Eurisko : développement d'une carte électronique sécurisée

{LAM,LCR,LRP,LSC}@ANSSI
prenom.nom@ssi.gouv.fr

ANSSI

Résumé. Cet article propose un retour d'expérience sur le développement d'une carte électronique mettant en œuvre une chaîne de démarrage sécurisé intégrant un mécanisme d'authentification pré-boot.
Cette chaîne repose sur l'utilisation d'un composant sécurisé certifié à un niveau EAL5+ comme racine de confiance sur la carte et en tant qu'élément d'authentification extractible.
L'article présente également les problématiques associées à l'intégration de composants sur étagère dans une plateforme visant des objectifs de sécurité.
Outre l'architecture matérielle de la carte, l'ensemble des développements logiciels réalisés - pour le *System on Chip* (SoC) animant la plateforme et pour les composants sécurisés - sont détaillés.

1 Introduction

Le principal objectif de l'article est de discuter, sur la base d'un prototypage concret, de la **conception de plateformes sécurisées** utilisant des **composants sur étagère**.

1.1 Plan de l'article

Les attendus fonctionnels et sécurité de la plateforme cible sont détaillés dans cette première partie introductive.

Une seconde partie de l'article propose au lecteur une (re)mise à niveau rapide sur les deux types de composants principaux utilisés dans le projet : les SoC et les composants sécurisés. Celle-ci vise à simplifier la lecture des sections suivantes en décrivant leurs possibilités et limites pour les besoins de mise en œuvre d'une chaîne de démarrage sécurisé.

Sur ces bases, la troisième partie présente de manière progressive l'architecture matérielle retenue pour la carte afin de prendre en compte les objectifs et attendus de la plateforme ainsi que les capacités et contraintes des différentes briques utilisées.

La quatrième partie détaille les aspects logiciels, fournissant un retour d'expérience, couvrant à la fois :

– l'ensemble des développements réalisés, des méthodologies, des contraintes à s'imposer et les outils utilisés pour garantir leur qualité ;
– la cryptographie sous tous ses aspects.

Enfin, la cinquième et dernière partie présente les conclusions du projet en fournissant une analyse critique de la sécurité obtenue face à différents types d'attaquants, permettant ainsi d'évaluer les limites des architectures à base de composants sur étagère. Elle fournit également un retour d'expérience sur divers aspects du projet tout en présentant les futurs développements envisagés et des voies d'amélioration.

1.2 Contexte

Pluridisciplinarité à l'ANSSI Les laboratoires de l'ANSSI, au sein de la division scientifique et technique, couvrent un spectre très large des domaines de la sécurité : les architectures matérielles et logicielles (LAM), la cryptographie (LCR), le réseau et les protocoles (LRP), les composants carte à puce et l'embarqué (LSC), le sans fil (LSF) et la détection d'intrusion (LED).

Ces laboratoires sont amenés à suivre à la fois les évaluations (CSPN, CC, qualifications) et la conception de systèmes sécurisés effectués par les industriels. C'est notamment le cas de systèmes embarqués complexes comme des chiffreurs, des HSM, et plus généralement des architectures logicielles et matérielles de confiance.

Prototypage d'un système sécurisé Etant partis du constat qu'il est difficile de préconiser des bonnes pratiques ou de cultiver de l'expertise sans maîtriser les implications réelles de celles-ci, nous avons décidé d'expérimenter les concepts de sécurité sur un cas concret de plateforme ; celle-ci intégrant les deux caractéristiques suivantes, attendues de nombreux équipements rencontrés en pratique :

- les cartes électroniques principales sont animées par des processeurs ou SoC disponibles sur étagère ;
- ces équipements mettent en œuvre une logique de démarrage sécurisé.

L'expérience montre que l'intégration de ces caractéristiques lors du développement d'un système sécurisé n'est pas aisée. C'est pourquoi, dans le but de garder un contact avec ces contraintes de développement et les problématiques d'industrialisation de ce type de système, il nous a semblé pertinent d'initier en interne la conception d'un prototype qui se rapproche au mieux d'un cas « industriel » avec les contraintes que cela implique et le niveau de sécurité attendu d'un tel produit.

1.3 Objectifs

Ce projet a pour objectif le prototypage d'une carte électronique et des éléments logiciels associés, visant son intégration dans un équipement réseau (sonde, passerelle, chiffreur, HSM) de petite taille. L'expérience acquise lors de ce projet doit quant à elle permettre aux membres de l'équipe d'approfondir les différents aspects du projet (composants, électronique, logiciel, architecture, outils, etc.) afin de maintenir et de développer une expertise.

Fonctionnalités attendues et contraintes A haut niveau, la plateforme doit fournir les fonctionnalités suivantes :

- support d'un système cible basé sur une distribution Linux sécurisée reposant sur un noyau *grsecurity*[1] ;
- routage de trafic *linerate*[2] entre ses deux interfaces réseau Gigabit Ethernet (GbE).

Par ailleurs, les objectifs et contraintes supplémentaires suivants sont envisagés :

- utilisation de composants sur étagère ;
- base de code maîtrisée (bas-niveau / haut-niveau) ;
- logique de démarrage sécurisé et maîtrisé ;
- faible consommation (quelques Watts) ;
- encombrement réduit (7×7 cm).

Fonctionnalités de sécurité du projet D'un point de vue sécurité, les objectifs principaux sont de disposer :

- d'une logique de démarrage sécurisé, permettant de garantir via une maîtrise du logiciel et du matériel que le code mis en œuvre par la plateforme est intègre et authentique à chaque démarrage ;
- d'un mécanisme d'authentification pré-boot (typiquement via un élément matériel extractible, éventuellement couplé à un *pin*) ;
- d'un mécanisme de mise à jour sécurisé ;
- d'un mécanisme de rétention de secrets.

[1] `https://grsecurity.net`
[2] Au débit maximal supporté par une interface

2 SoC et composants sécurisés

Pour simplifier la lecture du reste du document, cette section effectue une remise à niveau rapide pour le lecteur peu familier avec les deux mondes mis en relation dans le projet, celui des SoC et celui des microcontrôleurs sécurisés du monde de la carte à puce.

Elle vise ainsi à présenter les capacités et les limites de chaque type de produit, notamment sur les aspects fonctionnels et sécurité.

2.1 Les SoC modernes

Un SoC[3] est un composant électronique regroupant au sein d'un même boîtier ou *package* un processeur et différents blocs matériels - notamment d'interfaces - permettant d'étendre ses fonctions. Il n'est ainsi pas rare de trouver des SoC possédant plusieurs interfaces SPI, I^2C, UART, PCIe, SATA, Ethernet, CAN, ISO, SDIO, etc.

La grande majorité des SoC disponibles sur le marché sont animés par un ou plusieurs cœurs ARM. Ils équipent, par exemple, les téléphones, les box internet voire télévisions, les routeurs, les points d'accès Wi-Fi, etc.

En première approche, le choix d'un SoC spécifique pour un projet donné est souvent dicté par la « coloration » du projet, i.e. par les principales fonctionnalités attendues pour l'objet final. Ainsi, certains vendeurs orientent plus leurs produits vers le monde de la téléphonie mobile, d'autres vers le multimédia, d'autres vers le réseau, d'autres vers le stockage.

Les aspects ayant trait à la sécurité doivent également être pris en compte au moment de la sélection.

Démarrage d'un SoC Les aspects sécurité spécifiques au SoC sélectionné pour le projet seront détaillés par la suite mais les éléments suivants permettent au lecteur de se familiariser avec le fonctionnement classique de ce type de composant, notamment leur séquence de *boot*. L'ensemble des SoC ARM disponibles sur le marché présentent en effet, pour leur grande majorité, une logique de démarrage comparable.

Ceux-ci disposent d'un microcode embarqué stocké en ROM interne. Ce code - nommé *BootROM* - s'exécute à chaque mise sous tension du SoC. Lors du relâchement du signal de *reset*, le SoC commence généralement par lire les niveaux logiques présents à ce moment sur ses entrées de configuration. Cette information lui permet de définir le ou les périphériques de démarrage à utiliser. Par exemple une mémoire flash adressable sur 24

[3] *System on Chip* ou en français système intégré

bits raccordée au SoC via le bus SPI, une mémoire de type NAND, une EEPROM raccordée via le bus I^2C, etc.

La BootROM charge ensuite, depuis le périphérique configuré, une image de démarrage dont le format est spécifique au vendeur du SoC concerné. En pratique, cette image contient deux éléments principaux :

1. une procédure de *training DDR* - code spécifique au SoC, à la RAM utilisée et aux pistes réalisant la liaison entre ces deux derniers - permettant l'initialisation du contrôleur DDR interne au SoC, pour l'utilisation ultérieure de la mémoire RAM externe ;
2. un *bootloader* (classiquement U-boot ou Barebox) qui va généralement permettre de charger le noyau, ses paramètres et un `initrd`[4] depuis une autre interface de stockage (disque, NAND, μSD, etc.), pour ensuite démarrer le système d'exploitation.

En fonction du SoC, les fonctionnalités de la BootROM et sa documentation varient. Certaines BootROM restent très basiques, ne fournissant aucun mécanisme de sécurité spécifique. D'autres, en revanche, proposent des mécanismes de sécurité. Par exemple, les BootROM des SoC i.MX53 et i.MX6 de Freescale[5] proposent un mécanisme de démarrage sécurisé dénommé HAB[6] permettant grâce à des fusibles (*fuses*) de configurer de manière irréversible le condensat d'une clé publique utilisée pour la vérification d'authenticité du code présenté au SoC lors de son démarrage. D'autres vendeurs, comme Marvell, intègrent également sur leurs SoC récents ce type de fonctionnalité.

En pratique, même si le SoC dispose d'un mécanisme de sécurisation du démarrage implémenté dans la BootROM, le code source de cette BootROM n'est pas disponible. Au mieux, une documentation et des outils permettant de signer (voire de chiffrer) des images ainsi que de modifier les fusibles sont fournis par le vendeur du SoC. Ces mécanismes n'ont, par ailleurs, pas subi de **schéma de validation** tiers reconnus et éprouvés comme peuvent en bénéficier des composants sécurisés (EAL, schémas du monde bancaire, etc.). La seule garantie de sécurité apportée est celle du constructeur qui vend le SoC. L'utilisateur paranoïaque en vient donc à se poser les questions suivantes :

- quelle confiance peut-on avoir dans l'implémentation des mécanismes cryptographiques et de la logique de vérification présentes dans la BootROM ?

[4] *INITial RamDisk*
[5] Racheté depuis par NXP
[6] *High Assurance Boot*

- comment le code de la BootROM a-t-il été développé pour garantir une absence de bug (par exemple, lors de *parsing*) ?
- la BootROM implémente-t-elle des mécanismes de *recovery* ? Comment ceux-ci interagissent-ils avec les mécanismes de sécurisation du boot ?

Ne disposant jamais de réponses à ces questions lors de l'intégration d'un SoC, il semble opportun de ne pas fonder la sécurité du démarrage de la plateforme uniquement sur ces mécanismes propriétaires non vérifiés. Dans une logique de défense en profondeur, ces mécanismes peuvent évidemment être utilisés en sus pour rendre plus robuste la chaîne de démarrage, mais la **racine de cette chaîne doit être éprouvée, vérifiable et vérifiée**.

Fonctionnalités additionnelles ayant trait à la sécurité Cette section a pour objectif de présenter certaines briques matérielles des SoC modernes qui concernent la sécurité. Les éléments décrits sont à la fois des technologies de sécurité mises en avant par les fabricants de SoC (TrustZone, HAB, etc.) ainsi que des points d'intérêt à prendre en compte lors de l'analyse du modèle de menace d'une plateforme car ces éléments peuvent être utilisés par un attaquant pour détourner la chaîne de démarrage ou exécuter du code non authentique.

- **OTP[7], *fuses*, etc.** : comme nous l'avons déjà expliqué dans les sections précédentes, un des éléments principaux de la sécurité d'une plateforme est l'intégrité de sa chaîne de démarrage. Divers moyens existent pour assurer cette intégrité, mais on retrouve en général deux briques matérielles sur lesquelles celles-ci se fondent :
  - l'utilisation d'une zone OTP permettant de stocker des clés publiques, des condensats de clés publiques, parfois des clés privées ou secrètes (la zone OTP n'est alors pas accessible en lecture mais est utilisable uniquement par un coprocesseur cryptographique utilisant les clés, c'est le cas des *key ladders* de STB) ;
  - l'utilisation de fusibles (ou *fuses*) permettant de changer, de manière irréversible, un état du SoC durant son cycle de vie (activer une clé publique, bloquer l'accès JTAG, passer d'un mode développeur à un mode production, etc.).

  Le HAB de Freescale, l'équivalent chez Marvell et Atmel, chez Xilinx ou chez Altera (pour leurs SoC/FPGA) utilisent ces concepts. Leurs implémentations différent en général (utilisation de condensat

[7] *One Time Programmable*

de clés publiques et signature RSA, utilisation de cryptographie symétrique avec HMAC, etc.) mais rendent le même service de *boot* authentique/intègre.

