Développement d'une chaîne de boot sécurisée

Rapport de stage à Prove & Run

Dorian Lesbre

Supervisé par Patrice Hameau

Mars - Juillet 2020

Résumé

Ce rapport décrit le travail fait pendant mon stage de première année de master, effectué à Prove & Run, Paris. J'y ai travaillé sous la supervision de Patrice Hameau de mi-mars à fin juillet.

L'objectif du travail était l'étude et l'implémentation d'une chaîne de boot sécurisée sur microprocesseurs i.MX. Plus précisément, j'ai développé un bootloader capable de vérifier dans la mesure du possible l'intégrité du processeur, valider la signature du code de l'OS avant d'exécuter ce-dernier, et mettre à jour le code de l'OS si le nouveau code provient de source sure.

Ce travail s'est divisé en trois parties. Dans un premier temps, une phase de recherche sur l'offre existante de bootloaders sécurisés, des problèmes qu'ils faudra résoudre, et des algorithmes cryptographiques à utiliser. Ensuite, j'ai écrit une première version de cahier des charges auquel le programme devait répondre, et un premier pseudo-code. Finalement j'ai l'implémenté le programme et les librairies nécessaires en C sur PC.

Table des matières

1 Introduction

L'objectif du stage était d'étudier et coder un bootloader sécurisé pour processeurs i.MX de NXP (ce sont des architectures ARM Cortex A). Un tel programme est sensé vérifier et charger un noyau sécurisé et sert donc d'origine à la chaîne de confiance. Ce bootloader doit principalement répondre à trois besoins : s'assurer de l'état du processeur et l'intégrité de la mémoire, s'assurer que le code du noyau est valide et authentique avant de l'exécuter, et permettre de mettre à jour le noyau (en vérifiant l'authenticité de la nouvelle version).

Afin d'éviter de bloquer complètement le processeur dans des cas indésirables, le bootloader est conçu pour tenter de charger une version authentifié du code même en cas d'erreur de mise à jour, de corruption de données non-essentielles, ou de toute autre erreur qui n'empêche pas la vérification et le chargement du code. L'objectif principal est donc d'assurer une certaine robustesse du système et pas de le faire planter aux premières anomalies.

Ces fonctionnalités ne constituent pas la liste exhaustive des fonctionnalités du produit final. Il s'agit de celles sur lesquelles j'ai le plus travaillé. Afin d'assurer vraiment une chaîne de confiance, il faudrait également gérer le cycle de vie du processeur : configuration initiale au premier boot et décommission sécurisée.

Mon travail consistait à concevoir et implémenter ce bootloader avec l'aide de mon maître de stage. J'ai fait l'étude de l'offre existante et des fonctionnalités nécessaires, participé au choix des algorithmes cryptographiques à utiliser, établi les chaînes de clé utilisées, et ai implémenté en C sur ordinateur une version du bootloader et des librairies cryptographiques nécessaires. Du fait de la situation sanitaire, il m'a fallu travailler en télétravail pendant l'essentiel du stage, en restant en contact avec mon encadrant par téléphone. Cela nous a également empêcher de porter le programme sur microprocesseur.

2 Description du bootloader

2.1 Les processeurs i.MX

Ce bootloader est conçu pour un microprocesseur i.MX 6, 7 ou 8, fabriqué par NXP. C'est une famille de puces qui offrent plusieurs systèmes sécuritaires. Le principal système utilisé est la mémoire OTP (One Time Programmable). Il s'agit d'une petite mémoire persistante qui ne peut être écrite qu'une seule fois. C'est à partir de cette mémoire que partiront toute les chaînes de confiances utilisées par le bootloader.

Du fait de sa petite taille, la plupart des données ne sont pas stockées dans la mémoire OTP mais dans la mémoire flash, beaucoup plus grande. Cette dernière peut être découpée en quatre partitions, mais seules deux nous servirons. La partition boot 1 de la mémoire flash contiendra toutes les informations internes du bootloader, tandis que la partition boot 2 contiendra toutes les informations communiquées entre le bootloader et le niveau de boot suivant (le noyau de l'OS). Elle sert notamment pour les mises à jour du noyau.

Certains processeurs i.MX possèdent un module d'accélération cryptographique appelé CAAM. Ces modules peuvent être utilisés pour optimiser les calculs cryptographiques et offrent également un générateur de nombres aléatoires. Toutefois, comme ils ne sont pas présents sur chaque puce, j'ai recodé les algorithmes nécessaires.

Ces processeurs proposent aussi un mode appelé High Assurance Boot (HAB) qui permet de charger de manière sécurisée le code de notre bootloader au démarrage. Cela

permet de s'assurer que notre code est la première chose qui s'exécute au démarrage.

2.2 La séquence de boot standard

Lors d'une séquence de boot, le programme doit vérifier, dans la mesure du possible, l'intégrité de la puce. Cela inclut notamment la vérification des numéros de série de la puce de la mémoire flash pour empêcher les échanges de mémoire entre diverses puces. Pour éviter que la mémoire flash ne soit remplacée par une version précédente, un système de compteur monotonique est utilisé : les mémoires flash et OTP partagent un compteur qui est incrémenté après des évènements critiques (comme les mises à jour). Si jamais des erreurs apparaissent durant ces vérifications, le programme doit s'interrompre et l'OS ne sera pas démarré. Cela évite une exécution avec un hardware compromis qui pourrait faire fuir des données sensibles ou avoir un comportement inattendu. Si l'erreur peut s'expliquer par un défaut de lecture, la puce sera réinitialisée pour retenter un boot normal.

