Vers la sécurité mobile : caractérisation des attaques et contremesures

Jean-François Lalande

Centre Val de Loire UNIVERSITÉ
Spirit of the Loire Valley

UNIVERSITÉ D'ORLÉANS

UNIVERSITE D'ORLEANS

LABORATOIRE LIFO

HDR présentée par :

Jean-François Lalande

soutenue le : 30 novembre 2016

Discipline : Informatique

Vers la sécurité mobile : caractérisation des attaques et contremesures

RAPPORTEURS :

Yves Le Traon	Professeur, Université du Luxembourg
Jean-Yves Marion	Professeur des universités, Université de Lorraine
Fabio Massacci	Professeur, Università di Trento

JURY :

Sébastien Limet	Professeur des universités, Université d'Orléans Président du jury
Pascal Berthomé	Professeur des universités, INSA Centre Val de Loire
Sylvain Guilley	Professeur des universités, Telecom-ParisTech
Yves Le Traon	Professeur, Université du Luxembourg
Jean-Yves Marion	Professeur des universités, Université de Lorraine
Fabio Massacci	Professeur, Università di Trento

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

Remerciements

En premier lieu, je tiens à remercier les trois rapporteurs de ce manuscrit d'Habilitation à Diriger des Recherches, Yves Le Traon, Jean-Yves Marion et Fabio Massacci, ainsi que les membres du jury Pascal Berthomé, Sylvain Guilley, et Sébastien Limet.

Dans un deuxième temps, je souhaite remercier les collègues des deux équipes de recherche, SDS au LIFO et CIDRE à l'IRISA, dans lesquelles j'ai effectué mes activités de recherche. Je tiens aussi à partager ma gratitude avec les co-auteurs des publications évoquées dans ce manuscrit qui ont été un soutien indispensable, généré des discussions fertiles, et qui ont travaillé sans relâche pour produire une recherche de qualité. En particulier, je remercie Guillaume Hiet, Jérémy Briffaut, Pascal Berthomé, Patrice Clemente, Sébastien Gambs, Ludovic Mé. Je souhaite remercier trois collaborateurs particuliers avec lesquels le travail a été intense et particulièrement enrichissant : Karine Heydemann, Jacques Traoré et Valérie Viet Triem Tong. Je n'oublie pas les collaborateurs étrangers Luca Caviglione, Mauro Gaggero, Wojciech Mazurczyk, Waleed Smari et Steffen Wendzel avec qui, malgré la distance, nous avons pu établir des ponts entre nos domaines de recherche respectifs.

Enfin, je souhaite remercier les collègues enseignants-chercheurs, enseignants, personnels administratifs et techniques ainsi que les étudiants de l'INSA Centre Val de Loire pour le travail au quotidien.

Table des matières

Chapitre 1

Introduction

Ce manuscrit d'Habilitation à Diriger des Recherches présente une synthèse de mes travaux menés à l'INSA Centre Val de Loire dans le laboratoire LIFO (Laboratoire d'Informatique Fondamentale d'Orléans) depuis septembre 2005. Après ma thèse à l'Université de Nice - Sophia Antipolis sur le thème du dimensionnement des réseaux de télécommunication [281], j'ai intégré le LIFO pour participer aux activités de recherche en sécurité du laboratoire. Au début de cette reconversion thématique, j'ai commencé à travailler sur les politiques de sécurité et les systèmes de contrôle d'accès obligatoire. J'ai par la suite élargi le périmètre de mes recherches, à la fois d'un point de vue des problématiques abordées mais aussi des cas d'utilisations considérés.

La Figure 1.1 présente l'ensemble de mes publications scientifiques au travers d'une cartographie à plusieurs niveaux de lecture. Les références sont listées en fin de manuscrit. En colonne, on trouve les différents systèmes sur lesquels mes travaux ont porté ; en ligne, on peut voir les thématiques et objectifs principaux de mes contributions.

On trouve aussi en vert deux axes complémentaires lorsqu'on s'intéresse à la recherche en sécurité : comment protéger un objet digital et comment comprendre les malveillances commises ou qui pourraient survenir contre un de ces objets. La compréhension des malveillances est un axe qui nourrit les méthodes de protection ou de détection que l'on peut élaborer. Pour cette raison, ce manuscrit s'intéressera toujours aux attaques, à leur élaboration, leurs motivations, leur modélisation avant de s'intéresser aux mécanismes de protection qui permettent de s'en prémunir. Cet état de fait est d'autant plus vrai que pour chaque système étudié, en colonne, on remarque toujours deux groupes de publications, l'un portant sur la protection de ce système et l'autre portant sur l'analyse des malveillances pour ce système. Cela est rendu possible grâce à l'offre abondante en conférence en sécurité, certaines conférences ou workshops dédiés à la menace et d'autres plus généraux plutôt orientés sur les méthodes de protection. On remarquera une exception à cette remarque pour les systèmes distribués ou collaboratifs où l'analyse pure de la menace est très difficile puisqu'il faut disposer d'un système réel à étudier comme un réseau pair-à-pair, un réseau social, etc.

Je me suis intéressé à quatre grandes familles de systèmes. J'ai d'abord travaillé sur les systèmes distribués et les systèmes à haute performance. Ces systèmes ont la particularité de comporter un grand nombre de nœuds, parfois hétérogènes, et nécessitent un très haut niveau de sécurité quand ils hébergent des données ou des programmes précieux. Ces travaux se sont appuyés sur la collaboration avec le CEA qui dispose de tels clusters de calcul, et cette collaboration recherche a été pilotée par Christian Toinard pendant les premières années qui ont suivi mon arrivée. À cette occasion, j'ai aidé à l'encadrement de la thèse de Jérémy Briffaut [2]. J'ai aussi travaillé sur des systèmes plus distribués que des clusters de calcul, comme des réseaux pair-à-pair, pour lesquels les exigences de sécurité sont moins grandes et où la résilience du système et sa flexibilité deviennent plus importantes. En 2009, j'ai abordé un troisième système au travers de la thèse de Xavier Kauffmann-Tourkestansky [3], co-encadré avec Pascal Berthomé et financé par Oberthur Technologies. À cette occasion, j'ai travaillé sur la sécurité des cartes à puce, ce qui a conduit plus tard l'INSA à participer à l'ANR Lyrics (Lightweight privacY-enhancing cRyptography for mobIle Contactless Services). Puis, dans le même temps, j'ai commencé à étudier un système complémentaire à la carte à puce : les téléphones mobiles sous Android. Android étant basé sur un noyau Linux avec des applicatifs écrits en C et Java, un pont naturel technologique s'est formé entre les problématiques étudiées auparavant sur des clusters de calculs sous Linux et les nouvelles problématiques liées à l'utilisation des téléphones mobiles. Enfin, j'ai co-encadré avec Pascal Berthomé la thèse de Ghada Arfaoui [1] sur le thème de la sécurité des mobiles, des cartes à puce et des *Trusted Execution Environment*. En 2013, j'ai intensifié mes recherches sur la sécurité des téléphones mobiles en partant en délégation Inria dans l'équipe

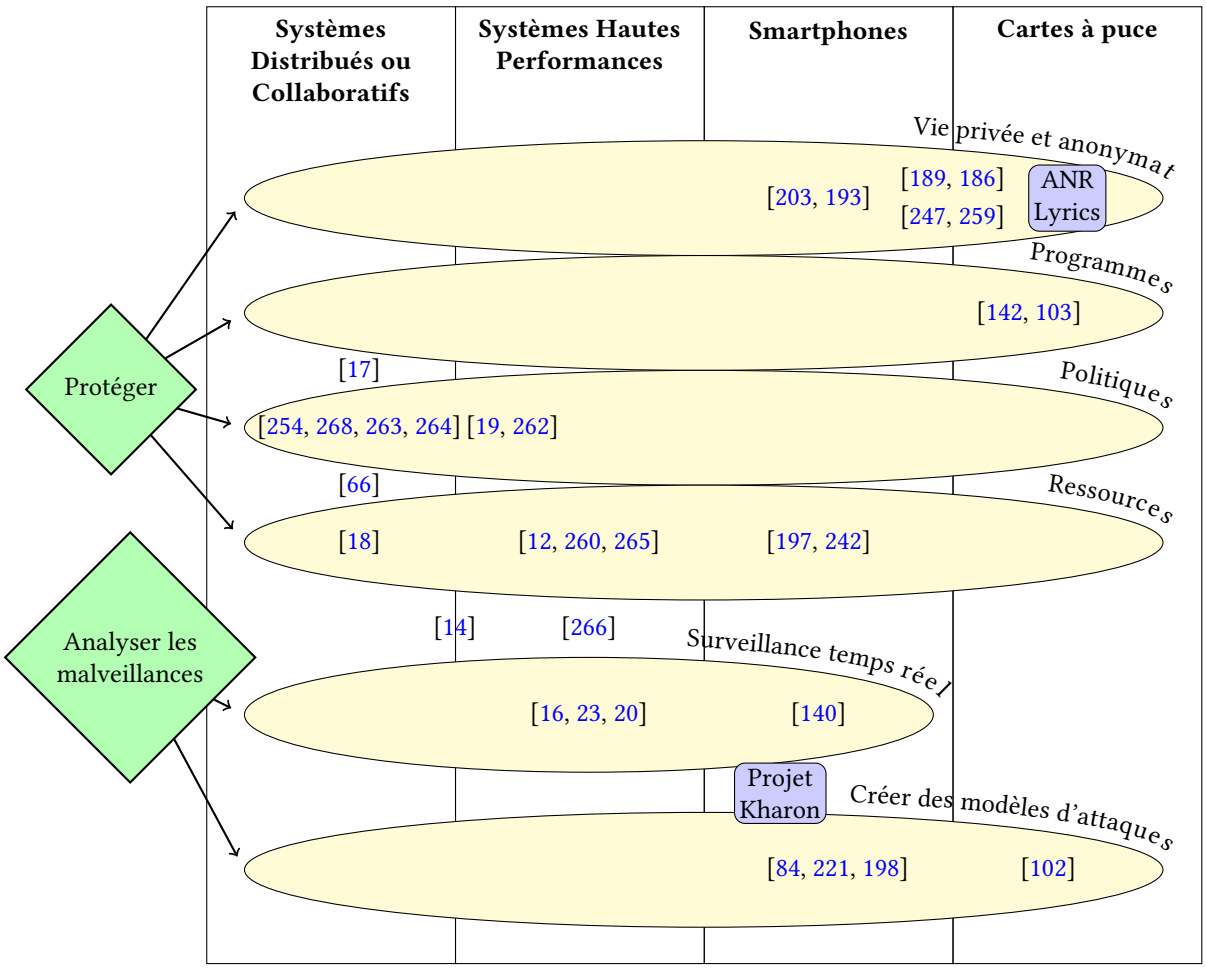


FIGURE 1.1 – Cartographie thématique des travaux menés

CIDRE basée à CentraleSupélec, sur le site de Rennes. Pendant deux années, j'ai noué de nouvelles collaborations avec cette équipe. Depuis, je participe au projet CominLabs Kharon qui s'intéresse à la caractérisation des *malware* Android et depuis 2015, je co-encadre la thèse de Mourad Leslous avec Valérie Viet Triem Tong sur le déclenchement et l'exécution de *malware* Android.

Pour chaque système que j'ai étudié, on peut qualifier plus finement les objectifs de recherche qui ont été traités. Pour les systèmes à haute performance et collaboratifs, je me suis principalement intéressé à la surveillance temps réel de ces systèmes, la conception de politiques de sécurité et la protection des ressources hébergées par ces systèmes. Pour les cartes à puce et les téléphones mobiles, je me suis focalisé sur la création de modèles d'attaques ou de nouvelles attaques, la protection des données de l'utilisateur et des programmes exécutés. Ces disjonctions dans ma recherche s'expliquent de par la nature des systèmes étudiés, les objectifs de sécurité et surtout la nature des attaques perpétrées. Par exemple, une attaque au travers du réseau suivi d'un *exploit* noyau sur un cluster est tout à fait différent d'un logiciel malveillant sur un téléphone mobile ou d'une attaque physique réalisée contre une carte à puce.

Malgré ces différences, je propose au lecteur de ce manuscrit une lecture en trois étapes de chaque contribution. Dans un premier temps, je propose d'étudier **les attaques** au travers de leurs modèles ou de la conception de démonstrateur de faisabilité. Cela permet d'une part de mieux comprendre les menaces à prendre en compte et d'autre part de dégager des modèles aidant à la conception de solutions de sécurité. Dans un second temps, je décrirai les **contremesures** proposées en décrivant le plus synthétiquement possible mais avec précision la méthode de protection ou de détection. Enfin, dans un troisième temps, j'aborderai **l'évaluation expérimentale et/ou formelle** de ces contremesures qui permettent de valider l'approche et d'en mesurer la portée. Cette démarche en trois temps sera redondante pour chaque chapitre.

Ce manuscrit est découpé en trois grands chapitres. Après ce chapitre introductif, le Chapitre 2 s'intéresse à la problématique de la conception de politiques pour les grands systèmes de calculs ou coopératifs en vue de protéger les ressources de ces systèmes. Le chapitre 3 traite de la sécurité du flot de contrôle pour les cartes à puce et les téléphones mobiles. Enfin, le chapitre 4 traite de mes contributions autour de la protection de la vie privée pour les systèmes opérant sur cartes à puce et téléphones mobiles. Le chapitre 5 conclut ce manuscrit.

Références

[1] G. Arfaoui. **Conception de protocoles cryptographiques préservant la vie privée pour les services mobiles sans contact**. Thèse de doctorat. Université d'Orléans, novembre 2012 (cf. p. 5, 80).

[2] J. Briffaut. **Formalisation et garantie de propriétés de sécurité système : application à la détection d'intrusions**. Thèse de doctorat. Université d'Orléans, décembre 2007 (cf. p. 5, 26).

[3] X. Kauffmann-Tourkestansky. **Analyses sécuritaires de code de carte à puce sous attaques physiques simulées**. Thèse de doctorat. Université d'Orléans, novembre 2012 (cf. p. 5, 37).

Chapitre 2

Conception de politiques

Lorsqu'on s'intéresse aux systèmes à large échelle, des clusters de calcul aux infrastructures de type *cloud computing*, on se focalise souvent sur les questions de perfomances, de scalabilité, ou de maintenabilité. La sécurité de ces systèmes vient le plus souvent dans un second temps. Or, assurer la sécurité de ces systèmes nécessite un savant dosage de plusieurs techniques, à la frontière entre la sécurité système, réseau et logiciel [78]. La difficulté, dans ce type d'architecture, est multiple. L'interdépendance des systèmes au niveau réseau menace la disponibilité globale du système en cas d'attaque d'un des nœuds. L'encapsulation grandissante des logiciels dans des conteneurs légers et machines virtuelles complexifie les défenses à mettre en place et multiplie les vecteurs d'attaque. De plus, le matériel ou les systèmes d'exploitation peuvent être hétérogènes, à cause de l'évolution de leur taille ou le remplacement de nœuds. Même pour deux matériels identiques, il est fréquent d'avoir des rôles différents, un nœud ayant un rôle de coordinateur, pendant que l'autre est un exécutant. Enfin, l'administration de grands systèmes ne peut se faire manuellement. Comme évoqué par Yurcik *et al.* [78], un cluster à la NSA peut comporter 1 500 nœuds [1], ce qui exclut une administration manuelle. Ainsi, tous ces éléments tendent à interférer avec la mise en place de mécanismes de sécurité. Si l'on vend la disponibilité d'un service de calcul ou d'hébergement, la sécurité ne rapporte pas d'argent mais peut en faire perdre.

Si l'on met de côté les aspects de la sécurité et de la surveillance du réseau, la sécurité d'un système à large échelle repose sur le contrôle des actions effectuées au sein de chaque système. Il faut donc définir ce qui est autorisé ou interdit au sein d'un système ou d'un ensemble de systèmes. On appelle une politique de sécurité l'ensemble de ces règles, exprimées dans un langage permettant à un système de contrôle d'accès et son point d'application, un PEP (*Policy Enforcement Point*), d'appliquer la politique de sécurité. Concevoir une politique n'est pas trivial : le pouvoir d'expression de la politique est borné par le langage utilisé et éventuellement par les capacités du PEP à la mettre en œuvre. Une politique dépend très fortement du cas d'utilisation considéré et plus son raffinement est important, plus son utilisation pratique est difficile. Il y a donc un réel effort à fournir pour rendre la politique utilisable par son administrateur et notamment au travers d'outils pour manipuler la politique ; rendre la politique déployable sur le système cible afin de garantir qu'elle soit utilisable sur des grands systèmes ; rendre une instance d'une politique conforme et non ambigüe, ou à tout le moins, vérifiable. Ce sont ces préoccupations qui sous-tendent les travaux présentés dans ce chapitre.

Cependant, avant de se lancer dans la conception d'une politique d'envergure pour un système à large échelle, il est essentiel de comprendre les moyens de l'attaquant pour menacer de tels systèmes. L'approche classique consiste à capturer des attaques dans un pot de miel, c'est-à-dire un système laissé à disposition d'un attaquant en lui laissant croire qu'il s'agit d'un système réel en production. Plus le pot de miel est proche d'un système réel, plus l'attaquant aura la conviction d'avoir pénétré un vrai serveur mais plus son attaque sera difficile, surtout si l'on ajoute des composants de sécurité. A l'inverse, un pot de miel pauvre en interactions, c'est-à-dire dont la majeure partie est simulée sera sans doute décourageante pour l'attaquant (et détectable). Or, tout au long de ce chapitre, nous considérons un attaquant doté d'une intelligence et d'une détermination certaines, ayant pour mission d'attaquer le système à des fins criminelles. Evidemment, si l'on cible de telles attaques, il est très difficile de les étudier et d'en avoir une vue précise. Cependant, il est intéressant de se familiariser avec les procédures d'attaque classiques et de caractériser quantitativement les attaques (fréquence, durée, localisation, etc.). Cela permet ensuite de mieux paramétrer les systèmes de défense que l'on met en place.

1. A la date de 2004 [78], ce qui a largement évolué depuis.

Dans ce chapitre, j'introduis tout d'abord les systèmes à large échelle (section 1) et notamment les clusters de calcul sur lesquels ont porté une partie de mes travaux. J'explique l'intérêt et les difficultés de la conception de politique de contrôle d'accès obligatoire pour ces types de systèmes et je donne quelques éléments sur une technologie particulière, SELinux, qui seront utiles pour la suite. Je présente mes contributions sur la thématique de la conception de politique et comme annoncé dans l'introduction de ce manuscrit, je procède en deux temps : je présente tout d'abord comment on peut étudier les attaques (section 2) et je bascule ensuite sur la conception de politiques de contrôle d'accès sur trois cas d'études particuliers (section 3). Pour chaque cas d'étude, je montre comment une évaluation peut être réalisée, soit formelle, soit plus expérimentale.