- **TrustZone** : la majorité des SoC étant basés sur des cœurs ARM, une partie d'entre eux supportent TrustZone[8] [4], une technologie ARM dont l'objectif annoncé est de permettre la cohabitation sur le processeur de deux environnements (mondes) d'exécution disjoints :
  - le *Secure World* (ou monde sécurisé) exécute une base de code de confiance ;
  - le *Non-Secure World* (ou monde riche) exécute classiquement le système utilisateur (notamment l'OS principal comme une distribution Linux ou Android).

  Ces deux modes de fonctionnement du SoC sont **orthogonaux** aux niveaux de privilèges classiques (utilisateur et privilégié) du CPU ARM. Il est donc possible d'exécuter en parallèle deux OS disjoints dans les deux mondes, avec l'équivalent d'appels systèmes permettant le passage du monde riche vers le monde sécurisé via un moniteur.
  En pratique, le niveau de support de TrustZone (e.g. granularité d'accès sur les différents périphériques, capacités du développeur final à utiliser le mécanisme, etc.) entre différents processeurs ARM est assez variable. Concernant l'implémentation du mécanisme au niveau du matériel, le développeur doit à la fois faire confiance à ARM et au vendeur du SoC mais également garantir la qualité des développements qu'il place en *Secure World* (diverses CVE d'élévation de privilèges du monde riche vers le monde sécurisé exploitent une mauvaise implémentation de l'OS qui s'exécute dans le monde sécurisé, permettant la prise de contrôle de smartphones fermés par exemple [5, 8]). Par ailleurs, TrustZone ne fournit pas directement une solution à la problématique de démarrage sécurisé puisque la chaîne de démarrage doit elle-même configurer les blocs matériels liés à cette technologie pour pouvoir l'exploiter.
  Autrement dit, TrustZone apparaît comme un mécanisme complémentaire intéressant pour prolonger l'intégrité d'une plateforme une fois démarrée, dans une logique de défense en profondeur. Malgré tout, cette possibilité est à mettre en face des investissements à réaliser pour sa mise en œuvre.

- **Interfaces de debug et versions « sécurisées »** : la plupart des SoC disposent d'interfaces de trace et de debug, classiquement ac-

[8] Notamment tous les processeurs de type Cortex-A

cessibles via un bus JTAG. Elles permettent un accès privilégié au processeur permettant un contrôle total sur celui-ci, parfois même lors de l'exécution de fonctionnalités protégées (e.g. mécanisme de boot sécurisé intégré à la BootROM).

Certains vendeurs ont ainsi étendu leurs interfaces de debug de manière à rajouter à celles-ci une logique d'authentification pour limiter leur accès uniquement aux développeurs ayant connaissance des éléments d'authentification configurés [7].

Nous en revenons malgré tout toujours à une confiance faite au fabriquant, notamment en ce qui concerne l'absence de *backdoor* sur cette interface JTAG. Nous pouvons citer à ce propos l'exemple des composants FPGA Microsemi/Actel ProASIC où une recherche exhaustive d'ordres JTAG a permis la compromission d'une plateforme censée être fermée avec une interface JTAG chiffrée et sécurisée [10].

Il en résulte, en pratique, qu'une bonne interface de debug est une interface de debug **désactivée** (généralement via un *fuse*) après réalisation des tests sur la chaîne de production et/ou alors **non routée** sur la version de production d'une carte.

- **Interfaces de recovery** : de nombreux SoC embarquent des mécanismes de *recovery*. Ceux-ci sont généralement intégrés au SoC par le vendeur pour des besoins de disponibilité. Ils permettent notamment de faire « revivre » une plateforme dont le bootloader fourni à la BootROM a été abîmé. Certaines plateformes permettent ainsi de réaliser une procédure de *recovery* via l'interface USB, certaines via l'interface UART ou un bus I^2C. Un exemple classique est l'utilisation d'une séquence d'octets (*magic values*) sur l'interface de *recovery* : la BootROM commence par vérifier la présence de cette séquence avant de lancer son démarrage classique si rien n'est détecté après un *timeout*.

  Sans prétendre en obtenir une preuve formelle, il est utile de faire en sorte lors du design d'une carte avec démarrage sécurisé qu'aucune de ces interfaces de *recovery* ne puisse être utilisée par un attaquant pour détourner le flot d'exécution vers du code non authentique, ou à défaut que ce détournement ne compromette en rien la sécurité de la plateforme (car celui-ci est détecté).

- **Sélection de l'interface de démarrage** : tout comme pour les interfaces de *recovery*, il est important de s'assurer que l'accès d'un attaquant à l'interface de démarrage soit limité ou à défaut détecté.

- **Formats des composants** : comme tous les composants, les SoC existent sous différents formats de boîtiers. Parmi les plus courants, on peut lister :
  - le *Quad Flat No-leads package* (QFN) : boîtier rectangulaire doté de broches de connexion sur ses 4 côtés ;

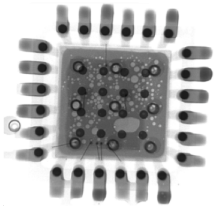


Fig. 1. Composant au format QFN (Rayons X). *Source : www.datest.com*

  - le *Ball Grid Array* (BGA) : boîtier rectangulaire doté de billes organisées sous forme de matrice et positionnées sur la face inférieure du composant. La taille des billes et la taille de la matrice dépendent du nombre de connexions externes du composant. Le pas entre billes est généralement de l'ordre du millimètre ;

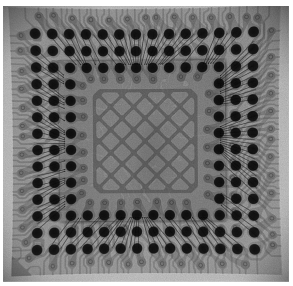


Fig. 2. Composant au format BGA (Rayons X). *Source : sec-xray.blogspot.com*

  - le *Chip Scale Package* (CSP) qui se présente sous la forme de boîtier BGA à l'échelle d'un *die*[9]. Le pas entre billes n'est que de quelques dixièmes de millimètre pour les composants au format WLCSP.

Fig. 3. Composant au format WLCSP. *Source : www.farnell.com*

[9] Tranche de silicium sur laquelle le composant est gravé

La plupart des SoC sont intégrés dans des boîtiers au format BGA. Ce type d'intégration est intéressant du point de vue sécurité car le positionnement des connexions sous le boîtier complexifie grandement l'accès aux signaux et limite donc fortement toute modification ou interaction simple avec le composant.

2.2 Les composants sécurisés

Les composants développés pour les besoins de l'écosystème carte à puce sont intéressants à plus d'un titre pour les besoins d'une plateforme visant des objectifs de sécurité. Plusieurs vendeurs développent et font ainsi certifier leurs composants (mécanismes de protection matériels, code embarqué, bloc cryptographique, générateurs d'aléa, etc.) face à des schémas exigeants (e.g. CC EAL5+).

Il est à noter que certains de ces composants servent également de base matérielle aux composants TPM[10] présents sur l'ensemble des plateformes PC. Ces composants exécutent alors un code dédié visant une interopérabilité avec les spécifications publiées par le TCG[11].

Le composant sélectionné pour la plateforme est un composant STMicroelectronics ayant passé une certification CC EAL5+ [2, 3], le ST33G1M2 [11]. Il s'agit d'un composant architecturé autour d'un cœur sécurisé ARM SC300 (SC pour *Secure Core*). La microarchitecture SC300 est une variante du ARMv7-M des microcontrôleurs Cortex-M3.

Il s'agit en première approche d'un microcontrôleur (ARM à 25Mhz, 30KB de RAM, 1.2MB de flash, 8 pins d'I/O) intégrant de nombreux mécanismes de protection (certains présents dans le SC300, d'autres ajoutés par ST). Ceux-ci sont évoqués ci-dessous :

- base ARM SecurCore® SC300™ RISC Core à 25MHz ;
- 1280KB de mémoire flash protégée en confidentialité et intégrité ;
- zone OTP ;
- *active shield* permettant de détecter les attaques par *probing* ou modification de signaux sur le circuit ;
- MPU[12] issue du SC300 ;
- surveillance des paramètres environnementaux ;
- protection contre les fautes (anti-DPA) ;
- TRNG AIS31 ;

[10] *Trusted Platform Module*
[11] *Trusted Computing Group*
[12] Memory Protection Unit

- accélérateurs matériels (multiplieurs et additionneurs pour cryptographie à clé publique, AES, 3DES).

D'un point de vue purement fonctionnel, malgré ses limitations en terme d'I/O (8 pins), le composant reste intéressant, à la fois comme composant de confiance pour le démarrage de la carte (dans son package BGA WLCSP), mais également comme composant externe d'authentification (dans son package ISO), comme nous le verrons par la suite :

- interface SPI (4 pins utilisables comme GPIO) ;
- interface ISO (2 pins aussi utilisables comme GPIO, donc bus I^2C) ;
- 1 GPIO supplémentaire (utilisée comme pin de *reset* du SoC) ;
- interface SWP.

Les interfaces précédentes sont présentées ci-dessous sur un package WLCSP dans lequel le composant est disponible.

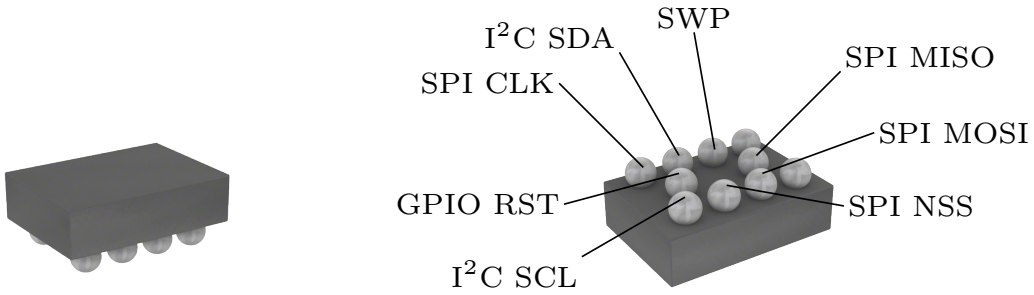


Fig. 4. Composant sécurisé au format WLCSP

3 Architecture matérielle de la carte

Avant de présenter progressivement les principaux composants constituant l'architecture matérielle de la carte, la figure 5 présente une vue d'ensemble de celle-ci.

3.1 Choix du SoC

Le SoC est le principal composant de la carte. Il supporte la majorité des fonctionnalités du système et c'est lui qui définit en grande partie la consommation, la taille et les performances de l'ensemble. Voici les différentes considérations qui ont permis de converger vers le SoC utilisé sur la plateforme, un SoC ARMv7-A Armada 370 de Marvell :

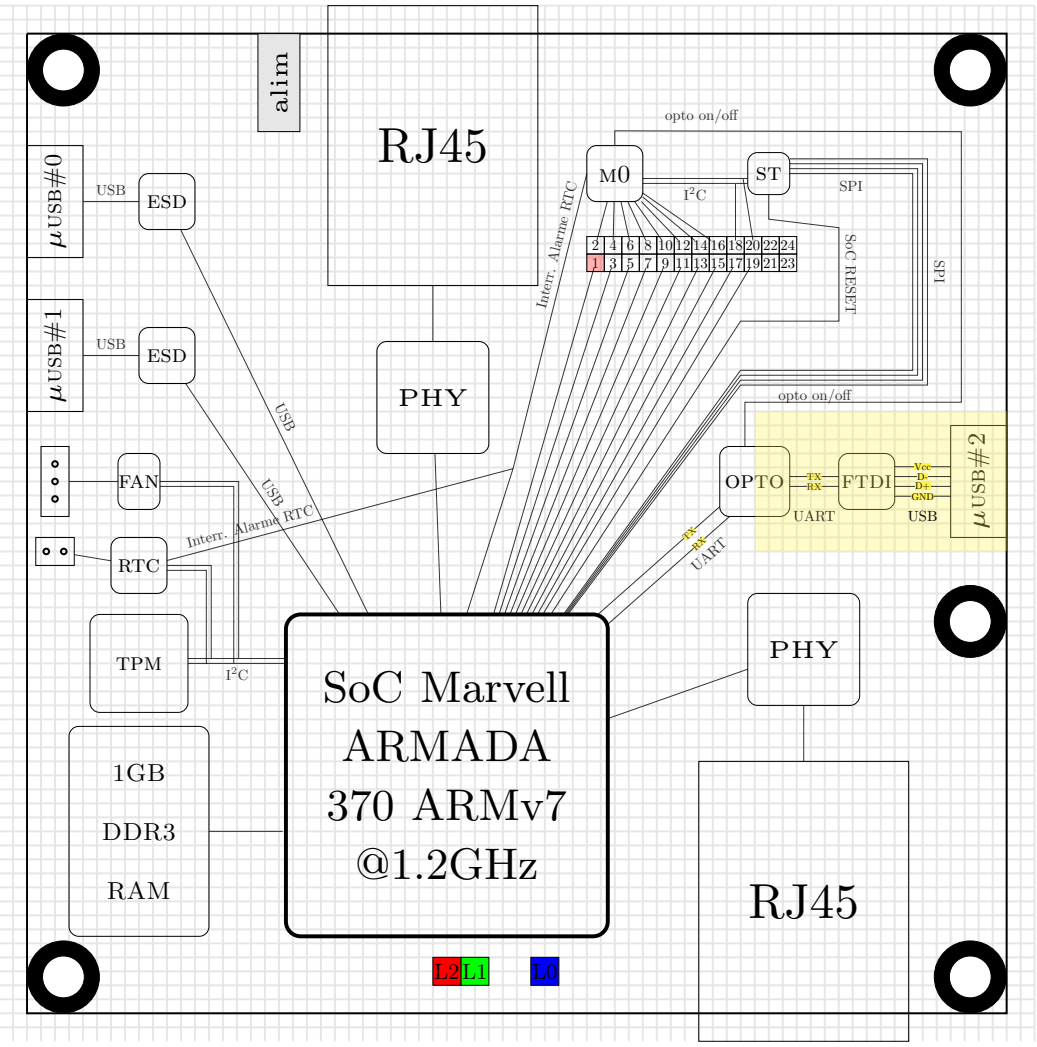


Fig. 5. Vue logique des principaux composants à intégrer sur le PCB et de leurs interconnexions