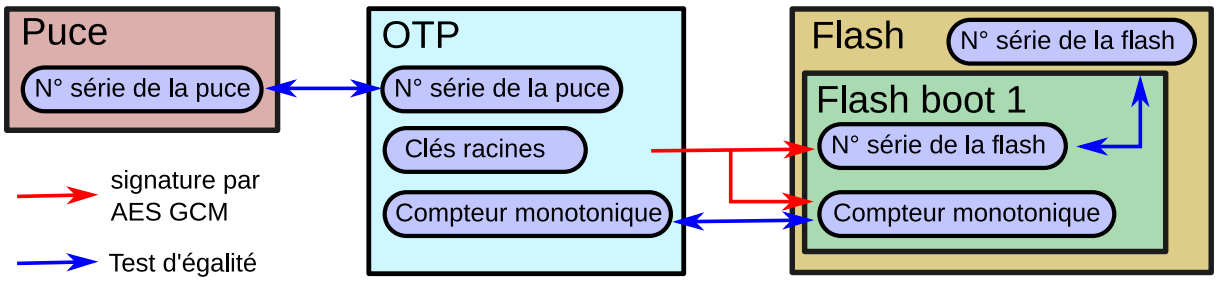

FIGURE 1 – Vérification du hardware lors d'une séquence de boot

Une fois ces vérifications faites, le bootloader regarde s'il y a une mise à jour à effectuer. Si ce n'est pas le cas, il vérifie la signature et l'état du code stocké en mémoire. Lorsque le code passe les tests d'authentification, le bootloader le déchiffre, le charge en mémoire puis l'exécute. S'il ne passe pas les tests, alors la puce est réinitialisée puis la séquence de boot redémarre pour essayer à nouveau. Au bout d'un certain nombre d'essais infructueux, le bootloader cherche si un code valide se trouve dans l'un des emplacements mémoire réservés aux mises à jour. Si oui, il l'installe à la place du code défectueux et le lance.

2.3 Gestion des mises à jour

Les ressources étant assez limitées à ce niveau de boot, ce n'est pas le bootloader qui va chercher des mises à jour en ligne auprès des fournisseurs. L'obtention d'une nouvelle version du code est donc effectuée par l'OS. Le bootloader se contente de prendre la version que l'OS lui a préalablement installé dans un espace mémoire prédestiné, la vérifier et l'installer si elle est valide.

Pour garantir un échange sécurisé avec les fournisseurs, le bootloader génère au premier boot un couple de clés publique/privée et diffuse la clé publique. Celle-ci est donc connue des fournisseurs de mise à jour et sert à obtenir le secret partagé qui permet d'envoyer le code de manière chiffrée.

Pour des raisons de sécurité, une mise à jour doit pouvoir reprendre si elle est interrompue (par exemple par une mise hors tension). Pour cela l'état de la mise à jour en cours est écrit en mémoire flash avant et après chaque opération sensible.

Afin d'avoir toujours une version du code intacte, il faut trois emplacements mémoire (appelés slots) pour stocker le code :

— le slot primaire contient la version courante du code. C'est celle qui sera exécutée lors d'une séquence de boot standard.
— le slot secondaire contient la nouvelle version du code. C'est là que l'OS installe ses mises à jour
— le slot tertiaire contient la version précédente. Il sert à annuler une mise à jour en cours ou revenir à une version antérieure en cas de d'invalidité ou malfonctionnement de la version actuelle.

Pour plus de modularité, chaque slot et découpé en un nombre fixe de segments de code. Cela évite d'avoir à tout stocker d'un seul bloc en mémoire et permet des mises à jour partielles.

Les segments à mettre à jour sont installés par l'OS dans le slot secondaire. Ce dernier indique aussi au bootloader qu'il doit faire une mise à jour et les segments concernés. Le déroulement d'une mise à jour est alors le suivant :

1. Vérifier la signature du slot secondaire. Si elle est invalide, reprendre le boot standard.
2. Enregistrer l'état « backup ».
3. Copier tout le slot primaire dans le slot tertiaire (sauvegarde de la version courante).
4. Enregistrer l'état « install ».
5. Copier les segments pertinents du slot secondaire dans le slot primaire (mise à jour proprement dite).
6. Enregistrer l'état « testboot ».
7. Reprendre la séquence de boot standard. En mode testboot, l'OS doit confirmer qu'il fonctionne correctement. Au prochain boot, si cette confirmation a eu lieu, enregistrer l'état « no update »et terminer. Sinon poursuivre
8. Enregistrer l'état « revert »
9. Copier tout le slot tertiaire dans le slot primaire (retour à la version précédente)
10. Enregistrer l'état « no update ».

Le testboot permet de s'assurer de ne pas faire de mise à jour qui casserait l'OS et permet d'éviter de bloquer la puce avec un système invalide.

Grâce à l'enregistrement systématique de l'état, il est facile de reprendre une mise à jour interrompue. Si pendant une séquence de boot, la variable ne vaut pas « no update », vérifier les deux slots sensés être valides à cette étape puis reprendre juste après l'enregistrement de la variable d'état.

Les erreurs lors d'une mise à jour sont moins fatales. Si jamais une lecture ou une copie échoue, le programme essaye de garder une version valide dans le slot primaire, quitte à ignorer la mise à jour. Par exemple, s'il n'arrive pas à sauvegarder une copie de la version courante dans le slot tertiaire, il ne fera pas la mise à jour et lancera la version non mise à jour. Ainsi seul le slot tertiaire est potentiellement corrompu.

3 Algorithmes cryptographiques

La sécurité repose en grande partie sur les algorithmes cryptographiques utilisés par le programme pour chiffrer et authentifier les données sensibles. Une partie de mes recherches était donc consacrée au choix et comparaisons de divers algorithmes cryptographiques pour trouver les plus adaptés. Le besoin sécuritaire étant principal, il faut également prendre en compte les ressources en temps et mémoire nécessaires pour chaque algorithme car elles peuvent être assez limitées sur microprocesseur, surtout au premier niveau de boot.

