REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

Université 08 mai 45 Guelma
Faculté des Sciences et de l'Ingénierie

Département d'Informatique

Ecole Doctorale d'Informatique de l'Est (pole : Annaba)
Option : Génie logiciel

Mémoire présenté pour l'obtention du diplôme de :

Magister en Informatique

Titre du Mémoire :

Vérification de code pour systèmes Embarqués

Présenté par :

Messolhi Naima

Composition du jury :

Président : Pr. H. Seridi, Université de Guelma
Rapporteur : Pr. M. Benmohammed, Université de Constantine
Examinateurs : Dr. A. Chaoui, Université de Constantine
Dr. A. Bilami, Université de Batna

2010

Remerciements

أشكر الله سبحانه وتعالى الذي أمدني بالقوة لإتمام هذا العمل

Je tiens à remercier mon encadreur, ***Mr. Benmohammed Mohammed****, Professeur à l'université de Mentouri Constantine, pour ses précieux conseils, ses encouragements prodigués tout au long de ce travail de recherche, sa disponibilité et surtout sa patience avec moi. Qu'il trouve ici l'expression de ma profonde gratitude.*

Je remercie chaleureusement ***Pr Seridi Hamid****, Professeur à l'université 8 mai 45 Guelma, d'avoir accepté de présider mon jury. Je le remercie également d'avoir toujours été à notre écoute ainsi que pour tous les efforts qu'il a fournis pour que notre formation se déroule dans des conditions favorables.*

Je remercie ***Dr Chaoui Allaoua****, maître de conférences à l'université Mentouri Constantine, d'avoir accepté d'être l'examinateur de ce mémoire.*

Je remercie ***Dr Bilami Azeddine****, maître de conférences à l'université de Batna, d'avoir accepté d'être l'examinateur de ce mémoire.*

Mes remerciements vont également à tous les enseignants de notre année théorique pour leurs encouragements et leur compréhension.

Je remercie chaleureusement mon amie Benati Nadia, qui était vraiment une grande sœur pour moi et m'a aidé à terminer ce travail jusqu'aux dernières étapes.

Un grand merci à tous mes collègues de l'Ecole Doctorale en Informatique de l'Est spécialement Anissa, Samira, Samia, Houda, Mohammed Salah et Ridha pour leurs encouragements et leur amitié. Je profite de ces quelques lignes pour vous souhaiter toute la réussite pour vos thèses respectives.

Merci également à tous mes chers amis : Souhila, Nadia, Wafa, Nadjette, Soulaf, Chafia, Karima, Mounira, Hamdi, Ridha.

A toute personne ayant contribué de prés ou de loin à l'élaboration de ce mémoire, Je dis merci.

Dédicace

Je dédié ce travail aux êtres qui me sont les plus chères au monde:

ma mère, mon père

mes frères Abdel Aziz, Lakhder, Said, Djeloul, Abdelkarim

ma petite sœur Wahiba

ces êtres qui ont été toujours à mes cotés pour me soutenir

et m'encourager

A la mémoire de ma grande sœur Naîma

A tous ceux qui me sont chers.

ملخـــــص

للتحقيق أهمية متزايدة في ميدان الحاسوب نظرا لكوننا أصبحنا نعتمد بشكل متزايد على النظم الحاسوبية. سواء للتواصل ، للتحرك ، للعلاج ... أصبحنا نستخدم الحاسوب والبرمجيات المصاحبة لها في مختلف الميادين. هذا التعميم في استعمال هذه الأداة أكسبها مزيدا من الوزن أكثر من غيرها فقد أصبحت مسؤولة عن إدارة الكثير من الجوانب الحساسة في حياتنا.

من جهة أخرى أضافت الأنظمة المضمنة (les Systèmes Embarqués) بعدا جديدا لأمن. بصفة عامة يمكن القول أنه يمكن إعادة برمجة الأنظمة المضمنة، وذلك بتنزيل برنامج جديد وتحميلها واستعمالها على أي نوع من أنواع هذه الأنظمة، وذلك في شكل سلسلة من التعليمات التي تنفذها المعالجات الدقيقة.

عملية تحميل برنامج جديد على نظام مضمن تثير عددا من المشاكل الأمنية. حيث أن إضافة برنامج جديد مكتوب بطريقة غير سليمة سواء بقصد أو دون قصد قد يغير البيانات الموجودة مسبقا على النظام ، أو قد يمنع البرنامج الرئيسي من تشغيل البرامج بشكل صحيح، أو قد يغير تطبيقات أخرى سبق تحميلها، أو يجعلها غير صالحة للاستعمال مطلقا.

في هذه الدراسة قمنا بتقديم طريقة للتحقق من البرنامج قبل استعمالها وقد اخترنا استخدام التوقيع والتصديق على البرنامج كحل لضمان محتواه. هذه الطريقة تتيح للمستخدمين تحميل تطبيقات موثوقة وتوفير معلومات شاملة عن البرامج حيث يتم ارفاقها بشهادة .

الكلمات الرئيسية : الأنظمة المضمنة ، الأمن ،التحقق من البرامج، التوقيع والتصديق الرقمي.

Résumé

La vérification prend une place croissante dans l'informatique moderne. En grande partie parce que nous nous reposons de plus en plus sur les systèmes informatiques. Que cela soit pour communiquer, pour nous déplacer, pour nous distraire ou encore pour nous soigner, nous utilisons des ordinateurs et les logiciels qui les accompagnent. De par cette généralisation de l'outil informatique, certains systèmes acquièrent plus d'importance que d'autres car ils ont la responsabilité de gérer des aspects sensibles de notre environnement.

Les systèmes embarqués ajoutent une nouvelle dimension à la sécurité des logiciels présents sur ces systèmes. D'une manière générale, on dit que les systèmes embarqués sont reprogrammables, on peut télécharger un nouveau programme constitué par un code exécutable, se présentant sous la forme d'une suite d'instructions exécutées par le microprocesseur.

L'opération de téléchargement d'application sur un système embarqué pose un certain nombre de problèmes de sécurité. Ainsi, un code involontairement mal écrit peut modifié de manière incorrecte des données déjà présente sur le système, empêcher le programme principal de s'exécuter correctement, ou encore modifier d'autre code d'applications téléchargées précédemment, en rendant ceux-ci inutilisables ou nuisibles.

Dans ce mémoire, nous allons présenté une méthode de vérification des codes pour les systèmes embarquées on utilisant la signature et la certification du code. Cette méthode permet aux utilisateurs de faire confiance aux applications embarquées on donnant des informations complètes concernant le producteur de code contenu dans un certificat transférer avec.

Mots clés : système embarqué, sécurité, vérification de code, signature et certification numérique.

Abstract

The verification takes a big importance in modern computer. In large part because we rely increasingly on computer systems. Whether it is to communicate, to move, to distract us or to treat us, we use computers and software that accompany them. By this generalization of the tool, have some gain more weight than others because they are responsible for managing the sensitive aspects of our environment.

Embedded systems add a new dimension to the security software on those systems. Broadly speaking, we say that embedded systems are reprogrammable, you can download a new program, a sequence of instructions executed by the microprocessor.

The process of downloading the application on a embedded system raises a number of security problems. Thus, a poorly a code can unintentionally modified incorrectly data already present on the system, preventing the main program to run properly, or change other code applications previously downloaded, making them unusable or harmful.

In our work we presented a method of verification codes for embedded systems using on-board signature and certification of code. This method allows users to trust the applications loaded on providing comprehensive information about the producer of container in the code certificate accompanying transfer.

Keywords: embedded system, security, code verification, signature and digital certification.

Table des matières

Table des figures

Introduction

1. Introduction générale

De nos jours, les systèmes embarqués envahissent notre quotidien. Beaucoup de fonctions, jadis, réalisées manuellement sont aujourd'hui automatisées grâce à l'électronique et l'informatique embarquée. Ceci a apporté des améliorations notables en termes de confort. Un confort auquel, il est de plus en plus difficile de renoncer.

La part grandissante de l'électronique et de l'informatique n'est pas sans conséquences sur les méthodes de conception. La complexité croissante des systèmes embarqués nécessite une adaptation des processus, méthodes et outils existants par rapport aux spécificités de ces systèmes pour mieux répondre aux exigences, notamment, celles liées à la sûreté de fonctionnement. En effet, la préoccupation des industriels est de proposer à leurs clients des produits intégrant les nouvelles innovations technologiques avec une qualité et des performances de plus en plus améliorées, mais aussi des produits de plus en plus sûrs.

La criticité de ces systèmes nécessite de garantir un niveau de fiabilité et de sécurité convenable. Des études de sûreté de fonctionnement doivent être menées tout au long du cycle de développement du système.

Un **système embarqué** peut être défini comme un système électronique et informatique autonome, qui est dédié à une tâche bien précise. Ses ressources disponibles sont généralement limitées. Cette limitation est généralement d'ordre spatial (taille limitée) et énergétique (consommation restreinte). Les systèmes embarqués font très souvent appel à l'informatique, et notamment aux systèmes temps réel. Il doit répondre à des impératifs de criticité, de réactivité, d'autonomie, de robustesse et de fiabilité.

Plutôt que des systèmes universels effectuant plusieurs tâches, les systèmes embarqués sont étudiés pour effectuer des tâches précises. Certains doivent répondre à des contraintes de temps réel pour des raisons de sécurité et de rentabilité. D'autres ayant peu de contraintes au niveau performances permettent de simplifier le système et de réduire les coûts de fabrication.

2. Objectifs de recherche

L'avancée technologique que les systèmes embarqués ont connue, lors de ces dernières années, les rend de plus en plus complexes. Ils sont non seulement responsables de la commande des différents composants mais aussi de leur surveillance. A l'occurrence d'événement pouvant mettre en danger la vie des utilisateurs, une certaine configuration du système est exécutée afin de maintenir le système dans un état dégradé mais sûr. Il est possible que la configuration échoue conduisant le système dans un état appelé "état redouté" avec des conséquences dramatiques pour le système et l'utilisateur.

Les méthodes de signature et de certification du code répondent aux besoins de sécurité et garantirent dans le même temps la flexibilité et la réactivité du système embarqué, en autorisant le chargement d'applications même après la phase de fabrication. De plus, comme ces systèmes (exp : le téléphone portable) permettent de gérer plusieurs applications au sein d'un même environnement sécurisé, elles permettent aussi de développer des applications collaborant de manière sécurisée sur un même support – chose auparavant assez difficile.

L'objectif de ce travail est d'augmenter la fiabilité et la sécurité du système embarqué en utilisant la signature et la certification dans le processus de vérification du code. L'idée est d'implémenter un outil qui sera utilisé d'une part pour signer et certifier les applications, et d'autre part pour vérifier cette signature avant d'installer le nouveau code dans notre système embarqué.

En nous appuyant sur les outils de sécurité fournis par Java et les aspects de certification et de signature du code, nous proposons de développer des outils permettant de garantir simultanément la flexibilité et la sécurité de ces systèmes. Au terme de ces processus de vérification, ces méthodes constituent en principe l'un des méthodes les plus sécurisées des marchés des applications embarqués.

Dans ce cadre, les applications serons évaluée et certifiée et pourront être chargées dans notre système.

3. Organisation du mémoire

Le mémoire est composé de cinq chapitres organisés comme suit:

Dans le premier chapitre nous évoquerons les systèmes embarqués, contraintes, architecture, caractéristiques, domaines d'applications…. Puis, nous expliquerons la nécessité de la vérification du code des applications embarquées.

Le deuxième chapitre aborde les exigences de sécurité pour le système embarqué en général ainsi des défis de conception et les attaques sur ces systèmes.

Le troisième chapitre est dédié à discuter les différentes méthodes permettant de produire un code fiable et sûr. Nous expliquerons les principes généraux, les différentes techniques, en montrons les avantages et les limites de chaque méthode. Nous terminerons le chapitre par une brève introduction à certaines études qui montrent certains des travaux de la vérification pour les systèmes embarqués qui fait l'objet de notre recherche.

Le but du quatrième chapitre est de présenter et discuter l'utilisation du langage Java dans les système embarqués, nous commencerons par présenter des exemples du langage de programmation pour l'embarqué, nous expliquerons les problèmes liés au déploiement des applications embarqués et nous terminerons par détailler certaines solutions permettant d'embarquer Java.

Le cinquième chapitre présente la signature et la certification du code comme une solution utilisée pour garantir un certain niveau de sécurité dans le domaine du système embarqué. Les technologies multi-applicatives résolvent beaucoup de problèmes du monde du système embarqué, elles en créent également de nouveaux et notamment dans le domaine essentiel de la sécurité. En effet, puisqu'il est possible d'embarquer plusieurs applications de différents fournisseurs, un émetteur d'application potentiellement malhonnête, ou tout simplement peu informé des recommandations sécuritaires, pourrait charger une application malicieuse ou involontairement mal programmée qui risquerait de mettre en péril la sécurité de toute la plate-forme.

Nous terminons ce mémoire par une conclusion générale dans laquelle nous évoquerons les perspectives pour notre travail.

Chapitre 1

Les systèmes Embarqués

Résumé

Ce chapitre donne une introduction aux systèmes embarqués, à leurs caractéristiques, les contraintes liées à ce type de système. Enfin nous expliquerons leurs enjeux sociaux et industriels et nous donnerons quelques exemples de systèmes.

Sommaire

1.1 Introduction aux systèmes embarqués

De nombreux produits de consommation (ordinateurs, téléphones portables, appareils électroménagers, téléviseurs, appareils photographiques..) comportent des circuits électroniques intégrés sur du silicium ou puces. Ces circuits apportent des fonctions nouvelles (produits sans fil par exemple) ou corrigent des imperfections.

Cette augmentation importante du nombre de puces n'a été possible que par une diminution exponentielle de leur coût de production. Cette diminution du coût est due à deux facteurs : la miniaturisation des dispositifs élémentaires et l'amélioration des techniques de fabrication collective.

Cette évolution spectaculaire se poursuivra dans les prochaines années car le marché mondial est loin d'être saturé et de nombreux produits n'intégrant pas encore de puces électroniques en seront équipés dans le futur (étiquettes d'identification des produits, implants médicaux, pneus de véhicule, cartables pour les écoliers, vêtements, matériaux de construction…).

1.1.1 Réduire encore et toujours les dimensions des transistors !

Les puces actuelles sont constituées d'un grand nombre de composants élémentaires appelés transistors qui permettent d'effectuer les trois principales fonctions des systèmes électroniques: *calculer* (unité centrale d'un processeur par exemple) *garder en mémoire* (stockage d'images par exemple) *échanger* de l'information avec l'environnement (affichage, capteur) détection de molécules, comptage de photons…).

Ces fonctions conduisent à réaliser des blocs de calcul, des blocs mémoire et des blocs d'interface. La force de la micro-électronique est d'avoir su réaliser ces blocs à partir d'un seul composant : le transistor.

Le coût de fabrication d'un circuit étant proportionnel à la surface, diminuer la dimension du transistor permet donc d'intégrer plus de transistors par unité de surface et donc de réduire le coût d'une fonction donnée. La miniaturisation du transistor a avancé avec une telle constance qu'elle est caractérisée par une loi, la loi de Moore.

Gordon Moore, cofondateur de la société Intel avait constaté en 1965 que le nombre de transistors par circuit de même taille doublait tous les 18 mois. La figure 1.1 montre cette évolution inexorable.

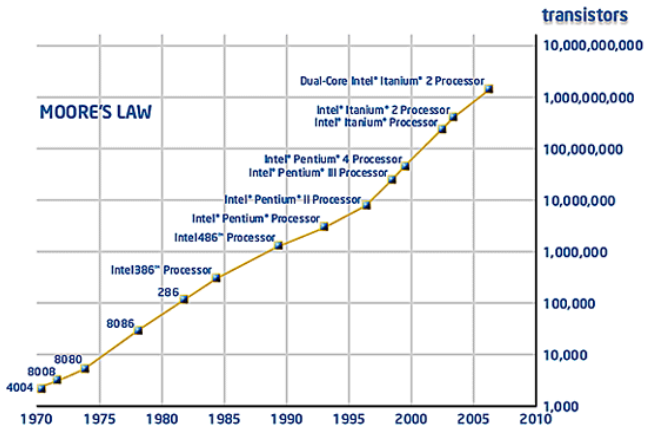


Figure 1.1 La loi G. Moore pour les processeurs Intel [89]

D'autre part, selon [43] : en 1999, il a été vendu pour le marché de l'embarqué

- 1,3 milliard de processeurs 4 bits.
- 1,4 milliard de processeurs 8 bits.
- 375 millions de processeurs 16 bits.
- 127 millions de processeurs 32 bits.
- 3,2 millions de processeurs 64 bits.

A côté de cela, à cette époque, il a été vendu seulement 108 millions de processeurs (famille x86) pour le marché du PC grand public !

Pour 2004, il a été vendu environ 260 millions de processeurs pour le marché du PC grand public à comparer aux 14 milliards de processeurs tout type confondu (microprocesseur, microcontrôleur, DSP) pour le marché de l'**embarqué**.

Le marché des processeurs pour l'embarqué selon *Electronics.ca Research Network* devrait croître de 6 % en 2005 pour un chiffre d'affaire mondial de 18 milliards USD ! Les chiffres parlent d'eux-même. Le marché du processeur pour les PC grand public n'est que la partie émergée de l'iceberg et n'est rien par rapport au marché de l'embarqué qui est la partie immergée de l'iceberg.

On peut aussi tirer de [43] le constat actuel suivant :

- Moins de 2 % des processeurs vendus sont pour le marché du PC contre 98 % pour l'embarqué.

- On utilise massivement pour le PC grand public le système d'exploitation commercial bien connu. Pour les 98 % autres processeurs vendus, on utilisera généralement un autre système d'exploitation. On trouvera ici dans 60 % des cas un système d'exploitation commercial spécialisé pour l'embarqué (VxWorks, QNX...). Pour le reste, il s'agit d'un système d'exploitation *home made* mais de plus en plus optent pour des systèmes d'exploitation libres comme Linux pour l'embarqué.
- Moins de 10 % des processeurs vendus sont des processeurs 32 bits pour près de 31% du chiffre d'affaire sur les processeurs. Cette part du chiffre d'affaire est estimée a près de 48 % pour 2008 : cela montre la migration rapide vers ces processeurs 32 bits dans l'embarqué, condition nécessaire pour pouvoir mettre en oeuvre Linux.
- Si l'on regarde le prix moyen d'un processeur tout type confondu, on arrive à 6 USD par unité à comparer au prix moyen de 300 USD par unité pour un processeur pour PC. Le marché du processeur pour PC est très faible en volume mais extrêmement lucratif.

Ces quelques chiffres permettent bien de prendre conscience de l'importance du marché de l'embarqué et de mettre fin à l'idée qu'en dehors du marché du PC, point de salut.

Cette constatation a marqué les esprits puisqu'elle est devenue un défi à tenir pour les fabricants de circuits intégrés. Une technologie est définie en microélectronique par une grandeur appelée noeud de la technologie. Les technologies avancées actuelles correspondent aux noeuds 90 et 65 nm mais des technologies plus avancées sont étudiées dans les laboratoires (45 et 22 nm). Le nanomètre étant le milliardième de mètre, les dimensions des transistors se rapprochent donc de la distance entre deux atomes dans un cristal, si bien que les dispositifs futurs ne comporteront que quelques milliers, voire quelques centaines d'atomes.

De nouveaux effets physiques sont alors possibles puisque le libre parcours moyen des électrons (la distance moyenne entre deux collisions avec le cristal) peut devenir supérieur à la dimension du composant. Les nanosciences et les nanotechnologies fournissent les outils de calcul et de mesure aptes à quantifier et contrôler ces effets liés à cette miniaturisation extrême.

1.1.2 Les difficultés de la conception des circuits intégrés

Les circuits intégrés actuels comportent des millions de transistors, des milliards pour les plus complexes. Les méthodes de conception des circuits intégrés ont subi une évolution vertigineuse au cours des deux dernières décennies. Dans ce domaine, l'évolution technologique se traduit d'abord par une diminution régulière de la taille des transistors induisant une densité d'intégration toujours plus importante et une augmentation constante de la fréquence de fonctionnement, mais aussi par un accroissement régulière de la taille des puces.

A la différence de la conception des cartes électroniques qui tolère des erreurs et des modifications, la conception des circuits intégrés n'en tolère aucune. Le coût des jeux de masques nécessaires à la fabrication d'un circuit est très élevé, il est donc indispensable que le circuit soit fonctionnel après un nombre très limité d'essais.

La maîtrise d'un flot de conception rigoureux est donc aussi importante que la maîtrise des procédés technologiques. Cette exigence a donné naissance à un ensemble d'outils logiciels qui assistent les concepteurs en particulier pour les opérations de vérification. Un circuit intégré avancé comprend un ou plusieurs processeurs, des circuits d'interface mais aussi une partie mémoire de taille relativement importante. Cette mémoire est suffisante pour contenir des logiciels relativement complexes appelés *logiciels embarqués* ou enfouis. La conception de ces logiciels très dépendants de l'architecture matérielle a considérablement modifié le flot de conception des circuits intégrés et transformé le métier de la conception de circuits numériques.

1.2 Systèmes Embarqués :

1.2.1 Définition :

Les systèmes embarqués sont des systèmes électroniques qui sont complètement intégré au système qu'ils contrôlent. On les utilise dans différents domaines d'application tel que dans le transport, l'astronautique, le militaire les télécommunications, l'électroménager, les équipements médical, les guichets bancaires automatiques [73]... Son utilisation dans ces multiples domaines traduit bien l'importance du marché de l'embarquer de nos jours.

Un système embarqué combine généralement diverses technologies qui relèvent des domaines de la mécanique, de l'hydraulique, de la thermique, de l'électronique et des

technologies de l'information. Il peut être décomposé en trois entités en interaction : les capteurs, les calculateur et les actionneurs. (Figure 1.2)

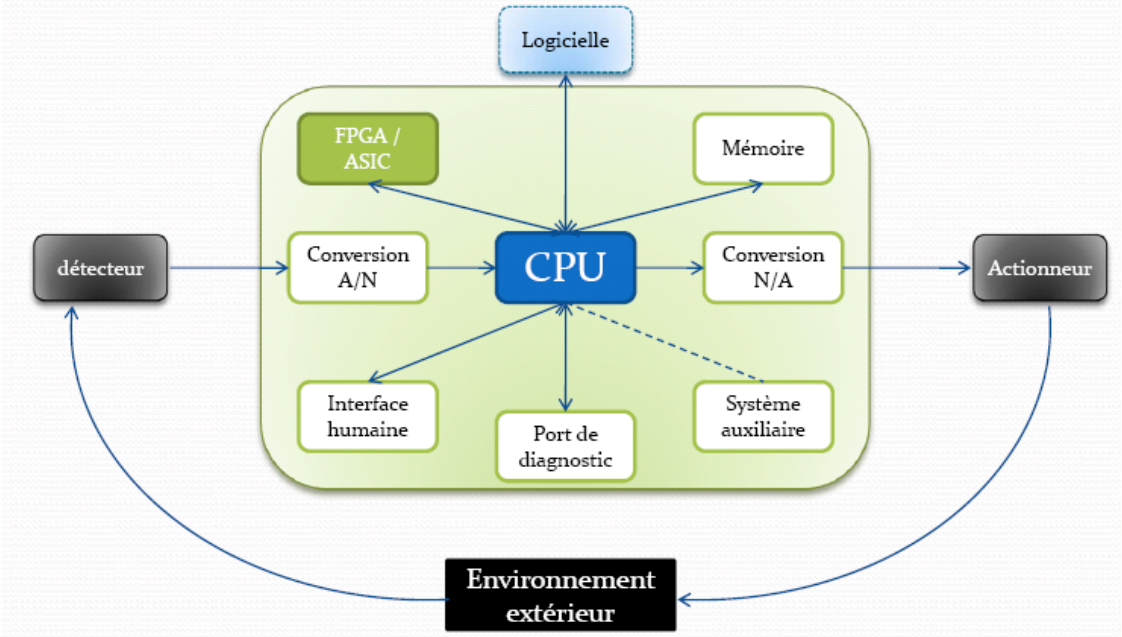


Figure 1.2. Système embarqué typique [36]

Les *capteurs* fournissent des informations sur l'état du système. Ces informations sont utilisées par le *calculateur* pour générer la commande à appliquer au système au moyen d'*actionneurs*.

Au cours du développement d'un système embarqué, le concepteur doit choisir entre plusieurs solutions d'architectures matérielles et logicielles répondant à des critères de performances et de sûreté de fonctionnement exprimés dans les spécifications.

Tout système n'est pas à l'abri d'erreur de conception est donc à l'abri de l'apparition de défaillance au cours de son cycle de vie. Le concepteur doit alors disposer de moyens pour éviter les fautes.

Une fiabilité absolue n'est jamais garantie lors de la conception d'un système. Le concepteur introduit alors des mécanismes de tolérance aux fautes afin d'éviter que des erreurs ou des fautes entraînent une défaillance du système.

1.2.2 Les contraintes de temps et les systèmes embarqués

On entend souvent parler de Temps Réel dès que l'on parle de système embarqué. En fait, un système embarqué doit généralement respecter des contraintes temporelles fortes *(Hard*

Real Time) et l'on y trouve enfoui un système d'exploitation ou un noyau Temps Réel (*Real Time Operating System,* RTOS).

Le Temps Réel est un concept un peu vague et chacun a sa propre idée sur la question. On pourrait le définir comme : *Un système est dit Temps Réel lorsque l'information après acquisition et traitement reste encore pertinente*. [79]
Plus précisément, cela veut dire que dans le cas d'une information arrivant de façon périodique (sous forme d'une interruption périodique du système), les temps d'acquisition et de traitement doivent rester inférieurs à la période de rafraîchissement de cette information. Un temps maximum d'exécution est garanti (pire cas) et non un temps moyen.

Pour cela, il faut que le noyau ou le système Temps Réel :

- Soit ***déterministe*** : les mêmes causes produisent les mêmes effets avec les mêmes temps d'exécution.
- Soit ***préemptif*** : la tâche de plus forte priorité prête à être exécutée doit toujours avoir accès au processeur.

Ce sont là des conditions nécessaires mais malheureusement pas suffisantes pour affirmer qu'un système embarqué est Temps Réel par définition.
Une idée reçue est de mélanger Temps Réel et puissance de calcul du système embarqué. On entend souvent : Etre temps Réel, c'est avoir beaucoup de puissance : des MIPS, des MFLOPS.
Ce n'est pas toujours vrai. En fait, être Temps Réel dans l'exemple donné précédemment, c'est être capable d'acquitter l'interruption périodique (moyennant un temps de latence de traitement d'interruption imposé par le matériel), traiter l'information et le signaler au niveau utilisateur (réveil d'une tâche, libération d'un sémaphore…) dans un temps inférieur au temps entre deux interruptions périodiques consécutives.
On est donc lié à la contrainte durée entre deux interruptions. C'est donc bien le process extérieur à contrôler qui impose ses contraintes temporelles au système embarqué et non le contraire. Si cette durée est de *l'ordre de la seconde* (pour le contrôle d'une réaction chimique par exemple), il ne sert à rien d'avoir un système à base de processeur 32 bits performant. Un simple processeur 8 bits voire même un processeur 4 bits fera amplement l'affaire ; ce qui permettra de minimiser les coûts sur des forts volumes de production. Si ce temps est maintenant de *quelques dizaines de microsecondes* (pour le traitement des données issues de l'observation d'une réaction nucléaire par exemple), il est alors

nécessaire de choisir un processeur nettement plus performant comme un processeur 32 bits (processeurs ARM, ColdFire..). Dans le pire des cas, le traitement en Temps Réel sera réalisé en logique câblé tout simplement. L'exemple donné est malheureusement idyllique (quoique fréquent dans le domaine des télécommunications et réseaux) puisque notre monde interagit plutôt avec un système embarqué de façon apériodique.