- la carte doit posséder une taille réduite (7×7 cm) et une consommation faible (quelques Watts) : ces contraintes nous ont orienté vers un SoC. Certains SoC ARM, par exemple, disposent d'une surface minimale, d'une consommation réduite (e.g. de 3W à 5W), d'un grand nombre d'interfaces et de fonctionnalités intéressantes. A l'inverse, même s'ils ont été envisagés, les processeurs x86 sont plus complexes à intégrer et malgré des TDP[13] moyens pouvant apparaître comme réduits, ils consomment encore trop. Pour d'autres raisons (e.g. un support Linux incomplet), des plateformes a priori intéressantes (certains SoC FPGA) ont été écartées ;

[13] *Thermal Design Power*

- l'objectif de faire tourner sur la carte une distribution Linux sécurisée impose évidemment de choisir un SoC disposant d'un bon support dans le noyau Linux. Il n'est, en effet, pas viable à long terme d'être limité à une version lourdement patchée présente dans le LSP[14] du vendeur de SoC. L'intégration de patchs complémentaires externes (e.g. *grsecurity*) devient alors impossible en pratique. Cette contrainte permet en quelque sorte de simplifier le choix en limitant l'ensemble des possibilités. Elle conforte également sur le type d'architecture : l'architecture ARM est probablement l'architecture de SoC la mieux supportée au sein du noyau Linux ; elle dispose également d'une large communauté de développeurs ;
- la carte doit permettre de router du trafic *linerate* entre deux interfaces réseau GbE : parmi l'ensemble des SoC ARM, peu disposent d'un support de deux interfaces GbE. Plusieurs disposent d'interfaces PCIe permettant de connecter des contrôleurs GbE mais l'impact sur la taille serait trop important. Au final, en étudiant les choix réalisés par de nombreux vendeurs de matériels réseau basse consommation (NAS, routeurs, points d'accès), les SoC Marvell Armada apparaissent comme un choix raisonnable. Leur support kernel est complet, les versions récentes disposent de deux voire trois interfaces GbE. Parmi les Armada, le 370, un mono-cœur ARMv7-A à 1.2Ghz, est un SoC mature parfaitement supporté et disposant de deux interfaces (MAC) GbE. Les différents Armada XP (bi ou quadricœur) présentent une consommation plus importante ;
- la carte doit disposer d'une logique de démarrage sécurisé. Comme évoqué précédemment, il ne semble pas opportun de faire reposer la sécurité uniquement sur les différents mécanismes propriétaires fournis par un SoC ARM, leur solidité n'ayant pas été évaluée. Au mieux, ceux-ci pourraient être utilisés pour de la défense en profondeur. Le SoC Armada 370 sélectionné à la date de développement ne disposait de toute manière pas d'une BootROM supportant un mécanisme de boot sécurisé[15]. Son grand frère l'Armada XP disposait quant à lui d'un tel mécanisme, mais aurait nécessité d'accepter une enveloppe thermique plus importante.

Le principal composant fonctionnel ayant été sélectionné, nous disposons maintenant de ses capacités en terme de démarrage. Ce SoC est

[14] *Linux Support Package*

[15] Même si une analyse du contenu du binaire de la BootROM laisse à penser que les fonctionnalités associées (signature RSA, etc.) y sont présentes

notamment capable de charger son code de démarrage depuis une flash externe sur son bus SPI.

3.2 Intégration du composant sécurisé

De manière à disposer d'une base de confiance, pour servir notamment un code intègre au SoC dès son démarrage, un composant sécurisé a été sélectionné. Disposant d'une interface SPI, ce dernier émule le comportement passif d'une mémoire flash servant au SoC son code de démarrage.

Notre choix s'est porté sur le ST33G1M2. Ce composant dispose d'une certification Critères Communs EAL5+, d'un nombre d'I/O acceptable (un bus SPI ainsi que quelques GPIO supplémentaires) et surtout d'un bon volume de flash sécurisée (1.2MB pour le code et les données). Ceci permet de stocker l'image de boot attendue par la BootROM du SoC. Celle-ci échantillonne au *reset* les niveaux de certaines de ces pins spécifiques pour en déduire sur laquelle de ses interfaces charger cette image ; dans notre cas une de ses interfaces SPI.

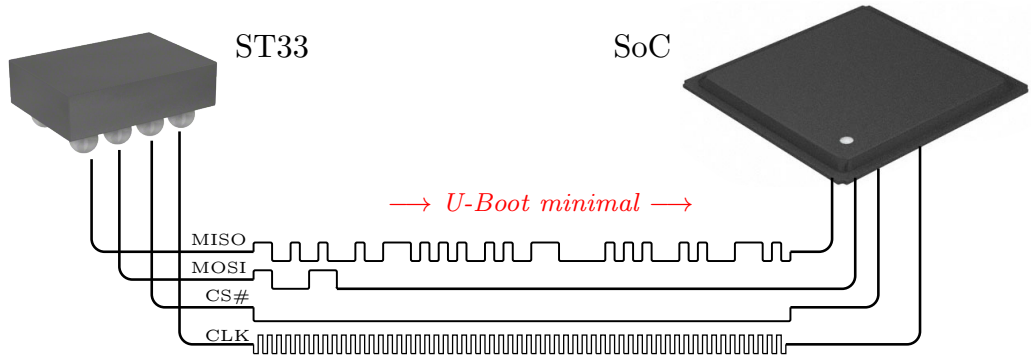


Fig. 6. Chargement du bootloader par le SoC depuis le composant sécurisé

Notons que la logique d'émulation de flash SPI choisie pour le démarrage sécurisé est parfaitement compatible avec le choix d'un autre SoC disposant de mécanismes de sécurité propriétaires (TrustZone, HAB, etc.) : il est donc possible de reprendre les concepts présentés dans la suite de l'article et de les composer avec des briques implémentées sur un autre SoC pour ajouter de la défense en profondeur à la plateforme. **La racine de confiance reste dans tous les cas le composant de sécurité évalué et maîtrisé**.

Disposant d'un composant **actif** de confiance sur la plateforme, il est dès lors possible d'implémenter une logique **d'authentification pré-boot**

faisant intervenir un second composant sécurisé extractible (intégré à un *token*). La mise en œuvre d'un tel mécanisme permet de réaliser le déploiement de code sur le SoC uniquement après une phase d'authentification cryptographique sur un élément matériel extractible.

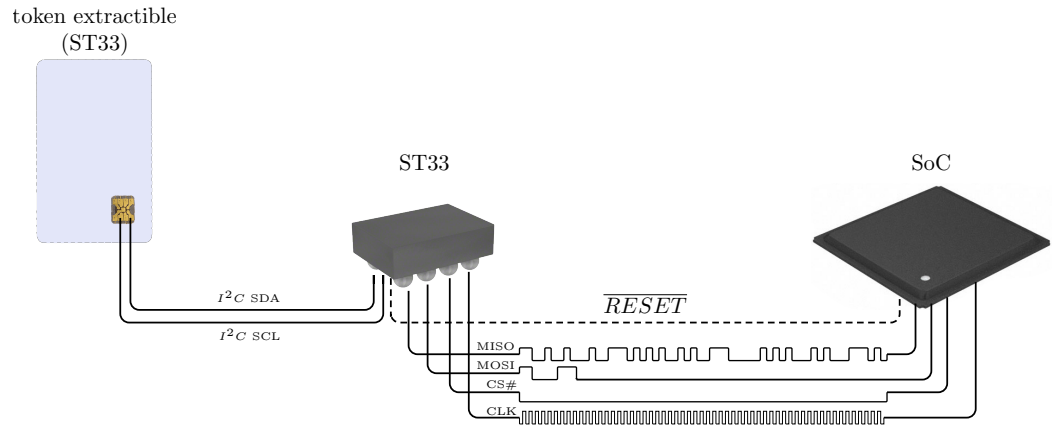


Fig. 7. Utilisation du bus I^2C pour l'ajout d'une logique d'authentification pré-boot contre un composant externe

Ces *tokens* peuvent supporter différents rôles, notamment le rôle d'utilisateur standard permettant le démarrage de la carte ou le rôle d'administrateur permettant de mettre à jour le code du ST33 ou le code de boot du SoC. Une ségrégation du matériel cryptographique confère une séparation plus efficace des rôles (plus de détails sont donnés dans la section décrivant l'architecture logicielle).

L'un des facteurs de forme idéal pour réaliser des *tokens* d'authentification est la carte à puce. Le composant ST33G1M2 étant disponible en version encartée, il nous a donc assez vite semblé naturel d'utiliser le même composant sécurisé ST33 que sur la carte principale pour concevoir ces *tokens*.

Le protocole de communication classiquement utilisé pour les cartes à puce est l'ISO-7816. Le protocole ISO nous a néanmoins semblé beaucoup trop lourd et complexe en termes de logique et de base de code à déployer (bien que le ST33 dispose d'un bloc matériel dédié, la quantité de code du pilote est disproportionnée en comparaison du service rendu dans notre cas). Nous n'avions par ailleurs pas de besoin spécifique d'interopérabilité avec d'autres plateformes ou du code déjà existant gérant de l'ISO.

Les deux pins de l'ISO-7816 pouvant être utilisées comme GPIO sur le composant, nous avons décidé de gérer notre authentification aux *tokens*

sur un bus I²C en mode *bit banging*[16], dont la logique reste très simple et portable.

Au delà des *tokens*, le bus I²C décrit précédemment - sur lequel le ST33 de la plateforme est maître - permet de gérer **différents esclaves**. Il est par exemple possible d'y connecter des EEPROM pour y lire des données, un écran LCD pour y afficher des informations de démarrage, etc.

En plus de l'authentification pré-boot, d'autres mesures ont été prises afin de limiter les capacités de l'attaquant :

- le SoC et le composant sécurisé sont tous deux présents sur le PCB sous forme de package BGA. Pour limiter l'accès aux signaux, les pistes du bus SPI entre les deux composants sont **routées sur des couches internes du PCB** comme représenté en figure 8 ;

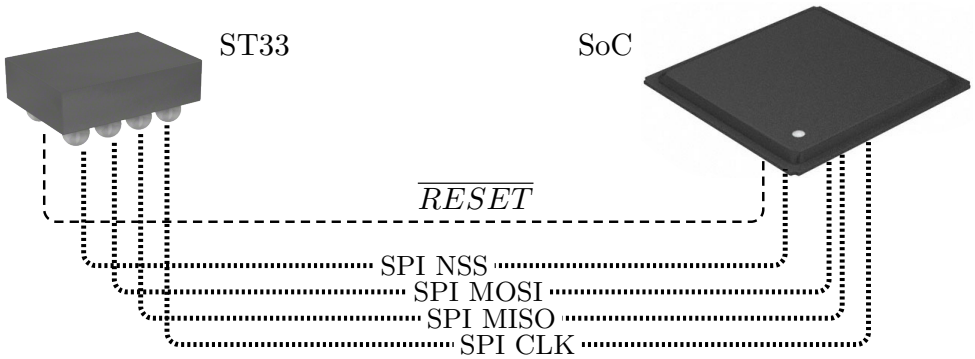


Fig. 8. Liaisons enterrées (SPI et *reset*) entre le SoC et le ST33

- l'invalidation du mode *recovery* supporté par le SoC (attente d'un motif sur ses interfaces UART au démarrage) est discutée plus avant dans le document, en sous-section 3.4 ;
- une des GPIO du composant sécurisé est directement connectée à la pin de *reset* du SoC via une piste routée sur une couche interne du PCB, permettant ainsi au composant de maintenir le SoC en état de *reset* tant qu'il le juge nécessaire ;
- du point de vue du séquencement des alimentations, le composant sécurisé est alimenté **avant le SoC**, lui permettant de contrôler totalement le démarrage de celui-ci.

[16] Deux GPIO sont ainsi pilotées pour recréer les signaux d'une interface I²C

3.3 Microcontrôleur compagnon du composant sécurisé (Cortex-M0)

Une des limitations du composant sécurisé est son nombre d'entrées/sorties ; 7 au total dont 4 utilisées par le bus SPI, 2 utilisées pour le bus I^{2}C pour la connexion au composant externe et une connectée à la pin de *reset* du SoC.