3.1 AES et modes de chiffrement par blocs

L'algorithme de Rijindael de chiffrement symétrique est une référence depuis qu'il a remporté le concours AES en 2000. Par ailleurs il utilise relativement peu de mémoire et certains processeurs i.MX possèdent des accélérateurs cryptographiques implémentant cet algorithme en hardware. Il était donc un choix naturel pour toutes les opérations de chiffrement symétrique, c'est-à-dire les opérations internes du bootloader (chiffrement du code, des variables d'état stockées en mémoire et de sa configuration...).

L'algorithme AES travaille sur une matrice 4×4 dont les entrées sont des éléments du corps fini de Galois à 256 éléments (noté $GF(256)$). Les éléments de ce corps se représentent par des octets, AES travaille donc sur des blocs de 128 bits en tout. Il est composé d'un nombre variable de passes dans une boucle possédant plusieurs étapes :

— chaque entrée est substituée par une bijection de $GF(256) \to GF(256)$
— les lignes sont décalées circulairement (la première de 0, la deuxième de 1, ...)
— les colonnes sont mélangées par multiplication matricielle
— la partie de la clé étendue correspondant au numéro de la passe et rajoutée par un xor bit-à-bit à la matrice.

Ces opérations sont toutes bijectives et donc inversibles. Elles assurent par mélange des colonnes et décalage de ligne que tout le bloc participe au chiffrement de chaque octet. La fonction de substitution a été choisie pour assurer la non-linéarité et éviter les points fixes.

Le nombre de passes et donc la taille de la clé étendue dépendent de la taille de la clé initiale. Il y a 11 passes pour AES 128 et 15 pour AES 256.

AES ne permet cependant que de chiffrer ou déchiffrer des blocs de 128 bits. Pour chiffrer des données de taille différentes il faut utiliser un mode de chiffrement par blocs. J'ai regardé plusieurs modes de chiffrement et codes d'authentification de message pour choisir lequel figurerait dans le programme.

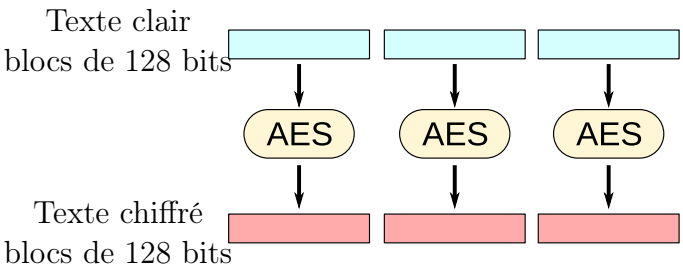

FIGURE 2 – chiffrement par bloc en mode EC

Le mode le plus naïf est l'EC (electronic codebook). Il consiste à découper l'entrée en bloc de 128 bits en rajoutant des bits nuls à la fin pour avoir une longueur multiple de 128, et chiffrer chacun séparément avec la clé. Cette méthode possède plusieurs vulnérabilités, notamment le fait que deux blocs identiques dans le clair le seront également dans le chiffré.

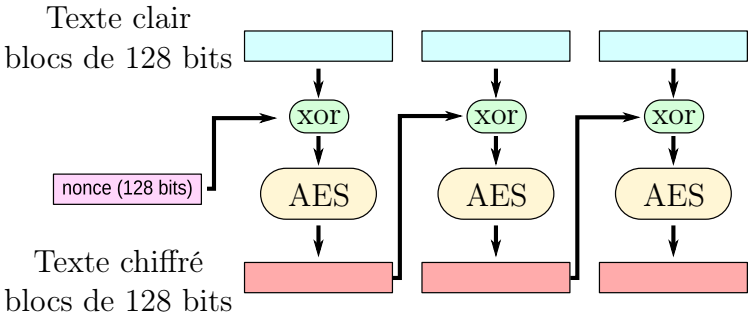

FIGURE 3 – chiffrement par bloc en mode CBC

Le mode CBC (cipher block chaining) travaille aussi sur des blocs de taille 128 bits. Il consiste à rajouter (par un xor bit à bit) au texte clair de chaque bloc le chiffré du bloc précédent, avant de le chiffrer. Là encore, tous les blocs AES utilisent la même clé. Ce mode n'a pas le désavantage d'EC : des blocs identiques donnent des chiffrés différents. Il nécessite un chiffrement séquentiel mais le déchiffrement peut être fait dans n'importe quel ordre. Néanmoins il présente quelques vulnérabilités : une erreur dans le chiffré d'un bloc corrompt celui-ci mais introduit une erreur dans le bit correspondant du bloc suivant. Cela peut permettre de changer des valeurs assez précisément si la structure des données est connue.

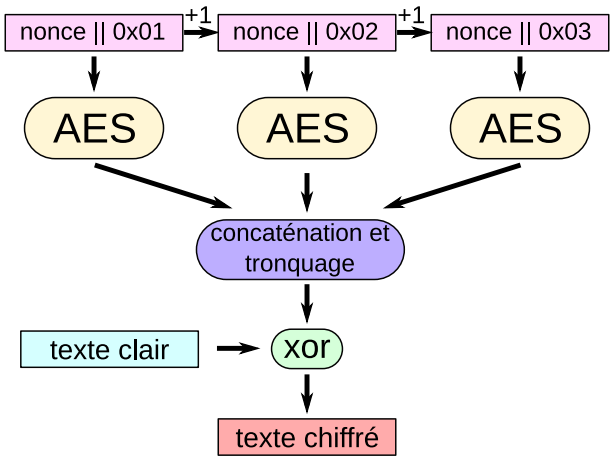

FIGURE 4 – chiffrement par bloc en mode CTR

Le mode CTR (counter) permet lui de chiffrer des données de tailles non multiples de 128. Il chiffre la valeur d'un nonce (nombre arbitraire à n'utiliser qu'une seule fois) concaténée à un compteur avec la clé, puis rajoute le résultat (par un xor bit-à-bit) au texte clair pour obtenir le chiffré. Là encore, le déchiffrage peut se faire dans n'importe quel ordre. Il est toutefois vulnérable à des modifications. Comme le chiffré est obtenu par simple xor, une inversion d'un bit du chiffré correspond à la même modification du texte clair. Cela peut permettre à un attaquant connaissant la structure des données de modifier une valeur en inversant certains bits.

