 4

Cas des attaques par observation sur *System-On-Chip*

Résumé de la partie :

Dans cette partie, nous allons nous intéresser aux attaques physiques ayant pour cible des SoC. Plus précisément, nous allons nous intéresser aux attaques en écoute. Dans un premier temps, nous ferons un récapitulatif des attaques existantes et des caractéristiques qui étaient ciblées par ces attaques dans les SoC. Ensuite, nous présenterons notre contribution dans ce domaine.

Sommaire

Nous allons maintenant nous intéresser aux attaques visant les SoC, ces derniers partagent des caractéristiques avec les μc. Les attaques vues précédemment sont donc toujours d'actualité. Des différences les caractérisent néanmoins. Ces différences qui ont un effet sur le taux de réussite et sur la tenue des attaques. Parmi ces différences on peut citer, la complexité croissante de l'architecture, du jeu d'instruction en résultant, la vitesse de fonctionnement bien supérieure, ou encore l'existence de processeurs multicœurs.

Ces différences nous ont conduit à évaluer différentes surfaces d'attaques. Dorénavant, nous avons souhaité distinguer les attaques en fonction de leur cible. On peut ainsi distinguer :

— les attaques ayant pour cible un composant interne. On considérera ainsi toutes les vulnérabilités que peuvent entrainer la présence d'un composant non contrôlé directement par le logiciel au moment de l'apparition de la vulnérabilité.

— les attaques sur la microarchitecture du processeur principal. Ici, on va considérer les vulnérabilités venant directement du (ou des) cœur exécutant le logiciel. Ainsi on considèrera de la même manière, que ce que l'on a pu retrouver dans les μc.

— les attaques sur les communications entre ces éléments.

Dans cette partie, nous avons souhaité nous concentrer sur les architectures les plus diffusées actuellement à savoir celles d'ARM. Pour les travaux que nous avons réalisés dans le cadre de nos contributions, nous expliquerons en détail l'architecture en question. Pour les autres, nous nous référons aux données disponibles dans les publications. Il pourra ainsi s'agir indifféremment de ARM utilisant le jeu d'instruction 32 bits (AArch32) ou 64 bits (AArch64).

4.1 Travaux existants

Nous nous sommes premièrement penchés sur les attaques par observation. Comme cela a été vu, ce type d'attaque est plutôt populaire dans le domaine des μc. Cela l'a conduit à être également utilisé dans le domaine des SoC. Cette revue ne constitue pas une liste exhaustive de toutes les attaques menées, mais permet de mettre en lumière quelles sont les vecteurs d'attaque habituels.

On va ici se concentrer sur les différentes surfaces d'attaques identifiées plus tôt.

4.1.1 Basées sur des périphériques internes

Des attaques basées sur les périphériques ont déjà été menée. Dans ce cas, diverses parties du SoC permettent de créer une vulnérabilité par leur observation selon une grandeur physique.

On peut citer comme cas particulier l'attaque menée par Owusu *et al.* [OHD+12]. Cette attaque est un approfondissement de nombreux travaux sur les attaques physiques basées sur des composants périphériques. Ici, elle a permis aux auteurs de retrouver un mot de passe à l'aide de l'accéléromètre présent à l'intérieur d'un smartphone. Cette attaque se base sur une mesure physique qu'est le mouvement de l'appareil et exploite la présence sur le système de l'accéléromètre.

Par le biais d'une application qui surveille l'état de ce périphérique lors de la saisie d'un code PIN, il est possible d'obtenir des relevés sur les mouvements de l'appareil. En couplant ces mesures avec l'agencement connu du clavier alphanumérique les auteurs ont donc pu retrouver quelles touches avaient été pressées. Par la même, ils en ont déduit les différents codes PIN.

Cette attaque concerne un très grand nombre de SoC, puisque qu'aujourd'hui pratiquement tous ceux utilisés dans les téléphones mobiles ou les tablettes sont pourvus de ce type de capteur. Pour des usages allant de la rotation de l'écran en fonction de l'orientation de l'appareil, jusqu'à la commande de certains jeux vidéos. Ainsi, cette attaque n'est pas limitée à un modèle de SoC en particulier mais à tous ceux ayant recours à ce périphérique.

Malgré la présence de ces attaques, une correction sur le périphérique ou sur sa communication avec les autres composants du SoC peuvent permettre d'assurer de nouveau une protection. Ainsi, limiter l'accès à l'accéléromètre lors d'opérations sensibles permet d'éviter ce type d'attaque.

4.1.2 Basées sur une fuite interne

Pour approfondir, d'autres attaques se sont concentrées sur des parties qui ne peuvent pas être modifiées ou anticipées lors de la conception.

On peut citer les travaux de Genkin *et al.* [GPP+16]. Dans ces travaux, l'attaque est de type DOA, le but est ici de retrouver la clef de chiffrement de l'algorithme de signature Elliptic Curve Digital Signature Algorithm (ECDSA) utilisé dans des applications bancaires pour smartphones tels que les portefeuilles Bitcoins ou Apple Pay.

Dans leurs travaux, la cible clairement identifiée est le fonctionnement de l'algorithme en fonction de l'implémentation qui en est faite dans différentes bibliothèques disponibles.

TABLE 4.1 – Récapitulatif des résultats obtenus en fonction de l'implémentation et de l'architecture testée. Extrait de Genkin *et al.* [GPP+16].

Numéro	bibliothèque utilisée	Version	Plateforme	Résultat
1	OpenSSL	1.0.x - 1.1.x	Android	Informations sur les opérations
2	OpenSSL	1.0.x - 1.1.x	iOS	Extraction possible de la clef
3	CommonCrypto	7.1.2 - 8.3	iOS	Informations sur les opérations
4	CoreBitcoin	Commit `81762ae3`	iOS	Extraction possible de la clef

L'algorithme ECDSA repose sur la multiplication scalaire, cette opération peut-être réalisée selon plusieurs algorithmes, ici il s'agit de `DOUBLE and ADD`.

En mesurant les émissions électromagnétiques, les auteurs sont en capacité d'observer des schémas de fuite sur les opérations du `DOUBLE and ADD`. Plus particulièrement, ils se basent sur la fréquence d'apparition de ces schémas.

Ainsi, dans les cas [1] et [3] du tableau 4.1, on remarque que la clef n'a pas été totalement retrouvée, mais la séquence d'opérations et donc le nombre de `ADD` et de `DOUBLE` sont identifiés, permettant ainsi d'obtenir des informations simplifiant les calculs nécessaires pour retrouver la clef.

Ici, la cible est donc clairement l'implémentation d'un algorithme de chiffrement spécifique. La microarchitecture laisse fuiter des informations, mais les auteurs se sont concentrés sur des blocs d'opérations récurrentes.

Ce choix, leur a permis de retrouver directement la clef dans au moins 2 cas de figure (les [2] et [4] du tableau 4.1), et donne de bons indices dans les deux autres cas.

Cependant, l'élément précis responsable de la fuite (le périphérique ou le transfert mise en cause), n'est pas directement identifiable, ce qui rend la transposition des résultats hors du cadre des algorithmes de chiffrement, qui ont des schémas d'exécution plutôt longs et réguliers, complexes.

Par la suite, nous avons voulu trouver une méthode permettant de la transposer sur des applications de tout type.

4.1.3 Basées sur les communications entre les éléments

Dans leurs travaux, Lipp *et al.* [MDR+16] proposent une attaque basée sur la Cache Timing Attack [GSM15].

Il s'agit d'une attaque déclenchée par le logiciel et a été classée au début dans cette catégorie. Cependant, nous allons voir que sa faisabilité repose sur des caractéristiques de la microarchitecture, nous l'avons donc replacé dans les attaques physiques.

On a rappelé plus tôt que les SoC s'appuyaient sur une architecture d'échange entre les différents composants, et que ceux-ci pouvaient fonctionner à diverses fréquences. Cependant, une interconnexion est faite entre tous ces composants au moyen d'un bus. Plus particulièrement, lorsque des données sont chargées depuis la mémoire vers le cache du cœur d'exécution celles-ci transitent par le bus. D'autres périphériques communiquant également sur le bus celui-ci effectue des arbitrages entre les données à échanger et les destinataires concernés.

Ces arbitrages et le grand nombre de transferts de types divers (allant des accusés de réception aux pings en passant par les messages de synchronisation) entrainent une certaine latence pour les transferts.

Couplé à cela, les auteurs ont utilisé la correspondance des caches (*les sets*) pour obtenir davantage d'informations sur l'allocation mémoire entre les applications et ainsi déduire le comportement de celles-ci.

Ainsi, en utilisant différentes stratégies d'accès à des zones mémoires appartenant à des *sets* partagés avec des applications sensibles, il devient possible en mesurant le temps de ces accès d'obtenir une information sur la présence ou non de cette application dans le cache. Ce temps d'accès étant totalement corrélé avec les «miss» ou les «hits» réalisés dans le cache.

Nous allons ainsi détailler le fonctionnement de l'attaque et expliquer quels transferts entrent en considération.

Plus précisément, on rappelle qu'ils ont mis en place 3 stratégies :

`Prime+Probe` :

1. Occuper certains *sets* du cache.
2. Lancer l'exécution du programme ciblé.
3. Examiner quels sont les *sets* modifiés.

Cette stratégie consiste à remplir entièrement le cache avec des valeurs connues ([1]). Après cela, une phase d'étude du temps d'accès aux données est effectué.

Par la suite, le programme à attaquer sera exécuté ([2]). À la fin de l'exécution de celui-ci ([3]), il faudra examiner le cache pour voir quelle sont les valeurs qui ont changé et donc être capable de déterminer quel *set* de cache a été utilisé par l'application à attaquer.

La phase d'analyse ([3]) se déroule comme suit, le programme de test va de nouveau chercher à accéder aux *sets* du cache qu'il occupait auparavant. En cas d'accès préalable par le programme ciblé, le *set* de cache aura été modifié et donc le temps d'accès aux différentes valeurs (ou adresses) s'en retrouvera augmenté. La donnée étant présente dans une zone de mémoire d'un niveau supérieur et devant être rapatriée dans un cache.

`Flush+Reload` :

1. Une zone de mémoire est évincée du cache.
2. Attente pendant l'exécution de l'application ciblée.
3. Relire la zone de cache évincée plus tôt

Ici, il s'agit d'une variante de la stratégie précédente. On se base ici sur la présence d'une instruction «flush» permettant d'évincer tout ou partie du cache afin de forcer à un rechargement depuis la mémoire. C'est ce qu'il se passe lors de l'étape [1].

Cette stratégie se prête davantage aux systèmes ayant un niveau de cache partagé entre plusieurs cœurs. Précédemment l'étape [2] consistait en un changement de contexte, l'application cible devait alors s'exécuter sur le même cœur. Là, on est sur une situation où l'étape [2] doit être dimensionnée afin d'englober l'accès mémoire réalisé par le programme cible.

Cette stratégie est plus détaillée dans [YF14]. L'étape [3] comme précédemment repose sur une mesure de temps d'accès qui dépendra de la présence ou non de la donnée et de la microarchitecture.

`Evict+Reload` : Cette technique est de nouveau une évolution des deux précédentes techniques. Ici l'instruction flush est remplacée par les différentes stratégies d'éviction. Les attaquants connaissant une partie de la correspondance, vont chercher à «chasser» du cache certaines données avant de mesurer les temps d'accès.

`Flush+Flush` : Cette stratégie est totalement différente des stratégies précédemment vues. Ici on ne se base plus sur des accès en mémoire pour retrouver des zones de mémoires utilisées, mais uniquement sur le temps d'exécution de l'instruction de «flush». Cette instruction doit évincer le cache, pour cela elle

doit surveiller sa présence sur le cache courant ainsi qu'éventuellement sur les autres caches. Au cas où la ligne ait été accédée, le temps d'exécution sera ainsi plus long (le temps d'évincer le cache) que si la ligne n'est pas présente.

4.1.4 Conclusion préliminaire

Plusieurs attaques ont été menées dans le domaine de l'observation sur les SoC. Ciblant à la fois des algorithmes cryptographiques et des applications généralistes, elles ont su démontrer leur efficacité.

Nous avons voulu les étendre en nous focalisant uniquement sur l'effet de la microarchitecture pour nous permettre de retrouver des informations. Notamment, nous avons voulu confronter la TZ à ce type d'attaque.

4.2 Contribution : mise en place d'une attaque par observation sur SoC

Les travaux présentés dans cette section ont fait l'objet d'une publication intitulée *How TrustZone could be bypassed ?* [BLB+17].

4.2.1 Cible

Nous avons sélectionné la Raspberry Pi 2 B pour nos études sur les SCA sur SoC. Son architecture partageant des caractéristiques avec les smartphones notamment. Parmi ces caractéristiques, on note :

- 4 cœurs Cortex A7 utilisant l'ISA ARMv7A et fonctionnant à 900 MHz.
- Un Graphical Process Unit (GPU) VideoCore IV fonctionnant à 250 MHz.
- Une mémoire RAM de 1GB fonctionnant à 400 MHz.
- La possibilité d'activer la TZ.
- Ports General-Purpose Input/Output (GPIO) afin de pouvoir communiquer selon le protocole voulu.

Il faut également ajouter que la mémoire utilisée dans ce SoC est hiérarchisée sur deux niveaux, rendant les transferts asynchrones.

L1 : Premier niveau de cache d'une taille de 32KB avec instructions et données séparées. Cache «4-way associative».

L2 : Second niveau de cache d'une taille de 512KB avec données et instructions unifiées. Partagé entre GPU et CPU. Cache «8-way associative».

Le choix de cette carte a été motivé par le modèle des cœurs utilisés, en effet le même modèle est utilisé sur un grand nombre de produits. Il s'agit également de cœurs Cortex non-rebadgés, ce qui signifie qu'ils ne sont pas modifiés comme peuvent l'être des cœurs produis par Apple, Samsung ou Qualcomm. Nous avons donc pu facilement nous reporter aux documentations fournies par ARM. Dernier point, la disponibilité de cette carte a eu pour effet de créer une grande communauté nous permettant d'obtenir des informations facilement sur des éléments non documentés directement.

4.2.2 Défis par rapport aux μc

Des attaques en écoute sont possibles sur des μc. Nous avons voulu explorer la possibilité sur des SoC, mais des différences architecturales entrainent une augmentation de la complexité. Nous avons donc dû résoudre un certain nombre de défis.

Vitesse de fonctionnement

Comparativement à la cible que nous avions précédemment, pour rappel un Cortex M3 fonctionnant à 24 MHz. La Raspberry Pi 2 (RPi2) a une vitesse de fonctionnement 36 fois supérieure. L'un des défis de nos test a donc été de conserver une synchronisation malgré l'augmentation de vitesse. Afin de pouvoir cibler nos mesures et les concentrer sur ce que nous souhaitions attaquer.

Complexification de la microarchitecture

Les SoC ont un grand nombre d'éléments supplémentaires en comparaison aux μc. Concernant, les attaques en écoute ceci accentue le bruit électromagnétique, puisque le nombre de transistors est multiplié. Ce bruit est composé pour partie des éléments supplémentaires et donc de tous les signaux émis à des fréquences différentes.

D'autres part, on a vu plus tôt que certains intègrent des options de sécurité, comme la TZ par exemple.

Ces modifications ou ces composants ont pour objectif d'accroitre la sécurité de ces systèmes, nous avons voulu voir à quel point, ils rendaient plus difficile la tenue d'attaques physiques.

Cas d'utilisations testés

Par rapport à ces défis, nous avons voulu mettre en place deux types d'attaque en écoute d'une part la CPA et d'autre part l'attaque par profilage.

Ces deux attaques ont permis de tester les défis qui ont été posés. Pour la CPA, nous avons ainsi souhaité mettre en lumière que la vitesse d'exécution ne rendait pas impossible la tenue de cette attaque si le matériel employé est le bon. Et pour le Template sa dimension plus globale permettait de montrer que malgré le bruit électromagnétique et la vitesse nous étions capable de retrouver une information contenue dans les émissions électromagnétiques. Ce qui nous a permis d'envisager d'autres champs d'application qui seront présentés dans les conclusions.

4.2.3 Attaque *CPA*

Dans un premier temps, nous nous sommes concentrés sur la «Correlation Power Analysis» (CPA). Il s'agit d'une attaque de type DOA, elle cible la clef de chiffrement.

Principe

Lorsque le SoC est en fonctionnement de nombreuses fonctions sont actives et les «tics» d'horloge retentissent. Comme on a vu l'activation de certains transistors mène à l'activation de circuits combinatoire qui mène à l'émission de rayonnement électromagnétique mesurable par nos équipements.

Le but de l'attaque CPA est de trouver une relation entre les émissions qui sont mesurées et le comportement au niveau logique. Pour réaliser cette attaque certaines informations doivent être connues, dans notre cas le texte envoyé, le calcul visé et le début de ce calcul.

À partir de là, nous avons voulu pour notre attaque identifier comment corréler les émissions avec une valeur intermédiaire d'un algorithme de chiffrement pour effectuer une attaque par clair connu, semblable à ce que l'on retrouve dans les travaux de Brier *et al.* [BCO04].

Mise en place

Notre environnement de test se divise en deux parties, dans un premier temps du côté du système à attaquer, nous avons voulu recréer des conditions proches de la réalité et tester l'effet de diverses fonctions

internes sur la sécurité contre les attaques en écoute.

Nous avons donc dans un premier temps mis en place un programme de chiffrement AES logiciel. Nous avons sélectionné cet algorithme de chiffrement dans cette situation, d'une part car sa faiblesse face à la CPA a déjà pu être prouvée sur d'autres architectures. D'autre part, nous verrons par la suite que son fonctionnement est facilement repérable dans des émissions électromagnétiques. Enfin, notre but était de prouver la faisabilité de ce type d'attaque malgré les défis imposés.

Enfin cet algorithme est également largement utilisé aujourd'hui dans le domaine des systèmes embarqués. Il sert à la fois, à chiffrer des communications, mais également à chiffrer des données. Android l'utilise par exemple pour le chiffrement de la mémoire interne des smartphones (voir sections Full-Disk Encryption et File-Based Encryption dans [And18]), en stockant les clefs dans la TZ (voir section Storing Encrypted Key dans [And18]).

Ainsi, il est possible sur la cible de modifier la clef, d'envoyer un texte clair et d'en obtenir le texte chiffré correspondant.

Une fois le programme sélectionné, nous avons voulu identifier l'effet de divers aspects architecturaux des SoC. Pour cela, nous avons mis en place 4 configurations.

Configuration `D` La première configuration notée `D` pour `DEFAUT`, consiste en un état le plus simple possible. Comme visible sur la Figure 4.1, on utilise un seul cœur pour faire le travail demandé. Le premier cœur de calcul est placé dans un état non sécurisé et effectue le calcul du chiffrement AES du texte clair qui lui est envoyé en entrée. Les autres cœurs sont placés dans un état `IDLE`.

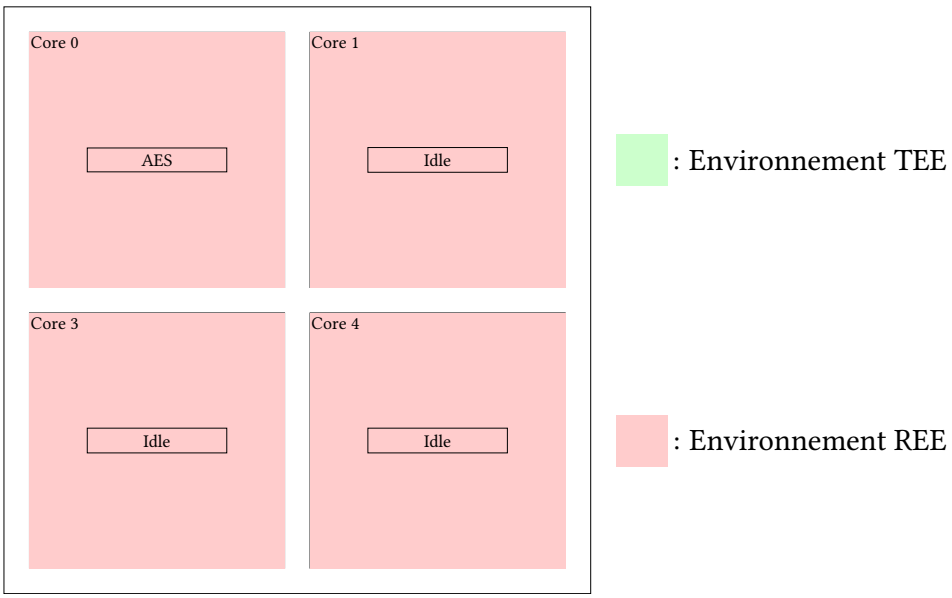

FIGURE 4.1 – État des cœurs dans la configuration `D`.

Cet état simule un non-fonctionnement des cœurs. La maitrise qui nous est offerte des cœurs sur notre système RPi2, ne nous permet pas de les arrêter complètement.

Nous avons donc placé les cœurs dans une boucle infinie d'attente. Le cœur va alors exécuter la même logique comportementale pendant toute la durée du test.

Nous avons considéré que le bruit électromagnétique généré par ces exécutions pouvait avoir un effet sur nos mesures et nous verrons plus tard ce qu'il en a été. Mais le but était de le rendre «prédictible» en lui donnant une forme la plus constante possible, afin de limiter son effet sur la corrélation.