Etant connecté au composant externe, le bus I^{2}C est *de facto* facilement accessible à l'attaquant. Sous cette hypothèse, il peut néanmoins être utilisé pour étendre les capacités du composant **pour des fonctions non critiques du point de vue de la sécurité**.

De manière à étendre les capacités d'I/O du composant **pour des besoins non critiques**, un *iomuxer* I^{2}C vers GPIO[17] a été connecté à ce bus pour permettre au composant de contrôler 8 GPIO externes.

Plutôt que d'utiliser un composant spécifique aux capacités figées, un microcontrôleur (Cortex-M0) a été proposé par notre sous-traitant pour remplir cette fonction. Ce composant est matérialisé sur la figure 9 sous le nom **M0** à gauche du composant sécurisé.

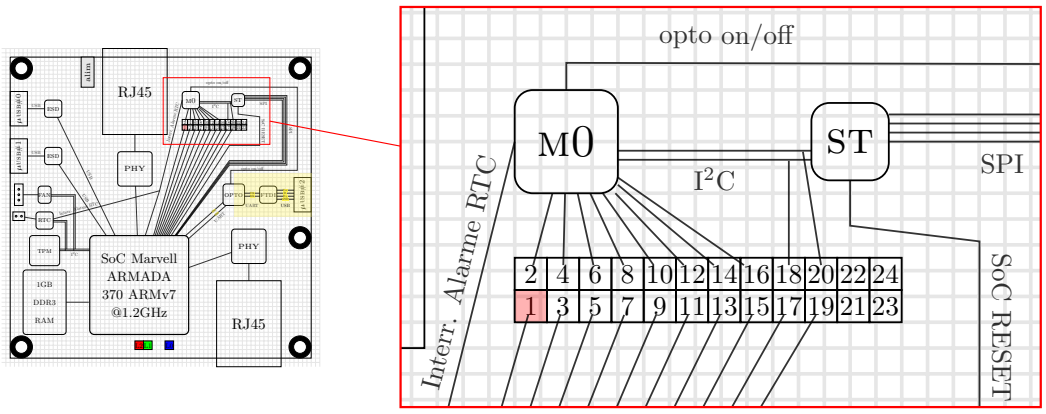


Fig. 9. Zoom sur le Cortex-M0 compagnon du composant sécurisé

Par ailleurs, ce microcontrôleur disposant d'une interface de communication avec le composant sécurisé présent sur la carte ainsi que de GPIO supplémentaires, il a également été utilisé pour contrôler les alimentations de la carte. Ainsi le composant sécurisé peut agir (pour des besoins fonctionnels) sur l'alimentation de la carte.

[17] Un composant pilotable via son interface I^{2}C permettant de contrôler des pins du composant agissant comme GPIO

Ce compagnon du composant sécurisé peut également prendre en charge quelques autres signaux externes pour des besoins fonctionnels de la carte (e.g. remontée de la pin d'interruption d'alarme du RTC pour le réveil de la carte hors tension, information fournie par le SoC pour le *reset*/*poweroff* de la plateforme, etc.).

3.4 Interface μSD pour le système

Une fois le bootloader du système chargé par le SoC depuis le composant sécurisé, celui-ci va lire le noyau Linux et son `initrd` depuis un périphérique de stockage de masse. L'ensemble des partitions du système sont également accessibles depuis ce périphérique.

Pour des contraintes d'encombrement, il a été décidé d'intégrer sur la face arrière du PCB un connecteur de carte μSD, routé vers le SoC. Une carte μSD est donc utilisée comme mécanisme de stockage de masse sur la plateforme.

Dans la continuité des efforts pour sécuriser le démarrage, il paraissait nécessaire d'assurer le chargement d'un noyau et d'un `initrd` confidentiels et intègres à chaque démarrage. Pour cela, le composant sécurisé fournit au SoC la clé (permettant d'effectuer l'opération de déchiffrement intègre nécessaire), au travers d'un canal sécurisé « au-dessus » du bus SPI (voir aussi la sous-section 4.2).

Comme on le voit, la confiance est propagée, de proche en proche, depuis la prise de contrôle du *boot* par le composant sécurisé jusqu'au chargement de la distribution Linux.

3.5 Interface UART

Comme indiqué précédemment, l'interface UART du SoC permettant l'accès à une console d'administration a été routée sur le PCB. Plutôt que de fournir un simple I/O header sur lequel un utilisateur aurait à brancher avec précaution un convertisseur USB/série 3.3V, il a été décidé d'intégrer directement sur le PCB un tel module USB/série FTDI et de placer en périphérie de la carte un connecteur μUSB. Un utilisateur a donc simplement à brancher un câble USB/μUSB à sa machine pour interagir avec la console du système.

L'interface du convertisseur USB série étant alimentée par le PC de l'utilisateur, une rupture électrique a été réalisée en intégrant un **optocoupleur** entre le convertisseur μUSB et le SoC. Cet optocoupleur est activable par le microcontrôleur compagnon du composant sécurisé, à la demande de celui-ci. Ceci fournit (en complément d'un suivi du timing

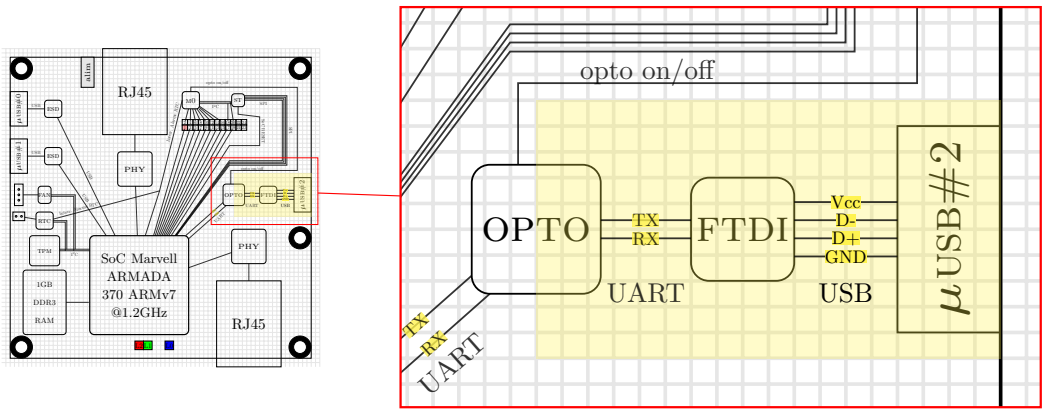


Fig. 10. Zoom sur l'optocouplage de l'UART

du dialogue du SoC avec le composant) une invalidation du mécanisme de *recovery* du SoC via son interface UART.

3.6 Autres éléments de l'architecture

Nous avons présenté dans les sous-sections précédentes les briques de l'architecture matérielle que nous estimons fortement liées à la sécurité de la plateforme (car sollicitées au boot et durant le fonctionnement de celle-ci).

Le PCB de la carte comporte bien évidemment d'autres éléments sur lesquels nous ne nous étendrons pas plus pour des raisons de place et de lisibilité de l'article : les deux interfaces réseau sont obtenues via deux PHY Gigabit connectées chacune à une interface MAC du SoC ; une RTC a été intégrée pour disposer d'une base de temps stable ainsi que le réveil de la carte hors tension ; un contrôleur de ventilateur permet la régulation en température en boîtier fermé ; deux interfaces USB du SoC sont exposées via des connecteurs μUSB ; et enfin un TPM (lui aussi à base de composant sécurisé ST33) est présent sur le bus I^2C du SoC. Ce dernier peut servir, en sus du composant de sécurité, à assurer l'intégrité du système Linux.

3.7 Résultat final

Le PCB final du premier *run* de carte est présenté en figure 11. On y retrouve à leur position définitive l'ensemble des composants présentés dans la vue logique de la figure 5 et détaillés dans les sous-sections précédentes.

Sur ce premier *run* de carte, le lecteur attentif notera la présence d'un I/O header 10 pins en lieu et place du composant sécurisé définitif

Fig. 11. Vue du PCB (crédit photo PCA)

au format WLCSP, pour permettre durant les phases de développement logiciel et d'intégration l'utilisation d'un émulateur du composant sécurisé.

4 Architecture logicielle de la carte

L'aspect matériel est un élément nécessaire mais non suffisant à la conception d'une plateforme sécurisée. L'architecture logicielle, et au delà de celle-ci les contraintes de développement imposées, doivent également faire l'objet d'une attention particulière afin de garantir le niveau de sécurité de la plateforme dans son ensemble. C'est pourquoi nous détaillons dans la présente section les garde-fous mis en œuvre pour pallier les diverses menaces envisagées.

Le projet étant composé de divers composants (ST33, SoC A370, Cortex-M0, etc.) qui ont nécessité de nombreux développements, nous avons décidé de présenter ces travaux au travers de cinq thématiques distinctes :

- **l'authentification pré-boot** ;
- **le boot sécurisé du SoC** ;
- **les fonctionnalités de mise à jour intègre et authentique de la plateforme** ;
- **la cryptographie** ;
- **le développement, l'analyse et la validation** ;

Ces thématiques sont décrites au travers de diverses sous-sections. Même si ces éléments sont complémentaires et s'articulent autour d'une architecture matérielle et logicielle **pensée dans son ensemble**, le découpage subjectif que nous en avons fait pourra paraître artificiel à certains lecteurs.

4.1 Authentification pré-boot

Dans l'objectif de bloquer un attaquant qui voudrait détourner la chaîne de démarrage et/ou voler des secrets de la plateforme, nous avons mis en œuvre un mécanisme **d'authentification pré-boot**. Avant de détailler le fonctionnement de ce mécanisme, nous présentons dans une première sous-section le principe de *token* d'authentification, et les bénéfices qu'il fournit, avec les divers rôles fonctionnels envisagés pour la plateforme.

Le *token* d'authentification pré-boot Le rôle du *token* d'authentification pré-boot est simple : un élément matériel externe à la plateforme est nécessaire au déverrouillage de celle-ci pour un fonctionnement nominal, et notamment au déploiement de tout code en dehors du composant plateforme. En terme fonctionnel, le scénario se résume aux actions suivantes :

- l'utilisateur insère son *token* d'authentification dans un emplacement dédié sur l'équipement ;
- les deux composants sécurisés (sur le *token* et la plateforme) réalisent un échange de secret authentifié mutuellement pour permettre la montée d'un canal sécurisé pour l'échange éventuel de données ;
- sur une montée de canal réussie, le composant plateforme continue l'exécution du code spécifique associé au rôle du *token*.

Le canal sécurisé mis en œuvre entre les deux composants garantit des propriétés de confidentialité, intégrité et anti-rejeu ; il permet ainsi de protéger les échanges entre composants sur le bus I^2C. Les clés utilisées sont dérivées d'un échange Diffie-Hellman sur courbe elliptique - garantissant une propriété de *perfect forward secrecy* (PFS) - couplé à un mécanisme de signature (également sur courbe elliptique).

Dans le projet, ce *token* prend la forme d'un composant ST33G1M2 au format **ISO encarté**, mais en utilisant le protocole I^2C au lieu et place du protocole ISO-7816 pour ses communications avec le ST33G1M2 de la plateforme.

Le choix du même composant permet d'obtenir des garanties de sécurité ainsi qu'une architecture logicielle et une base de code partagées communes.

Le choix du bus I^2C en remplacement du bus ISO-7816 rend (de notre point de vue) l'implémentation du protocole beaucoup plus simple ; ce qui facilite l'analyse et la maintenance de code.

Modes de démarrage et rôles des utilisateurs La ségrégation des rôles des utilisateurs qui interviennent lors du cycle de vie de la plateforme doit être assurée de manière physique (via au moins un facteur d'authentification) et cryptographique. Le système de *token* actif est idéal grâce à son composant ST33 certifié ainsi que l'utilisation d'une cryptographie à l'état de l'art. Nous identifions trois principaux rôles pour l'équipement :

1. le rôle utilisateur : il permet le démarrage de la plateforme en mode nominal ;
2. le rôle de mise à jour : il permet de mettre à jour à la fois des éléments de code du ST33 de la plateforme ainsi que le *stage 1* qui s'exécute sur le SoC. Nous y revenons plus en détail en 4.3 ;
3. le rôle d'administration : il permet de mettre à jour les clés cryptographiques qui servent la fonction de l'équipement (e.g. le biclef de montée de tunnels IPsec dans le cas d'un chiffreur par exemple).

4.2 Chaîne de démarrage sécurisé

Afin de garantir le contrôle, l'intégrité et l'authenticité de ce qui s'exécute sur le SoC à tout moment sur la plateforme, nous avons intégré une chaîne de démarrage sécurisé. Pour cela, nous utilisons le concept classique de propagation de confiance entre *stages* (i.e. étages) séquentiels de démarrage. La racine de confiance, premier maillon exécuté dans cette chaîne, est exécutée sur le composant sécurisé ST33 qui est intégré sur la carte électronique principale. Cela nous permet de profiter des divers mécanismes de protection intégrés au niveau matériel sur le ST33 et par conséquent d'empêcher (tout du moins limiter) les actions réalisables par un attaquant.