La partie traitant des attaques (section 2) est purement expérimentale : l'enjeu consiste à capturer des attaques réelles pour en tirer des informations quantitatives et pour mesurer la difficulté technique de la sécurisation d'un système. Il n'y a pas réellement de modèle d'attaque qui émerge de l'étude mais on arrive cependant à se faire une idée générale d'un processus d'attaque classique sur un système Linux. Cette partie de l'étude est directement liée avec la partie qui suit directement celle-ci, à savoir la sécurité des grands clusters de calcul.

Dans la section qui suit (section 3), je propose trois contributions autour de la conception de politiques de sécurité. Tout d'abord, en lien avec l'étude des attaques, je montre comment sécuriser un cluster de calcul multi-utilisateurs contre des malveillances internes ou externes. Les solutions techniques apportées sont directement liées à la nature du cas d'étude, notamment sur l'aspect très important des performances. Une deuxième contribution concerne les politiques de sécurité pour les systèmes collaboratifs à large échelle. Le cas d'étude proposé est un logiciel de gestion de crise humanitaire. Les contraintes sont ici complètement différentes du cas précédent, tant d'un point de vue technique que d'un point de vue des propriétés de sécurité visées par la politique. Je termine enfin cette section avec une contribution sur l'administration et le contrôle des politiques lorsque celle-ci sont dynamiques. Pour chacun de ces cas d'étude, je donne une évaluation expérimentale de ces solutions.

Je termine le chapitre en décrivant brièvement les logiciels développés relatifs à ces contributions et je donne quelques perspectives de recherche.

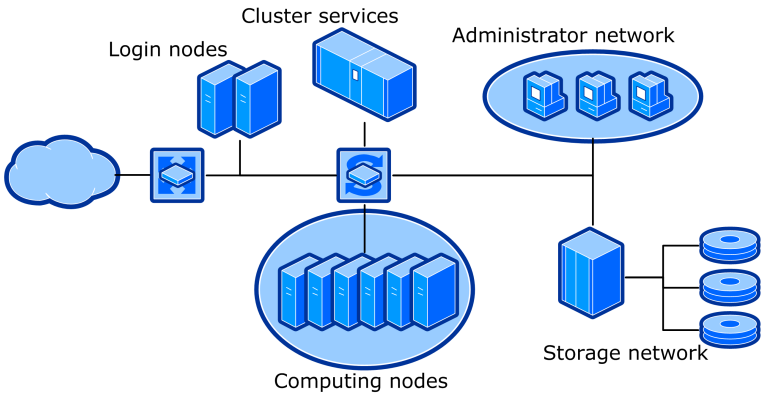


FIGURE 2.1 – Représentation classique d'un cluster de calcul

1 Introduction à la sécurité des systèmes à large échelle

1.1 Les systèmes à large échelle

Historiquement, beaucoup de grands systèmes sont basés sur des distributions GNU/Linux, abrégé en "Linux" par la suite. Ces distributions sont modulaires, *open-source* et ont été le terreau de nombreux outils de sécurité. La plupart des systèmes à large échelle utilisent des variantes de Linux comme cœur des systèmes et bâtissent autour des logiciels pour leur gestion à un plus haut niveau.

Un cluster de calcul est un bon exemple d'un grand système dont l'objectif est de fournir une puissance de calcul partagée entre des clients. Comme représenté en figure 2.1, il est composé de différents services, de plusieurs sous-réseaux et de différents accès. Les nœuds sont hétérogènes (*login nodes*, *computing nodes*, *storage nodes*). On accède au cluster depuis l'extérieur directement en console ou par une interface graphique. On peut ensuite se rendre sur un nœud ou configurer des tâches pouvant s'exécuter sur un ou plusieurs nœuds. Les logiciels de gestion du cluster, du point de vue de l'utilisateur, restent sommaires. Un cluster répond au besoin primaire de pouvoir exécuter des processus, éventuellement parallélisables, le plus rapidement possible.

On distingue un cluster de calcul d'une grille de calcul qui est un concept plus général. Une grille de calcul offre un ensemble de services, gérés par des *middleware* et dont l'objectif est de fournir des services aux utilisateurs. Le système gérant la grille est souvent beaucoup plus complexe qu'un cluster de calcul et s'apparente plus à l'ancêtre de l'informatique en nuage, sans son élasticité. Dans une grille, des *middleware* gèrent l'authentification, la délégation des droits et possède des services d'autorisation plus avancés. Les politiques de sécurité peuvent s'adapter aux requêtes des utilisateurs [30] et s'appuient, comme les clusters, sur des mécanismes classiques bas niveau du système d'exploitation mais aussi à des mécanismes de sécurité de plus haut niveau pour les *middleware*.

Un autre type de système à large échelle est le réseau totalement décentralisé aussi appelé réseau pair-à-pair. Il s'agit d'un type d'architecture qui se passe d'un serveur central. En général, chacun des utilisateurs joue un rôle équivalent au sein du réseau, bien que certains protocoles permettent d'élire certains nœuds pour leur confier des tâches différentes. De manière très générale, l'intérêt d'un tel système est de tolérer les pannes et d'être *scalable*. D'un point de vue de la sécurité, il est très difficile de garantir des propriétés de manière certaine. En effet, chaque nœud du réseau doit appliquer la politique qui est elle-même décidée d'un accord commun. Le meilleur exemple d'un tel succès est le réseau Bitcoin, où la validité de la *blockchain* est décidée à la majorité [29].

1.2 Le contrôle d'accès obligatoire

Dans un tel système, la sécurité ne peut pas reposer sur l'observation unique des communications entre les nœuds. Une attaque peut tout à fait passer inaperçue sur le réseau et l'observation ou la prévention doit se faire au niveau des systèmes, de surcroit si le traffic est chiffré. Ainsi, deux grandes familles de contrôle d'accès se sont développées sur les systèmes de type Linux : le contrôle d'accès discrétionnaire [45] (DAC), géré par l'utilisateur, et obligatoire [7] (MAC), géré par un tiers. En utilisant une politique de contrôle d'accès, on peut alors préserver la confidentialité et l'intégrité des ressources du système, si celle-ci est correctement configurée et appliquée par le système et si, de surcroît, on est réellement sûr qu'un attaquant ne peut pas la contourner.

Un nombre important de politiques de contrôle d'accès a été proposé dans la littérature et il serait fastidieux de faire un historique complet. Une politique typique appliquée dans les systèmes actuel est RBAC pour *Role-Based Access Control* [60] qui associe à des rôles des capacités quand l'utilisateur choisit un rôle. Une autre politique intéressante est DTE pour *Domain and Type Enforcement* [13] où l'on se focalise sur l'interaction entre les sujets du systèmes (par exemple les utilisateurs) et les objets (par exemple des données). On définit alors des domaines pour autoriser des ensembles de permissions sur des groupes de sujets et les types permettent de faire de même sur des groupes d'objets.

Quand on souhaite implanter une politique de contrôle d'accès de type MAC dans un système d'exploitation réel, les difficultés surgissent. Par exemple, le modèle Bell-LaPadula [7] introduit la notion de "No Read up" and "No Write down" afin de garantir la confidentialité des données entre différents niveaux. Ces niveaux permettent d'empêcher des sujets d'accéder à des objets de niveaux supérieurs afin d'éviter des lectures ou écritures non autorisées. Cette règle est difficile à mettre en œuvre d'un point de vue technique, par exemple dans le répertoire */tmp* où tout processus peut créer un fichier. Biba a proposé le modèle dual [10] de Bell-LaPadula en inversant les règles d'accès : la propriété obtenue est l'intégrité des données au lieu de la confidentialité. Les problèmes implémentatoires ne sont pas résolus pour autant. Ainsi, une politique DTE est plus simple à mettre en œuvre dans un système réel : on confine les ressources accédées par des programmes, même celles accédées par l'utilisateur *root*.

L'implémentation de mécanismes de contrôle d'accès dans les clusters Linux s'est appuyée sur les outils proposés dans les distributions GNU/Linux. Plusieurs tentatives comme Medusa [51], RSBAC [55], LIDS [11], ont conduit à l'émergence de deux technologies majeures : SELinux, un résultat du projet DTOS [53], et grsecurity [70]. Ils implémentent tous deux les modèles RBAC et DTE.

1.3 SELinux

Quand SELinux a été introduit dans le noyau Linux comme un patch, le binaire de la politique était un unique fichier monolithique [50] que l'on pouvait charger dans le noyau. Cette politique allant en augmentant, elle a été découpée en fichiers et on manipule désormais des modules de politique.

La politique utilise des contextes de sécurité associés à des processus et des fichiers [65, 33]. La politique exprime alors comment interagissent les contextes de sécurité entre eux. Un contexte de sécurité SELinux est composé de trois identifiants, l'utilisateur SELinux (indépendant de l'utilisateur du système d'exploitation), le rôle (lié à RBAC) et le type (lié à DTE), et deux *range* optionnels. Les *range* sont en général des niveaux MLS (Multi-Level Security) et des catégories MCS (Multi-Category Security) permettant encore plus de flexibilité dans la politique. Un exemple de contexte peut être *(root, system_r, unconfined_t, s0:c0.c1023)* : l'utilisateur *root* SELinux avec le rôle *system_r* pour les activités systèmes et le type non confiné *unconfined_t* plus le range *c0.c1023*. Ce contexte est typiquement utilisé pour un processus *root* qui n'aura pas de restriction dans ses actions.

Les règles SELinux mettent en relation des contextes de sécurité sujets et objets, qui sont tous deux des triplets comme vu précédemment. On exprime une règle sous la forme *allow Source Target:Class Permission;* où l'on accorde une permission au contexte de sécurité Source sur le contexte de sécurité *target* pour une certaine classe d'objets (fichier, socket, etc.). D'autres types de règles peuvent être exprimés, notamment pour faire transiter un contexte de sécurité d'un processus qui *fork* vers un autre.

Les avantages et désavantages de l'utilisation de politiques SELinux ont été largement débattus [32]. Avec le recul, SELinux est déployé dans de nombreuses distributions et a maintenant conquis nos téléphones mobiles. Il est en effet déployé par défaut dans les versions d'Android supérieures à 5. L'aspect modulaire de la politique permet aux industriels d'implémenter des politiques spécifiques à leurs cas d'usage [74], ce qui sera utilisé plus loin en section 3.1 lorsque nous présentons la sécurisation d'un cluster de calcul.

Avant d'aller plus avant sur la sécurité des systèmes à large échelle en section 3, en particulier sur les aspects politiques, nous nous intéressons tout d'abord à la caractérisation des attaques sur de tels systèmes. En se familiarisant avec les pratiques des attaquants, il devient ensuite plus facile de concevoir des politiques adaptées à de tels systèmes.

2 Collecte et analyse d'attaques

Pour mieux comprendre les enjeux de la sécurité des systèmes Linux, notamment dans les clusters de calcul, nous avons entrepris d'étudier des attaques de systèmes réels. Pour ce faire, il faut disposer de systèmes ayant été compromis et analyser une photo *post-mortem* de l'attaque. D'une part, cela nécessite d'avoir par exemple

accès à des serveurs d'entreprise ayant été attaqués, et d'autre part, cela requiert d'avoir de solides compétences en *forensic* [75]. Par ailleurs, si l'attaquant est précautionneux, il cherchera à couvrir ses traces, ce qui augmente la difficulté de l'analyse. N'étant pas l'ANSSI et n'ayant pas de doctorant travaillant dans une entreprise du domaine, l'approche que l'on pourrait considérer comme plus "académique" consiste à monter un pot de miel pour capturer les attaques et à le rendre disponible sur internet [43]. C'est ce que nous avons réalisé pendant toute l'année 2008.

Écrivons directement en préambule de cette partie ce qui relève de l'espoir par rapport à la réalité des attaques capturées : on capture beaucoup de bruit et d'attaques automatisées que de réelles attaques dues à des humains et élaborant des stratégies de pénétration très poussées. D'un point de vue de la recherche, la valeur des informations collectées est donc toute relative mais reste cependant intéressante pour plusieurs raisons. Une expérience aussi longue nécessite de mettre en œuvre une infrastructure robuste et sécurisée, sous peine de devoir répondre à des coups de téléphones de RENATER, ce qui est survenu une fois pendant l'expérience. Les techniques mises en œuvre permettent donc de tester, en interne, des solutions de sécurité. D'autre part, l'observation des attaques, même simples, est source d'idées pour la conception plus formelle de propriétés de sécurité. Enfin, les données collectées permettent de disséquer certaines attaques et dans le même temps de réaliser un panorama statistique des attaques. Ces analyses n'étant que très partiellement communiquées par les entreprises, elles possèdent donc une certaine valeur intrinsèque.

2.1 Les pots de miel hautes interactions

Contrairement aux pots de miel basses interactions qui émulent une partie limitée d'un système réel [59], un pot de miel hautes interactions est un système réel accueillant des attaquants. Sur ces types de pots de miel, la littérature est donc plus restreinte que pour les pots de miel basses interactions. Par ailleurs, les pots de miels ne sont plus à la mode, en plus d'être mal tolérés par les RSSI des Universités…

Au début des pots de miel hautes interactions, la difficulté résidait dans leur mise en œuvre technique, notamment à cause de la réinstallation fréquente et nécessaire du système. Avec l'arrivée de la virtualisation, ceux-ci sont devenus plus aisés à administrer [5] mais il fallait prendre soin de cacher l'hyperviseur sous peine d'être découvert par l'attaquant. Avec le temps, la virtualisation s'étant démocratisée dans l'entreprise, un attaquant n'était plus découragé par la découverte d'un système virtualisé. D'autres architectures ont été proposées comme des pots de miel hybrides [6]. Un pot de miel basse interaction est utilisé pour la première partie de l'attaque jusqu'à reconnaitre le protocole. Puis, une fois le protocole reconnu, le début de l'attaque est rejoué et se poursuit avec une réplique du logiciel ciblé afin de faire avancer l'attaque.

Sécuriser un pot de miel correctement est une tâche difficile. En effet, un pot de miel trop facile à pénétrer pourrait laisser penser qu'il ne s'agit pas d'un système réel. En revanche, un pot de miel trop bien caché et trop sécurisé risque de ne capturer que peu d'attaques. Pour résoudre ce dilemme, plusieurs solutions simples ont été proposées. Hecker *et al.* [39] proposent de configurer un pot de miel dynamiquement avec les *scans* d'attaques réseau. Kuwatly *et al.* [44] construisent un pot de miel qui peut être branché à un réseau local et détecte automatiquement les IPs non utilisées afin de les occuper. Cela permet de faciliter le déploiement d'un grand pot de miel, par exemple au sein d'une université. Sobesto *et al.* ont par exemple déployé un pot de miel basses interations sur 2 000 IPs de l'Université du Maryland [68].

D'un point de vue de l'attaquant, la recherche s'est intéressée à la détection des pots de miel [40, 41]. Les techniques utilisées s'appuient sur l'analyse du noyau afin de différencier une installation Linux sur un système réel d'un système virtualisé, du temps où la virtualisation n'était pas largement utilisée. A cette époque, il était donc important de déployer un pot de miel sur un système réel pour maximiser les chances de capturer une attaque.

Dans l'ensemble de ces travaux, peu d'approches déploient des pots de miel hautes interactions. La difficulté est d'en assurer la maintenance, la sécurité, notamment pour éviter les attaques par rebonds. Dans la suite, nous proposons une architecture pour résoudre ce problème.

2.2 La sécurisation d'un pot de miel hautes interactions

Nous avons proposé une architecture sécurisée de pot de miel hautes interactions [20]. Notre objectif est double : offrir un terrain d'attaque sur des machines réelles à l'attaquant et minimiser nos efforts d'administration, malgré les attaques. Notre pot de miel est représenté en figure 2.2. Il est constitué de quatre parties : la gestion du réseau, le cluster de machines réels hébergeant les attaques, la collecte des traces des différentes sondes de sécurité et la partie corrélation de données.

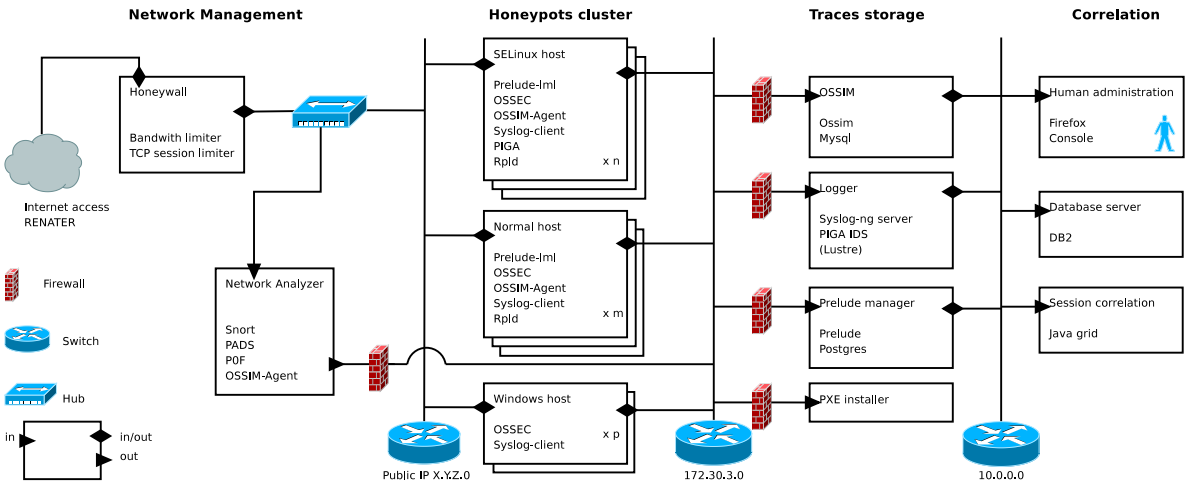


FIGURE 2.2 – Architecture d'un pot de miel hautes interactions

L'administration du réseau est primordiale dans l'architecture. Outre les différents cloisonnements afin d'éviter qu'un attaquant "fuite" vers l'intérieur du réseau local (dans la partie de collecte des traces ou de corrélation), il faut limiter la capacité de l'attaquant à perpétrer des attaques depuis la machine du cluster compromise vers internet. Pour ce faire, plusieurs règles réseau sont implantées dans un *Honeywall*, une machine en charge du filtrage entre le pot de miel et l'extérieur. Ces règles limitent les capacités de l'attaquant mais lui permettent de réaliser des opérations réseau "raisonnables" :

- Limitation de la bande passante sortante à 10MB par heure ;
- Limitation du nombre de connexions TCP à 100 par heure ;
- Limitation de la bande passante entrante à 100MB par heure.