Le but premier de cette configuration est de valider la faisabilité d'une CPA sur un système plus rapide et complexe qu'un μc.

Configuration `S` La deuxième configuration notée `S` pour `SECURE`, est une reprise de l'état précédent en utilisant cette fois-ci l'isolation logicielle offerte par l'utilisation de la TEE d'ARM, la TZ.

Ici, comme visible sur la Figure 4.2, le calcul du chiffrement est toujours dévolu à un seul cœur, et les autres sont toujours dans un état de boucle infinie constante.

Cependant, le calcul est effectué en activant la TZ. Pour rappel cela signifie que le programme s'exécute de manière à être totalement isolé de tous les programmes qui ne s'exécutent pas dans ce mode.

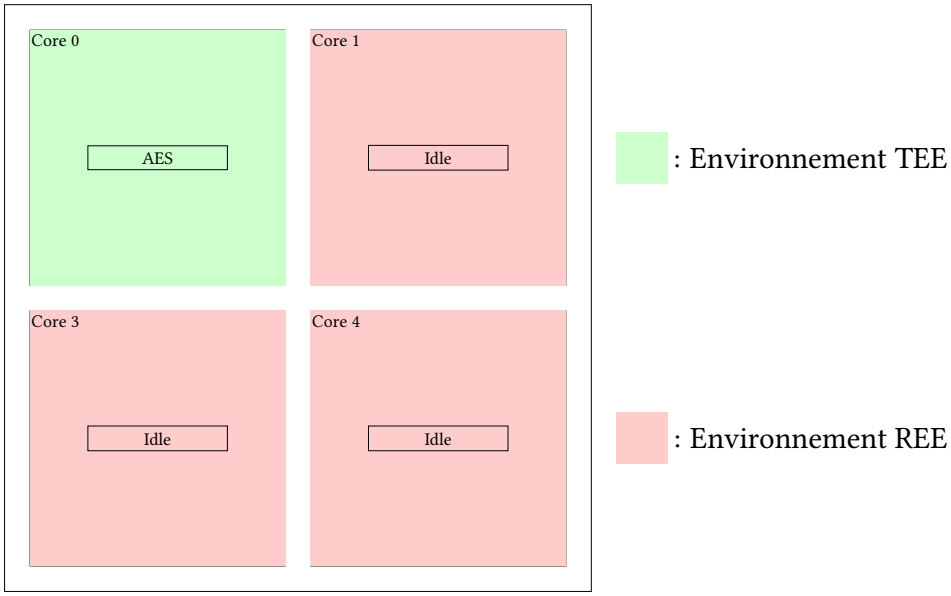

FIGURE 4.2 – État des cœurs dans la configuration `S`.

L'implémentation de la TZ étant en partie matérielle, nous avons par ce moyen voulu tester l'effet des modifications qu'impliquent la TZ. L'étendue des modifications, qui consistent principalement en l'ajout d'un bit lors des transferts et l'ajout de nouveaux niveaux de privilèges nous laisse cependant à penser que la TZ n'aura qu'un effet limité en comparaison de la configuration `D`.

Configuration `D+M` La troisième configuration notée `D+M` pour `DEFAUT + MULTICŒUR` est cette fois-ci différente. Nous avons voulu reprendre la base de la configuration `D` en générant cette fois davantage de bruit. Ce qui nous a conduit à adopter la configuration visible sur la Figure 4.3.

Pour cela, les 3 autres cœurs effectuent des calculs aléatoires. Chacun des autres cœur va générer 2 valeurs aléatoires en utilisant l'algorithme de génération de nombre pseudo-aléatoires (Pseudo-Random Number Generator (PRNG)) Xorshift [M$^+$03]. Ensuite, le cœur va calculer le résultat du Plus Grand Commun Diviseur (PCGD) de ces deux valeurs, l'algorithme de calcul de cette valeur rend le nombre d'opérations différents et difficilement prédictible.

Ces calculs ont une durée aléatoire et des cœurs désynchronisés. Nous avons voulu générer un maximum de bruit électromagnétique.

Ainsi, le but de cette configuration était de simuler le fonctionnement normal d'un SoC multicœur non sécurisé, on multiplie les calculs générant du bruit électromagnétique.

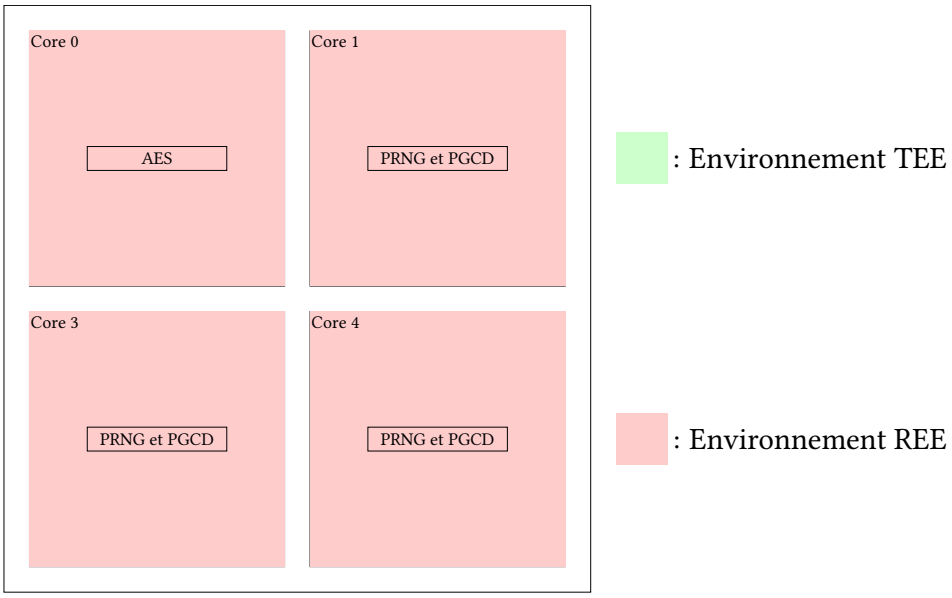

FIGURE 4.3 – État des cœurs dans la configuration `D+M`.

Configuration `S+M` La dernière configuration notée `S+M` pour `SECURE + MULTICŒUR` est une reprise de la configuration précédente pour l'état des 3 cœurs voisins. Pour le cœur principal, cette fois-ci nous avons souhaité reprendre la sécurité mise en place dans la configuration `S`. Ainsi on peut voir dans la Figure 4.4 l'état des divers cœurs au cours de ce test.

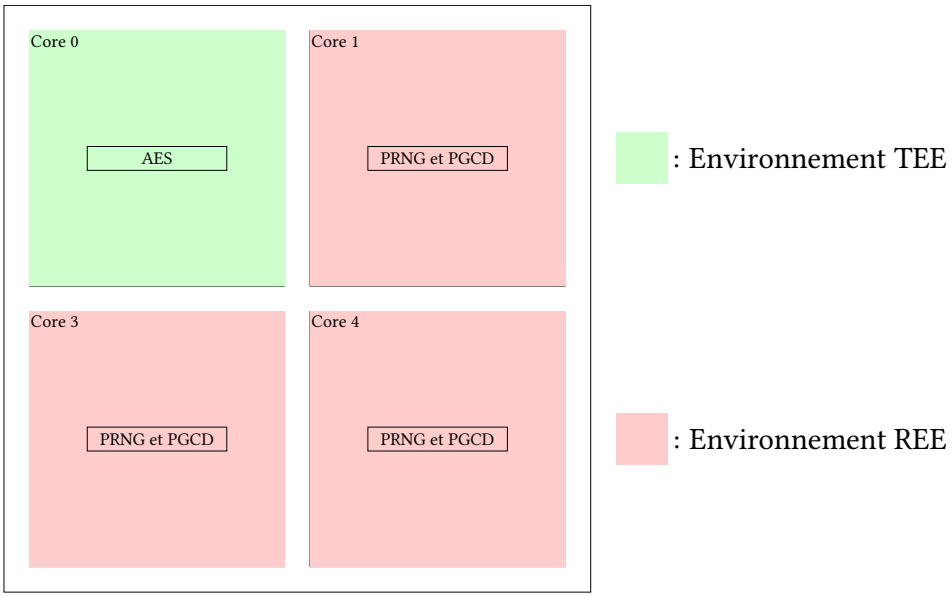

FIGURE 4.4 – État des cœurs dans la configuration `S+M`.

Le but de cette configuration est de se rapprocher d'une configuration normale pour un programme sensible. En effet, celui-ci est censé s'exécuter en TZ et comme on l'a vu précédemment des programmes peuvent s'exécuter en parallèle sur d'autres cœurs. Ces cœurs créent ainsi du bruit électromagnétique.

Communication Sur notre plateforme de test un programme permet d'envoyer divers textes clairs connus pour qu'ils soient chiffrés par le système que nous souhaitions attaquer. Pour la communication principale nous avons utilisé les GPIO dans une configuration permettant d'obtenir une sortie UART et donc de déporter l'affichage qui devrait être sur le système sur notre banc de test. Ce port nous a donc permis de récupérer les valeurs des chiffrés.

Pour la communication secondaire, nous avons mis en place une connexion JTAG toujours via les ports GPIO. Ceci afin d'avoir des informations sur le fonctionnement interne du système et vérifier que nos programmes s'exécutaient correctement.

Derniers point, nous avons utilisé également deux ports GPIO afin d'envoyer des signaux de synchronisation au reste de nos équipements.

Corrélation Nous avons donc mesuré les émissions électromagnétiques lors du premier tour de l'AES.

Notre cible disposait d'une clef fixée que nous souhaitions retrouver. La cible recevait alors un texte clair en entrée pour renvoyer un texte chiffré.

À l'aide de ces deux textes et des mesures venant de l'oscilloscope, notre banc de test était en capacité de calculer des valeurs de corrélation. À partir du texte clair connu, plusieurs hypothèses de clef vont être testées.

Point supplémentaire Dans nos mesures, nous n'avons pas pris en compte l'effet de la hiérarchie mémoire. En effet, en fonction de la section de code et de la position dans la mémoire des différentes variables des échanges entre les différents niveaux de mémoires (RAM, cache, etc) peuvent apparaitre. Ces échanges ont un effet certain sur les émissions électromagnétiques, mais souhaitant se mettre dans une situation proche de la réalité (où la structure exacte de la mémoire est inconnue), nous avons ignoré ces effets.

4.2.4 Observations et résultats de l'attaque CPA

Les résultats de ces diverses attaques nous ont permis d'en apprendre beaucoup sur la faisabilité des attaques par observation sur des systèmes de type SoC.

L'algorithme de chiffrement AES repose sur plusieurs tours d'exécutions. Nous avons pour notre part concentré nos efforts sur le premier tour de l'AES, plus précisément sur le passage au travers de la S-Box qui est connue et fixe. Lors du premier tour, l'algorithme AES effectue un «ou-exclusif» entre chacun des 16 octets de la clef et les 16 octets du texte clair, la valeur résultante est permutée suivant la S-Box. En réalisant l'écoute à ce moment, il faut alors être capable de déduire les octets de la clef avec pour information les octets du texte clair et la valeur de la S-Box et donc toutes les permutations possibles.

Pour nos mesures nous avons disposé de plusieurs sondes. Ainsi, nous avons utilisé les sondes Langer *RF-R 0.3-3* (sonde droite, pointe une zone de la puce) et *RF-R 400-1* (sonde circulaire, couvre toute la puce).

Observations préliminaires

Tout d'abord, nous avons tâtonné afin de trouver une zone d'écoute nous semblant intéressante. Pour cela, nous nous sommes basés sur la particularité de l'AES d'être un algorithme par tours. C'est-à-dire que les mêmes traitements sont effectués plusieurs fois. Nous avons donc cherché à isoler ce schéma sur l'oscilloscope.

Avec la sonde *RF-R 0.3-3*, nous nous sommes donc déplacés sur la puce tout en exécutant en boucle un chiffrement AES dans la configuration `D`.

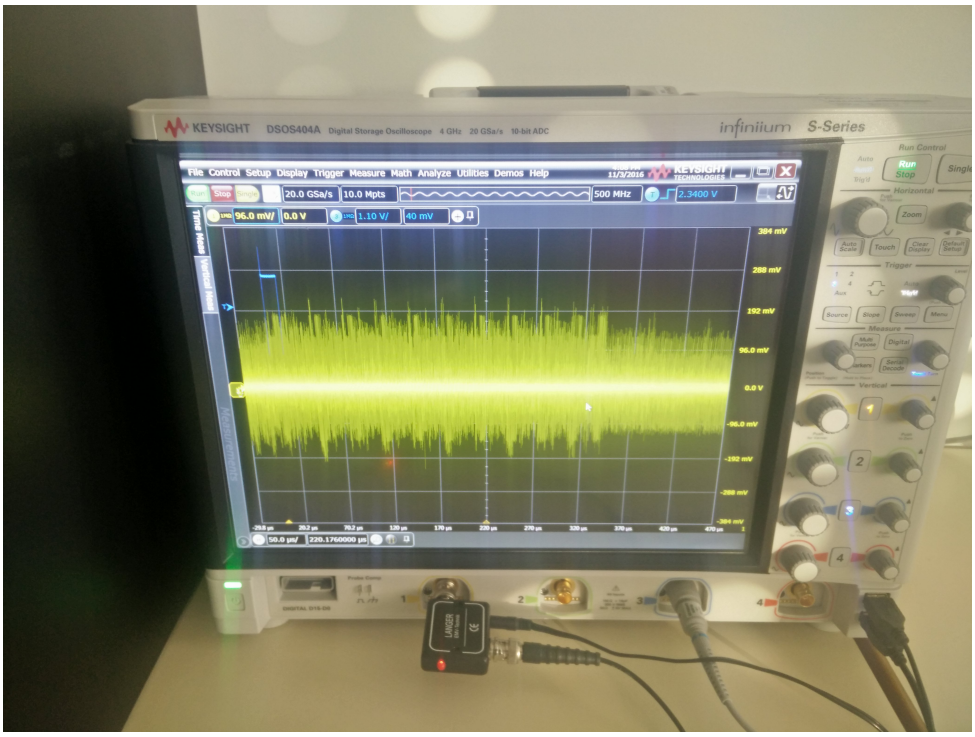

FIGURE 4.5 – Signal visible sur l'oscilloscope si la fuite est correctement identifiable. En bleu, on aperçoit le signal de synchronisation et en jaune la trace d'exécution de l'AES complet, les 10 tours sont identifiables par 10 motifs identiques.

Ceci nous a permis d'identifier le schéma visible sur l'image 4.5. Sur cette image, on distingue sur la gauche un signal carré bleu, il correspond à notre signal de synchronisation qui encadre le traitement du SubBytes du premier tour par l'algorithme AES, c'est cette zone que nous avons souhaité attaquer. Sur cette image, nous pouvons également remarquer que le comportement spécifique de l'AES est bien visible. Celui-ci, se caractérise par un motif se répétant 10 fois. Ce motif observable dans la zone jaune correspond aux 10 tours de l'AES.

Ce premier test nous a permis de fixer une position spatiale pour notre sonde. Ensuite nous nous sommes concentrés sur la zone que nous souhaitions attaquer afin d'essayer de comprendre quelles étaient les opérations responsables de ces fuites.

En nous focalisant sur l'exécution du SubByte, on remarque l'apparition de plusieurs pics successifs tels que visible sur la Figure 4.6. En analysant la fréquence d'apparition de ces pics, nous avons pu remarquer qu'elle était d'environ 400 MHz. Cette fréquence correspond à celle de la mémoire RAM. Nous en avons donc conclu qu'il s'agissait là d'opérations en rapport avec la mémoire RAM.

Cependant, malgré la constatation de ces deux phénomènes qui témoignent de la présence manifeste d'une fuite, nous n'avons pas été capables de réaliser une corrélation avec ces informations. La méthode généralement utilisée, pour retrouver la valeur est la corrélation avec le poids de Hamming. On considère que les niveaux des émissions sont corrélés avec les poids de Hamming des valeurs manipulées.

Par la suite, nous avons modifié notre méthode de mesure. Nous avons utilisé la sonde *RF-R 400-1*, mais du fait de la taille de celle-ci nous l'avons placé sur le dessous de la carte.

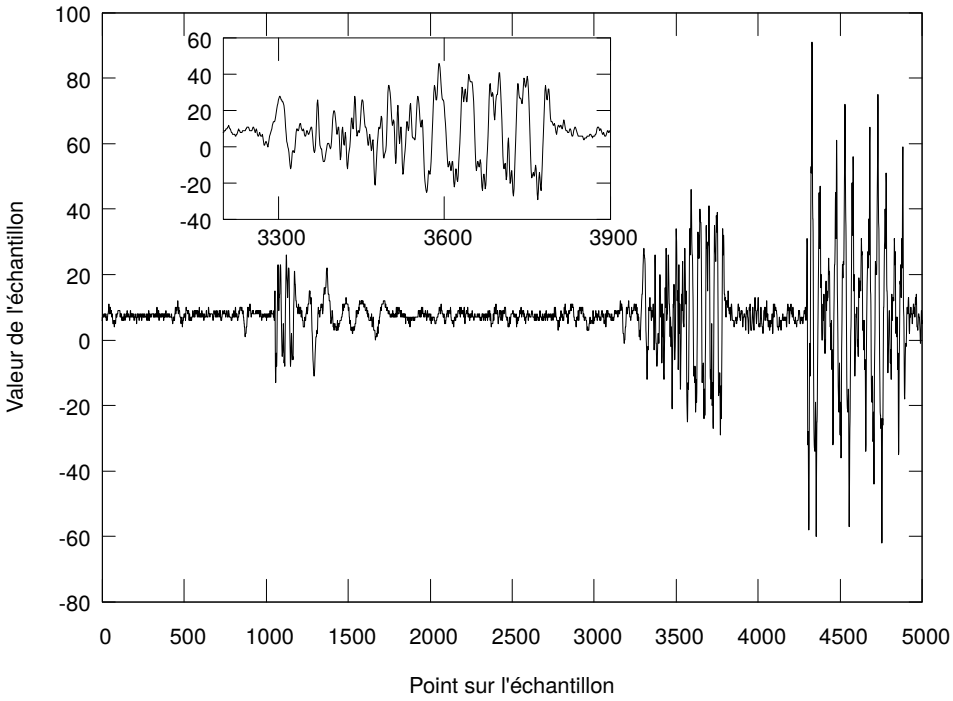

FIGURE 4.6 – Une trace d'exécution du SubByte. On identifie un schéma ressemblant à un accès mémoire, avec un motif pour l'accès RAM→L2, un autre pour L2→L1 et un dernier pour L1→registres.

Résultats finaux

Avec ce placement nous avons de nouveau tenté de faire une corrélation. Néanmoins, nous n'avons également pas été en capacité d'obtenir une corrélation à ce niveau.

Nous avons donc modifié notre modèle de corrélation en essayant de le lier cette fois à la valeur elle-même. À ce moment là nous avons été capables d'obtenir une corrélation. Comme cela a été précisé plus tôt nous nous sommes intéressés au premier octet de la clef uniquement, étant donné que nous ciblions la microarchitecture et non l'algorithme de chiffrement en lui-même.

Nous avons donc obtenu les résultats visibles sur la Figure 4.7

De plus, nous avons été en mesure d'identifier divers points de corrélation. En effet, malgré le faible niveau de corrélation, les pics sont clairement identifiables d'une part et sont au nombre de 3. Une hypothèse est que les divers pics représentent la traversée de la hiérarchie de la mémoire.

En effet, notre étude se base sur la microarchitecture, or dans le cas de ce SoC, les différents niveaux de mémoires se divisent en :

— La mémoire principale RAM qui contient toutes les données nécessaires à l'exécution de notre programme et donc toutes les variables.

— La mémoire cache qui contient un certain nombre de ces éléments à un instant donné, et qui les verra tous passer au cours de l'exécution.

— Les registres qui au cours de l'exécution vont contenir tous les éléments.

Notre hypothèse est donc que l'on corrèle, avec plus ou moins d'importance, avec la valeur du premier octet de la clef lors de ses passages dans ces divers niveaux de mémoire.

