ALGORITHMES DE SÉCURITÉ

pour les systèmes embarqués

Andrew O'Shei
andrewoshei@gmail.com

Licence 3 Informatique, Micro-Informatique et Machines Embarqués
Université Paris 8, Vincennes Saint-Denis

Table des matières

Introduction....1
L'évolution des systèmes embarqués modernes....1
Sélection d'algorithmes pour les systèmes embarqués....2
Algorithmes de hachage....3
Comment fonctionne un algorithme de hachage ?....3
Évaluation de la sécurité des algorithmes de hachage....4
Algorithmes de hachage sécurisés, mots de passe et signatures....5
Algorithmes cryptographiques....7
Chiffrement de bloc....7
Chiffrement de flux....8
Générateurs de nombres aléatoires....9
Registre à décalage de rétroaction linéaire (LFSR)....10
Algorithmes d'aléa modernes....10
Conclusion....11

Introduction :

La conception de systèmes embarqués a toujours présenté un ensemble unique de défis pour les ingénieurs. La puissance de calcul et les fonctionnalités d'un système embarqué doivent toujours être équilibrées avec l'efficacité énergétique, les contraintes de taille, la durabilité physique, la rentabilité et l'autonomie du système. Tous les systèmes embarqués nécessitent des compromis dans leur conception afin de satisfaire des objectifs particuliers. Cela signifie que des considérations spéciales doivent être prises lors de l'utilisation de protocoles de sécurité sur un système embarqué. Dans un monde idéal, nous préférerions que le niveau maximum de sécurité soit utilisé comme pratique standard. Cependant, en raison des contraintes de ressources des systèmes embarqués, nous constatons généralement qu'en utilisant le maximum de sécurité possible, nous ne parviendrons pas à atteindre les objectifs fixés pour un projet donné.

L'évolution des systèmes embarqués modernes :

Les systèmes embarqués sont apparus pour la première fois au début des années 1970 avec l'avènement du microprocesseur et du microcontrôleur, avec le premier microprocesseur connu développé en secret de 1968 à 1970 pour le Central Air Data Computer du F-14 Tomcat[1] (L'avion de chasse américain présenté dans le film *Top Gun*, 1986). Depuis les années 1970, des progrès incroyables dans la technologie des semi-conducteurs nous ont apporté des applications plus puissantes et plus omniprésentes de la technologie embarquée. Aujourd'hui, presque tous les humains du monde industrialisé se promènent avec l'application la plus réussie de la technologie embarquée dans leur poche, le smartphone. Si les progrès des systèmes embarqués ont permis des solutions innovantes et de nouveaux produits remarquables, son succès s'accompagne d'un souci toujours croissant de sécurité. Comme nous nous appuyons de plus en plus sur les technologies embarquées, un appareil compromis a le potentiel de causer beaucoup plus de dommages. Malheureusement, alors que les systèmes embarqués passaient des applications militaires et industrielles aux produits de consommation fabriqués en masse, de nombreux fabricants ont négligé les problèmes de sécurité en raison du coût accru que cela ajouterait au développement[2].

Heureusement, avec les progrès de la puissance et de l'efficacité des composants à bas prix au cours de la dernière décennie, il n'est plus une évidence que nous devons sacrifier les performances pour avoir une bonne sécurité. De plus, les optimisations des algorithmes de sécurité modernes permettent d'augmenter le niveau de sécurité avec un impact assez linéaire sur les performances[3]. Cela signifie que si le niveau de sécurité requis est déterminé tôt dans le processus de conception, les ingénieurs seront en mesure de sélectionner plus facilement des composants qui satisferont aux exigences de performance souhaitées. À son tour, le produit final atteindra tous les objectifs du projet sans compromettre la sécurité.

Bien que les systèmes embarqués soient confrontés à une variété de menaces de sécurité, y compris des attaques physiques, des analyses d'injection de fautes, des attaques de canaux auxiliaires, des logiciels malveillants et des virus. Je me concentrerai spécifiquement sur l'état des algorithmes de sécurité modernes et leurs implémentations sur les systèmes embarqués. La plupart des algorithmes de sécurité sont orientés vers le contrôle d'accès au niveau logiciel et le cryptage des données, mais

en raison de la portabilité des systèmes embarqués, ils sont confrontés à un risque élevé d'attaque physique. Certains des algorithmes dont je vais parler, tels que les générateurs de nombres aléatoires, peuvent également être utilisés pour se défendre contre l'injection de fautes et les attaques de canaux auxiliaires. J'espère que cet article pourra servir de guide pour prendre des décisions de sécurité lors de la phase de conception d'un projet de systèmes embarqués.

Sélection d'algorithmes pour les systèmes embarqués :

L'étape la plus importante de la conception d'un système embarqué consiste à déterminer le niveau de sécurité requis. Cependant, des précautions doivent être prises lors de l'évaluation des besoins en matière de sécurité. Avec les menaces nouvelles et évolutives telles que les botnets distribués[4], tout système pouvant accéder à Internet devrait disposer d'un contrôle d'accès bien sécurisé, sans exception. Cela étant dit, pour certaines applications, une sécurité renforcée peut ne pas être nécessaire et des ressources peuvent être conservées pour d'autres processus. Le guide suivant est un bon point de départ pour évaluer les besoins de sécurité d'une application intégrée.