3.2 Codes d'authentification de messages

AES et ses modes d'opération permettent de chiffrer des messages mais ne garantissent pas leur authenticité ou leur intégrité. Cela est particulièrement gênant pour le mode CTR où une attaque peut modifier le clair de manière ciblée, et pour le mode CBC où une modification corrompt un bloc et peut modifier de manière contrôlée le bloc suivant. Des codes détecteurs d'erreur peuvent permettre de vérifier l'intégrité d'un message en y rajoutant des bits de vérification. Cependant ils peuvent facilement être régénéré par un attaquant pour correspondre à un message modifié. Une solution est d'utiliser des codes d'authentification de messages (MAC en anglais), qui sont des codes détecteurs d'erreur impliquant une clé privé (qui peut-être la même, ou une dérivée de la clé de chiffrement).

Le code HMAC (keyed hash message authentification code) consiste à partir d'une fonction de hachage h et du message m de calculer

$$\mathrm{HMAC}_k(m) := h\big(k \oplus \mathrm{opad} \,||\, h(k \oplus \mathrm{ipad} \,||\, m)\big)$$

où k est la clé, $opad = 0x5c5c5c...$ et $ipad = 0x363636...$ sont deux constantes, $||$ représente la concaténation et $\oplus$ le xor bit-à-bit. Le double hash est utilisé pour éviter les attaques d'extension de longueur. L'avantage de ce MAC est qu'il est général et ne suppose pas un mode de chiffrement par blocs particulier. Le message peut avoir n'importe quelle longueur et être constitué d'une partie claire et d'une partie chiffrée réparties à souhait.

Le code CMAC ou CBC-MAC (block cipher based message authentication code) est un code conçu pour travailler avec le mode d'opérations CBC. Il consiste simplement à refaire une passe de CBC sur le chiffré et garder le dernier bloc comme bits de vérification. Il est important de rajouter la longueur du message (au début ou à le fin) dans les code d'authentification pour éviter des attaques par extensions. Ce mode à l'avantage de pouvoir réutiliser le code du CBC sans trop de modifications.

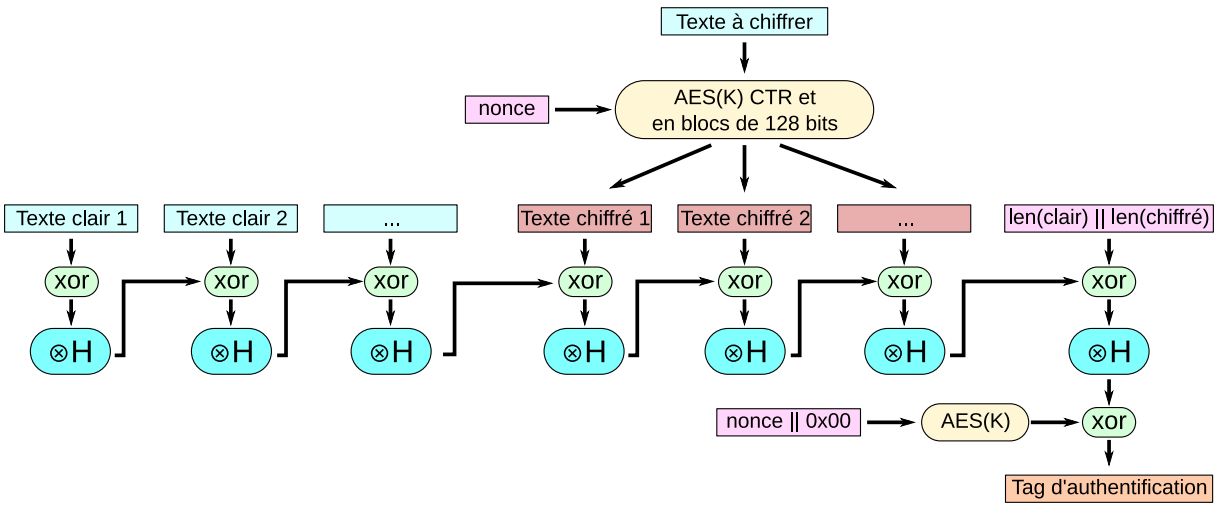

FIGURE 5 – Code d'authentification GCM avec une clé K

Le code GCM (Galois counter-mode) s'utilise avec le mode CTR. Il travaille dans le corps de Galois $GF(2^{128})$. Il utilise un deuxième clé H obtenue par le chiffrement d'un bloc nul avec la clé de chiffrement K. Le MAC est généré en découpant la sortie en blocs de 128 bits, en les multipliant par H dans le corps de Galois, puis en les ajoutant par xor

bit-à-bit au bloc suivant. Ce mode permet également de rajouter des données non chiffrées comme premiers blocs. Pour plus de sécurité, les longueurs des données (non-chiffrées et chiffrées) sont ajoutées en dernier bloc (deux valeurs 64 bits formant un bloc de 128), et le résultat final est le obtenu par un xor bit-à-bit entre le dernier bloc et chiffré du nonce, afin d'inclure ce dernier dans le code.