Cette hypothèse est de plus renforcée par notre position sur la puce. En effet, nous avons dans ce cas

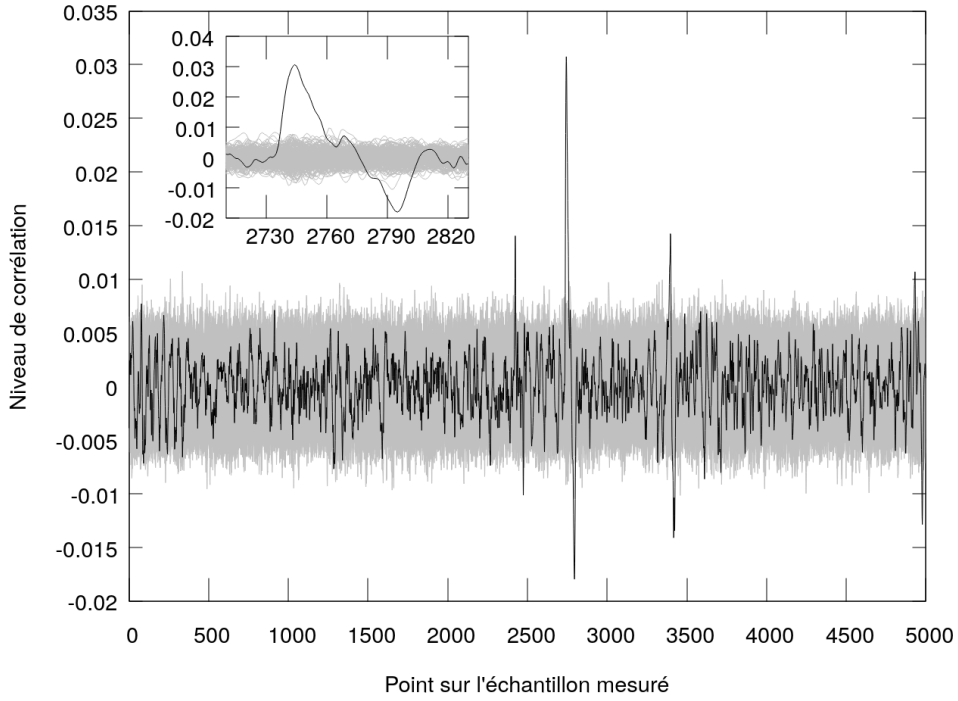

FIGURE 4.7 – Niveau de corrélation pour la configuration `D`, issus de $200k$ traces. La courbe noire représente le niveau de corrélation de la bonne hypothèse de clef. Les autres sont en gris.

placé la sonde en dessous de la cible. La sonde a alors couvert alors la quasi-totalité du package, ce qui nous a permis d'écouter toute la microarchitecture.

Résultats de la CPA pour les différentes configurations Après avoir été en mesure de trouver une corrélation, nous avons reproduit l'attaque en modifiant uniquement les configurations possibles à savoir `S`, `D+M` et `S+M`. Les résultats sont présentés dans le tableau 4.2

TABLE 4.2 – Pour 200k traces de 5000 points. Chaque mesure couvre une durée d'exécution de 250ns (soit 20GP/s).

Configuration	Corrélation maximale	Point d'apparition
`D`	3.1%	2745
`S`	3.4%	2746
`D+M`	3.7%	2746
`S+M`	3.6%	2747

On constate que les niveaux de corrélation sont sensiblement les mêmes et se produisent exactement au même niveau temporel. En effet, la fréquence du composant le plus rapide du SoC est d'environ 900 MHz. Ce qui signifie qu'une instruction de celui-ci dure environ 1.1 ns, or 20 points couvrent une durée de 1 ns. Ce qui signifie que dans un intervalle de 20 points, nous pouvons considérer qu'il s'agit exactement du même endroit.

Ici l'intervalle est de deux points ce qui nous permet de confirmer que la fuite à bien lieu au même niveau exactement pour toutes les configurations.

Grâce à ces résultats nous sommes donc en mesure de confirmer que la microarchitecture des SoC demeure sensible au même titre que celle des μc face aux attaques par observation. De plus, l'ajout de la TrustZone qui est une protection logicielle assistée par le matériel n'a, comme attendu, eu aucun effet sur la sécurité des SoC contre ces attaques.

Hypothèse sur les résultats préliminaires

Nous avons réussi à obtenir des résultats concluant et nous avons pu émettre diverses hypothèses pour expliquer nos échecs successifs.

Dans un premier temps, la corrélation était impossible selon les méthodes habituellement utilisées du poids ou de la distance de Hamming. En modifiant le modèle de fuite pour qu'il corresponde à la valeur elle-même, nous avons émis l'hypothèse que ce sont les chargements, dans les niveaux de hiérarchie mémoire, que nous étions capable de mesurer. Cela nous permet également de définir une future direction que pourraient prendre les attaques de type CPA sur des SoC.

Dans un second temps, nous avons été confrontés à l'impossibilité de corréler malgré la présence manifeste du signal visible assez clairement sur l'oscilloscope. Encore une fois, cette situation est assez inédite puisqu'elle était suffisante dans le cas des μc. Restant sur notre hypothèse de la hiérarchie de la mémoire, nous avons voulu élargir la zone sur laquelle nous écoutions pour mesurer la totalité des mouvements.

Observations pour des travaux futurs

D'autres éléments ont été observés et constituent des points d'amélioration pour de futurs travaux.

Nous avons vu que la hiérarchie de la mémoire avait une importance sur nos résultats.

Dans notre cas, nous avons réalisé nos tests sur un système sur lequel nous avions un contrôle total, notre application s'exécutait en `bare metal` sans OS qui viendrait apporter du multitâche, des interruptions ou des opérations non prévues. Cela nous a simplifié le travail pour réaliser nos expériences et nous synchroniser avec le système.

Cependant, nous avons tout de même pu expérimenter des gigues ou «jitter», qui pourraient être améliorés pour réaliser des mesures plus précises. Notamment, en limitant au maximum les appels à la hiérarchie de la mémoire.

De plus, nous avons également constaté que l'implémentation des S-Box, pouvait causer de la désynchronisation par le parcours de la hiérarchie mémoire. Pour palier à ce problème, nous avons utilisé un signal se déclenchant avant le début du second appel à la S-Box, permettant ainsi de nous synchroniser avec le premier appel quelle que fût sa gigue.

Enfin, pour réaliser ces expérimentations, nous avons passé un certain temps afin de trouver la bonne position d'écoute, en partie dû à nos premières tentatives infructueuses. Pas la suite, une fois la sonde correctement sélectionnée et positionnée, le temps le plus long fut celui de mesure. Dans notre cas, du fait de notre implémentation telle qu'elle a été présentée plus tôt, la campagne de mesure a duré environ 18 minutes pour obtenir les 200000 courbes de 5000 points chacune. Le temps de calcul n'a pas été calculé précisément mais dépend de l'implémentation choisie de la CPA.

4.2.5 Attaque par profilage

Dans un second temps, nous nous sommes concentrés sur l'attaque par profilage qui cette fois-ci permet de caractériser un comportement. Il est donc possible de cibler n'importe quel type de programme et plus uniquement les algorithmes de chiffrement. Il s'agit d'une attaque pouvant être de type DOA ou IOA selon ce qui est caractérisé.

Principe

Il s'agit d'une attaque permettant d'attaquer un circuit que l'on aura au préalable caractérisé. Un attaquant dispose donc d'un système (S_{profil}) sur lequel il peut effectuer toutes les mesures nécessaires, il en a le contrôle total. Et un second système (S_{cible}) qui est celui qui sera attaqué et sur lequel il ne peut réaliser qu'une quantité limitée d'action ou pour lequel il ne dispose que d'un accès restreint.

À l'aide de S_{profil}, l'attaquant doit réaliser suffisamment de mesures pour établir une caractérisation du comportement qu'il souhaite attaquer. Si la cible est par exemple l'exécution ou non de l'instruction `ldr`, il devra l'utiliser avec tous les registres cibles et destinations possibles et réaliser de nombreuses mesures. Ainsi, lorsqu'il fera une mesure sur S_{cible}, il sera en mesure de déterminer si l'instruction `ldr` a ou non été exécutée, en la comparant aux mesures déjà effectuées sur S_{profil}.

Le Template permet de donner des estimations de distance entre le comportement mesuré et le comportement caractérisé. Offrant ainsi la possibilité de donner des probabilités, en cas d'échec.

Mise en place

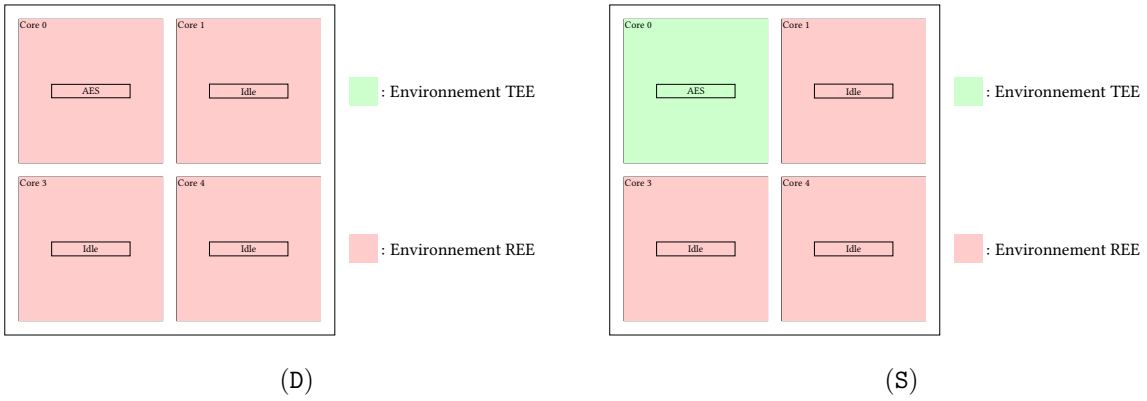

FIGURE 4.8 – Les configurations sont les mêmes que pour l'AES. `D` pour `DEFAUT`. `S` pour `SECURE`.

De la même manière que pour l'AES, nous avons séparé les configurations, en deux configurations monocœurs visibles sur la Figure 4.8 et deux configurations multicœurs visibles sur la Figure 4.9. Dans le but de tester la faisabilité de ce genre d'attaque dans des conditions proches du réel et simulant plusieurs types d'usages.

Pour cette expérimentation, nous avons souhaité modéliser un fonctionnement que l'on retrouve dans une large majorité de smartphone également. Il s'agit de la vérification de code PIN.

Dans ce cas, nous avons placé sur la cible à attaquer, un programme capable d'effectuer une vérification de code PIN tel que proposé par Rivière dans [Riv15]. Cet algorithme permet d'assurer une protection contre les attaques par chronométrage, en particulier, il est étudié pour s'exécuter en temps constant.

Ainsi le logiciel inclus sur le système à attaquer est capable de recevoir un code PIN candidat (valeur à tester) qu'il compare à un code PIN inclus dans la mémoire.

Nous nous sommes concentrés sur le code PIN car il s'agit d'une des fonctionnalités critiques les plus utilisées sur les téléphones mobiles qu'il s'agisse d'appareils fonctionnant sous Android ou iOS. Ce PIN permet en effet de protéger l'allumage du système, son déverrouillage, et est également dans de nombreux cas une solution de secours lorsque les autres méthodes d'authentification sont indisponibles. Enfin, il sert de base pour créer les clefs utilisées pour le chiffrement des données sensibles du système (voir [And18]).

Après la sélection de l'application à tester, nous avons caractérisé notre système :

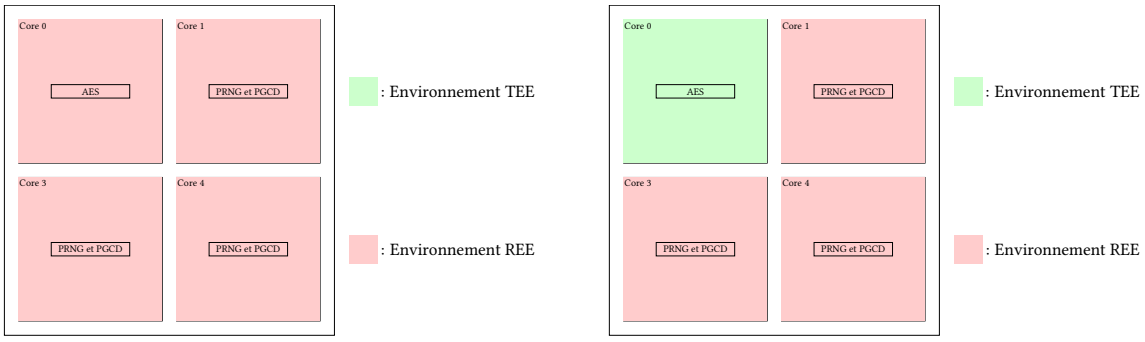

FIGURE 4.9 – Les configurations sont les mêmes que pour l'AES. `D+M` pour `DEFAUT + MULTICŒUR`. `S+M` pour `SECURE + MULTICŒUR`.

1. Sélectionner une valeur pour le code PIN candidat, qui sera entré par l'utilisateur (non modifié au cours des tests).
2. Sélectionner une valeur pour le code PIN secret qui a été placé dans le système (modifié après avoir acquis assez de mesures : 150000 traces dans notre cas).
3. Mesurer les émissions électromagnétiques durant la comparaison (200000 mesures pour chaque valeur candidate).
4. Alimenter le Template avec ces mesures.

Le choix d'un candidat unique pour plusieurs valeurs secrètes nous a permis de caractériser les opérations induites par plusieurs types de valeur secrètes, mais également le comportement lors de leur comparaison.

Notre but comme depuis le début est de nous concentrer sur la faiblesse de la microarchitecture, ainsi nous avons souhaité accélérer l'attaque en nous concentrant uniquement sur un seul chiffre du code PIN.

Il faut cependant noter que les chiffres sont comparés un à un dans le cas de notre vérification de PIN. Et ne prend pas directement en compte l'entièreté du PIN.

Cependant, l'attaque reste totalement valable, en effet les chiffres sont indépendants les uns par rapport aux autres. L'attaque ne vise donc pas à retrouver exactement le code PIN, mais à mettre en lumière la possibilité de retrouver la valeur d'une information qui transiterait au travers de la microarchitecture. Une fois tous les chiffres obtenus, le code PIN peut être reconstitué.

4.2.6 Observations et résultats de l'attaque par profilage

Observations préliminaires

Les deux sondes Langer *RF-R 400-1* et *RF-R 0.3-3* ont été utilisées pour ces expérimentations.

Avec la sonde *RF-R 0.3-3* nous avons pu obtenir des résultats concluants. Cependant, le choix de cette attaque ne nous a pas permis d'évaluer les données brutes afin de confirmer ou d'infirmer l'hypothèse de la hiérarchie de la mémoire.

Par la suite, nous sommes repassé sur la sonde *RF-R 400-1*. Celle-ci nous a permis d'obtenir des résultats beaucoup plus reproductibles.

Une des difficultés expérimentales dans ce cas par rapport à la CPA a été la quantité d'informations nécessaires.

En effet, nous avions besoin de beaucoup de mesures basées sur des expériences très variées. À savoir la comparaison de toutes les valeurs secrètes pour une valeur candidate donnée. Avec un nombre d'itérations suffisant pour chaque paire testée. Tout en prenant en compte la précision de la mesure, qui devait être suffisante pour permettre d'isoler le bruit généré, par tous les transistors et autres circuits du système, du comportement que l'on souhaitait observer.

Phase de caractérisation

Dans cette première phase nous avons effectué des mesures selon notre protocole précisé précédemment.

L'étape de la Principal Component Analysis (PCA) n'est pas obligatoire, mais elle nous a permis d'obtenir des mesures les plus significatives possibles pour alimenter notre phase de caractérisation et réduire l'ensemble de mesures dont nous disposions. Le but avec celle-ci est donc de maximiser la variance.

Les différentes durées pour cette phase se décomposent comme suit :

— 1 heure pour la génération de la matrice de transformation pour l'application de la PCA ;

— 3 heures pour l'application de cette matrice à nos mesures préalables ;

— 10 heures pour générer les différentes classes du templates

Phase d'attaque

Après caractérisation, nous avons dû tester notre modèle en le confrontant à de véritables mesures.

Avec 150000 traces lors de la caractérisation, nous en avons conclu qu'il nous fallait 15000 traces afin d'obtenir une confiance suffisante dans nos résultats.

Pour cela, nous nous sommes basés sur l'erreur type, permettant de lier l'erreur maximale avec la confiance dans ce taux d'erreur maximum.

En considérant r le taux de réussite réel, et p le taux de réussite mesuré. Nous pouvons définir l'erreur maximale comme une valeur E garantissant $|p - r| < E$. Soit σ notre niveau de confiance, c-à-d. $\sigma = 3$ signifie que nous avons un écart type de confiance de 3 ($\approx 99,7\%$).

Afin de lier notre erreur maximale et notre confiance dans cette erreur nous avons utilisé l'équation 4.1.

$$E = \sigma \cdot \sqrt{\frac{p \cdot (1-p)}{n}} \tag{4.1}$$

où n est le nombre de mesures.

Pour $p = 0.5, n = 15000$ et $\sigma = 3$, on obtient $E = 1.22\%$ (E est maximisé pour $p = 0.5$). Dans les tableaux de résultats suivants, le taux de réussite est donné avec un écart type de confiance de 3 ($99,7\%$) que l'erreur est inférieure à $1,22\%$.

Par conséquent, pour chacune des valeurs possibles pour chaque chiffre candidat à savoir de 0 à 9, nous avons réalisé 15000 mesures. Ce qui nous a conduit à avoir un ensemble de 150000 mesures supplémentaires.

Résultats finaux

Une méthode aléatoire, de détection des chiffres, aurait un taux de succès de 10%. Une méthode plus efficace que la recherche aléatoire doit donc avoir des résultats supérieurs à ce chiffre. Ceci permettant ainsi de limiter le nombre d'essais pour retrouver le code complet.

Table 4.3 – Taux de succès dans la recherche du code PIN.

Configuration	Fréquence d'échantillonnage	Durée de la mesure	Taux de succès
`D`	10 Gsamples/s	$2\mu s$ (20kpts)	37.84%
`S`	10 Gsamples/s	$2\mu s$ (20kpts)	38.3%
`D+M`	10 Gsamples/s	$2\mu s$ (20kpts)	37.88%
`S+M`	10 Gsamples/s	$2\mu s$ (20kpts)	36.40%
`D`	20 Gsamples/s	$2.5\mu s$ (50kpts)	35.19%
`S`	20 Gsamples/s	$2.5\mu s$ (50kpts)	32.85%
`D+M`	20 Gsamples/s	$2.5\mu s$ (50kpts)	18.45%
`S+M`	20 Gsamples/s	$2.5\mu s$ (50kpts)	17.81%

Sur le tableau 4.3, on constate que les taux de succès sont supérieurs à 10% dans tous les cas, ce qui confirme donc que l'attaque permet d'avoir des résultats meilleurs que l'aléatoire. En dehors des 2 derniers cas, on obtient même des taux supérieurs à 30%.

En considérant uniquement la première partie de ce tableau, le taux de succès est en moyenne de 37.3% pour toutes les configurations (en prenant en compte l'intervalle d'erreur de 1.22%). Ceci nous permet d'affirmer que malgré la présence de la TrustZone ou de calculs sur les autres cœurs du SoC, aucun effet n'est visible sur la faisabilité d'une attaque par observation sur un SoC.

La seconde partie est quant à elle intéressante puisqu'elle a mis en lumière une potentielle limite à l'attaque Template. En effet, dans cette partie, nous avons augmenté la fréquence d'échantillonnage ainsi que la durée afin d'obtenir plus d'informations sur chacune de nos mesures. Cependant, nous avons conservé la PCA. Ainsi en maximisant la variance et en conservant la même taille d'échantillon de sortie (la PCA donnant des traces de 10000 points dans les deux cas), notre hypothèse est que nous avons perdu une grande quantité d'informations, en particulier parce que nous avons augmenté la variance d'éléments n'étant pas en rapport avec le secret recherché. En particulier, on remarque que la chute du taux de succès est très visible sur les configurations multicœurs, nous pensons donc que ce sont ces informations plus particulièrement qui ont perturbé ces expériences.

Néanmoins, même dans ces cas-là, l'attaque reste plus efficace que l'aléatoire pur.

Observations pour des travaux futurs

De par sa capacité à retrouver des comportements typique l'attaque par profilage peut être un support pour des attaques en faute. En effet, nous allons le voir par la suite mais parmi les difficultés expérimentales que l'on retrouve entre les attaques en écoute ou en perturbation, la nécessité de synchroniser la mesure ou l'injection respectivement avec le programme ciblé est la phase la plus délicate et devrait faire l'objet d'études spécifiques.

Néanmoins, en utilisant les capacités du Template dans la reconnaissance de schémas spécifiques de fuites, il devient possible d'identifier des comportements et donc de s'en servir comme déclencheur pour les attaques.

4.2.7 Conclusion préliminaire

Malgré la complexification, nous avons été capables de mettre en place des attaques en observation sur des SoC.

Contrairement aux attaques précédentes, nous avons voulu nous concentrer sur la fréquence du système afin de trouver des informations sur la microarchitecture en elle-même.

Les attaques menées par ce moyen, nous ont permis de montrer que tout logiciel reposant sur cette couche, devient vulnérable par la confiance qu'il accorde à la microarchitecture.

La synchronisation n'a pas été traitée dans nos travaux en observation mais reste le défi le plus imposant à résoudre.

4.3 Conclusion

Dans cette partie, nous avons vu les attaques par observation sur SoC. Dans un premier temps, nous nous sommes concentré sur les attaques déjà réalisées en expliquant leur fonctionnement et sur quelles caractéristiques matérielles elles étaient basées.

Dans un second temps nous avons présenté notre contribution qui se compose de deux attaques la CPA et l'attaque par profilage. Ces attaques nous ont permis de confirmer la faisabilité des attaques physiques avec pour cible la microarchitecture. Comme nous l'attendions les ajouts mineurs de la TZ n'ont pas eu d'effet sur la tenue de ces attaques.