Il convient donc avant de concevoir le système embarqué de *connaître la durée minimale entre 2 interruptions* ; ce qui est assez difficile à estimer voire même impossible.

C'est pour cela que l'on a tendance à concevoir dans ce cas des systèmes performants et souvent surdimensionnés pour respecter des contraintes Temps Réel mal cernées à priori. Ceci induit en cas de surdimensionnement un surcoût non négligeable.

1.2.3 Contexte technique et industriel du logiciel embarqué

Les systèmes embarqués font partie de notre quotidien. Bien que nous ne les percevions pas toujours, ils font partie intégrante des automobiles, des infrastructures de transport, des appareils électroménagers, des réseaux de télécommunications, des laboratoires médicaux; ils occupent tous les secteurs de marché.

Un logiciel embarqué est un logiciel permettant de faire fonctionner une machine, équipée d'un ou plusieurs microprocesseurs, afin de réaliser une tâche spécifique avec une intervention humaine qui peut être limitée [18] (pour la commande de systèmes autonomes) ou au contraire primordiale dans les technologies mobiles (téléphonie, etc).

Exemples : Calculateurs dans une voiture (pour le freinage ABS...) Calculateurs dans un avion (commandes de bord...) Cartes à puce, applications transactionnelles (bancaire, carte vitale, etc) Téléphones mobiles, systèmes embarqués par eux-mêmes, dont on peut noter qu'ils intègrent aussi un sous-système embarqué constitué par la carte SIM Lecteurs mp3, décodeurs...

Selon [11], Le logiciel embarqué doit prendre en compte les caractéristiques suivantes des systèmes, chacune d'elles nécessitant d'inventer des concepts adéquats pour définir, construire, analyser, développer et exploiter les logiciels adaptés.

- **Autonomie :**

  Les systèmes embarqués doivent en général être autonomes de plusieurs points de vue. Tout d'abord ils doivent remplir leur mission pendant de longues périodes avec une

intervention humaine qui peut être limitée. Cette autonomie nécessite des capacités d'autoconfiguration et d'auto-réparation. L'autonomie est aussi d'ordre énergétique: l'optimisation de la consommation électrique des System on chip complexes destiné aux équipements nomades notamment est une préoccupation majeure.

- **Interaction :**

  Les systèmes embarqués interagissent avec l'environnement physique et humain. Ils doivent donc embarquer des logiciels capables de gérer les sources multiples de l'interaction (et de la communication) et les différentes échelles de temps qui peuvent être impliquées.

- **Communication :**

  Bien que la première caractéristique des systèmes embarqués soit l'autonomie, donc une capacité de traitement locale, les échanges d'information avec les autres systèmes prennent une part de plus en plus essentielle dans le fonctionnement de tous les systèmes informatiques. Au-dessus de la couche physique de communication et du traitement du signal associé, le logiciel doit prendre en compte toutes les autres couches de la communication, de celles qui vont assurer la surveillance de la liaison à la gestion de la communication au niveau des systèmes perceptifs et cognitifs humains, en passant par le routage dans différents réseaux et environnements, la mise en forme de l'information numérique, la sécurité des communications etc.

- **Réactivité:**

  Les logiciels embarqués sont déployés dans un environnement physique, avec lequel ils interagissent en permanence. Ceci implique qu'ils sont soumis à des contraintes de temps-réel reliant leur temps d'exécution au temps de réaction de l'environnement. Comme l'environnement ne peut pas attendre, le système et l'environnement ne peuvent pas se synchroniser. Au-delà des contraintes temps-réel, les logiciels embarqués sont très souvent soumis à des contraintes dites non fonctionnelles, concernant par exemple l'occupation mémoire, la consommation d'énergie, ou les paramètres de sûreté de fonctionnement (sûreté, fiabilité, disponibilité...).

- **Robustesse, sécurité et fiabilité:**

  L'environnement est souvent hostile, pour des raisons physiques (chocs, variations de température, impact d'ions lourds dans les systèmes spatiaux,..) ou humaines (malveillance). C'est pour cela que la sécurité -au sens de la résistance aux

malveillances- et la fiabilité -au sens continuité de service- sont souvent rattachées à la problématique des systèmes embarqués.

- **Criticité:**

  Les systèmes embarqués sont souvent critiques. En effet, comme un tel système agit sur un environnement physique, les actions qu'il effectue sont irrémédiables.

  Le degré de criticité est fonction des conséquences des déviations par rapport à un comportement nominal, conséquences qui peuvent concerner la sûreté des personnes et des biens, la sécurité, l'accomplissement des missions, la rentabilité économique...

Pour mieux comprendre la part croissante des logiciels embarqués dans la valeur des produits, on peut citer quelques exemples [04] :

- En moyenne, un tiers du coût global d'un avion est aujourd'hui lié aux systèmes embarqués dont 40% est du développement de logiciels.
- Environ 20% du coût d'une automobile vient de la conception et de la réalisation des systèmes embarqués. Un véhicule peut regrouper aujourd'hui jusqu'à 70 ECU (Electronic Control Unit). Le système embarqué a également un effet de levier important pour la valeur ajoutée qu'il apporte ; le succès commercial d'une automobile dépend de plus en plus de la qualité des systèmes embarqués et de l'offre de services à l'utilisateur.
- En 2005, 142,5 millions de téléphones portables soit 84% des téléphones vendus en Europe disposaient de fonctionnalités intégrées d'appareil photo.

Ces exemples montrent combien les logiciels embarqués sont de plus en plus un enjeu de compétitivité pour l'industrie. A propos de Syntec informatique [05], Le marché des logiciels et systèmes embarqués dans son ensemble représente 4 350 M€ pour l'année 2007 (850 M€ pour les logiciels embarqués - licence et maintenance - et 3 500 M€ pour les services autour des systèmes embarqués) et emploie au total 220 000 personnes (soit plus que l'industrie automobile).

Plusieurs études faites sur le marché du système embarqué en France qui présente l'évolution et le déploiement de ce domaine et leur influence dans la vie quotidien, et les compétences autour des logiciels embarqués on peut noté par exemple :

OPIIEC [06] qui donne comme résultat de ces études que la répartition en nombre et par secteurs des industriels interrogés est la suivante (figure 1.3):

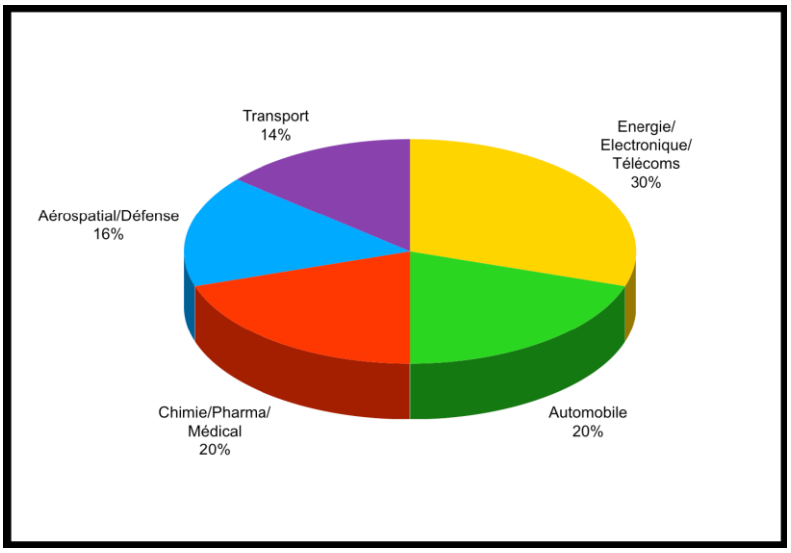


Figure 1.3. Répartition en nombre et par secteurs des industriels [06]

Autre étude [07] présente la situation et la répartition de l'emploi dans les systèmes embarqués (figure 1.4)

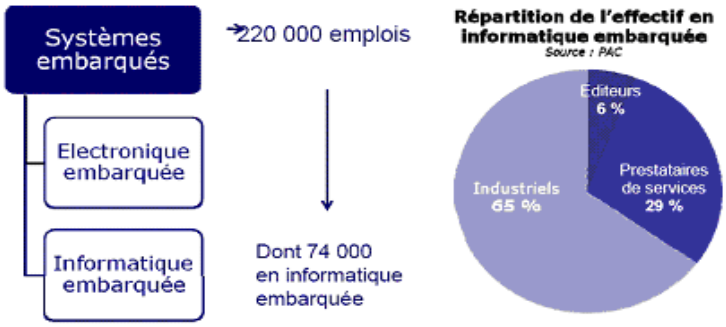


Figure 1.4.Situation et réparation de l'emplois dans les systèmes embarqués 2007 [07]

Il donne aussi comme prévention concernant l'évolution quantitative des emplois en informatique embarquée 2007-2012 :

- Un développement du secteur embarqué qui génère des besoins importants : 34 000 emplois supplémentaires environ en 5 ans (soit +46 %)
- Des besoins globalement plus importants chez les sociétés de service et les éditeurs que chez les industriels

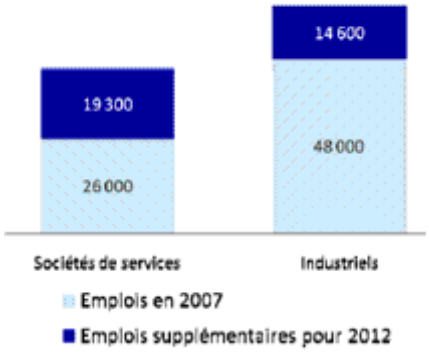


Figure 1.5 Evolution quantitative des emplois en informatique embarqués 2007-2012 [07]

1.2.4 L'art de bien concevoir un système embarqué

Du point de vue technique, la conception d'un système embarqué demande à son concepteur d'être pluridisciplinaire : électronique, informatique, réseaux, sécurité… Mais le concepteur se doit aussi d'être un bon gestionnaire car concevoir un système embarqué revient finalement à un exercice d'optimisation : *minimiser les coûts de production pour des fonctionnalités optimales*.

Le système embarqué se doit d'être :

- Robuste.
- Simple. La simplicité est gage de robustesse.
- Fiable.
- Fonctionnel. Le système doit toujours fonctionner correctement.
- Sûr surtout si la sécurité des personnes est en jeu.
- Tolérant aux fautes.

D'autres contraintes sont aussi à prendre en compte :

- L'encombrement.
- Le poids.
- Le packaging : difficulté de faire cohabiter dans un faible volume, électronique analogique, électronique numérique et RF sans interférences.
- L'environnement extérieur.
- La consommation électrique. Le système embarqué nomade doit être faible consommation car il est alimenté par des batteries. Une consommation excessive augmente le prix de revient du système embarqué car il faut alors des batteries de plus forte capacité.

- Le coût. Beaucoup de systèmes embarqués sont fabriqués en grande série et doivent avoir des prix de revient extrêmement faibles.
- Le temps de développement. Dans un marché concurrentiel et de niches, il convient d'avoir un système opérationnel le plus rapidement possible pour être le premier sur le marché.

Devant toutes ces contraintes, le concepteur adopte des règles de bon sens :

- Faire simple.
- Utiliser ce que l'on a déjà fait ou fait par d'autres. On appellera design reuse.
- Ne pas se jeter sur les technologies dernier cri. Quelle est leur pérennité dans le temps ?
- Ne pas se jeter sur le dernier composant sorti surtout s'il est grand public. Quelle est sa pérennité dans le temps surtout s'il l'on travaille pour la défense où l'on demande une maintenance sur 30 ans !
- Utiliser des technologies éprouvées qui ont fait leur preuve. Ces technologies peuvent d'ailleurs avoir plusieurs générations de retard par rapport à leurs homologues grand public. Pour le grand public, le concepteur de systèmes embarqués peut sembler faire de l'inertie face aux nouvelles technologies mais il faut le comprendre : c'est pour le bien du système qu'il conçoit surtout si la sécurité des personnes est en jeu... Cela explique en partie le décollage difficile des logiciels libres et de Linux pour l'embarqué. Mais ceci est oublié, la déferlante logiciels libres balaie une à une toutes les réticences.

1.2.5 De la nécessité de vérifier les systèmes embarqués

De nombreux systèmes embarqués ont un impact direct sur la sûreté et la sécurité des biens et des personnes. Une part essentielle des fonctions critiques de nombreux systèmes industriels est en effet assurée par des logiciels, notamment de contrôle/commande ou des systèmes de sécurité (cryptage, identification, etc..), qui doivent atteindre un niveau de fiabilité maximale.

Si l'on prend l'exemple d'un avion de ligne moderne gros porteur, une part majoritaire du temps de vol est assurée par un pilote automatique, gros logiciel de plusieurs centaines de milliers de lignes de code qui doit être exempt d'erreurs et obéir à des règles extrêmement strictes de certification. D'autres logiciels critiques comme le contrôle/commande de centrales nucléaires, les systèmes d'aiguillage ou de protection contre la survitesse des

trains, ou plus près de nous les logiciels enfouis dans les ABS ou les airbags doivent tous obéir à des exigences de fiabilité et de sûreté draconiennes.

Ces quelques exemples plaident en la faveur de la certification des systèmes informatiques. La solution idéale consiste à vérifier ces systèmes, c'est à dire, à examiner chacun de leurs comportements et à s'assurer qu'ils satisfont les propriétés (dites comportementales) souhaitées.

1.2.6 Exemples des systèmes embarqués

Les systèmes embarqués sont aujourd'hui enfouis dans la majeure partie des équipements du quotidien et concernent la quasi-totalité des secteurs d'activité: le transport automobile et aéronautique, le spatial, la défense, la santé, l'industrie, l'électronique grand public, les télécommunications, l'agriculture… Le logiciel prend également de plus en plus de place dans ces systèmes avec des enjeux très importants d'intégration au matériel (hardware) tout en respectant les contraintes très fortes auxquelles sont soumis ces systèmes.

Pour donner quelque exemples des systèmes embarqués, on peut prendre le résultat de la **deuxièmes Assies Franco-Allemandes de l'Embarqué**[1] qui a été organisé le 9 juin 2009 par le Syntec Informatique et CAP'TRONIC et avec le soutien de la Direction Générale des Entreprises (DGE) du Ministère de l'Economie, de l'Industrie et de l'Emploi. On peut cité quelque exemple des trophées de l'Embarqué :

– **Trophée Grand Public**

➢ **CAIRPOL: Observ'Air, appareil miniature de mesure en temps réel de la pollution de l'air**

Les alertes à la pollution sont transmises trop tard à la population et sont imprécises quant à leur situation géographique. Plus de 26 000 personnes souffrent en France d'asthme grave dont 50% d'enfants (2,7 millions dans le monde) et sont particulièrement sensibles à l'imprécision de ces alertes. Pour répondre à cette problématique la société CAIRPOL[2] a mis au point l'Observ'Air qui mesure là où on se trouve et en continu la teneur en Ozone et NO2.

Figure 1.6 Observ'Air

[1] *http://www.embedded-symposium.eu/page/*
[2] *http://www.cairpol.com/*

L'Observ'Air s'adresse également aux autres asthmatiques (3 millions en France, 300 millions dans le monde) et aux sportifs soucieux de faire leur jogging dans de bonnes conditions. Le projet a nécessité de développer de nouveaux composants originaux à très basse consommation (circuits, micro-ventilateurs…), de fiabiliser les mesures dans des environnements très variables (avec des tests terrain très rigoureux) et de mettre en œuvre une interface aisément compréhensible et non stressante (pictogrammes, leds couleurs…)(mémorisation jusqu'à un an). Le produit est en cours de déclinaison pour les autres gaz notamment les COV (qualité de l'air intérieur dans le cadre du projet Vaicteur Air2). Le développement d'un détecteur miniature de particules et de pollens vient d'être initié.

– **Trophée du Capteur Embarqué**

➢ **KARVER: SMART-BLOCK « poulie communicante » : connaître, en temps réel, les efforts sur les bateaux de course pour plus de performance et plus de sécurité**

Figure 1.7 KARVER: Smart-Block

La société KARVER[1], qui équipe les bateaux de course les plus prestigieux (80% des bateaux du dernier "Vendée globe" étaient équipés de produits KARVER), a souhaité moderniser ses équipements (accastillage marine) en y introduisant plus d'intelligence. Elle a développé le système Smart Block qui intègre la capture d'efforts aux endroits stratégiques d'un bateau, par l'introduction de capteurs au sein de poulies ou emmagasineurs de voiles. L'électronique et le logiciel embarqués dans les poulies apportent ainsi la mesure et la transmission sans fil et en temps réel des efforts. Cette solution permet très rapidement à un équipage d'optimiser une voilure et donc d'augmenter la performance en mer.

[1] *http://www.karver-systems.com/*

- **NAV ON TIME: Système MOW-BY-SAT de tondeuse automatique guidée par GPS au centimètre près**

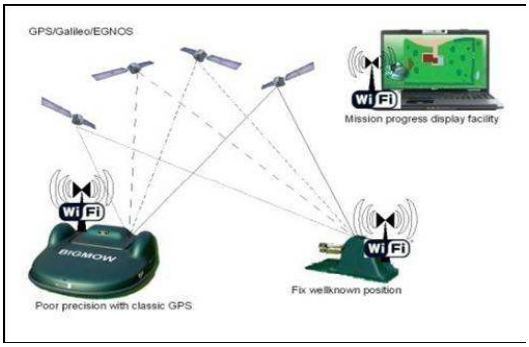


Figure 1.8 Système MOW-BY-SAT de tondeuse automatique

Les tondeuses de gazon autonomes existent déjà mais elles vous imposent d'enterrer un cable électrique pour délimiter votre terrain. Et la machine ne connait que cette limite figée pour effectuer son travail. La société NAV ON TIME[1] a mis au point le système MOW-BY-SAT basé sur deux modules communicants. L'un intégré à la machine, l'autre intégré à la station de recharge. Il n'est plus nécessaire d'enterrer un câble et la zone à tondre peut être définie directement depuis un PC ou un téléphone portable.

Grâce aux fonctions de communication, le logiciel embarqué effectue un calcul GPS différentiel entre la base et la machine pour une localisation précise au centimètre. Il devient alors possible de piloter le robot de tonte en automatique de manière très précise sur un fairway de golf ou toute autre jardin public ou privé. Le "green keeper" ou le propriétaire du terrain peut également vérifier à tout instant que la machine fait son travail, et peut, par exemple selon la météo, ordonner à la machine de s'arrêter ou de ne tondre qu'une partie de la surface...

- **Trophée Communicant**

- **Les nominés : E-STERESYS - NAVOCAP_ E-STERESYS pour sa radio Diabolo, terminal radio numérique et internet avec écran tactile LCD multiservices**

Figure 1.9 Radio Diabolo

[1] *http://www.navontime.com*

L'arrivée de la radio numérique terrestre va donner un sacré coup de vieux à notre fidèle poste de radio. Et pourquoi ne pas en profiter pour repenser les services rendus par ce dernier ? La société E-Steresys[1] a ainsi conçu le premier terminal de radio numérique terrestre français. Du fait de sa reconnaissance et de son exploitation sophistiquée de la radio diffusion numérique (au standard TDMB) et de sa connectivité Internet par WIFI, "Diabolo", la première "kitchen radio mobile device de l'habitat" permet un pont entre le monde du Broadcast (radio numérique) et le monde du Broadband (Internet). Diabolo donne ainsi accès, grâce à sa connectivité et à son écran LCD, à des services simplifiés et faciles d'usage dans l'environnement de la maison sans qu'il soit nécessaire d'allumer son PC ou sa télévision.

- **NAVOCAP pour son service ANGEO d'aide à l'orientation destiné à rassurer les malvoyants et non-voyants qui souhaitent recouvrir leur autonomie dans leurs déplacements**

Cet ensemble de services ont été développés en collaboration avec le CNRS, le CNES ainsi que les associations de malvoyants. L'offre est basée sur deux piliers : un équipement embarqué intégrant un système de localisation innovant 3 à 4 fois plus précis en ville que tout récepteur GPS standard, et qui permet ainsi de connaître à coup sûr le côté de la rue parcourue. Par ailleurs, afin de libérer les mains, l'interfaçage homme machine est réalisé à partir de reconnaissance et synthèse vocales. Un centre d'appel pour toute demande d'informations complémentaires (exemple: l'arrêt de bus le plus proche, calcul d'un nouvel itinéraire, etc...)

Trois modes de navigation possibles : le mode « reconnaissance » qui permet d'être orienté dans les sens aller et retour en toute autonomie sur un parcours préenregistré avec une très grande précision. Ce mode est utilisé sur des parcours non cartographiés, pour retrouver une adresse précise ou en surveillance passive activable en cas d'imprévu. Le mode «aventure » qui assure le guidage sur des itinéraires inconnus et préparés à l'aide du logiciel de calcul d'itinéraire ANGEO Home[2].

Le mode « suivi » qui permet l'accès à une plateforme d'accueil humanisée apte à fournir tout type d'information liée au déplacement en cas de nécessité et de géolocaliser en temps réel l'appelant. L'ensemble embarqué se porte à la ceinture, est simple d'utilisation et dispose d'une autonomie de 8 heures.

[1] *http://www.e-steresys.com/*
[2] *http://www.angeo-technology.com/*

– **Trophée de l'Embarqué Critique**

➢ **RENAULT: châssis « 4Control » à 4 roues directrices**

Le système 4Control développé par Renault[1] permet de moduler l'angle de braquage des roues arrière en fonction de la vitesse. En dessous de 60 km/h, les roues arrière braquent dans le sens opposé à celles du train avant pour une plus grande agilité.

Le rayon de braquage réduit facilite également les manoeuvres. Les mouvements de volant réduits favorisent la maniabilité sur route sinueuse, dans un grand confort de conduite.

Au-delà de 60 km/h, les roues arrière braquent dans le même sens que les roues avant. En courbe, le train arrière s'inscrit ainsi parfaitement dans la trajectoire, pour une sérénité optimisée.

Figure 1.10 Le système 4Control développé par Renault

Couplé aux systèmes électroniques ESP, ce châssis à quatre roues directrices apporte davantage de sécurité lors de conditions de freinage difficiles, mais surtout lors des manoeuvres d'évitement. La mise en oeuvre de ce système éminemment critique s'est notamment appuyée sur une redondance et une surveillance des calculs effectués par le calculateur, sur la sécurisation des échanges entre les calculateurs 4Control et le calculateur ESP, et sur la sécurisation des mesures physiques utilisées grâce à des capteurs spécifiques et à des algorithmes de surveillance. Ce châssis équipe actuellement les Renault Laguna GT et coupé avec plusieurs milliers de véhicules vendus.

➢ **GEENSYS pour son outil ARTOP (AutosaR TOol Platform)**

Créé en 2003, le consortium AUTOSAR[2], rassemble aujourd'hui au plan mondial l'ensemble des acteurs du secteur automobile: constructeurs, équipementiers de rang 1 et 2, éditeurs d'outils, sociétés de service et organismes de recherche. Il s'est fixé l'objectif de standardiser en 9 ans une architecture logicielle normalisée pour les systèmes électriques et électroniques automobile, permettant la réutilisation de tous les logiciels applicatifs embarqués dans le véhicule et leur indépendance totale de l'architecture matérielle sous-jacente.

[1] *http://www.renault.com/*
[2] http://www.geensys.com/

Dans le domaine de voitures & transports : La RDTL (Régie Départementale des Transport Landais) en collaboration avec Geensys créent e-Bus, une plateforme ouverte qui répond aux nouvelles attentes des exploitants de réseaux de transport.

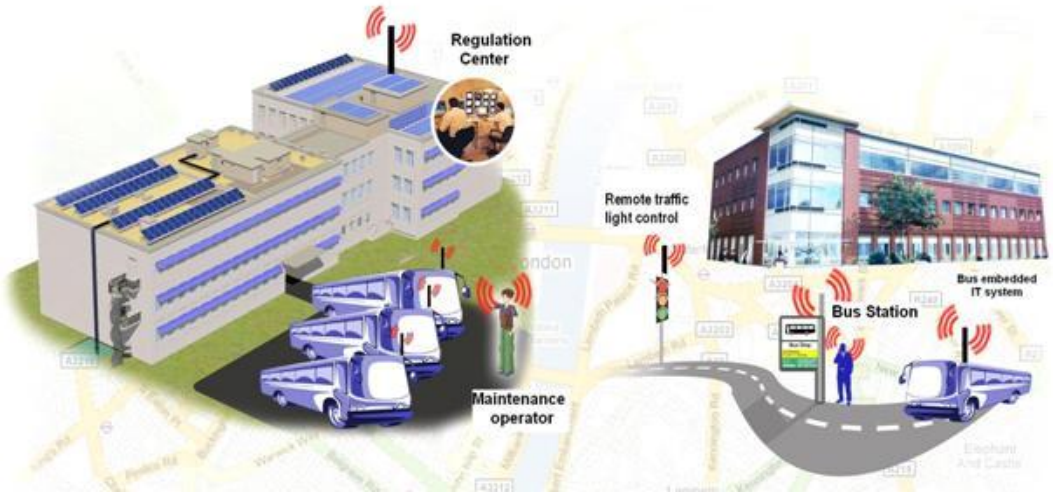


Figure 1.11 E-nove, une plateforme ouvert pour les application de transport intelligent [91]

L'ensemble des bus ainsi que le back office de l'exploitant constituent un système en ce sens que chaque bus est susceptible de communiquer avec la station fixe.
Le système e-Bus est constitué d'une plate-forme logicielle ouverte bâtie sur des standards (OSGi et les services Web) adossée à une plate-forme matérielle (PC embarqué équipé d'un écran tactile ; entrées/sorties vers des capteurs/actionneurs sur le bus et aussi vers des moyens de communications – WiFi et GPRS – vers la station fixe).

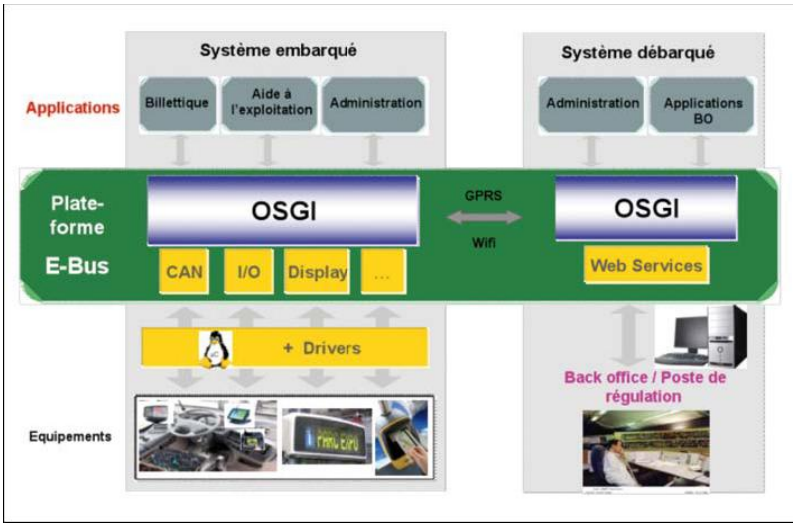


Figure 1.12 Plate-forme e-Bus

Pour l'Assistance et localisation, Dreamap SaS développe deux solutions : VigeoLife et VigeoCare.

La première propose aux seniors nomades un service d'assistance médicalisé 24hx24h, grâce à la géolocalisation GMS et GPS rendue possible par un appareil téléphonique mobile à l'utilisation simplifié (une touche SOS déclenchant un appel vocal vers l'assistance médicale ainsi qu'une localisation par sms et 4 touches d'appel préprogrammées vers des proches).