Pour notre programme, j'ai retenu un seul mode. Le mode CTR combiné au code GCM offre les garanties de sécurité nécessaires et permet d'être flexible sur la longueur des messages gérés. De plus, le chiffré et son tag d'authentification peuvent être calculés en même temps, ce qui permet de le faire par étapes, par exemple si les données sont trop grandes pour être chargées directement dans la RAM dû aux ressources limitées pendant le boot.

J'ai donc recodé en C les algorithmes AES-128 et AES-256 avec le mode d'opération CTR et le code d'authentification GCM. La double taille de clé permet de moduler l'utilisation de mémoire selon les besoins de sécurité. En pratique, les clés-racines du bootloader font 256 bits et les autre clés peuvent être plus petites.

3.3 Cryptographie sur les courbes elliptiques

Pour effectuer des mises à jour de manière sécurisée, il faut un moyen de communiquer de manière chiffrée et d'authentifier les données échangées entre le processeur et le fournisseur de mises à jour. Pour cela, il faut un mode de chiffrement asymétrique afin d'éviter d'avoir une clé AES privée partagée entre le processeur et le fournisseur. Le processeur a donc une liste de clés publiques correspondant aux fournisseurs qui peuvent lui envoyer des mises à jour, et un couple de clés publique/privé généré au premier boot. La clé publique est diffusée aux fournisseurs.

Le principe de la cryptographie sur courbe elliptique repose sur une loi de groupe sur ces courbes. Une courbe est elliptique si elle a une équation de la forme

$$y^2 = x^3 + ax + b$$

avec a et b des éléments fixé du corps de travail. En cryptographie, ce corps est souvent un corps premier $\mathbb{Z}/p\mathbb{Z}$ avec p grand (de l'ordre de 2^{256}).

Ces courbes sont symétriques en y. Pour définir un loi de groupe dessus nous prenons un point à l'infini comme élément neutre, puis nous définissons les symétriques comme étant opposés pour l'addition. Pour deux points non-opposés, s'ils sont différents alors la droite les reliant coupe la courbe en un unique troisième point. Son symétrique est défini comme la somme. Finalement s'il s'agit de deux fois le même point, nous utilisons la tangente à la courbe qui intersectera elle aussi la courbe en un unique point.

La cryptographie sur les courbes elliptique repose sur le problème de pseudo-logarithme de cette loi de groupe itérée : à partir d'un point G et de sa somme itérée $[n] \cdot G$, il est difficile de retrouver le nombre n. Une clé privée est donc un nombre $n \in [1, p-1]$, et la clé publique associée est le point $[n] \cdot G$ où G est un paramètre publique (tout comme p, a et b).

Lors d'une mise à jour, le bootloader a deux besoins de communication avec le fournisseur : vérifier que ce dernier à approuvé le code et obtenir la clé de chiffrement utilisée

pour déchiffrer le code. De ce fait il utilise deux algorithmes : ECDSA pour la signature et ECDH (échange de Diffie-Hellman sur les courbes elliptiques) pour générer un secret partagé donc la clé sera dérivée.

L'échange de Diffie-Hellman s'adapte très bien à cette loi de groupe. Pour deux couples de clé $m/[m] \cdot G$ et $n/[n] \cdot G$, chaque propriétaire d'une des clés privées peut facilement obtenir la valeur $[mn] \cdot G$ (dite secret partagé) à partir de sa clé et la clé publique de l'autre (par commutativité de l'opération addition itérée).

Par simplicité d'implémentation, nous travaillerons avec une seule courbe fixée à l'avance, ce qui permet quelques optimisations et simplifie le code.

4 Chaînes de confiance et gestion de clés

L'origine de nos chaînes de confiance est la mémoire OTP. Elle est écrite une unique fois au premier boot et contient les clés racines et certificats des fournisseurs de mises à jour. Pour des raison de place en mémoire OTP, les autres clés seront stockées en mémoire flash, chiffrées avec les clés racines de la mémoire OTP, et seuls les hashs des certificats seront stockés en mémoire OTP, ainsi qu'une section non-écrite permettant de marquer d'éventuels certificats révoqués.

Les clés racines sont propre à chaque puce. Elles sont générées aléatoirement au premier boot.

4.1 La chaîne interne

La première chaîne sert à sécuriser les informations internes du bootloader. Notamment, elle sécurise la configuration (ensemble de réglages saisis lors du premier boot) et les variables d'état (permettant de reprendre une exécution de mises à jour si elle a été interrompue). Cette chaîne sert aussi à authentifier le numéro de série de la mémoire flash. Ce dernier est stocké sur une partition de la mémoire flash et est authentifié (indirectement) par la clé racine (stockée en mémoire OTP).