Nous avons ainsi pu valider que la sensibilité aux attaques physiques de la microarchitecture rend la TEE qui repose dessus elle aussi sensible. Le cas des enclaves internes tel que proposé par ARM ne permet pas d'accroître la difficulté expérimentale.

À la suite de ces travaux, nous préconisons donc la mise en place de protections matérielles pour la microarchitecture des SoC. En particulier, ces protections doivent empêcher la fuite électromagnétique, compte-tenu des faibles répercussions produites par l'ajout de bruit, la solution la plus adaptée nous parait celle de la mise en place de boucliers.

Par la suite, nous allons nous concentrer sur les attaques physiques par perturbation.

Chapitre 5

Cas des attaques en perturbation sur *System-On-Chip*

Résumé de la partie :

Dans cette partie, nous allons nous intéresser aux attaques physiques par perturbation. La cible des attaques sont ici les SoC, nous allons nous intéresser dans un premier temps aux attaques ayant déjà été présentées dans la littérature. Afin de montrer de quelle manière des attaquants ont réussi à perturber les systèmes jusqu'à présent.

Nous avons ici identifié trois attaques, nous allons donc présenter chacune d'entre elles en effectuant un parallèle avec les parties matérielles sur lesquelles reposent les perturbations.

Dans un second temps, nous allons présenter nos travaux qui se concentrent sur la microarchitecture. Avec pour but de proposer un modèle de faute exploitable et qui permet ainsi d'exposer quelles sont les parties de la microarchitecture qui semblent sensibles.

Sommaire

Dans un premier temps, nous avons vu la capacité des attaques en observation, montrant ainsi leur pouvoir dans des attaques de types DOA ou IOA. Nous allons ici nous concentrer les attaques par perturbation.

Pour rappel, nous avons séparé les attaques physiques en fonction de la cible :

— les attaques ayant pour cible un composant ou un périphérique interne auxiliaire.

— les attaques ayant pour cible l'architecture du processeur principal.

— les attaques ayant pour cible les communications entre ces éléments.

On va ici voir plusieurs attaques générant des fautes physiques, en exploitant divers moyens et qui visent la microarchitecture. Cette revue ne se veut pas exhaustive mais permet d'effectuer un état des lieux de quelques attaques afin de montrer quelles sont les faiblesses exploitées jusqu'à aujourd'hui.

5.1 Visant des périphériques internes

Avec l'augmentation de la surface d'attaque, par apparition de nombreux composants auxiliaires, les attaques portant sur ces parties se basent sur divers mesures physiques et sur des composants diversifiés.

5.1.1 Attaque par martellement de mémoire : «Rowhammer»

L'attaque dite «Rowhammer» ou par martelage, est une attaque où une zone de mémoire va être modifiée, sans que l'utilisateur n'y ai accédé.

Il s'agit ici d'une faute induite par la faiblesse de l'architecture d'un périphérique largement utilisé dans des systèmes à base de SoC, la DRAM. En accédant de manière continue et répétitive (le martellement) à une zone de mémoire précise, on va être capable de créer une interaction avec des zones de mémoire adjacentes auxquelles on n'aura pas accédé, allant même jusqu'à la possibilité de les modifier.

Il s'agit dans ce cas d'un dysfonctionnement matériel induit par le logiciel, ce dysfonctionnement a été étudié sur des systèmes de type micro-ordinateur [KDK+14, SD15]. Récemment cependant, cette attaque a été reproduite dans le domaine des systèmes mobiles en particulier dans [vdVFL+16], où Van Der Veen *et al.* ont mis en œuvre cette attaque sur un smartphone, le Nexus 5, fonctionnant sous la dernière version d'Android au moment de leur publication (Android 6.0.1).

5.1.2 Fonctionnement de la mémoire DRAM

La mémoire DRAM est une mémoire rapide et volatile. Cela signifie que d'une part les temps d'accès aux informations qu'elle contient sont très court, mais également qu'en absence de courant électrique, toutes les informations sont perdues. Pour cela, son architecture est simple et les transferts d'informations se font dans des circuits simples.

Cette attaque repose sur l'architecture particulière des cellules de DRAM (voir Figure 5.1). Ces cellules ont pour rôle de conserver la valeur d'un bit. Cela se fait au moyen d'un transistor et d'un condensateur, qui forment donc la cellule de DRAM.

Une cellule fonctionne de la manière suivante : le condensateur sert de réceptacle pour l'information, le transistor permet quant à lui de commander la connexion du condensateur au reste du circuit. Il est donc possible d'attribuer une valeur binaire à un niveau de charge du condensateur. En général, un condensateur totalement chargé équivaut à un 1 et un condensateur vide à un 0.

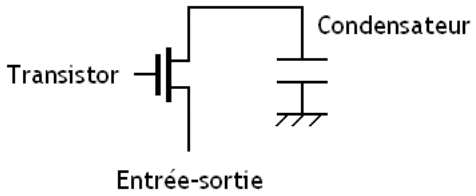

FIGURE 5.1 – Schéma simplifié d'une cellule de DRAM.

Plusieurs cellules sont chainées afin de constituer une rangée (*row*) de mémoire. Il s'agit de la plus petite unité de mémoire accessible. Cette rangée est connectée à une zone «tampon». Celle-ci possède le même nombre de cellules qu'une rangée de mémoire et permet de récupérer d'un coup tous les niveaux de charge des condensateurs de celle-ci afin de les lire.

Un élément important de cette architecture est à prendre en compte, la décharge intrinsèque du condensateur. Celle-ci pouvant causer la perte de l'information, nécessite donc une recharge à des intervalles de temps réduit, des différents condensateurs.

Selon la spécification ([ddr12]) de la DRAM de type DDR3, la durée minimale de rafraichissement est de 64ms. On a donc sur toutes ces mémoires, chaque rangée qui est totalement lue puis réécrite dans cet intervalle, assurant ainsi le maintien des valeurs sauvegardées.

5.1.3 Dysfonctionnement causé par l'attaque «Rowhammer»

C'est ce rafraichissement lié à la taille de plus en plus réduite des cellules, qui rend possible l'attaque «Rowhammer». En augmentant la densité des cellules, réduisant ainsi leur taille, les niveaux de charge entre le niveau 0 et le niveau 1 sont devenus d'une part très proche. Et d'autre part les condensateurs de différentes cellules sont plus proches les uns des autres.

L'effet électromagnétique du chargement et du déchargement d'un condensateur dans le voisinage d'un autre, cause ainsi une augmentation de la vitesse de décharge intrinsèque. De la même manière, la température des condensateurs a le même effet. Augmenter leur densité revient donc à augmenter les interactions électromagnétiques, ainsi que la température globale.

Par conséquent lorsque l'on va accéder à une cellule et donc lui faire subir un cycle de chargement, déchargement celle-ci va avoir un léger effet électromagnétique sur le condensateur de la cellule la plus proche d'elle. En multipliant les accès à plusieurs cellules, toutes celles se trouvant dans le périmètre vont être affectées.

L'accès se faisant par rangées (toute une rangée est activée pour être lue), toutes les cellules des rangées proches vont être affectées (voir Figure 5.2).

Par conséquent, les différents condensateurs de ces rangées vont se décharger plus rapidement. La quantité de courant contenue dans le condensateur devant être rechargée à un intervalle régulier, si celle-ci passe sous le seuil représentant le niveau d'un bit à 1, avant le nouveau rechargement, celui-ci conservera alors le nouveau niveau de charge et la valeur du bit sauvegardé passera ainsi de 1 à 0, il y aura bit flip.

Leur attaque est donc réussie si en moins de 64ms, assez d'accès mémoire sont réalisés sur des cellules permettant de décharger suffisamment des condensateurs dans le voisinage. Causant ainsi une modification de l'état qui sera sauvegardé lors du rafraîchissement de l'état du condensateur.

5.1.4 Exploitation du dysfonctionnement

Cependant, une phase d'identification des zones mémoires à affecter a été nécessaire afin de pouvoir exploiter le dysfonctionnement. En particulier, ils ont du :

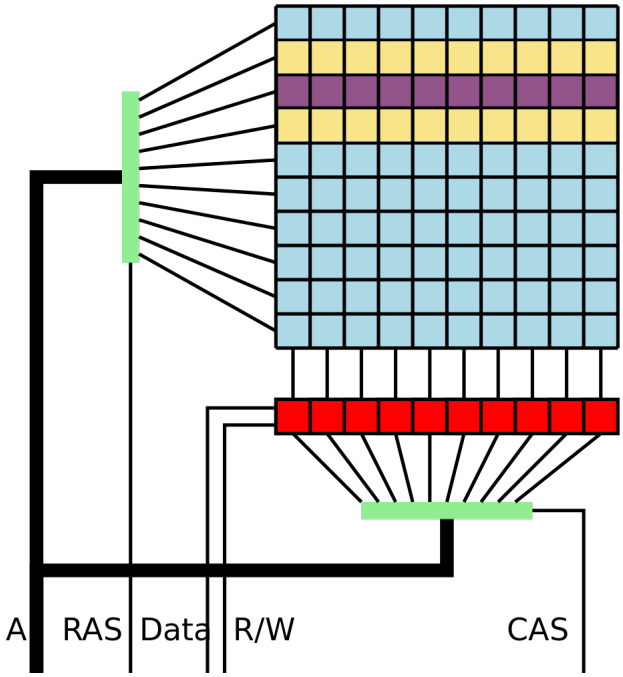

FIGURE 5.2 – La rangée en violet est celle qui est accédée, des accès répétitifs à celle-ci vont avoir un effet sur les condensateurs des rangées attenantes.

1. Identifier la reproductibilité du «bit flip».
2. Trouver une méthode afin de déclencher des accès mémoires rapides, afin d'ignorer l'effet du cache.
3. Identifier les zones de mémoire disponibles et ciblées.
4. Placer la zone de mémoire contrôlée (sur laquelle ils pouvaient effectuer de nombreux accès) sur des rangées adjacentes à des zones mémoires à affecter.

À partir de là, les auteurs ont entrepris de réaliser une montée en privilège afin de devenir utilisateur root sur le système.

Pour cela, ils ont examiné une structure de donnée `struct cred` qui contient, les informations de sécurité des applications possédant la structure. En modifiant certains bits de cette structure, ils ont donc été en capacité de modifier le groupe d'appartenance de l'application et ainsi modifier les droits d'accès de leur application pour gagner les droits root.

On constate donc que pour réaliser leur attaque, ils se sont servi d'une injection de faute «interne». En utilisant différentes caractéristiques physiques du composant, ils ont été capables de modifier des valeurs en mémoire. En sélectionnant correctement les zones mémoires à modifier ils ont pu monter en privilège.

Cette attaque a ainsi permis de montrer qu'une faiblesse dans la conception de la microarchitecture permet de déclencher des effets visibles jusque sur le logiciel et permettent dans le cas des SoC d'aller jusqu'à une montée en privilège.

La limitation de leurs travaux se situe au niveau de la surface d'attaque en premier lieu. Ainsi dans le cas où la zone de mémoire à attaquer est trop importante ou ne peut pas se trouver à proximité d'une zone mémoire manipulable, l'attaque devient impossible. En second lieu, une modification de paramètres comme la durée de rafraichissement permet de proposer une contre-mesure efficace à cette attaque. Malgré tout cette modification, force une augmentation de la consommation électrique de manière très importante, des travaux ayant démontré qu'il est nécessaire de diviser par 8 [KDK+14] la durée de

rafraichissement pour supprimer toute possibilité de mener cette attaque. Enfin ici, c'est le cas d'un composant qui détériore la microarchitecture dans sa globalité.

Il est ainsi possible d'imaginer, qu'en modifiant ce composant, on peut revenir à une situation sans faiblesse. Par la suite, nous allons voir que malgré l'identification d'une faiblesse, la surface d'attaque étant très importante, il y a davantage de possibilités pour un système d'être affecté.

5.2 Visant les communications entre ces éléments.

D'autres attaques se concentrent sur la multiplication des éléments par rapport aux μc et donc la nécessité de communications entre ces derniers ainsi que les besoins de synchronisation.

5.2.1 L'attaque «CLKSCREW»

Les travaux de Tang *et al.* [TSS17], sont centrés sur la perturbation des communications au sein du SoC. Au même titre que le «Rowhammer», il s'agit d'une classe d'attaque à part, ce sont des attaques en fautes ne nécessitant aucun accès physique au système.

Dans leur cas, le système qu'ils ont attaqué est un téléphone mobile, le Nexus 6, utilisant un SoC produit par Qualcomm pouvant fonctionner à une cadence allant jusqu'à 2.7GHz.

Le principe de l'attaque se base sur la technique Dynamic voltage & frequency scaling (DVFS). Cette technique permet d'ajuster la fréquence et le voltage de chaque processeur à la demande. Pour cela, un système doit disposer de divers mécanismes :

— Au niveau matériel, des régulateurs de fréquence et de voltage doivent être mis en place. Généralement sous la forme de Power management integrated circuit (PMIC) pour le voltage et de diverses Phase lock loop (PLL) pour la fréquence.

— Ensuite, au niveau logiciel le kernel doit être en mesure de communiquer avec tous ces éléments. C'est par exemple le cas des systèmes fonctionnant avec l'OS Linux [PS06]. Android dérivant de cet OS, on y retrouve ce mécanisme.

En particulier, l'horloge du système permet de cadencer et de synchroniser tous les éléments du SoC. La logique combinatoire de chaque élément s'effectue pendant une certaine période et le «tic» signale la fin des opérations et le début de nouvelles. La propagation correcte des données dépend donc de chaque élément, et la vitesse de propagation dans ceux-ci est variable. C'est ainsi l'élément pour lequel la propagation est la plus longue qui défini la durée de tous les autres, on le considère comme étant le «chemin critique».

Dans leur cas, le DVFS est basé sur un ensemble de duos fréquence/voltage qui correspondent à des domaines pour lesquels, le «chemin critique» reste consistant et les données y sont transmises de manière correcte. Pour induire des fautes dans ce système, ils ont donc voulu sortir hors de ces domaines d'utilisation afin de générer des fautes.

Ils stipulent avoir identifié 4 règles leur permettant de sortir de ces domaines et donc d'induire des fautes :

1. Absence de limitation matérielle dans la possibilité de choisir les couples fréquence/voltage. Ils ont ainsi pu modifier le fichier contenant ces informations et sortir des domaines, stipulés par le constructeur sans qu'un composant matériel ne limite le fonctionnement du système.
2. Réduire la tension de fonctionnement abaisse la fréquence minimale requise pour induire des fautes. Ils ont ainsi pu envisager de cibler différents éléments qu'ils pouvaient affecter.

3. L'isolation de chaque cœur permet de ne pas propager les fautes aux autres. Permettant dans cette attaque de ne cibler qu'une application s'exécutant sur un seul cœur permettant à l'OS s'exécutant sur un autre cœur par exemple de continuer son travail parfaitement.
4. La régulation s'applique de manière indifférenciée à tous les niveaux de sécurité (niveaux d'abstractions, TrustZone, etc.). Permettant par la même à un attaquant ne disposant pas d'un accès à des programmes ou des données sécurisée d'avoir tout de même un effet sur celles-ci.

Leur attaque utilise deux threads s'exécutant en parallèle sur le même cœur. Le premier est celui qu'ils cherchent à attaquer. Le second est celui qu'ils contrôlent et qui sera chargé de modifier en temps réel le couple fréquence/voltage pour induire une faute. Dans leur cas, le scénario d'attaque est celui d'un utilisateur étant capable d'installer un driver sur sa machine et donc d'être au même niveau de privilège que le kernel. Ce driver permet de contrôler les couples fréquences/voltage permettant de générer des fautes. Le thread attaquant va donc faire appel à ce driver, pour induire des fautes sur le thread victime. Celui-ci, est dans cet exemple un thread qui fonctionne une utilisant des mécanismes d'isolation comme la TZ ou Software Guard eXtensions (SGX).

Par la suite leur attaque se découpe en plusieurs phases :

1. Vidanger le cache. Cette étape permet de s'assurer que le cache du cœur ciblé ne contient pas d'autres données ou d'autre code que l'application ciblée et le code nécessaire pour lancer l'attaque. Pour cela, ils ont exécuté plusieurs fois les deux codes sur le cœur. Ceci permettant d'assurer que toutes les cases de caches pouvant contenir des données autres soient évincées.
2. Établir le profil d'exécution du code ciblé. Cette phase permet de trouver un point intéressant d'injection pour l'attaquant. Ce sera le moment où le code devra être exécuté avec un couple fréquence/voltage différent.
3. Introduire un délai, si nécessaire. En utilisant l'application d'attaque qu'ils contrôlent, les attaquants sont capables d'introduire un délai en effectuant des «no-operations». Offrant ainsi la possibilité de cibler un traitement spécifique si la latence de la mise en place est trop importante.
4. Injecter effectivement la faute. Le thread attaquant va faire appel au driver, afin de modifier le couple fréquence/voltage. Le thread victime s'exécutant sur le même cœur conservera la valeur de ce couple, il en résultera une faute si l'attaquant a réussi à trouver un couple ayant l'effet nécessaire.

Cette attaque leur a permis de créer deux schémas d'attaques :

— Retrouver une clef de chiffrement AES sauvegardée dans la TZ. Ils stipulent cependant que cette vulnérabilité ne peut être créée que s'il est possible de chronométrer l'exécution d'un thread s'exécutant en TZ, par un thread hors de la TZ.

— Lancer le chargement d'applications signées par leurs soins dans le domaine de la TZ. Avec une limitation, les lectures faites dans des domaines non sécurisés par la TZ, doivent permettre d'évincer des lignes de caches sécurisées par la TZ. Ceci afin de pouvoir réaliser une attaque en chronométrage de cache en amont, sur des applications TZ.

Prouvant ainsi les capacités de leurs attaques, les auteurs ont démontré qu'une attaque en faute introduite de manière logicielle pouvait avoir un effet significatif sur le système.

En effet, ici on remarque qu'une attaque logicielle a permis d'obtenir des fautes sur la microarchitecture. Les auteurs, ont ainsi pu rendre inopérantes des fonctions sur lesquelles reposent la chaine de confiance d'exécution. Ils ont pu créer un nouveau flot de contrôle.

En utilisant un mécanisme d'amélioration des performances, accessible au niveau logiciel, permettant notamment de réduire la consommation ou d'accélérer certains traitements, ils ont pu corrompre le fonctionnement du système.

5.3 Visant le processeur principal

Lorsque l'on s'intéresse aux attaques ayant pour cible le processeur principal, on peut prendre en exemple les travaux de Timmers *et al.* ([TSW16]).

5.3.1 Principe de l'attaque : «Controlling PC on ARM SoC»

Dans leurs travaux, ils ont été capables de modifier la valeur du registre `PC`, registre déterminant la valeur de l'instruction à exécuter. En le contrôlant on peut donc exécuter le code voulu d'une manière identique au ROP.

Contrairement aux attaques précédentes, l'injection est ici réalisée grâce à un accès physique externe. Les auteurs ont en effet utilisé une configuration, où ils modifient le voltage du cœur pour corrompre son fonctionnement.

Les autres ont mis en place deux scénarios d'attaques :

1. Une attaque sur le démarrage sécurisé du système.
2. Une attaque au runtime sur le fonctionnement d'une application isolée par le biais de la TZ.

Les deux scénarios se basent sur la corruption des instructions de chargement et de lecture des données.

5.3.2 Corruption d'instruction

Les auteurs se sont ici concentrés sur les instructions de chargement de registres depuis la mémoire. Plus précisément, les instructions `LDR Ry, [Rx]` permettant de charger une valeur dans le registre `Ry`, si elle est présente en mémoire à l'adresse contenue dans le registre `Rx`. Ainsi que l'instruction `LDMIA Rx, {Ry,...,Rz}` qui permet quant à elle de charger successivement les registres spécifiés de `Ry` à `Rz` avec des données présentes en mémoire à partir de l'adresse de base spécifiée par le registre `Rx`.

Ces instructions, ont ainsi la capacité de copier n'importe quelle donnée présente en mémoire (et pointée par le registre `Rx`) dans le ou les registres spécifiés par les instructions `LDR` et `LDMIA`. Ceci comprend également le registre `PC`, qui permet de contrôler le code exécuté.

Ils ont donc logiquement cherché à corrompre ces instructions pour spécifier le `PC` comme registre cible de données en mémoire qui étaient des instructions placées là par leurs soins.

L'instruction devant être modifiée, les auteurs ont réalisé une simulation des modifications nécessaires sur une instruction de chargement légitime pour qu'elle devienne une menace.

Dans le cas de l'instruction `LDR`, en modifiant l'expression binaire de l'instruction, une différence dans la distance de Hamming de deux permet de passer d'une instruction légitime permettant de charger une valeur en mémoire dans le registre voulu, à une instruction qui charge la même valeur dans le registre `PC`. Cela signifie qu'il suffit de modifier deux bits pour être assuré d'atteindre le registre `PC`.

Dans le cas de l'instruction `LDMIA`, la distance de Hamming pour générer une instruction affectant le registre `PC`, à partir de l'instruction de base est cette fois-ci de 1. En effet, il suffit de placer un bit à 1 pour activer le chargement dans le registre `PC`.