Le cluster de machines qui hébergent les systèmes attaqués est quant à lui sécurisé en utilisant SELinux pour une partie. Ainsi, il n'est pas nécessaire de réinstaller ces machines, sauf si un opérateur humain constate ou suspecte une attaque ayant pu endommager le système ce qui n'est pas facile à voir et que nous n'avons jamais constaté. Pour les machines dites "normales" c'est-à-dire sans SELinux ou avec Windows, il faut les réinstaller fréquemment avec le serveur PXE.

Lorsque l'attaquant se connecte au pot de miel sur une des IPs publiques, il faut lui fournir une machine et un compte utilisateur. En s'inspirant des travaux de Hecker *et al.* [39], nous avons modifié le serveur SSH pour accepter aléatoirement 1% des tentatives de *login/password*. Une fois accepté, le compte utilisateur est créé dynamiquement sur une des machines prise au hasard. Nous avons plus tard perfectionné cette méthode [23] en la déclinant dans un *cloud* local sous Eucalyptus, compatible avec Amazon EC2. Le *cloud* alloue une machine virtuelle dynamiquement à l'attaquant et se rappelle de l'association entre le login utilisé et la machine virtuelle allouée. Ainsi, on peut éteindre la machine virtuelle en sauvegardant les données utilisateur et si l'attaquant revient on peut la relancer en restaurant l'état de la machine. Cette solution permet d'avoir un pot de miel élastique en cas d'attaques multiples simultanées, mais surtout de faire croire à l'attaquant que son système est persistant alors que l'on économise des ressources en éteignant les machines virtuelles non utilisées. Ainsi, avec ces économies, on peut fournir à l'attaquant des systèmes virtuels plus conséquents en terme de ressources CPU et mémoire afin de lui faire croire qu'il a pénétré un vrai serveur.

2.3 Analyse des activités d'attaque collectées

Les activités collectées ont été analysées de deux manières différentes. D'une part, une analyse manuelle assistée de scripts permet de caractériser les attaques. D'autre part, des outils de corrélation permettent d'aller plus loin dans les investigations, notamment lorsqu'on souhaite corréler différentes alertes entre elles. Nous donnons dans cette section quelques éléments illustratifs de ce que l'on peut obtenir en terme de compréhension des attaques collectées dans notre pot de miel hautes interactions [16].

La table 2.1 et la figure 2.3 caractérisent les sessions d'attaque en fonction du nombre de commandes entrées par l'attaquant dans le shell obtenu. De nombreuses sessions ne sont pas exploitées. Si elles le sont, on

Table 2.1 – Nombre de sessions en fonction de la taille de l'historique

Session type	Number
no history	1627
empty history	7
history < 3 lines	49
history >= 3 lines	173
Sum	1849 / 1856 (total)

Table 2.2 – Classification des sessions d'attaque

Type of session	.bash_history	rpld
Malware	36%	42%
Inspection	9%	17.2%
IRC bot	9%	9%
Local Network Inspection	2.7%	2.7%
Unrecognized sessions	43%	29%

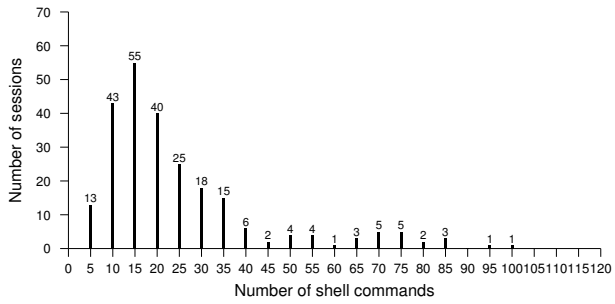


Figure 2.3 – Nombre de sessions par taille de session

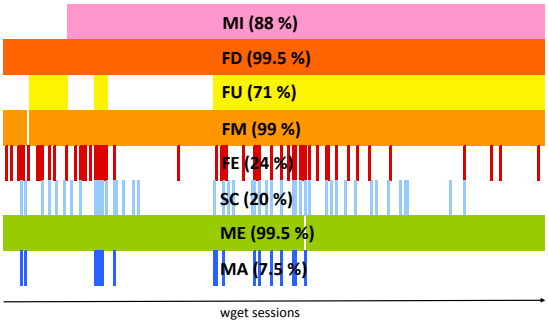


Figure 2.4 – Classification des 223 sessions "wget"

voit clairement que les sessions comportent entre 5 et 35 commandes. A partir de l'analyse des commandes de l'historique du bash ou bien de l'outil *rpld* qui permet d'enregistrer les frappes d'une console, nous avons caractérisé les sessions en fonction des noms de commandes tapés, comme montré en figure 2.2. On constate que de nombreuses sessions servent à lancer des logiciels malveillants, notamment pour tenter d'exploiter des vulnérabilités système. On remarquera la différence de pourcentage pour les sessions d'inspection : l'attaquant pense à nettoyer le *.bash_history* avant de se déloguer de ce type de sessions, mais ne peut éviter d'être trahi par la capture de *rpld*.

Nos procédures de corrélations ont permis d'aller plus loin dans l'analyse de ces sessions. En loguant l'ensemble des appels système générés par ces sessions, on obtient des millions d'évènements systèmes à traiter. Nous nous sommes focalisés sur les 223 sessions où un téléchargement a été opéré. En interrogeant les données collectées, nous avons construit une classification plus fine de ces 223 sessions représentée en Figure 2.4. La majorité des sessions réalise des inspections (MI), des téléchargements de fichiers (FD), des manipulation de fichiers (FM) et des exécutions de logiciels malveillans (ME). Il manque cependant la détection de 0.5% d'appels systèmes montrant un téléchargement, à moins que le téléchargement ait échoué. L'édition de configuration (FE) ou la compilation d'outils (SC) est plus marginal. Certaines sessions sont classées comme des sessions comportant une demande d'accès à l'administration de la machine, le plus souvent après avoir exécuté un logiciel malveillant, comme par exemple un *exploit*. Avec ce type d'analyse, on apprend donc que les attaquants téléchargent leurs programmes et ne les compilent quasiment pas.

Ce travail a été poursuivi dans le cadre d'un projet pédagogique avec les étudiants de l'INSA Centre Val de Loire et d'un stage ingénieur. Il a permis de réaliser Synema [14], un démonstrateur de SIEM (*Security Information Management System*) que nous décrivons dans la section 4.

Les résultats obtenus sur nos pots de miel hautes interactions montrent une séquence d'attaques classique : après avoir pénétré un système, l'attaquant cherche à contourner les mécanismes de sécurité pour obtenir plus de droits sur le système compromis. Il est donc important de concevoir des contremesures pour une telle hypothèse. Ces contremesures s'appuieront sur la construction de politiques de sécurité, principalement en vue de déployer du contrôle d'accès obligatoire. Ainsi, par la suite, nous proposons de décrire trois cas d'usage pour trois besoins en sécurité différents.

3 Conception de politiques de sécurité

3.1 Contrôle d'accès d'un cluster multi-utilisateurs

Dans cette partie, on considère un besoin spécifique : sécuriser un cluster utilisateur utilisé pour des calculs hautes performances. Les utilisateurs accèdent au cluster comme présenté précédemment en figure 2.1. Ils co-existent donc sur des mêmes nœuds de calcul et souhaitent accéder à leurs données propres, sans interférence aucune. C'est un besoin classique de confinement ou d'isolation. Cependant, nous souhaitons rajouter une contrainte forte : on suppose qu'une vulnérabilité potentielle peut exister sur un nœud de calcul du cluster ou d'accès au cluster et qui n'est pas connu à l'avance. Ainsi, une telle vulnérabilité pourrait être exploitée par un des utilisateurs afin d'escalader ses privilèges et devenir *root*. Avec une telle hypothèse, un contrôle d'accès discrétionnaire n'est pas suffisant. L'objectif est donc de présenter une architecture complète de déploiement d'un mécanisme de contrôle d'accès répondant à ce problème.

3.1.1 Le contrôle d'accès et les performances

Une des premières approches qui s'intéresse à l'intégration de contrôle d'accès obligatoire dans un cluster a été proposée par Pourzandi *et al.* [57]. Travaillant chez Ericsson, ils proposent un *framework* de déploiement de politiques avec un langage de politique spécifique, n'ayant encore que peu de fonctionnalités par rapport à celui de SELinux. Ce framework a été abandonné par la suite et ne fonctionne plus sur des distributions récentes de Linux. D'autres compagnies se sont essayées à l'exercice, sans pour autant fournir toutes les descriptions des politiques employées, comme par exemple NimbusOS de Linux Labs International. Il est probable que le langage de politique utilisé fût celui de SELinux, utilisé principalement pour l'application de règles de sécurité réseau. De nombreuses solutions industrielles ne sont pas publiées et l'information reste donc difficile d'accès.

En 2005, Leangsuksun *et al.* [46] présentent une architecture complète pour la sécurité de clusters. Leur but est de garantir la disponibilité d'un cluster en intégrant HA-OSCAR [38] et la solution de Pourzandi *et al.* [58, 56]. HA-OSCAR fournit une architecture transparente pour déployer ou re-déployer des systèmes dans le cluster. Chaque service est monitoré par un démon et est réparé en cas de panne. DSI implémente un mécanisme de contrôle d'accès obligatoire au niveau noyau, permettant de garantir l'isolation des processus. La politique est similaire à SELinux et utilise des classes de sujets et d'objets. Les performances obtenues sont bonnes, particulièrement pour les appels systèmes qui ne perdent que 1% de performance. Cependant, seul un sous ensemble d'appels systèmes est contrôlé. De plus, il faut implanter des contextes de sécurité directement dans chaque programme ce qui est une contrainte difficile à mettre en œuvre.

D'un point de vue des performances, de nombreux travaux évaluent l'impact de la sécurité sur un cluster dédié à des calculs hautes performances. Fox *et al.* [32] étudient par exemple de manière pratique l'impact de grsecurity et de SELinux sur les performances du système. L'étude montre que grsecurity est plus léger que SELinux, notamment en terme d'accès mémoire (pas d'*overhead* pour la création de processus et les opérations arithmétiques). Néanmoins, la comparaison n'est pas donnée avec un système sans mécanisme de sécurité. De plus, les systèmes récents étant dotés de multi-cœurs, les *overheads* ont tendance à se réduire [76]. Pour les travaux de Leangsuksun *et al.* [46], les performances sont comparables à SELinux. Cependant, l'utilisation de DSI peut induire un overhead de plus de 20% pour le contrôle entre les nœuds du cluster, puisqu'il faut filtrer chaque paquet qui circule entre les nœuds.

Lorsque les techniques de virtualisation ont été introduites dans les systèmes à hautes performances, après 2005, il a fallu utiliser des hyperviseurs peu gourmands en ressources pour que cela reste acceptable. Par exemple, Youseff *et al.* [77] évaluent les performances d'un système utilisant XEN et traitant des jobs MPI sur un cluster de 4 nœuds, chacun muni de 4 processeurs. L'impact sur les performances est très réduit mais d'un point de vue sécurité, l'apport n'est pas très modulable contrairement à des technologies purement logicielles comme SELinux. L'essor de l'informatique en nuage découle de ces remarques : la virtualisation n'est plus un frein d'un point de vue des performances mais la sécurité n'est pas spécialement renforcée dans de tels systèmes.

La virtualisation légère a permis elle aussi de proposer des solutions peu coûteuses d'un point de vue performances [69]. Les utilisateurs sont isolés dans des "conteneurs" et l'isolation permet de cloisonner les ressources (fichiers) et aussi les objets du systèmes (mémoire, CPU). Suivant la virtualisation utilisée, on peut avoir un impact de 0% à 30% pour les accès disque et aucune différence d'un point de vue puissance de calcul. Cependant, chaque conteneur doit disposer des programmes et des données et aucun partage n'est possible.

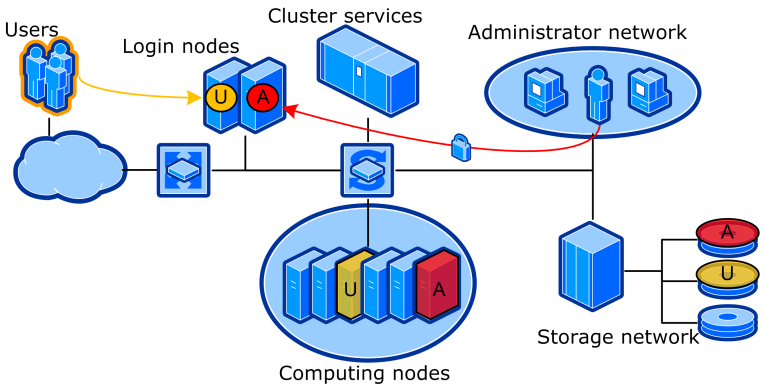


FIGURE 2.5 – Différents accès suivant les utilisateur projetés dans différents conteneurs

3.1.2 La création et l'installation de politiques

Créer et installer une politique de sécurité adéquate pour un cluster n'est pas une tâche triviale. Comme rapporté par Leangsuksun *et al.* [46], les composants de sécurité sont souvent des patchs noyaux qu'il faut appliquer successivement ce qui peut s'avérer complexe. Une fois ces composants mis en place, il faut créer la politique et la compiler [64]. Souvent, on utilise des techniques d'apprentissage pour éviter ce problème. Les outils *gradm* and *audit2allow* sont par exemple utilisés pour grsecurity et SELinux pour ce processus [32]. Cependant, les politiques résultantes peuvent contenir des règles non souhaitées ou conflictuelles avec des règles écrites manuellement par l'administrateur. Si l'on procède ainsi, il faut alors chercher ces inconsistances, par exemple, vérifier qu'un TCB (*Trusted Computing Base*) n'est pas violé est une tâche complexe [42].

Enfin, l'introduction d'une erreur aléatoire dans une politique peut avoir de graves conséquences. Ihde *et al.* [52] montrent qu'une erreur d'un bit dans un fichier binaire de politique provoque un rejet de chargement de la politique dans 33% des cas. Il reste donc 67% de cas où la politique erronée est chargée dans le système. Les auteurs identifient notamment 8 cas précis où l'erreur de bit provoque des vulnérabilités permettant de violer l'intégrité ou la confidentialité de données. Bien entendu, réaliser une telle attaque sur le binaire de la politique est hautement hypothétique du point de vue d'un attaquant extérieur.

3.1.3 Besoins en sécurité pour un cluster multi-utilisateur

Pour sécuriser le cluster et confiner les utilisateurs, nous présentons informellement l'expression de ce besoin. Une description plus précise peut être trouvée dans [12]. Les propriétés attendues dans notre système sont de :

- Garantir la confidentialité des données déposées par les utilisateurs ;
- Confiner les profils utilisateurs et les services de manière à ne pas compromettre la sécurité du système d'exploitation même en cas d'escalade de privilèges ;
- Différencier les accès distants publics (pour les utilisateurs) et privilégiés (pour les administrateurs) ;
- Sécuriser et contrôler les interactions avec les gestionnaires de tâches et les systèmes d'authentification.

La confidentialité des données et le confinement des utilisateurs peut se formaliser à l'aide de la notion de conteneur [12]. Le conteneur emprisonne l'utilisateur, même si celui-ci escalade ses privilèges. Le conteneur doit empêcher les interactions avec les systèmes de configuration, le système d'exploitation et restreindre les interactions aux seuls processus et données de l'utilisateur.

D'un point de vue des accès, il faut garantir la traçabilité de l'accès au système au travers du réseau. Comme présenté en figure 2.5, il faut que les accès soient différenciés, surtout quand l'utilisateur parvient sur un nœud du cluster : un accès normal utilisateur ne doit pas pouvoir configurer ce nœud (représenté en jaune) alors que l'administrateur le peut (représenté en rouge). Là encore, la notion de conteneur servira à implémenter ce besoin.

Enfin, un soin particulier doit être apporté à des processus spéciaux, comme le gestionnaire de tâches auquel on soumet des jobs de calcul. En effet, ce processus permet d'envoyer des tâches sur tous les nœuds ce qui fait de lui un vecteur d'attaque de choix.

```
interface (‘ cea_unprivileged_user ’, ‘
        userdom_unpriv_login_user($2)
        userdom_basic_networking_template($2)
        kernel_read_network_state($1)
        corenet_udp_bind_all_nodes($1)
        corenet_udp_bind_generic_port($1)
        corenet_tcp_bind_all_nodes($1)
        corenet_tcp_bind_generic_port($1)
        cea_userdom_login_user($2)
’)
```

Listing 2.1 – Extrait de la politique `ccc_guest.if`

```
policy_module(ccc_guest, 1.0.1)
cea_unprivileged_user(ccc_guest_t, ccc_guest);
cea_unprivileged_user(ccc_xguest_t, ccc_xguest);
userdom_restricted_xwindows_user_template(ccc_xguest);
gen_user(ccc_guest_u, user, ccc_guest_r ccc_xguest_r, s0, s0 – mls_systemhigh, mcs_allcats)
```

Listing 2.2 – Extrait de la politique `ccc_guest.te`

3.1.4 Création de la politique

Nous avons choisi de bâtir une politique SELinux, étant donné le support apporté à cet outil, la flexibilité du langage et les avantages présentés précédemment. Notre politique sera un module SELinux, ce qui permet ensuite de corriger la politique si des utilisateurs rencontrent des difficultés avec la politique déployée. En effet, les logs générés permettent de déterminer les sujets qui ont tenté d'interagir avec les objets et la règle qui a interdit cet accès, même si parfois les logs sont difficiles à comprendre, voire muets [74].

La conception de la politique est donc pragmatique, et contraire aux recommandations des auteurs de SELinux. Après avoir chargé une politique par défaut et exécuté des tâches utilisateur dans des scénarios standards, nous examinons les logs de */var/log/audit/audit.log*. Ces logs sont convertis en règles qui sont manuellement inspectées et intégrées à la politique initiale. A partir de ce procédé, deux modules de politiques SELinux sont extraits et des règles manuelles sont ajoutées pour obtenir les propriétés de sécurité voulues :

- **ccc_guest** qui implémente le confinement des utilisateurs : les "*guest*" par l'accès SSH et "*xguest*" pour les accès graphiques ;
- **sshd_admin** qui implémente deux niveaux d'accès SSH, l'un publique et l'autre administratif. Ainsi, deux contextes de sécurité seront associés à deux serveurs SSH distincts.

ccc_guest Par défaut, avec une politique SELinux classique, les utilisateurs sont placés dans le contexte de sécurité *(user_u, system_r, unconfined_t)*. La première étape consiste donc à dupliquer ce contexte pour confiner les utilisateurs dans *ccc_guest* ou *ccc_xguest* suivant le type de connexion (SSH ou X). Ce confinement limite au minimum les privilèges accordés aux utilisateurs. Notamment, un utilisateur connecté en SSH obtient le contexte *(ccc_guest_u, ccc_guest_r, ccc_guest_t)*. Si cet utilisateur devient *root* grâce à une vulnérabilité du système, il ne pourra cependant pas interagir avec d'autres contextes de sécurité autres que *(ccc_guest_u, ccc_guest_r, ccc_guest_t)*.