Note: la sécurité évolue constamment, cette liste n'est donc qu'une directive générale. Les besoins de votre application spécifique doivent être évalués en fonction de toute menace potentielle qui pourrait émerger tout au long de la durée de vie du projet. Il vaut mieux faire preuve de plus de prudence plutôt que de se rendre compte qu'un système est critique seulement après avoir été compromis. Cependant, en général, la plupart des applications embarquées ne seront concernées que par la sécurité jusqu'au niveau 3 ou 4.

Niveau 0 - **Pas de sécurité**: Les systèmes hors réseau qui ne stockent ni ne transmettent des données critiques.

Niveau 1 - **Sécurité de bas niveau**: Le hachage et données en texte brut (MD5 ou SHA-2). Pour les applications et les systèmes en réseau fournissant des informations accessibles au public. Cela inclut les systèmes où l'écoute clandestine n'est pas un problème mais l'intégrité des données est nécessaire.

Niveau 2 - **Sécurité faible-moyenne**: Le hachage et l'authentification, généralement utilisés pour les systèmes de contrôle industriels et les communications réseau d'entreprise.

Niveau 3 - **Sécurité de niveau moyen**: Cryptographie avec authentification et petites tailles de clés (AES et 3DES). Applications traitant des informations générales de l'entreprise ou du commerce et du contrôle industriel dans des environnements non contrôlés.

Niveau 4 - **Sécurité moyenne-élevée**: SSL avec des tailles de clé moyennes, RSA, AES et VPN. Applications traitant des transactions de commerce électronique, des informations importantes de l'entreprise et de la surveillance des données critiques.

Niveau 5 - **Sécurité de haut niveau**: SSL avec des clés de taille maximale et des VPN. Applications traitant des informations financières importantes, des communications militaires non critiques, des données médicales et des informations d'entreprise critiques.

Niveau 6 - **Niveau de sécurité critique**: L'isolation physique des systèmes « air-gapped », cryptage maximal avec des clés de taille maximale, systèmes de communication dédiés et « One-Time Pads ». Utilisé pour sécuriser les codes de lancement nucléaire, les clés privées de l'autorité de certification racine et les données militaires critiques.

Source: Cette liste a été adaptée de *Cryptography for embedded systems – Part 1: Security level categories and hashing*, Timothy Stapko Juillet 2010. https://www.eetimes.com/cryptography-for-embedded-systems-part-1-security-level-categories-hashing/

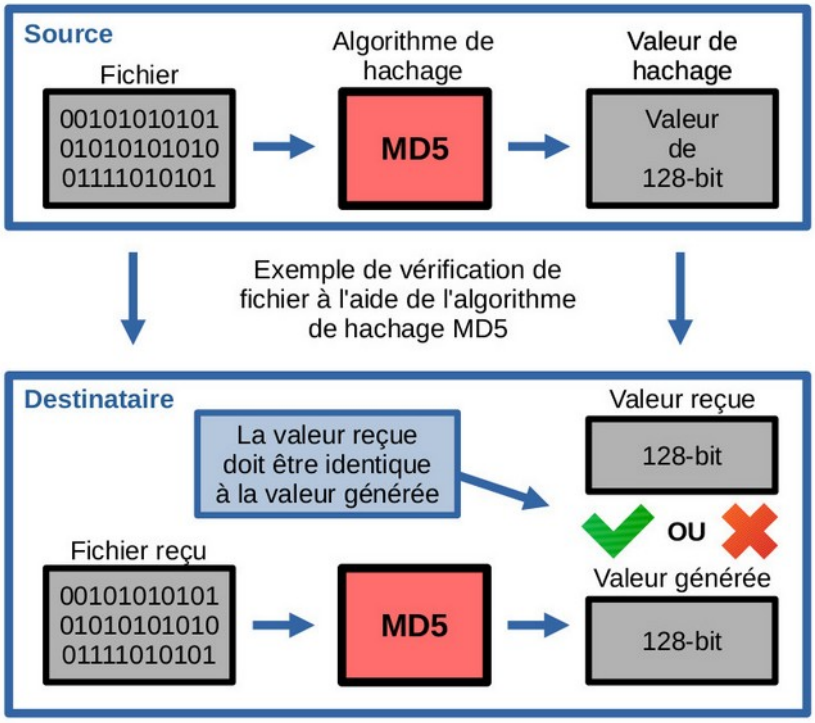
Algorithmes de hachage :

Les algorithmes de hachage, tels que MD5, SHA-1, SHA-256, SHA-3 et BLAKE2 sont utilisés dans diverses applications. Ils constituent un outil important pour sécuriser les mots de passe, la vérification de l'intégrité, l'authentification des messages, le cryptage à clé publique et les signatures numériques. Les algorithmes de hachage jouent un rôle essentiel dans toutes les technologies modernes d'Internet et des communications. Les algorithmes de hachage sont intégrés directement dans une variété de protocoles, y compris mais certainement pas limité à HTTPS, SSH et Bitcoin. Ainsi, que vous vous connectiez à une machine distante, que vous envoyiez des e-mails ou que vous envoyiez des messages sur votre téléphone mobile, une fonction de hachage est impliquée.