Ils ont ainsi pu démontrer qu'en suivant le bon modèle de faute il était possible de recréer le comportement voulu.

La mise en place effective de cette attaque passe par un FPGA qui permet de gérer l'alimentation et donc de générer le saut de tension électrique qui conduit à la faute.

Grâce à ces sauts de tension, les auteurs ont été capables de modifier le comportement global du système, avec une forte probabilité. Et même plus précisément les instructions `LDR` et `LDMIA` mais cette fois-ci avec des probabilités plus faibles. Dans le cas de l'instruction `LDR`, 0.007% des injections ont permis d'obtenir le comportement recherché. Pour l'instruction `LDMIA`, le taux de réussite grimpe à 0.27%.

Les auteurs expliquent cette divergence par la différence d'encodage et par l'écart dans la distance de Hamming «à parcourir» pour atteindre le comportement souhaité.

Cependant, il est important de noter que lors de la mise en place de leur expérimentation, les auteurs ont dû opérer certaines modifications sur la cible. En particulier, ils ont dû retirer des condensateurs en charge du maintien de la tension partout sur le SoC. Ceci dans le but de contrôler de manière la plus précise possible l'alimentation de celui-ci. Il s'agit là de la plus grosse différence entre notre approche et celles qui ont été vues précédemment, où aucune modification physique n'était apportée au système. Cependant, il convient de noter que dans leur cas, retirer ces éléments n'a pas eu d'effet sur le fonctionnement normal de leur équipement.

5.4 Récapitulatif

Technique	**Cible**	**Partie visée**	**Type d'attaque**	**Résultat**
Rowhammer [vdVFL+16]	Tous les systèmes utilisant de la DRAM DDR3, testé sur plusieurs smartphones Android	Périphérique externe : DRAM	COA	Obtenir un accès root pour son programme
Clkscrew [TSS17]	Tous les systèmes à base de SoC, testé sur un Google Nexus 6	Périphérique externe et communication entre les composants	COA, DOA et IOA	Obtenir un accès normalement réservé ou des données et des informations sécurisées sur son propre appareil (cas des applications signées et inaccessibles aux utilisateurs pour la TZ par exemple).
Controlling PC on ARM [TSW16]	Tous les systèmes suivant le jeu d'instruction ARMv7-A	Processeur principal	COA	Possibilité d'exécuter du code arbitraire présent en mémoire

Les différentes attaques qui ont été vues ici, touchent toutes à la microarchitecture. De manière plus ou importante, elles exploitent les faiblesses de celle-ci afin de créer des vulnérabilités.

Nous avons donc par celle-ci obtenu une indication qu'il était possible d'injecter des fautes dans les systèmes à base de SoC. Cependant, deux problèmes n'ont pas été adressés par ces techniques.

— Est-il possible d'injecter des fautes par un accès physique ?

— Est-il possible d'injecter des fautes sans modifier la cible ?

Les travaux ayant été vus, permettant de répondre à l'une ou à l'autre des questions mais pas aux deux. Nous avons souhaité mettre en place pour la dernière partie de mes travaux de thèse, une méthodologie permettant de répondre à ces deux questions, essentielles pour mettre à l'épreuve la résistance de la microarchitecture face à notre modèle d'attaquant.

5.5 Contribution : attaques par perturbation sur la Raspberry Pi 3 B

Au début de nos travaux, nous n'avons pas eu connaissance de travaux sur les injections de faute électromagnétiques sur SoC.

De la même manière que pour les μc, nous avons voulu exploiter les fautes comme vecteur d'introduction de vulnérabilités.

Il est alors nécessaire de disposer d'un modèle de faute suffisamment précis. Celui-ci va décrire le comportement de la microarchitecture lors de l'injection d'une faute et les répercussions sur les différentes couches d'abstractions. Il nous permet par conséquent d'évaluer les potentielles vulnérabilités exploitables par le biais des injections de fautes.

5.5.1 Enjeux et déroulement des tests

Identifier à bas niveau le comportement nécessite d'une part une bonne connaissance de la microarchitecture et d'autre part un contrôle fin du système. Cependant, cela entre en contradiction avec notre modèle d'attaquant. Ici, nous allons donc nous placer dans le cas d'un attaquant avec davantage de pouvoir, en particulier celui de pouvoir disposer d'un accès très fin au système. De la même manière que lors de l'écoute dans le cadre de l'attaque par profilage sur SoC, nous considérons que l'attaquant dispose d'un composant identique sur lequel il peut réaliser des tests avant de réellement mettre en place son attaque.

Par conséquent, nous avons souhaité mettre en place une série de tests permettant d'identifier ce modèle de faute en exploitant les informations disponibles dans les documentations fournies par ARM [ARM14, ARM17, ARM09].

Ces tests ont pour but de proposer une projection du modèle de faute sur un maximum de niveau d'abstraction. De cette manière, nous avons surveillé le plus de données possibles après l'injection et en les comparant avec les valeurs attendues ou décrites dans les documentations ARM.

Du fait de la grande quantité d'informations disponible et donc de la multitude d'états possibles, dans la suite de nos travaux, nous ne présenterons que nos hypothèses en les motivant selon les données présentes dans ces documentations.

Les SoC disposent de davantage de composants que les μc. Par conséquent, il est possible d'affecter un plus grand nombre d'entre eux et les répercussions visibles du point de vue de l'attaquant peuvent être diverses.

Nous avons donc dans un premier temps, tenté de générer des fautes afin de valider la faisabilité d'une attaque de ce type sur SoC. Le but ici est uniquement de valider qu'une faute a un effet observable et que des injections peuvent donc être utilisées comme vecteur d'introduction de vulnérabilités. On rappelle que l'attaquant n'a accès qu'à la couche logicielle lors de son attaque.

Par la suite, nos tests ont porté sur la définition la plus précise possible du modèle de faute.

5.5.2 Complexité dans l'inférence du modèle de faute

Les modèles de fautes génériques sont des projections sur les niveaux d'abstractions logicielles de l'effet d'une faute. Cette projection est généralement reportée sur le jeu d'instruction, comme dans le cas des μc que nous avons vu précédemment et qui traduisent la faute comme l'exécution d'un «no-operation».

Sur les SoC, la présence de nombreux composants, plus particulièrement du nombre de transistors et de connexions, entraine une surface d'attaque d'autant plus grande. Par conséquent, une faute peut se produire dans l'un de ces composants et le modèle sera plus complexe à retrouver.

Pour exemple une modification d'un bit dans un registre de configuration du snooper du bus, peut entrainer une cascade de conséquences. Il peut devenir possible de limiter l'écriture sur le bus à un seul composant, ainsi le système peut devenir inopérant. De la même manière, une faute peut entrainer l'apparition d'une boucle permettant à un périphérique d'être contacté à la place d'un autre.

Pour éviter la possible explosion et la complexification des combinaisons de comportement, nous avons souhaité isoler les composants dans nos tests. Par conséquent, chaque test visera spécifiquement un composant afin de limiter l'effet des autres en cas de faute.

5.5.3 Cible des tests

Notre cible pour ces travaux a été la Raspberry Pi 3 B. Cette cible a été choisie, pour sa ressemblance architecturale avec les smartphones.

Nous nous sommes concentrés sur la microarchitecture du SoC dont voici les caractéristiques :

— SoC Broadcom BCM2837

— Cœurs de calculs : 4*Cortex-A53

— Fréquence d'horloge : 1.2GHz

— ISA ARMv8-A

— 4*Cache L1 séparés Instructions et Données

— Cache L2 unique et partagé entre les cœurs

Certains composants notamment restent non documentés, étant des éléments de propriété intellectuelle du fondeur ou du concepteur. Ils ne seront donc pas détaillés dans nos travaux.

Ainsi, la présence de plusieurs caractéristiques comparables à celles des smartphones nous a ainsi permis de nous assurer que l'on restait dans un environnement existant dans le commerce. Cependant, il existe d'autres composants qui ne sont pas directement sur le SoC, parmi lesquels on peut citer, les contrôleurs Universal Serial Bus (USB) ou la RAM. Nous avons donc considéré qu'ils n'ont pas été affectés par l'injection de faute, ils ne produisent ainsi pas d'effet sur la microarchitecture.

5.5.4 Montage expérimental

Dans l'optique d'injecter des fautes au sein de notre système, nous avons mis en place un montage de base, qui permet d'injecter des fautes et de récupérer les informations sur ces dernières, en vue de l'analyse.

Un schéma du montage est visible sur la Figure 5.4.

Il est possible d'envoyer des commandes au logiciel interne par l'intermédiaire de la connexion UART relayée par le composant FTDI Friend. De plus, le montage repose sur un axe XYZ, permettant ainsi de déplacer la sonde à l'endroit désiré sur la puce, et d'atteindre ainsi la reproductibilité des expérimentations. Nous avons utilisé les sondes Langer *RF U 5-2* et *RF B 0.3-3*.

5.5.5 Tests préliminaires

Pour trouver un modèle de faute, nous avons débuté par une phase de découverte. Celle-ci est nécessaire afin de valider la faisabilité de l'injection de faute sur ce type d'architecture.

Pour cela, nous avons mis en place le système embarqué à attaquer sur notre banc de test 5.3.

Sur celui-ci la cible n'était pas une application à proprement parlé. Nous avons souhaité attaquer la séquence de démarrage sécurisée de l'OS.

Ceci pour diverses raisons, la première est que la séquence de démarrage consiste en une suite assez longue d'opérations diversifiées. Par conséquent, une injection non ciblée, a davantage de chance de

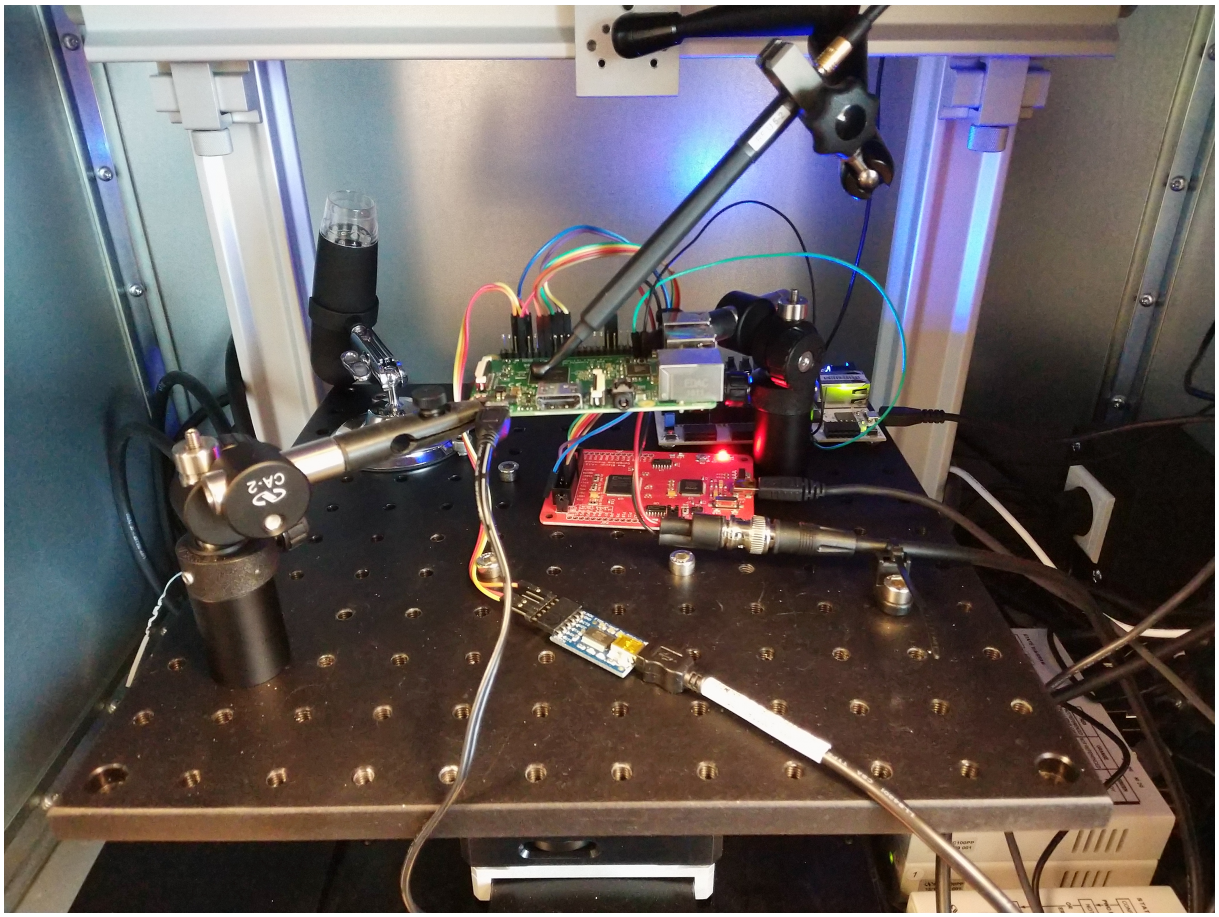

FIGURE 5.3 – Montage classique pour l'injection de faute sur Raspberry Pi 3

produire un effet visible. Cela est d'autant plus vrai que la sécurisation consiste en une de vérification de la signature du code. Une autre raison à ce choix est que le système étant en train de démarrer, toute la hiérarchie de la mémoire est à compléter. Ainsi, les caches sont vides et chaque accès à une donnée va produire un «miss», permettant d'une part d'allonger le temps d'exécution et donc de nous permettre d'explorer temporellement de manière plus large. D'autre part, de nombreux transferts ont lieu, nous avons émis l'hypothèse qu'il serait plus évident d'en corrompre un ou plusieurs et donc d'obtenir un effet visible.

En nous concentrant sur le début de la séquence de démarrage, nous avons réussi à obtenir une faute. Celle-ci nous a permis de refuser la vérification de la signature d'un programme pourtant correctement signé.

Un second test a été réalisé permettant d'identifier spatialement les zones d'injection. La Figure 5.5 montre en jaune les zones où l'injection de faute a produit un crash. En modifiant les paramètres d'injection, ces zones ont été conservées pour injecter. Par conséquent, une zone de crash devient une zone sensible aux fautes selon les paramètres utilisés.

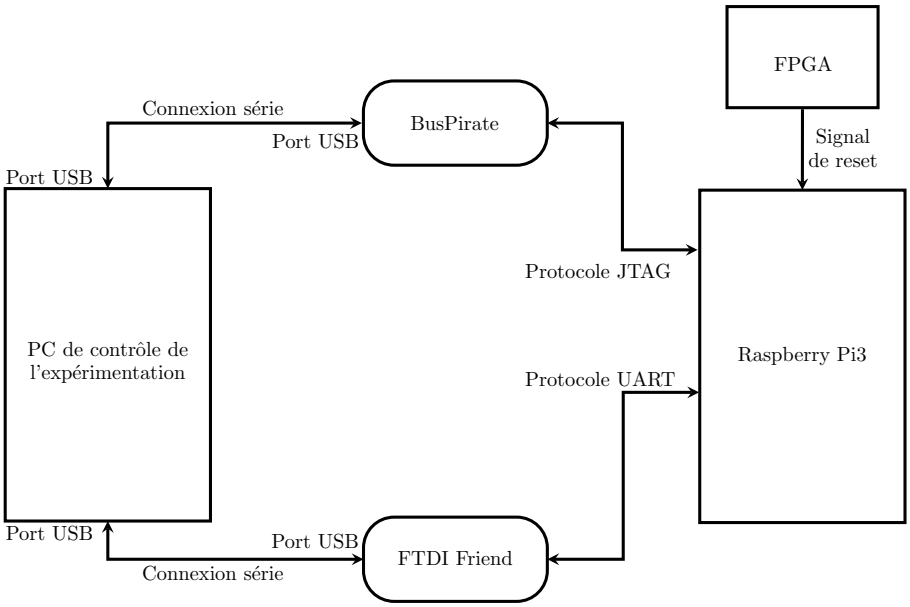

FIGURE 5.4 – Schéma de connexion classique entre la Raspberry Pi 3 et l'ordinateur de contrôle

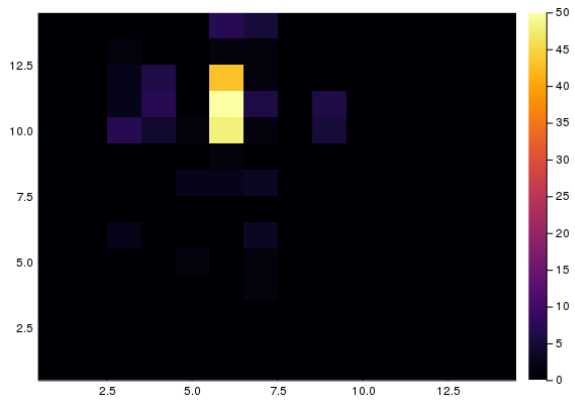

FIGURE 5.5 – Zone de crash lors de l'injection de faute sur le SoC BCM2837 de la Raspberry Pi3.

5.5.6 Phase d'identification du modèle de faute

Après avoir mis en lumière la possibilité d'injecter des fautes dans le système, nous avons voulu les exploiter. Pour cela, il faut passer par une phase d'identification du modèle de faute. À partir de celui-ci il est possible de mettre en place des vulnérabilités et d'exploiter les injections pour de véritables attaques.

D'une part, afin d'avoir un contrôle le plus précis possible, nous sommes partis sur l'exécution d'un logiciel bare metal. Le but est de tester la vulnérabilité de la microarchitecture, il nous fallait donc pouvoir interagir avec chaque partie, et avoir la possibilité d'obtenir des informations très bas niveau simplement.

D'autre part, le bare metal permet également d'effectuer des diagnostics post mortem plus simplement

qu'avec un OS.

En effet, certains registres d'états du SoC (en particulier ceux de mise en place et de la gestion de la hiérarchie de la mémoire) ne sont pas reconnus directement par les différents OS, dans d'autres cas ils sont réservés à des configurations précises (niveau de privilège, état mémoire etc), et enfin l'utilisation d'un OS modifie l'état des registres en plaçant une couche d'abstraction supplémentaire, nous avons donc préféré ignorer cette couche plutôt que de créer des outils permettant de la masquer.

Notre logiciel repose donc sur un socle basique de démarrage du SoC, dans le cas de la Raspberry Pi3, le démarrage est en 2 parties :

1. Démarrage non contrôlable du GPU
   - Mise en place de la mémoire : copie des données depuis la carte SD vers la mémoire RAM.
   - Mise en place de certains registres d'environnement pour le lancement des cœurs A53.
   - Démarrage des périphériques externes.
   - Mise sous tension des cœurs A53.
2. Démarrage contrôlable des cœurs de calcul.

À partir de la seconde phase de démarrage, les cœurs de calcul exécutent un code écrit par nos soins, il consiste en :

1. Une sélection des cœurs à démarrer.
2. Une désynchronisation des cœurs (pour simplifier la gestion des accès concurrents).
3. Une allocation de la pile pour les différents cœurs (taille et emplacement en mémoire).
4. Un branchement vers notre OS maison.
5. Une assignation du vecteur interruption, permettant ainsi d'avoir différents comportements en cas de crash.
6. Lancement du logiciel permettant de réaliser nos tests.

Une fois le démarrage effectué, nous avons donc pu mettre en place les divers tests. Le but étant l'identification du modèle de faute, nous avons tenté d'explorer un maximum de paramètres, cependant compte-tenu de leur nombre, en particulier les paramètres de positionnement spatial pour l'injection et la puissance de celle-ci, tous n'ont pas été exploré.

La récupération d'informations de bas niveau s'est faite au moyen d'une connexion utilisant le protocole JTAG. La traduction s'est effectué au moyen du logiciel OpenOCD.

Cependant, nous avons été confrontés à un certain nombre de limitations comparativement à l'utilisation qui a été faite d'OpenOCD auparavant.

Parmi celles-ci on peut noter :

- Impossibilité de placer des points d'arrêt.
- Lisibilité réduite de la mémoire.

Ainsi ces deux contraintes expliquent certaines de nos décisions de programmation. Elles seront expliquées au cas par cas.

Dans cette partie, les hypothèses retenues sont issues des informations telles quelles sont présentées dans les documentations d'ARM pour l'ISA ARMv8 [ARM17], l'architecture Cortex-A53 [ARM14] et pour la TrustZone [ARM09].

Après chaque cas de test, nous tenterons de résumer nos observations et nos hypothèses qui nous ont amené à mettre en place le cas suivant.

Exploration du flot d'exécution

Dans un premier temps, nous avons souhaité reproduire une faute semblable à celle observée dans le cas des μc. Nous avons donc utilisé un seul cœur du processeur et mis en place notre environnement de test.

Mise en place Le processeur du SoC a été placé dans la configuration visible sur la Figure 5.6. Le cœur ici testé, effectuait 2 opérations. Un démarrage classique, et le lancement d'une application réalisant un chiffrement AES.