Les listings 2.1 et 2.2 montrent des extraits de la politique *ccc_guest*. La macro *cea_unprivileged_user* du listing 2.1 est appelée deux fois dans le listing 2.2 pour les deux types d'utilisateurs (SSH et X). Seuls les utilisateurs ayant une interface graphique possèdent des règles supplémentaires gérées par une autre macro.

sshd_admin Dans la politique classique SELinux, un serveur SSH est exécuté dans le contexte de sécurité particulier *sshd_t*. En tant que service exposé sur l'extérieur, nous proposons d'introduire dans le module *sshd_admin* son confinement associé à deux rôles : le rôle de l'utilisateur et celui d'administrateur. Ainsi, il devient possible de dupliquer le processus et de les isoler dans deux contextes différents associés aux types *sshd_public_t* et *sshd_admin_t*.

Il faut par la suite connecter ces nouveaux contextes de sécurité aux contextes précédents *ccc_guest_t*. En effet, une fois connecté à un des serveurs SSH, l'utilisateur doit pouvoir *transiter*, c'est-à-dire, d'un point de vue du système d'exploitation, *forker* un processus associé à *ccc_guest_t*. Pour les administrateurs, la transition

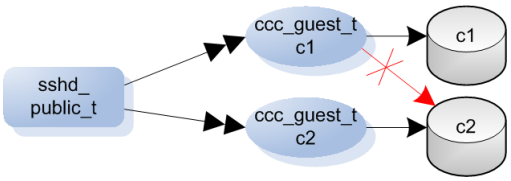


FIGURE 2.6 – Contrôle d'accès sur les ressources à l'aide de catégories

TABLE 2.3 – Exemple de clusters de production au CEA

FEATURE	COMPUTING CLUSTER	VISUALIZATION. CLUSTER	DEFENSE CLUSTER
Total nodes	1140	50	4300
CPU/node	2 Nehalem Quad	2 Xeon 5450	4 Xeon 7500
RAM/node	24 GB	64/128 GB	128 GB
Network FS	500 TB	100 TB	20 PB
User accounts		≈ 4000	...
Avg. running jobs	100-150	10	...
Avg. queued jobs	100-1000	N/A	...

se fera de *sshd_admin_t* vers *unconfined_t* où l'on retrouve la politique classique SELinux d'un administrateur permettant un accès aux fonctions d'administration du système.

L'isolation des utilisateurs est implémentée en utilisant les catégories MCS. De manière simple, il suffit d'associer une catégorie par utilisateur et pour leurs ressources. Pour lire un fichier f, un utilisateur U devra avoir au moins toutes les catégories associées à f :

$$U \text{peut accéder à } f \Leftrightarrow Cat(f) \subset Cat(U)$$

Disposant de 1024 catégories $c0$ à $c1023$, nous proposons de scinder l'ensemble des utilisateurs en groupes d'utilisateurs, par exemple par groupes de partenaires de recherche, par entreprise ou université, etc. L'isolation n'est pas optimale, mais la politique DAC permet tout de même à chaque utilisateur d'une même catégorie de protéger ses fichiers. En combinant plusieurs catégories ensemble, il est possible de créer des nouveaux groupes d'utilisateurs, par exemple le groupe associé à c_1, c_{1000}. En fait, le nombre de partitions étant de 2^{1024}, les possibilités sont largement suffisantes : la difficulté réside dans la projection des groupes sur ces ensembles. Lorsqu'un utilisateur se connecte au cluster, il reçoit un ensemble de catégories. Comme montré dans la figure 2.6, deux utilisateurs initialement dans le contexte *sshd_public_t* et projetés dans le type *ccc_guest_t* obtiennent des catégories différentes et ne peuvent alors interférer.

3.1.5 Évaluation des scénarios d'attaque

Deux types d'attaques sont pris en compte par la politique précédente.

Un utilisateur malveillant pourrait tenter d'exploiter une vulnérabilité sur un service du système d'exploitation ou le serveur SSH. Ces services ayant un contexte de sécurité précis, l'*exploit* ne permettrait pas de corrompre le système ou de lire des données d'autres utilisateurs. De plus, ces services n'ayant pas de catégorie, les fichiers utilisateurs ne pourraient être lus. Enfin, corrompre un service ne permet pas de changer de contexte de sécurité et donc de poursuivre cette attaque.

D'autre part, une attaque qui ciblerait un processus utilisateur pourrait gagner le contexte de sécurité de cet utilisateur. Le processus corrompu pourrait accéder aux données de cet utilisateur. Même *root*, ce processus n'obtiendrait pas plus de droits que la catégorie de cet utilisateur. L'attaque resterait donc très limitée.

3.1.6 Impact sur les performances

La politique proposée a été déployée sur deux types de cluster, comme précisé en table 2.3. Le premier cluster (Computing cluster) est un cluster scientifique de 1 068 nœuds, 48 nœuds GPU et 24 nœuds de stockage. Les performances sont de 100 Tflops pour les CPU et de 200 Tflops pour les GPU. Le second cluster (Visualization cluster) est dédié à la réduction de données et la visualisation. Il est composé de 38 nœuds, 2 nœuds pour l'accès et 10 nœuds de stockage. A titre de comparaison, nous donnons quelques caractéristiques d'un cluster de calcul

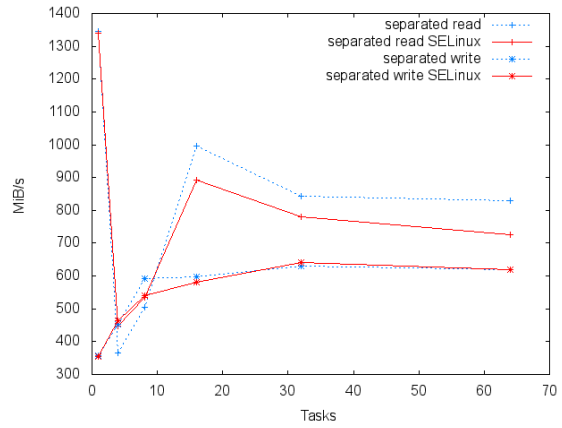


FIGURE 2.7 – Impact sur les entrées/sorties MPI

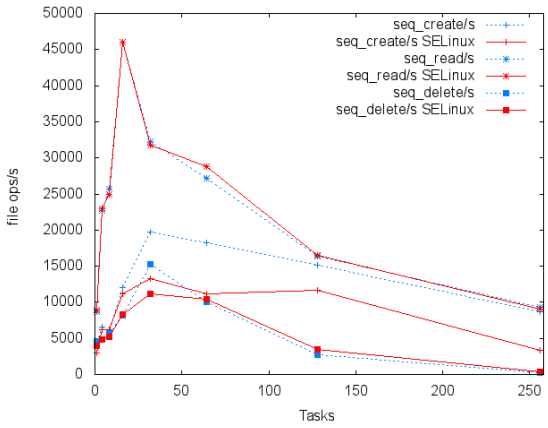


FIGURE 2.8 – Impact sur les méta-données (bonnie++)

militaire (Defense cluster), ici "Tera-100" : composé de 4 300 nœuds, il comporte 140 000 cœurs Intel Xeon 7500, 300 TB de mémoire, et 20 PB d'espace disque.

Nous avons aussi quantifié le nombre moyen de tâches en attente sur ce cluster ($\sim$ 100). Chaque tâche nécessite 10 à 100 nœuds du cluster et prend plusieurs heures pour s'exécuter ce qui explique le nombre de tâches en attente, malgré la taille du cluster. Exceptionnellement certaines tâches peuvent occuper l'intégralité du cluster.

Les figures 2.7 et 2.8 présentent respectivement l'impact sur les opérations d'entrée/sortie MPI et sur l'accès au méta-données des fichiers. On remarque que l'impact sur les opérations d'entrée sortie reste faible, comme attendu. Cependant, SELinux impacte le temps de création des fichiers (environ 30%). Cette baisse de performances est due à l'utilisation de Lustre, un système de fichiers distribué, auquel SELinux doit envoyer des requêtes pour demander le nom du contexte de sécurité lors de la création d'un fichier. Sans Lustre, nous avons mesuré un *overhead* de 10% seulement.

3.2 Contrôle d'accès dans un système collaboratif de gestion de crise

Dans cette partie, nous étudions un deuxième besoin spécifique : implanter du contrôle d'accès dans un système collaboratif, distribué et de surcroît, dédié à la gestion de crise. Le but de ce système est d'être le support dans le cas de crises humanitaires comme par exemple des tremblements de terre, des attaques terroristes, etc. Dans ce cas particulier, les infrastructures peuvent être partiellement endommagées, ce qui impose de prévoir une architecture robuste et tolérante aux pannes. De plus, l'urgence de telles situations peut imposer d'adapter les politiques de sécurité. Par exemple, s'il s'agit d'un accident, on souhaitera sans doute compartimenter les informations entre les secours de manière différente que s'il s'agit d'une attaque terroriste.

3.2.1 La gestion de crise humanitaire

Peu de travaux traitent de ce problème très particulier. Underwood *et al.* [73] donnent un panorama des outils conçus par des gouvernements nationaux afin de faciliciter la gestion de crise. Pour réussir, il faut un fort investissement politique car de tels outils n'ont que peu d'intérêt dans la vie quotidienne. Les efforts se concentrent sur la modélisation correcte et la gestion des acteurs d'une crise [21] et sur les possibilités de simulation de ces outils. Par exemple, le système DEFACTO [62] est un environnement de simulation qui permet d'améliorer la coordination des équipes sur le terrain. Un panomara des outils plus étendu a été écrit par Liu *et al.* [48].

En 2011, Bessis *et al.* [9] proposent une vision plus moderne de la gestion des crises avec l'arrivée des services web, du web 2.0 et des réseaux sociaux. Avec la percée des téléphones mobiles et l'arrivée de l'informatique embarquée dans de petits objets, il devient possible d'imaginer des outils de terrain connectés à de larges systèmes pour gérer l'intelligence de la crise. L'aspect collaboratif et participatif est primordial, ce qui sera illustré plus tard lorsque Google déploiera la solution *Google Crisis Response* par exemple. L'aspect décisionnel et contextuel reste limité dans la littérature. Bessis *et al.* proposent un scénario de gestion d'un tremblement de terre [8] afin d'aider les autorités locales à prendre des décisions pour les opérations de sauvetage. L'aspect géolocalisation

est lui mieux traité comme par Cai *et al.* [21] qui montrent comment mieux collaborer avec des informations géographiques relatives à la crise. L'outil proposé s'appuie sur des composants logiciels de type client-serveur et propose une visualisation avancée de la crise.

Aucun des travaux mentionnés ne traite spécifiquement d'implantation de politiques de sécurité pour contrôler les informations relatives à la crise. Cependant, certains travaux s'intéressent au déploiement de politiques dans des systèmes collaboratifs.

3.2.2 Les politiques de contrôle d'accès et les systèmes collaboratifs

Après l'introduction de RBAC [60], des variantes ont été proposées qui permettent de mieux traiter certains aspects collaboratifs. Thomas *et al.* introduisent TMAC (*Team Base Access Control*) en 1997 [71] et TBAC (*Task Base Access Control*) en 1998 [72]. Ces politiques traitent directement les insuffisances des rôles quand une équipe ou une tâche particulière doit être réalisée. La notion d'équipe permet de faire varier les règles de contrôle d'accès tandis que la notion de tâche permet de donner des nouveaux droits quand une tâche se termine.

En 2003, les politiques organisationnelles (OrBac) font leur apparition [27], répondant à une problématique de collaboration au sein des organisations. Chaque organisation possède sa politique et la notion de contexte permet de faire évoluer les politiques dynamiquement en fonction des changements de contexte. La notion de contexte est centrale pour un système de gestion de crise, car elle permet d'apporter la flexibilité nécessaire en fonction du type de crise. Dès 2001, des politiques contextuelles sont proposées. Par exemple, Convington *et al.* [24] proposent une politique de type RBAC dépendante du contexte. Différents rôles sont activés en fonction des conditions de l'environnement (participants, ressources, tâches) ou du moment où les requêtes sont honorées.

Par la suite, les notions de confiance et de réputation font leur apparition dans les politiques [49, 35, 31, 22]. Ces notions qui proviennent des systèmes collaboratifs, et notamment des réseaux pair à pair. Chakraborty *et al.* [22] proposent par exemple d'ajouter aux rôles des niveaux de confiance et de définir des règles de contrôle d'accès prenant en compte ces niveaux de confiance.

La flexibilité dans la définition des politiques augmentent encore avec l'introduction de politiques basées attributs ABAC (*Attribute Based Access Control*) [79]. Ces politiques permettent de donner des règles qui dépendent de certains attributs des sujets ou des objets de la politique. La notion de rôle s'efface pour ces types de politiques. Seuls les sujets agissent sur les ressources, ceci en fonction de règles dépendantes des attributs. Il faut aussi prévoir que les attributs peuvent ne pas exister et que des décisions peuvent être prises quand même. Combinées à la notion de confiance et de réputation [67], ces politiques sont plus adéquates pour les systèmes collaboratifs hautement dynamiques, notamment quand ils comportent des utilisateurs auxquels on ne peut accorder beaucoup de confiance.

Pour ces dernières raisons, ce sont ces types de politiques que nous utilisons par la suite. Elles permettent une grande flexibilité vis-à-vis du contexte de la crise et de la gestion des utilisateurs.

3.2.3 Conception d'une politique de gestion de crise

Pour concevoir notre politique, nous avons créé un formalisme *ad-hoc*, qui s'appuie sur la famille de politiques basées attributs ABAC (*Attribute Based Access Control*). En effet, il n'est pas très utile de se rapprocher d'un langage d'un système cible comme SELinux car il est peu probable que les systèmes utilisés dans la crise propose un tel mécanisme de sécurité. Cependant, si tel était le cas, il est toujours possible de projeter une politique vers SELinux, comme nous l'avons montré pour les politiques d'équipe TMAC (*Team Based Access Control*) [18]. Si l'implantation de la politique se fait au niveau du logiciel lui-même, comme expliqué brièvement lors de la présentation du démonstrateur en section 3.2.4, on peut plutôt craindre des attaques contre le logiciel. Dans ce cas, on préfère implanter la politique au niveau d'une machine virtuelle Java, par exemple, comme nous l'avons montré dans [17]. Par la suite, nous donnons les principaux éléments qui permettent de construire les règles de la politique. Plus de détails sont disponibles dans [66].

La conception de la politique répond à quatre principaux objectifs :

- définir des règles de contrôle d'accès classiques entre des sujets et des objets ;
- introduire la notion de confiance dans les prises de décision, en fonction de la confiance que l'on accorde aux sujets ;
- introduire la notion d'objectifs dans les règles de contrôle d'accès.

Une règle simple de contrôle d'accès entre un sujet s et un objet o est exprimée sous la forme :

$$pa_i(s.ATTR, o.ATTR, CTXT, OP) \in PERM$$

où ATTR est un ensemble d'attributs, CTXT un ensemble de contextes, OP un ensemble d'opérations (*read*, *write*, etc.), et PERM une décision *allow*, *deny* ou *NA* (*Not Applicable*). La gestion des possibles conflits entre les règles pa_i peut être résolu en évaluant les règles dans l'ordre d'écriture de la politique, ou bien en cherchant à détecter ces conflits par exemple en choisissant toujours *deny* si une des règles en conflit retourne *deny* [54]. Pour représenter pa_i, on peut par exemple exprimer qu'un militaire peut consulter les coordonnées et le type d'un médecin :

```
pa_1(s.ATTR, o.ATTR, CTXT, OP):
if (s.ATTR['type'] == Military && (o.ATTR['type'] == MedicalDoctor || o.ATTR['type'] == Firemen)
        && o.ATTR == {location, type}
        && OP == {read})
        return allow
else
        return deny
```

On voit ici que cette règle d'exemple de la politique est donnée sous forme d'un programme mais il aurait été possible de l'écrire sous la forme d'une assertion. Un exemple plus complexe permet de comprendre comment faire intervenir la notion de contexte de crise. Dans l'exemple ci-dessous, on autorise tous les acteurs de la crise à consulter les types d'acteurs et les localisations en cas de crise terroriste, et sinon on applique la politique "normale" :

```
pa_4(s.ATTR, o.ATTR, CTXT, OP):
if (! o.ATTR.isSubSet({type, location })
        return deny
if (CTXT == {terrorism })
        return allow
if (s.ATTR['type'] == Military && o.ATTR['type'] == MedicalDoctor && s.ATTR['role'] == commander || s.ATTR['role'] == general || s.ATTR['
    credential'] == "authorized")
        return allow
else
        return deny
```

L'incorporation de la confiance est réalisée par la comparaison du niveau de confiance *Ptr(s)* que l'on possède dans le sujet *s* vis à vis du niveau requis pour la règle :

$$pa_i(s.ATTR, o.ATTR, CTXT, OP) \in PERM \wedge$$
$$Ptr(s) \geq req\_trust(s.ATTR, o.ATTR, CTXT, OP)$$

Par exemple, la confiance d'un militaire accédant à la localisation d'un médecin pourrait être de 0.5 en lecture et de 1 en écriture. Par contre, le nom d'un médecin devrait être librement accessible par un utilisateur pour lequel nous n'aurions pas de confiance particulière, i.e. une confiance de 0. La fonction ci-dessous *req_trust* propose l'implémentation de cet exemple :

```
req_trust (s.ATTR, o.ATTR, CTXT, OP):
if (o.ATTR.contains({type, location }) && o.ATTR['type'] == MedicalDoctor)
if (s.ATTR['type'] == Military && OP == {read})
        return 0.5
if (s.ATTR['type'] == Military && OP == {write})
        return 1
if (o.ATTR.contains({type,name}) && o.ATTR['type'] == MedicalDoctor)
        return 0
return 1
```

L'évaluation de la confiance du sujet s n'est pas traitée en détail ici. On peut s'inspirer de solutions qui établissent un système de réputation [34] ou bien décider que chaque utilisateur renseigne pour lui même les niveaux de confiance qu'il accorde à des types d'utilisateurs. Dans ce cas, on s'éloigne clairement d'une politique de contrôle d'accès obligatoire.