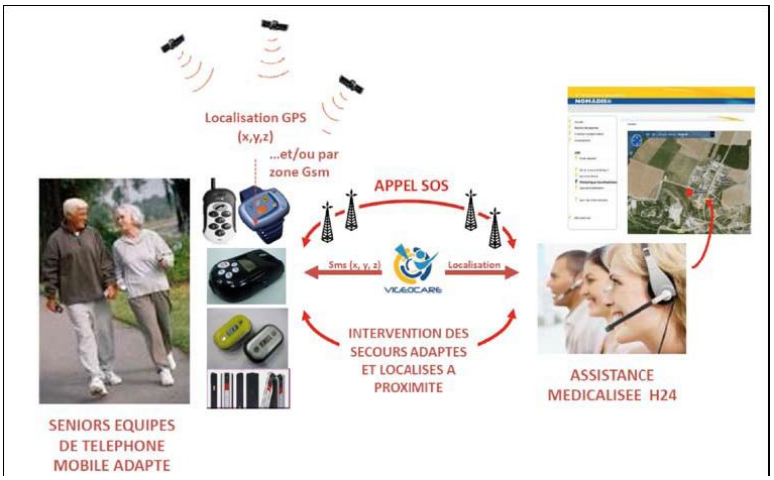


Figure 1.13 VigeoLife: assistance médicalisé

La seconde, VigeoCare, a été conçue spécifiquement pour assister les personnes dépendantes, atteintes par exemple de troubles de la mémoire (Alzheimer). Un bracelet-montre Gsm/GPS et doté d'un bouton SOS permet l'assistance et la localisation. Il est également possible de suivre les déplacements dans les zones de vie de la personne et – grâce à des capteurs spécifiques (comportement, physiologiques) – de détecter les situations critiques.

- **le coeur artificiel** : Start-up créée et financée par Truffle Capital[1], Oséo, EADS et la fondation du professeur Carpentier, Carmat Sas a pour objectif de proposer à la communauté médicale un coeur artificiel total implantable.[81]

Figure 1.14 Le coeur artificiel

Actuellement au stade des essais précliniques, cet appareil piloté de manière autonome par des systèmes embarqués apporte des fonctionnalités similaires à celles d'un coeur naturel,

[1] *http://www.truffle.com/*

sur les plans anatomique et physiologique. Il permet une régulation automatique des débits et fréquences cardiaques en fonction des besoins physiologiques du patient, lors d'efforts physiques notamment.

1.3 Conclusion

C'est une véritable révolution que celle des systèmes embarqués, à la fois pour l'informatique qui voit s'ouvrir des champs d'applications de plus en plus vastes, mais également pour toutes nos sociétés, en raison de leurs impacts dans les domaines technologiques, économiques, industriels et environnementaux.

Les systèmes embarqués sont des composants électroniques (des puces) équipés de logiciels, intégrés dans des appareils ou des équipements. On les trouve dans de multiples objets de la vie quotidienne (téléphones, électroménager, jeux, loisirs numériques, équipement électrique …), auxquels ils confèrent de nouvelles fonctionnalités en termes d'usage, d'intelligence et de communication. Ils se retrouvent également dans les secteurs où la fiabilité, la réactivité et la robustesse sont cruciaux comme le transport (ferroviaire, aérospatiale et défense, automobile, énergie…), la santé ou le bâtiment.
Vu l'importance des systèmes embarqués comme nous l'avons montrée dans ce chapitre, il est nécessaire d'assurer la sécurité de ces systèmes, de leurs données et des autres applications qui peuvent être présentes.

Dans le prochain chapitre, nous allons nous concentrer sur l'aspect de la sécurité des systèmes embarqués en tant que défi.

Chapitre 2

Systèmes embarqués sécurisés : un défi

Résumé

Dans ce chapitre, nous présenterons les concepts de base de ce que l'on appelle "les exigences de sécurité" dans le domaine de l'embarqué. Nous mettrons le point sur la difficulté de développement dans le monde des systèmes embarqués souvent sous strict ressources contraintes. Enfin, nous donnerons des exemples de quelques accidents de systèmes embarqués critiques. Ce qui permettra de mettre en évidence les principaux enjeux sécuritaires qui ont motivé notre travail.

Sommaire

2.1 Introduction

Les systèmes embarqués sont de plus en plus connectés, au cœur d'infrastructures informatiques, et accèdent à des services en ligne et bases de données. De plus, ces systèmes doivent répondre à des contraintes fortement liées à leur environnement ou à leur fonctionnalité propres et ne peuvent se permettre d'avoir des failles. La question de la sécurité informatique se pose donc lors du développement d'applications et de systèmes embarqués.

Ce chapitre va présenter l'aspect de sécurité en tant qu'un défi pour les systèmes embarqués, et d'illustrer la façon dont les exigences de sécurité se traduisent par la conception du système défis.

2.2 Les objectifs de la sécurité

La sécurité numérique brigue trois objectifs : la confidentialité, l'intégrité et la disponibilité des ressources, des informations et des systèmes :

- la **confidentialité**, vise à assurer que seuls les sujets (les personnes, les machines ou les logiciels) autorisés aient accès aux ressources et aux informations auxquelles ils ont droit. La confidentialité a pour objectif d'empêcher que des informations secrètes soient divulguées à des sujets non autorisés. L'objectif des attaques sur la confidentialité est d'extorquer des informations ;
- l'**intégrité** vise à assurer que les ressources et les informations ne soient pas corrompues, altérées ou détruites par des sujets non autorisés. L'objectif des attaques sur l'intégrité est de changer, d'ajouter ou de supprimer des informations ou des ressources ;
- la **disponibilité** vise à assurer que le système soit bien prêt à l'emploi, que les ressources et les informations soient en quelque sorte consommables, que les ressources ne soient pas saturées, que les informations, les services soient accessibles et que l'accès au système par des sujets non autorisés soit prohibé. L'objectif des attaques sur la disponibilité est de rendre le système inexploitable ou inutilisable.

2.3 La sécurité des systèmes embarqués

Les systèmes embarqués ou les logiciels enfouis (*embedded systems or software*), sont des entités autonomes qui remplissent une mission indépendante, parfois critique, sans intervention humaine, en général en interaction directe avec l'environnement extérieur que celui-ci soit physique ou informatique. Ces systèmes sont soumis à des contraintes

fonctionnelles qui mettent en jeu leur définition, leur robustesse, leur conception, leur capacité à accomplir une tâche avec des ressources déterminées souvent liées aux astreintes temporelles ou à la consommation énergétique.

Le comportement de ces systèmes doit être blindé, voire garanti avec un haut niveau de sécurité et de sûreté, pendant tout leur cycle de vie. Ces systèmes doivent être protégés, mais ils doivent avant tout fonctionner correctement tant dans leurs fonctions purement intestines que dans leurs interactions avec le monde extérieur.

Avec la diffusion massive des capteurs et des actuateurs électroniques, les systèmes complexes bénéficient d'une instrumentation importante. Les échanges d'information avec l'extérieur et la faculté d'adaptation de ces systèmes à l'environnement nécessitent des contrôles sévères sur la bonne marche de ces systèmes avec des modules flexibles, autochargeables, autoconfigurables. Les systèmes embarqués ou les logiciels enfouis doivent donc posséder des propriétés de sécurité et de sûreté de fonctionnement, localement, puisque ces systèmes autonomes doivent résister seuls à des sollicitations imprévisibles de leur entourage. Sur la base des informations fournies par l'instrumentation en place et des connaissances disponibles par des modèles mathématiques, physiques et de sécurité, il s'agit d'implanter localement une carapace de sécurité et de sûreté, voire de résilience pour d'abord percevoir (observer, détecter, localiser, diagnostiquer), analyser la situation en fonction des événements imprévus, des occurrences aléatoires, des dangers et réagir (corriger, se cicatriser, tolérer, se maintenir, s'adapter, se dégrader, survivre en autarcie, voire s'anéantir) en fonction des évolutions et des déviations par rapport à un état ou un comportement de référence normal, souhaitable ou nominal.
La sécurité et la sûreté des systèmes et logiciels embarqués se construisent donc en plusieurs phases :

- pendant la spécification du système : maîtrise de sa complexité, de son architecture et claire séparation entre le module autonome et son infrastructure en considérant l'énergie consommée et les flux de communication : application traditionnelle des méthodologies pour la spécification de sécurité et de sûreté;
- pendant la conception du système : utilisation de techniques de modélisation, d'abstraction, méthodes mathématiques,…;
- pendant l'implantation : épargne rigoureuse des ressources, parcimonie dans les communications avec les périphériques, …;

- pendant la validation : vérification formelle, simulation et test, évaluation de l'assurance de sécurité et de sûreté.

La conception du système doit prendre en compte les aspects mathématiques, informatiques, électroniques et architecturaux, les aspects de normalisation et de standardisations (interopérabilité) et les impératifs économiques. La difficulté porte en général sur l'assemblage hétérogène de composants et de modules.

2.4 Attaques sur le système embarqué : le défi d'aujourd'hui

Plusieurs définitions du code malicieux sont données dans la littérature. A titre d'exemple, McGraw et Morrisett [26] définissent le code malicieux comme étant tout code ajouté, modifié ou enlevé à un logiciel dans le but de causer intentionnellement des dommages ou de porter atteinte au fonctionnement normal du système.

Dans [27], Robert Charpentier et Martin Salois adoptent une définition plus large du code malicieux en le considérant comme un fragment de code qui affecte ou permet à un autre programme d'affecter l'intégrité, les données, le flot de données et de contrôle ou le fonctionnement d'un système. Cette définition est plus large car elle inclut dans le code malicieux des problèmes qui sont exclus de la définition précédente telles que les failles de programmation.

Les attaques peuvent être classifiées aussi dans deux larges catégories : attaques physiques et *side-channel*, et attaques logiques. Les attaques physiques et *side-channel* [28] se réfèrent aux attaques qui exploitent l'implémentation du système et/ou identifient les propriétés de l'implémentation. La Figure 2.1 cite les diverses catégories des techniques utilisées pour attaquer un dispositif.

Des attaques physiques et *side-channel* sont généralement classifiées en des attaques *invasives* et *non-invasives*. Les attaques invasives ont pour but d'obtenir l'accès au dispositif pour observer, manipuler, et interférer dans les systèmes internes. Puisque les attaques invasives exigent typiquement une infrastructure relativement chère, elles sont assez difficiles à déployer. Les exemples des attaques invasives incluent *micro-probing* et rétro-ingénierie (*reverse engineering*) de la conception. Il y a beaucoup de formes d'attaques non invasives telles que des attaques de synchronisation, des techniques d'induction de faute, des attaques basées sur l'analyse électromagnétique et l'analyse de consommation d'énergie.

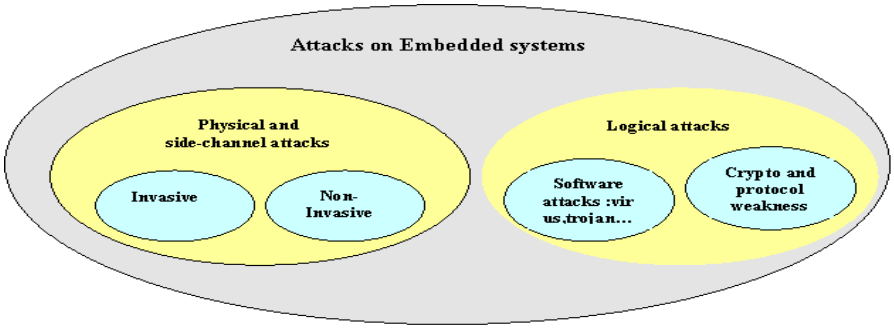


Figure 2.1 Les attaques sur les systèmes embarqués [73]

Les attaques logiques sont faciles à déployer contre les capacités d'exécuter les applications téléchargées, et exploitent des faiblesses ou des bogues dans l'architecture globale (matériel/logiciel) aussi bien que des problèmes dans la conception de l'algorithme cryptographique ou du protocole de sécurité. Dans la pratique, les attaquants utilisent souvent une combinaison des plusieurs techniques pour atteindre leurs objectifs. Par exemple, pour un smartphone, du point de vue de l'attaquant, les objectifs visés sont : de rendre inutilisable le téléphone portable; de modifier le comportement du téléphone; d'accéder aux données sensibles; d'émettre des appels voix ou données à l'insu de l'utilisateur; d'outrepasser les politiques de gestion des droits numérique.

2.4.1 Attaques logicielles

Les attaques logicielles représentent une menace importante pour les systèmes embarqués qui sont capables de télécharger et d'exécuter des applications. Par rapport aux attaques physiques et *side-channel*, les attaques logicielles exigent typiquement une infrastructure qui est facilement disponible. Ces attaques sont implémentées par des agents malveillants comme les virus, les vers, les chevaux de Troie, etc. (voir section 2.4.2), et ils peuvent exploiter des failles dans l'OS ou le logiciel d'application, obtenir l'accès aux systèmes internes, et perturber ses fonctions normales afin de manipuler des données ou des processus sensibles (attaques d'intégrité), relever des informations personnelles, et/ou dénier l'accès aux ressources de système (attaques de disponibilité),

Des agents malveillants effectuent des attaques logiciels par l'exploitation des faiblesses dans l'architectures du système final sont étudiés dans [87,86]. Ils se présentent typiquement en raison des imperfections dans le logiciel, qui peuvent être nommés comme vulnérabilités.

Le problème de débordement de mémoire (*buffer overflow*) est un trou commun dans les systèmes d'exploitation et les logiciels d'application, qui peut être exploité dans les attaques logicielles. Le problème peut avoir lieu à cause de la vérification réduite des limites des tampons. Les effets de débordement de mémoire peuvent réécrire (*overwriting*) la mémoire de la pile, les tas, et les pointeurs de fonction. L'attaquant peut utiliser le débordement de mémoire pour réécrire les adresses du programme stocké tout près, ceci lui permet de transférer le contrôle au code malveillant [30,31], qu'une fois exécutée, peut avoir des effets indésirables.

2.4.2 Quelques définitions

- **Bombes logiques** :

  Une bombe logique [92] est un programme, ou une partie d'un programme, qui est en sommeil jusqu'à ce qu'une partie spécifique de la logique de programme est activé. Le plus souvent, l'activateur pour une bombe logique est une date. La bombe logique vérifie la date du système et ne fait rien jusqu'à ce qu'une pré-programmé la date et l'heure est atteinte. À ce moment-là, la bombe logique active et exécute son code.

  L'utilisation classique d'une bombe logique est d'assurer le paiement pour les logiciels. Si le paiement n'est pas effectué par une certaine date, la bombe logique est activée et le logiciel se supprime automatiquement lui-même. La bombe logique peut également supprimer d'autres données sur le système.

- **Vers**

  Un ver (worm en anglais) [33] est un programme qui peut s'auto-reproduire et se déplacer à travers un réseau en utilisant les mécanismes réseau, ils n'ont pas réellement besoin d'un support physique ou logique (disque dur, programme hôte, fichier, etc..) pour se propager; un ver est donc un virus réseau.

- **Virus**

  Un virus [90] est un morceau de programme informatique malicieux conçu et écrit pour qu'il se reproduise. Cette capacité à se répliquer, peut toucher votre ordinateur, sans votre permission et sans que vous le sachiez. En termes plus techniques, le virus classique s'attachera à un de vos programmes exécutables et se copiera systématiquement sur tout autre executable que vous lancez. Il n'y a pas de génération spontanée de virus informatiques. Ils doivent avoir été écrits dans un but spécifique.

A part se répliquer, le virus peut avoir ou non une action plus ou moins néfaste, allant de l'affichage d'un simple message à la destruction de toutes les données.

– **Chevaux de Troie**

Un cheval de Troie [35] est considéré comme un logiciel ou morceau de logiciel - également appelé code exécutable (qui se présente comme un logiciel anodin mais qui en réalité, et de façon très similaire à un virus, a pour vocation d'infecter une machine, dans le but d'ouvrir une porte, à l'insue des utilisateurs. A la différence d'un virus ou d'un ver, un cheval de Troie n'a pas vocation à se reproduire ou à se propager, mais peut cependant être aussi destructeur dans certains cas.

– **Code mobile hostile**

Le code mobile hostile est exécuté à l'ultérieur de pages web. Les applets Java forment un exemple de code mobile. Ce code est exécuté sur un ordinateur sans l'accord de l'usager et peut contenir des actions malicieuses dommageables ou inattendues. Il s'active au moment où la page web est chargée.

– **Portes arrière**

Les portes arrière sont des accès laissés par le programmeur d'un système comme un mot de passe. Ces accès donnent la possibilité de contrôler un ordinateur à distance et d'accéder à ses données ou ressources par l'intrus qui les utilise. Une fois à l'ultérieur, il peut accéder à des données confidentielles ou faire des opérations inattendues.

2.4.3 Classification des Attaquants

Les risques de sécurité peuvent être évalués selon le niveau de préparation de l'attaque, le temps et les moyens nécessaires. Ainsi, une classification des attaquants a été définie par IBM [29]:

- **Classe I** (Clever outsiders): Cette classe regroupe les attaquants très astucieux, utilisant du matériel obsolète et ayant une connaissance très limitée du système. Ils exploitent uniquement les failles existantes.
- **Classe II** (Knowledgeable insiders): Les attaquants de cette classe sont généralement des spécialistes expérimentés ayant une connaissance parfaite du système visé. Très souvent, ils ont accès à des équipements sophistiqués.
- **Classe III** (Funded organizations): Dans cette classe, les attaquants sont des organisations (Multinationales, pays, etc.) capables de mettre sur pied des équipes de spécialistes dans des domaines très variés et complémentaires. Ces groupes de

spécialistes sont capables d'analyser en profondeur le système visé et de concevoir de nouvelles attaques. D'une manière générale, ils ont accès à des équipements ultrasophistiqués et ont des moyens illimités.

2.4.4 Niveaux de sécurité

La norme FIPS140-2 [85] précise les exigences en matière de sécurité qui devront être respectées par le module cryptographique utilisé dans un système de sécurité servant à protéger des informations. La norme fournit quatre niveaux de sécurité:

- **Niveau 1**: C'est le niveau de sécurité le plus bas défini par le standard. Il définit des exigences de sécurité très basiques comme l'utilisation d'un algorithme standard (AES, etc.).
- **Niveau 2**: Outre les exigences du niveau 1, le niveau 2 suppose des mécanismes physiques de sécurité spécifiques. En particulier, une couche de protection doit être placée sur le dessus de la puce afin d'empêcher tout accès physiques non-autorisé.
- **Niveau 3**: Ce niveau de sécurité suppose des mécanismes physiques de sécurité capables de détecter et d'empêcher tout accès aux paramètres de sécurité contenus dans le module cryptographique. En plus des fonctions de sécurité du niveau 2, un réseau de capteurs peut être rajouté pour déclencher la destruction des paramètres de sécurité en cas d'intrusion.
- **Niveau 4**: C'est le niveau de sécurité le plus élevé défini par la norme FIPS140-2. A ce niveau, le module cryptographique doit être enveloppé par un bouclier de protection dont le rôle est de détecter et de réagir à toute tentative d'accès physique. D'autre part, un module cryptographique de niveau 4 doit être robuste aux modifications anormales des paramètres externes du circuit à savoir: les signaux d'alimentation, le signal d'horloge, la température,

2.5 Le développement dans le monde des systèmes embarqués

Un langage de programmation est un code de communication permettant à un être humain de dialoguer avec une machine en lui soumettant des instructions et en analysant les données matérielles fournies par le système. Le langage permet à la personne qui rédige un programme de faire abstraction de certains mécanismes internes tels que le modèle de calcul de la machine qui aboutissent au résultat désiré.

L'activité de rédaction du code source d'un programme est nommée programmation. Elle consiste en la mise en œuvre de techniques d'écriture et de résolution d'algorithmes

informatiques, lesquelles sont fondées sur les mathématiques. À ce titre, un langage de programmation se distingue du langage mathématique par sa visée opérationnelle (une fonction et par extension un programme doit retourner une valeur), de sorte qu'un langage de programmation est toujours un compromis entre la puissance d'expression et la possibilité d'exécution.

Les langages de programmation permettent de définir des ensembles d'instructions effectuées par l'ordinateur lors de l'exécution d'un programme. Il existe plusieurs langages de programmation chacun d'eux utilisant un paradigme particulier de programmation.
Comme nous l'avons vus au premier chapitre, un système est dit embarqué lorsqu'il est destiné à une ou plusieurs tâches précises. Il est dans certains cas une partie d'un produit ou d'un système bien plus gros. Nous en utilisons de plus en plus dans la vie quotidienne : téléphones portables, lecteurs de musique digitale, logiciels de voitures, feux de signalisation routière, télécopieurs, pacemakers, pour ne citer que ceux-là.

Ces systèmes ont pour caractéristique commune d'être contraints. Selon la machine sur laquelle un programme embarqué est destiné à être exécuté, il peut avoir à satisfaire différentes de ces contraintes :

- **L'utilisation mémoire :**
  La miniaturisation du matériel entraîne dans certains cas une réduction des capacités mémoires de l'appareil. Le programme embarqué doit donc tenir compte de cette contrainte et, par exemple, ne pas dépasser certaines limites d'occupation mémoire statique et d'utilisation mémoire lors de son exécution.
- **Le temps de calcul :**
  L'utilisation de programmes embarqués se fait souvent dans des contextes où le temps est un paramètre essentiel du système. Les délais d'exécution sont alors connus et bornés. De tels systèmes ont des propriétés *temps réel* dur (*safety critical*) ou mou (video, ...).
- **La consommation d'énergie :**
  Le caractère embarqué d'un système fait que ce dernier n'est pas toujours relié à une source infinie d'énergie. Le système doit alors gérer une certaine quantité d'énergie fournie par une batterie autonome (panneaux solaires, pile).
- **La sûreté/sécurité :**
  Certaines pannes de systèmes embarqués peuvent avoir des conséquences désastreuses tant d'un point de vue humain (train d'atterrissage d'un avion, appareillage médical)

que d'un point de vue économique (système d'exploitation d'un distributeur d'argent). De tels systèmes sont dits *critiques*.

Un système embarqué peut avoir à satisfaire toutes ou parties de ces contraintes bien qu'elles puissent être contradictoires. L'exemple type est le gain de temps d'exécution d'une boucle ordonnancée par pipeline logiciel. Cette optimisation augmente la taille du code généré et donc son occupation en mémoire. Si un programme devait être optimisé de manière à augmenter sa vitesse et diminuer sa taille, le pipeline logiciel demanderait de faire des compromis. La demande d'optimisation du programme à exécuter est très forte, voire primordiale. Certaines optimisations sont réalisables par le programmeur. Ce dernier peut de lui-même chercher à réduire par exemple le nombre d'exécutions d'un corps de boucle, ou limiter le nombre de variables utilisées.

L'utilisation d'allocation de mémoire dynamique peut permettre de minimiser l'espace mémoire nécessaire à l'exécution du programme. Toutes ces astuces de programmation sont d'autant plus efficaces que le programmeur connaît la plate-forme d'exécution et le compilateur utilisé pour générer le fichier binaire.

2.6 Exigences de sécurité

La composante informatique des systèmes d'aujourd'hui est en forte croissance, et cela parmi des secteurs où les niveaux de sécurité peuvent être élevés. Citons l'automobile, l'aéronautique, le spatial ou encore le nucléaire. Une défaillance logicielle ou matérielle peut conduire à de graves pertes humaines et/ou matérielles. Il est par conséquent crucial de se doter de moyens qui assureront la mise au point de logiciels sûrs. Cette sûreté est appréhendée par différents acteurs qui :

- Par les **concepteur**s : qui doivent assurer que le système qu'ils livrent soit correct par rapport à ce qu'on leur demande
- Par les **intégrateurs** : qui doivent pouvoir contrôler que ce qu'on leur livre est correct et sûr par rapport à ce qui était attendu et que l'intégration ne met pas en cause l'intégrité de l'ensemble.
- Par les **organismes de certification** : qui doivent avoir tous les éléments en main pour permettre la certification des systèmes à certifier.

Cette différence de points de vue sur le même système ajoute de la complexité aux procédures d'élaboration et de mise au point des systèmes embarqués. Afin de clarifier les différentes attentes, des normes sont définies dans différents domaines (les attentes sont souvent liées aux spécificités métier) afin de cadrer les tenants et aboutissants des attentes

en certification. Ainsi, ces dernières années, de nouvelles normes se mettent de place afin de préciser les attentes dans tel ou tel domaine particulier de la production logicielle & matérielle. Dans le contexte de l'automobile, la norme ISO26262 [78] est censée aller dans ce sens.

Dans [38], Barthe présente EVEREST un exemple des études effectués dans la domaine de la vérification pour les systèmes embarqués. Une environnements de vérification et sécurité du logiciel EVEREST, ce projet vise à assurer la sécurité des systèmes pour le code mobile et embarqué. Les domaines d'application privilégies sont les petits objets portables sécurisés, leurs systèmes d'exploitation, et les applications qui s'y exécutent.

En parallèle, les systèmes embarqués sont élaborés selon des processus complexes mêlant différents formalismes depuis les phases initiales de cahier des charges jusqu'au code produit et embarqué. Ces processus ont été élaborés de façon décorrélée des normes de certification car ils sont dans la majeure partie des cas construits de façon empirique en fonction des réalités métier et de l'existant. La prise en compte des normes de certification dans ce contexte demande ainsi une réelle justification des différentes phases du processus de modélisation afin de répondre aux attentes. Une meilleure prise en compte des exigences est donc primordiale afin de s'assurer que l'on produit un système qui soit non seulement correct, mais également que l'on produit le bon système.

Les systèmes embarqués fournissent souvent les fonctions critiques qui pourraient être sabotées par des entités malveillantes. Avant de discuter les exigences communes de sécurité des systèmes embarqués, il est important de noter qu'il y a beaucoup d'entités impliquées dans la chaîne de conception, de fabrication, et d'utilisation d'un système embarqué. Les exigences de sécurité changent selon les perspectives que nous considérons. Par exemple, considérons un smartphone qui est capable de communications de données, de multimédia, et de voix sans fils. Les exigences de sécurité selon les points de vue des différents participants sont comme suit :

Utilisateur final	• Intimité et intégrité des données personnelles • Appels et transactions frauduleux • Perte/vol • Exécution sécurisée des applications téléchargées
Fournisseur de contenu	• Sécurité de contenu, gestion des droits numériques
Fournisseur de service des applications	• Communications sécurisées • Non-répudiation
Fournisseur de service	• Accès sécurisé au réseau
Fabricant de téléphone	• Utilisation frauduleuse de service
Fournisseur de logiciel et de matériel	• Protection des propriétés intellectuelles

Figure 2.2 Les exigences de sécurité pour un Smartphone

Les soucis de l'utilisateur peuvent inclure la sécurité des données personnelles stockées et communiquées par le téléphone, pendant que l'inquiétude du fournisseur de contenu peut être la protection des contenus multimédias fournis au téléphone, et le fabricant de téléphone pourrait en plus s'intéresser au secret du logiciel de propriété industrielle qui réside dans le téléphone. Pour chacun de ces cas, l'ensemble des entités non sûres (potentiellement malveillantes) peut également changer. Par exemple, dans la perspective du fournisseur de contenu, l'utilisateur du téléphone peut être une entité non sûre. Le téléphone mobile sera l'objet de notre étude dans le dernier chapitre de ce mémoire, et sera présenté d'une façon plus détaillée dans le cadre de l'objectif global qui est le vérification du code pour les systèmes embarqués.