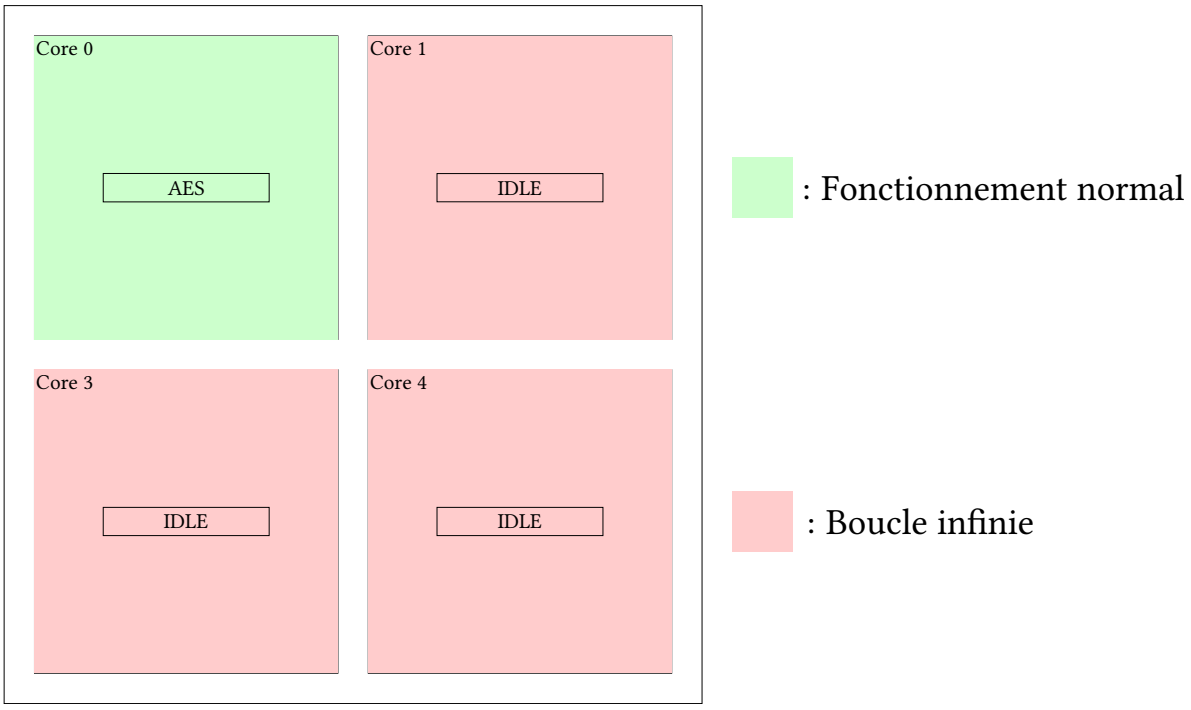

FIGURE 5.6 – Schéma de l'architecture logicielle du système pour le premier cas de test.

Nous avons souhaité obtenir un chiffré incorrect lors de l'exécution de cette application.

Les autres cœurs ont été placés dans une boucle infinie, sans effet sur la mémoire et sans avoir activé les caches de ces cœurs. Cela relève de l'impossibilité de les mettre totalement hors tension, ils ont cependant été «isolé», limitant ainsi leur effet sur l'analyse du reste du système. La «mise en sommeil» des autres cœurs à ainsi lieu, au tout début de la seconde phase de démarrage.

Observations Divers comportements ont été observés :

— Crash quasi instantané → `Data abort`.

— Modification du registre ELR.

— Modification du chiffré en sortie.

Le crash a causé un passage par le vecteur d'interruption correspondant à un `Data abort`, en se référant à la documentation (page D1-1811 de la documentation [ARM17]) :

Extrait de la documentation ARM

... These EC values are used for the following exceptions if the exception is generated by a data access : MMU faults ; alignment faults other than SP alignment faults and PC alignment faults ; synchronous external aborts, including synchronous parity or ECC errors ...
Ces codes d'exceptions sont générés dans les cas suivants : faute sur la MMU, faute sur l'alignement de l'adresse contenue dans des registres autre que le `PC` ou `SP`, ou enfin si une erreur externe touchant l'intégrité de la mémoire est transmise ...

Une faute au sens d'ARM est ici un dysfonctionnement quelconque, il n'est pas forcément lié à une injection de faute physique.

Afin d'obtenir des informations supplémentaires, nous avons pu utiliser plusieurs registres d'état. Cependant, nous nous sommes heurtés à des limitations. Le fait d'être en monocœur rend impossible l'utilisation d'un autre cœur pour observer l'état de ces registres et nous le transmettre.

Il nous était impossible de stopper l'exécution à l'endroit désiré, OpenOCD étant incapable de mettre en place des points d'arrêt. Par conséquent, l'état de ces registres à ces moments exacts nous était inconnu. Nous avons dû nous fier uniquement à sa valeur lorsqu'elle était récupérée après la fin de l'exécution normale.

Enfin, OpenOCD ne nous a pas offert pas la possibilité d'obtenir l'état de ces registres par l'intermédiaire de la connexion JTAG.

Par conséquent, une mise à niveau d'OpenOCD a été envisagée, mais n'a pas été réalisée ne disposant pas de la documentation adéquate permettant de connaitre la commande JTAG à ajouter pour avoir accès à ce registre. La raison exacte de l'exception reste donc inconnue à ce niveau.

Le registre `ELR` quant à lui est directement accessible, par le biais de la banque de registres, celui-ci a été régulièrement (pas lors de toutes les injections, ni selon un schéma particulier) modifié lorsque des fautes ont été injectées. Sa fonction est de contenir l'adresse de retour après une exception, dans notre cas il contient l'adresse suivant l'instruction où a eu lieu l'exception. Ainsi, il nous était possible de savoir que différentes parties du programme pouvaient être affectées.

Enfin la modification du chiffré de sortie lorsque la faute n'avait pas pour effet de causer un crash, nous a permis de savoir qu'un autre effet avait lieu permettant de modifier le chiffré mais sans pour autant savoir à quel niveau se produisait la modification. De plus, lorsque l'on demandait à nouveau le chiffrement d'une valeur, sans modifier les entrées, la même valeur fauté apparaissait. Par contre, lorsqu'une nouvelle demande de chiffrée était effectuée en changeant les valeurs en entrée (clef et texte clair), le chiffrement était correct. Laissant penser que la mémoire, lorsqu'elle n'est pas rechargée conserve la faute mais que lorsque celle-ci change l'effet de la faute disparait.

Conclusion Nous n'avons pas pu reproduire le comportement observé sur μc. À savoir une traduction directe du modèle de faute sur le flot d'exécution.

Il nous a été possible de générer une faute et d'avoir un effet significatif sur le fonctionnement d'une application complexe sur le système. Cependant, l'injection n'est pas ciblée, et la sonde utilisée rayonne sur une grande surface de la puce, à ce stade nous n'avons donc pas la possibilité de valider quelle partie de la microarchitecture est en cause et donc quel est le modèle de faute.

Néanmoins, nous avons pu émettre diverses hypothèses en recoupant les informations présentes dans les différentes documentations.

Observations	Hypothèses
Crash	— Erreur de traduction entre adresses physiques et virtuelles. — Erreur d'alignement sur `SP` ou `PC`. — Erreur sur la source externe de mémoire (RAM ou autre).
Modification `ELR`	— Faute affectant l'exécution de plusieurs instructions.
Modification valeur en sortie	— Faute permettant une modification de la mémoire. — Faute non-constante, peut-être «effacée» si l'on réécrit en mémoire.

Table 5.1 – Hypothèses après la première campagne de test.

Observations et hypothèses

Les hypothèses du tableau 5.1 sont basées principalement sur les informations glanées dans la documentation, aucun test comparatif n'ayant permis de confirmer ou d'infirmer une de nos hypothèses.

Cependant, elles semblent toutes pointer une modification de la mémoire ou des opérations ayant un lien avec celle-ci. En particulier, parce qu'à aucun moment, nous n'avons été confrontés à des modifications d'instructions ou à des sauts comme dans le cas des injections de fautes sur STM32 dans le cadre de l'étude des fautes sur μc.

Ensuite parce que les raisons du crash et la possibilité de retrouver un fonctionnement normal en modifiant la mémoire, ajoutaient des éléments à cette hypothèse.

Ceci nous a conduit à nous concentrer sur cette piste, adaptant les tests suivants.

Approfondissement et passage au multicœur

Après avoir pris en compte nos hypothèses, une question se posait : À quel niveau de la mémoire doit-on réécrire pour faire disparaitre la faute ? Ceci nous permettrait alors de savoir à quel niveau de la hiérarchie mémoire la faute apparait.

Mise en place Dans ce cas-ci, nous avons souhaité obtenir plus d'informations venant du cœur affecté par la faute. Nous avons donc mis en place un comportement multicœur. De plus, nous voulions également obtenir une gestion plus fine de la mémoire afin de la manipuler selon nos désidératas.

Cependant, l'utilisation d'un cœur permettant d'observer et de remonter différentes informations sur l'état des autres, pose un problème important, celui d'être également affecté par la faute injectée.

Il nous a donc fallu adapter son fonctionnement. Nous avons en plus remarqué que les autres cœurs n'ont pas été affectés lors de leur «dysfonctionnements» au cours du premier test. Nous avons donc voulu placer le cœur de contrôle dans une boucle d'attente lorsque les cœurs à attaquer étaient au travail.

Nous avons donc sélectionné une architecture logicielle telle que présentée sur la Figure 5.7.

Dans ce cas, nous avons placé le cœur de contrôle dans une boucle infinie lorsqu'il était en attente de commande, et aucune commande n'était envoyée lorsque l'injection avait lieu.

Les trois autres cœurs ont été placés dans une boucle d'exécution par le cœur de contrôle ou d'attente lorsque celui-ci ne leur donnait pas d'ordre, comme visible sur la Figure 5.8.

1. Vérifier l'état de la variable `start` indiquant un ordre de départ de la part du cœur de contrôle
2. Effectuer un chiffrement AES selon un texte clair et une clef placée en mémoire au préalable par le cœur de contrôle.

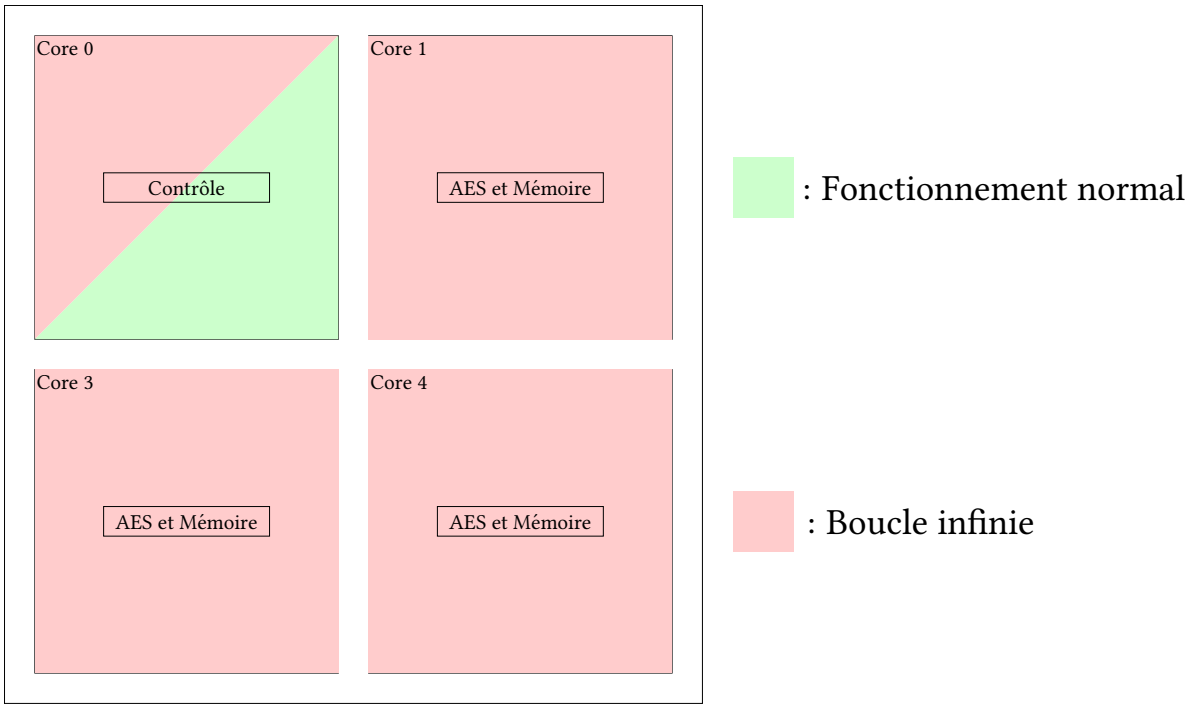

FIGURE 5.7 – Schéma de l'architecture logicielle du système pour le deuxième cas de test.

3. Mémoriser le texte chiffré dans une zone de mémoire qui pourra être lue par le cœur de contrôle.
4. Vérifier de nouveau l'état de la variable `start`.

Avec la configuration mémoire suivante :

Sur la Figure 5.9, chaque case représente 32 bits de données. Les cases sont reparties de la manière suivante :

0. Variable `start`.
1. Variable d'accusé de réception du message de départ du cœur 1, contenant la valeur d'un registre d'état le `MPIDR`, ou du `SPSR` en cas de passage dans une exception.
2. Variable d'accusé de réception du message de départ du cœur deux, contenant la valeur d'un registre d'état le `MPIDR`, ou du `SPSR` en cas de passage dans une exception.
3. Variable d'accusé de réception du message de départ du cœur 3, contenant la valeur d'un registre d'état le `MPIDR`, ou du `SPSR` en cas de passage dans une exception.
4. Pour chaque cœur de travail, la valeur de la clef de chiffrement utilisée. Sur 128 bits.
5. Pour chaque cœur de travail, la valeur du texte clair. Sur 128 bits.
6. Pour chaque cœur de travail, la valeur du chiffré résultant. Sur 128 bits.
7. Un compteur du nombre d'exécutions et d'autres registres d'état pour suivre l'exécution.

Lors de la phase de démarrage, le cœur de contrôle est chargé de désynchroniser les différents cœurs, ceci afin que les accès à la mémoire soient non-concurrents. Il est alors possible avec une seule injection d'affecter les cœurs à des moments différents, sans nécessité de poser des points d'arrêt.

Ensuite ce même cœur de contrôle reste dans sa boucle d'attente de commande lorsque l'on injecte la faute. Ceci permettant de limiter au maximum le potentiel effet d'une faute.

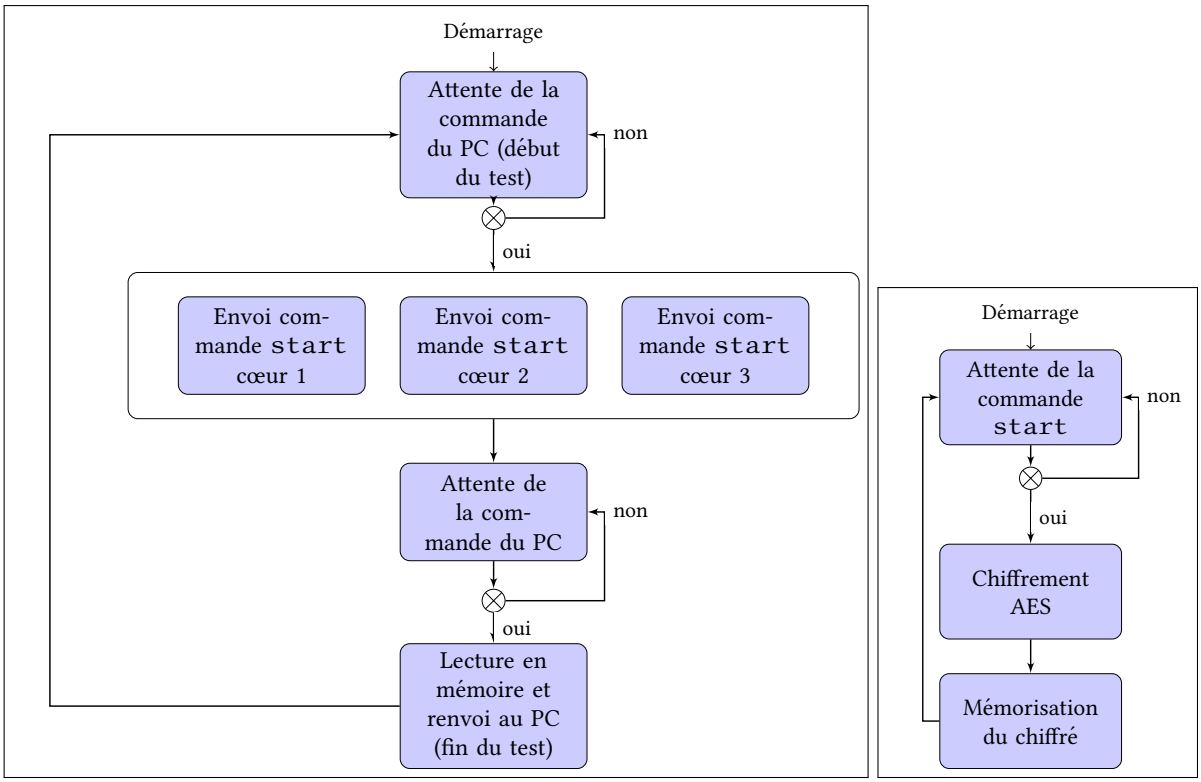

FIGURE 5.8 – Graphes de flot de contrôle simplifiés pour le deuxième cas de test. À gauche pour le cœur de contrôle. À droite les autres cœurs.

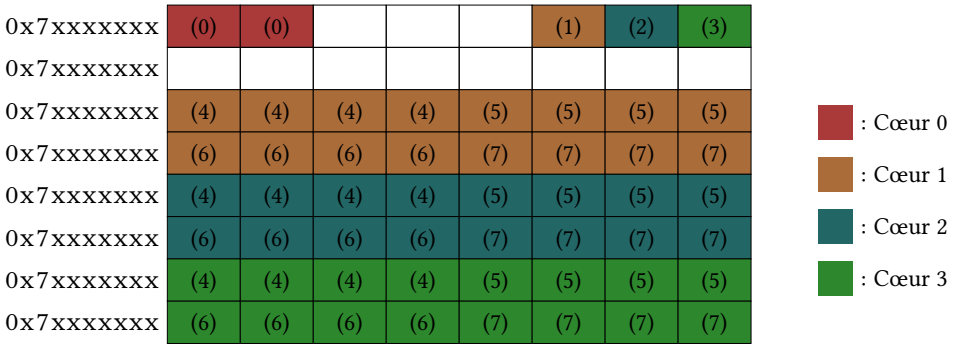

FIGURE 5.9 – Schéma de la mémoire. Chaque cœur ne peut écrire que dans sa zone colorée. Le cœur de contrôle est amené à toutes les lire.

Les données que nous avons souhaité récupérer ont été conservées à deux endroits. Dans les variables d'origine tout d'abord, puis nous avons également spécifié une zone mémoire éloignée des autres variables afin que les 4 cœurs (contrôle et travail) y écrivent des copies.

Cette zone mémoire contiguë est séparée en une zone totalement partagée, où tous les cœurs lisent et écrivent à tour de rôle pour rendre compte de leur état. Il s'agit ici d'une copie d'un registre d'état du processeur le `SPSR` qui permet de connaitre l'état dans lequel se trouvait un cœur lorsqu'il a pris une exception et d'un registre d'identification le `MPIDR` qui permet d'identifier un cœur unique.

La seconde zone de mémoire a été placée un peu plus loin pour conserver un alignement. Elle est à son tour séparée en 3 unités. Où cette fois-ci, chaque cœur de travail disposait de son espace où lui seul allait écrire. Dans cet espace, chaque cœur plaçait les variables qu'il allait utiliser pour son chiffrement (la clef et le texte clair), ainsi que le résultat, la valeur du texte chiffré. En plus de ces variables, nous avions également ajouté une variable de compteur du nombre d'exécutions pour savoir combien de calcul chaque cœur aurait effectué avant son arrêt.

Cette disposition de manière contiguë permet également de profiter du mécanisme de cohérence des caches. Les SoC ARM et en particulier le Cortex-A53 comporte une logique de cohérence assurée par le Snoop control unit (SCU). Cet élément permet d'assurer que dans la hiérarchie de la mémoire, les caches de données de niveau L1 des différents cœurs contiennent la même valeur pour les lignes de cache qui seraient partagées. En cas d'écart, le SCU peut réaliser les transferts entre les caches directement.

Ainsi placer les éléments dans des zones continues permet de nous assurer qu'une ligne de cache contient à la fois des données venant de plusieurs cœurs, ainsi le rôle de le SCU sera d'assurer la cohérence entre ces lignes.

Observations De nouvelles observations sont apparues :

— Crashs → `Data abort`.

— Modifications du registre `ELR`.

— Modification des données stockée en mémoire.

— Apparition en mémoire de valeurs aléatoires.

— Décalage dans la mémoire des données devant être lues.

Nous avons constaté les deux mêmes événements que lors des tests de la première campagne. À savoir des crashs et une modification du registre `ELR`. Les crashs ont généralement eu lieu sur les cœurs de travail et pas sur celui de contrôle.

En utilisant donc la connexion JTAG, nous avons pu avoir accès à l'état de la mémoire. On y a observé, une modification de la zone d'écriture entre un fonctionnement normal et le même fonctionnement contenant une faute. Les données stockées ont été décalées de plusieurs bits, avec un remplissage par des bits à 0. Des artefacts apparaissent également, soit sous la forme de suites répétées des mêmes octets, ou sous la forme de répétitions à des adresses alignées.