Enfin, l'ajout des objectifs collaboratifs est assuré en ajoutant une clause d'inclusion sur les buts poursuivis par les sujets et dont les objets sont prévus pour y participer :

$$pa_i(s.ATTR, o.ATTR, CTXT, OP) \in PERM \wedge$$
$$Ptr(s) \geq req\_trust(s.ATTR, o.ATTR, CTXT, OP) \wedge$$
$$purp\_attr(s.ATTR) \subset purp\_attr(o.ATTR)$$

A titre d'exemple, si l'ensemble des buts est $P = \{p_1 = sauver\ des\ gens, p_2 = coordonner\ le\ sauvetage\ des\ gens, p_3 = coordoner\ la\ crise, p_4 = informer\ les\ mdias\}$ et que l'on définit les buts suivants au travers de la fonction *purp_attr()* :

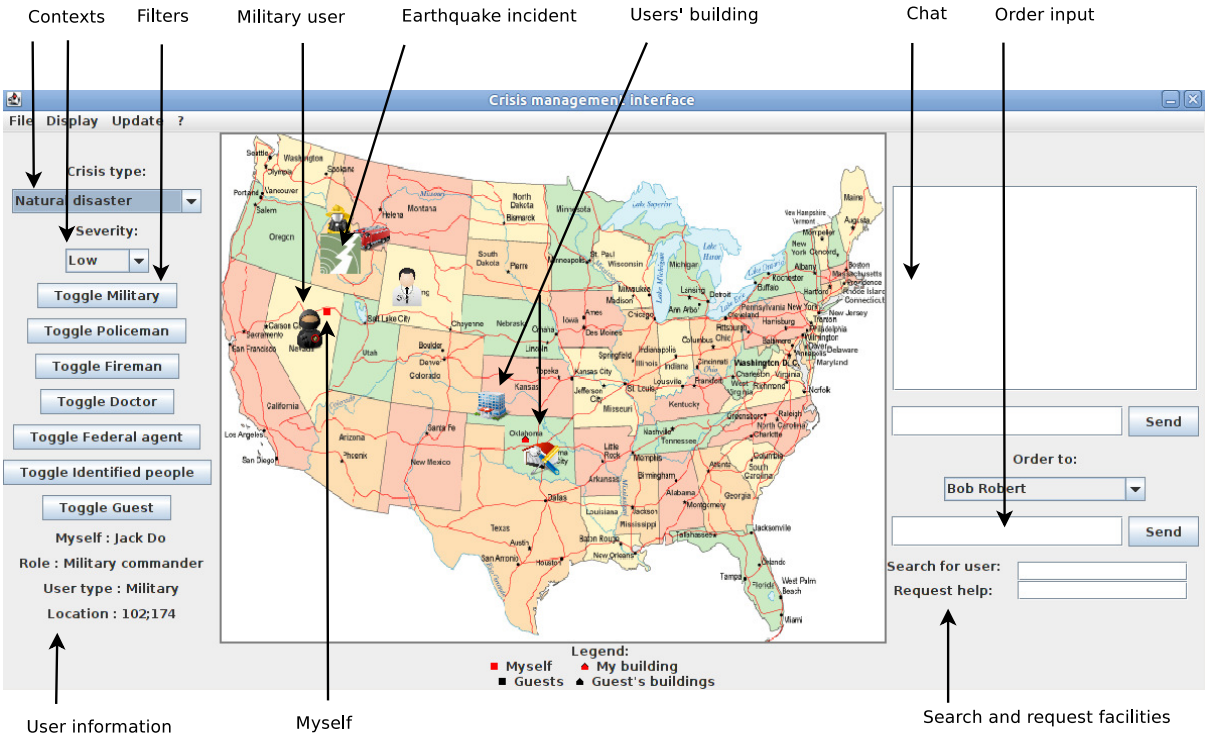


FIGURE 2.9 – Interface du logiciel de gestion de crise

```
purp_attr(s.ATTR):
if (s.ATTR['type'] == Military)
        return {p2}
if (s.ATTR['type'] == MedicalDoctor)
        return {p2, p3}
return {NA}
```

alors, la requête d'accès d'un militaire à une localisation d'un médecin peut être honorée parce que le but d'un militaire est inclus dans l'ensemble des buts d'un médecin.

3.2.4 Implémentation d'un démonstrateur

Un démonstrateur a été réalisé pour illustrer notre approche. Pour simplifier les développements, le code implémente un serveur et de multiples clients, même si le logiciel pourrait être implémenté sous forme d'un logiciel pair-à-pair ce qui nécessiterait beaucoup plus de développement. La figure 2.9 présente l'interface du démonstrateur qui a été implémenté pour un scénario de tremblement de terre impliquant des policiers, pompiers, médecins et utilisateurs anonymes. On trouve sur la gauche de l'interface, le contexte, qui possède deux facteurs : le type de crise et sa gravité. Un certain nombre de filtres sont disponibles ainsi que l'identité de l'utilisateur connecté. Au centre, la carte représente la crise c'est-à-dire positionne les sujets et les objets avec leurs coordonnées. A droite, une partie messagerie permet de communiquer.

Suivant le sujet qui est connecté, la politique contrôle principalement la lecture des attributs, notamment la géolocalisation, en fonction des attributs sujets, et du contexte de crise. Les figures 2.10 et 2.11 montrent que pour un sujet de type "anonyme", l'application de la politique de sécurité supprime la lecture de nombreux attributs et supprime la visibilité de tous les sujets et objets sauf la position de l'épicentre du tremblement de terre. Dans le cas où un militaire bascule le contexte de sévérité au niveau "*Highest*" à la place de "*Low*", alors un utilisateur anonyme retrouve la visibilité sur tous les sujets et objets du système. Cette flexibilité de contexte permet de relâcher la politique qui serait trop sévère si l'évènement est très grave et que les informations doivent circuler au plus vite.

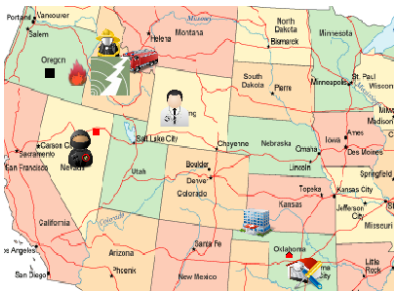


FIGURE 2.10 – La vue d'un militaire dans le contexte de sévérité "Low"

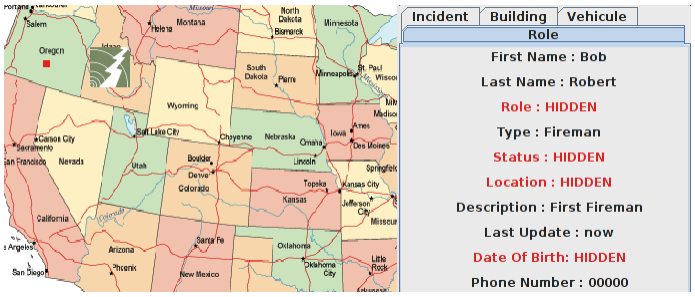


FIGURE 2.11 – La vue d'une personne anonyme dans le contexte de sévérité "Low"

3.3 Administration et contrôle des politiques

Les deux contributions précédentes supposent que les politiques sont écrites statiquement à l'avance et n'évoluent pas au cours du temps. Avec l'aide du contexte de la situation, on donne plus de dynamicité aux règles de la politique, mais si le contexte souhaité n'est pas prévu à l'avance, il n'y a aucun moyen de modifier la politique. Ainsi, nous nous sommes intéressés à la problématique de l'évolution des politiques, par exemple lorsqu'un administrateur système souhaite ajouter des règles nouvelles dans une politique de contrôle d'accès obligatoire pour un cluster de calcul. Le problème majeur réside dans le fait qu'il n'y a pas de contrôle sur ces changements, puisque l'hypothèse est faite que l'administrateur n'est pas malveillant. Si l'on rajoute cette hypothèse, il faut alors introduire une vérification de la modification de la politique P, et un contrôle de cette nouvelle politique avant son application. On appelle alors méta-politique MP la politique d'administration de la politique P. Dans cette partie, nous décrivons synthétiquement comment mettre en œuvre une telle méta-politique, ses bénéfices et ce que l'on peut vérifier formellement sur celle-ci [19].

3.3.1 Les politiques dynamiques

La plupart des travaux qui s'intéressent aux politiques dynamiques définissent un super rôle autorisé à modifier les politiques [61, 63] alors que les administrateurs du système réalisent les tâches courantes d'administration, sous contrainte de la politique. Les premiers travaux menés par Sandhu *et al.* sur l'administration des politiques RBAC [61] formalisent comment contrôler les évolutions de politique à partir de rôles spécifiques d'administration. Par exemple, la règle *can-assign(x,y,Z)* permet au rôle d'administration *x* qui répond à la condition *y*, d'ajouter des permissions aux rôles de *Z*. Crampton *et al.* raffinent le procédé en proposant un arbre de rôles RBAC où chaque rôle parent peut administrer les rôles enfants [26, 25]. Par la suite, Damiani *et al.* [28], s'inspirant de cette hiérarchie, définissent un arbre hiérarchique de domaines. Les domaines correspondent à des secteurs organisationnels spatialement situés (une sorte d'extension de rôle ou de but). Chaque nœud de l'arbre correspond à un domaine de la politique et un sous-nœud est un sous-domaine du domaine parent. Chaque administrateur d'un domaine peut créer des sous-domaines et créer ou détruire des règles de politiques dans ce sous-domaine.

En 2007, Li *et al.* [47] remettent à plat les modèles existants d'administration des politiques en définissant les propriétés souhaitables pour que l'administration soit aisée. Premièrement, la gestion des politiques doit être adaptée à de larges systèmes et doit être plus flexible que l'administration par hiérarchie de rôle. Un des arguments est que des droits peuvent être donnés depuis un rôle organisationnel vers un rôle technique, n'étant pas hiérarchiquement parents. Deuxièmement, l'administration doit être réversible, c'est-à-dire qu'une opération d'administration de la politique doit pouvoir être défaite et retrouver son état antérieur, ce qui n'est pas le cas avec la hiérarchie de rôle de Sandhu *et al.* Enfin, Li *et al.* rappellent que l'administration des politiques doit être simple. Ainsi plutôt que d'introduire un formalisme particulier d'administration, une politique RBAC doit être administrée par une politique RBAC. Une règle RBAC d'administration de la politique stipule donc qu'un ajout de permission nécessite une méta-permission exprimant que l'utilisateur est administrateur et qu'il est dans le rôle plus puissant de modification.

C'est dans un esprit similaire que nous avons conçu une méta-politique de contrôle d'accès [19]. Cependant, notre but n'était pas de revisiter un modèle plus efficace d'administration de politiques mais de fournir un langage *simplifié et contraint* à des fins de calcul de vérification. En effet, dans la littérature précédente, si les règles de la

```
# Default policy
enableIV( login_d, admin_d, transition )
enableIV( login_d, user_d, transition )
enableIV( ssh_d, user_d, transition )
enableIV( user_d, webserv_d, transition )
enableIV( admin_d, webserv_d, transition )
enableIV( admin_d, apache_d, transition )
enableIV( apache_d, webserv_d, transition )
enableIV( admin_d, apache_conf_t, { read, write } )
enableIV( apache_d, apache_conf_t, { read } )
enableIV( apache_d, var_www_t, { read, write, execute } )
enableIV( webserv_d, .* _info_t , { read } )
enableIV( user_d, user_info_t , { read, write } )

# Meta-policy for php installation
enableAddSC(admin_d, php.*)
enableAddIV(admin_d, (php.*, php.*, {.*}) )
enableAddIV(admin_d, (webserv.*, php.*, {.*}) )
enableAddIV(admin_d, (php.*, apache_conf.*, {r,w}))
enableAddIV(admin_d, (php.*, var_www.*, {r,w,e}))
```

Listing 2.3 – Exemple de politique et de méta-politique

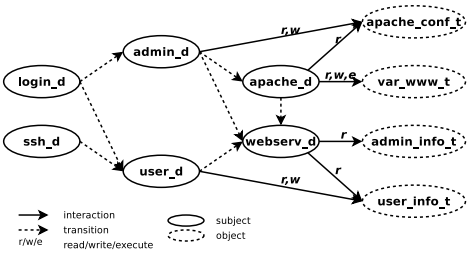


FIGURE 2.12 – Graphe d'interactions

méta-politique d'administration sont respectées, l'administrateur est libre de créer des rôles, des permissions et de créer des associations. Cela ne contraint notamment pas les chaînes de caractères de ces nouveaux rôles ou permissions. En revanche, la portée de la politique (le domaine d'application) est souvent contrôlée. En laissant autant de liberté à l'administrateur, on ne peut pas vérifier de méta-propriétés sur les politiques qu'il est possible de créer. C'est ce problème qui est traité par la suite.

3.3.2 Conception d'une méta-politique

La conception d'une méta-politique, dans le sens précédemment évoqué n'est possible que si des *patterns* plus contraints que des textes libres sont utilisés pour les méta règles. Pour écrire une telle règle, on définit donc qu'un contexte de sécurité qui fait une requête de modification $sc_{requester}$ est autorisé à ajouter à la politique un triple $(sujet, objet, permissions)$ respectant un *pattern* prévu dans la méta-politique. Cela n'empêche pas de marier ce principe avec un modèle d'administration de type RBAC comme proposé par Li *et al.* [47]. Le contrôle de l'ajout, la modification ou la suppression de règles, s'écrit donc sous la forme ("IV" signifiant *Interation Vector*) :

enableAddIV($sc_{requester}$, ($pattern_S$, $pattern_O$,$pattern_{oper}$))
enableModIV($sc_{requester}$, ($pattern_S$, $pattern_O$,$pattern_{oper}$))
enableDelIV($sc_{requester}$, ($pattern_S$, $pattern_O$,$pattern_{oper}$))

On administre de la même façon les contextes de sécurité ("SC" signifiant *Security Context*) :

enableAddSC($sc_{requester}$, $pattern_{SC}$)
enableDelSC($sc_{requester}$, $pattern_{SC}$)

L'exemple du listing 2.3 présente une politique simple pour un serveur Linux hébergeant un serveur Apache. Les règles *enableIV* constituent la politique de base. Cette politique est représentée en figure 2.12 où les règles de transitions sont en pointillé et les permissions en traits pleins. Dans cette politique seuls les utilisateurs logués localement peuvent obtenir le contexte *admin_t*. Les autres règles du listing 2.3 constituent la méta-politique qui permet d'ajouter des règles relatives à la possibilité d'installer PHP en laissant la liberté de créer des contextes de sécurité commençant par le label "php".

Dans le listing 2.4, nous donnons un exemple de modification de la politique respectant la méta-politique : l'administrateur ajoute les contextes *php_d* et *php_exec_t* et crée des interactions avec la configuration apache, les fichiers du serveur web et le processus PHP. La figure 2.13 montre l'état de la nouvelle politique une fois les modifications effectuées. L'intérêt de telles règles de méta-politique pour PHP est que dans un système à large échelle, plusieurs administrateurs de politique peuvent créer des configurations de politiques différentes pour différents clusters. Éventuellement, dans un même cluster, plusieurs processus PHP peuvent exister, afin de cloisonner les exécutions. Il y a donc une certaine flexibilité à laisser l'administrateur local ajouter ou pas des variantes de ces règles pour PHP.

```
enableSC(php_d)
enableSC(php_exec_t)
enableIV(webserv_d,php_d, { transition })
enableIV(php_d,php_exec_t, {e}) )
enableIV(php_d,apache_conf_t, {r,w}))
enableIV(php_d,var_www_t, {r,e}) )
```

Listing 2.4 – Modifications de la politique

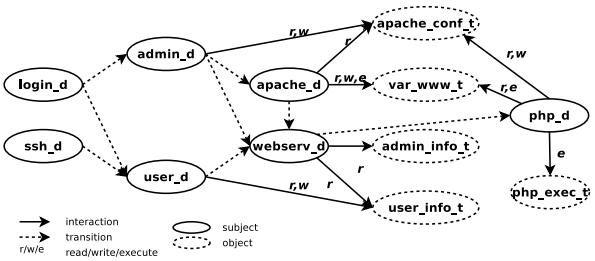


FIGURE 2.13 – Graphe d'interaction après modifications

TABLE 2.4 – Violations possibles sur une politique SELinux standard

		INITIAL	GATEWAY	USER
Graph	*SC*	595	577	3 017
	IV	18 215	17 684	314 582
Security	integrity	140	137	9 461
Property	confidentiality	29 510	29 510	726 842
Rules	duties_separation	270	243	16 405

3.3.3 Vérification de la méta-politique

L'intérêt d'avoir utilisé des *patterns* dans l'expression de la méta-politique est que l'on peut vérifier si des propriétés de sécurité peuvent être violées en modifiant la politique. Si l'on reprend la politique du listing 2.3, on peut par exemple vérifier si la configuration d'Apache est intègre et confidentielle vis-à-vis des utilisateurs qui se loguent à distance, c'est-à-dire par le contexte *ssh_d*. En effet, si l'on regarde sa représentation en figure 2.12, il n'y a aucun chemin possible entre *ssh_d* et *apache_conf_t*. On exprime ces deux propriétés dans un langage de propriétés [2, 19], non détaillé dans ce manuscrit, à l'aide des primitives suivantes :

```
integrity ( sc1:="ssh_d", sc2:="apache_conf_t" );
confidentiality ( sc1:="ssh_d", sc2:="apache_conf_t" );
```

La nouvelle politique modifiée par l'administrateur, représentée en figure 2.13 viole clairement ces propriétés. L'enjeu est donc de vérifier si les modifications autorisées par la méta-politique, comme celles du listing 2.4 permettraient de générer une politique violant les propriétés de sécurité que l'on souhaite assurer. Cette vérification reste assez simple à mettre en œuvre à condition que les *pattern* utilisés ne soient pas trop complexes. Il suffit, pour chaque règle de la méta-politique d'introduire des méta-nœuds correspondant à l'ensemble des nœuds que la règle peut générer et de lier ces méta-nœuds aux autres nœuds de la politique. On réalise ensuite un calcul d'atteignabilité sur ce graphe modifié pour vérifier qu'une propriété de sécurité ne peut être mise en danger.