Comme indiqué dans les applications de niveau 1, ou applications à faible sécurité, la principale préoccupation est la validation de l'intégrité des données et des fichiers. C'est-à-dire que dans une application donnée, nous ne sommes pas concernés si un attaquant est capable de lire nos données mais nous tenons à nous assurer que les données ne sont pas falsifiées. Pour cela, il est le plus courant d'utiliser un algorithme de hachage. Les algorithmes de hachage, en général, ont l'avantage d'être peu coûteux en termes de calcul. Les deux algorithmes de hachage les plus couramment utilisés sont MD5 et SHA-1, cependant, il est fortement conseillé d'envisager des algorithmes de hachage plus avancés dans les applications modernes. **SHA-1 est maintenant considéré comme complètement cassé et ne doit plus être utilisé dans aucune application[5]**. MD5 doit être utilisé avec prudence car il présente des vulnérabilités connues, mais MD5 est toujours généralement considéré comme sûr pour des cas d'utilisation non critiques spécifiques, tels que la validation de l'intégrité d'un fichier téléchargé. **N'utilisez pas le MD5 standard pour stocker les mots de passe**[6]. Je discuterai des algorithmes de stockage des mots de passe à la fin de cette section.

Comment fonctionne un algorithme de hachage ? :

Tous les algorithmes de hachage fonctionnent essentiellement en prenant un fichier d'entrée ou un message de taille arbitraire, puis génèrent une valeur de hachage de taille fixe en sortie. Le processus peut ensuite être inversé afin de vérifier que le fichier ou le message reçu est bien le même que celui qui a généré la valeur de hachage au départ. Le concept général est que chaque fichier unique génère une valeur de hachage unique. Toute modification du fichier d'origine générera à son tour une nouvelle valeur de hachage unique. Ainsi, si la valeur de hachage générée par un fichier reçu ne correspond pas à la valeur de hachage fournie, nous pouvons supposer que le fichier a été compromis.



Évaluation de la sécurité des algorithmes de hachage :

L'élément le plus critique d'un algorithme de hachage est qu'il s'agit d'une opération à sens unique. C'est-à-dire que pour qu'un algorithme de hachage soit considéré comme sécurisé, il ne doit pas être possible de dériver le fichier ou le message d'origine à partir de sa valeur de hachage générée. C'est également un point clé pour comprendre la différence entre les algorithmes de hachage et la cryptographie. Cette caractéristique des algorithmes de hachage est appelée résistance de préimage. La résistance est une autre façon de décrire la complexité mathématique, le temps de calcul et les ressources nécessaires pour réussir une attaque (Plus de ressources requises = plus de résistance). La résistance à la préimage indique essentiellement qu'un attaquant, étant donné une valeur de hachage ***H***, sera incapable de trouver un message ou un fichier ***M***, de sorte que **Hash**(***M***) = ***H [7]***. Compte tenu de l'exemple de l'algorithme de hachage MD5, il devrait être clair pourquoi cela est important. Si un attaquant parvient à trouver des messages ou des fichiers arbitraires qui génèrent le même hachage, il peut remplacer le fichier d'origine par un fichier arbitraire. Ceci est particulièrement problématique pour les systèmes embarqués où des algorithmes de hachage sont souvent utilisés pour valider les mises à jour du firmware. Le chargement d'une mise à jour de micrologiciel compromise, même si ce ne sont que des données aléatoires, pourrait rendre un système intégré inopérant.

Un autre problème avec les algorithmes de hachage est la résistance aux collisions et la résistance à la seconde pré-image. La résistance aux collisions est le degré auquel l'algorithme de hachage résiste à la capacité d'un attaquant à trouver deux messages ou fichiers arbitraires qui donnent la même valeur de hachage. Bien que la résistance aux collisions soit importante, en particulier dans les algorithmes de hachage qui traitent de la sécurité et de l'authentification des mots de passe, elle n'est pas essentielle pour toutes les applications. La résistance aux collisions de MD5 a été