Présentation de la stratégie de démarrage Elle se décompose en quatre étapes macroscopiques, après authentification réussie :

1. le code exécuté sur le ST33 passe en mode émulation de flash SPI 24-bit et fournit le *stage 1* à la BootROM du SoC : le *challenge* à ce niveau a été de pouvoir suivre la fréquence SPI (8Mhz) imposée par la BootROM du SoC qui joue le rôle de *master* sur le bus (facilité par le bloc matériel SPI du ST33, le CPU n'étant cadencé qu'à 25Mhz) ;

2. le *stage 1* s'exécute uniquement en SRAM du SoC. Il initialise la DDR, active les blocs matériels dont il a besoin (SPI, SDIO et éventuellement UART). Il établit un canal mutuellement authentifié avec le ST33 de la carte principale sur le bus SPI et récupère la **clé de déverrouillage** - au sens déchiffrement intègre - de la plateforme. Cette clé est **unique par plateforme** ;
3. il utilise cette clé (qui ne quitte pas la SRAM) pour déchiffrer en RAM le *stage 2* qui peut correspondre directement à un `initramfs` et un noyau Linux ;
4. le système d'exploitation démarre (en utilisant éventuellement le TPM), et active les services nécessaires (montée de tunnels IPsec par exemple). Les services ne manipulent jamais directement de clés maîtres : le ST33 est utilisé par le SoC comme coprocesseur de sécurité permettant de dériver des secrets, générer de l'aléa, etc.

Chacune de ces étapes de démarrage peut *a priori* être analysée et détournée par un attaquant via des vecteurs logiciels, matériels, ou combinés. Nous discutons le détail de ces vecteurs ainsi que des contre-mesures dans les deux sous-sections suivantes.

Du côté du ST33 Le ST33 est au cœur de la plateforme. Ce composant sécurisé, bien que certifié CC EAL5+, doit donc être bien configuré et programmé afin que l'attaquant ne puisse pas exploiter une vulnérabilité et casser le modèle de sécurité.

ST33 *Démarrage sécurisé*

La configuration et l'utilisation des blocs matériels du ST33 dédiés à la sécurité sont nécessaires (*jitter*, contre-mesures DPA, etc.). Cette opération suppose de s'approprier les *datasheets* du composant, et de comprendre les implications des divers modes accessibles.

L'assurance d'un démarrage sécurisé et intègre des éléments **internes au ST33** est toute aussi indispensable. Pour cela, nous utilisons la même logique de chaîne de confiance que celle présentée en 4.2. Lors du démarrage du ST33, chacun des niveaux s'assure que le code du niveau suivant est intègre avant de l'exécuter. L'authenticité est assurée via la vérification de signature des mises à jour du code interne. Le détail des mécanismes de mise à jour est présenté dans la section dédiée 4.3.

ST33 *Utilisation de la* MPU

La mise en place d'une architecture logicielle a pour objectif de limiter

les effets d'une attaque par détournement de flot d'exécution (*buffer overflows*)[18], durant les différentes parties de l'automate d'état du ST33, par exemple lors du démarrage, de la mise à jour ou encore lors des échanges sur les bus de communication. Notre architecture logicielle repose principalement sur l'utilisation de la MPU[19] du ST33 pour :

- **cloisonner** les diverses briques fonctionnelles et les données au sein du composant sécurisé ;
- assurer le principe de **moindre privilège** : presque tout le code s'exécute en mode utilisateur (le mode superviseur est limité à très peu de code, localisé à un endroit unique non réinscriptible) ;
- **assurer du `W⊕X`** ;
- rendre les attaques **non persistantes** en protégeant le code et les données critiques contre les écritures en mémoire flash.

Il n'y a par exemple aucune raison qu'un attaquant qui prendrait le contrôle du driver SPI par injection d'une commande mal parsée puisse modifier le code et les données d'un autre composant logiciel, ou d'un *stage* qui s'est précédemment exécuté.

La MPU du ST33 n'a pas la granularité que peut apporter une MMU[20] de CPU plus évolué (il n'y a pas de pagination, mais des régions configurables, à rapprocher d'une segmentation mémoire). Les trois permissions `RWX`[21] sont néanmoins disponibles et utilisables. Il est également possible d'interdire l'accès à une région depuis un mode utilisateur. Nous pourrions donc assurer une protection mémoire telle qu'on l'attendrait d'une MMU. Toutefois, le nombre de régions configurables est limité et impose de réfléchir au meilleur compromis pour atteindre un niveau de protection optimal.

ST33 ***Protection du code à la compilation***

En plus de ce que peut apporter le matériel comme sécurité avec la MPU, nous ajoutons des mesures logicielles pour la protection du code :

- contraintes de développement, comme par exemple ne pas utiliser d'allocation dynamique ;
- *toolchain* de compilation personnalisée apportant des garanties comme des canaries ou du code `XO` uniquement exécutable et non lisible ;

[18] Contrairement à la logique de démarrage qui est entièrement développée et validée, cette partie est un chantier encore **en cours de développement**.
[19] Memory Protection Unit
[20] Memory Management Unit
[21] `Read`, `Write`, `eXecute`

– analyse statique de code décrite en section 4.4.

Ces éléments permettent notamment de limiter les attaques par ROP[22] qui échappent au W⊕X.

ST33 Watchdog et heartbeat

Nous pensons intégrer dans le ST33 un **watchdog** permettant de s'assurer lors du boot qu'un attaquant n'a pas pris le contrôle du SoC en lieu et place du *stage 1*. La phase de boot via SPI (*stage1*) utilise en effet un timing parfaitement déterministe. Cette opération est possible notamment du fait d'un code synchrone sans interruption côté ST33 et SoC. Lorsqu'un attaquant essaie de prendre la main sur le SoC par injection de code, au travers des interfaces de *recovery* (UART), des interfaces de debug (JTAG), ou de l'injection sur le bus SPI, il perturbe ce timing et devient **détectable**. Dans ce cas le ST33 qui contrôle le signal de *r*eset du SoC le redémarre, ce qui rend l'attaque caduque.

Une fois le *stage 1* poussé en SRAM sur le SoC, l'attaquant ne peut plus utiliser l'interface de *recovery* ou l'injection sur le bus SPI car un canal authentifié est monté. Toutefois, dans un souci de défense en profondeur, le *stage 1* peut implémenter une logique de *heartbeat* analogue sur le bus SPI. Ainsi, l'attaque et la manipulation de la mémoire par d'autres bus tels que le JTAG sont fortement contraintes et surtout détectables. Le SoC est en effet dans un mode *halt* lorsque l'attaquant interagit avec lui.

Bien que ce type de logique de *heartbeat* soit plus complexe à intégrer au sein d'un système comme Linux (problématique de respect strict de timings) il est possible de l'implémenter pour détecter toute modification dans le chemin d'exécution lorsque le système passe en mode de fonctionnement nominal (après le *stage 1*).

Notons que toutes ces contre-mesures logicielles s'ajoutent aux contre-mesures matérielles déjà décrites en section 3 : enterrement des pistes SPI, pins JTAG non câblées et difficilement accessibles (BGA), optocouplage sur UART relâché après l'exécution de la BootROM.

Du côté du SoC A370

A370 Cold boot attacks, espionnage du bus DDR

Le SoC reçoit via le bus SPI un *stage 1* que la BootROM place en SRAM (car la RAM DDR n'est pas encore initialisée). L'exécution en

[22] *Return Oriented Programming*

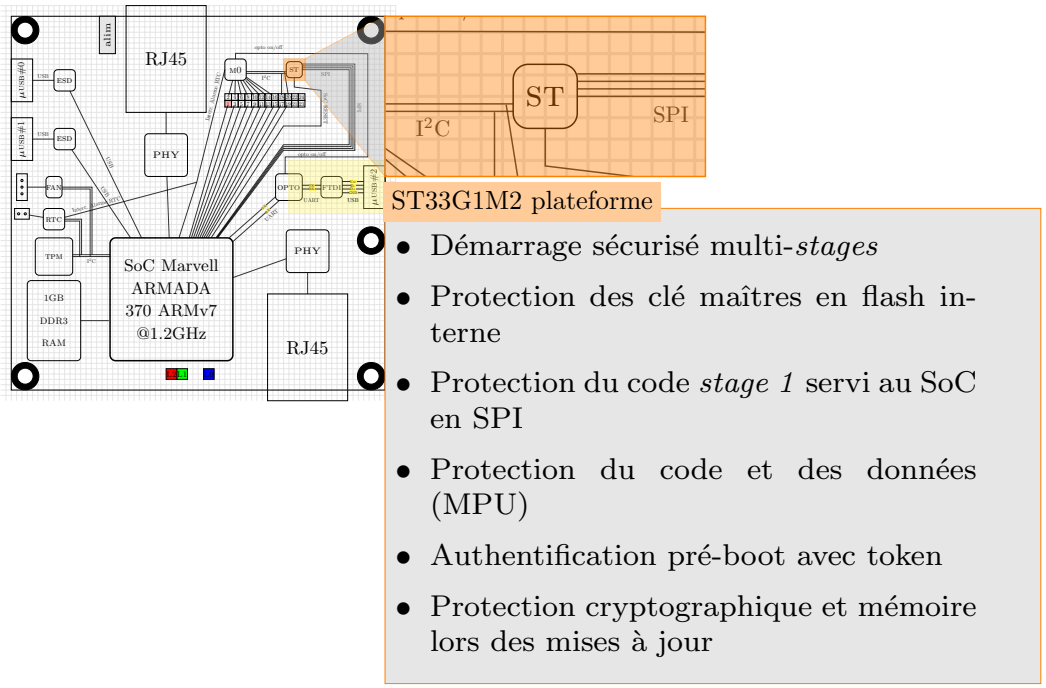


Fig. 12. Résumé des éléments de sécurité du côté du ST33G1M2 plateforme

SRAM (qui est dans le *package* du SoC) permet de se prémunir d'attaques de type *cold boot* [6,9] qui profitent de la rémanence de données après un *reset*, par exemple pour récupérer des clés cryptographiques.

La SRAM est en effet réutilisée par la BootROM au redémarrage - notamment pour sa pile et son tas - et les données précédentes écrasées, contrairement aux données en RAM.

Par ailleurs, l'exécution du *stage 1* en SRAM empêche la récupération d'éléments sensibles via un espionnage du bus entre le SoC et la DDR [1].

A370 Protection du bus SPI

L'une des premières actions réalisées par le *stage 1* exécuté sur le SoC est de monter un canal sécurisé avec le composant plateforme permettant le transfert pour exécution du *stage* suivant au SoC. La mise en œuvre de ce canal impose à un attaquant de se rendre actif sur le bus pour accéder aux échanges, et donc au code du *stage* suivant.

A370 Protection du code, MMU

Afin de protéger le *stage 1* contre les attaques de type *buffer overflows*, les mêmes concepts de cloisonnement mémoire et d'analyse et vérification de

code que sur ST33 sont appliqués (une MMU est évidemment disponible et utilisable sur A370).

Le *stage 1* est protégé en intégrité grâce au ST33. Une fois Linux démarré, l'exploitation d'une vulnérabilité est empêchée/détectée/cloisonnée via l'utilisation de moyens de défense classiques (PaX/grsec, MAC[23], etc.).

A370 Autres développements

Enfin, en plus de tout ce qui concerne le démarrage sécurisé, de nombreux développements pour le système d'exploitation ont été nécessaires, dont :

- le développement d'une DTS[24] dédiée pour le noyau Linux/grsec cible ;
- le développement des modules nécessaires au support d'éléments matériels de la plateforme pour le noyau Linux (RTC, contrôleur de ventilateur, etc.) ;
- l'intégration d'une distribution sécurisée (e.g. CLIP [12]) pour ARM.

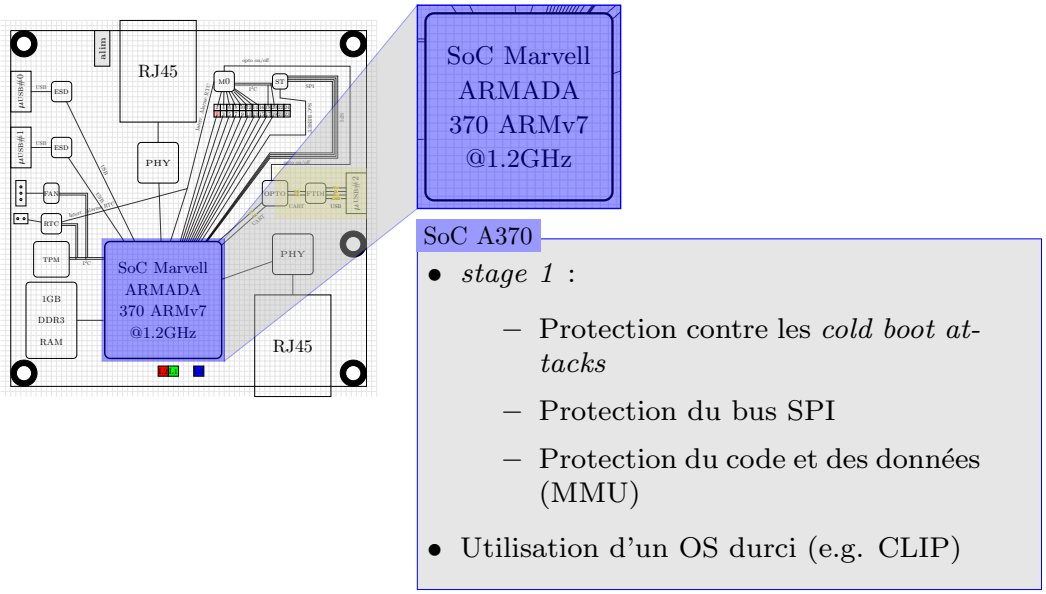


Fig. 13. Résumé des éléments de sécurité du côté du SoC A370

Du côté du Cortex-M0 Le Cortex-M0 (microcontrôleur compagnon du ST33 présenté en 3.3) est connecté au bus I^2C du composant sécurisé.