3.3.4 Exemple d'évaluation d'une politique pour un pot de miel

Lors des expériences menées sur les pots de miel hautes interactions décrites en section 2.2, nous avons évalué les politiques standard proposées par SELinux suivant un ensemble de propriétés de sécurités très simples :

```
integrity (sc1 :=".*",  sc2  =".*:.*:.*  _exec_t");
confidentiality (sc1:=user_u:user_r:user_t, sc2:="system_u:object_r :.*"  );
duties_separation (sc1 :=".*") ;
```

La règle d'intégrité permet d'exprimer le fait qu'aucun contexte sujet ne doit pouvoir modifier un binaire. La règle de confidentialité exprime le fait qu'aucun utilisateur ne doit pouvoir lire un fichier système. La règle de séparation des privilèges dit qu'aucun contexte ne doit pouvoir modifier un fichier puis l'exécuter.

La table 2.4 présente en colonne *Initial*, l'ensemble des variations de chemins trouvées dans une politique standard SELinux permettant de violer les propriétés de sécurité. Le nombre de chemins possibles est relativement important concernant la confidentialité du système. Ainsi, pour un système réel comme la passerelle d'un pot de miel (*Gateway*), la politique comporte à peu de choses près le même nombre de violations. Pour le système accueillant l'attaquant (*User*), nous appliquons une méta-politique qui permet de faire évoluer la politique

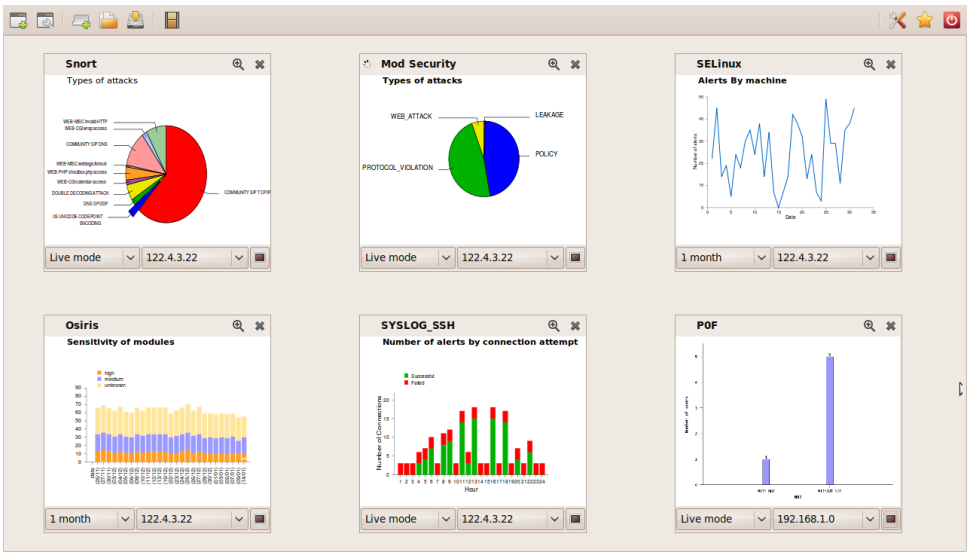


FIGURE 2.14 – Une vue de l'interface de Synema

initiale : des règles additionnelles permettent de faire fonctionner des logiciels comme Gnome desktop, Firefox, OpenOffice, etc. Avec une telle méta-politique, le nombre de violations possibles augmente très fortement.

4 Logiciels

Cet axe de mes recherches était lié à plusieurs développements logiciels au sein de l'équipe. Une grande partie de ces développements ont été réalisés par Jérémy Briffaut ou bien par des collaborateurs appartenant notamment au CEA. Ils se sont poursuivis intensivement au travers d'autres travaux menés par l'équipe [37, 15, 36, 4] et ne sont donc pas décrits dans le cadre de ce manuscrit.

Il faut noter un logiciel particulier développé dans le cadre de la surveillance de sonde de sécurité système et réseau : Synema [2]. Synema est un projet pédagogique que j'ai encadré avec Patrice Clemente pour les étudiants ingénieurs du cursus STI (Sécurité et Technologies de l'Information) de 1re année du cycle ingénieur. Il consiste à développer des outils d'analyse de logs de sonde de sécurité (Snort, Prelude, SELinux, p0f, etc.) en deux parties logiciels : une partie traite les données du côté de la machine produisant les logs afin de filtrer les informations et de préparer les données, une autre partie charge les données préparées et réalise différents graphiques et statistiques. Chaque sonde a été réalisée par un groupe d'étudiants et l'ensemble des sondes est intégré sous forme de plugins C dans une interface graphique qui fait donc office de démonstrateur de SIEM (*Security Information Management System*). En effet, l'interface doit pouvoir afficher plusieurs vues à la fois, pour une seule machine ou un ensemble de machines, et dans deux modes : un mode "live" pour suivre l'évolution temps réel des logs et un mode "historique" pour rejouer des logs antérieurs. Un exemple de vue de l'interface est montré en figure 2.14.

Le développement de Synema a fait l'objet d'un stage pour implémenter un plugin de corrélation entre les évènements réseaux et systèmes [14]. Ainsi, il permet de détecter une session d'attaque caractérisée par une intrusion réseau, suivie d'une tentative d'escalade de privilèges détectée au niveau système.

5 Bilan et perspectives de recherche

Ces travaux ont contribué à présenter et expliciter une démarche de sécurisation de systèmes à large échelle. Les aspects techniques sont importants car ils permettent aux solutions proposées d'être directement exploitables dans un contexte industriel. La conception de politique reste une affaire artisanale où l'on doit maitriser les familles de politiques disponibles dans la littérature, les mécanismes de contrôle d'accès existants ou ceux qu'il est raisonnable d'implémenter soi-même. Ces aspects sont d'ailleurs peu enseignés dans nos cursus d'ingénieurs ou universitaires. Ils sont peu mis en œuvre dans les PME et un effort certain de démocratisation doit être fait.

2. https://traclifo.univ-orleans.fr/SYNEMA/

Cependant, certaines avancées comme l'arrivée de langages comme le XACML permet de pousser la thématique de conception de politique. Les logiciels de manipulation de politiques doivent se développer pour rendre les choses plus accessibles. Pour ces raisons, je pense que l'effort doit porter sur les politiques de contrôle d'accès pour les logiciels et que le déploiement des politiques peut être fait à plusieurs niveaux : le réseau, le système d'exploitation, la machine virtuelle, les *middleware* de communication. Il faut pour cela créer des langages et des outils pour que les développeurs puissent exprimer des propriétés de sécurité sur les données manipulées et intégrer ces propriétés dans le code source des applications. J'ai débuté un travail dans ce sens pour les politiques RBAC pour les logiciels Java [17] et cet axe mériterait d'y travailler davantage.

D'un point de vue des attaques, il reste un effort de recherche considérable à fournir en s'appuyant sur des partenariats industriels. La capture d'attaques reste, de mon point de vue, un sujet totalement neuf et ouvert mais qui ne peut être traité sans des données de systèmes réels. L'anonymisation des données est une piste intéressante pour pouvoir obtenir la matière nécessaire à l'étude d'attaques. A l'inverse, la découverte de vulnérabilités est aussi une activité peu rémunératrice, d'un point de vue académique. Elle est cependant bien reconnue en France dans certaines manifestations comme SSTIC. Le prototypage d'attaques sur des grands systèmes type *cloud* ou réseaux sociaux est une activité qu'il faut intensifier. De mon point de vue, cela fait partie d'une démarche scientifique nécessaire lorsque l'on propose des contributions de recherche en sécurité.

6 Références

[2] J. Briffaut. **Formalisation et garantie de propriétés de sécurité système : application à la détection d'intrusions**. Thèse de doctorat. Université d'Orléans, décembre 2007 (cf. p. 5, 26).

[4] Z. Afoulki, A. Bousquet, J. Briffaut, J. Rouzaud-Cornabas et C. Toinard. **MAC protection of the OpenNebula Cloud environment**. Dans : *International Conference on High Performance Computing & Simulation*. Madrid, Spain : IEEE Computer Society, juillet 2012, p. 85–90. doi : 10.1109/HPCSim.2012.6266895 (cf. p. 27).

[5] E. Alata, V. Nicomette, M. Kaâniche, M. Dacier et M. Herrb. **Lessons Learned from the deployment of a high-interaction honeypot**. Dans : *6th European Dependable Computing Conference*. Coimbra, Portugal : IEEE Computer Society, octobre 2006, p. 39–46. doi : 10.1109/EDCC.2006.17 (cf. p. 13).

[6] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos et A. D. Keromytis. **Detecting Targeted Attacks Using Shadow Honeypots**. Dans : *14th USENIX Security Symposium*. T. 14. Baltimore, USA, août 2005, p. 129–144 (cf. p. 13).

[7] D. E. Bell et L. J. La Padula. **Secure Computer Systems : Mathematical Foundations and Model**. Technical Report M74-244. Bedford, MA : The MITRE Corporation, mai 1973 (cf. p. 11, 12).

[8] N. Bessis, E. Asimakopoulou, T. French, P. Norrington et F. Xhafa. **The Big Picture, from Grids and Clouds to Crowds : A Data Collective Computational Intelligence Case Proposal for Managing Disasters**. Dans : *2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. IEEE Computer Society, novembre 2010, p. 351–356. doi : 10.1109/3PGCIC.2010.58 (cf. p. 20).

[9] N. Bessis, E. Assimakopoulou, M. E. Aydin et F. Xhafa. **Utilizing Next Generation Emerging Technologies for Enabling Collective Computational Intelligence in Disaster Management**. Dans : Springer Berlin Heidelberg, 2011, p. 503–526. doi : 10.1007/978-3-642-20344-2_19 (cf. p. 20).

[10] K. J. Biba. **Integrity Considerations for Secure Computer Systems**. Rapport technique MTR-3153. The MITRE Corporation, juin 1975 (cf. p. 12).

[11] P. Biondi. **Security at Kernel Level : LIDS**. Dans : *Free and Open source Software Developers' European Meeting*. Février 2002 (cf. p. 12).

[12] M. Blanc et J.-F. Lalande. **Improving Mandatory Access Control for HPC clusters**. Dans : *Future Generation Computer Systems* 29.3 (avril 2013), p. 876–885. doi : 10.1016/j.future.2012.03.020 (cf. p. 6, 17).

[13] W. E. Boebert et R. Y. Kain. **A Practical Alternative to Hierarchical Integrity Policies**. Dans : *The 8th National Computer Security Conference*. Gaithersburg, MD, USA, octobre 1985, p. 18–27 (cf. p. 12).

[14] A. Bousquet, P. Clemente et J.-F. Lalande. **SYNEMA : visual monitoring of network and system security sensors**. Dans : *International Conference on Security and Cryptography*. Sous la dir. de J. Lopez et P. Samarati. Seville, Spain : SciTePress, juillet 2011, p. 375–378. doi : 10.5220/0003516203750378 (cf. p. 6, 15, 27).

[15] J. Briffaut, E. Lefebvre, J. Rouzaud-Cornabas et C. Toinard. **PIGA-Virt : An Advanced Distributed MAC Protection of Virtual Systems**. Dans : *6th Workshop on Virtualization in High-Performance Cloud Computing*. Bordeaux, France : Springer Berlin Heidelberg, août 2012, p. 416–425. doi : 10.1007/978-3-642-29740-3_47 (cf. p. 27).

[16] J. Briffaut, P. Clemente, J.-F. Lalande et J. Rouzaud-Cornabas. **Honeypot forensics for system and network SIEM design**. Dans : *Advances in Security Information Management : Perceptions and Outcomes*. Sous la dir. de G. S. de Tangil et E. Palomar. Computer Networks and Computer Science, Technology and Applications. Nova Science Publishers, 2013. Chap. 8, p. 181–216 (cf. p. 6, 14).

[17] J. Briffaut, X. Kauffmann-Tourkestansky, J.-F. Lalande et W. Smari. **Generation of role based access control security policies for Java collaborative applications**. Dans : *Third International Conference on Emerging Security Information, Systems and Technologies*. Athens/Glyfada, Greece : IEEE Computer Society, juin 2009, p. 224–229. doi : 10.1109/SECURWARE.2009.41 (cf. p. 6, 21, 28).

[18] J. Briffaut, J.-F. Lalande et W. Smari. **Team-based MAC policy over Security-Enhanced Linux**. Dans : *Second International Conference on Emerging Security Information, Systems and Technologies*. Cap Esterel France : IEEE Computer Society, août 2008, p. 41–46. doi : 10.1109/SECURWARE.2008.35 (cf. p. 6, 21).

[19] J. Briffaut, J.-F. Lalande et C. Toinard. **Formalization of security properties : enforcement for MAC operating systems and verification of dynamic MAC policies**. Dans : *International journal on advances in security* 2.4 (2009), p. 325–343 (cf. p. 6, 24, 26).

[20] J. Briffaut, J.-F. Lalande et C. Toinard. **Security and Results of a Large-Scale High-Interaction Honeypot**. Dans : *Journal of Computers* 4.5 (2009), p. 395–404. doi : 10.4304/jcp.4.5.395-404 (cf. p. 6, 13).

[21] G. Cai, A. M. MacEachren, R. Sharma, I. Brewer, S. Fuhrmann et M. McNeese. **Enabling GeoCollaborative crisis management through advanced geoinformation technologies**. Dans : *2005 national conference on Digital government research*. Atlanta, USA : Digital Government Society of North America, mai 2005, p. 227–228 (cf. p. 20, 21).

[22] S. Chakraborty et I. Ray. **TrustBAC - Integrating trust relationships into the RBAC model for access control in open systems**. Dans : *ACM Symposium on Access Control Models and Technologies*. Lake Tahoe, USA : ACM Press, mai 2006, p. 49–58. doi : 10.1145/1133058.1133067 (cf. p. 21).

[23] P. Clemente, J.-F. Lalande et J. Rouzaud-Cornabas. **HoneyCloud : elastic honeypots - On-attack provisioning of high-interaction honeypots**. Dans : *International Conference on Security and Cryptography*. Rome, Italy : SciTePress, juillet 2012, p. 434–439. doi : 10.5220/0004129604340439 (cf. p. 6, 14).

[24] M. J. Covington, W. Long, S. Srinivasan, A. K. Dev, M. Ahamad et G. D. Abowd. **Securing context-aware applications using environment roles**. Dans : *Sixth ACM symposium on Access control models and technologies*. Lake Tahoe, USA : ACM Press, mai 2006, p. 49–58. doi : 10.1145/373256.373258 (cf. p. 21).

[25] J. Crampton. **Understanding and developing role-based administrative models**. Dans : *12th ACM conference on Computer and communications security*. Alexandria, USA : ACM Press, novembre 2005, p. 158. doi : 10.1145/1102120.1102143 (cf. p. 24).

[26] J. Crampton et G. Loizou. **Administrative scope : A foundation for role-based administrative models**. Dans : *ACM Transactions on Information and System Security* 6.2 (mai 2003), p. 201–231. doi : 10.1145/762476.762478 (cf. p. 24).

[27] F. Cuppens et A. Miege. **Modelling contexts in the Or-BAC model**. Dans : *19th Annual Computer Security Applications Conference*. Las Vegas, USA : IEEE Computer Society, p. 416–425. doi : 10.1109/CSAC.2003.1254346 (cf. p. 21).

[28] M. L. Damiani, C. Silvestri et E. Bertino. **Hierarchical Domains for Decentralized Administration of Spatially-Aware RBAC Systems**. Dans : *Third International Conference on Availability, Reliability and Security*. Barcelona, Spain : IEEE Computer Society, mars 2008, p. 153–160. doi : 10.1109/ARES.2008.44 (cf. p. 24).

[29] C. Decker et R. Wattenhofer. **Information propagation in the Bitcoin network**. Dans : *IEEE Thirteenth International Conference on Peer-to-Peer Computing*. Trento, Italy : IEEE Computer Society, septembre 2013, p. 1–10. doi : 10.1109/P2P.2013.6688704 (cf. p. 11).

[30] Y. Demchenko, O. Mulmo, L. Gommans, C. de Laat et A. Wan. **Dynamic security context management in Grid-based applications**. Dans : *Future Generation Computer Systems* 24.5 (2008), p. 434–441. doi : 10.1016/j.future.2007.07.015 (cf. p. 11).

[31] N. Dimmock, A. Belokosztolszki, D. Eyers, J. Bacon et K. Moody. **Using trust and risk in role-based access control policies**. Dans : *Ninth ACM symposium on Access control models and technologies*. Yorktown Heights, USA : ACM Press, juin 2004, p. 156–162. doi : 10.1145/990036.990062 (cf. p. 21).

[32] M. Fox, J. Giordano, L. Stotler et A. Thomas. **SELinux and grsecurity : A Side-by-Side Comparison of Mandatory Access Control and Access Control List Implementations**. Rapport technique (cf. p. 12, 16, 17).

[33] Frank Mayer, Karl MacMillan et David Caplan. *SELinux by Example*. Prentice Hall, 2006 (cf. p. 12).

[34] T. S. French, N. Bessis et C. Maple. **A High-Level Semiotic Trust Agent Scoring Model for Collaborative Virtual Organsations**. Dans : *IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*. Perth, Australia : IEEE Computer Society, avril 2010, p. 1114–1120. doi : 10.1109/WAINA.2010.173 (cf. p. 22).

[35] P. D. Giang, L. X. Hung, S. Lee, Y.-K. Lee et H. Lee. **A Flexible Trust-Based Access Control Mechanism for Security and Privacy Enhancement in Ubiquitous Systems**. Dans : *International Conference on Multimedia and Ubiquitous Engineering*. Seoul, Korea : IEEE Computer Society, avril 2007, p. 698–703. doi : 10.1109/MUE.2007.13 (cf. p. 21).

[36] D. Gros, M. Blanc, J. Briffaut et C. Toinard. **Advanced MAC in HPC Systems : Performance Improvement**. Dans : *2th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. Ottawa, Canada : IEEE Computer Society, mai 2012, p. 699–702. doi : 10.1109/CCGrid.2012.83 (cf. p. 27).

[37] D. Gros, M. Blanc, J. Briffaut et C. Toinard. **PIGA-cluster : A distributed architecture integrating a shared and resilient reference monitor to enforce mandatory access control in the HPC environment**. Dans : *International Conference on High Performance Computing & Simulation*. Helsinki, Finland : IEEE, juillet 2013, p. 273–280. doi : 10.1109/HPCSim.2013.6641426 (cf. p. 27).

[38] I. Haddad, C. Leangsuksun et S. L. Scott. **HA-OSCAR : the birth of highly available OSCAR**. Dans : *Linux Journal* 115 (novembre 2003) (cf. p. 16).

[39] C. Hecker, K. L. Nance et B. Hay. **Dynamic Honeypot Construction**. Dans : *10th Colloquium for Information Systems Security Education*. University of Maryland, USA, juin 2006 (cf. p. 13, 14).