La Figure 2.3 cite les exigences de sécurité pour les systèmes embarqués, qui sont décrits comme suit :

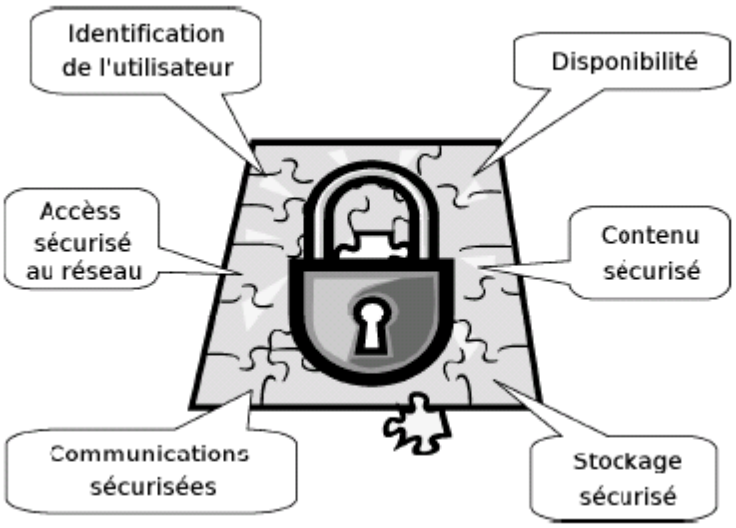


Figure 2.3 Les exigences communes de sécurité de systèmes embarqués [37]

- **Identification de l'utilisateur** indique le processus d'authentifier les utilisateurs avant de leur permettre d'utiliser le système.
- **Accès sécurisé au réseau** fournit une connexion de réseau ou un accès de service seulement si le dispositif est autorisé.
- **Communications sécurisées** se composent d'authentification de communication entres les égaux, d'assurance de confidentialité et d'intégrité des données communiquées, d'empêchement de répudiation d'une transaction, et de protection de l'identité des entités communiqués.
- **Stockage sécurisé** garantit la confidentialité et l'intégrité des informations sensibles stockées dans le système.
- **Contenu sécurisé** impose les restrictions d'utilisation du contenu numérique stocké ou consulté par le système.
- **Disponibilité** s'assure que le système peut exécuter sa fonction attendue et servir les utilisateurs légitimes à tout moment, sans être interrompu par des attaques de déni de service.

Plusieurs primitives fonctionnelles de sécurité ont été proposées dans le contexte de la sécurité du système informatique. Ceux-ci incluent divers algorithmes cryptographiques utilisés pour chiffrer et déchiffrer les données, et pour vérifier l'intégrité des données. En général, les algorithmes cryptographiques [39] peuvent être classifiés en trois classes : chiffrements symétriques (l'expéditeur et le récepteur emploie la même clé secrète pour chiffrer et déchiffrer des données), chiffrements asymétriques (également appelés algorithmes à clés publiques, emploient une clé secrète privée pour le déchiffrage, et une clé publique relative non-secrète pour le chiffrage ou la vérification), et les algorithmes de hachage [40] (fournissent des "signatures" pour des messages et la vérification d'intégrité).

Les solutions de sécurité pour répondre aux exigences de sécurité se fondent typiquement sur les primitives cryptographiques mentionnés ci-dessus, ou sur les mécanismes de sécurité qui emploient une combinaison de ces primitives d'une façon spécifique (par exemple, des protocoles de sécurité). Des technologies et des mécanismes de sécurité ont été conçus autour de ces algorithmes cryptographiques afin de fournir des services de sécurité spécifiques. Par exemple :

- **Les protocoles de sécurité** fournissent des manières d'assurer les canaux de transmission pour le système embarqué. IPSec et SSL [41] sont des exemples

populaires des protocoles de sécurité, largement utilisés pour les réseaux privés virtuels (VPNs) et les transactions web.

- **Les certificats numériques** fournissent des manières d'associer l'identité à une entité, et les technologies biométriques comme la reconnaissance d'empreinte digitale et la reconnaissance de voix aident à l'authentification d'utilisateur final. Les signatures numériques, qui fonctionnent comme équivalents électroniques des signatures manuscrites, peuvent être utilisées pour authentifier la source de données et également pour vérifier son identité.
- **Les protocoles de gestion des droits numériques**, tels que OpenIPMP, MPEG, ISMA, et MOSES, fournissent les cadres sécurisés pour protéger le contenu d'application contre l'utilisation non autorisée.
- **Le stockage sécurisé et l'exécution sécurisée** exigent que l'architecture du système soit adaptée pour des considérations de sécurité. Les exemples simples incluent l'authentification du progiciel qui s'exécute sur le système, l'isolement d'application pour préserver l'intimité et l'intégrité du code et les données associées à une application ou à un processus donnée [42], les techniques matériels/logiciels pour préserver l'intimité et l'intégrité des données dans toute la hiérarchie de mémoire [88], l'exécution du code chiffré [44], etc.

2.7 Embedded software

Dans les systèmes embarqués, le logiciel est une source importante de sécurité du système. Trois facteurs, que nous appelons *Trinity of Trouble :* la complexité, l'extensibilité et la connectivité, concourent à faire de la gestion des risques de sécurité dans le logiciel un défi majeur.

- **Complexité**: Le logiciel est complexe, et deviendra encore plus compliquée dans les années prochaines. Plus de lignes de code accroît la probabilité de bugs et failles de sécurité. Comme les systèmes embarqués convergent avec l'Internet et plus de code sont ajouté, le logiciel du système embarqué devient de plus en plus complexe. Le problème de la complexité est croissante par l'utilisation des langages de programmation non sécurisé (par exemple, C ou C + +) qui ne protègent pas contre les types d'attaques simples, tels que les débordements de buffer (buffer overflows). Pour des raisons d'efficacité, C et C + + sont des langages très populaires pour les systèmes embarqués. En théorie, on pourrait analyser et prouver qu'un petit programme ne

contient pas de problèmes, mais cette tâche est très difficile pour des programmes complexes.

- **Extensibilité**: Les systemes embarqués avancée sont conçus pour être extensible (par exemple, J2ME, Java Card).
- **Connectivité**: De plus en plus de systèmes embarqués sont connectés à l'Internet. A cause de ce haut degré de connectivité, il est possible pour les petites pannes de propager et de provoquer une violation massive de la sécurité. Dans ce cas, un attaquant n'a plus besoin de l'accès physique à un système pour lancer des attaques pour exploiter des logiciels embarqué. L'extension dans ce connectivité signifie qu'il y a plus d'attaques, plus de logiciels embarqués a attaqué, et plus de risques de mauvaises pratiques de sécurité des logiciels.

2.8 Défis de conception d'un système embarqué sécurisé

Dans le monde des systèmes embarqués, les concepteurs d'un système embarqué doivent supporter diverses solutions de sécurité afin de traiter un ou plusieurs des exigences de sécurité décrites ci-dessus. Ces exigences présentent des empêchements dans le processus de conception, qui sont brièvement décrits ci-dessous :

- **Brèches de traitement :** Dans les systèmes avec les ressources modestes de traitement et de mémoire, les demandes élevées de calculs pour les services de sécurité exigent normalement des changements dans la technologie (par exemple, couper le processus de code d'octet Java en deux phases) ou l'utilisation des composants dédiés.
- **Flexibilité :** Un système embarqué est souvent conçu pour répondre aux exigences d'exécution des multiples et diverses protocoles de sécurité et standards pour supporter: des objectifs de sécurité, l'interopérabilité dans différents environnements, le traitement de sécurité dans différentes couches de réseau (par exemple, un PDA permettant le réseau local sans fil pour la connexion au réseau privé virtuel et supportant la navigation sécurisé de web doit exécuter WEP, IPSec et SSL [41]). En outre, avec des protocoles de sécurité constamment visée par des hackers, il n'est pas étonnant qu'ils continuent à se produire de nouvelles attaques. Il est souhaitable de permettre à l'architecture de sécurité d'être assez flexible (programmable) pour adapter facilement aux changements. Cependant, la flexibilité peut également causer des difficultés envers la sécurité.

- **Tamper-résistance :** Les attaques des logiciels malveillants tels que des virus et des chevaux de Troie sont les menaces les plus communes à n'importe quel système embarqué qui est capable d'exécuter des applications téléchargées. Puisque ces attaques peuvent compromettre la sécurité de tous points de vue d'intégrité, des données personnelles, la disponibilité), il est nécessaire de développer et déployer de diverses contre-mesures matériels/logiciels contre ces attaques.
- **Brèches d'assurance *:*** Il est bien connu qu'il est beaucoup plus difficile de construire des systèmes véritablement plus fiables que ceux qui fonctionnent simplement dans la plupart du temps. Les systèmes sûrs doivent pouvoir manipuler des situations qui peuvent se produire par hasard et ils relèvent un plus grand défi : ils doivent continuer à fonctionner sûrement en dépit des attaques réalisées par des adversaires intelligents qui cherchent intentionnellement des modes de défaillances indésirables.

  D'autres facteurs comme : le coût et la batterie, influent également sur l'efficacité des mesures de sécurité d un système embarqué.

2.9 Quelques accidents de systèmes informatisés critiques

2.9.1 Le crash du vol 501

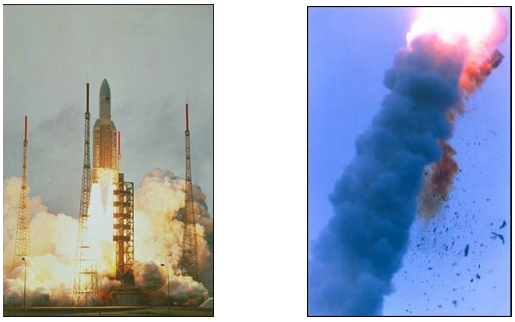
Le 4 Juin 1996, le premier vol du lanceur Ariane V se solda par un échec. Environ 40 secondes après son allumage, la fusée se brisait en deux provoquant ainsi son auto-destruction immédiate (voir le message de John Rushby sur comp.risks, figure 2.4). Le 13 Juin 1996, un groupe indépendant de scientifiques avec à sa tête J. L. Lions, fut nommé pour découvrir l'origine de cette défaillance. Le compte-rendu de ses investigations fut rendu public le 19 Juillet 1998.

```
Newsgroups: comp.risks
Subject: Ariane 5 failure
John Rushby <RUSHBY@csl.sri.com>
Wed 5 Jun 96 14:54:47-PDT

>From cnn's web page www.cnn.com:

Faulty computer blamed in Ariane rocket failure

Experts studying the moments before the Ariane-5 rocket explosion say
faulty computer software may be to blame for the rocket veering off
course. Apparently, the rocket was misfed information that made it think
it was not following the right path. The rocket then changed direction,
causing the upper part to began to break apart.
```

Figure 2.4 Première annonce suspectant un problème logiciel sur Ariane-5

En tout premier lieu l'erreur logicielle provient d'un module qui a été conçu et vérifié pour Ariane IV, mais qu'on avait omis de vérifier pour Ariane V [13]. Ce module contenait une

fonction utile uniquement lorsque la fusée était sur sa plate-forme de lancement et qui aurait due être désactivée ensuite. Or, cette fonction, non seulement était active pendant le vol, mais c'est elle qui causa la perte d'Ariane V.



(a) Avant **(b) Après**

Figure 2.5 L'accident du vol inaugural de la fusée Ariane 5

Les conclusions du rapport sur la défaillance d'Ariane V portent en grande partie sur la méthodologie utilisée pour la qualification et la validation des systèmes informatiques utilisés qui sont jugées insuffisantes.

2.9.2 La banque de New York.

Le 21 novembre 1985, une nouvelle version du logiciel de traitement des bons du Trésor fut chargé sur l'ordinateur de la Banque de New York [09]. Pour une variable entière passant malencontreusement de 32 768 à 0 :

- la crise des bons du Trésor.
- un emprunt de 20 milliards de dollars de la banque au gouvernement.
- les intérêts de l'emprunt.

Le problème venait en fait du compteur qui dénombrait les demandes d'achat ou de vente sur les bons du Trésor. Habituellement codé sur 32 bits, celui du nouveau programme avait été codé sur 16 bits. Evidemment, lorsque la 32.768^{ème} demande d'achat ou de vente est arrivé dans la file, le compteur a indiqué zéro et 32.768 demandes sont restées ignorées du système.

2.9.3 Le Therac-25.

Le Therac-25 [14] est un appareil médical doté d'une interface convivial pour le traitement des cancers. Il émet un faisceau soit d'électrions (tissus superficiels), soit de photons (très hautes énergies pour les tissus internes)

1976 : premier prototype utilisé en milieu hospitalier.

1982 : première vente.

juillet 1985 : Idem pour une tumeur au bassin, le patient décède 5 mois plus tard. L'analyse conclue par une défaillance de micro-commutateurs.

décembre 1985, mars 1986 : Deux nouveaux incidents, 1 mort. L'analyse conclue par la présence de chocs électriques dus à un court-circuit.

avril 1986 : Un nouvel incident, 1 nouveau mort, mais le second pour un opérateur qui peut alors reproduire l'erreur. La société reconnaît l'erreur logicielle. L'erreur est corrigé.

janvier 1987 : Un dernier incident, 1 dernier mort. De nouvelles erreurs logicielles sont découvertes, arrêt d'exploitation du Therac-25.

On peut donnée comme bilan de cette défaillance comme suite :

- 6 accidents avérés.
- 4 morts.
- 2 ans pour mettre en cause le logiciel et retirer les machines.

2.9.4 Phobos 1

Lorsqu'il s'agit d'espace, à peu près toutes les quantités que l'on considère sont multipliées par un facteur 'astronomique'. Les distances, les vitesses, les coûts,… Et cela est d'autant plus sensible qu'il faut parfois très peu de chose pour faire échouer une mission.

Le Samedi 10 Septembre 1988, le centre de contrôle russe de la mission Phobos 1 [10] en direction de Mars déménageait. Il quittait la Crimée pour rejoindre la banlieue de Moscou. Un programme de 20 à 30 pages a été envoyé à Phobos 1 à cette occasion pour faire quelques réajustements concernant ce changement. Après avoir reçu le programme, la sonde, au lieu de prendre note des changements, réorienta ses panneaux solaires à l'opposé du Soleil. Dû à la latence des communications, le temps que le centre de contrôle reçoive le message d'acquittement de la sonde sur le mouvement des panneaux solaires et que la réponse du centre de contrôle demandant la correction immédiate du positionnement des panneaux solaires parvienne à la sonde, il était trop tard. La sonde avait épuisé ses batteries et ne répondait plus. L'ensemble de cette mission avait coûté l'équivalent d'un milliard de dollars.

2.10 Conclusion

Au cours de ce chapitre nous avons montré avec des exemples réels (section 2.9), que la sécurité des systèmes embarqués est un véritable défi, à la fois technologique et économique.

Ce chapitre a pour objectif de sensibiliser aux problèmes de sécurité réels pouvant exister dans des systèmes embarqués complexes. Ce problème est une partie importante du développement pouvant représenter jusqu'à 80% du coût de développement du système [71]. Il est le point clé pour l'utilisation de ce type de système sous les aspects de confidentialité et d'intégrité.

Pour cela, les chercheurs dans ce domaine décident que le seul moyen de comprendre les codes embarqués qui peuvent être malveillants est de les analyser et déterminer leur fonctionnement interne.

Chapitre 3

La sécurisation : Techniques & Outils

Résumé

L'objectif de ce chapitre est de présenter, à travers le revue de plusieurs études existants, différentes techniques permettant de produire un code informatique fiable et sûr.

Sommaire

3.1 Introduction : Vérifier : Pourquoi le faire ?

Dans un monde parfait, la vérification n'aurait pas lieu d'être. Mais, du fait est que nous sommes loin de cette perfection, les programmeurs font des erreurs, le matériel tombe en panne, et l'environnement est bien souvent hostile. Plutôt que d'essayer de rendre le monde parfait autour du système informatique, il semble plus raisonnable de rendre le système informatique aussi parfait que possible en éliminant les erreurs de conception.

La vérification sert justement à s'assurer que les logiciels vérifient certaines propriétés jugées cruciales par leur concepteur. Cependant, le coût de la vérification d'un système informatique est souvent très élevé. A la fois à cause des compétences engagées et du temps nécessaire à cette vérification. La question qui se pose alors vraiment est de savoir si les défaillances qui risquent de survenir peuvent avoir des répercussions qui justifient les investissements consentis.

Pourquoi ne pas simplement faire confiance aux programmeurs, avoir du matériel de rechange sous la main et supposer que l'environnement ne sera pas trop hostile en moyenne ?

Comme vont nous le démontrer les exemples qui suivent, cet optimisme béat est souvent générateur de catastrophes. Evidemment, les proportions que prennent alors ces incidents dépassent de loin ce à quoi on aurait pu s'attendre.

3.2 La vérification de code

Avant d'être mis sur le marché ou à la disposition du plus grand nombre, un code est généralement soumis à de nombreux tests qui ont plusieurs objectifs. L'un d'eux est d'assurer que le code n'entraîne pas un comportement erratique et dysfonctionnel de la plate-forme sur laquelle il s'exécute. Dans le processus de déploiement traditionnel, le logiciel est remis au client final de manière plus ou moins sécurisée, dans une boite fermée, sur un cédérom, ou sur un autre support peu ou pas altérable par un tiers.

Cependant, dans les nouveaux schémas de déploiement, l'application arrive directement sur nos ordinateurs, sans que nous en aillions nécessairement conscience. Apparaît alors le problème de la sécurité de notre système : il se peut que le code que nous ayons chargé soit altéré ou bien encore qu'il s'agisse d'un virus, tel un code mobile se propageant à travers le

réseau. Pour se prémunir de ce genre de danger, il est nécessaire de définir une politique de sécurité de la plate-forme et de mettre en place les mécanismes assurant cette politique de sécurité, *le processus de vérification*.

Une politique de sécurité contient l'ensemble des règles que doit respecter un code pour ne pas altérer le fonctionnement correct de la plate-forme sur laquelle il s'exécute. Bien sûr, cette politique dépend du niveau de sécurité que nous désirons appliquer. La politique de sécurité doit être cohérente et rendue publique. Il faux alors montrer qu'un code suit bien une politique de sécurité donnée et ne met pas en danger la sécurité intrinsèque de la plate-forme d'exécution.

Le but n'est pas de certifier qu'un programme effectue bien la tâche pour laquelle il a été conçu mais d'assurer qu'il ne menace pas la sécurité de la plate-forme en effectuant des opérations illégales.

3.2.1 Approches et techniques d'exécution sécurisée du code

Cette section va étudier les techniques actuellement connues en tant que mécanismes d'exécution sûre et sécuritaire pour les codes mobiles. Plusieurs études sont effectuées dans ce domaine, ces études menées en vue d'améliorer la sécurité des systèmes dans le cas du chargement d'un nouveau code. Comme exemples de ces études, on a [02,30,52,57,60]. Parmi les approches existantes, ces études résume que les dernières technologies en relation avec la production de logiciels sûrs. Sûreté de fonctionnement et sécurité. Le code auto−certifiant apparaît comme l'approche la plus prometteuse. Dans le reste de cette section, nous présenterons un résumé de ces études et quelques-unes des techniques utilisées.

3.2.1.1 Sanbox (bac à sable)

Le Sandboxing consiste à exécuter le code d'une application dans un environnement restreint appelé le "sandbox". [58]

Un code non testé ou de provenance douteuse peut être exécutée sans préoccupante dans le bac à sable (comme le montre la figure 3.1).

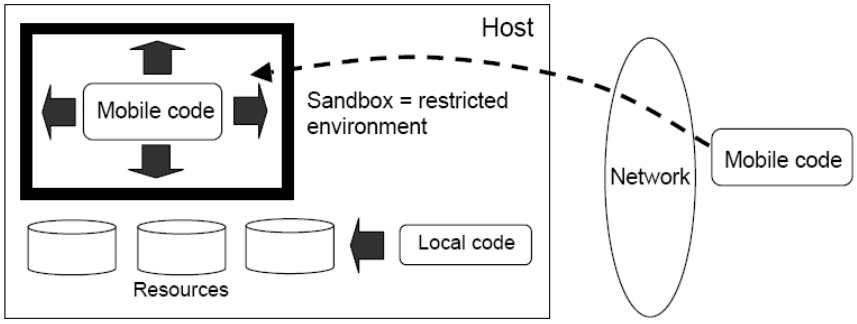


Figure 3.1 Sandboxing [02]

Un sanbox peut être caractérisé par deux mécanismes différents:

- L'utilisation des domaines de protection pour empêcher la subversion du code de confiance et,
- Il impose une politique fixée pour l'exécution de code.

L'inconvénient majeur de cette mécanisme de sécurité est dû au fait que les applications qui s'exécutent dans un environnement aussi restrictives sont eux-mêmes que rarement utile.

3.2.1.2 Signature du code

La signature de code est le processus par lequel un code est signé numériquement par le producteur de code afin d'assurer l'authentification et l'intégrité du code au consommateur.

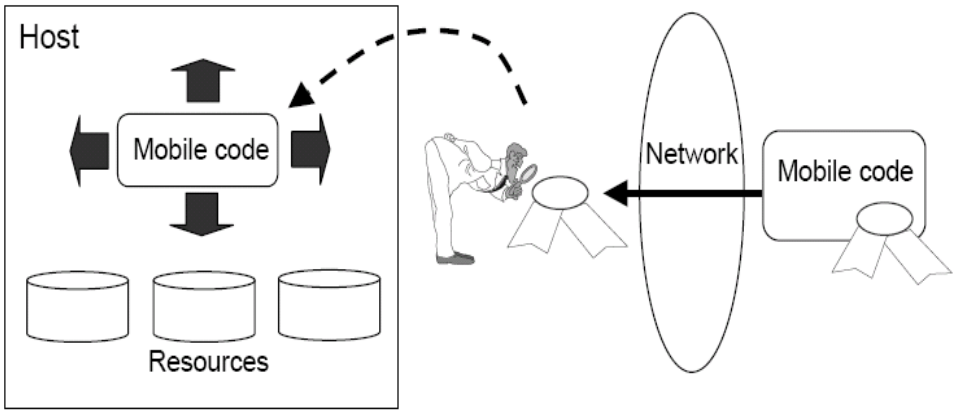


Figure 3.2 Signature du code [02]

Ce modèle a d'abord été introduit par Microsoft [57]. Le Code Signing, parfois appelé la signature d'objet, est une technologie qui a pour objectif premier d'aider les développeurs à distribuer plus de logiciels. Le Code Signing permet d'instaurer la même

confiance chez les clients avant d'installer et d'utiliser un logiciel [54]. Quand on achète un logiciel dans un magasin, la sécurité de son installation et de son utilisation paraît évidente. On peut voir qui a édité le logiciel et si l'emballage a été ouvert. Ces facteurs permettent de décider du logiciel à acheter et de la confiance à lui accorder.

Au contraire, les clients qui téléchargent des logiciels sur Internet ne peuvent pas être aussi confiants. Sans garantie sur l'intégrité du logiciel ou des informations relatives à l'éditeur, les clients rencontrent des difficultés à décider du niveau de confiance à accorder.
Pour augmenter les téléchargements de logiciels, préserver la réputation de leur entreprise et instaurer une relation de confiance avec leurs clients, les éditeurs de logiciels doivent protéger leurs produits des personnes malveillantes.

A partir de l'infrastructure PKI, des signatures et des certificats numériques, le Code Signing permet aux éditeurs de logiciels et de contenu de protéger leurs produits, leurs clients et leur marque [53]. Les fournisseurs de réseaux mobiles peuvent ainsi protéger aussi bien leurs réseaux que leurs abonnés. Cela permet également aux utilisateurs finaux de facilement vérifier la véritable identité d'un éditeur et de confirmer la présence d'une offre logicielle numérique.

Le Code Signing devient obligatoire pour les éditeurs de logiciels. Le Code Signing est le meilleur moyen de répondre aux exigences de l'environnement de transfert de codes actuel [59], qui demande la vérification de l'éditeur et l'intégrité du code.

VeriSign[1] est l'un des fournisseurs de confiance de services d'infrastructure sur Internet pour le monde en réseau. D'après [55], des milliards de fois par jour, les entreprises ainsi que les particuliers utilisent les services de leur infrastructure Internet pour communiquer et effectuer des transactions en toute confiance.

VeriSign offre les caractéristiques techniques et les services les plus désirés des développeurs. Cela comprend les trois fonctionnalités de Code Signing les plus demandées: l'horodatage, la signature en mode noyau et les services de signature pour les plateformes sans fil[2].

[1] *http://www.verisign.fr/*

[2] *Sondage interactif en ligne réalisé parmi des développeurs de logiciels et des décisionnaires chez des éditeurs de logiciels, réalisé par Authentic Response, 07/08*

3.2.1.3 Le Proof-Carrying Code

Le code auto-certifié (Proof Carrying Code ou PCC) permit un code quelconque d'être exécuté à condition d'un valide preuve de sécurité accompagne celui [15]. C'est un technique très promis. Utilisez en associant avec les autres mécaniques prouvera un outil de sécurité très puissante.

Le Proof-Carrying Code a été présenté par Necula et Lee [17] [20] en 1997. Le PCC est un mécanisme qui permet au "consommateur de code" de définir une politique de sécurité et de vérifier que cette politique est respectée par les programmes fournis par le "producteur de code". Aucune confiance n'est accordée au producteur ni au réseau servant à transporter le code. Le consommateur de code ne croit qu'en sa plate-forme et ses outils.

Cette technique, consiste à dépoiler le code avec une preuve formelle de sa correction [56]. Dans le PCC, la preuve de la validité d'un programme accompagne celui-ci. Pour établir la preuve, un théorème mathématique est construit à partir du programme à vérifier. Le producteur fournit alors la démonstration de ce théorème appelée la "preuve".

Une fois celle-ci achevée, le code est expédié avec sa preuve au consommateur. Ce dernier n'a pas besoin d'accorder sa confiance au producteur. Il considère le code reçu comme potentiellement dangereux pour sa sécurité. Il applique lui aussi la même phase de génération du théorème (figure 3.3) mais il n'a pas à calculer sa preuve. Il s'assure simplement que la preuve accompagnant le code correspond bien à la démonstration du théorème généré. Si une erreur au cours de la vérification apparaît alors, le code n'est jamais exécuté par le consommateur.

Intuitivement, cette technique est correcte [52]. En effet, imaginons que le code et/ou la preuve ont été altérés. Lorsque le consommateur va régénérer le théorème correspondant au programme et qu'il va utiliser la preuve fournie avec le programme pour vérifier la correction de ce dernier, deux scénario sont possibles. Le premier, et le plus évident, est que la démonstration ne fonctionne plus avec le nouveau code : l'application a été modifiée, elle ne se plie plus à la politique de sécurité et doit donc être refusée. Le deuxième est moins évident : la démonstration permet de démontrer le nouveau théorème, malgré les modifications. Cela signifie que malgré les modifications,

le code reste valide par rapport à la politique de sécurité définie par le consommateur du code. Même si l'application ne fait plus forcément ce pour quoi elle a été conçue, elle ne menace pas pour autant la plate-forme d'exécution.

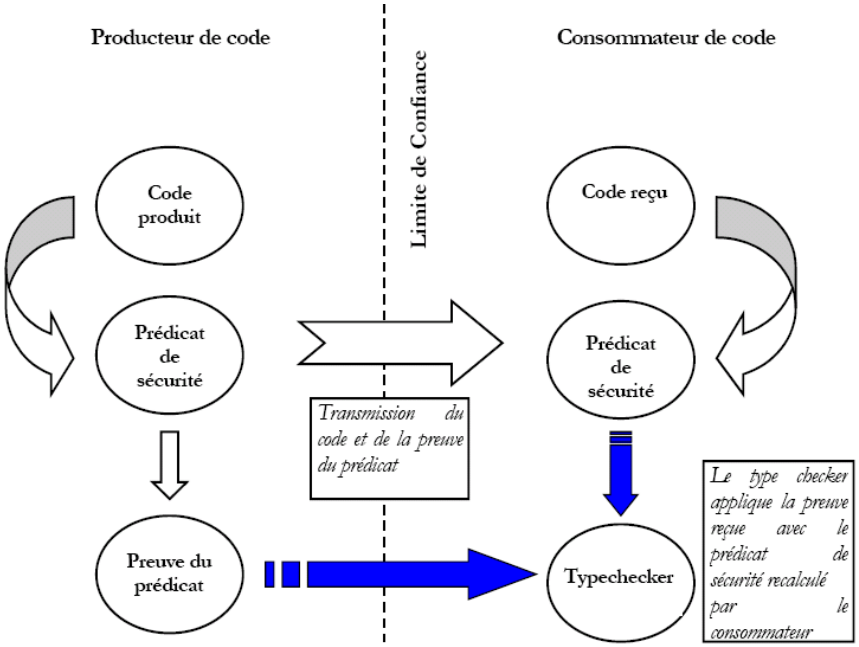


Figure 3.3 Shéma de pricipe du Proof-Carrying Code [52]

L'avantage de cette technique est de disposer d'une forte expressivité qui permet d'augmenter les possibilités de vérification statique. Comme vérifier la preuve sur un programme est plus simple que la générer, l'algorithme de vérification s'adapte parfaitement aux appareils ayant de fortes contraintes en mémoire et en puissance de calcul[33].