[23] *Mandatory Access Control*
[24] *Device Tree Source*

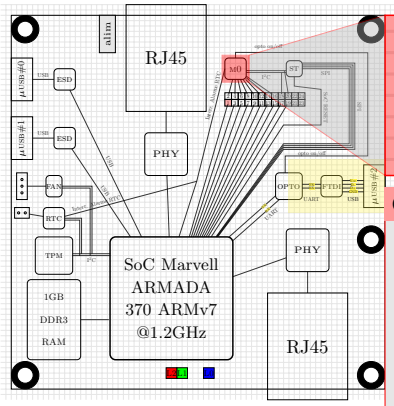
M0 Rôles et sécurité

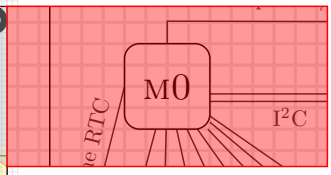
Le Cortex-M0 a principalement comme rôles :

- de gérer les étages d'alimentation lors du démarrage, l'automate de gestion étant commandé par le ST33 via des ordres I^2C ;
- de faire office d'I/O *expander* pour le composant sécurisé. Les I/O supplémentaires sont aussi commandées via des ordres sur le bus I^2C.

Le Cortex-M0 ne disposant pas de mécanismes de protection évalués, il a été considéré comme compromis. Les GPIO obtenues ne visent donc à servir que des besoins fonctionnels non liés à la sécurité. Nous avons également considéré les contre-mesures matérielles et logicielles suivantes :

- le Cortex-M0 gère le rail d'alimentation principal de la carte, mais la configuration des autres rails fait en sorte que le SoC A370 ne puisse être alimenté avant le ST33. Nous avons donc l'assurance **matérielle** que le ST33 est toujours démarré avant le SoC ;
- le Cortex-M0 est en écoute sur le bus I^2C : l'utilisation de canaux protégés avec authentification mutuelle entre *tokens* d'authentification pré-boot et composant plateforme permet d'empêcher le microcontrôleur d'accéder à leur contenu.





Cortex-M0

- Gère les étages d'alimentation de la carte
- Ne contient aucun élément critique pour la sécurité :
  - Le ST33 est alimenté avant
  - Le *reset* du SoC est directement géré par le ST33

Fig. 14. Résumé du rôle du Cortex-M0

4.3 La gestion des mises à jour logicielles

Les mises à jour logicielles peuvent être divisées en trois types :

- les mises à jour du code qui s'exécute sur le ST33. Ce code est dans la flash interne du ST33. Rappelons que pour des raisons de cloisonnement et de défense en profondeur, le ST33 implémente un démarrage sécurisé en plusieurs *stages* ;
- les mises à jour du *stage 1* : ce code s'exécute sur le SoC Marvell mais est stocké en flash du ST33 (envoyé sur le bus SPI à la BootROM) ;
- les mises à jour du reste du code qui s'exécute sur le SoC A370, à savoir le système d'exploitation stocké sur la carte SD.

Les deux premiers types de mises à jour sont gérés par un *token* avec authentification pré-boot ayant un rôle dédié. Pour réaliser ces opérations, plusieurs implémentations sont possibles, nous avons choisi de gérer les mises à jour d'une manière assez simple. Nous ne rentrons pas ici dans les détails pour des raisons de concision, mais nous en donnons les grands principes ci-dessous.

- Afin de cloisonner les divers *stages* de démarrage du ST33 nous avons restreint le nombre d'éléments pouvant être mis à jour par chacun des composants de la chaîne de démarrage du ST33. Cette restriction est généralement limitée aux étages suivants le *stage* en cours d'exécution. Par exemple, le *stage* qui envoie le code du *stage 1* SoC au travers du bus SPI est autorisé à mettre à jour uniquement le *stage 1* en flash interne du ST33.
- Le *token* de mise à jour peut contenir les mises à jour de plusieurs éléments différents. Chaque *stage* du ST33 de la plateforme vérifie donc sur le bus I^2C (dans le canal authentifié) s'il y a une mise à jour disponible pour les éléments qu'il a le droit de mettre à jour (la version de l'élément à mettre à jour fait foi).
- Une logique de FLIP/FLOP a été mise en place afin d'éviter de *bricker* le ST33.
- Le *token* de mise à jour contient dans sa flash le code de la mise à jour à envoyer sur le bus I^2C au travers d'un canal confidentiel intègre (nous profitons du fait que nous utilisons le même composant ST33G1M2 comme *token*, avec une taille de flash conséquente).
- Nous implémentons grâce à la MPU un automate à états très strict : lorsque le *token* de mise à jour s'est authentifié, le ST33 de la plateforme passe dans un mode où seules des mises à jour sont possibles (avec les droits `RWX` adaptés sur les régions associées). Il en va de même

quand un *token* utilisateur s'authentifie : la flash devient par exemple intégralement non inscriptible dans le sous-automate à états du mode utilisateur.
– Le composant ST33G1M2 disposant d'une quantité de RAM limitée, il n'est pas possible de vérifier la signature d'un élément à mettre à jour avant de le mettre en flash. Nous utilisons donc un système de **flags d'état** permettant de gérer une vérification en flash[25].

Enfin, le dernier type de mise à jour (mise à jour du système d'exploitation chiffré sur la carte SD) peut être géré directement depuis un démon exécuté par l'OS qui communique avec le ST33 de la plateforme sur le bus SPI, lui-même communiquant avec le *token* utilisateur et de mise à jour sur l'I^2C afin d'assurer le transchiffrement du *stage 2* (Linux) avec la clé de plateforme et le stocker sur la carte SD. La mise à jour de l'OS pouvant être de taille conséquente, il est prévu d'utiliser un support externe de type stockage de masse (chiffré avec une clé de transport connue du *token* de mise à jour), à mettre sur le port USB par exemple.

4.4 Développement, analyse de code et validation

Cette section aborde les contraintes de développement que nous nous sommes fixées afin d'assurer au maximum la production d'un code aisément analysable, en limitant les potentiels problèmes d'asynchronisme et autres *race conditions*. Ces contraintes auto-imposées ne sont évidemment pas en elles-mêmes une preuve de code inviolable. Nous pensons néanmoins qu'elles représentent, au delà de la réflexion sur les architectures matérielles et logicielles, un parfait complément de défense en profondeur. Elles apportent une meilleure confiance dans les binaires générés pour la plateforme cible (et s'inscrivent dans une démarche d'amélioration de la qualité, de la lisibilité et de la simplicité du projet dans son ensemble).

Contraintes de développement et leitmotivs Nous nous sommes imposé comme contraintes de développement sur le ST33G1M2 (côté plateforme et *token*) ainsi que sur le *stage 1* du SoC :

1. aucune allocation dynamique : nous utilisons des tableaux de taille fixe et connue à la compilation ;
2. pas d'utilisation de la `libc`, mais plutôt une récupération ou réécriture des fonctions dont nous avons besoin (ce qui nous amène souvent à réfléchir à la réelle nécessité d'utiliser telle ou telle fonction externe) ;

[25] Il faut dans ce cas éviter les TOCTOU (*Time Of Check Time Of Use*)

3. base de code critique linéaire, sans évènement asynchrone (i.e. pas d'interruption), avec des automates à états simples et aisément analysables ;
4. maîtrise parfaite de toutes les zones de code et de données stockées en mémoire flash et en mémoire RAM ;
5. utilisation de protocoles cryptographiques à l'état de l'art (que cela soit pour les signatures des mises à jour, pour de l'authentification mutuelle, etc.). Les implémentations doivent utiliser les accélérateurs cryptographiques certifiés lorsque cela est possible, et implémenter des contre-mesures aux attaques par canaux auxiliaires (*timing*, observations EM, injections de fautes, etc.) lorsque celles-ci ne sont pas fournies par le matériel ;
6. maîtrise complète de la chaîne de production des binaires qui vont s'exécuter sur la plateforme (plus de détails en sous-section suivante) ;
7. pousser au maximum l'analyse statique de code (plus de détails en sous-section suivante) ;
8. factoriser au maximum le code entre les divers composants de l'équipement (ST33, SoC, *token*, Cortex-M0). Ceci permet de minimiser la base d'analyse de code et de s'assurer que les mêmes logiques/automates sont utilisés lorsque cela est possible. Cela nécessite d'isoler des drivers matériels spécifiques à chaque plateforme, et d'abstraire le reste du code au dessus de ceux-ci. Un exemple explicite est l'abstraction des communications sur bus qui est commune au SPI et à l'I^2C mais également au SoC et aux composants sécurisés. Un autre exemple est l'utilisation d'une bibliothèque cryptographique dont une grande partie du code est partagée (plus de détails en 4.5).

Chaînes de compilation, outils de production Comme nous l'avons déjà évoqué, l'un des objectifs du projet eurisko est la maîtrise totale du code produit, et ceci nécessite la maîtrise des chaînes de compilation et autres outils permettant de générer les divers binaires qui s'exécutent sur la plateforme. Nous distinguons ainsi trois principales chaînes de compilation qui correspondent aux trois types de composants utilisés dans le projet :

- **le SoC Marvell A370** : il utilise une chaîne de compilation croisée GNU/gcc classique pour architecture ARMv7-A ;

- **le Cortex-M0** : il utilise une chaîne de compilation croisée GNU/gcc classique pour architecture ARMv6-M et cible Cortex-M0 ;

- **le ST33G1M2** : il est normalement associé à une chaîne de compilation croisée utilisant l'outil Keil et le compilateur ARMCC sous Windows, ainsi que le SDK ST33G1M2 associé. Bien que cette chaîne de compilation fonctionne parfaitement, son aspect fermé et propriétaire la rend limitée : elle rend plus difficile l'intégration d'outils ou scripts complémentaires durant les diverses phases allant des sources au binaire final (par exemple des scripts de signature, ou des scripts de *sanity check* des éléments produits à chaque phase). Nous avons donc porté le SDK ST33G1M2 pour qu'il puisse être intégré à une chaîne GNU/gcc pour ARMv7-M Cortex-M3. Nous pouvons dès lors générer tous les binaires pour le composant sécurisé sous Linux via des `makefiles`, et y ajouter tous les scripts nécessaires dans la chaîne de production du *firmware*.

Même si le choix des *toolchains* peut paraître anecdotique, celui-ci a un impact important sur la sécurité des développements (voir la sous-section suivante).

Passer du temps à comprendre et à sélectionner les bonnes options du compilateur, à développer et à maîtriser sa chaîne de `makefiles`, son mapping mémoire et ses *linker scripts* n'est au final pas un travail anodin.

Analyse et validation du code Cette section couvre les différentes stratégies mises en oeuvre dans le projet pour permettre l'analyse du code développé. L'approche choisie ici est pragmatique : nous n'avons pas l'objectif (ni la prétention) de produire ou d'utiliser un formalisme donnant des preuves sur le code ; nous apportons plutôt la vision du développeur de C « embarqué » qui a à sa disposition les outils d'analyse classiques de l'industrie.

Il convient également de noter que cette section n'a pas vocation à comparer des outils existants, payants ou gratuits, que nous aurions passés sur le projet pour produire un classement de ceux-ci. Elle vise plus à démontrer qu'il est possible de s'assurer via quelques compromis en amont du projet que son code sera analysable durant et en sortie de projet.

Code *Contraintes de développement*

L'analyse de code dans un projet mêlant des développements pour différentes cibles (SoC applicatif, microcontrôleur) mais également pour des besoins aussi variés que la cryptographie ou le support de blocs matériels nécessite une prise en compte minimale en amont dans le projet.

Une des premières étapes du travail visant à rendre le code développé pour la plateforme analysable a donc consisté à s'imposer les contraintes suivantes en vue d'une analyse de ce code ; celles-ci permettant de garantir une simplicité du code et une auditabilité à la fois par des humains et des outils.

- langage C simple (C99 basique) ;
- typage fort (dans les limites du C) ;
- aucune allocation dynamique ;
- aucune dépendance externe ;
- code synchrone et linéaire.

Même si ces contraintes ne sont pas suffisantes pour garantir que le code sera analysable par des outils, nous verrons qu'elles permettent pour certaines de s'approcher des capacités des outils.