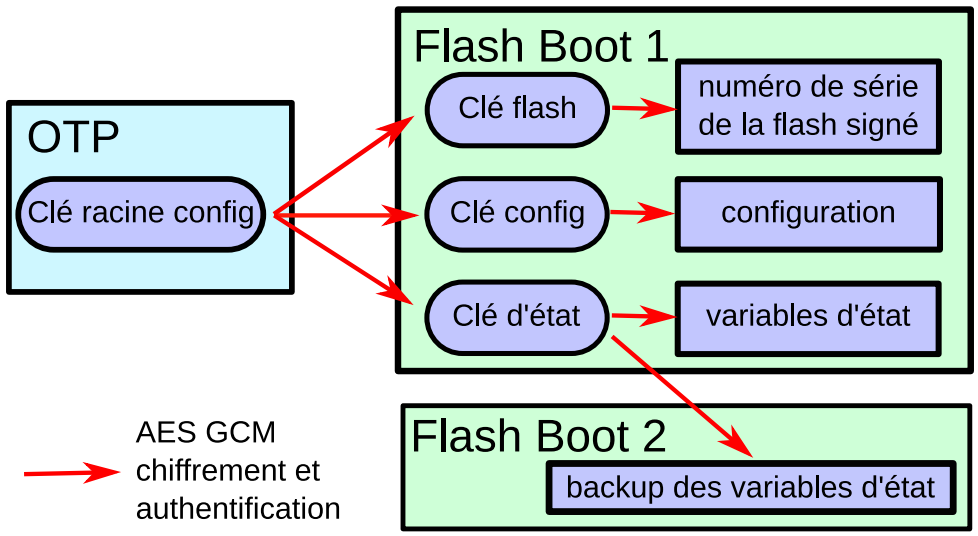

FIGURE 6 – La chaîne de clé interne

Pour plus de sécurité, chaque fonction a sa propre clé, qui est chiffrée par la clé racine. Les variables d'état pouvant changer, elles sont sauvegardées en double sur les deux

partitions boot de la mémoire flash. Cela permet de s'assurer qu'au moins un des deux emplacements contient une copie valide des variables d'état, même en cas d'interruption pendant l'écriture d'un des deux emplacements.

4.2 La chaîne du code

Une deuxième chaîne sert à vérifier le code dans les slots primaires et tertiaires. Ces deux slots n'étant touchés que par le bootloader, ils sont chiffrés et signés par des clés internes à la puce (générées aléatoirement au premier boot). Chaque segment est chiffré par sa propre clé, appelée clé de code. Toutes les clés de code sont stockées en mémoire flash, chiffrées et signées par la clé code racine.

Chaque clé de code étant utilisée deux fois (pour le segment correspondant des slots primaires et tertiaires), il faut utiliser des nonces différents pour éviter des fuites de clés. Ces nonces n'ont pas besoin d'être secret et peuvent être stockés en clair dans l'entête des segments.

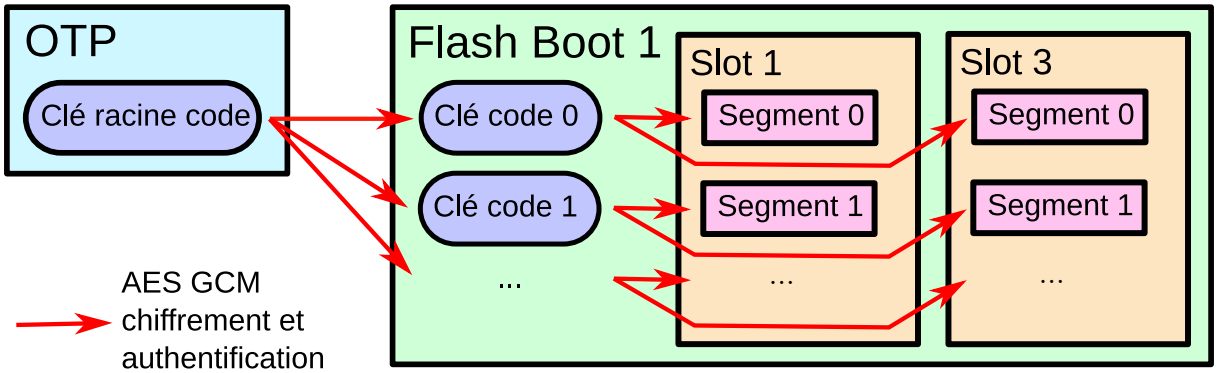

FIGURE 7 – La chaîne de clés du code

4.3 La chaîne des mises à jour

La validation d'une mise à jour est plus complexe. Chaque segment de la mise à jour est validé individuellement. Un segment est composé des informations de signature, d'un entête clair et du code chiffré. L'entête est la partie non-chiffrée mais signée et authentifiée du code. Il contient notamment le numéro de version du code.

Les informations de signature contiennent deux certificats (éventuellement identiques) : celui du fournisseur qui a chiffré le code (clé publique IES) et celui du fournisseur qui l'a signé (clé de signature). Il contient également les signatures ECDSA et AES GCM, et, de manière chiffrée, la clé de chiffrement utilisée pour chiffrer le code.

La puce connaît sa clé privée et la clé publique du fournisseur, et le fournisseur connaît sa clé privée et la clé publique de la puce. Par un échange de Diffie-Hellman sur les courbes elliptiques (ECDH), ils peuvent tous les deux obtenir un secret partagé. Ce secret sert a chiffrer la clé de chiffrement, et assure donc la confidentialité du code même s'il est transmis de manière publique.

La validation d'un segment et la mise à jour procède donc comme suit :

— lire les certificats des fournisseurs autorisés (qui ont été inscrits dans la mémoire flash au premier boot), les comparer aux hashs stockés en mémoire OTP et ne garder que ceux qui correspondent et n'ont pas été révoqués.

— vérifier que les deux certificats (de chiffrement et de signature) du segment font partie de ces certificats.
— Vérifier la signature ECDSA avec le certificat de signature.
— Obtenir le secret partagé entre la puce et le fournisseur avec la clé privée de la puce et la clé publique du certificat de chiffrement (clé IES). En déduire la clé partagée.
— Déchiffrer la clé de chiffrement et s'en servir pour déchiffrer et valider le code.
— Chiffrer le segment avec la clé de code correspondante et le copier dans le slot primaire

L'utilisation d'une clé de chiffrement plutôt que directement la clé partagée permet un peu plus de sécurité en rajoutant un niveau d'abstraction.