[40] T. Holz et F. Raynal. **Detecting honeypots and other suspicious environments**. Dans : *Sixth Annual IEEE SMC Information Assurance Workshop*. University of Maryland, USA : IEEE Computer Society, juin 2005, p. 29–36. doi : 10.1109/IAW.2005.1495930 (cf. p. 13).

[41] S. Innes et C. Valli. **Honeypots : How do you know when you are inside one ?** Dans : *4th Australian Digital Forensics Conference*. Sous la dir. de C. Valli et A. Woodward. Perth, Western Australia : School of Computer et Information Science, Edith Cowan University, 2005. doi : 10.4225/75/57b131f0c7052 (cf. p. 13).

[42] T. Jaeger, R. Sailer et X. Zhang. **Analyzing Integrity Protection in the SELinux Example Policy**. Dans : *USENIX Security Symposium*. San Antonio, Texas, USA, août 2003 (cf. p. 17).

[43] M. Kaaniche, Y. Deswarte, E. Alata, M. Dacier et V. Nicomette. **Empirical analysis and statistical modeling of attack processes based on honeypots**. Dans : *Workshop on Empirical Evaluation of Dependability and Security*. Philadelphia, USA, juin 2006, p. 119–124. arXiv : 0704.0861 (cf. p. 13).

[44] L. Kuwatly, M. Sraj, Z. Al Masri et H. Artail. **A dynamic honeypot design for intrusion detection**. Dans : *The IEEE/ACS International Conference on Pervasive Services*. Beirut, Lebanon : IEEE Computer Society, p. 95–104. doi : 10.1109/PERSER.2004.1356776 (cf. p. 13).

[45] B. W. Lampson. **Protection**. Dans : *The 5th Symposium on Information Sciences and Systems*. Princeton University, mars 1971, p. 437–443 (cf. p. 11).

[46] C. Leangsuksun, A. Tikotekar, M. Pourzandi et I. Haddad. **Feasibility study and early experimental results towards cluster survivability**. Dans : *6th International Conference on Linux Clusters*. Cardiff, UK : IEEE Computer Society, mai 2005, p. 77–81. doi : 10.1109/CCGRID.2005.1558537 (cf. p. 16, 17).

[47] N. Li et Z. Mao. **Administration in role-based access control**. Dans : *2nd ACM symposium on Information, computer and communications security*. Fairfax, USA : ACM Press, novembre 2007, p. 127. doi : 10.1145/1229285.1229305 (cf. p. 24, 25).

[48] S. B. Liu et L. Palen. **A Survey of Current Tools to Inform Next Generation Crisis Support**. Dans : *6th International Conference on Information Systems for Crisis Response and Management*. Sous la dir. de J. Landgren et S. Jul. Gothenburg, Sweden, 2009 (cf. p. 20).

[49] Y. Liu. **Trust-Based Access Control for Collaborative System**. Dans : *International Colloquium on Computing, Communication, Control, and Management*. Guangzhu, China : IEEE Computer Society, août 2008, p. 444–448. doi : 10.1109/CCCM.2008.203 (cf. p. 21).

[50] P. Loscocco et S. Smalley. **Integrating Flexible Support for Security Policies into the Linux Operating System**. Dans : *11th USENIX Security Symposium*. Boston, Massachusetts, USA, juin 2001 (cf. p. 12).

[51] Medusa DS9. *Medusa DS9 Security System*. 2006 (cf. p. 12).

[52] Michael Ihde et Tom Brown. *An Experimental Study of File Permission Vulnerabilities Caused by Single-Bit Errors in the SELinux Kernel Policy File* (cf. p. 17).

[53] S. E. Minear. **Providing policy control over object operations in a mach based system**. Dans : *The 5th conference on USENIX UNIX Security Symposium*. Salt Lake City, Utah : USENIX Association, juin 1995, p. 141–156 (cf. p. 12).

[54] A. Mohan et D. M. Blough. **An Attribute-based Authorization Policy Framework with Dynamic Conflict Resolution**. Dans : *9th Symposium on Identity and Trust on the Internet*. Gaithersburg, USA : ACM Press, avril 2010, p. 37–50. doi : 10.1145/1750389.1750395 (cf. p. 22).

[55] A. Ott. **Rule Set Based Access Control as proposed in the 'Generalized Framework for Access Control' approach in Linux**. Mémoire de Master. Universität Hamburg, novembre 1997 (cf. p. 12).

[56] M. Pourzandi. **A new distributed security model for Linux clusters**. Dans : *The annual conference on USENIX Annual Technical Conference*. Boston, USA, 2004, p. 231–236 (cf. p. 16).

[57] M. Pourzandi, A. Apvrille, E. Gingras, A. Medenou et D. Gordon. **Distributed Access Control for Carrier Class Clusters**. Dans : *Parallel and Distributed Processing Techniques and Applications Conference*. Las Vegas, Nevada, USA : CSREA Press, juin 2003, p. 132–137 (cf. p. 16).

[58] M. Pourzandi, I. Haddad, C. Levert, M. Zakrzewski et M. Dagenais. **A Distributed Security Infrastructure for Carrier Class Linux Clusters**. Dans : *Ottawa Linux Symposium*. Ottawa, Canada, juin 2002, p. 439–450 (cf. p. 16).

[59] N. Provos. **A virtual honeypot framework**. Dans : *13th USENIX Security Symposium*. San Diego, USA, août 2004 (cf. p. 13).

[60] R. S. Sandhu, E. J. Coyne, H. L. Feinstein et C. E. Youman. **Role-Based Access Control Models**. Dans : *IEEE Computer* 29.2 (février 1996), p. 38–47. doi : 10.1109/2.485845 (cf. p. 12, 21).

[61] R. Sandhu, V. Bhamidipati, E. Coyne, S. Ganta et C. Youman. **The ARBAC97 model for role-based administration of roles**. Dans : *Second ACM workshop on Role-based access control*. Fairfax, USA : ACM Press, novembre 1997, p. 41–50. doi : 10.1145/266741.266752 (cf. p. 24).

[62] N. Schurr, J. Marecki, J. P. Lewis, M. Tambe et P. Scerri. **The DEFACTO System : Training Tool for Incident Commanders**. Dans : *17th Conference on Innovative Applications of Artificial Intelligence*. Sous la dir. de Bruce Porter. Pittsburgh, USA : AAAI Press, juillet 2005, p. 1555–1562 (cf. p. 20).

[63] L. Seitz, E. Rissanen, T. Sandholm, B. Firozabadi et O. Mulmo. **Policy administration control and delegation using XACML and Delegent**. Dans : *The 6th IEEE/ACM International Workshop on Grid Computing, 2005*. Seattle, USA : IEEE Computer Society, novembre 2005, 6 pp. doi : 10.1109/GRID.2005.1542723 (cf. p. 24).

[64] S. Smalley. **Configuring the SELinux Policy**. Rapport technique NAI Labs Report #02-007. Février 2002, p. 1–35 (cf. p. 17).

[65] S. Smalley et T. Fraser. **A Security Policy Configuration for the Security-Enhanced Linux**. Rapport technique. NSA, décembre 2000 (cf. p. 12).

[66] W. W. Smari, P. Clemente et J.-F. Lalande. **An extended attribute based access control model with trust and privacy : Application to a collaborative crisis management system**. Dans : *Future Generation Computer Systems* 31.- (février 2014), p. 147–168. doi : 10.1016/j.future.2013.05.010 (cf. p. 6, 21).

[67] W. W. Smari, J. Zhu et P. Clemente. **Trust and privacy in attribute based access control for collaboration environments**. Dans : *11th International Conference on Information Integration and Web-based Applications & Services*. Kuala Lumpur, Malaysia : ACM Press, décembre 2009, p. 49. doi : 10.1145/1806338.1806356 (cf. p. 21).

[68] B. Sobesto, M. Cukier, M. Hiltunen, D. Kormann, G. Vesonder et R. Berthier. **DarkNOC : dashboard for honeypot management**. Dans : *25th international conference on Large Installation System Administration*. USENIX Association, 2011 (cf. p. 13).

[69] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier et L. Peterson. **Container-based operating system virtualization : a scalable, high-performance alternative to hypervisors**. Dans : *European Conference on Computer Systems* 41.3 (2007), p. 275–287. doi : 10.1145/1272998.1273025 (cf. p. 16).

[70] B. Spengler. **Detection, Prevention, and Containment : A Study of grsecurity**. Dans : *Libre Software Meeting*. Bordeaux, France, juillet 2002 (cf. p. 12).

[71] R. K. Thomas. **Team-based access control (TMAC)**. Dans : *Second ACM workshop on Role-based access control*. Fairfax, USA : ACM Press, novembre 1997, p. 13–19. doi : 10.1145/266741.266748 (cf. p. 21).

[72] R. K. Thomas et R. S. Sandhu. **Task-Based Authorization Controls (TBAC) : A Family of Models for Active and Enterprise-Oriented Autorization Management**. Dans : *IFIP TC11 WG11.3 Eleventh International Conference on Database Security*. Lake Tahoe, USA : Chapman & Hall, Ltd., août 1997, p. 166–181 (cf. p. 21).

[73] S. Underwood. **Improving disaster management**. Dans : *Communications of the ACM* 53.2 (février 2010), p. 18. doi : 10.1145/1646353.1646362 (cf. p. 20).

[74] D. Walsh. *A step-by-step guide to building a new SELinux policy module*. Red Hat Magazine. 2007 (cf. p. 12, 18).

[75] D. Wolfe. **Computer forensics**. Dans : *Computers & Security* 22.1 (janvier 2003), p. 26–28. doi : 10.1016/S0167-4048(03)00105-6 (cf. p. 13).

[76] C. Wright, C. Cowan, J. Morris, S. Smalley et G. Kroah-Hartman. **Linux security modules : general security support for the linux kernel**. Dans : *Foundations of Intrusion Tolerant Systems*. Los Alamitos, California : IEEE Computer Society, décembre 2003, p. 213–226. doi : 10.1109/FITS.2003.1264934 (cf. p. 16).

[77] L. Youseff, R. Wolski, B. Gorda et C. Krintz. **Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems**. Dans : *Second International Workshop on Virtualization Technology in Distributed Computing*. Tampa, Florida : IEEE Computer Society, novembre 2006. doi : 10.1109/VTDC.2006.4 (cf. p. 16).

[78] W. Yurcik et G. Koenig. **Cluster security as a unique problem with emergent properties : Issues and techniques**. Dans : *5th LCI International Conference on Linux Clusters*. Austin, Texas, USA, 2004 (cf. p. 9).

[79] J. Zhu et W. W. Smari. **Attribute Based Access Control and Security for Collaboration Environments**. Dans : *IEEE National Aerospace and Electronics Conference*. Dayton, USA : IEEE Computer Society, juillet 2008, p. 31–35. doi : 10.1109/NAECON.2008.4806511 (cf. p. 21).

Chapitre 3

Sécurité du flot de contrôle

Le flot de contrôle d'un programme représente l'ordre dans lequel les instructions d'un programme sont exécutées. Evoqué dans [86], Frances Allen présente le flot de contrôle d'un programme comme un graphe orienté où les nœuds du graphe sont des *basic blocks* et les arcs des enchainements possibles de *basic blocks*. Dans un tel graphe, certains nœuds sont des nœuds d'entrée (le programme démarre) ou des nœuds de sortie (le programme s'arrête). Frances Allen définit alors un chemin comme une suite de *basic blocks* qui peuvent éventuellement se répéter dans ce chemin. On obtient donc un chemin d'exécution sur le graphe de flot de contrôle, et cette définition n'a pas changé aujourd'hui.

Au début des années 2000, l'intégrité du contrôle de flot devient une préoccupation importante des chercheurs en sécurité [138, 130, 82] : si le programme s'écarte de son flot de contrôle normal, il est probable qu'une erreur ou un attaquant ait perturbé une variable ou un registre. Même s'il n'est pas encore clair du bénéfice obtenu par une modification du flot de contrôle, un scénario simple d'utilisation est le saut d'un code réalisant des vérifications de sécurité (authentification, calcul de checksum ou hash, etc.). En 2007, Hovav Shacham introduit l'attaque ROP (*Return-Oriented Programming*) [165] qui montre comment construire un code d'attaque en exploitant des sauts arbitraires dans un code exécutable. En cherchant des morceaux de codes intéressants pour l'attaquant, des gadgets, et en combinant ces gadgets, on peut exécuter un code malveillant en utilisant le code bénin originel ou du code connu à l'avance comme celui de bibliothèques. La sécurisation du flot de contrôle devient d'autant plus un enjeu important. Récemment, Schuster et al. ont introduit les attaques COOP *Counterfeit Object-Oriented Programming* qui s'appuient sur des appels malveillants de méthodes virtuelles en C++ [162]. Ces attaques sont encore plus difficiles à détecter.

Pour des systèmes embarqués auxquels les attaquants ont un accès physique et qui sont sensés résister à des attaques physiques [90], la perturbation du flot de contrôle peut malgré tout être très précise car les attaques peuvent être reproduites de nombreuses fois. Cela peut permettre de faire fuiter des secrets ou d'exécuter des fonctionnalités normalement non autorisées. Parmi ces systèmes, les cartes à puce représentent une cible de choix étant donné qu'elles hébergent des secrets et des algorithmes sensibles comme par exemple un système d'authentification. En 2006, Bar-El et al. réalisèrent une synthèse des attaques physiques connues qui peuvent influer sur les algorithmes cryptographiques d'une carte à puce [98]. Il se développa alors une littérature abondante sur les attaques par fautes, qui peuvent se produire au niveau d'un programme natif ou dans une machine virtuelle Java. Le flot de contrôle n'est pas la seule préoccupation de la communauté car la corruption de données peut avoir de lourdes conséquences sur la robustesse des algorithmes cryptographiques. Pour chaque attaque découverte, des remédiations sont proposées et en 2011, Verbauwhede et al. proposent une synthèse entre les modèles de fautes et les contremesures possibles [171]. Les attaques ciblant le flot de contrôle y sont présentées et leur description se concentre sur l'intégrité des calculs cryptographiques.

Du point de vue du développeur de systèmes embarqués, aucune méthode particulière n'est préconisée pour le développement de code, vis-à-vis des menaces référencées dans la littérature. Le nombre de logiciels embarqués augmentant rapidement, l'impact potentiel d'une attaque se multiplie : carte bancaires, smartphones, objets connectés, box internet, routeurs peuvent se retrouver confrontés à une faille potentielle affectant plusieurs systèmes qui partagent une brique logicielle commune. Conscient de ce problème, l'industrie de la carte à puce se concentre sur l'obtention de la certification EAL4+ pour les applications de paiement, de diffusion télévisuelle ou de gestion d'identité [164], afin de garder une position de choix dans l'embarqué à l'heure ou d'autres acteurs économiques tentent d'imposer des solutions alternatives comme les *Trusted Execution Environments*. Ainsi, il persiste un intérêt certain à développer des attaques, élaborer des contremesures et proposer des méthodes pour

augmenter les défenses logicielles pour de tels systèmes. L'intégrité du flot de contrôle est un des problèmes importants qu'il convient d'adresser.

Dans ce chapitre, je présente les contributions qui touchent à l'intégrité du flot de contrôle d'un programme. Après avoir donné quelques éléments de base sur le flot de contrôle et comme annoncé en introduction, j'adopte une démarche en trois temps.

Je présente tout d'abord les attaques permettant de perturber le flot de contrôle et leur intérêt opérationnel (section 1). Un point original de ce chapitre réside dans le fait que je montre que l'on peut utiliser les attaques du flot de contrôle à des fins bienveillantes, en l'occurrence pour aider l'étude de logiciels malveillants pour les téléphones mobiles. Dans ce cas précis, l'étude expérimentale permet d'extraire de nouvelles connaissances d'un ensemble de *malware* Android. A l'inverse, je présente aussi la dangerosité des attaques contre le flot de contrôle pour les cartes à puce lorsque celles-ci sont soumises à des attaques physiques. Les attaques physiques nécessitant un matériel particulier, l'étude proposée se focalise sur la simulation logicielle de ces attaques et les connaissances que l'on peut en extraire. Ces deux exemples complémentaires traitant des attaques du contrôle de flot montrent que la problématique est transverse à plusieurs systèmes. Ainsi, cela motive l'intérêt de se concentrer sur la problématique du flot de contrôle à haut niveau, c'est-à-dire au niveau du code source d'un programme, afin de pouvoir réutiliser les solutions dans plusieurs systèmes cibles.

Je présente ensuite les contremesures élaborées spécialement pour les codes C (section 3), possiblement embarquées sur carte à puce, qui sécurisent le flot de contrôle. Faisant écho aux attaques physiques présentées juste avant, ces contremesures cherchent à détecter une rupture dans le flot de contrôle afin d'arrêter l'exécution du programme et lever une réponse sécuritaire. Ces contremesures cherchent à s'intégrer dans le processus de développement et sont donc conçues pour s'intégrer directement au code source, ceci afin de pouvoir passer par une étape de certification.

Enfin, une étude expérimentale et formelle est proposée. L'étude expérimentale donne une idée précise des coûts et des performances des contremesures proposées. Elle montre aussi l'intérêt de contremesures à haut niveau dans un processus de développement ou une campagne de test de sécurité doit pouvoir être menée rapidement après la phase de développement. L'étude formelle propose quant à elle de garantir la justesse des contremesures proposées c'est-à-dire de vérifier à l'aide d'un *model checker* qu'aucune attaque du contrôle de flot n'a été oubliée.

Je termine le chapitre en décrivant brièvement les logiciels développés relatifs à ces contributions et je donne quelques perspectives de recherche.

1 Introduction aux modèles d'attaque du flot de contrôle

Le flot de contrôle est influencé par la nature des instructions exécutées : soit les instructions se font en séquence, soit une instruction est une rupture de cette séquence. Lorsque le processeur rencontre une instruction de type *call* qui indique l'adresse de l'instruction suivante, il transfert le contrôle à la portion de code qui correspond à l'appel. Lorsque la procédure se termine, le processeur redonne le contrôle à l'instruction qui a suivi le *call*. Ce mécanisme nécessite de sauver l'adresse de retour, l'état des registres vivants, de connaître la localisation des fonctions et de traiter le passage des paramètres. Suivant le matériel et les implémentations, la gestion de ces données varie, notamment leur emplacement sur la pile.