Un autre élément rapidement mis en oeuvre dans le projet, notamment pour l'ensemble du code développé pour le composant sécurisé, a été la capacité à compiler celui-ci via différentes *toolchains* (ARMCC au travers de l'environnement de développement KEIL, gcc et clang au travers de `makefiles` simples). Cet effort s'est montré payant pour au moins deux raisons :

1. il permet en pratique - via le passage par le spectre d'outils différents - de disposer de retours complémentaires sur le code. Pour ne citer qu'un exemple trivial, concernant les remontées d'avertissement sur le code non atteignable, le comportement de compilateurs est le suivant :
   - **ARMCC** : ARMCC informe nativement sur le code non atteint d'une fonction (e.g. un `return` précédent un `break` dans un `switch`/`case`) ;
   - **gcc** : un gcc récent (e.g. 4.9) configuré en `-W -Wall -Wextra` restera silencieux sur la présence de code non atteignable. Même l'utilisation d'une option `-Wunreachable` explicite (incluse dans `-Wall`) ne produira aucun effet ;
   - **clang** : le même code traité par clang avec les mêmes options produira bien quant à lui ce type d'avertissement.
2. une bonne partie des outils d'analyse statique prennent en charge du code préprocessé et sont parfois même capables de s'intégrer directement à un jeu de `makefiles` pour bénéficier de l'ensemble des logiques d'inclusion et de définitions de variables dans le projet.

Par ailleurs, le choix de *toolchains* récentes permet de bénéficier de protections complémentaires sur le code produit, permettant notamment

de complexifier voire empêcher des exploitations. Ceci n'est généralement obtenu que par l'appel des options adéquates. Un premier exemple classique est l'option `-fstack-protector-all` de gcc, disponible à partir de la version 4.1. Un second est le fait de pouvoir générer du code eXecute-Only (`XO`) i.e. sans droit de lecture[26]. Cela permet de gérer le cloisonnement mémoire de manière plus fine : un attaquant pouvant détourner le flot d'exécution ne pourra pas accéder en lecture au code qui s'exécute, ce qui compliquera son travail. La production de code `XO` sur ARM est au moins supportée par ARMCC (via l'option `-execute_only`) et clang ; gcc ne fournit pas quant à lui ce type d'option.

Par ailleurs, les compilateurs comme gcc ou clang permettent aussi d'étendre les fonctionnalités d'analyse et de robustesse du code via des *plug-ins* (par exemple la « constification » des structures faite par le *plug-in* `CONSTIFY` de grsec).

Code *Analyse statique*

Différentes stratégies de conception et d'analyse de code existent, auxquelles sont associées différents outils. Nous ne considérons ici que ceux capables de s'adapter au code développé en langage C complet, et non ceux nécessitant l'utilisation d'un langage spécifique (par exemple, sous-ensemble strict du C ou annotations poussées).

Les outils d'analyse statique sont capables de détecter différentes classes de bugs : *buffer overflow*, *integer overflow*, accès à des zones non initialisées, manipulation et déréférencement de pointeurs invalides, divisions par zero, etc. Ils se différencient entre eux par le niveau de garantie que leur analyse fournit. Notamment, certains outils dits **sound** assurent que tout bug (dans la limite des classes couvertes) sera listé en fin d'analyse ; en contrepartie, ces outils produisent en général un nombre plus important de faux positifs.

Comme évoqué précédemment, le code développé dans le cadre du projet présente certaines particularités qui complexifient la prise en charge par les outils d'analyse statique. Ceux-ci sont listés ci-dessous.

Code embarqué : le développement *bare metal* sur un composant sécurisé nécessite pour diverses raisons (mise en place de zones mémoires dédiées comme une pile, exécution de zones de code associées à un *stage*, performance) l'écriture de routines en assembleur (ARM, voire Thumb-2 dans notre cas). Ce type de code n'est nativement pas pris en compte par les outils d'analyse. Au mieux, certains proposent d'ignorer ce code en

[26] Notamment en évitant de placer les *literal pools* dans la section de code

considérant une absence d'effet de bords (ce qui est généralement faux), ou à défaut de définir des *placeholders*.

Un autre aspect associé au développement bas niveau sur composant concerne la manipulation régulière de registres I/O mappés permettant de contrôler les fonctionnalités d'un bloc donné (e.g. piloter un bloc SPI, les GPIO d'un bus I2C *bitbangé*), parfois en lien avec une zone de SRAM du bloc. Ces accès en lecture et écriture dans des zones mémoires non allouées et initialement inconnues de l'outil sont logiquement assez mal perçus par les outils d'analyse. Ils nécessitent en règle générale un travail d'analyse et d'annotation manuelle préalable.

Code crypto : une bibliothèque cryptographique implémente les fonctionnalités de signature sur courbes et contient diverses couches (support de grands entiers, routines et fonctions pour $\mathbb{F}_p$, formules sur courbes). En pratique, le code associé passe la majorité de son temps à réaliser des manipulations de tableaux de mots représentant de grands entiers. Dans ce cadre, les limitations imposées initialement sur le code et notamment les contraintes strictes d'allocations statiques permettent de simplifier le travail des outils en leur permettant souvent de converger dans des cas dans lesquels l'utilisation d'allocations dynamiques préviendrait cette convergence.

Code *Compléments d'analyse*

Nous avons pu voir que la mise en œuvre de quelques contraintes en amont dans le projet permettent de faciliter la prise en charge du code par les outils d'analyse. Quelques efforts complémentaires (*placeholders*, corrections, réécriture, annotations, etc.) permettent pour certains outils (dits *sound*) d'obtenir en fin d'analyse une garantie formelle d'absence de bugs. Malgré tout, ces garanties fournies par l'analyse via de tels outils ne sont applicables qu'à certaines classes de bugs présentées précédemment.

Malgré tout, ces outils ne sont évidemment pas capables de remonter les éventuels bugs logiques présents dans le code (inversion d'une logique de prise en compte d'une valeur de retour de fonction, absence d'effacement d'un buffer critique en sortie de fonction, inversion de l'ordre d'exécution de deux fonctions, etc.), sauf au travers des effets de bords et des bugs que ces erreurs logiques peuvent introduire.

Il est donc nécessaire de compléter l'utilisation de tels outils par des passes d'analyse manuelle du code produit. A cet effet, la simplicité du code et ses propriétés de synchronisme permettent de simplifier ce travail d'analyse manuelle. De la même manière, l'emploi de différents automates

de traitement simples aux conditions d'entrée et de sortie claires permet encore de simplifier les étapes de reprise et de revue de code en limitant la charge contextuelle pour le relecteur.

4.5 Bibliothèque cryptographique

Une bonne partie des propriétés de sécurité attendues pour le démarrage de la carte puis par les services rendus par le composant ainsi que par la distribution nécessite la mise en œuvre de fonctions cryptographiques.

Après analyse des solutions existantes, il a été décidé pour des questions de maîtrise de développer *from scratch* les briques nécessaires, et notamment une bibliothèque de courbes elliptiques supportant divers mécanismes de signature. Les différents éléments suivants ont été développés pour celle-ci :

- bibliothèque de grands entiers (structures et fonctions associées) ;
- structures et fonctions pour le support de $\mathbb{F}_p$;
- structures et fonctions pour le support des courbes elliptiques ;
- divers mécanismes de signature (EC-*DSA).

La bibliothèque a été développée en s'imposant un certain nombre de contraintes, précédemment détaillées en 4.4. Les contraintes suivantes viennent s'y ajouter :

- pas d'objectif de performance à tout prix (e.g. préférer une fonction à temps constant à une fonction plus efficace mais moins sûre) ;
- typage fort - dans les limites du langage C - des structures des diverses couches de la bibliothèque (grands entiers, $\mathbb{F}_p$, courbes) ;
- vérification de pré et post conditions en entrée et sortie de fonction ;
- indépendance vis-à-vis de la taille des mots machines (support de mots de 16, 32 et 64 bits indépendamment de l'architecture) ;
- tests unitaires et de non-régression ;
- vecteurs de test pour les principaux algorithmes de signature implémentés.

5 Conclusion

Les sections précédentes ont permis de présenter en détail et de manière progressive l'architecture matérielle et logicielle de la carte. Cette section vise maintenant à analyser le résultat obtenu du point de vue de la sécurité, à fournir un retour d'expérience sur le projet et à considérer les évolutions possibles.

5.1 Analyse de la sécurité obtenue

L'objectif de sécurité principal attendu était de garantir - à tout instant et notamment au démarrage - l'intégrité du code tournant sur le processeur principal de la carte. A cet effet, les protections et contre-mesures reprises ci-après ont été intégrées au design matériel et logiciel de la carte.

Celles-ci peuvent être classées en fonction de leur efficacité avérée ; certaines fournissant en effet des garanties fortes, d'autres n'agissant que comme retardant vis-à-vis d'un attaquant ou participant à une logique de défense en profondeur.

Pour ne pas alourdir le document et notamment ses premières sections, il a été décidé de ne pas présenter jusqu'ici les différents modèles d'attaquants et leurs capacités, même si ces éléments ont été pris en compte lors de la conception de la carte. Cette sous-section vise à corriger ce manque.

L'analyse menée montre tout d'abord que l'assemblage d'éléments COTS (*Commercial Off-The-Shelf*) seuls et notamment l'utilisation d'un SoC embarquant du code non maîtrisé pour gérer son boot (i.e. sa BootROM) ne permet pas d'obtenir de garantie forte sur l'intégrité du code déployé initialement sur la plateforme.

Ce constat est valide dans le cas d'une BootROM ne fournissant pas de mécanismes de vérification d'intégrité cryptographique, puisqu'un attaquant aura tout loisir de remplacer le code de démarrage bas niveau par celui de son choix.

Le cas d'un SoC dont la BootROM embarque ce type de mécanisme reste en pratique tout aussi problématique, puisque le niveau de sécurité obtenu au final n'est pas évaluable. L'acceptation d'un tel mécanisme comme racine de confiance revient donc à supposer que les implémentations matérielles (*fuses*), cryptographiques (logique d'intégrité) et logicielles (automate d'état et code de la BootROM) sont toutes exemptes de bugs ou de fonctionnalités non désirées, alors qu'elles n'ont pas été étudiées de manière indépendante. Ceci revient donc à faire un pari sur la bonne foi et les compétences du vendeur. A ce titre, il nous semble donc pertinent de considérer, comme évoqué précédemment, que les fonctionnalités de sécurité d'une telle BootROM non évaluée peuvent au mieux servir des objectifs de défense en profondeur.

Au final, le moyen retenu pour obtenir des garanties quantifiables sur le démarrage de la plateforme a donc consisté à recentrer celles-ci sur un composant de confiance certifié à un niveau EAL5+. Le code de démarrage de la plateforme est donc stocké dans la flash protégée du composant ; ceci incluant le code du composant lui-même mais également le code servi par celui-ci au SoC lors de son démarrage. Deux éléments complémentaires

permettent de réellement garantir d'un point de vue logique un contrôle de la plateforme par le composant :

- le démarrage du SoC via son interface SPI (4 signaux) sur le composant : ce dernier contrôle intégralement ce qui peut-être chargé par celui-ci sur son interface de démarrage ;
- le contrôle de la patte de *reset* du SoC (1 signal) par le composant : ce dernier a donc droit de vie et de mort sur le SoC. Il est en capacité de le faire démarrer, de le mettre en *reset* en cas de comportement déviant (*timing* de chargement incorrect, *heartbeat* non reçu, trames incorrectes, etc.).

Malgré tout, même si cette analyse s'avère valable face à un attaquant disposant d'un accès ponctuel au code tournant sur le SoC, elle ne tient pas nécessairement face à un attaquant disposant de compétences matérielles et d'un accès local (ponctuel ou de longue durée) à la plateforme. En effet, il convient en plus d'analyser sa capacité à détourner les signaux précédents, voire à remplacer complètement le composant sécurisé par un équivalent maîtrisé par lui. Pour adresser ces points, les contre-mesures suivantes ont été mises en œuvre :

- utilisation de composants au format BGA (SoC et composant sécurisé) ;
- enterrement des signaux sensibles (SPI et *reset*) ;
- « non-routage » de l'interface JTAG du SoC ;
- invalidation du mécanisme de *recovery* associé à l'UART via :
  - désactivation de l'optocouplage au *boot* ;
  - suivi du *timing* de démarrage du SoC sur le bus SPI.

Ces contre-mesures peu coûteuses ne sont bien entendu que des **retardants** face à un attaquant motivé et outillé mais elles limitent malgré tout fortement ses capacités d'actions sur le terrain. En effet, contrairement à un package SOIC voire QFP qu'il est envisageable de dessouder sur place, s'attaquer à un SoC dans un format BGA de près de 300 pins ou un composant au format WLCSP pour s'insérer ou le remplacer nécessite un environnement de travail adapté. L'accès aux pistes enterrées (voire leur reroutage) est également envisageable mais nécessite de la même manière un outillage difficilement transportable.

Même si ce type d'opération n'est pas inconcevable dans le but de piéger une plateforme, elle nécessite un accès physique, du temps, de l'expérience et du matériel. Il convient également de noter deux aspects supplémentaires que l'attaquant devra prendre en compte :

- il faudra que les modifications physiques réalisées ne soient pas visibles à l'utilisateur. En pratique, si la plateforme est intégrée à un boîtier fermé, il lui sera probablement possible de les masquer au moins temporairement ;
- il faudra que les modifications fonctionnelles soient invisibles à l'utilisateur, i.e. que la plateforme semble rendre le service attendu tout en favorisant l'attaquant.