D'une autre part, parce que le code non sécurisé est vérifié de façon statique, avant d'être exécuté, il faut non seulement économiser du temps d'exécution, mais nous détectons des activités potentiellement dangereuses, évitant ainsi les situations où le consommateur code doit tuer le processus non fiable après avoir acquis les ressources ou l'état modifié [61].

Le point noir de cette technique est qu'il faut s'assurer de la validité de l'algorithme de vérification. C'est-à-dire trouver une bonne représentation pour la politique de sécurité et pour les éléments de preuve ainsi que fournir la preuve mathématique que l'algorithme n'accepte pas de programmes faux et ne rejette pas de programmes corrects.

Finalement, cette technique semble prometteuse pour plusieurs raisons : son expressivité, sa force de vérification, son mode de fonctionnement autonome (sans tiers de confiance).

3.2.1.4 Le Typed Assembly Language

Le Typed Assembly language (TAL) peut être considéré comme un sous-ensemble du PCC. Typed Assembly Language (TAL) a été développé à l'Université Cornell, par le Professeur Greg Morrisett et ses collègues [49][51]. L'objectif principal du TAL est également de certifier qu'un programme est sûr. Le principe fondamental régissant TAL est qu'au moment de la compilation, les types des annotations sont ajoutées au code. Par la suite, un vérificateur de type peut analyser le code et déterminer s'il est sécuritaire ou non, par rapport à une politique de sécurité [30]. Cette approche est semblable à PCC parce que la preuve PCC peuvent être considérés comme des annotations de type et le vérificateur de preuve PCC en tant que vérificateur de type.

Cependant, même si TAL partage le même but que PCC, il sacrifie sa force d'expression au profit de l'efficacité. Des informations de typage que le programme possédait à un plus haut niveau sont rajoutées au code assembleur. En gardant les informations de typage, il est alors possible d'inférer les types des variables et de s'assurer de leur utilisation correcte. Ainsi, il s'assure de la validité du typage du programme dans son ensemble et de sa correction en fonction de la politique de sécurité choisie. Contrairement au PCC, son application au monde de la carte à puce semble plus évidente : il demande moins d'espace mémoire et il utilise l'assembleur, alors que le PCC est une technique plus générale [52].

3.2.1.5 L'Efficient Code Certification

Les deux technologies PCC et TAL sont très générique. Cependant, ils souffrent de grand taille du preuves/annotations (proofs/annotations) et par conséquent, la génération du preuve/annotation est relativement lent [30]. L'objectif principal d'Efficient code certification (ECC), développé à l'Université Cornell, par le Professeur Dexter Kozen et ses collègues [21][22][16], est de sacrifier les avantages d'être générique pour la certification du code. Par conséquent, ECC ne peut garantir que les propriétés fondamentales de sécurité. Comme TAL, ECC produit des annotations de type et utilise un vérificateur de type. Toutefois, ces annotations sont très compact et beaucoup plus facile de vérifier que TAL. C'est pourquoi ECC est plus efficace que TAL et PCC [30].

3.2.2 État de l'art de la vérification du système embarqué

Dans cette section, nous présentons quelque travaux portant sur la vérification et la sécurité des systèmes embarqués.

- L**'article** [23], à l'objectif de montrer la faisabilité de la vérification formelle de propriétés mixtes fonctionnelles et temps-réel sur un système embarqué. Trois approches sont développées. Une première comparaison oppose d'une part une technique opérant sur un modèle de temps continu (le vérifieur UPPAAL associé aux automates temporisés) à des techniques opérant sur des modèles de temps discrets. Parmi ces dernières, une seconde comparaison oppose les techniques de "model checking", illustrées par le vérifieur SMV associé au langage LUSTRE, à des techniques plus couramment utilisées dans le domaine de l'optimisation combinatoire, illustrées dans cet article par une méthode de satisfaction de contraintes. Ces comparaisons, réalisées sur un exemple du monde avionique, contribuent à montrer la robustesse des approches par contraintes face au phénomène de l'explosion combinatoire.
- **Gemplus.** Fournisseur mondial de solutions basées sur les cartes à puce, Gemplus a développé une formalisation de la machine virtuelle Java Gard. Son équipe de recherche a employé pour cela, comme cela devient fréquent dans l'industrie, la méthode B. L'équipe a commencé par spécifier une machine virtuelle défensive [24], c'est-à-dire effectuant les vérifications de type à l'exécution. Cette spécification est ensuite raffinée, selon les techniques de la méthode B, en un vérificateur de code octet et un interpréteur de code octet sans vérification de types (une machine virtuelle offensive). La correction de cette approche tient aux obligations de preuves déchargées lors de chaque étape de raffinement. Ici encore, seulement un sous-ensemble de la JCVM est pris en compte, même si une étude de passage à l'échelle a été effectuée [48]. En complément de ces travaux, l'équipe de Gemplus, et plus particulièrement CASSET. a développé un vérificateur de code octet léger. destiné à être embarqué sur une carte à puce [45]. Les possibilités d'extraction de l'Atelier B vers du code C ont permis de tester ce BCV sur une carte à puce. Enfin, l'équipe de Gemplus a aussi étudié la conversion d'un sous-ensemble de code JVM vers du code JCVM [46]. Ils obtiennent ainsi une preuve que la spécification de la sémantique du code de la JCVM est un raffinement de données du sous-ensemble de la JVM.

- **Axalto et Trusted Logic.** Des chercheurs de Trusted Logic et Axalto ont formalisé un modèle presque complet de la plate-forme Java Gard [77]. Cette modélisation, effectuée avec l'assistant de preuve COQ, couvre les structures du format capfile, tous les codes octet, les APIs basiques et les méthodes natives, un vérificateur de code octet.

  Le modèle inclut une machine défensive, une machine offensive et deux autres machines abstraites à partir de la machine défensive dont l'une vérifie le typage et l'autre les propriétés du pare-feu. Les preuves de correction de chaque composant vis-à-vis d'une politique de sécurité sont construites. Les machines, en particulier les relations de transition, sont ici exprimées par des prédicats et non par des fonctions. Les preuves sont ainsi facilitées aux dépends d'une exécutabilité plus difficile à obtenir. Du côté vérification de code octet, LEROY [70] propose un BCV pouvant être embarqué sur une carte à puce. Le BCV requiert une transformation de code et des conditions sur le programme à vérifier, là où les approches actuelles préfèrent les techniques de Proof Carrying Code Ce travail constitue un effort très important et représente l'équivalent de 6 ans / homme avec 2 à 5 experts en continu sur la formalisation. Les auteurs montrent ainsi que COQ peut être utilisé en milieu industriel, avec un coût certes très élevé mais avec l'accès aux niveaux d'évaluation maximaux. Leurs conclusions vont dans le sens de la construction d'outils permettant de gérer des modèles imposants.

- **AIRBUS: Application de techniques de vérification formelle au développement d'un logiciel avionique critique** [47]. Le système de Commandes de vol électriques (CDVE) d'un Airbus est principalement constitué de capteurs (pression, gyroscopes, mini-manches, etc), d'actionneurs (servo-commandes, vérins hydrauliques, etc) et de calculateurs qui, outre la fonction à réaliser, doivent assurer la sûreté de fonctionnement et la disponibilité requises par son haut niveau de criticité.

  AIRBUS a appliqué trois techniques de vérification formelle au développement de ces logiciels. Ces techniques se caractérisent par leur sûreté et leur efficacité : à l'inverse du test, les résultats qu'elles délivrent valent pour toutes les exécutions du logiciel – ou bout de logiciel – vérifié, et dans le fait qu'elles sont très largement automatisées. De par leurs principes même, ces méthodes outillées permettront aux

logiciels d'arriver à maturité plus tôt dans le cycle de développement et sont une réponse à l'augmentation de la complexité des logiciels avioniques.

Les trois outils de vérification formelle utilisés sont issus de la collaboration d'AIRBUS avec le CEA, exemple de transfert du nucléaire vers l'aéronautique, Saarland Universität, et un éditeur de logiciel très impliqué dans larecherche : AbsInt GmbH.

- **Jérôme Falampin prsenete dans: Sécurité informatique et transport « B dans le transport ferroviaire »** [50]. Siemens Transportation Systems (STS), anciennement connu sous le nom Matra Transport International, utilise des méthodes formelles pour ses automatismes ferroviaires depuis 1992. Dans cette présentation, on décrit l'évolution de ces méthodes formelles chez STS, ainsi que le cycle de développement actuel, en le comparant à un cycle de développement classique. Le cycle de développement comprend une phase de génération automatique de logiciel sécuritaire, dont on démontre la sûreté de fonctionnement grâce à la preuve formelle. Le processeur sécuritaire codé DIGISAFE permet d'utiliser pour du logiciel sécuritaire un traducteur et un compilateur non certifié, tout en démontrant formellement un niveau de sécurité.**TUCS.** l'objectif de ce projet [62] est d'étudier la diffusion des technologies basée sur la notation semi-formelle UML dans le contexte du développement de logiciels critiques, et notamment au regard des exigences très spécifiques en matière de validation et de certification. Ce projet est réalisé par la collaboration de l'Université technologique de Compiègne (UTC), de l'Université de Haute-Alsace (UHA) et de l'INRETS/ESTAS.
- **Solange Coupet-Grimal**. Preuve formelle d'un récupérateur de mémoire pour cartes à puce: Cet article [72] présente un travail de recherche en informatique, mené dans le cadre d'une collaboration entre l'Université de Provence, l'INRIA et la société Gemplus. Il illustre l'interaction entre mathématique et informatique. Des résultats récents de la recherche en logique y sont utilisés concrètement dans une application industrielle. Plus précisément, il montre comment spécifier et démontrer formellement la correction d'un récupérateur de mémoire embarqué sur cartes à puce. Il montre également comment vérifier automatiquement la preuve de correction avec l'assistant de preuve Coq, dont il présente brièvement les fondements logiques.

– Dans leur travail [08], W. Enck et ses collègues présentent la certification comme une solution au problème lié au téléchargement de logiciels pour les téléphones mobiles.

3.3 Conclusion

Dans un monde où les logiciels deviennent omniprésents dans tous les domaines, les méthodes de vérification commencent à s'imposer comme un moyen de renforcer le niveau de sécurité et de correction des systèmes informatiques et en particulier des programmes. La diversité des systèmes à analyser, des propriétés visées et des moyens mis en oeuvre ont donné naissance à une grande variété de méthodes et d'outils de vérification, répondant à des objectifs différents, comme il a été décrit dans ce chapitre.

Le monde de système embarqué n'échappe pas à cette tendance. Sa vocation sécuritaire motive la nécessité de la vérification de propriétés de sécurité. Par ailleurs, le coût d'une erreur importante dupliquée à des millions d'exemplaire (exemple : carte à puce) justifie un besoin de renforcer la vérification de la correction des programmes embarqués. Les nombreuses études présentées dans ce chapitre illustrent la possibilité et l'intérêt d'appliquer ces méthodes dans le monde du système embarqué.

Dans le Chapitre suivant, nous décrivons l'utilité du langage Java dans les systèmes embarqués que nous utiliserons par la suite comme support pour nos travaux.

Chapitre 4

Java dans les systèmes embarqués

Résumé

Dans ce chapitre, faisons un tour d'horizon des différentes solutions permettant d'utiliser Java pour le monde de l'embarqué. Nous constaterons que l'utilisation de Java dans des équipements contraints et spécifiques oblige à prendre une certaine liberté par rapport à la spécification originale de Java.

Sommaire

4.1 Introduction

L'utilisation de Java dans les systèmes embarqués et temps réel présente des particularités, par rapport à d'autres langages, qu'il convient d'analyser. Notamment, il faut porter une attention spéciale à ce qui lui manque pour répondre aux besoins des systèmes embarqués contraints, dans le but de présenter un certain nombre de recommandations.

En effet, pour en faire une solution dédiée à l'embarqué, le langage Java nécessite des adaptations, et parfois le programmeur doit adopter d'autres pratiques de programmation que pour une application Java classique.

Plusieurs raisons sont à l'origine de l'engouement pour Java. En premier lieu, l'orientation objet facilite la conception de composants logiciels avec des interfaces définies strictement, jusqu'au code source. Ensuite, le langage Java est conçu pour être indépendant de la plate-forme, et donc satisfait aux exigences d'indépendance vis-à-vis du matériel, intéressantes dans une optique d'approvisionnement multisource. Enfin, sa facilité d'apprentissage (on trouve un grand nombre de programmeurs déjà formés) et sa robustesse contribuent à une meilleure productivité et à la réduction des défauts ; elles permettent à l'équipe de développement de se concentrer sur des activités plus « haut niveau » comme la définition des composants logiciels.
Dans ce cas, les programmeurs se concentrent sur la réalisation des logiciels embarqués qui sauraient conserver tous les avantages mentionnés précédemment du langage Java, tout en le renforçant sur les points importants que sont le temps réel, le déterminisme et l'adaptation à des milieux très contraints tels que l'on en trouve dans le milieu automobile (par exemple, une unité de contrôle électronique avec 256 ko de ROM et 16 ko de RAM). Ces points importants concernent l'environnement de développement, la programmation système, la synchronisation, les exceptions, le modèle d'initialisation des applications, la gestion de la mémoire et l'interface Java.

4.2 Langages de programmation pour L'embarquer

Les langages typiquement utilisés aujourd'hui pour le développement de systèmes embarqués sont :

- l'**assembleur** [76], encore maintenant, de nombreux systèmes embarqués sont écrits en assembleur, pas forcément à cause d'un manque d'expérience ou de savoir-faire, mais le plus souvent pour des raisons économiques : dans beaucoup d'applications où la complexité logicielle est très faible, l'assembleur permet de

réaliser des programmes de taille minimale, donc d'utiliser des composants bon marché;

- le **C** [32], la plupart des systèmes automobiles par exemple sont écrits en C. Ce langage est devenu incontournable dans les années 1990 quand les systèmes électroniques ont gagné en complexité;
- le **C++** [82], jugé parfois trop complexe pour être totalement maîtrisé par une large communauté de programmeurs et posant des problèmes difficiles de vérification par des outils automatiques, n'a pas réellement percé pour les logiciels embarqués complexes ;
- **Ada** [12], il est utilisé dans beaucoup de systèmes embarqués complexes comme les applications militaires, avioniques ou spatiales. Le logiciel du lanceur Ariane par exemple est écrit en Ada.

4.3 Problèmes liés au déploiement des applications embarquées

Le chargement de nouvelles applications dans un système embarqué déjà déployé n'est pas toujours faisable : elle est notamment techniquement impossible si l'équipement fonctionne en circuit complètement clos, sans connexion avec l'extérieur qui permettrait de transférer les applications à charger. D'après [19], si une telle connexion est présente, et que le système embarqué est capable de charger du code dynamiquement, plusieurs problèmes sont alors à prendre en compte :

1. Le code chargé dynamiquement occupe une mémoire réinscriptible, contrairement au code chargé initialement qui peut être placé en ROM.
2. Le chargement de code venant d'un agent extérieur pose de sérieux problèmes de sécurité. Il n'est en effet pas prudent d'exécuter un binaire venant de l'extérieur sans vérification.
3. Un autre problème concerne la taille occupée par le code des programmes. D'une manière générale, il est souhaitable de minimiser la taille du code, que celui-ci soit chargé dynamiquement ou pas.

Pour ces raisons, Java s'impose progressivement comme solution embarquée extensible et sûre.

4.4 Java

Java est un vrai langage orienté objet, créé par la société américaine Sun Microsystems et publié en 1995. Java[1] est une marque déposée par Sun Microsystems[2] aux États-Unis et dans le reste du monde.

Les exigences suivantes ont servi de base à la définition de ce langage :

- **simple :** le nombre de constructions du langage est réduit au minimum. La syntaxe Java est proche de celle du C/C++ pour faciliter la migration ;
- **distribué :** vérifié par la pénétration historique de Java dans le World Wide Web, le langage apporte par exemple le RMI (Remote Method Invocation) pour l'invocation de méthodes distantes ;
- **interprété :** le compilateur génère du byte-code, pas du code natif. Un interpréteur est nécessaire pour exécuter ce byte-code. Cette architecture facilite le transfert d'un programme sur des plates-formes d'exécution différentes. Une plate-forme d'exécution, que l'on appelle JVM (Java Virtual Machine), est constituée d'un interpréteur et des bibliothèques de base ;
- **robuste :** c'est un langage fortement typé qui permet de nombreuses vérifications dès la compilation. La déclaration explicite des méthodes est obligatoire. L'absence de pointeurs évite les effacements ou corruption de mémoire. Le mécanisme de ramasse-miettes évite les fuites mémoires et les bugs pernicieux dus aux cycles d'allocation et de libération de mémoire. À tout ceci s'ajoutent de nombreuses vérifications effectuées automatiquement à l'exécution, comme la validité d'un index de tableau ;
- **protégé :** l'absence de pointeur empêche le programmeur de contourner les protections mises en place dans le langage. De plus, le compilateur ne décide pas de l'agencement des objets, effectué par la JVM, donc le programmeur ne peut pas deviner comment les données Java sont stockées en regardant simplement le code source, ce qui rend difficile toute stratégie de contournement par des moyens extérieurs au langage ;
- **neutre du point de vue de l'architecture** : une application Java doit pouvoir être exécutée sur n'importe quel système muni d'une JVM ;

[1] *http://www.java.com/fr/*
[2] *http://www.sun.com/*

- **portable :** en plus de la caractéristique précédente, la portabilité est facilitée par le retrait du langage de tout aspect lié au type de matériel. Par exemple, chaque type de base est défini explicitement : un entier de type int est codé sur 32 bits, quel que soit le microprocesseur. Cela évite les contorsions auxquelles étaient habitués les programmeurs de systèmes embarqués en C jonglant avec des contrôleurs 8 bits, 16 bits ou 32 bits (la taille d'un int y est généralement celle du bus) ;
- **performant :** certes, les applications Java sont généralement moins performantes que leur équivalent C car elles sont interprétées. Cependant, les nouvelles architectures d'exécution comprenant un compilateur JIT (Just In Time : une partie du code est compilée sur la cible avant d'être exécutée, pour être réutilisée efficacement) ou AOT (Ahead Of Time : une partie du code est compilée au chargement) le rendent comparable au C;
- **multithread (multiples fils d'exécution) :** Java fournit un support pour l'exécution de plusieurs threads (classe Thread), ainsi qu'un mot-clef dédié à la synchronisation de ces threads (synchronized);
- **dynamique :** les classes sont chargées par la JVM au moment où elle en a besoin, sans nécessiter l'interruption du flot d'exécution.

4.5 Variantes embarquées de Java

À l'origine, Java est le produit de recherches visant à développer une plate-forme adaptée à la gestion des périphériques réseau et des systèmes embarqués [71]. Pourtant, le succès de Java est surtout venu de son utilisation dans les applications web et les serveurs d'entreprise. Ce succès s'explique par la promesse qu'un programme compilé pour la machine virtuelle Java soit en mesure de s'exécuter sur n'importe quelle plate-forme supportant celle-ci ("Compile once, run everywhere").

Cependant, et malgré la portabilité accrue apportée par cette plate-forme, Java n'a pas pu s'imposer comme une solution convenant à toutes les échelles. En effet, même si la philosophie sous-jacente laisse supposer que la plate-forme Java soit utilisable sur tout type d'équipement, les détails pratiques introduisent des contraintes à sa mise à l'échelle. Un environnement d'exécution Java est en effet composé des éléments suivants :

- Une machine virtuelle, chargée d'exécuter les programmes Java,
- Un ensemble d'APIs et de bibliothèques,
- Des outils pour le déploiement des applications et la configuration du système.

Les outils et formats de déploiement de l'édition standard de Java ne sont notamment pas adaptés à toutes les situations. Le format class utilisé pour charger des classes Java est trop demandeur en ressources pour des équipements tels que la carte à puce.

Une machine virtuelle Java, en tant que telle, n'est pas grosse consommatrice de ressources. Cependant, ses performances sont grandement améliorées par l'adjonction d'outils tels qu'un compilateur Just In Time (JIT) [68].

De telles implémentations de la machine virtuelles améliorent grandement ses performances, mais réclament des ressources supplémentaires. Sun a mis à disposition plusieurs implémentations de sa machine virtuelle :

- **HotSpot** implémentation complète et performante des spécifications de la machine virtuelle Java ;
- **La Compact Virtual Machine (CVM)** [34] implémentation toujours complète, mais plus légère (de l'ordre du méga-octet) ;
- **La Kilobyte Virtual Machine (KVM)** qui propose une plus faible empreinte mémoire (dizaines de kilo-octets) et dont certaines fonctionnalités peuvent être désactivées à la compilation ;
- **La machine virtuelle Java Card** qui implémente la spécification homonyme, version très dégradée de la spécification Java originale ;
- **Squawk** [67] projet de recherche pour une machine virtuelle supportant les mêmes fonctionnalités que la KVM, mais conçue pour fonctionner sur la nouvelle génération de cartes à puce.

Concernant les APIs, l'édition standard de Java propose un très large ensemble d'interfaces aux fonctionnalités du matériel sous-jacent. Celles-ci sont suffisamment génériques pour couvrir la variété des cibles supportées par Java, et peuvent être complétées par d'autres APIs couvrant des problèmes spécifiques, comme la gestion du déploiement d'applications serveurs.

Cependant, cette richesse n'est pas adaptée au monde de l'embarqué. L'étendue et la généricité des APIs de base de Java les rendent bien trop lourdes pour être utilisées sur un équipement limité. D'une part, beaucoup des fonctionnalités d'une machine de bureau ne sont pas présentes sur les équipements embarqués, ce qui rend une grande part des APIs inutiles, à l'inverse, certaines fonctionnalités spécifiques à des équipements embarqués ne sont pas gérées. D'autre part, le champ d'action des équipements embarqués est en général

plus restreint que celui des stations de travail : certaines fonctionnalités pourraient donc devenir plus légères si elles étaient spécialisées.

Au vu de ces différences de besoins, l'offre Java de Sun se décline en quatre versions principales :

1. Java 2, Enterprise Edition (J2EE), s'adressant aux solutions serveurs d'entreprise,
2. Java 2, Standard Edition (J2SE), l'édition de référence pour stations de travail,
3. Java 2, Micro Edition (J2ME), destiné au marché de l'embarqué, notamment de la téléphonie mobile,
4. Java Card, pour les cartes à puce.

La figure 4.1 reprend ces différentes éditions, avec les cibles qu'elles visent, leur jeu d'APIs et les implémentations de la machine virtuelle utilisées.

Dans la suite de cette section, nous détaillons certaines des solutions permettant d'embarquer la technologie Java, et nous intéresserons plus particulièrement à leur adhérence à l'édition de référence.

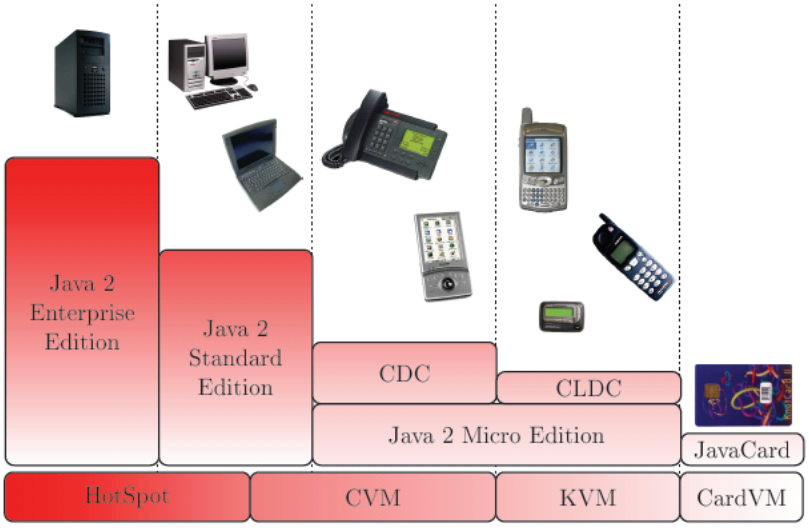


Figure 4.1 Les différentes déclinaisons de Java proposées par Sun

4.5.1 Java 2, Micro Edition

Java 2, Micro Edition (ou J2ME) est un dérivé de Java qui cible les équipements dotés d'au moins 128 Ko de mémoire. Il est né du constat de Sun que "One size does not fit ail" (Une seule taille ne peut pas convenir à tout) [01] et se veut être une version dégradée mais embarquable de Java.

J2ME est disponible sous la forme de plusieurs configurations, chacune adaptée à un type d'équipement particulier. À l'heure actuelle, deux configurations sont disponibles sur le marché :

- **la Connected Device Configuration (CDC)** vise les équipements connectés et sédentaires, tels que les téléphones fixes ou les stations multimédias. Elle se base généralement sur la CVM.
- **la Connected Limited Device Configuration (CLDC)** est quant à elle destinée à des équipements mobiles, plus restreints et ayant une connexion réseau intermittente, comme les téléphones portables. La machine virtuelle de choix pour cette configuration est la KVM.

La configuration J2ME choisie forme le "bloc" de base sur lequel viennent se greffer des profils. Un profil enrichit l'API de J2ME par des fonctionnalités spécifiques à un certain type d'équipement ou une certaine famille d'applications. Par exemple, le profil MIDP (Mobile Information Device Profile) pour CLDC fournit des interfaces adaptées aux écrans basse résolution et aux fonctionnalités réseau limitées des téléphones portables.

Adhérence à la spécification Java : Les limitations de J2ME au regard de la spécification Java standard dépendent de la configuration. CDC fournit un sous-ensemble des APIs de J2SE, ainsi que certaines APIs spécifiques. Ces dernières sont implémentées dans le paquet javax.microedition.

CLDC est lui-même un sous-ensemble de CDC. Les APIs sont largement plus réduites, et beaucoup de fonctionnalités dépendent de la présence de différents profils. Les paquets de J2SE considérés comme coûteux sont réimplémentés par d'autres paquets incompatibles : par exemple, le paquet javax.microedition.io fournit un système de connexions qui remplace de manière plus légère la gestion réseau de java.net. La machine virtuelle de CLDC est également bridée : dans sa version 1.1, CLDC ne supporte pas les nombres flottants, la finalisation des instances, la réflexion ou la gestion de certaines erreurs d'exécution.

4.5.2 Java Card

Java Card [66] a été conçu pour permettre de faire profiter des caractéristiques de Java (sûreté d'exécution, faible empreinte mémoire du code applicatif, etc.) à des équipements aussi restreints que la carte à puce. Les applications Java Card (ou cardlets) sont

programmées en utilisant les outils standards de développement Java, et sont ensuite converties vers le format de chargement cap pour être transférées sur la carte.