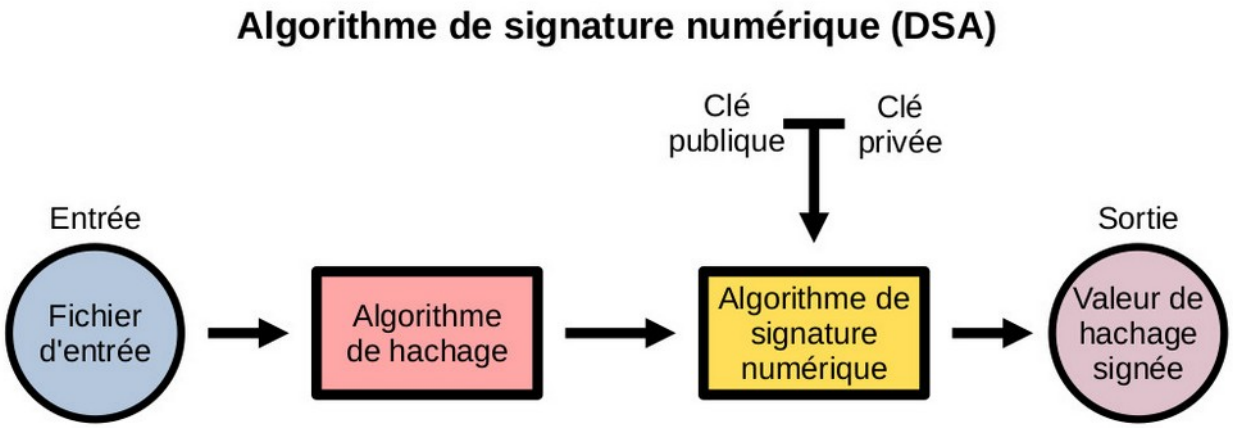
clairement rompue depuis au moins 2005[8], avec des vulnérabilités connues dès 1996[9], mais MD5 est encore largement utilisé aujourd'hui (principalement pour vérifier les fichiers téléchargés). La résistance à la seconde préimage indique qu'un attaquant, étant donné un message $\boldsymbol{M^1}$, ne devrait pas être en mesure de trouver un message $\boldsymbol{M^2}$, où $\boldsymbol{M^1 \neq M^2}$ et $\mathbf{Hash}(\boldsymbol{M^1}) = \mathbf{Hash}(\boldsymbol{M^2})$ [7]. Bien que similaire à la résistance aux collisions, la distinction avec la résistance à la seconde pré-image est que l'attaquant a un fichier cible connu ou un message qu'il souhaite falsifier.

À ce stade, vous vous demandez peut-être à quoi servent les algorithmes de hachage si deux fichiers peuvent avoir la même valeur de hachage ? Cela se résume aux mathématiques, une valeur de hachage a une taille fixe. Étant donné la taille fixe $\boldsymbol{N}$ d'une valeur de hachage, il ne peut y avoir que $\mathbf{2}^{\boldsymbol{N}}$ permutations possibles en sortie d'un algorithme de hachage. Étant donné que pour qu'un algorithme de hachage soit utile, il doit accepter des messages de n'importe quelle combinaison et longueur, il y a beaucoup plus d'entrées possibles dans un algorithme de hachage que de sorties. Donc, mathématiquement, il est inévitable que certaines combinaisons distinctes de messages d'entrée produiront la même sortie. Le but devient alors de créer des algorithmes de hachage qui nécessitent un tel niveau de ressources et de temps pour se briser qu'il devient impossible dans un sens pratique. Ou en d'autres termes, à un certain moment, le coût de la rupture de l'algorithme devient si élevé qu'il l'emporte sur tout avantage de le casser en premier lieu. Une chose à garder à l'esprit est que, historiquement, la puissance de calcul augmente à mesure que son coût diminue avec le temps. Ainsi, des attaques qui peuvent être irréalistes aujourd'hui pourraient en théorie être effectuées à l'avenir sur le matériel le plus élémentaire. C'est déjà la réalité avec certaines attaques sur des algorithmes tels que SHA-1 et MD5[8].

Algorithmes de hachage sécurisés, mots de passe et signatures :

Comme je l'ai brièvement mentionné précédemment, il existe aujourd'hui des algorithmes de hachage plus sécurisés que MD5. Deux des principales fonctions des algorithmes de hachage sur Internet sont la création de signatures numériques et la protection des mots de passe des utilisateurs pour les sites Web. Bien qu'en principe, ces nouveaux algorithmes fonctionnent de manière similaire dont j'ai déjà discuté pour certains cas d'utilisation, une complexité supplémentaire est ajoutée pour améliorer la sécurité des algorithmes. Les signatures numériques, introduites pour la première fois avec l'algorithme de chiffrement RSA publié en 1977[7], ajoutent un algorithme supplémentaire à la sortie d'une fonction de hachage. Les valeurs de hachage signées sont la méthode la plus courante pour garantir que seul un code valide est exécuté sur un système embarqué. Une fois signée, une valeur de hachage est combinée avec une clé privée aléatoire. Non seulement cela garantit que tous les messages produisent une sortie unique, mais il protège également contre la non-répudiation, si un signataire de message prétend à tort ne pas avoir signé le message. Un algorithme couramment utilisé pour les signatures numériques est EdDSA (Edwards-curve Digital Signature Algorithm). EdDSA est une évolution de l'algorithme DSA créé par le NIST (National Institute of Standards and Technology) en 1991[10]. Une amélioration notable de l'algorithme EdDSA est qu'il ne repose plus sur un générateur de nombres aléatoires, comme c'était le cas avec le DSA d'origine. Cette amélioration rend l'algorithme EdDSA assez robuste contre les attaques par canal latéral qui affligent les systèmes embarqués[11]. DSA et EdDSA reposent tous deux sur une méthode de clé

publique / privée qui sera discutée plus tard car elle est utilisée dans de nombreux algorithmes de chiffrement.



Note: La clé publique est envoyée au destinataire afin de valider la signature numérique.