En fonction du rôle de la plateforme, ce dernier point peut devenir difficile à adresser. Sans contre-mesure supplémentaire, l'attaquant disposant d'une capacité à lire le bus SPI sera capable d'accéder au *stage 1* montant un canal sécurisé avec le composant pour servir au SoC le *stage* suivant et les secrets de déchiffrement de la μSD. Il lui sera donc nécessaire de se rendre actif sur le bus SPI pour aller plus avant dans la récupération des secrets protégeant le contenu de la carte μSD.

Si l'on ne préjuge pas de ses capacités et que l'on considère pour le moment qu'il en est capable, on peut noter que les secrets utilisateur restent protégés. Ils sont en effet enracinés dans le composant sécurisé (e.g. clés d'authentification ou de signature) et restent inaccessibles.

Comme évoqué ci-dessus, un attaquant motivé disposant du temps, des compétences et du matériel nécessaire parviendra à accéder à l'ensemble du code dédié au SoC (*stage 2* et contenu de la μSD). En pratique, il est possible de parer à cette éventualité en limitant totalement le déploiement de code à l'extérieur du composant. Ceci permet également de complexifier son analyse du code déployé.

C'est à cet effet que le mécanisme d'authentification pré-boot a été intégré à la plateforme. Pour atteindre son but, il sera donc nécessaire à l'attaquant de s'emparer d'une plateforme et d'un *token* d'authentification (voire d'un *pin* en cas de couplage d'un *pin* au *token*). A défaut, celui-ci devra se contenter d'émuler intégralement et de manière fidèle le comportement de la plateforme sans avoir accès au code déployé nominalement. On notera que cette contre-mesure techniquement très efficace repose malgré tout en partie sur la coopération de l'utilisateur (i.e. le possesseur légitime du *token*). C'est néanmoins une garantie **formelle** de sécurité de la plateforme lorsqu'elle est respectée.

Nous verrons en sous-section 5.3 qu'il est envisageable avec un peu d'effort côté *token* de complexifier la tâche de l'attaquant et de permettre à l'utilisateur de détecter certains types de piégeages.

5.2 Retour d'expérience

Cette sous-section livre un retour d'expérience sur divers aspects de ce projet de prototypage.

Ressources humaines, financières et temporelles Concernant l'aspect financier, un projet comme celui-ci reste au final dans des montants raisonnables. Dans notre cas, le premier *run* de huit cartes fonctionnelles a nécessité de débourser auprès d'un sous-traitant efficace de l'ordre de 40000 euros.

Il convient de noter que le choix du sous-traitant constitue un aspect essentiel pour éviter toute déconvenue majeure. Il est ainsi nécessaire de prendre en compte l'expérience de celui-ci sur les composants principaux envisagés et les projets déjà réalisés sur ce type de composant.

Concernant l'aspect temporel, l'obtention de ce premier *run* a nécessité (une fois les caractéristiques précises définies) un peu moins de 20 semaines correspondant au travail de conception (sélection des composants, routage, vérifications), approvisionnement des composants, fabrication des PCB, pose des composants et tests.

Au final, ceci démontre qu'il est possible d'obtenir des prototypes matériels fonctionnels pour un coût et dans un temps très raisonnables. Il convient néanmoins de replacer cette étape précise dans l'ensemble du projet. Pour des raisons de concision, nous passerons sur les aspects amont ayant attrait aux marchés publics pour nous focaliser sur le temps nécessaire à la prise en main des composants (et notamment au debug) ainsi qu'au développement (conception, code, analyse).

Ce projet de prototypage ayant été réalisé en grande partie sur les moments de disponibilité des intervenants des différents laboratoires, il est au final difficile de quantifier précisément le temps total passé. Néanmoins, la prise en main des blocs matériels et fonctionnalités non encore maîtrisés des composants s'est révélée être une étape parmi les plus coûteuses en temps.

Il nous semble donc *a posteriori* très difficile voire impossible de prévoir *a priori* la durée nécessaire à la prise en main d'éléments matériels nouveaux. Ceci vient donc appuyer l'intérêt réel de prototyper ; ce travail amont permet de garantir à la fois le fonctionnement d'une brique spécifique pour un projet utilisateur mais également d'éviter des retards dans un projet.

Concernant les ressources allouées aux développements liés au matériel (architecture, code et debug), il est clair qu'un travail en binôme ou plus permet au final de gagner du temps.

Composants sécurisés Un des principaux aspects du projet a au final concerné le travail sur composant sécurisé ; du développement des drivers bas niveau (émulation de flash SPI, bus I^2C, accélérateur cryptographique, etc.) à la mise en œuvre de l'architecture logicielle (logique de *stages*, de mise à jour, etc.).

Au final, le monde des composants sécurisés reste sur beaucoup d'aspects très proche du monde des microcontrôleurs (PIC, ATmega, etc.) : ces produits sont en effet animés par des versions sécurisées de Cortex-M3, cœur ARM que l'on retrouve classiquement en embarqué. Le développement pour ce type de cible reste donc très proche du matériel ; une grande partie du temps est passée à utiliser un debugger hardware, un analyseur logique voire un oscilloscope.

Même si ceci représente un petit investissement en temps, un aspect appréciable de ce type de cible est qu'il est facile de faire en sorte que l'intégralité du code mis en œuvre soit totalement maîtrisé car développé spécifiquement. En pratique, l'approche prise dans le projet a ainsi effectivement été de partir d'une feuille totalement blanche et de réaliser l'ensemble des développements uniquement sur la base de la *datasheet* du composant. Le code a même en pratique été développé de manière à permettre les phases de compilation et d'édition de liens à la fois avec ARMCC sous Keil et via gcc ou clang.

Au final, s'abstraire de bibliothèques externes non maîtrisées ou même d'un OS sous-jacent permet de définir ses propres contraintes de développement, notamment sur les aspects sécurité. Dans notre cas, la simplicité et la sécurité sont restées les principaux objectifs, dans le but de permettre une auditabilité du code, notamment par des outils d'analyse statique. Il a également été décidé de tirer progressivement parti de l'ensemble des fonctionnalités de sécurité du produit (MPU, utilisation des accélérateurs cryptographiques matériels, *jitter*, etc.).

Il convient tout de même de préciser ici que la possibilité de réaliser ses développements *bare-metal* sur ce type de cible peut tout à fait mener à une diminution de la sécurité. Par exemple, si les mécanismes de protection supportés par le matériel (e.g. MPU) ne sont pas mis en œuvre (comme ceci peut parfois l'être nativement dans l'environnement d'un OS), tout bug exploitable (lié à une absence d'analyse de code) donne potentiellement un accès complet à la cible. De la même manière, le fait de travailler *bare-metal* impose de gérer soi-même la répartition des tâches s'exécutant dans les différents modes supportés par la cible. Comme nous l'avons déjà évoqué, même si un composant sécurisé évalué à un niveau EAL5+ fournit

une base matérielle de confiance, il ne vient en aucun cas améliorer le niveau de sécurité d'un mauvais code.

Une part importante du travail de développement sur ce type de composant est donc liée à la mise en œuvre des différents mécanismes de sécurité et la validation du code développé. En effet, si la prise en charge de code haut niveau par des outils d'analyse reste généralement assez simple (si le code a été développé dans cette optique), la prise en charge de code bas-niveau (utilisant des routines assembleurs, touchant des zones de mémoire associées à des registres IO-mappés ou encore un *layout* mémoire dédié) nécessite plus de travail.

5.3 Evolutions envisagées

Le projet a permis aux intervenants de monter en compétences ou d'améliorer celles-ci sur divers sujets (composants sécurisés, aspect matériel, développement sécurisé, architecture, cryptographie, etc.). Il a également permis de disposer de nouvelles briques logicielles (bibliothèque de courbes elliptiques, OS composant, bootloader, etc.) et matérielles (plateformes obtenues, *token*, etc.). Ces éléments ont d'ores et déjà permis d'envisager des évolutions au projet, ils ont aussi donné des idées de nouvelles pistes à explorer.

Bibliothèque cryptographique Concernant la bibliothèque cryptographique, celle-ci est actuellement bien avancée. Elle a déjà subi diverses passes d'analyse qui donne un premier niveau de confiance dans son code.

Il reste néanmoins à finaliser ce travail. De plus, certains aspects concernant la résistance aux attaques par canaux auxiliaires nécessitent d'être encore améliorés : il est notamment prévu de passer plus de temps sur les aspects liés au temps constant.

Un autre travail restant encore à réaliser concerne l'amélioration de la documentation interne associée aux différents algorithmes de signature supportés.

Il est en tout cas envisagé, à terme, de publier cette bibliothèque.

Retour utilisateur sur l'intégrité de la plateforme Comme évoqué en sous-section 5.1, il reste envisageable pour un attaquant d'un niveau élevé de remplacer le composant sécurisé présent sur la plateforme par un équivalent fonctionnel maîtrisé (celui-ci ne pouvant malgré tout disposer des secrets enfouis dans le composant initial).

L'utilisation d'un *token* actif tel que présenté dans le projet ne permet en pratique pas de contrer ce scénario, celui-ci ne fournissant aucun retour de confiance à l'utilisateur.

Une évolution envisageable consisterait donc à travailler sur un *token* fournissant un retour à l'utilisateur (e.g. via une LED pilotée par le composant du token suite à authentification du composant plateforme) sur l'intégrité de la plateforme. Ceci imposerait alors à l'attaquant de modifier matériellement la plateforme pour conserver au moins en partie le composant sécurisé pour ne pas être détecté par l'utilisateur.

Prototypages fonctionnels La disponibilité de la plateforme permet d'envisager de continuer le prototypage à plus haut niveau de différents types d'équipements, pour explorer plus avant les divers aspects techniques associés : HSM, chiffreur IPsec/IKE, sonde réseau, téléphone IP (les pins du bloc TDM de l'A370 sont en effet routées vers l'I/O header de la carte).

Dans des cas courants, ces différents types d'équipements peuvent, en effet, bénéficier directement de la présence d'un composant cryptographique utilisable comme ressource mais également de la logique de démarrage sécurisé présentée dans l'article.

Evolution du SoC Comme évoqué précédemment, le SoC Marvell sélectionné dans le cadre du prototypage l'a été en grande partie pour des raisons fonctionnelles (support, consommation, fonctionnalités réseau, etc.). Pour bénéficier d'autres fonctionnalités ou compléter certains aspects liés à la sécurité, l'architecture de la plateforme pourrait être modifiée pour mettre en œuvre un FPGA, un SoC/FPGA, un SoC bénéficiant d'un mécanisme de boot sécurisé propriétaire et/ou de TrustZone, un SoC 64-bit plus puissant (e.g. ARMv8).

Références

1. Andrew « bunnie » Huang. Keeping Secrets in Hardware : the Microsoft XBox Case Study. `http://web.mit.edu/bunnie/www/proj/anatak/AIM-2002-008.pdf`, 2002. CHES.
2. ANSSI. Page certification du ST33G1M2. `http://www.ssi.gouv.fr/certification_cc/microcontroleur-securise-st33g1m2-revision-f-firmware-revision-9-incluant-optionnellement-la-bibliotheque-cryptographique-neslib-4-1-et-la-bibliotheque-mifare-desfire-ev1-revision-3-7-ou/`, 2014.
3. ANSSI. Rapport de certification ANSSI-CC-2014/46 du ST33G1M2. `https://www.commoncriteriaportal.org/files/epfiles/ANSSI-CC-2014-46.pdf`, 2014.

4. ARM. ARM Security Technology. Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.
5. Di Shen. Exploiting Trustzone on Android. https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf, 2015. Black Hat.
6. J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, Edward W. Felten. Cold Boot Attacks on Encryption Keys. https://citp.princeton.edu/research/memory/, 2008. USENIX.
7. Kurt Rosenfeld, Ramesh Karri. Attacks and Defenses for JTAG. http://isis.poly.edu/~kurt/papers/design_and_test_final.pdf, 2009.
8. laginimaineb. Full TrustZone exploit for MSM8974. http://bits-please.blogspot.fr/2015/08/full-trustzone-exploit-for-msm8974.html, 2015.
9. Sergei Skorobogatov. Low temperature data remanence in static RAM. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf, 2002.
10. Sergei Skorobogatov, Christopher Woods. Breakthrough silicon scanning discovers backdoor in military chip. https://www.cl.cam.ac.uk/~sps32/ches2012-backdoor.pdf, 2012. CHES.
11. STMicroelectronics. ST33G1M2 Product brief - Secure MCU with 32-bit ARM® SecurCore® SC300™ CPU, SWP, ISO, SPI and GPIO interfaces and high-density Flash memory. http://www2.st.com/content/ccc/resource/technical/document/data_brief/81/c6/7d/4e/ae/23/4e/72/DM00095982.pdf/files/DM00095982.pdf/jcr:content/translations/en.DM00095982.pdf, 2016.
12. Vincent Strubel. CLIP : une approche pragmatique pour la conception d'un OS sécurisé. https://www.sstic.org/2015/presentation/clip/, 2015. SSTIC.