Adhérence à la spécification Java Java Card ne reproduit qu'un sous-ensemble très restreint des fonctionnalités de la machine virtuelle Java. La version 2.2 de la spécification Java Card ne supporte ainsi pas les nombres longs, les flottants, les chaînes de caractères, le multitâche ou la collection de la mémoire allouée. Elle redéfinit également la taille de certains types primitifs (les entiers sont ainsi codés sur 16 bits au lieu de 32). Enfin, elle implémente certaines fonctions spécifiques non-présentes dans la machine virtuelle originale, comme le mécanisme de pare-feu permettant d'isoler les applications.

L'API de base de Java Card est également très limitée. Seule une partie très réduite des paquets *java.** est implémentée, la plupart des fonctionnalités étant fournies par des paquets dédiés.

4.5.3 LeJOS et TinyVM

LeJOS[1] et TinyVM[2] sont deux implémentations de l'environnement d'exécution Java pour la plateforme *Lego Minstorm*. Devant fonctionner dans un milieu extrêmement limité (32 Kilobits de mémoire) et spécifique (contrôle du mouvement, télécommande, caméra, senseurs, ...), ceux-ci ne proposent qu'un minuscule sous ensemble de la machine virtuelle et de l'API standard de Java (une cinquantaine de classes et interfaces de *java.**), et complètent leurs besoins par des paquets dédiés.

4.5.4 VM

VM [65] est un canevas dédié à la construction de machines virtuelles Java spécialisées pour réseaux de capteurs, prenant en compte les contraintes de puissance, de mémoire disponible et de consommation énergétique propres à ces équipements.

Les machines virtuelles sont construites au cas par cas par assemblage de composants, en fonction des besoins de leurs applications et de l'équipement sur lequel elles sont déployées. Elles peuvent être mises à jour en même temps que leurs applications par édition des liens incrémentale [25]. Toutefois, ce type de mise à jour ne protège pas le système contre les injections de code malicieux.

[1] *http: lejos.sourceforge.net/*
[2] *http: tinyvm.sourceforge.net/*

4.5.5 JEPES

JEPES [34] est une plate-forme Java embarquée visant les équipements extrêmement contraints, capable de fonctionner avec 512 octets de RAM et 4 Ko de ROM. Elle ne supporte qu'un sous-ensemble du langage Java, prend une liberté totale vis-à-vis de l'API standard qu'elle redéfinit complètement, et change comme Java Card la taille des types de base. JEPES comprend un compilateur qui prend en entrée un ensemble de classes Java et produit un binaire homogène incluant les classes compilées et optimisées ainsi que le support d'exécution du système.

JEPES se pose comme une alternative au développement en C des applications pour équipements extrêmement contraints, et promet de combiner la sûreté de programmation

de Java avec les performances et la faible empreinte mémoire d'un système natif, les classes étant en effet compilées vers du langage natif pour éviter d'embarquer une machine virtuelle dans le système final. Sur des équipements dotés de si peu de mémoire, le coût en terme d'empreinte mémoire de la machine virtuelle est en effet très difficile à amortir par le gain obtenu grâce à la forme bytecode des applications.

4.5.6 JDiet

Ce projet n'est pas à proprement parler un environnement d'exécution Java embarqué, mais permet en revanche d'embarquer des applications qui ne sont initialement pas prévues pour cela. JDiet[1] est un sous-projet du projet Spoon, un préprocesseur pour Java proposant un méta-modèle du langage qui permet de modifier un programme source écrit en Java. JDiet est capable de transformer un code source écrit pour J2SE en une variante adaptée à J2ME CLDC. L'approche adoptée consiste donc en une adaptation du programme à un environnement contraint au niveau du source.

Les transformations effectuées par JDiet restent cependant limitées : elles consistent, pour la plupart, à changer des conteneurs non disponibles dans CLDC par ceux qui y sont présents (par exemples, les déclarations de Set et de List sont transformées en Vector). Ainsi, un programme invoquant une interface de J2SE pour laquelle JDiet ne propose pas de transformation ne pourra pas être totalement converti. Un utilitaire nommé ASMJDietVerfier permet de vérifier qu'un code source ne référence pas d'entité non-présente dans l'API de CLDC.

[1] *http://spoon.gforge.inria.fr/JDiet/Main*

4.6 Conclusion

Le langage Java, porté par ses atouts indéniables comme la robustesse, la portabilité et surtout sa capacité à coopérer avec les composants existants écrits en C, est en train de faire sa révolution en douceur. Dans ce chapitre nous avons donné une idée de l'utilisation de Java dans les systèmes embarqués.

Les objectifs de cette partie du mémoire étaient d'expliquer que le développement des applications pour les systèmes embarqués doit prendre en compte :

- de mieux connaître la problématique de l'intégration d'applications Java dans l'embarqué ;
- de connaître les critères de choix d'une bonne solution Java adaptée à une application donnée ;
- d'anticiper les problèmes typiques rencontrés par les programmeurs de logiciels embarqués en passant à Java ;
- d'anticiper les problèmes typiques rencontrés par les programmeurs Java en écrivant du logiciel embarqué.

Chapitre 5

Signature & Certification du code

Résumé

Ce chapitre présente la dernière partie de notre travail, qui consiste à prouver pratiquement que la signature et la certification des codes assure les propriétés de confidentialité et d'intégrité des données des applications. on prend les téléphone portable comme un exemple des système embarqué, nous avons utilisé les outils de sécurité fournis par Java pour signer les codes et générer des certificats. Les signatures numériques offrent les mêmes avantages que les signatures manuscrites, en ce sens qu'elles peuvent uniquement être créées par le signataire, qu'elles sont vérifiables et qu'elles ne peuvent pas aisément être contrefaites ou répudiées.

Sommaire

5.1 Introduction

Les systèmes ouverts ajoutent une nouvelle dimension à la sécurité des logiciels présents sur ces systèmes. D'une manière générale, on dit que les systèmes embarqués sont reprogrammables. On peut télécharger un nouveau programme constitué par un code exécutable qui se présente sous la forme d'une suite d'instructions exécuté par le microprocesseur.

L'opération de téléchargement d'application sur un système embarqué pose un certain nombre de problèmes de sécurité. Ainsi, un code involontairement mal écrit peut modifier de manière incorrecte des données déjà présentes sur le système, empêcher le programme principal de s'exécuter correctement, ou encore modifier d'autre code d'applications téléchargées précédemment, en rendant ceux-ci inutilisables ou nuisibles.

Egalement, à partir d'un programme écrit dans un but malveillant, il est possible d'influencer le bon fonctionnement de tout le système. Il devient aussi possible d'avoir accès à des données confidentielles ou non autorisées stockées dans ce système, telle que le code d'une carte bancaire par exemple, ou d'attaquer l'intégrité d'une ou plusieurs applications présentes sur la carte. Enfin, si la carte est connectée au monde extérieur, les dysfonctionnements peuvent se propager à l'extérieur.

Comme nous l'avons présenté dans le chapitre 2, les attaques logicielles représentent une menace important pour les systèmes embarqués.

Une solution apportée à ce problème est d'interdire le chargement de code après le dépoilement des systèmes, dans ce cas là, notre système devient fermé. En d'autres termes, il n'est pas possible pour un utilisateur de télécharger un programme malveillant en vue d'attaquer la sécurité et le fonctionnement correct du système embarqué. Cette solution donne à l'utilisateur un certain niveau de sécurité, mais signifie que les services des systèmes s'appuient exclusivement sur les services présents par défaut dans le système.

Une autre solution consiste à le rendre dans un lieu dont il reconnaît l'autorité pour charger un programme sur le système, mais cette solution à les mêmes effets que la première, restreint considérablement la facilité d'extension apportée par les systèmes ouverts.

Dans ce chapitre nous allons expliquer le principe de la signature et la certification du code, on utilisant les outils fournis avec Java, nous démontrerons une solution utilisée pour garantir un certain niveau de sécurité dans le domaine des systèmes embarqués.

5.2 Méthodes pour exécuter du code embarqué

Les concepteurs des systèmes embarqués ont le choix entre différentes méthodes d'exécution du code: la méthode XIP (eXecute-In-Place), la méthode "shadowing" et la pagination à le demande. L'utilisation de la meilleure méthode pour l'application considérée est la clé pour une solution rentable. Toutes ces méthodes ont différentes implications en terme d'usage de RAM et de Flash, affectant le coût et les performances générales du système. En choisissant la solution appropriée il est possible d'optimiser les performances et le coût final.

Avec la **méthode XIP** (eXecute-In-Place) [43], le code est exécuté directement à partir de la mémoire Flash où il est stocké. Seules les données volatiles sont stockées dans la RAM. Il n'est pas possible d'exécuter du code directement à partir d'une mémoire Flash NAND du fait de son accès séquentiel par bloc ; seules les flash de type NOR peuvent être utilisées à cet effet.

La **méthode du "shadowing"**[03] consiste à stocker toutes les données et le code en mémoire Flash (souvent en Flash NAND), dont le contenu est copié (et décompressé si nécessaire) dans la RAM au démarrage par le bootloader. L'ensemble du code doit être transféré dans la RAM avant que l'exécution ne débute. C'est évident qu'il faut donc autant de RAM supplémentaire, équivalente à la taille du code à charger. Cette RAM supplémentaire se paye en termes de consommation électrique et de coût. Avec cette méthode, il est difficile d'avoir un temps de boot très court : vérification mémoire, copie en Ram, décompression, lancement du système, de l'application ….

La **méthode de pagination** à la demande copie des pages de code de la mémoire Flash en la RAM suivant la demande, et retire les pages en RAM qui ne sont plus en service. Presque tous les systèmes d'exploitation utilisent la pagination par ce que l'on appelle communément le "swap". Cette méthode réduit l'utilisation de la RAM par rapport à la méthode "shadowing" et il n'est pas nécessaire de charger la totalité du système / code en RAM. Pour les systèmes embarqués sous Linux, les pages de code en Flash sont compressées ce qui permet de réduire aussi la taille de la Flash. L'inconvénient est que si le système doit exécuter plusieurs applications simultanément, il faut plus de RAM pour charger les pages de code, et donc on se rapproche de la méthode "shadowing". Si la Ram viens à manquer le système passera son temps à faire du swap de pages de codes, avec une perte évidente de performances.

La **compression de code** apporte un bénéfice certain en termes de taille de Flash nécessaire pour le stockage du code. Les méthodes de shadowing et de pagination utilisent régulièrement la compression, alors que la méthode XIP ne peut pas appliquer la compression car les processeurs ne peuvent pas exécuter du code compressé directement, le code devant être décompressé avant. Dans la méthode de pagination à la demande, la compression se fait sur la taille d'un bloc qui est un multiple de la taille d'une "page" mémoire. De cette façon 1 ou N pages sont décompressées quand une page de code est nécessaire.

Pour conclure, le meilleur compromis consiste à mixer la méthode XIP avec la pagination afin d'équilibrer la taille de la RAM / Flash pour avoir l'empreinte la plus faible possible, sans sacrifier les performances. Après le choix se fait en fonction de l'application, des performances requises et du prix (les mémoires sont toujours des éléments coûteux).

5.3 Le concept de multi-application embarquée

En général, un système embarqué multi-applicatif est un système qui permet de fournir des services aux différentes applications qui sont en mémoire. Selon le type de la mémoire qui stocke le code de l'application, divers cas peuvent être envisagés. En effet, si le code de l'application réside dans une mémoire de type ROM, cela signifie qu'elle a été fournie par l'émetteur du système (e.g. la banque dans le cas de la carte à puce) et, par conséquent, elle peut-être considérée comme "sûre". Le code de l'application peut aussi résider dans une mémoire de type EEPROM. Dans ce cas, il s'agira souvent d'une application (d'un fournisseur de service) chargée via le service du système d'exploitation permettant l'installation de nouvelles applications. Le système d'exploitation propose aussi souvent la possibilité d'effacement d'une telle application. Ces applications ajoutées après que le système ait été émis, sont celles qui posent le plus de problèmes d'un point de vue sécuritaire.

5.4 Sécurité des téléphones portables

Pour expliquer le problème du sécurité et la nécessité de la vérification du code pour les systèmes embarqués, en prend le téléphone portable comme un exemple de ces systèmes. De nombreux codes malicieux arrivent sur nos smartphones par le biais du Bluetooth, mais également via SMS ou MMS.... Demain avec la VoIP sur Wifi [83], les attaques utiliseront ce canal de communication.

Les Téléphones d'aujourd'hui appelés PDA ou Smartphones ou le tout intégré utilisent des systèmes d'exploitation qui se rapprochent de nos ordinateurs. Certains smartphones embarqueront Symbian[1], d'autres embarqueront Windows Mobile[2].

Les hackers possèdent déjà la technologie [80]:

- Envoi de SMS (technologie malicieuse : RedBrowzer)
- Copie de fichiers sur d'autres tel (technologie malicieuse : CxOver)
- Propagations bluetooth et MMS (technologie malicieuse : StealWar)
- Propagation par mail (technologie malicieuse : Letum)
- Remplacement et hook d'applications système (technologie malicieuse : Mobler)
- Remplacement des fichiers de boot (technologie malicieuse : FlerProx)
- Concaténation de fichiers applicatifs (technologie malicieuse : HideMenu)
- La technologie malicieuse : Wesber : technologie de vol d'argent s'appuyant sur espionnage des données et transactions financières
- Etc.

Comme on peut le constater, les codes malicieux mobiles peuvent accéder au système d'exploitation du téléphone, à partir de là, le code malicieux sur le téléphone a tous les droits. Le problème est d'assurer à l'utilisateur, dans la mesure où il a maintenant la possibilité de charger d'autres programmes sur le système, que ces programmes n'inférons pas avec les programmes déjà présents. Alors la sécurité du système repose sur l'hypothèse que le programme chargé ait été vérifié.

5.4.1 Architecture du téléphone portable

Avant d'expliquer le processus de vérification du code que nous avons implémenté pour être utilisé, avec les mobiles phones en tant qu'exemple très répondu du système embarqué, et très utilisé dans notre vie quotidienne, nous allons donner une vue simplifiée et rapide sur l'architecture du mobile phone.

L''architecture matérielle du téléphone portable repose généralement sur un processeur de type RISC (Reduced Instruction Set Computer) de la famille ARM (Advanced RISC Machine) dont la capacité de traitement varie de quelques dizaines de MHz à plus de 100 MHz, la puissance du système étant intimement liée aux services proposés: Vidéo, Audio, Jeux, animation 2D/3D. On trouvera différentes mémoires autour de l'unité centrale:

[1] *http://www.symbian.org/*
[2] *http://www.microsoft.com/windowsembedded/*

mémoire vive RAM (Random Access Memory) et mémoire flash avec des volumes supérieurs aux méga-octets, parfois un peu de mémoire morte ROM (Read Only Memory) et OTP (One Time Programmable) pour des services dédiés à la sécurité. De plus, des processeurs dédiés multimédia peuvent également être présents afin de compenser le manque de puissance de certains processeurs. Finalement, les interfaces écran, clavier, connecteurs (série, IrDA, BlueTooth), haut-parleur, micro, etc., habillent le tout. Ainsi, l'architecture matérielle est comparable à ce que l'on connaît dans le monde PC, avec bien souvent les mêmes fabricants sauf une différence importante, la configuration matérielle d'un téléphone portable n'est pas modifiable par l'utilisateur.

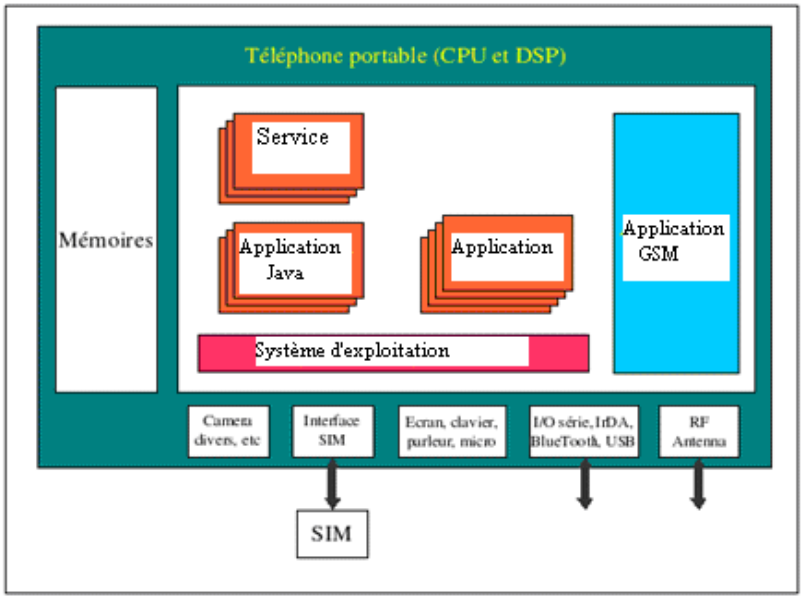


Figure 5.1 L'architecture logiciel du téléphone portable

Au niveau du logiciel, l'architecture est plus variable suivant les fabricants de téléphones portables. L'ensemble des applications du téléphone s'exécute au-dessus du système d'exploitation (Figure 5.1) à l'exception de l'application GSM (Global System for Mobile Communications) et des protocoles associés qui reposent généralement sur une implémentation propriétaires pour des raisons de performance. Le système d'exploitation fournit l'interface de programmation qui tient compte de l'intégration de téléphone de n'importe quelle application, donc ces types d'OS sont dits «ouverts». De plus, les applications peuvent profiter des capacités de communication de données de téléphone. Toutes les applications typiques d'Internet telles que des browsers, des clients d'email, et des messagers instantanés fonctionnent au-dessus du service des données du téléphone. La tendance actuelle va vers des téléphones portables à base d'OS ouverts, tels que Embedded

Linux[1], Symbian et WindowsCE, la mise à jour de l'OS étant par ailleurs possible dans certains cas.

Les mécanismes de sécurité doivent prendre en compte les exigences qui ont été présentées dans le chapitre 2. Actuellement l'opérateur se positionne en tant que fournisseur de service vis-à-vis de l'utilisateur. A ce titre, il est à même d'assurer que le niveau de sécurité est conforme à l'attente de l'utilisateur. En particulier, dans le contexte d'un commerce électronique, l'opérateur garantit à l'utilisateur la confidentialité, l'authenticité et la validité de la transaction.

5.4.2 Utilisation systématique des composants matériels dédiés sécurité

Comme nous allons le voir dans les sections suivantes, on s'intéressons à la vérification du code durant la phase de l'installation dans le système embarqué (dans notre cas le mobile phone).

La sécurité des applications du téléphone portable est moins bien gérée, ceci s'explique par plusieurs rasions : la disparité des architectures matérielles, la disparité des architecture du système d'exploitation et particulièrement des services de sécurité, disponibles dans chaque système, et la diversité des standards. L'absence d'attaque à un grande échelle n'incite pas les acteurs à prendre les mesures nécessaires, malheureusement, encore une fois il faudra s'attendre à une telle attaque pour qu'enfin la sécurité soit vue comme un argument commercial, et donc prise en compte.

La solution la plus simple est d'empêcher l'exécution d'applications malicieuses par éviter de charger de nouvelles applications sur le téléphone portable. D'ailleurs, c'est ce que font la plupart des opérateurs : sur une téléphonie portable standard.
Dans le cas d'un vrai Smartphone, une solution consiste à sécuriser une partie du système et de s'assurer que le monde non sécurisé ne peut pas agir sur le monde sécurisé. C'est assez difficile et aucun téléphone actuel n'utilise cette approche. En particulier, il faut s'assurer que les applications sécurisées peuvent acquérir un contrôle exclusif sur l'écran, le clavier et les autres interfaces quand elles doivent interagir avec l'utilisateur.

L'aspect matériel concentre sur le niveau le plus bas du système. L'utilisation des composants de matériel additionnels augmente la sécurité considérablement.

[1] *http://www.embedded-linux.org/*

Si on stocke un secret, tel qu'une clé, sur un périphérique de stockage, il y'a alors de bonnes chance que quelqu'un d'autre puisse le lire. Une solution pour éviter cette attaque est de stocker la clé dans un endroit séparé, auquel aucun logiciel malveillant ne peut accéder. Ceci ne résout pas le problème que la clé doit être transférée à la mémoire vive (RAM) pour être traitée par l'unité centrale. La solution est d'avoir un processeur séparé à l'endroit où la clé est stockée. Ce genre de stockage et d'unité de traitement dédiée est implémenté en tant que carte intelligente.

L'usage d'un module de co-processeur, sécurisé séparé, qui est dédié au traitement de toutes les informations sensible dans le système est trouvé dans [69] [63] [84].

Beaucoup d'architectures des systèmes embarqués se fondent sur l'identification et la maintenance des zones choisies de son sous-système (volatile ou non-volatile, hors puce ou sur puce) en tant que des endroits de stockage sécurisé. L'isolement physique est souvent utilisé pour limiter l'accès des zones de mémoire sécurisées aux composants de confiance du système. Quand ce n'est pas possible, un mécanisme de protection de mémoire adopté dans beaucoup de SOCs (System On Chip) embarqués comporte l'utilisation de matériel surveillant le bus, qui peut distinguer l'accès légal et illégal à ces endroits. Par exemple, la solution de sécurité de CryptoCell de discretix[1] comporte BusWacher qui exécute cette fonction. Le travail dans [42] décrit un modèle d'exécution dans la mémoire (*eXecutable Only Memory*} - XOM), et des techniques architecturales pour le mettre en oeuvre, en utilisant des élargissements de matériel des champs additionnels dans les caches…

Notons que ces composants spécifiques qui sont utilisés pour intégrer des services de sécurité, se traduit par une augmentation du coût du téléphone portable.

5.4.3 Vérification des applications du téléphone portable

Comme nous l'avons mentionné dans les sections précédentes de ce mémoire, les exigences de sécurité changent selon les perspectives que nous considérons. Par exemple, dans le cas des systèmes d'exploitation propriétaires fermés (ne supportant pas le téléchargement d'applications), du point de vue d'un fournisseur de services, cela signifie soit que son service s'appuie exclusivement sur des services présents par défaut dans les téléphones portables, soit qu'il négocie un accord pour inclure le code nécessaire dans les téléphones portables.

Dans le premier cas, l e constructeur d u téléphone portable pourrait (à la demande de l'opérateur) s'assurer, éventuellement via une évaluation, que les services présents dans le

[1] *http://www.discretix.com/CryptoCell/*

téléphone portable ne puisse être détournés à des fins malveillantes. Dans le second cas, l'accord devrait exiger l'évaluation du code avant l'installation, afin de vérifier qu'il ne contient ni porte dérobée (*backdoor*), ni bogue résiduel. Dans les deux cas, le fournisseur de services ne serait pas en mesure d'introduire un programme malveillant dans le téléphone portable.

Du point de vue de l'utilisateur, les systèmes propriétaires fermés rendent impossible le téléchargement de nouvelles applications. En d'autres termes, il n'est pas possible, pour l'utilisateur, de télécharger un programme malveillant en vue d'attaquer le téléphone portable et/ou les services.

Au contraire des systèmes d'exploitation fermés, les systèmes d'exploitation ouverts supportent au minimum le téléchargement de services ou de contenu via des applications écrites en Java, mais aussi le téléchargement d'application (généralement, en C++) exécutées par le système d'exploitation, et dans certains cas, la mise à jour du système lui-même. Dans ce contexte, le risque est grand de voir l'utilisateur installer, volontairement ou non, une application malveillante. Pour faire face à ce problème, les constructeurs de téléphones portables proposent des solutions complémentaires dont les objectifs sont de garantir la sécurité tout au long du cycle de vie des applications. Nous présentons les solutions actuellement déployées dans les différentes phases d'une application (le développement, le téléchargement, l'installation, et l'exécution), ainsi que leurs limites.

5.4.3.1 Développement

Les problèmes de sécurité rencontrés dans la phase de développement sont les mêmes que ceux rencontrés dans le monde PC. Les applications sont généralement écrites en Java ou C++ en s'appuyant en grande partie sur les APIs standards, mais les développeurs utilisent aussi des APIs propriétaires afin d'optimiser le code, ou d'accéder à des fonctions spécifiques. Des telles pratiques augmentent les risques liés à la présence de failles de sécurité, ces APIs contournant les mécanismes de sécurité du système d'exploitation. De plus, l'existence de bogues résiduels ou de portes dérobées peut être exploitée pour attaquer le téléphone portable.

5.4.3.2 Téléchargement

La phase de téléchargement correspond au chargement d'une application dans le téléphone portable à partir d'un serveur. Notons que le téléchargement peut être à l'initiative de l'opérateur (par exemple, pour une mise à jour logicielle), du fournisseur de services (par

exemple, lors de l'accès à un portail), ou bien de l'utilisateur (par exemple, le chargement de nouveaux jeux). Les applications peuvent également être chargées à partir d'un poste de travail ou d'un autre téléphone portable et dans ce cas, le problème revient à assurer la sécurité dans la phase de l'installation (voir section suivante).

Dans cette phase, les problèmes de sécurité se posent au niveau de l'accès au service et du transfert de l'application. Concernant l'accès au service, les mécanismes à mettre en place sont côté serveur (contrôle d'accès par mot de passe, gestion des abonnements, etc.) et ne rentrent pas dans le carde de ce mémoire. Au niveau transfert, il s'agit de garantir la confidentialité et/ou l'intégrité des échanges, afin d'empêcher un attaquant d'accéder au code et aux données de l'application, ou de modifier ce code et d'y introduire une faille.

A l'image de ce qui fait sur Internet, l'utilisation de TLS/SSL [75] pour sécuriser le niveau transfert nous semble une bonne solution. Ainsi, le WAP Forum a spécifié WTLS [74] qui n'est autre que l'implémentation de TLS dans WAP, bien que, généralement implémentée dans les téléphones portables, signalons cependant qu'une telle solution n'est pas souvent utilisée.

En résumé, les solutions pour sécuriser le niveau transport, bien qu'existantes, ne sont généralement pas ou mal mises en oeuvre. Les opérateurs préfèrent considérer que la sécurité au niveau transport est assurée par la sécurité “physique” du coeur de réseau, oubliant que ce dernier est de plus en plus souvent connecté à Internet.

5.4.3.3 Installation

Une fois l'application téléchargée, et avant son installation, il s'agit de *vérifier* qu'elle est saine, en particulier, qu'elle ne contient pas de virus ou autres programmes malveillants. Ceci peut être réalisé soit par des antivirus, soit par des techniques de certification de code. La première n'est, à l'heure actuelle, que rarement utilisée, mais le tapage qu'a alimenté le ver Cabir[1] donne l’idée que les distributeurs d'antivirus vont rapidement se lancer dans la course. Notons que les antivirus s'appuient sur la “signature virale” propre à chaque virus pour les détecter. Il s'agit de la méthode de "recherche de signature", la plus ancienne méthode utilisée par les antivirus.

Cette méthode n'est fiable que si l'antivirus possède une base virale à jour, c'est-à-dire comportant les signatures de tous les virus connus. Toutefois cette méthode ne permet pas la détection des virus n'ayant pas encore été répertoriés par les éditeurs d'antivirus.

[1] *http://www.f-secure.com/v-descs/cabir.shtml*

De plus, les programmeurs de virus les ont désormais dotés de capacités de camouflage, de manière à rendre leur signature difficile à détecter, voire indétectable. Certains antivirus utilisent un contrôleur d'intégrité pour vérifier si les fichiers ont été modifiés. Ainsi le contrôleur d'intégrité construit une base de données contenant des informations sur les fichiers exécutables du système (date de modification, taille, et éventuellement une somme de contrôle). Ainsi, lorsqu'un fichier exécutable change de caractéristiques, l'antivirus prévient l'utilisateur.