Comme vous devez le savoir, stocker les mots de passe en texte simple, que ce soit dans une base de données ou sur un appareil, est une très mauvaise idée. Les mots de passe en texte simple seront découverts puis exploités. Une bien meilleure approche consiste à stocker un hachage du mot de passe, puis lorsqu'un utilisateur se connecte, vous vérifiez ses informations d'identification avec la valeur de hachage. De cette façon, en cas de violation de données ou d'appareil volé, un attaquant peut trouver la valeur de hachage mais il sera impossible (espérons-le) de déduire le mot de passe d'origine. Comme je l'ai mentionné précédemment, l'algorithme MD5 standard est insuffisant pour stocker les mots de passe. C'est en grande partie parce qu'il existe une relation un à un entre les mots de passe en texte simple et les valeurs de hachage qu'ils génèrent. Ainsi, le même mot de passe utilisé par deux utilisateurs différents produira la même valeur de hachage[12]. Tout comme l'algorithme DSA, les algorithmes de hachage modernes pour le stockage des mots de passe reposent sur le caractère aléatoire appliqué. La méthode utilisée pour le hachage de mot de passe est appelée SALT (Salted Secure Hash Algorithm). Avant de hacher un mot de passe, SALT ajoute une valeur générée aléatoirement au mot de passe[6]. Cela garantit que deux utilisateurs qui insistent pour utiliser le mot de passe «pasword123» auront toujours des valeurs de hachage uniques dans une base de données. Tout cela étant dit, salting un mot de passe ne protégera jamais un utilisateur s'il a choisi un mauvais mot de passe. Il est néanmoins important de garder cela à l'esprit lors de la conception du contrôle d'accès pour un système embarqué. Si un mot de passe est brisé, un attaquant peut être en mesure d'accéder librement à des données cryptées sensibles ou pire.

SALT en pratique : https://en.wikipedia.org/wiki/Salt_(cryptography)#Unix_implementations

USERNAME	PASSWORD	SALT VALUE	STRING TO HASH	HASH VALUE (SHA-256)
User1	**password123**	E1F53135E559C253	**password123**E1F53135E559C253	72AE25495A7981C40622D49F9A52E4F1565C90F048F59027BD9C8C8900D5C3D8
User2	**password123**	84B03D034B409D4E	**password123**84B03D034B409D4E	B4B6603ABC670967E99C7E7F1389E40CD16E78AD38EB1468EC2AA1E62B8BED3A

Algorithmes cryptographiques :

Contrairement aux algorithmes de hachage, les algorithmes de cryptographie sont conçus pour crypter les données, puis les ramener à leur forme d'origine ultérieurement. Les notions de cryptographie s'étendent bien dans le passé avant l'invention des systèmes informatiques, les algorithmes de cryptographie d'origine étaient appelés chiffrements. Mais là où les premiers exemples tels que le chiffre César ou le chiffre Vigenère ont été conçus pour être déchiffrés à l'aide de la pensee, la cryptographie moderne repose sur des principes mathématiques robustes qui ne peuvent être déchiffrés pratiquement qu'à l'aide d'un ordinateur. Cependant, comme les algorithmes de hachage, il n'existe pas d'algorithme de cryptographie parfait. Il existe de nombreuses nuances de gris, de moins sûres à très sûres. Bien qu'il puisse être angoissant de savoir qu'il n'existe pas de cryptographie parfaite lors de la conception d'applications critiques, dans le domaine des systèmes embarqués, cela nous offre une grande flexibilité. À savoir, nous pouvons adapter nos exigences matérielles pour répondre à nos besoins de sécurité.

Chiffrement de bloc :

Développés à la fin des années 1970, les chiffrements par blocs sont l'une des premières formes de cryptographie informatisée encore en usage aujourd'hui. Pendant la guerre froide, les communications sécurisées étaient essentielles. Ainsi, les États-Unis et l'Union soviétique ont tous deux développé des systèmes concurrents qui ont été utilisés dans les communications diplomatiques et militaires. DES (Data Encryption Standard) était la norme pour le gouvernement fédéral américain de 1979 à 2005. GOST 28147-89 était le chiffrement par blocs concurrent développé par le KGB. GOST a été gardé secret jusqu'en 1990 et il voit encore une certaine utilité aujourd'hui. AES (Advanced Encryption Standard), développé en Belgique vers 2000, a supplanté l'algorithme DES comme chiffrement de choix du gouvernement américain[7]. En raison de sa disponibilité publique, AES est désormais le chiffrement par blocs le plus utilisé aujourd'hui.

Un chiffrement par blocs consiste en fait en une paire d'algorithmes. Le premier algorithme consiste à crypter un message et le second à décrypter un message (ou un fichier). L'algorithme de chiffrement fonctionne comme ceci $\boldsymbol{C = E(K, M)}$, où $\boldsymbol{E}$ est la fonction de chiffrement, $\boldsymbol{K}$ est une clé définie par l'utilisateur, $\boldsymbol{M}$ est le message à chiffrer et $\boldsymbol{C}$ est la version chiffrée de $\boldsymbol{M}$. Les chiffrements par blocs utilisent la clé pour générer une permutation pseudo-aléatoire d'un message. L'algorithme de décryptage ressemble à ceci : $\boldsymbol{M = D(K, C)}$. AES, et tous les chiffrements par blocs d'ailleurs, sont des algorithmes à clé symétrique, ce qui signifie qu'ils utilisent la même clé pour chiffrer et déchiffrer un message[13].