En utilisant des espaces de mémoires proches, nous avons ainsi pu activer le SCU. S'agissant d'un élément peu documenté sur son fonctionnement, nous nous en sommes assuré en effectuant des essais d'écriture et de lecture. Une écriture par le cœur 2 dans sa zone entrainant ainsi une recopie des données, sur le cache du cœur 3, lorsque celui-ci cherchait à lire une information dans sa propre zone mémoire.

Lors de certains tests, nous avons pu remarquer un autre phénomène qui reste sans explication. À savoir que le cœur de contrôle s'est parfois retrouvé à afficher des informations incohérentes alors que son fonctionnement est censé être correct. Ainsi, parmi les messages qu'il nous renvoie, il a pu à certains moments envoyer des messages sans aucun rapport avec la commande envoyée. Laissant penser également que malgré l'exécution d'une boucle, il pouvait se trouver lui aussi affecté. Les informations renvoyées à l'écran étaient des messages programmés donc présent aussi en mémoire.

Conclusion Compte-tenu de l'effet visible de la faute, celle-ci semblait une nouvelle fois en rapport avec la mémoire. Au travers de ces tests, nous avons pu prouver que l'effet n'est pas uniquement lié à un seul cœur mais touche de manière aléatoire un ou plusieurs cœurs au cours de leur traitement.

Après cet ensemble de tests, nous n'étions cependant pas en mesure de conclure sur la cause de la faute. Cependant, l'apparition d'artefacts en mémoire nous a fait considérer deux cibles potentielles :

— Le SCU

— La MMU

Le SCU, tout d'abord est chargé de recopier les données entre les différents caches des différents cœurs, or en analysant les agencements mémoires des différents cœurs, nous avons remarqué que le cœur de contrôle, le 0 et parfois un ou plusieurs des cœurs de travail, ne disposait pas de la copie des artefacts visibles sur les autres. Nous avons alors émis l'hypothèse que la faute corrompait ce composant lui faisant copier des données vers des zones aléatoires, expliquant ainsi les artefacts.

La MMU, ensuite est chargé de réaliser et de conserver la traduction des adresses physiques vers des adresses virtuelles pour les demandes d'écritures ou de lectures effectuées par les cœurs. Ainsi, nous avons émis l'hypothèse qu'une modification de son fonctionnement entrainerait une modification de l'adresse d'écriture des données et pourrait expliquer le décalage des données que l'on a observé. Cette hypothèse est renforcée par le fait que ce soit uniquement les données copiées par les cœurs de travail (et donc pas celles manipulées par le cœur de contrôle) qui ont été modifiées.

Cependant, le remplissage avec des 0 à l'endroit des décalages, et les valeurs des artefacts, restent inexpliquées. Pour les artefacts, une piste possible est présente dans un autre espace mémoire contenant des données placées par le compilateur, on y retrouve des séquences identiques. Une copie de ces séquences dans la zone des données a été envisagée mais n'a pas pu être confirmée à ce niveau de test.

Observations et hypothèses

Observations	Hypothèses
Crash	— Erreur de traduction entre adresses physiques et virtuelles. — Erreur d'alignement sur `SP` ou `PC`. — Erreur sur la source externe de mémoire (RAM ou autre). } `MMU` ?
Modification `ELR`	— Faute affectant l'exécution de plusieurs instructions. → `MMU` ?
Modification valeur en sortie	— Faute permettant une modification de la mémoire. — Faute non-constante, peut-être «effacée» si l'on réécrit en mémoire.
Apparition d'artefacts en mémoire	— Effet sur le `SCU` entrainant des copies intempestives.
Décalage des données en mémoire	— Effet sur le `SCU` et la `MMU`, combinant copie et modification d'adresse.

Table 5.2 – Hypothèses après la deuxième campagne de test

Ces tests, nous ont permis d'approfondir l'effet de la hiérarchie de la mémoire. Nous avons à ce niveau réussi à identifier deux candidats potentiels, avec des hypothèses fortes expliquant les phénomènes observés à bas niveau, et donc leur effet à plus haut niveau observé lors des premiers tests.

Ces deux candidats disposent aussi d'une surface d'effet importante, leur comportement dépend d'un ensemble d'opérations de base et de registres internes.

Il semblait nécessaire de devoir essayer d'isoler plus spécifiquement leur comportement afin d'isoler leurs effets, ce que nous avons entrepris par la suite.

Mise à l'épreuve de la MMU et du SCU

Dans les éléments mis en cause, il existe diverses possibilités de perturbation. Le SCU ne dispose pas d'une documentation très détaillée, afin de rester en accord avec notre modèle d'attaquant, nous avons donc considéré que les opérations ne pouvaient pas être découpées et que le comportement décrit dans la documentation était exactement celui ayant cours, comme une opération non atomique.

Pour ce cas, nous avions également noté auparavant qu'un rafraichissement de la mémoire pouvait entrainer la disparition de l'effet de la faute, nous avons donc investigué ce phénomène afin d'évaluer s'il était possible de retrouver à quel niveau de la hiérarchie la faute avait un effet.

Mise en place Pour ce faire, nous avons mis en place le système dans l'état visible sur la Figure 5.10.

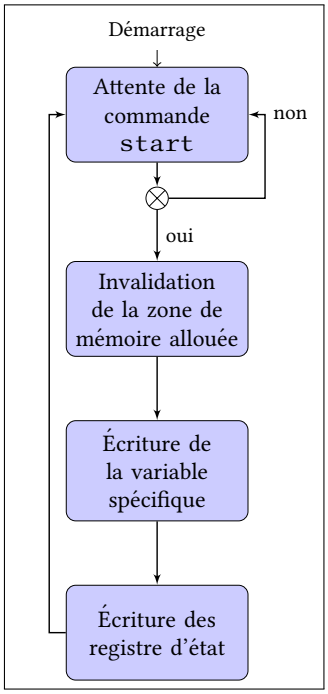

FIGURE 5.10 – Graphe du flot de contrôle simplifié des cœurs de travail.

Le cœur de contrôle dispose toujours du même rôle de communication, de gestion des autres cœurs et d'acquisition des informations de débogage.

Sur les cœurs de travail cette fois la partie AES disparait, nous l'avons remplacé par des mouvements de données en mémoire de diverses natures. Nous avons également donné à chaque cœur la possibilité de transmettre les informations venant de sa propre MMU.

Par conséquent les cœurs de travail, ont la possibilité d'écrire une suite de bits prédéfinis. La configuration mémoire sur la Figure 5.11, change donc légèrement.

On se retrouve cette fois avec chaque cœur écrivant dans sa zone une valeur qui est unique et différentiable aisément. Identifiée sur la Figure 5.11 par les cases `(4)`, `(5)` et `(6)`. Les cases `(7)` contiennent quant à elles des registres d'état de la mémoire ainsi qu'un compteur du nombre d'exécutions.

Observations L'injection de faute était toujours réalisée de manière aléatoire au cours du temps, pendant le traitement des 4 cœurs.

Dans ce cas, nous n'avons pas été capables d'observer de modification du cache.

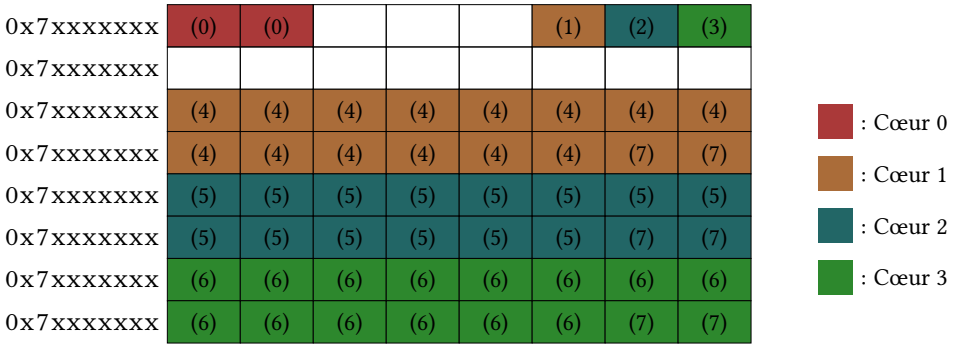

FIGURE 5.11 – Schéma de la mémoire. Chaque cœur ne peut écrire que dans sa zone colorée. Le cœur de contrôle est amené à toutes les lire. Les cœurs de contrôle peuvent en évincer certaines.

L'ajout d'une éviction de cache L2 a permis de retrouver un état de mémoire cohérent.

Conclusion Dans ce cas, les observations restent les mêmes que précédemment. L'éviction du L2, nous a permis d'avoir une plus grande confiance dans le fait que nous ciblions bien le SCU.

Tentative d'exploitation

Nous avons noté que le SCU fonctionnait de manière non documentée. Nous souhaitions savoir s'il ne pouvait pas être un vecteur d'introduction de vulnérabilité. En effet, les copies sont effectuées de manière automatique, sans que les cœurs de calcul n'aient d'emprise directe sur ces transferts. Nous avons donc voulu tester la vérification de contrôle d'accès. En d'autres termes, si lorsqu'une donnée est copiée d'un cache $L1_x$ vers un cache $L1_y$ les droits d'accès doivent être les mêmes, ou s'il est possible d'effacer une donnée de plus haut privilège par une donnée de plus bas privilège.

Mise en place Pour cela, nous avons placé les 4 cœurs dans des niveaux de privilèges différents. Dans le but de voir si le comportement de copie se produisait toujours en injectant une faute. Ceci constitue une potentielle vulnérabilité, dans le cas où il serait par exemple possible de modifier en mémoire une variable utilisée pour du contrôle d'accès comme un code PIN.

Chaque cœur dispose de la même zone d'écriture que précédemment, mais cette fois, chaque cœur se trouve dans un niveau de privilège différent :

#	Niveau de privilège	Rôle
0	$EL3$ (Mode moniteur)	Contrôle et communication
1	$EL3$ (Mode moniteur)	Copie en mémoire
2	$EL1$ (Mode OS)	Copie en mémoire
3	$EL0$ (Mode utilisateur)	Copie en mémoire

Observations Lors de l'injection de faute, il n'a pas été possible d'obtenir une copie des informations sans un crash. Ainsi seul le cœur de contrôle et le cœur 1 étaient encore en état de fonctionnement.

Les deux cœurs inactifs n'ont pas pu écrire du tout dans la zone, ce qui laisse à penser qu'il s'agissait d'une requête d'écriture de leur part qui a été rejetée, le code d'erreur des différents cœurs nous a confirmé cela. Nous avons donc eu un décalage sur les données en mémoire de la même manière que dans les cas précédents, avec cette fois une impossibilité de gérer les autres cœurs.

Après avoir observé la mémoire, nous avons remarqué que les données n'ont pas été copiées ainsi le système est protégé contre ce genre de copie rendant ce genre d'attaque inopérante.

Conclusions Cette campagne de test, nous a permis de préciser davantage le comportement le SCU comme n'étant pas un simple élément de copie de données, mais qu'il intégrait des tests, notamment en ce qui concerne le contrôle d'accès. Ce qui assure qu'une donnée venant d'un espace mémoire réservé à un cœur avec un niveau de privilège particulier, ne puisse être modifiée sans un niveau d'accès au moins identique.

5.6 Conclusion

Au cours de ces travaux, nous avons été en mesure de perturber la microarchitecture du système. L'identification du modèle de faute n'est cependant pas complète et devra faire l'objet d'une étude plus poussée, pour cette raison, nous avons souhaité détailler nos expérimentations.

Nous avons pu mettre en lumière les difficultés d'évaluation du modèle de faute dans le contexte des SoC, ceux-ci se rapprochent dans leur microarchitecture des μc. Néanmoins l'un des éléments différenciant, la hiérarchie de la mémoire, est le point qui rend justement plus complexe l'évaluation du modèle de faute.

À la suite de nos expérimentations, la MMU et la SCU sont les deux éléments qui semblent affectées par l'injection de faute. Contrairement à ce qui a été réalisé sur μc, il nous a ici été impossible d'affecter simplement et directement le flot d'exécution.

Néanmoins, nous n'avons pas été capables de définir la manière précise dont étaient affectés la MMU et la SCU. Ce qui nous a également empêché de mettre en place une attaque logicielle avec comme vecteur d'introduction une injection de faute. La tentative d'exploitation a montré que le système pouvait prévenir une copie non autorisée.

Afin d'obtenir un modèle précis, il est donc souhaitable de continuer d'étudier le comportement de ces deux éléments face aux fautes. Plus précisément, la MMU qui est pour sa part davantage documentée constitue le premier axe sur lequel se concentrer.

L'accroissement de la complexité de la microarchitecture n'empêche pas la tenue des attaques par perturbation, mais rend leur exploitation et leur protection plus complexe.

Chapitre 6

Conclusion générale

6.1 Bilan

Nous avons su relever un certain nombre de défis qui se sont posés à nous au début de nos travaux, mais d'autres restent en suspend.

Nos contributions ont démontré qu'il était possible d'utiliser des attaques physiques contre la microarchitecture afin de mettre en œuvre tous types d'attaques quelles soient de type COA, DOA ou IOA.

Cependant, dans le cas des microarchitectures de type SoC, une tâche essentielle reste à effectuer, celle de définir de manière définitive le modèle de faute.

Dans ce cas, les attaques de type COA peuvent y être envisagées. La conclusion reste cependant que des comportements non prévisibles ont été mis en lumière, ces types de microarchitecture sont donc totalement sensibles à des analyses, même si des vulnérabilités n'ont pas été clairement identifiées.

Nos contributions ont également permis de mettre en lumière que les mesures de protections actuelles, sont insuffisantes pour protéger les systèmes et leurs microarchitectures, en particulier, les techniques logicielles assistées par le matériel comme la TZ. En effet, elles disposent d'une visibilité restreinte des couches d'abstraction et ne protègent en réalité que le logiciel contre des attaques logicielles.

Nous nous sommes concentrés sur des microarchitectures largement diffusées sur le marché actuel, et il s'est avéré qu'aucune protection physique spécifique empêche de reproduire les attaques que nous avons pu mener sur des produits disponibles dans le commerce. Seule la difficulté expérimentale, et des compétences suffisantes de la part de potentiels attaquants sont nécessaires.

Dans le cas des μc, des vulnérabilités ont pu être créées par le biais d'injection d'une faute. Ce médium, de création de vulnérabilité n'étant pas soluble par des mises à jour, il nous parait essentiel de proposer des moyens de protection efficaces.

Ne s'agissant pas de l'objet de nos travaux, les protections matérielles existantes qui ont été présentées dans le chapitre 2, devraient être complétées avec des solutions permettant de protéger efficacement la microarchitecture.

Du point de vue des SoC, des attaques en écoute ont pu être menées. Les différentes modifications, apparue entre la mise en place de ces expérimentations sur un μc et la même version sur SoC, n'ont pas eu un effet significatif sur la capacité à les reproduire.

6.2 Travaux futurs

Au travers de ces travaux, quatre pistes nous semblent de potentielles candidates à la continuation des travaux.

— Affiner la tenue des attaques qu'il s'agisse d'écoutes ou d'injections.

— Reproduire des attaques sur des vrais appareils du commerce, pour des systèmes IoT.

— Reproduire des attaques sur des vrais appareils du commerce, pour des systèmes SoC.

— Imaginer des solutions de protection contre ce type d'attaque en repensant la manière dont la microarchitecture fonctionne, ou en lui ajoutant des gardes-fous permettant de rendre inefficaces les attaques en écoute et inopérantes les perturbations.

6.2.1 Précision des résultats

La première orientation concerne la continuation des travaux d'évaluation du modèle de faute. Celui-ci a été identifié, mais n'est en l'état pas applicable à une attaque réelle comme nous avons pu le mener pour les travaux sur μc. Il nous parait donc évident que le finaliser pour le rendre reproductible et exploitable constitue l'axe majeur pour la continuation des travaux en cours.

Une autre possibilité est également celle de comparer notre implémentation «bare metal» avec une implémentation avec OS. En effet, dans notre cas de nombreux périphériques ne sont pas utilisés et d'autres n'ont pas de système d'auto-correction d'erreur. Un OS peut mettre en place des stratégies permettant de limiter les effets des injections de faute, que ce soit ou non volontaire. Par exemple, en limitant (ou multipliant) le nombre d'évictions du cache, ce qui rend obligatoire de multiplier les injections de faute, ou en modifiant la stratégie de lecture du snooper etc.

Ensuite, nous pouvons aborder l'affinage de la méthodologie d'attaque. En effet, nous sommes restés dans un environnement de laboratoire et nous avons réussi à obtenir des résultats qui prouvent les différentes vulnérabilités. Néanmoins, certains résultats semblent pouvoir être améliorés, et ils passent par une amélioration des techniques que nous avons pu employer.

La synchronisation avec la cible est un défi important à résoudre qui pourra permettre de simplifier la tenue d'attaques physiques.

De la même manière, les techniques d'injections peuvent être affinées en se concentrant sur le couplage avec les différents matériaux par exemple.

6.2.2 Focalisation sur des IoT du marché

La deuxième orientation concerne la continuation des travaux dans le domaine des IoT. Pour ces systèmes, nos travaux sont venus faire suite à ceux déjà présents dans la littérature, en particulier dans la thèse de Nicolas Moro [Mor14]. Par conséquent, le passage à des systèmes usuels nous semble être une orientation intéressante. En effet, le modèle de faute est connu et permet désormais de lancer des attaques logicielle diverses. Envisager une stratégie complète d'attaque sur un système connu, pour en prendre le contrôle est donc une des suites potentielles à nos travaux.

6.2.3 Focalisation sur des smartphones du marché

La troisième orientation est quant à elle à focaliser sur les systèmes utilisant des microarchitectures de type SoC. Permettant notamment d'étendre les réflexions sur l'utilité des coprocesseurs. En particulier, les téléphones mobiles qui disposent de SoC étant uniquement des cibles du processeur de bande de base qui lui est le maitre du système et dispose d'un accès supérieur ainsi qu'une mémoire séparée. De plus, ces

travaux supplémentaires permettront de valider en conditions réelles que des éléments non documentés mais offerts par les constructeurs via des drivers propriétaires, ne permettant pas de protéger certaines opérations.

6.2.4 Étude des possibilités de protection

Des méthodes de protections basées sur une modification de la microarchitecture peuvent être envisagées. Nous avons vu tout au long de nos travaux, qu'il s'agissait de la couche présentant les atouts et devant donc être protégée.

D'une part les protections présentées qui agissent sur la représentation matérielle sont des pistes à envisager. Ainsi, les techniques de blindage ou qui permettent de «masquer» la microarchitecture et de la protéger. Cependant, il s'agit de techniques de sécurité par l'obscurité. Celles-ci ont déjà été mises à l'épreuve dans la littérature dans d'autres domaines et ont démontrées qu'elles n'assuraient pas une protection suffisante.

D'autres part des techniques basées sur des modifications de la microarchitecture peuvent être explorées. Qu'il s'agisse d'y ajouter des circuits de protections d'une part ou de libérer les sources de ces microarchitectures pour permettre d'évaluer les modèles de faute ayant cours sur ces architectures et de permettre aux programmeurs de considérer les éventuelles failles lors de la conception de leurs programmes.

Ajout de circuits

Dans le premier cas, on maintient la confiance dans la microarchitecture en ajoutant des circuits de prévention, mais d'un autre côté on limite cette confiance en laissant la possibilité de prendre en compte les éventuelles failles.

Nous considérons donc, que l'ajout de coprocesseurs sécurisés est une des pistes potentielles. Leur certification assure qu'ils ont été testés et que dans les cas d'utilisation prévus, il est possible de maintenir la confiance en eux. Les Secure Elements, ou des cartes à puces nous semblent être des éléments à considérer.

En parallèle, des travaux en cours sur la modification de la microarchitecture par l'ajout de circuits spécifiques comme [LLBT18, dCDKC+16], basés sur l'ISR qui permet d'assurer la CFI, peuvent constituer des alternatives aux coprocesseurs sécurisés. En particulier, dans le cas où le modèle de faute est identifié et lié au contrôle de flot. Cependant, l'ajout de sous-circuits dans la microarchitecture entraine l'augmentation de la surface d'attaque, ceux-ci doivent donc faire l'objet d'une évaluation face aux attaques physiques.

Matériel libre

Enfin, les microarchitectures présentées dans le cas des matériels libres comme les cœurs développés selon l'ISA RISC-V [Wat16, PW17], permettent d'avoir accès à des informations de plus bas niveau. Ceux-ci permettent d'une part de tester différentes techniques d'injections (EBFI ou SiBFI par exemple). Et d'autre part, de mettre en place plus rapidement et de manière plus transparente des méthodes de protections, passant par la conception de coprocesseurs de manière simplifiée par exemple. Mettre en place ce type de travaux pour l'avenir nous semble donc une piste envisageable.

Bibliographie

[AA04] Starr Andersen and Vincent Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3 : Memory protection technologies, 2004.

[AAA+90] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, J-C Fabre, J-C Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation : A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2) :166–182, 1990.

[AAD+16] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat : Control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 743–754, New York, NY, USA, 2016. ACM.

[ABEL09] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1) :4, 2009.