L'utilisation d'antivirus est une protection passive, nous pouvons détecter les virus seulement lorsque nous avons le fichier de définition de virus. D'autre part, cette approche ne fonctionne correctement que si nous avons déjà une séparation forte entre un domaine de confiance où faire tourner l'antivirus et un domaine "non sécurisé" où exécuter les applications à surveiller. Dans notre cas, intéressons-nous plus particulièrement à la certification de code. Le principe fondamental est de signer numériquement le code afin de s'assurer de son authenticité. Cette approche sera étudiée d'une façon plus détallée dans les sections suivantes.

5.4.3.4 Exécution

Lorsqu'une application est installée, elle est "exécutable". A ce niveau, il peut sembler inutile de mettre en place des mécanismes de sécurité, dans la mesure où, pour être installée, l'application a dû être jugée saine. Néanmoins, une application saine ne veut pas dire qu'elle ne contient pas de portes dérobées ou de bogues résiduels. En outre, le pire ennemi de la sécurité est l'utilisateur à qui on donne encore une fois la possibilité d'outrepasser les mécanismes de sécurité, par exemple en lui demandant de confirmer une action jugée non sûre (message "*Unable to verify. Continue anyway?"* pour Cabir).

L'API Symbian [67] est extrêmement riche. D'autre part, les applications Symbian sont compilées directement en code assembleur ARM, et ont des accès illimités à la plate-forme.

Une application native Symbian peut virtuellement effectuer n'importe quelle opération sur le téléphone, à cause de l'absence de modèle de sécurité (pas de notion d'utilisateur, de liste de contrôle d'accès aux fichiers, etc.).

De plus, une grande partie de l'API se trouve dans des librairies et non dans le noyau, ce qui permet de contourner plus facilement les éventuels mécanismes de sécurité. Par exemple, pour envoyer un SMS, et en supposant qu'une confirmation utilisateur soit

imposée par l'API, il suffirait d'appeler directement la fonction interne d'envoi de SMS plutôt que de passer par l'API standard pour contourner cette sécurité.

Il est donc préférable de mettre en place des mécanismes de sécurité afin de contrôler les actions des applications lors de l'exécution. De manière générale, les mécanismes de sécurité à l'exécution dépendent fortement des mécanismes utilisés lors de l'installation.

5.5 Signature et certification du code embarqué

Dans la Figure 5.2, trois parties sont en présence : le porteur final du système, le fournisseur d'applications et un organisme indépendant. Le rôle de l'organisme indépendant est d'assurer au utilisateur final du système embarqué, et par la même occasion à l'émetteur du système, que l'application proposée par le fournisseur d'applications ne menace pas l'intégrité et le fonctionnement même du système. Pour cela, l'organisme indépendant teste et vérifie l'application. Après cela, il produit un certificat cryptographique identifiant l'application. Ce certificat est envoyé avec le code de l'application au client. L'utilisateur vérifie le certificat et s'assure qu'il correspond bien à l'application. Si c'est bien le cas, l'application est installée sur le système embarqué et peut s'exécuter à la demande du client. Cette méthode redonne au système embarqué sa fonctionnalité de chargement du code.

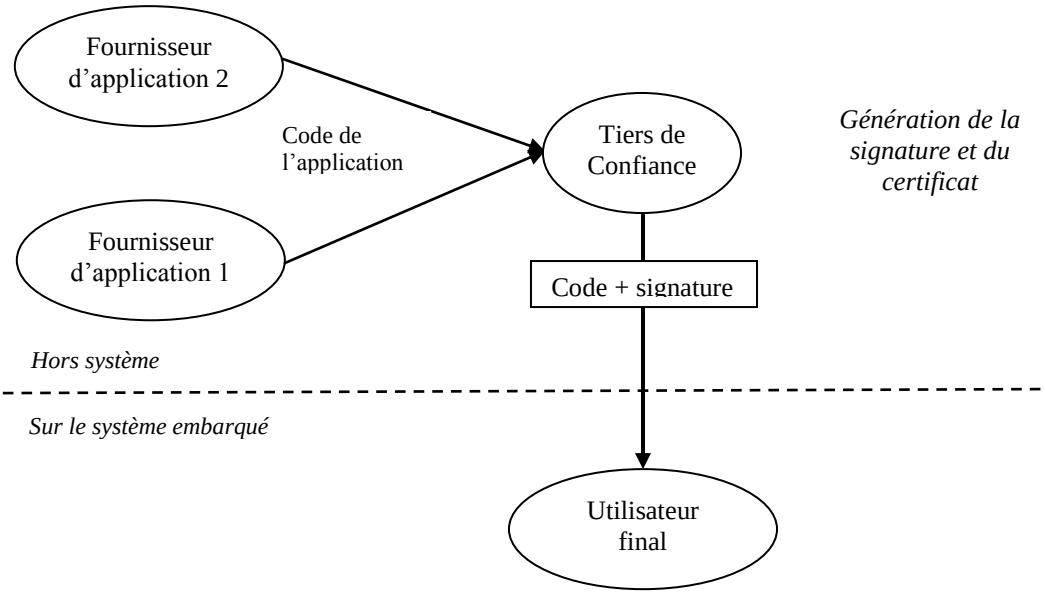


Figure 5.2 Le schéma de déploiement du code à l'aide de la cryptographie

Dans cette partie de ce mémoire, nous décrivons d'une façon pratique la signature et la certification du code embarqué, nous présentons notre résultat se forme de deux applications : l'une pour signer et vérifier le code, et l'autre pour générer un certificat

associé pour le vérifier avant d'être chargé et exécuté dans le système embarqué. En exploitant comme nous l'avons signalé précédemment le téléphone portable comme un exemple du système embarqué.

Mais avant de commencer expliquer l'implémentation de cette solution, nous devons répondre à cette question : pourquoi signer et certifier un code ?

➢ **Pourquoi signer et certifier un code ?**

Les utilisateurs font confiance aux logiciels qu'ils achètent du magasin, car ils peuvent savoir qui a publié les produits et peut voir si le paquet a été ouvert ou non. Le problème apparaît lorsque les clients téléchargent le logiciel à partir d'Internet, ou reçoit un code à partir de quelqu'un d'autre et veut l'utiliser sur son système embarqué.

La signature du code permet à un développeur de signer son application. Sur la base de cette signature, l'utilisateur décide de faire confiance au logiciel ou non. Lorsqu'un client télécharge un logiciel avec un certificat de signature de code, il peut être assuré:

La source de contenu:

Le logiciel est vraiment de l'éditeur qui l'a signé.

L'intégrité du contenu:

Le logiciel n'a pas été modifié ou endommagé.

Les développeurs bénéficie de code signé et certifié car il soutient leur réputation et rend leurs produits plus difficiles à falsifier. En signant le code, ils créent une relation de confiance avec les utilisateurs.

5.5.1 Notions relatif

5.5.1.1 Sécurités fournies par Java

Les fonctionnalités de sécurité fournies par le Java Development Kit (JDK ™) sont destinés à un public varié:

– **Les utilisateurs de programmes**

Travaillez dans un environnement sécurisé protégé contre les programmes malveillants, maintient la confidentialité des fichiers et informations, et authentifie l'identité de chaque fournisseur de code.

– **Développeurs**

Les développeurs pouvant utiliser les méthodes de l'API pour intégrer la fonctionnalité de sécurité dans sons programmes y compris les services de cryptographie et les contrôles de

sécurité. Elles permet d'intégrer leurs propres autorisations (contrôle de l'accès à des ressources), implémente les services de cryptographie,…

– **Administrateurs de systèmes, les développeurs et les utilisateurs**

Les outils JDK gèrent les magasins de clé (base de données de clés et certificats), génèrent des signatures numériques pour les fichiers JAR, et vérifient l'authenticité de ces signatures et de l'intégrité du contenu signé, créent et modifient les fichiers qui définissent la politique d'installation de votre politique de sécurité.

5.5.1.2 La cryptographie à clé publique

La cryptographie à clé publique est une méthode utilisée pour transmettre et échanger des messages de manière sécurisée (authentification de l'émetteur, garantie d'intégrité et de confidentialité).

Cette technique repose sur le principe de "paire de clés asymétriques", ou "bi-clé", qui sont des clés de chiffrement. Dans ce cas, ce qui est chiffré par une clé ne peut être déchiffré que par la clé qui lui est associée et vice versa.

Chaque individu engagé dans une transaction est muni d'une "clé privée" et d'une "clé publique". Sa clé privée ne doit être communiquée à personne, tandis que sa clé publique est transmise à tous ses interlocuteurs, sans aucune restriction.

Si un individu A veut envoyer un message crypté et veut être sur que seul B pourra le lire, il devra crypter le message avec la clé publique de B. Celui-ci ne pourra déchiffrer le message qu'avec sa clé privée (secrète).

5.5.1.3 Keytool et Keystores

Les clés privées et leurs certificats associés sont enregistrés dans les bases de données nommé *keystores* protégées par un mot de passe demandé**.**

L'outil keytool fourni par le JDK de Sun permet de gérer les clés et les certificats, qui sont stockés dans un keystore.

5.5.2 La vérification de code mobile

La plupart de ces nouvelles techniques de certification du code suivent le schéma de la figure 5.3. Dans ce schéma, le producteur de code développe son application. Parce qu'aucune confiance entre le producteur et le consommateur n'est instaurée, le producteur doit faire "la preuve" de sa bonne foi. Pour cela, il fournit, en plus du code, des certificats qui servent à établir la correction de l'application.

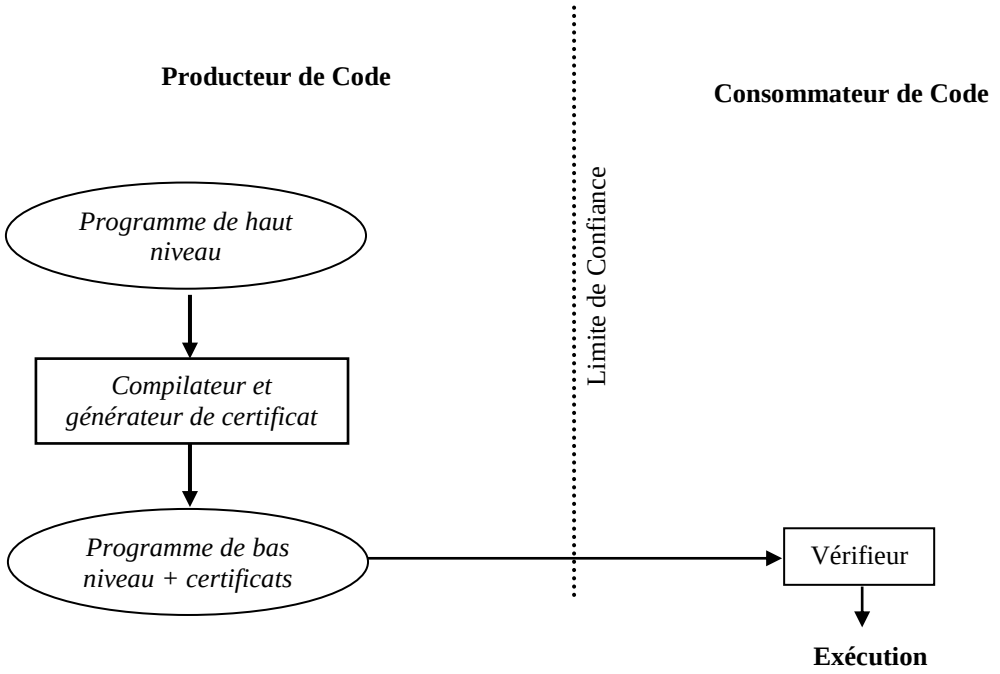


Figure 5.3 Schéma de déploiement de la vérification de code mobile [52]

L'idée générale de ce schéma est qu'il revient au producteur du code de montrer la correction de son application. Il doit ainsi faciliter la tâche de la vérification faite par le consommateur de code.

5.6 Signature du code

Nous avons identifié dans les chapitres précédents les fonctionnalités critiques des systèmes embarqués, il convient maintenant de choisir une méthode de vérification afin d'améliorer la fiabilité du code et la sécurité générale de ce type de système. Dans notre cas, nous avons choisi la signature et la certification des applications, comme une méthode de vérification du code et les mobiles phones comme un exemple disponible du système embarqué.

Comme un signe d'autographe, qui vise à identifier la personne ou l'entité responsable de la délivrance d'un document papier, la signature numérique a l'intention de signer les codes et l'identifier son origine. Il utilise le principe de cryptage à clé publique, le chiffrement est réalisé avec la clé privée, et le déchiffrement avec la clé publique. Un individu A peut signer un code avec sa clé privée et l'envoyer à B. Ce dernier ne peut décrypter le code de A qu'avec la clé publique de A. Dans ce cas B aura la possibilité de vérifier que A lui a envoyé le code et que ce dernier n'a pas été altéré en cours de route.

Un mécanisme de signature numérique doit présenter les propriétés suivantes :

– Il doit permettre à l'utilisateur d'une application d'identifier la personne ou l'organisme qui a apposé sa signature.
– Il doit garantir que le code n'a pas été altéré entre l'instant où le développeur l'a signé et le moment où l'utilisateur le consulte.

Pour cela, les conditions suivantes doivent être réunies :

- **Authentique** : L'identité du signataire doit pouvoir être retrouvée de manière certaine.
- **Infalsifiable** : La signature ne peut pas être falsifiée. Quelqu'un d'autre ne peut se faire passer pour un autre.
- **Non réutilisable**: La signature n'est pas réutilisable. Elle fait partie du code signé et ne peut être déplacée sur un autre code.
- **Inaltérable** : Un code signé est inaltérable. Une fois qu'il est signé on ne peut plus le modifier.
- **Irrévocable** : La personne qui a signé ne peut le nier.

Notre travaille sera diviser en deux parties :

- Coté du développeur du code et,
- Coté utilisateur du système embarqué.

Nous démontrons comment peut-on utiliser les API de sécurité JDK pour générer et vérifier une signature numérique. De point de vue de sécurité et de vérification, nous expliquons comment les développeurs peuvent utiliser cette méthode pour intégrer la fonctionnalité de sécurité dans leur programme qui sera chargé par la suite dans les systèmes embarqués.

Pour cela, nous avons fait une simulation de cette méthode sous forme de deux parties ; nous présentons comment peut-on signer notre application en utilisant l'algorithme DSA (Digital Signature Algorithme), par la suite nous passons dans la deuxième partie de notre travail à la vérification de la signature de ce code avant de le charger dans notre système embarqué.

L'idée de base dans l'utilisation des signatures numériques est la suivante :

1. "Signer" le code en utilisant l'un des clés privées,
2. Envoyez le code signé à l'utilisateur du système embarqué,
3. Donner à notre destinataire notre clé publique**.** Cette clé correspond à la clé privée que nous avons utilisée pour générer la signature,

4. Le destinataire utilise la clé publique pour vérifier que l'application est venue de nous, et n'a pas été modifié avant d'atteindre sa destination.

5.6.1 Génération d'une signature numérique

Dans cette étape, nous utilisons les outils JDK Security API pour générer une paire de clés; une clé publique et une clé privée, et une signature numérique pour le code. La figure 5.4 ci-dessous montre la série d'opérations qu'un développeur doit exécuter pour envoyer son application signée à un utilisateur du système embarqué.

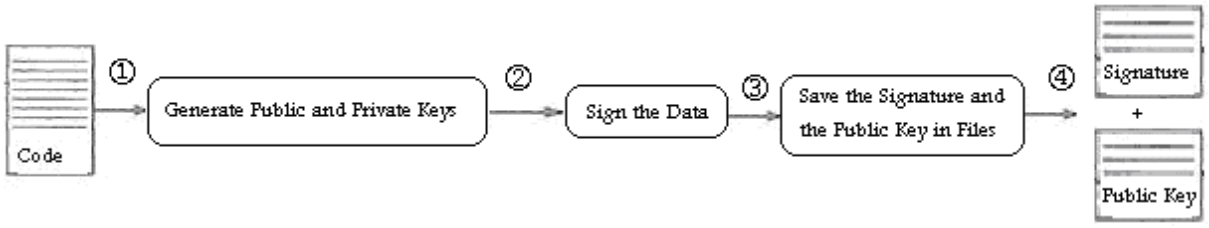


Figure 5.4 Générer une signature numérique du code

- **Générer les clés publiques/privées**

Pour créer une signature numérique, nous avons besoin d'une clé privée (La clé publique correspondante sera nécessaire par la suite afin de vérifier l'authenticité de la signature). Cette paire de clés est générée en utilisant la classe KeyPairGenerator et l'algorithme DSA (Digital Signature Algorithm).

```
KeyPairGenerator  keyGen =
        KeyPairGenerator.getInstance("DSA");
/* généré la paire de clés */
KeyPair pair = keyGen.generateKeyPair();
PrivateKey priv = pair.getPrivate();
PublicKey pub = pair.getPublic();
```

- **Signer le Code**

Après avoir créer la clé publique et la clé privée, nous allons procéder à la signature du code en utilisant l'algorithme DSA, le même algorithme pour lequel le programme a généré les clés précédemment.

```
Signature dsa = Signature.getInstance("SHA1withDSA");
```

- **Enregistrer la signature et la clé publique dans des fichiers**

La signature et la clé publique devront être enregistrées dans deux fichiers indépendants pour être transmis à l'utilisateur final, qui va les utiliser pour authentifie et vérifier la fiabilité du code.

La signature a été placée dans un fichier `"signe"`.

```
/ * Mettre la signature dans un fichier * /
      FileOutputStream sigfos = new FileOutputStream("signe");
      sigfos.write(realSig);
      sigfos.close();
```

La clé publique sera enregistrée dans un fichier nommé *"pubC*le"

```
/ * Mettre la clé publique dans un fichier * /
      byte[] key = pub.getEncoded();
     FileOutputStream keyfos = new FileOutputStream("pubCle");
     keyfos.write(key);
          keyfos.close();
```

5.6.2 Vérification de la signature numérique

La figure 5.5 illustre de façon schématique les étapes de vérification de la signature d'un code donné avant d'être utilisé et chargé dans un système embarqué.

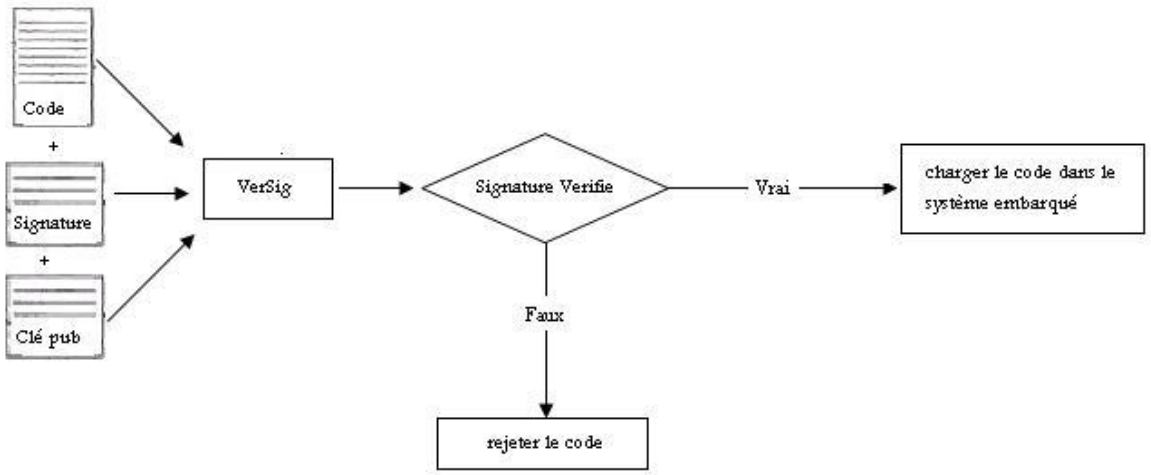


Figure 5.5 Vérification de la signature

Le destinataire (dans notre cas l'utilisateur du système embarqué) reçoit le code accompagné de la signature et de la clé publique, vérifie l'authenticité de la signature et obtient un rapport du résultat :

```
boolean verifies = sig.verify(sigToVerify);

System.out.println("signature verifies: " + verifies);
```

Si pour une raison quelconque, le code est modifié après la signature, le résultat sera False en signifiant que le logiciel n'est pas fiable sinon le résultat de la vérification est True ce qui signifie que le logiciel est fiable et que l'utilisateur peut le télécharger en toute sécurité.

5.7 Certification du code

Les certificats de code permettent aux fabricants des logiciels de mettre en oeuvre une signature numérique dans les programmes et les macros développées, afin de garantir leur droit d'auteur, tout en transmettant à l'utilisateur final l'authenticité et la confiance. Si pour une raison quelconque, le code est modifié après la signature, un avertissement sera généré pour les utilisateurs en leur disant que le logiciel n'est pas fiable. Il n'est donc pas seulement un moyen efficace d'authentification, mais aussi une garantie d'intégrité. Le but du certificat est de garantir qu'une signature correspond bien à une identité donnée.

La seule méthode pour que le récepteur vérifie si un certificat est valable ou non, est de vérifier sa signature numérique, en utilisant la clé publique fournie avec le code.

Il s'agit d'un document sous forme électronique attestant du lien entre les données de vérification de signature électronique et un signataire.

Le certificat électronique est une véritable carte d'identité électronique contenant différentes séries d'informations:

- Le titulaire : contenant les informations relatives à la personne qui certifié le code (nom, prénom, service, fonction),
- Période de validité : Période de temps pendant laquelle un certificat numérique est valide. Date de début et date d'expiration.
- La clé : permettant d'authentifier la signature du titulaire du certificat.

Dans la partie précédente de notre mémoire nous avons présenté la signature numérique. Nous avons implémenté cette solution en deux parties : une pour la signature des fichiers et l'autre pour la vérification de cette signature. Dans ce but, nous avons utilisé les outils de

sécurité fournie par Java et l'algorithme à clé asymétrique DSA. Cette solution nous a permis de signer tout type de fichiers ou de programmes, et de vérifier cette signature. Le destinataire peut alors avoir la certitude que le code a été envoyé par l'émetteur et n'a pas été altéré. Si ce n'est pas le cas, il est possible que :

1. Le code n'a pas été signé par l'émetteur ou que
2. Le code a été altéré.

Dans les deux cas, le fichier doit être rejeté.

Dans cette section, nous présentons une autre partie de notre travail, ce n'est pas trop loin de ce que nous avons présenté précédemment, à la différence que nous avons testé sur un système embarqué réel 'mobiles phones'. A son tour, cette méthode est basée sur le principe de la signature et de la paire des clés que nous avons utilisée et détaillée dans la section précédente. En effet, pour avoir confiance à une application, il faut examiner le certificat associé à la signature.

Bien que les certificats numériques soient électroniques, il ne faut pas oublier qu'étant donné qu'ils sont normalisés, ils peuvent être utilisés sur des ordinateurs personnels mais aussi sur de nombreux périphériques tels que des ordinateurs de poche, des téléphones portables et des cartes à puce. Les cartes à puce peuvent être employées sur un large éventail de périphériques différents et constituent par de nombreux aspects le support idéal pour les certificats numériques. Elles permettent de transporter et d'utiliser les certificats numériques à la manière d'un permis de conduire ou d'un passeport.

Dans le cas des téléphones portables (Nokia 6600), nous avons constaté que les fichiers de type JAR (Java ARchive files) sont des fichiers exécutables, nous utilisons les outils de sécurité fournis par Java pour générer notre certificat, nous pouvons ajouter une phase supplémentaire pour transformer n'importe quel type de fichier ou d'application en un code exécutable pour le téléphone portable (jar). Pour simplifier le processus de la vérification, cette transformation s'effectue hors de notre système embarqué et avant le chargement et l'éxécution. Une phase de transformation du code est donc réalisée hors phone, il s'agit par conséquent d'un code intermédiaire modifié et normalisé prêt à être installé, plutôt que le code original. Ainsi, ce code sera envoyé avec un certificat est sera plus facilement vérifié en mode statique.

Cette solution laisse au porteur du téléphone portable le pouvoir d'accepter ou non le code via le certificat associé avant d'être installé et exécuté.

A cet effet, notre travail concerne donc un procédé de gestion d'un code exécutable formant un programme destiné à être charger dans un système informatique embarqué reprogrammable tel que le phone. Le code généré possède une signature et est exécuté par le microprocesseur du système embarqué après la vérification de la validité du certificat. Le procédé comprend les étapes :

– **Hors phone :**

Pour simplifier l'analyse et la vérification du code embarqué, certaines transformations sont effectuées hors phone avant le chargement :

- Identifier un code exécutable modifié correspondant au code original, adapté à notre utilisation,
- Signer le nouveau code,
- Générer un composant logiciel (certificat),
- Télécharger le code original signé et le certificat dans le phone.

– **Sur le phone :**

- Vérifier respectivement le code signé et le certificat associé;
- Exécuter le code.

La figure 5.6 illustre de façon schématique les étapes du procédé réalisé hors notre système embarqué (phone).

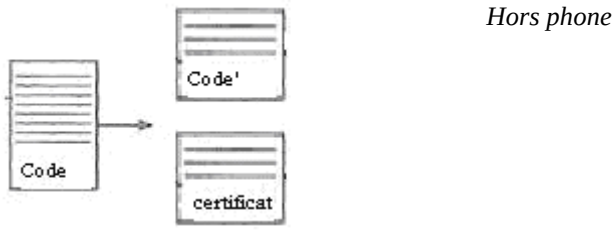


Figure 5.6 Générer signature et certificat

La figure 5.7 illustre l'étape de téléchargement du code intermédiaire et du certificat associé dans le phone.

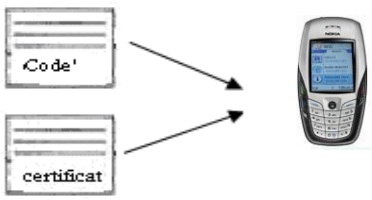


Figure 5.7 Télécharger le code et le certificat

La figure 5.8 illustre les étapes du procédé réalisé sur le phone.

Sur phone

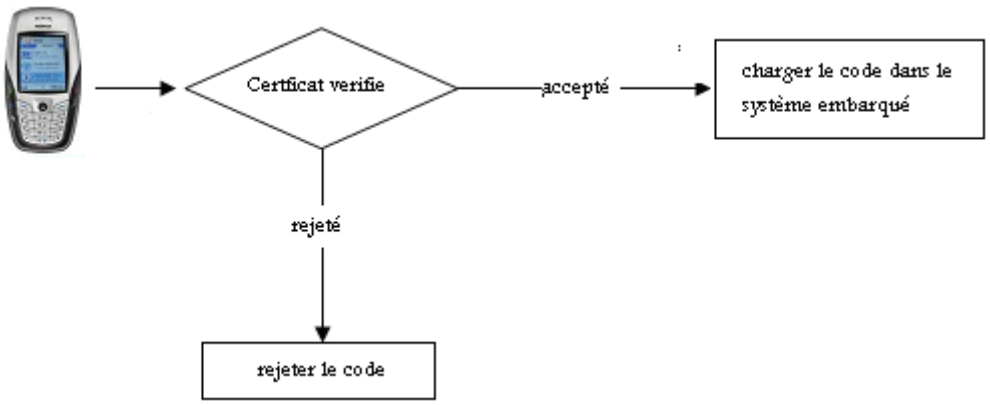


Figure 5.8 Vérification du certificat

Cette approche est orientée vers les applications dans le contexte de système ouvert, et plus particulièrement celui des systèmes embarqués reprogrammables qui supporte ce type d'application.

Dans la figure 5.7 : le code téléchargé est un code exécutable, exécuté par le microprocesseur. Dans le cas où le certificat associé à ce code est vérifié et accepté par le porteur de phone, l'application est installée et exécutée.

Alors il est possible de télécharger des applications signées dans le phone ou d'autre type de système embarqué et de vérifier cette signature. Ceci est rendu possible grâce au composant logiciel appelant le Certificat téléchargé en même temps que le code original. Une fois téléchargés, les certificats se présentent sous la forme de petits fichiers. Il suffit de les installer sur l'appareil et exécuter l'application.