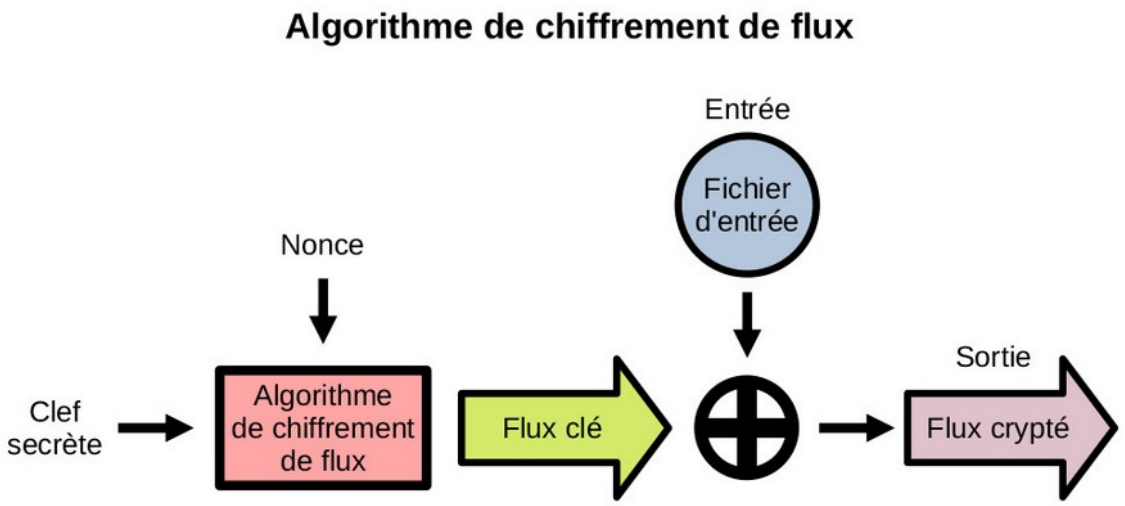
Les chiffrements de bloc tirent leur nom du fait qu'ils divisent un message en blocs afin de le chiffrer. En fait, les deux caractéristiques les plus importantes dans l'évaluation de la sécurité et des performances d'un chiffrement par blocs sont sa taille de bloc et sa taille de clé. La plupart des chiffrements par blocs modernes utilisent des tailles de bloc de 64 bits ou 128 bits. Il y a cependant un compromis, car des blocs plus volumineux nécessiteront plus de mémoire pour le chiffrement. Il est également important de vérifier quelle taille de bloc est optimale pour votre microprocesseur. Les processeurs ARM, par exemple, ont tendance à être plus efficaces avec des tailles de bloc de

128 bits par opposition à 64 bits[14]. En général, pour une meilleure sécurité, des blocs et des tailles de clé plus grands sont préférables. Cependant, cela pose un problème pour certains systèmes embarqués, bien que les chiffrements par blocs modernes soient généralement très sécurisés, ils peuvent être gourmands en ressources en fonction de la taille des messages chiffrés. Ils ne sont pas non plus idéaux dans les situations où les données doivent être cryptées et décryptées rapidement, comme un flux de données entre deux composants sur la même carte de circuit imprimé, car plusieurs copies de la clé devront être rendues disponibles. Il existe cependant des chiffrements par blocs qui ont été optimisés pour les systèmes à ressources limitées. Tiny AES[15] s'agit d'un exemple de chiffrement par blocs optimisé conçu pour s'exécuter sur des systèmes à mémoire limitée.

Chiffrement de flux :

Comme leur nom l'indique, les chiffrements de flux sont une classe d'algorithmes de cryptographie optimisés pour les flux de données. Ils se trouvent dans une variété de protocoles, y compris les communications mobiles Wifi, Bluetooth et 4G. Dans les systèmes embarqués, les chiffrements de flux sont généralement l'approche par excellence pour les communications inter-composants entre les circuits intégrés spécifiques à une application (ASIC), les dispositifs logiques programmables (PLD) et les matrices de portes programmables sur site (FPGA). Contrairement au chiffrement d'un message dans son ensemble, les chiffrements de flux sont conçus pour un chiffrement et un déchiffrement continus. Les chiffrements de flux ont deux architectures de base: **Basé sur l'état**: maintient un état interne qui évolue à mesure que les données sont chiffrées. **Basé sur un compteur**: cryptez les flux en morceaux[7].

Un chiffrement de flux typique a plus de pièces mobiles qu'un chiffrement par bloc typique. L'algorithme commence par créer un flux de clés. Le flux de clé est une série de bits aléatoires déterministes. Afin de générer le train de bits, l'algorithme nécessite une clé secrète et un nonce. Un nonce, abréviation de « number used only once » (nombre utilisé une seule fois), est généralement une valeur de 64 à 128 bits. Le chiffrement de flux peut être décrit par la formule suivante **KS(K, N) ⊕ D = C**, où **KS(K, N)** est le flux de clé généré par la clé secrète **K** et le nonce **N**. Le flux de clé est ensuite XOR avec les données à chiffrer **D,** résultant dans le flux crypté **C**[16]. Le résultat est une méthode très efficace pour crypter un flux de données. Il est important de noter que le nonce n'a pas besoin d'être secret pour assurer la sécurité. Cependant, afin d'assurer la sécurité, toute combinaison d'une clé et d'un nonce ne doit être utilisée qu'une seule fois pour crypter un flux de données. En effet, un attaquant ayant accès aux données non chiffrées $\mathbf{P_1}$, la version chiffrée de ces données ($\mathbf{C_1}$) peut casser le chiffrement sur un nouveau flux chiffré ($\mathbf{C_2}$) avec la formule suivante : $\mathbf{P_1 \oplus C_1 \oplus C_2 = P_2}$. Si la clé ou le nonce est changé avant de créer C2 cette méthode d'attaque sera rendue inefficace[7].