[AEPSQ] Cédric Archambeau, Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Template attacks in principal subspaces. In *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 1–14. Springer.

[AFS97] William A Arbaugh, David J Farber, and Jonathan M Smith. A secure and reliable bootstrap architecture. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 65–71. IEEE, 1997.

[ALF02] Lorinc Antoni, Régis Leveugle, and M Feher. Using run-time reconfiguration for fault injection in hardware prototypes. In *Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings. 17th IEEE International Symposium on*, pages 245–253. IEEE, 2002.

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of computer system dependability. In *Workshop on Robot Dependability : Technological Challenge of Dependable Robots in Human Environments*, pages 1–16. Citeseer, 2001.

[And18] Android. System and kernel security. https://source.android.com/security/overview/kernel-security, 2010-2018.

[Arm72] D.B. Armstrong. A deductive method for simulating faults in logic circuits. *IEEE Transactions on Computers*, 21(undefined) :464–471, 1972.

[ARM09] ARM. Arm security technology : Building a secure system using trustzone technology. *ARM PRD29-GENC-009492C*, 2009.

[ARM14] ARM. Arm cortex-a53 mpcore processor technical reference manual. *ARM DDI 0500D ID021414*, 2014.

[ARM17] ARM. Arm architecture reference manual : Armv8, for armv8-a architecture profile. *ARM DDI 0487B.a (ID033117)*, 2017.

[ARM19] ARM. Arm trusted firmware. https://www.trustedfirmware.org/, 2019.

[ASA+15] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-At-The-End attacks : Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48 :44–57, February 2015.

[BBKN12] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices : Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11) :3056–3076, Nov 2012.

[BCLL18] Sebanjila Bukasa, Ludovic Claudepierre, Ronan Lashermes, and Jean-Louis Lanet. When fault injection collides with hardware complexity. In *11th International Symposium on Foundations & Practice of Security (FPS 2018)*, Montréal, Canada, Nov 2018.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 16–29. Springer, 2004.

[BDS03] Sandeep Bhatkar, Daniel C. DuVarney, and Ron Sekar. Address Obfuscation : An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Usenix Security*, volume 3, pages 105–120, 2003.

[Bet16] Elyse Betters. Apple pay : How it works, 2016. Accessed : 2017-02-14.

[BJFL11] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming : a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[BLB+17] Sebanjila Kevin Bukasa, Ronan Lashermes, Hélène Le Bouder, Jean-Louis Lanet, and Axel Legay. How TrustZone Could Be Bypassed : Side-Channel Attacks on a Modern System-on-Chip. In Gerhard P. Hancke and Ernesto Damiani, editors, *11th IFIP International Conference on Information Security Theory and Practice (WISTP)*, volume LNCS-10741 of *Information Security Theory and Practice*, pages 93–109, Heraklion, Greece, September 2017. Springer International Publishing. Part 3 : Trusted Execution.

[BLLL18] Sebanjila Kevin Bukasa, Ronan Lashermes, Jean-Louis Lanet, and Axel Legay. Let's shock our IoT's heart : ARMv7-M under (fault) attacks. In *13th International Conference on Availability, Reliability and Security (ARES 2018)*, pages 1–6, Hambourg, Germany, Aug 2018. ACM Press.

[BMS13] E. Blem, J. Menon, and K. Sankaralingam. Power struggles : Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2013.

[BN08] A. Bosio and G. D. Natale. Lifting : A flexible open-source fault simulator. In *2008 17th Asian Test Symposium*, pages 35–40, Nov 2008.

[BPT10] Alessandro Barenghi, Gerardo Pelosi, and Yannick Teglia. Improving first order differential power attacks through digital signal processing. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, SIN '10, pages 124–133, New York, NY, USA, 2010. ACM.

[BWK+87] SP Buchner, D Wilson, K Kang, D Gill, JA Mazer, WD Raburn, AB Campbell, and AR Knudson. Laser simulation of single event upsets. *IEEE Transactions on Nuclear Science*, 34(6) :1227–1233, 1987.

[CK] Benjamin Morin Chaouki Kasmi. État des lieux de la sécurité des communications cellulaires. *Cesar 2011*.

[Cla07] Christophe Clavier. An improved scare cryptanalysis against a secret a3/a8 gsm algorithm. *Information Systems Security*, pages 143–155, 2007.

[CMR+01] Pierluigi Civera, Luca Macchiarulo, Maurizio Rebaudengo, Matteo Sonza Reorda, and A Violante. Exploiting fpga for accelerating fault injection experiments. In *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, pages 9–13. IEEE, 2001.

[CMS+98] Joao Carreira, Henrique Madeira, João Gabriel Silva, et al. Xception : Software fault injection and monitoring in processor functional units. *Dependable Computing and Fault Tolerant Systems*, 10 :245–266, 1998.

[Cor17] John Corpuz. Mobile password managers, 2017. Accessed : 2017-02-14.

[CRR03] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 13–28. Springer, 2003.

[CW14] Nicholas Carlini and David Wagner. Rop is still dangerous : Breaking modern defenses. In *USENIX Security*, volume 14, 2014.

[dCDKC+16] Ruan de Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen de Bosschere, Bart Preneel, Bjorn de Sutter, and Ingrid Verbauwhede. SOFIA : Software and control flow integrity architecture. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1172–1177. IEEE, 2016.

[DDE+12] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI : A Framework to Mitigate Control-Flow Attacks on Smartphones. In *NDSS*, volume 2, page 27, 2012.

[ddr12] Jedec solid state technology association. In *DDR3 SDRAM Specification JESD79-3F*, 2012.

[DDSW10] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Return-oriented programming without returns on arm. Technical report, Technical Report HGI-TR-2010-002, Ruhr-University Bochum, 2010.

[DMM+13] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. Electromagnetic glitch on the aes round counter. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 17–31, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on bliss lattice-based signatures : Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1857–1874, New York, NY, USA, 2017. ACM.

[EGH+14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid : an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2) :5, 2014.

[FKK10] Denis Foo Kune and Yongdae Kim. Timing attacks on pin input devices. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 678–680. ACM, 2010.

[GBBF15] Benoit Gonzalvo, Eric Bourbao, Lilian Bossuet, and Majeric Fabien. Jtag combined attacks. In *Workshop on Secure Hardware and Security Evaluation (Co-located with CHES 2015)*, 2015.

[GBTP08] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 426–442. Springer, 2008.

[GPP+16] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. Ecdsa key extraction from mobile devices via nonintrusive physical side channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1626–1638, New York, NY, USA, 2016. ACM.

[GPPT15] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from pcs using a radio : Cheap electromagnetic attacks on windowed exponentiation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 207–228. Springer, 2015.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks : Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, pages 897–912, 2015.

[GVNG94] Daniel D Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. Specification and design of embedded systems. 1994.

[HD14] Roee Hay and Avi Dayan. Android keystore stack buffer overflow, 2014.

[HS67] Fred H Hardie and Robert J Suhocki. Design and use of fault simulation for saturn computer design. *IEEE Transactions on Electronic Computers*, (4) :412–429, 1967.

[HSK97] Takashi HARADA, Hideki SASAKI, and Yoshio KAMI. Investigation on radiated emission characteristics of multilayer printed circuit boards. *IEICE Trans. Commum., B*, 80(11) :1645–1651, nov 1997.

[HSR95] Seungjae Han, Kang G Shin, and Harold A Rosenberg. Doctor : An integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213. IEEE, 1995.

[Jel68] Frederick Jelinek. Buffer overflow in variable length coding of fixed rate sources. *IEEE Transactions on Information Theory*, 14(3) :490–501, 1968.

[JFGEM17] Damien Jauvart, Jacques JA Fournier, Louis Goubin, and Nadia El Mrabet. First practical side-channel attack to defeat point randomization in secure implementations of pairing-based cryptography. In *SECRYPT*, pages 104–115, 2017.

[JT12] Marc Joye and Michael Tunstall. *Fault analysis in cryptography*, volume 147. Springer, 2012.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them : An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[KDN14] Maha Kooli and Giorgio Di Natale. A survey on simulation-based fault injection tools for complex systems. In *Design & Technology of Integrated Systems In Nanoscale Era (DTIS), 2014 9th IEEE International Conference On*, pages 1–6. IEEE, 2014.

[KKA95] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. Ferrari : A flexible software-based fault and error injection system. *IEEE Transactions on computers*, 44(2) :248–260, 1995.

[KLD+94] Johan Karlsson, Peter Liden, Peter Dahlgren, Rolf Johansson, and Ulf Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE micro*, 14(1) :8–23, 1994.

[Koc96] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[KS05] François Koeune and François-Xavier Standaert. *A Tutorial on Physical Security and Side-Channel Attacks*, pages 78–108. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[LBBC+16] Hélène Le Bouder, Thierno Barry, Damien Couroussé, Jean-Louis Lanet, and Ronan Lashermes. A Template Attack Against VERIFY PIN Algorithms. In *SECRYPT 2016*, pages 231 – 238, Lsbonne, Portugal, July 2016.

[Lev00] Régis Leveugle. Fault injection in vhdl descriptions and emulation. In *Proceedings-IEEE-International-Symposium-on-Defect-and-Fault-Tolerance-in-VLSI-Systems*, pages 414–19. IEEE Comput. Soc, Los Alamitos, CA, USA, 2000.

[LFG13] Ronan Lashermes, Jacques Fournier, and Louis Goubin. Inverting the final exponentiation of tate pairings on ordinary elliptic curves using faults. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 365–382. Springer, 2013.

[LLBT18] Ronan Lashermes, Héiène Le Bouder, and Gaël Thomas. Hardware-Assisted Program Execution Integrity : HAPEI. In *NordSec 2018 - 23rd Nordic Conference on Secure IT Systems*, Oslo, Norway, November 2018.

[M+03] George Marsaglia et al. Xorshift rngs. *Journal of Statistical Software*, 8(14) :1–6, 2003.

[Man02] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *International Conference on Information Security and Cryptology*, pages 343–358. Springer, 2002.

[MC78] Premachandran R. Menon and Stephen G. Chappell. Deductive fault simulation with functional blocks. *IEEE Transactions on Computers*, 27(8) :689–695, 1978.

[MDH+14] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection : towards a fault model on a 32-bit microcontroller. *arXiv preprint arXiv :1402.6421*, 2014.

[MDR+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. *ARMageddon : Cache Attacks on Mobile Devices*. Proceedings of the 25th USENIX Security Symposium August 10–12, 2016 • Austin, TX. USENIX Association, Berkeley, CA, 2016. OCLC : 83294375.

[Mob16a] Samsung Mobile. White paper : An overview of samsung knox platform. *White paper*, 2016.

[Mob16b] Samsung Mobile. White paper : Samsung knox a security solution. *White paper*, 2016.

[MOP08] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks : Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[Mor14] Nicolas Moro. *Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués. (Security of assembly programs against fault attacks on embedded processors)*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2014.

[Ngu16] Lanh Nguyen. Samsung pay : How it works, 2016. Accessed : 2017-02-14.

[NS05] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[OHD+12] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. Accessory : Password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 9 :1–9 :6, New York, NY, USA, 2012. ACM.

[Osw02] Elisabeth Oswald. Enhancing simple power-analysis attacks on elliptic curve cryptosystems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 82–97. Springer, 2002.

[Pau99] Paul C. Kocher and Joshua Jaffe and Benjamin Jun. Differential Power Analysis. In *CRYPTO*, pages 388–397, 1999.

[Pes] Alexander Peslyak. Return-to-libc attack.

[PS06] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, volume 2, pages 215–230. sn, 2006.

[PSC07] Stefan Potyra, Volkmar Sieh, and M Dal Cin. Evaluating fault-tolerant system designs using faumachine. In *Proceedings of the 2007 workshop on Engineering fault tolerant systems*, page 9. ACM, 2007.

[PW17] David Patterson and Andrew Waterman. *The RISC-V Reader : An Open Architecture Atlas*. Strawberry Canyon, 1st edition, 2017.

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA) : Measures and counter-measures for smart cards. In *Smart Card Programming and Security*, pages 200–210. Springer, 2001.

[RBLC15] Lionel Riviere, Julien Bringer, Thanh-Ha Le, and Hervé Chabanne. A novel simulation approach for fault injection resistance evaluation on smart cards. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–8. IEEE, 2015.

[Riv15] Lionel Riviere. *Sécurité des implémentations logicielles face aux attaques par injection de faute sur systemes embarqués*. PhD thesis, Telecom Paris Tech, 2015.

[RK10] Kurt Rosenfeld and Ramesh Karri. Attacks and defenses for jtag. *Design & Test of Computers, IEEE*, 27 :36 – 47, 03 2010.

[SD15] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[SPS09] Matthias Sand, Stefan Potyra, and Volkmar Sieh. Deterministic high-speed simulation of complex systems including fault-injection. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 211–216. IEEE, 2009.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok : Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[SST07] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 203–217. IEEE, 2007.

[STB97] Volkmar Sieh, Oliver Tschache, and Frank Balbach. Verify : Evaluation of reliability using vhdl-models with embedded fault descriptions. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 32–36. IEEE, 1997.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In Claudio A. Ardagna and Jianying

Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 224–233, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[TOT+13] Sébastien Tiran, Sébastien Ordas, Yannick Teglia, Michel Agoyan, and Philippe Maurine. A frequency leakage model and its application to cpa and dpa. *IACR Cryptology ePrint Archive*, 2013 :278, 2013.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW : Exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, Vancouver, BC, 2017. USENIX Association.

[TSW16] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling pc on arm using fault injection. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop on*, pages 25–35. IEEE, 2016.

[UBW72] Ernst G Ulrich, T Baker, and LR Williams. Fault-test analysis techniques based on logic simulation. In *Proceedings of the 9th Design Automation Workshop*, pages 111–115. ACM, 1972.

[vdVFL+16] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer : Deterministic Rowhammer Attacks on Mobile Platforms. pages 1675–1689. ACM Press, 2016.

[VL18] Gopal Vishwakarma and Wonjun Lee. Exploiting jtag and its mitigation in iot : A survey. *Future Internet*, 10(12) :121, 2018.

[Wat16] Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. PhD thesis, UC Berkeley, 2016.

[Wei] Ralf-Philipp Weinmann. Baseband attacks : Remote exploitation of memory corruptions in cellular protocol stacks.

[YF14] Yuval Yarom and Katrina Falkner. Flush+ reload : A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, volume 1, pages 22–25, 2014.

[ZDT+14] Loic Zussa, Amine Dehbaoui, Karim Tobich, Jean-Max Dutertre, Philippe Maurine, Ludovic Guillaume-Sage, Jessy Clediere, and Assia Tria. Efficiency of a glitch detector against electromagnetic fault injection. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

Glossaire

μc Microcontroller.

AES Advanced Encryption Standard.
ALU Arithmetic-Logic Unit.
ANSSI Agence Nationale de la Sécurité des Systèmes d'Information.
ARM Advanced RISC Machine.
ASLR Address Space Layout Randomization.

BYOD Bring Your Own Device.

CESTI Centre d'Évaluation de la Sécurité des Technologies de l'Information.
CFG Control Flow Graph.
CFI Control Flow Integrity.
CISC Complex Instruction Set Computing.
COA Control-oriented attack.
COP Call Oriented Programming.
CPA Correlation Power Analysis.
CPI Code Pointer Integrity.
CPU Central Processing Unit.

DEP Data Execution Prevention.
DES Data Encryption Standard.
DFI Data Flow Integrity.
DOA Data-Oriented Attack.
DPA Differential Power Analysis.
DRAM Dynamic Random-Access Memory.
DSP digital signal processing.
DTA Dynamic Taint Analysis.
DVFS dynamic voltage & frequency scaling.

EBFI Emulation-based Fault Injection.
ECC Elliptic Curve Cryptography.

ECDSA Elliptic Curve Digital Signature Algorithm.
EMFI Electromagnetic Fault Injection.

FI Fault Injection.
FIST Fault Injection system for Study of Transient fault effects.
FP Floating-point.
FPGA Field-Programmable Gate Array.

GPIO General-Purpose Input/Output.
GPU Graphical Process Unit.

HBFI Hardware-Based Fault Injection.
HDL Hardware Description Language.

IFT Information Flow Tracking.
IOA Information-oriented attack.
IoT Internet of Thing.
ISA Instruction Set Architecture.
ISR Instruction Set Randomization.

JOP Jump-Oriented Programming.
JTAG Join Test Action Group.

KI Kernel Integrity.

LHS Laboratoire de Haute Sécurité.

MATE Man-At-The-End.
MDM Mobile Device Management.
MIPS Microprocessor without Interlocked Pipeline Stages.
MMU Memory Management Unit.

OS Operating System.

PCA Principal Component Analysis.
PCGD Plus Grand Commun Diviseur.
PIC Programmable Intelligent Computer.
PIN Personal Identify Number.
PLL phase lock loop.
PMIC power management integrated circuit.
PRNG Pseudo-Random Number Generator.

RAM Random Access Memory.
REE Rich Execution Environment.

RILC Return-into-libc.

RISC Reduced Instruction Set Computing.

ROP Return-Oriented Programming.

RoT Root–Of–Trust.

RPi2 Raspberry Pi 2.

RTL Register Transfer Level.

SCA Side-Channel Attack.

SCU snoop control unit.

SGX Software Guard eXtensions.

SiBFI Simulation-Based Fault Injection.

SoC System-on-Chip.

SoC System-on-Chip.

SOFIA Software and cOntrol Flow Integrity Architecture.

SoftI Software Integrity.

SPA Simple Power Analysis.

SWIFI Software implemented fault injection.

TA Trusted Application.

TEE Trusted Execution Environment.

TEE PP Trusted Execution Environment Protection Profile.

TIMA TrustZone based Integrity Management.

TPM Trusted Platform Module.

TZ TrustZone.

UART Universal Asynchronous receiver-transmitter.

USB Universal Serial Bus.

VHDL Very high speed integrated circuit Hardware Description Language.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Analyse de vulnérabilité des systèmes embarqués face aux attaques physiques.

Mot clés : Systèmes embarqués ; Sécurité ; Attaques physiques ; Attaques par canaux auxiliaires ; Émissions électromagnétiques ; Injections de fautes

Resumé :

Au cours de cette thèse, nous nous sommes concentrés sur la sécurité des appareils mobiles. Pour cela, nous avons exploré les attaques physiques par pertubation (injection de fautes) ainsi que par observation, toutes deux basées sur les émissions électromagnétiques.

Nous avons sélectionné deux types de cibles représentant deux catégories d'appareils mobiles. D'une part les microcontrôleurs qui équipent les appareils de type IoT. Et d'autre part les System-on-Chip (SoC) que l'on retrouve sur les smartphones. Nous nous sommes concentrés sur les puces conçue par ARM.

Au travers d'attaques physiques nous avons voulu montrer qu'il était possible d'affecter la microarchitecture sur laquelle repose tout le fonctionnement de ces systèmes. Toutes les protections pouvant être mises en place par la suite au niveau logiciel, sont basées sur la microarchitecture et deviennent donc inopérantes lorsque l'on s'attaque à celle-ci.

Pour les appareils de type IoT, nous avons mis en évidence la possibilité d'obtenir des informations ou un contrôle total de l'appareil à l'aide d'une injection de faute. Les injections de fautes sont dans ce cas les déclencheurs d'attaques logicielles et permettent d'outrepasser des protections logicielles.

Pour les appareils de type smartphone, nous avons dans un premier temps été capable d'extraire des informations contenue à l'intérieur d'un SoC, à l'aide d'une écoute électromagnétique et de la caractérisation du comportement de celui-ci. Dans un deuxième temps, nous avons pu montrer qu'en cas de faute des comportements aléatoire peuvent se produire, tout en caractérisant ces comportements. Démontrant ainsi que sur des systèmes plus complexes, il est tout de même possible d'avoir recours à des attaques physiques.

Enfin nous avons proposé des pistes d'améliorations en lien avec nos différentes constatations au cours de ces travaux.

Title : Vulnerability analysis of embedded systems against physical attacks.

Keywords : Embedded systems ; Security ; Physical Attacks ; Fault Injections ; Side-Channel Attacks ; Electromagnetic emissions

Abstract :

During this thesis, we focused on the security of mobile devices. To do this, we explored physical attacks by perturbation (fault injections) as well as by observation, both based on electromagnetic emissions.

We selected two types of targets representing two categories of mobile devices. On the one hand, the microcontrollers that equip IoT devices. And on the other hand the System-on-Chip (SoC) that can be found on smartphones. We focused on the chips designed by ARM.

Through physical attacks we wanted to show that it was possible to affect the microarchitecture on which the entire functioning of these systems is based. All the protections that can be implemented later at the software level are based on the microarchitecture and therefore become ineffective when it is attacked.

For IoT devices, we have highlighted the possibility of obtaining information or total control of the device by means of a fault injection. In this case, fault injections are used as software attack triggers. They also allow software protection to be bypassed.

For smartphone devices, we were initially able to extract information contained within a SoC, using electromagnetic listening and characterization of its behavior. In a second step, we were able to show that in the event of a fault, random behaviours can occur, we characterized and proposed explanations for these behaviours. Demonstrating and on systems more advanced than IoT, it is still possible to use physical attacks.

Finally, we proposed possible improvements in relation to our various findings during this work.