Suivant les techniques de sécurité de Java fournis par SUN, nous devrons suivre les étapes suivantes : signer l'application, générer le certificat, envoyer et charger dans le système embarqué. La figure 5.9 résume ces étapes :

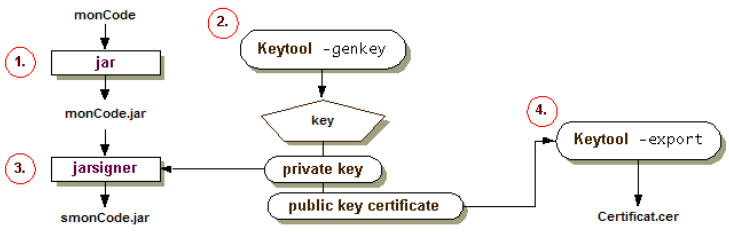


Figure 5.9 les étapes de signature d'un application[1]

[1] *http://java.sun.com/*

5.7.1 Créer un fichier JAR contenant le Code

Phase optionnelle, peut être ignorée si l'application est de type jar.

5.7.2 Générer la Clé

Pour signer le fichier monCode.jar , nous avons besoin de générer les clés, en utilisant l'outil keystore décrit précédemment pour produire : un clé privé (private key) utilisée dans la signature du code, et une clé publique (public key) que nous donnerons à l'utilisateur du système pour vérifier la signature et le certificat du code.

A la fin de cette étape Nous aurons un fichier contenant les informations relatif à l'application qu'on veut signer et certifier, par exemple : le nom , département, société, lieu de résidence, état,…etc.

5.7.3 Signer le fichier JAR

Le fichier jar (monCode.jar) sera signé en utilisant la clé privée généré dans la phase précédente. Nous obtenons smonCode .jar signé :

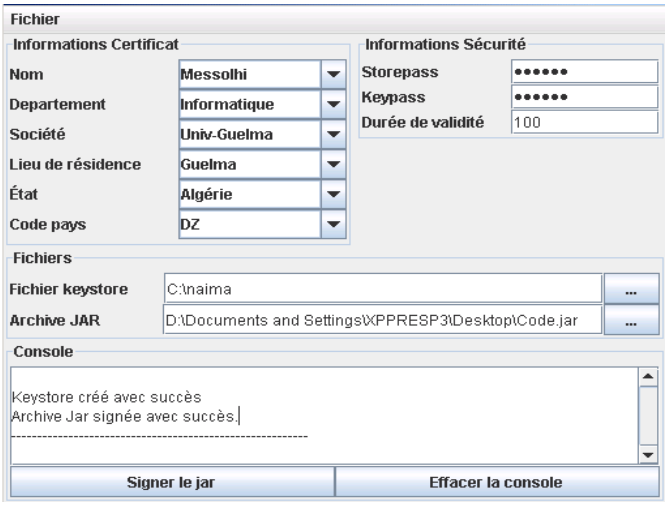


Figure 5.10 Signer le fichier JAR

5.7.4 Exporter le certificat de la clé publique

L'utilisateur qui souhaite charger le code dans son système embarqué, doit vérifier notre signature pour identifier leur source avant de l'installer. La dernière étape avant de publier notre code ou notre fichier à destination d'un client est de produire un certificat qui permettra aux porteurs de téléphones portables de vérifier l'authenticité du code avant de l'exécuter (c'est-à-dire de s'assurer que personne n'a remplacé notre code par une version modifiée).

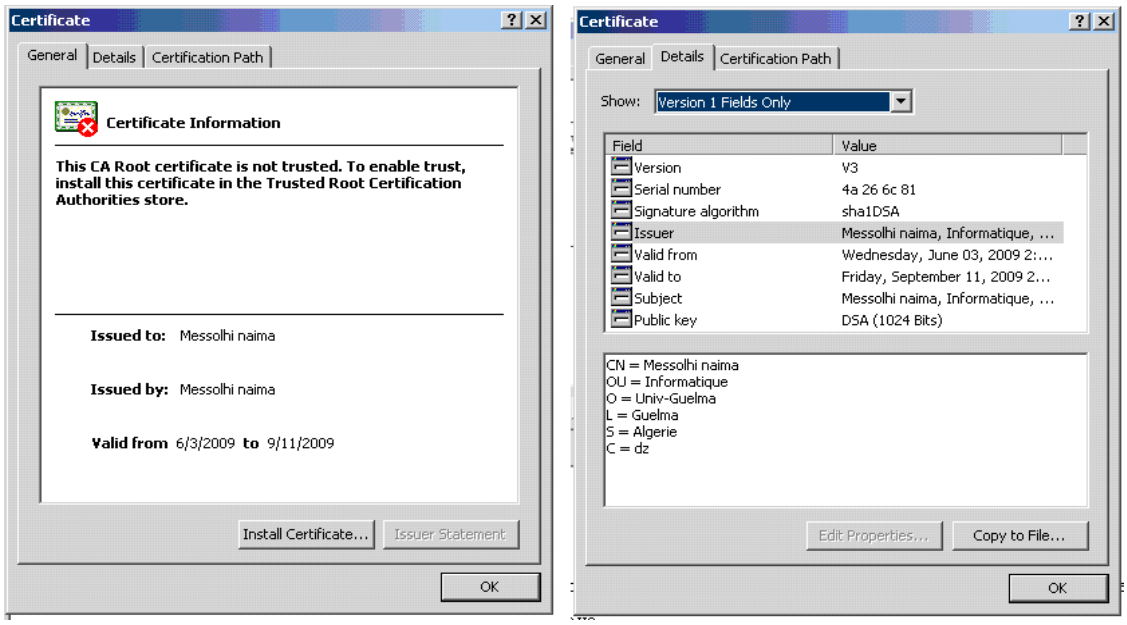


Figure 5.11 Certificat obtenu

Pour cela, l'utilisateur a besoin de la clé publique qui correspond à la clé privée que nous avons utilisée pour générer la signature dans l'étape précédente. Nous fournissons cette clé publique par l'envoi d'une copie du certificat (Certificat.cer) qui contient la clé publique.

Une fois que le récepteur a le code signé et le certificat associé, il peut examiner toutes les informations qui confirment sa source, y compris le nom du développeur, l'adresse, La compagnie qui les emploie…etc. Parfois, ces informations ne sont pas liées à un individu, mais à toute une entreprise ou institution qui produit ce type de certificats fournis à ses clients avec le produit logiciel qu'elle développe. L'utilisateur a toute la liberté d'installer et d'utiliser le fichier de l'expéditeur ou le rejeter.

5.8 Conclusion

Du fait d'une croissance importante des systèmes embraqués sur le marché, l'aspect sécurité est devenu primordial pour des raisons financière, de confidentialité et d'intégrité. Il est même devenu un facteur commercial.

Dans ce chapitre nous avons présenté une solution pour protéger les systèmes embarqués. L'état de l'art des différentes méthodes étudiées nous a permis de choisir d'utiliser la signature et la certification du code pour vérifier les codes dans le but de l'amélioration de l'aspect sécurité du système embarqué.

Cette méthode permet aux utilisateurs de faire confiance aux applications embarquées en donnant des informations complètes (le nom d'une personne, société,…) concernant le producteur de code contenant dans le certificat transféré accompagnant.

Ce chapitre montre qu'en termes de sécurité, la signature et la certification numérique du code sont des solutions performantes. C'est une méthode de vérification statique, l'utilisateur peut vérifier le code avant de l'exécuter. Nous avons expliqué cette méthode en utilisons les outils de sécurité Java.

Conclusion et perspectives

Il est inutile de rappeler l'importance de la vérification du code dans le monde moderne de l'informatique et plus particulièrement pour les systèmes embarqués. En effet, il ne se passe pas une journée sans qu'on apprenne l'apparition d'une nouvelle menace, d'une faille de sécurité nouvellement découverte ou encore d'un nouvel outil qui permettrait d'assurer la sécurité.

Le code malicieux peut être définis comme étant tout code ajouté, modifie ou enlevé à un logiciel dans le but de causer intentionnellement des dommages ou de porter atteinte au fonctionnement normal du système.

Dans ce mémoire on s'intéresse au chargement et à l'exécution de nouvelles applications dans un système embarqué. Pour cela on a présenté l'idée de "signer" et de "certifier" du code, afin que l'utilisateur du système puisse vérifier que le code vient en effet d'une source bien définie, et qu'il n'a pas été modifié.

La signature du code permet à un développeur de signer son application. Et sur la base de cette signature, l'utilisateur décide de faire confiance à l'application ou non. Lorsqu'un client télécharge un logiciel avec un certificat de signature de code, il peut s'assurer de la source et de l'intégrité du contenu.

Dans ce travail nous avons développé un outil pour signer et certifier les applications dans le but de la vérification des codes embarqués.

Durant ce travail, le schéma de déploiement de notre application n'en reste pas moins lourd: il fait intervenir dans la transaction commerciale entre le client et son fournisseur, une troisième partie qui doit vérifier l'application et produire un certificat. Ainsi, le temps de déploiement d'applications est soumis à un goulet d'étranglement qu'est l'organisme indépendant. Le recours à cette troisième partie alourdit et rallonge les temps de déploiement des applications. Il est alors difficile de réagir rapidement dans un marché tendu et spécialement au marché qui évolue rapidement comme le système embarqué.

De plus, si le code cryptographique est cassé, c'est tout le système qui est mis en péril. Pour ces raisons, nous avons a fixé que les perspectives envisageables de nos travaux s'articulent autour des deux axes :

- Pourquoi ne pas mettre de la vérification de code dans le système embarqué ? Dans ce cas le système lui-même a la possibilité de vérifier les applications avant l'exécution. Cet outil doit respecter la nécessité d'optimisations du code pour rendre le vérifieur embarquable. Cet objectif est ambitieux, il nous a conduit à la possibilité de supprimer le tires de confiance afin de ne laisser que deux protagonistes : le producteur et le consommateur de code. Dans ce cas, le système embarqué assure seule sa propre sécurité et doit embarquer le processus de vérification.

  Pour cela, Il apparaît que le vérifieur joue un rôle crucial dans la nouvelle architecture du système embarqué. Nous devons faire confiance à cet élément de la sécurité du système : c'est lui qui autorise ou refuse l'installation d'une nouvelle application. Le problème touche alors son implémentation. Peut-on lui faire confiance ou doit-on le vérifier ? Ce qui nous conduit à notre deuxième axe de perspectives de notre travail futur ;

- Pour nous, son implémentation doit être irréprochable et l'utilisation des méthodes formelles pour aider à son développement est fortement conseillée, ces méthodes sont l'avenir du développement logiciel. Il s'agit maintenant d'oeuvrer pour leur intégration dans les nouveaux développements. Cette intégration va prendre du temps, le temps nécessaire au développement des outils adaptés et des méthodologies adéquates. Mais aussi le temps de modifier les habitudes des développeurs.

Pour conclure, il semble évident que la vérification et la détection du code malicieux reste un domaine de recherche qui nécessite encore une attention soutenue de la part des chercheurs. La résolution de ce problème exigera inévitablement une synergie des diverses disciplines liées à l'informatique et à la programmation. Mais les enjeux sont tels qu'il est indispensable de mettre tous les efforts pour radier le fléau du code malicieux. Nous espérons que ce mémoire a contribué un tant soit peu à ce domaine.

Acronymes

CLDC Connected Limited Device Configuration. Spécification pour une configuration J2ME.

WIM Wap Identity Module. Un module de sécurité intégré à la carte SIM.

J2ME Java 2 Micro Edition. framework Java spécialisé dans les applications mobiles.

OS Operating system.

WAP Wireless Application Protocol. protocole normalisé utilisé par les téléphones mobiles pour accéder aux applications Internet.

SMS Short Message Service**.** un protocole pour envoyer et recevoir des SMS sur les réseaux cellulaires numériques.

VDM Vienna Development Method. Un ensemble d'outils de développement informatique.

UML Unified Modeling Language. Langage de modélisation unifié**.**

CVM Compact Virtual Machine. Une machine virtuelle Java optimisée.

SIM Subscriber Identification Module. Carte à puce assurant de nombreux services dans les téléphones GSM.

JEPES Joint Engineer Planning and Execution System. Plate-forme Java embarquée visant les équipements extrêmement contraints.

XIP eXecute-In-Place. Méthode d'exécution : Le code est exécuté directement à partir de la mémoire Flash où il est stocké.

JAR Java Archive. Un fichier ZIP utilisé pour distribuer un ensemble de classes ***Java***

SOCs System On Chip. un système complet embarqué sur une puce

ASM Abstract State Machine. Machines à états abstraits.

EEPROM Electrically Erasable Read Only Memory. Mémoire à contenu modifiable et non-volatile.

GSM Global System for Mobile communication. Le standard de la téléphonie mobile en Europe.

JCVM Java Card Virtual Machine. Machine virtuelle Java Card.

JCRE Java Card Runtime Environment. Environnement d'exécution Java Card.

JVM Java Virtual Machine. Machine virtuelle Java.

PIN Personal Identification Number. Code d'identification personnel protégeant l'utilisation d'un service.

RAM Random Access Memory. Mémoire à contenu modifiable et volatile.

ROM Read Only Memory. Mémoire à contenu non modifiable.

Bibliographie

[01] Sun Microsystems. “J2ME Building Blocks for Mobile Devices - Whitepaper on KVM and the Connected, Limited Device Configuration CLDC”. Sun Microsystems, May 2000.

[02] S. Loureiro, R. Molva, and Y. Roudier. Mobile Code Security. Institut Eurécom, Valbonne, France

[03] S. Park, H. Lim, H. Chang, et W. Sung. “Compressed swapping for NAND flash memory based embedded systems”. Lecture notes in computer science. Springer, Berlin, Allemagne, 2005.

[04] Syntec Informatique. Livre blanc des premières assises Françaises du logiciel embarqué. Collection Thématic, 2007.

[05] Syntec Informatique et BITKOM. Assises franco-allemandes de l’Embarqué. Communiqué de presse. Paris, juin 2008

[06] R. Smagghe, OPIIEC. “Etude sur le marché et les compétences autour des logiciels embarqués”. Synthèse pour la CNPE, Février 2009.

[07] J.F. Lécole. Adéquation entre les besoins en compétences et l’offre de formation dans les 10 métiers clés de l’informatique embarquée. Présentation aux Assises de l’Embarqué. Deuxièmes Assises Franco-allemandes de l’Embarqué, 2009

[08] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. The Pennsylvania State University.

[09] C. Mauriange. “Chronologie des grandes crises”. Manuscrit, Février 2009.

[10] L. Mullen. “**New** Missions Target Mars Moon Phobos”. Astrobiology Magazine, April 2009.

[11] E. Pilaud. Minalogic, rapport d’activité 2006.

[12] B. Brosgol, et J. Ruiz. “Ada enhances embedded-systems development”. EE Times: Design News, 2007.

[13] J.L. Lions. Ariane 5: flight 501 failure, report by the Inquiry Board. Technical report, European Space Agency (ESA) and Centre national d'études spatiales (CNES), July 1996.

[14] N.G. Leveson, et C.S. Turner. "Investigation of the Therac-25 accidents". IEEE Computer, 1993.

[15] N. V. Cong, Institut de la Francophonie pour l'Informatique, Travail d'intérêt personnel encadré, Rapport final, Réseau actif. Hanoi, Juillet 2005.

[16] Matt Stillerman, Dexter Kozen. Demonstration: Efficient Code Certification for Open Firmware. Proceedings of the DARPA Information Survivability Conference and Exposition, IEEE, 2003.

[17] G. C. Necula. « Proof-carrying code ». ACM Press, New York, Janvier 1997.

[18] Marche et competences dans le logiciel embarque, Etude pour le compte de l'OPIIEC. Juin 2008.

[19] A. Courbot. Spécialisation tardive de systèmes Java embarqués pour petits objets portables et sécurisés. Thèse de doctorat. Université des Sciences et Technologies de Lille. Septembre 2006

[20] G. C. Necula, P. Lee. The design and implementation of a certifying compiler. In Programming Languages Design and Implementation, New York, NY, USA, 1998. ACM Press.

[21] D. Kozen. Efficient Code Certification. Technical Report98-1661, Computer Science Department, Cornell University, January 1998.

[22] Adelstein F, illerman M, Kozen D. Malicious Code Detection for Open Firmware. Computer Security Applications Conference, 2002, Proceedings 18th Annual 9-13 December 2002.

[23] F. Boniol, G. Bel, et J. Ermont. "Trois approches pour la modélisation et la vérification de systèmes embarqués". Technique et science informatiques, Hermès, 2003.

[24] L. Casset, et J.L. Lanet. "A Formal Specification of the Java Byte Code Semantics using the B Method". Proceedings of the ECOOP'99 workshop on Formal Techniques for Java Programs, 1999.

[25] J. Koshy, et R. Pandey. "Remote incremental linking for energy-efficient reprogramming of sensor networks". In: Proceedings of the Second European Workshop on Wireless Sensor Networks, pp. 354-365, 2005.

[26] G. McGraw, et G. Morrisett. "Attacking malicious code: A report to the infosec research council". IEEE Software, 2000.

[27] R. Charpentier, et M. Salois ."Detection of Malicious Code in COTS Software via Certifying Compilers". Defence Research Establishment, QuCbec, Canada. 2000

[28] R. Anderson, et M. Kuhn."Tamper Resistance". *The Second USENIX Workshop on Electronic Commerce Proceedings*, Oakland, California, November 18-21, 1996.

[29] D. Abraham, G. Dolan, G. Double, et J. Stevens, "Transaction Security System", in *IBM Systems Journal* v 30 no 2 (1991) pp 206-229.

[30] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, et N. Tawbi. "Static Detection of Malicious Code in Executable Programs". Symposium on Requirements Engineering for Information Security (SREIS'01).

[31] C.E. Letwehr, A.R. Bull, J.P. McDermott, et W.S. Choi. "A taxonomy of computer program security aws". ACM Comput, 1994.

[32] K**.** Zurell . C programming for embedded systems. CMP Books, 2000.

[33] L. Burdy, L. Casset, and A. Requet. Développement formel d'un vérifieur embarqué de byte-code Java. Published in Technique et Science Informatiques (TSI) 22, 2003.

[34] The Project Monty Virtual Machine. JAVA™ 2 platform, Micro edition (J2ME™). White paper, Sun Microsystems, INC., 2002.

[35] Chevaux de troie. Fiche thématique 0003. Cyberworld Awarness and Security Enhancement Structure CASES.

[36] J. Boukhobza. Systèmes d'exploitation pour l'embarqué. Cour, Université de Bretagne Occidentale, Août 2007.

[37] P. Kocher†, R. Lee, G. McGraw, A. Raghunathan, et S. Ravi. "Security as a New Dimension in Embedded System Design". Annual ACM IEEE Design Automation Conference, 2004.

[38] G. Barthe. "EVEREST: Verification and Software Security". INRIA Sophia-Antipolis, France, November 2006.

[39] T. Ritter. Learning About Cryptography A Basic Introduction to Crypto, 2006

[40] S. Radack. "The cryptographic Hash algorithm family: Revision of the secure Hash standard and ongoing competition for new hash algorithms". National Institute of Standards and Technology. Gaithersburg, USA, march 2009.

[41] A.N. Alshamsi, et Takamichi Saito. "A Technical Comparison of IPSec and SSL". Tokyo University of Technology.

[42] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J.C Mitchell, et M. Horowitz. Architectural support for copy and tamper resistant software. Proceedings of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.

[43] T. Benavides. "The Enabling of an Execute-In-Place Architecture to Reduce the Embedded System Memory Footprint and Boot Time". Journal of computers, vol. 3, no. 1, January 2008.

[44] M. Kuhn. "The TrustNo 1 Crytoprocessor Concept". University of Erlangen-Nürnberg, Erlangen, April 1997.

[45] L. Casset, L. Burdy, et A. Requet. "Formal Development of an Embedded Verifier for JavaCard ByteCode". IEEE International Conference on Dependable SystemsNetworks, 2002.

[46] J.L. Lanet, A. Requet. "Formal Proof of Smart Card Applets Correctness". Third Smart Card Research and Advanced Application Conference, September, 1998.

[47] Syntec informatique, CAP'TRONIC**.** Assises Franco-Allemandes de l'Embarqué. Premiers Trophées de l'Embarqué, juin 2008.

[48] A. Requet. "A B model for ensuring soundness of a large subset ofthe Java Card virtual machine". Science of Computer Programming, pages 283 306, 2003.

[49] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A Realistic Typed Assembly Language. In Second Workshop on Compiler Support for System Software. ACM SIGPLANT, May 1999.

[50] D. Dollé, D. Essamé, and J. Falampin. "B dans le transport ferroviaire : L'expérience de Siemens". Technique et Science Informatique, 2003.

[51] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. ACM Transactions on Programming Languages and Systems. San diego California, USA, January 1999.

[52] E. Dupuis. Les Compilateurs certifies (Garantie et certification des outils de la chaine de production). Mini mémoire, 2005.

[53] Sécurisation du transfer de code grace à la signature: apercu technique. Document technique. VerSign France S.A.S, 2009.

[54] Comment signer numériquement du code téléchargeable pour le transfert de contenu sécurisé. Document technique. VerSign France S.A.S, 2009.

[55] Pour augmenter les téléchargements, instaurez d'abord la cnfiance: Le Code signing de VerSign certife l'éditeur et l'ntégrité du code. Document technique. VerSign France S.A.S, 2009.

[56] C. Kunz. Préservation des preuves et transformation de programmes. These de Doctorat. CMA- Centre de mathématiques appliquées, ENSMP. Février 2009.

[57] M. Hefeeda and B. Bhargava. On Mobile Code Security. Center of Education and Research in Information Assurance and Security And Department of Computer Science, Purdue University West Lafayette, IN 47907, U.S.A.

[58] S. Oaks. Java Security. O'Reilly & Associates, Inc., Sebastopol, CA, 1998.

[59] R. Oppliger, "Security issues related to mobile code and agent-based systems," Computer Communications 22, 1999, pp. 1165-1170.

[60] K. Kato. Safe and Secure Execution Mechanisms for Mobile Objects. Institute of Information Sciences and Electronics. University of Tsukuba, Japan.

[61] P. Lee, G. Necula. Reserach on Proof-Carrying Code for Mobile-Code Security. DARPA Workshop on Foundations for Secure Mobile Code. March 26-28, 1997.

[62] P. *Bon*, et *G. Mariano*. Développement de logiciels critiques pour les transports guidés, les Fiches d'activité scientifique de l' I N R E T S, AXE 3 – Accroître la fiabilité et la durabilité des systèmes de transport, optimiser leur consommation énergétique et réduire leur impact sur l'environnement. Institut national de recherche sur les transports et leur sécurité, France, avril 2009.

[63] J.G. Dyer, Lindemann, M., Perez, R., Sailer, R., Smith, S.W., van Doorn, L., Weingart, et S. Weingart. "The IBM Secure Coprocessor: Overview and Retrospective". IEEE Computer, October 2001.

[64] F. Painchaud. Increasing Efficiency, Flexibility, and Robustness in JavaTM Security. Defence Research & Development Canada – Valcartier. Technical Memorandum. November 2003.

[65] J. Koshy, et R. Pandey. "VMSTAR : synthesizing scalable runtime environments for sensor networks". Proceedings of the 3rd international conference on Embedded networked sensor systems, pp. 243-254, ACM Press, New York, NY, USA, 2005.

[66] Z. Chen. Java card technology for smart cards: Architecture and Programmer's Guide, Addison-Wesley Publishing Company, 2000.

[67] N. Shaylor, D. N. Simon, et W. R. Bush. "A java virtual machine architecture for very small devices". The 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, pp. 34 41, ACM Press, 2003.

[68] T. Cramer, R. Friedman, T. Miller, D. Seherger, R. Wilson, et M. Wolczko. "Compiling Java Just in Time : Using runtime compilation to improve Java program performance". IEEE Micro, Vol. 17, No. 3, pp. 36 43, 1997.

[69] B. Yee. "Using Secure Coprocessors". PhD thesis, Carnegie Mellon University, 1994.

[70] K.W. Hamlen, G. Morrisett, et F.B. Schneider. "Computability classes for enforcement mechanisms". Technical Report TR2003-1908, Cornell University, 2003.

[71] J. Gosling, et H. McGilton. The Java Language Environment : A white paper. Sun Microsystems, May 1995.

[71] M. Martel ."Interprétation abstraite et validation de logiciels critiques". Université de Perpignan Via Domita. Manuscrit, 2008.

[72] S.C. Grimal, "Preuve formelle d'un récupérateur de mémoire pour cartes à puce". Espace Mathématique Francophone 2003, Décembre 2003.

[73] A. Barreteau, et R. Lebreton. "Moniteurs matériels pour la sécurité des systèmes embarqués". l'université Bretagne-Sud de Lorient, France, 2007.

[74] D. Singelee, et B. Preneel. “The Wireless Application Protocol”. International Journal of Network Security, 2005.

[75] S. Horman. SSL and TLS : An Overview of A Secure Communications Protocol. The Security Mini-Conf at Linux.Conf.Au. Canberra, ACT, Australia. April 2005.

[76] E. Rohou, F. Bodin, et A. Seznec. “Salto: System for assembly-language transformation and optimization”. Sixth Workshop Compilers for Parallel Computers, Konferenzen des Forschungszentrums Jülich, Forschungszentrum Jülich, Aachen, December 1996.

[77] E. Gimnez, et O. Ly. “Formal Modeling and Verification of the Java Card Security Architecture: from Static Checkings to Embedded Applet Execution”. The verificard’02 meeting, Marseille, 7-9 January 2002.

Webographie

[78] ETAS. Publication de la nouvelle norme ISO/DIS 26262.
http://www.etas.com/fr/news-11867.php

[79] M. Coquard, et A. Grison. Images satellitaires terrestres avec une application sur système embarqué.
http://www.lita.univ-metz.fr/~paris/Cours_Multimedia/exposes/ImaSat.ppt

[80] Définition & Explications : Les paiements avec les téléphones portables : Les NFC : Near Field Communication. *http://marc-blanchard.com/blog/index.php/toc/toc*

[81] B. Crumley. Can an Artificial Heart Replace the Real Thing?
http://www.time.com/time/health/article/0,8599,1857216,00.html

[82] Using Embedded C++ In Embedded System Development .
http://www.ghs.com/wp/ppframe.htm

[83] VoIP Over WiFi WILL Disrupt the Cellular Industry
http://blog.tomevslin.com/2006/07/voip_over_wifi_.html

[84] IBM PCI Cryptographic Coprocessor.
http://www-03.ibm.com/security/cryptocards/pcicc/overview.shtml

[85] IT Security Certification. FIPS 140-2 Overview.
http://www.rycombe.com/short140.htm

[86] CERT Coordination Center. Vulnerability notes database.
http://www.kb.cert.org/vuls/

[87] Symantec Corporation. Latest Virus Threats.
http://www.symantec.com/avcenter/vinfodb.html

[88] Discretix Technologies LTD, Embeddiding security solutions.
http://www.discretix.com/

[89] S.J. Cartwright**,** et J. Smyth. The Reality Behind Moore's Law. 2008.
http://www.cyber-aspect.com/features/feature_article~art~104.htm

[90] Les virus informatiques : définitions. Symantec Corporation.
http://www.symantec.com/fr/fr/norton/theme.jsp?themeid=virus

[91] SODA Consortium. Intelligent transport challenges addressed.
http://www.soda-itea.org/Demonstrators/Automotive/default.html

[92] Qu'est-ce qu'une bombe logique?. *http://fr.tech-faq.com/logic-bomb.shtml*

[93] Introduction aux systèmes embarqués.
http://www.electroniciens.aquitaine-limousin.cnrs.fr/article135.html