Les chiffrements de flux étaient la méthode de chiffrement matérielle la plus courante. En effet, la mise en œuvre du registre à décalage de rétroaction (Feedback Shift Register, FSR) des chiffrements de flux était plus facile et moins coûteuse à fabriquer. Cependant, ces dernières années, les chiffrements par blocs implémentés par le matériel se sont améliorés et en raison de la réduction du coût des semi-conducteurs, il n'y a plus de différence de prix entre les deux et les chiffrements par blocs implémentés par matériel sont monnaie courante[7]. Cela étant dit, les chiffrements de flux sont toujours très importants dans les exemples de systèmes à ressources limitées. Le Grain-128a en est un exemple, offrant une grande sécurité même lorsqu'il est implémenté sur des microcontrôleurs 8 bits. L'algorithme Espresso est un autre chiffrement de flux largement utilisé développé pour les communications sans fil 5G[17]. Dans un projet embarqué, votre choix de chiffrement devra être basé sur l'application.

Générateurs de nombres aléatoires :

Les générateurs de nombres aléatoires se trouvent partout dans la cryptographie. Ils sont essentiels pour créer des solutions de sécurité robustes, quel que soit le chiffrement ou l'algorithme que vous utilisez. Bien que certains algorithmes aient des générateurs de nombres aléatoires intégrés, beaucoup s'appuient sur des algorithmes supplémentaires pour fournir les nombres aléatoires. Dans les systèmes embarqués, le caractère aléatoire peut être une forme de sécurité en soi. Les attaques par canal secondaire reposent en particulier sur l'analyse de la régularité des opérations à l'intérieur des composants pour déterminer leur objectif et leur fonction. L'introduction du caractère aléatoire dans un système embarqué rend les attaques par canaux secondaires beaucoup plus difficiles et, dans certains cas, irréalisables. Bien qu'il existe des générateurs de nombres aléatoires basés sur le matériel, il est encore courant de s'appuyer sur des algorithmes logiciels, car cela réduira le coût des composants dans un projet.

Dans un ordinateur, techniquement parlant, il n'y a pas de bit aléatoire. C'est pourquoi vous avez peut-être entendu le terme pseudo-aléatoire. Un ordinateur est une machine à états finis, ce qui signifie qu'il y a toujours un nombre limité d'états internes possibles qui peuvent exister. C'est-à-dire

qu'un nombre généré à partir d'une machine dont l'état initial est connu, avec un nombre calculable d'états possibles, ne peut pas être vraiment aléatoire.

Registre à décalage de rétroaction linéaire (LFSR) :

La base des générateurs de nombres aléatoires algorithmiques sont les registres à décalage à rétroaction linéaire (LFSR). Voici un exemple de code pour générer des nombres pseudo-aléatoires de 16 bits avec des LFSR similaires à la fonction rand() à partir de la librairie open source glibc[18].

Note: Ce code n'est pas adapté à la cryptographie et ne doit pas être utilisé dans les applications de sécurité.

```
unsigned short lfsr = 0xACE1u;
unsigned bit;

unsigned rand()
{
      bit  = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1;
      return lfsr =  (lfsr >> 1) | (bit << 15);
}
```

Source: Ce code a été adapté de l'article de wikipedia sur des LFSR
https://en.wikipedia.org/wiki/Linear-feedback_shift_register#Uses_in_cryptography

Le programme démarre avec une constante fixe, car une opération sur un registre mis à zéro produira toujours zéro. Le LFSR fonctionne essentiellement par une séquence de bits de décalage et l'exécution d'un XOR sur la position de bit libérée. Bien que cela fonctionne pour la génération de nombres aléatoires de base et soit important dans le contrôle de redondance cyclique (CRC), un seul LFSR n'est pas suffisant pour la cryptographie. En effet, l'algorithme n'est pas assez aléatoire, si un attaquant connaît l'état du système, il peut prédire quels nombres seront générés. Certaines méthodes pour améliorer ce programme consisteraient à enchaîner plusieurs LFSR ou à utiliser un LFSR de plus grande taille.[19].

Algorithmes d'aléa modernes :

En 1997, le Mersenne Twister a été développé par M. Matsumoto et T. Nishimura comme une meilleure alternative aux générateurs de nombres pseudo-aléatoires LFSR traditionnels. L'algorithme de Mersenne Twister, bien que beaucoup plus complexe que le simple programme LFSR décrit ci-dessus, repose toujours sur certains des mêmes principes sous-jacents. Aujourd'hui, le Mersenne Twister est le générateur de nombres pseudo-aléatoires le plus largement utilisé, bien qu'il existe maintenant des algorithmes plus avancés pour les applications critiques.[20].