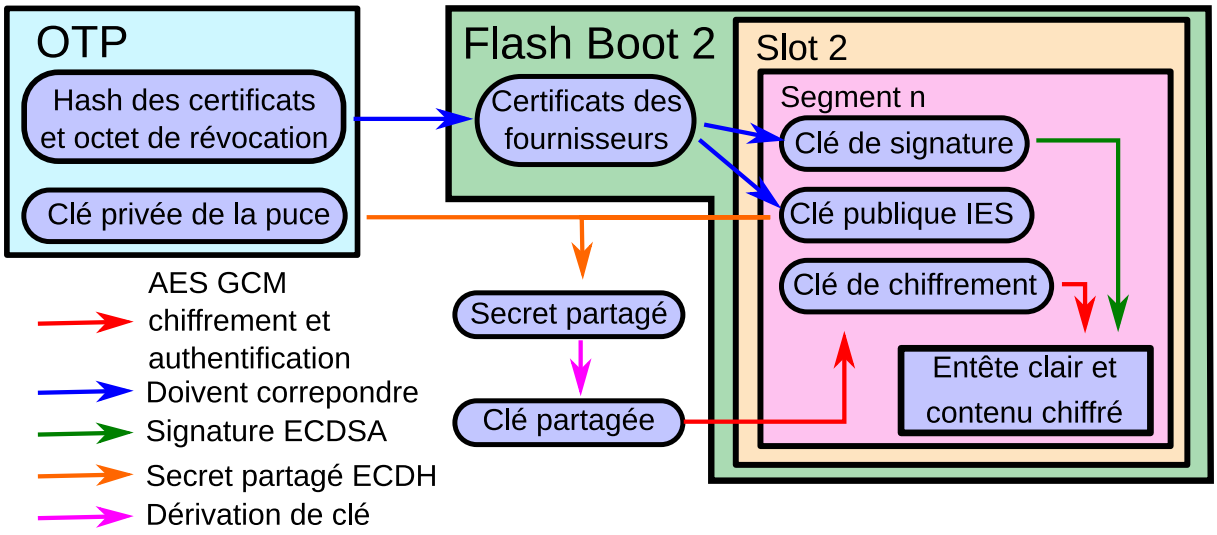

FIGURE 8 – La chaîne de confiance lors d'une mise à jour

5 Résultats d'implémentation

Afin de simuler sur PC l'exécution sur puce, j'ai créé un système de lecture de fichiers ayant l'API du pilote de la mémoire flash pour la lecture de mémoire. Le programme n'étant pas un vrai bootloader, il se contente d'afficher une trace de ses résultats de test et des branches qu'il parcourt. Cela permet de tester ses réactions aux différentes erreurs qui peuvent se produire pendant l'exécution. Pour une mise à jour, il affiche aussi les copies faites, les adresses mémoire concernées et les clés de fournisseur utilisées.

La simulation d'erreur pendant l'exécution peut se faire de plusieurs façons : en écrivant des données invalides dans les fichiers représentant la mémoire flash pour les erreurs de vérification, ou en bloquant la lecture/écriture dans certaines zones mémoires dans le faux pilote de la mémoire flash. Ces options permettent de tester le programme de manière assez approfondie pour vérifier qu'il exhibe bien le comportement souhaité dans chaque cas.

```
main start
main - read OTP and checked serial
main - check flash
main - read and checked config
main - read and checked status - compared monotonic counters
main - read and checked code keys
main - read os_info
main - checked update status
  Checking slot 1, decrypting and loading to RAM
main - checked and deciphered code
```

TABLE 1 – Trace d'une séquence de boot standard. Le bootloader vérifie le numéro de série de la puce et de la mémoire flash, puis lit la configuration, les variables d'état, les clés de code, les informations provenant de l'OS (indiquant s'il y a des mise à jour à faire), puis vérifie et déchiffre le code

```
main start
main - read OTP and checked serial
main - check flash
main - read and checked config
FATAL ERROR - exiting with code 0x00000515
FatalError - wiped data_sensitive: 0x426960 -> 0x426a98
FatalError - wiped bss_sensitive: 0x426960 -> 0x426960
This error can lead to reset and retry
```

TABLE 2 – Trace d'une séquence de boot standard où la configuration du bootloader n'est pas valide. Cela conduit à une erreur fatale

```
main start
main - read OTP and checked serial
main - check flash
main - read and checked config
main - read and checked status - compared monotonic counters
main - read and checked code keys
main - read os_info
main - read 11 valid certificates (out of 11)
  update_init start
    Checking slot 2, No decryption
    Checking slot 1, No decryption
  update_init - code validation for slots 1 & 2 passed
  update_backup
  Copying slot 1 to slot 3
    segment 0: no decrypt/reencrypt (0x00000010@boot_1 -> 0x00000050@boot_1)
    segment 1: no decrypt/reencrypt (0x00000020@boot_1 -> 0x00000060@boot_1)
    segment 2: no decrypt/reencrypt (0x00000030@boot_1 -> 0x00000070@boot_1)
  Checking slot 3, No decryption
  update_install
  Copying slot 2 to slot 1
    segment 0: reencryt with local keys (0x00000010@boot_2 -> 0x00000010@boot_1)
      ECDSA key : 0x03ad8f5bb26973543c461df9ede4fa642f14f8499b945a1ad6b7fdf9da36b30140
      ECDH key  : 0x03ad8f5bb26973543c461df9ede4fa642f14f8499b945a1ad6b7fdf9da36b30140
    segment 1: reencryt with local keys (0x00000020@boot_2 -> 0x00000020@boot_1)
      ECDSA key : 0x03ad8f5bb26973543c461df9ede4fa642f14f8499b945a1ad6b7fdf9da36b30140
      ECDH key  : 0x03ad8f5bb26973543c461df9ede4fa642f14f8499b945a1ad6b7fdf9da36b30140
    segment 2: reencryt with local keys (0x00000030@boot_2 -> 0x00000030@boot_1)
      ECDSA key : 0x03ad8f5bb26973543c461df9ede4fa642f14f8499b945a1ad6b7fdf9da36b30140
      ECDH key  : 0x03ad8f5bb26973543c461df9ede4fa642f14f8499b945a1ad6b7fdf9da36b30140
  Checking slot 1, No decryption
  update_install - update successful - entering test boot
main - update completed successfully
main - checked update status
main - rewrote os_info to flash
  Checking slot 1, decrypting and loading to RAM
main - checked and deciphered code
```