Dans les systèmes embarqués, un algorithme générateur de nombres aléatoires présente des avantages et des inconvénients. Si le caractère aléatoire est nécessaire pour votre application intégrée, il vaut la peine d'envisager des implémentations basées sur le matériel. Ceux-ci reposent sur les propriétés d'entropie des matériaux physiques et peuvent, en théorie, produire des nombres à un degré plus aléatoire que n'importe quel algorithme. Les générateurs de nombres aléatoires basés sur le matériel ont également l'avantage de ne pas consommer de ressources de traitement.

Conclusion :

Choisir la bonne solution de sécurité pour votre prochain projet de systèmes embarqués peut être une tâche ardue. Cependant, la sécurité devient rapidement un élément essentiel de presque tous les projets embarqués. À mesure que les appareils de l'Internet des objets prolifèrent, le nombre de vecteurs d'attaque potentiels augmente. L'IoT devient rapidement un vecteur d'attaque principal en tant qu'intermédiaire pour accéder à des réseaux sécurisés[21]. La sécurité n’est pas toujours une question de confidentialité. Une autre chose à considérer est l'expérience de l’utilisateur. Il est facile de justifier le coût de la sécurité lors du développement d'un système embarqué pour une application industrielle critique. Cependant, même si vous développez des articles de consommation à faible coût, des failles dans votre sécurité peuvent entraîner des appareils compromis et une expérience de l’utilisateur négative. À long terme, une mauvaise réputation peut détruire des entreprises. J'espère que je vous ai fourni un bon aperçu pour évaluer vos besoins en matière de sécurité. Le paysage moderne de la sécurité intégrée est un défi en constante évolution. Les meilleures solutions de sécurité envisagent les menaces des années dans le futur. Mais ne laissez pas la sécurité intégrée vous intimider, comme la programmation, le développement de solutions de sécurité est une compétence développée au fil du temps. Même si vous ne développez pas vous-même des solutions de sécurité, il est essentiel d'avoir une certaine compréhension de la sécurité intégrée afin de pouvoir poser les bonnes questions aux experts.

Bibliographie

1: Sarah Fallon, *The Secret History of the First Microprocessor, the F-14, and Me*, 2020. https://www.wired.com/story/secret-history-of-the-first-microprocessor-f-14/

2: W. Stout, V. Urias, *Challenges to Securing the Internet of Things*, 2016. https://www.osti.gov/servlets/purl/1380087

3: T. Stapko, *Cryptography for embedded systems – Part 1: Security level categories & hashing*, 2010. https://www.eetimes.com/cryptography-for-embedded-systems-part-1-security-level-categories-hashing/

4: IBM Security, *The weaponization of IoT devices*, 2017. https://www.ibm.com/downloads/cas/6MLEALKV

5: Dennis Fisher, *SHA-1 'Fully and Practically Broken' by New Collision*, 2020. https://duo.com/decipher/sha-1-fully-and-practically-broken-by-new-collision#:~:text=UPDATE%2D%2DSHA%2D1%2C%20the,chosen%2Dprefix%20collision%20for%20it.

6: M. Turan, E. Barker, W. Burr et L. Chan, *Reccomendation for Password-Based Key Derivation*, 2010. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf

7: J-P. Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption*, 2018

8: V. Klima, *Finding MD5 Collisions - a Toy For a Notebook*, 2005. https://cryptography.hyperlink.cz/md5/MD5_collisions.pdf

9: D. Bindel, M. Chew, C. Wells, *Extended Cryptographic File System*, University of California Berkley, 1999. https://people.eecs.berkeley.edu/~kubitron/courses/cs252-F99/projects/reports/project6_report.ps.gz

10: B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 1996

11: S. Josefsson et I. Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, 2017. https://datatracker.ietf.org/doc/html/rfc8032

12: R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2020

13: H. M. Heys, *A Tutorial on the Implementation of Block Ciphers: Software and Hardware Applications*, 2020. https://eprint.iacr.org/2020/1545.pdf

14: P. Schwabe et K. Stoffelen, *All the AES You Need on Arm Cortex-M3 and M4*, 2016. https://eprint.iacr.org/2016/714.pdf

15: , *Tiny-AES-c*, 2019. https://github.com/kokke/tiny-AES-c

16: The University of Sydney, Elementary Cryptography: Stream Ciphers, 2020

17: S. Deb et B. Bhuyan, *Performance evaluation of Grain family and Espresso ciphers for applications on resource constrained devices*,ICT Express 2018. https://www.sciencedirect.com/science/article/pii/S2405959517303612

18: Free Software Foundation, Inc., *19.8 Pseudo-Random Numbers*, 2020. https://www.gnu.org/software/libc/manual/html_node/Pseudo_002dRandom-Numbers.html#Pseudo_002dRandom-Numbers

19: Maxim Integrated Products, Inc., *APPLICATION NOTE 4400: RANDOM NUMBER GENERATION USING LFSR*, 2010. https://www.maximintegrated.com/en/design/technical-documents/app-notes/4/4400.html#:~:text=In%20general%2C%20a%20basic%20LFSR,the%20register%20for%20your%20number.

20: S. Marsland, *Machine Learning*, 2011

21: Z. Zorz, *The FBI warns about compromised IoT devices*, 2018.
https://www.helpnetsecurity.com/2018/08/06/spot-compromised-iot-devices/