TABLE 3 – Trace d'une séquence de boot avec mise à jour. Elle commence comme la standard, puis lit les certificats, vérifie la mise à jour et le code courant, fait un backup, installe la mise à jour en indiquant les clés utilisées pour l'authentifier, et reprend le boot standard

```
main start
main - read OTP and checked serial
main - check flash
main - read and checked config
main - read and checked status - compared monotonic counters
main - read and checked code keys
main - read os_info
main - read 11 valid certificates (out of 11)
  update_init start
  Checking slot 2, No decryption
  Checking slot 1, No decryption
  update_init - code validation for slots 1 & 2 passed
  update_backup
  Copying slot 1 to slot 3
    segment 0: no decrypt/reencrypt (0x00000010@boot_1 -> 0x00000050@boot_1)
    segment 1: no decrypt/reencrypt (0x00000020@boot_1 -> 0x00000060@boot_1)
    segment 2: no decrypt/reencrypt (0x00000030@boot_1 -> 0x00000070@boot_1)
  Checking slot 3, No decryption
  update_install
  Copying slot 2 to slot 1
    segment 0: reencryt with local keys (0x00000010@boot_2 -> 0x00000010@boot_1)
      ECDSA key : 0x0223a1f61a4ed17e27c3679d7fe70d6ca1a2b48935117da2ca67080dfc13f2eba6
      ECDH key  : 0x0223a1f61a4ed17e27c3679d7fe70d6ca1a2b48935117da2ca67080dfc13f2eba6
    segment 1: reencryt with local keys (0x00000020@boot_2 -> 0x00000020@boot_1)
      ECDSA key : 0x0223a1f61a4ed17e27c3679d7fe70d6ca1a2b48935117da2ca67080dfc13f2eba6
      ECDH key  : 0x0223a1f61a4ed17e27c3679d7fe70d6ca1a2b48935117da2ca67080dfc13f2eba6
# write interruption on 0x00000020@boot_1
    segment copy failed - erasing segment
  update_install - copy failed (code 0x00000801) - reverting
  update_revert
  Copying slot 3 to slot 1
    segment 0: no decrypt/reencrypt (0x00000050@boot_1 -> 0x00000010@boot_1)
    segment 1: no decrypt/reencrypt (0x00000060@boot_1 -> 0x00000020@boot_1)
# write interruption on 0x00000020@boot_1
    segment copy failed - erasing segment
  update_revert - copy failed
main - update not completed (error or simple testboot check)
main - checked update status
main - rewrote os_info to flash
  Checking slot 1, decrypting and loading to RAM
FATAL ERROR - exiting with code 0x00000602
FatalError - wiped data_sensitive: 0x4269a0 -> 0x426ad8
FatalError - wiped bss_sensitive: 0x4269a0 -> 0x4269a0
This error can lead to reset and retry
```

TABLE 4 – Trace d'une séquence de boot avec mise à jour où le bootloader ne peut pas écrire sur le deuxième segment du slot primaire. Cela corrompt le slot primaire et mène à l'échec total (le prochain boot retentera de copier le slot tertiaire intact dans le slot primaire)

6 Conclusion

Ce stage était ma première expérience d'un tel projet informatique impliquant recherche, établissement du cahier des charges et implémentation. Cela m'a appris à gérer un projet de tel envergure, notamment le besoin de bien documenter le code, l'évolution du cahier des charges au fur et à mesure de l'implémentation, et la gestion propre des erreurs pour le débogage. Le programme final comprenait plus de 11 000 lignes de code.

Ce travail a fourni un boot securisé propriétaire sur processeur i.MX de NXP. De plus, le fait d'avoir tout recodé au lieu d'utiliser des librairies déjà existantes le rend assez facilement modulable selon les besoins des clients.

Malheureusement, le programme n'a pas pu être porté sur processeurs i.MX faute de temps et à cause des complications liées au travail à distance.

Références

[1] Mehmet Adalier and Antara Teknik. Efficient and secure elliptic curve cryptography implementation of curve p-256. *NIST Computer Security Resource Center*, June 2015.

[2] Morris Dworkin. Recommendation for Block Cipher Modes of Operation : Galois/-Counter Mode (GCM) and GMAC. *NIST Special Publication 800-38D*, November 2007.

[3] JEDEC. Embedded MultiMediaCard (e•MMC) Card Product Standard. *JESD84-A441*, March 2010.

[4] NIST. Specifications for the Secure Hash Standard. *Federal Information Processing Standards Publication (FIPS) 180-4*, August 2015.

[5] NIST. Specification for the Advanced Encryption Standard (AES). *Federal Information Processing Standards Publication (FIPS) 197*, November 26, 2001.

[6] NXP. i.MX 6 Linux High Assurance Boot (HAB) User's Guide. *Document Number : IMX6HABUG*, January 2013.

[7] NXP. Secure Boot on i.MX 50, i.MX 53, i.MX 6 and i.MX 7 Series using HABv4. *Document Number : AN4581*, May 2018.