Dans la littérature sur les attaques contre le flot de contrôle [82, 126], il est classique de considérer les attaques influant la pile pour les programmes natifs. Par exemple, dans une *frame* de la pile, un calcul de pointeur simple à partir d'une variable locale peut permettre d'aller modifier l'adresse de retour de la fonction associée à cette *frame*. Un autre exemple d'attaque est le débordement de pile qui peut permettre d'aller écrire dans les variables de la section BSS du segment de données (en bout de pile) où l'on peut par exemple écraser des variables globales. On peut tenter des attaques similaires pour du bytecode interprété par une machine virtuelle mais bien souvent, des vérifications sont faites à l'exécution, ce qui rend ces attaques plus difficiles à mettre en œuvre [107].

Ainsi, pour décrire précisément des attaques contre le contrôle de flot, il faut considérer le programme à un niveau plus bas que son code source. Il est nécessaire de disséquer le fonctionnement de l'interface binaire-programme, la gestion de la pile et le *mapping* mémoire (notamment la gestion de la pile lors des appels de fonctions). Ces éléments sont spécifiques à chaque architecture et différents suivant que l'on considère un programme natif ou interprété. Dès lors, il est difficile de modéliser ceux-ci, notamment pour la gestion de la mémoire. Dans la suite, nous décrivons les éléments communs à des programmes pour caractériser le flot de contrôle et nous introduisons un modèle général d'attaque de ce flot de contrôle, quelle que soit l'architecture considérée. Ces éléments permettront ensuite d'élaborer des attaques pour deux types de systèmes distincts en section 2, les cartes à puce et les téléphones mobiles.

1.1 Niveaux de représentation d'un modèle d'attaque

Une attaque contre un programme peut être considéré à différents niveaux de représentation de ce programme ou du matériel sur lequel il s'exécute [171] :

- Niveau matériel : des composants spécifiques du matériel sont ciblés et peuvent être altérés ; il peut s'agir de bus de transmission, de zone mémoire ou de composants spécialisés comme l'horloge ou l'alimentation ;
- Niveau assembleur : des registres ou des cases mémoire peuvent être altérés ;
- Niveau *bytecode* : dans le cas de l'utilisation d'une machine virtuelle, à ce niveau, le code ou les variables peuvent être altérés ;
- Niveau du code source : à ce niveau, les attaques peuvent être conceptualisées et prises en compte mais l'attaquant dispose rarement de la capacité à modifier le code source puisqu'il attaque un composant cible où le code compilé s'y exécute.

Pour chacun de ces niveaux, les entités à considérer sont différentes. Partons d'un exemple simple ou l'on appelle une fonction vérifiant l'égalité entre deux codes pin. Comme le montre les trois listings ci-dessous (à gauche le cas natif, à droite le cas d'une machine virtuelle), les entités du programme à considérer sont très différentes :

```
CALL  check_pin
MOV   R0,#pin
MOV   A,R7
MOV   @R0,A
MOV   R7,#pin
RET
```

Assembleur

```
pin = check_pin(1234,0000);
return pin;
```

Code source

```
invokestatic   #3
istore_0
iload_0
ireturn
```

Bytecode

Il est alors clair que le modèle d'attaque est plus facile à exprimer si l'on considère le code au plus près de sa nature lors de son exécution, en l'occurrence l'assembleur pour du code natif ou du *bytecode* pour du code interprété. Cependant, il est bien évident qu'un attaque décrite à un tel niveau de détail du code sera plus complexe à comprendre d'un point de vue de l'expert en sécurité, surtout s'il veut faire un lien avec le source de l'application.

Pour pouvoir décrire une attaque, il faut arriver à caractériser ce que l'attaquant est en capacité de réaliser sur le programme, en l'exprimant sur un des niveaux de description du programme vus précédemment. La difficulté réside dans le fait que les moyens dont dispose l'attaquant dépendent du système cible et de la nature physique

de l'attaque, et qu'il y a donc de très grandes variations suivant les hypothèses considérées. Par exemple, pour un code malveillant ayant été introduit à l'intérieur d'un binaire compilé, celui-ci dispose de beaucoup plus de privilèges pour modifier son code ou les variables du programme. A l'inverse, dans le cas d'attaques physiques ciblant des carte à puce, l'attaquant ne maîtrise pas forcément les modifications qu'il opère sur les valeurs des variables ou les instructions du programme [171]. Cependant, quelles que soient les hypothèses, on peut synthétiser ces niveaux de précision dans un ensemble de grandes classes générales d'effets d'attaque permettant de caractériser le pouvoir de l'attaquant [174, 171] :

— *Bit flip/set* : l'attaquant peut effectuer une modification au bit près ;
— *Word randomize/set* : l'attaquant peut effectuer une modification sur un mot, en général un octet ;
— *Variable ou code du programme* : à ce niveau de précision, on cherche à se rendre indépendant de l'architecture cible et la plage de modification est considérée très grande.

Cette classification est issue des modèles utilisés dans les travaux de tolérance aux fautes, comme le précise Abadi *et al.* [82]. Malheureusement, ceux-ci notent que ces classes sont peu adaptées quand on considère la sécurité vis-à-vis d'un attaquant. Notamment le modèle bit flip représente une erreur qui survient à cause d'une défaillance matérielle et les contremesures de la littérature utilisent par exemple des codes correcteurs d'erreur pour vérifier l'intégrité en sortie d'un *basic block* [151, 157]. Si les garanties dépendent de probabilités, elles ne conviennent plus pour un modèle d'attaquant qui cherche à corrompre de manière délibérée un programme en utilisant éventuellement des vulnérabilités.

Ces différents niveaux de précision de l'attaque sont liés à l'ensemble des éléments qui influent sur le contrôle de flot. Il peut s'agir d'attaquer le *program counter* [108], un élément d'une *frame* de la pile [97], une variable globale dans une section de données [126], une variable locale de la RAM [120], un registre qui influe sur un test conditionnel [106], une instruction (opcode et/ou opérande) du code exécuté [172], etc. Pour tous ces éléments, on peut considérer une attaque influant sur un bit, un octet ou bien directement sur l'ensemble des octets constituant la valeur de l'élément.

On peut ajouter un raffinement si l'on distingue le cas où l'attaquant contrôle la valeur de la modification ou si la modification prend une valeur aléatoire. Si l'on considère par exemple une attaque permettant d'écraser un mot avec une valeur contrôlée (e.g. *0F0H*) on peut par exemple écraser la variable pin de l'exemple précédent, c'est-à-dire une partie des opérandes pour le code assembleur mais un *opcode* pour du bytecode :

```
CALL check_pin
MOV  R0,#pin
MOV  A,#0F0H // instead of R7
MOV  @R0,A
MOV  R7,#pin
RET
```

Assembleur

```
pin = check_pin(1234,0000);
pin = 0x0F0;
return pin;
```

Code source

```
invokestatic  #3
istore_0
iload_1 // instead of iload_0
ireturn
```

Bytecode

L'écrasement d'un *opcode* par une valeur fixe est un exemple classique de la littérature [163, 149, 136] : elle consiste à remplacer une opération par un NOP (qui peut être une valeur particulière au niveau binaire, par exemple des 0 ou des 1). Au niveau d'un langage interprété, la machine virtuelle peut se rendre compte d'un problème de consistance de la pile, comme dans l'exemple ci-dessous qui illustre l'effet d'un NOP sur les trois niveaux de représentation :

```
NOP
MOV  R0,#pin
MOV  A,R7
MOV  @R0,A
MOV  R7,#pin
RET
```

Assembleur

```
return pin;
```

Code source

```
invokestatic  #3
istore_0
nop // instead of iload_0
ireturn // oops !
```

Bytecode

Cependant, cela suppose que le système considéré dispose d'un vérificateur de bytecode, ce qui peut ne pas être le cas. Pour des programmes natifs, il n'y a en général pas de vérification de ce type, et le remplacement d'un *opcode* par un NOP permet de réaliser le saut d'une instruction, réalisant une attaque contre le flot de contrôle.

1.2 Temporalité de l'attaque

Suivant la manière dont l'attaquant influe sur le programme ou les données de celui-ci, l'attaque n'aura pas la même portée temporelle.

Si l'attaquant modifie les *opcodes* ou opérandes du programme dans le conteneur matériel qui les stocke (Flash, ROM, RAM, cache) [172], l'attaque sera persistante temporellement. A chaque fois que le processeur exécutera cette partie du code en le chargeant depuis le conteneur matériel, alors les *opcodes* ou opérandes modifiés

par l'attaquant seront exécutés. Ce type d'attaque est une attaque dite "permanente". Elle persiste pendant toute la durée de l'exécution du programme (sauf pour une attaque de cache ou l'attaque persiste sur une certaine durée) et s'apparente plus au chargement d'un programme malveillant qu'à la modification lors de l'exécution d'un code sain.

Inversement, l'attaque peut produire des effets qui ne sont pas persistants temporellement. On parle alors d'attaque "transiente" [90, 166] pour signifier que l'effet de bord sur le programme obtenu par l'attaquant ne persiste pas au-delà de l'instant où l'attaque est perpétrée. Si, par exemple, l'attaque s'appuie sur la modification des instructions qui circulent sur le bus de chargement du programme avant leur exécution sur le processeur, alors une attaque pendant un premier chargement d'une portion de code n'aura pas d'influence sur le prochain chargement de cette même portion de code. De même, une *frame* comportant des variables locales modifiées n'aura pas de conséquence sur cette même *frame* pour un prochain appel de fonction. Ainsi, de nombreux cas où l'attaque est dynamique, c'est-à-dire réalisée au moment de l'exécution du programme sera transiente si elle touche des ressources du programme qui sont elles-mêmes temporaires (*frame*, registre, bus de donnée, etc.).

1.3 Modèles d'attaque du flot de contrôle

Nous proposons dans cette section trois modèles d'attaque du flot de contrôle [103, 3]. La particularité de ces modèles est de corrompre le flot normal du programme tout en préservant les données manipulées par le programme. Ces modèles peuvent paraître équivalents, mais nous montrons par la suite qu'ils ne permettent pas de représenter les mêmes ensembles d'attaques réelles.

Modèle d'attaque du flot de contrôle au niveau C Au niveau source, nous avons proposé de représenter une telle attaque par l'insertion d'une instruction *goto label;* au sein du code source comme montré en listing 3.1. Lorsque l'attaque n'est pas permanente sur le chemin d'exécution, on implémente une variante qui permet de déclencher l'attaque au moment voulu, comme montré en listing 3.2.

```
int f() {
  ...
  goto label;
  ...
  ...
  label:
  ...
}
```

Listing 3.1 – Attaque permanente en C

```
int attack_trigger = false; /* is the attack on? */
int time_is_reached = false; /* synchronization condition */
int time_is_over = false; /* stop condition */
int f() {
  ...
  if (! attack_trigger && time_is_reached && !time_is_over)
  {
    /* perform the attack */
    attack_trigger = true;
    goto label;
  }
  ...
  label:
}
```

Listing 3.2 – Attaque transiente en C

Un tel modèle d'attaque n'a de sens que pour des sauts intra fonction (et non inter fonctions). Toutefois, il est réaliste car un saut en dehors d'une fonction met en général le programme dans un état instable car le contenu de la pile est inconsistant.

Modèle d'attaque du flot de contrôle au niveau assembleur Au niveau assembleur, une démarche similaire permet d'implanter des sauts inconditionnels à l'aide de l'opcode *jmp*. Lorsque l'attaque est transiente, le code se complexifie. Il est donné en annexe 1 : il consiste à sauter sur une portion de code *.hack* qui va sauvegarder l'état des registres, évaluer si l'attaque doit être déclenchée, puis soit revenir en séquence dans le code initial, soit sauter à la destination de l'attaque *.desthack*.

```
  .file "main.c"
  .text
  ...
  jmp .desthack
  ...
  ...
  .desthack:
  ...
}
```

Listing 3.3 – Attaque permanente en assembleur

```
..
  .debuthack:
  /** Saut vers .hack pour evaluer le declenchement **/
  jmp .hack
  .finhack:
...
...
/** Destination de l'attaque **/
.desthack:
```

Listing 3.4 – Attaque transiente en assembleur

Modèle d'attaque du flot de contrôle au niveau du bytecode Enfin, pour le bytecode Java, on procède de façon similaire avec l'instruction *goto*, comme montré ci-dessous dans la version Jimple de la représentation du bytecode.

```
// Version Jimple par Soot
public static int main(int) {
 ...
goto label1 ;
 ...
label1 :
 ...
}
```

Listing 3.5 – Attaque permanente en bytecode

```
// Version Jimple par Soot
public static int main(int) {
 ...
$i0 := ...
if $i0 != 1 goto label1 ;
 ...
label1 :
}
```

Listing 3.6 – Attaque transiente en bytecode

Les modèles d'attaque du flot de contrôle présentés dans cette section auront deux utilisations différentes. D'une part, ils vont permettre de caractériser les attaques subies afin d'élaborer des défenses dans les cartes à puce ; d'autre part, ils permettront d'élaborer des attaques afin d'attaquer les *malware* Android. Pour les attaques subies, on cherchera à comprendre si une attaque physique peut provoquer la rupture du flot de contrôle et quelles sont les conséquences d'un point de vue sécurité. A l'inverse, pour les attaques élaborées, le modèle d'attaque permettra de construire des attaques contre des *malware* afin d'obtenir un gain sur ceux-ci, en l'occurrence une meilleure compréhension de ces *malware*.

2 Elaboration d'attaques

Dans cette section, nous présentons deux cas d'usage des modèles d'attaque du flot de contrôle définis précédemment.

Le premier cas d'usage concerne les applications Android. Pour ce cas, l'approche est inversée : au lieu d'étudier les attaques contre le flot des applications bienveillantes [119], nous proposons de perturber le flot de contrôle d'une application malveillante (*malware* Android) afin d'améliorer son exécution. L'objectif est d'améliorer la compréhension du *malware* étudié, notamment lorsque celui-ci tente d'échapper à une analyse dynamique [84].

Le deuxième cas d'usage présenté concerne les attaques physiques contre les cartes à puce. L'objectif pour ce cas est de réaliser une campagne de simulation de fautes physiques contre des codes de carte à puce [142]. Une telle campagne permettra d'évaluer l'impact des attaques sur un code C, aidant le développeur à comprendre précisément pourquoi une attaque permet de mettre en danger la sécurité de son code. Elle permettra ensuite d'évaluer la robustesse des contremesures que nous proposons par la suite.

2.1 Attaques contre les malware Android

De nombreuses applications (entre 1% et 9% [133]) du Google Play Store sont identifiées comme des *malware* et les *markets* alternatifs extérieurs en sont infestés. Les éditeurs d'antivirus, les chercheurs et les professionnels du monde de la sécurité s'attellent donc au difficile travail de *reverse engineering* de ces *malware*, afin d'en comprendre les dangers. Une partie de ces investigations consiste à s'intéresser à l'analyse dynamique de *malware* (le pendant de l'analyse statique) afin d'observer, dans un environnement contrôlé, leurs agissements [123]

L'analyse dynamique de *malware* doit cependant faire face à de nouveaux problèmes. Les *malware* peuvent utiliser des techniques pour éviter ces analyses : le chargement dynamique de code [154], la détection d'un environnement de type bac à sable virtuel [173], l'utilisation de transformations pour déjouer des techniques de détection basées sur les signatures [159]. Ainsi, l'analyse dynamique devient un réel challenge, surtout si l'on souhaite la réaliser à large échelle. Si l'environnement est virtualisé, si le réseau n'est pas disponible, si certaines API sont manquantes, il est probable qu'un *malware* moderne n'exécute pas son code malveillant. C'est cette problématique que nous souhaitons résoudre ici, en utilisant notre modèle d'attaque du flot de contrôle.

2.1.1 Analyse dynamique de malware

L'analyse dynamique de malware Android peut être vue comme un sous-problème de l'observation d'applications Android. Une des méthodes les plus intéressantes pour observer une application consiste à tracer les flux d'informations internes aux applications en propageant des marques attachées aux variables d'un programme et/ou aux processus du système [94, 124, 91]. Ces méthodes ne résolvent pas particulièrement le problème du

TABLE 3.1 – Plate-formes et outils d'analyse de malware

Nom	Open Source	Online	Analyse Statique	Stimulation Dyn. GUI	Stimulation Dyn. Composants	Coop. Stat.+Dyn.	Surveillance
AASandBox [104]			✓	M			Appels systèmes et librairies
ANANAS [122]			✓	S			Fichiers, réseau, APIs
Andrubis [176, 145]		†	✓	M	✓	✓	*Taint tracking*, méthodes, code natif
AppsPlayground [158]				✓	✓		*Taint tracking*, kernel calls
CopperDroid[160, 168]		✎			✓		Appels systèmes, fic./net., objets
Crowdroid [110]				H			Appels systèmes
DroidRanger [184]			✓				Appel systèmes, code natif
DroidScope [180]	✎						Instructions natives/bytecode, *Taint*
SmartDroid [182]			✓	✓		✓	APIs sensibles
Mobile SandBox [167]	✎	✎	✓	M		✓	Code natif, réseaux
Vetdroid [181]				M			Permissions
GroddDroid [84]	✎	✎	✓	✓		✓	*Taint tracking*

Légende : ✎ : disponible † : plus disponible ✓ : oui (automatique) M : Monkey H : Humain S : Script

déclenchement de malware. Par ailleurs, elles peuvent être contournées, comme présenté en chapitre 4 en utilisant des canaux cachés. Une autre méthode courante pour observer un malware consiste à le confiner dans un environnement de type bac à sable dans lequel on peut surveiller les appels dans la machine virtuelle ou dans le noyau [104]. On prend alors le risque de voir le bac à sable détecté par l'application malveillante. Pour cette dernière raison, nous préférons donc exécuter nos *malware* dans un téléphone réel afin de maximiser les chances de leur bonne exécution.

Si l'on ne focalise pas sur les malware à propremenent parler, de nombreux outils ont été développés pour automatiser les tests d'applications sous Android [115]. Les premiers outils apparus en 2012, comme GUIRipper [88], parcourent l'interface graphique et découvrent les évènements qui peuvent être générés et potentiellement déclencher du nouveau code. Tous les outils orientés tests développés dans ce sens mettent plus ou moins d'intelligence pour augmenter la couverture du code. Choudhary *et al.* [115] distinguent les outils construisant un modèle de l'application afin d'en déduire les évènements intéressants à générer, par exemple à partir de l'analyse du code source ou des ressources de l'application. D'autres outils utilisent des approches qui calculent des évènements à partir de l'analyse du code non encore couvert. Il peut s'agir de tests concoliques [89], d'algorithmes évolutionnaires [147], etc. Toutes ces méthodes ne sont cependant pas dédiées à l'analyse de *malware* et manquent souvent de la partie observation des activités de celui-ci.

Lorsqu'il s'agit d'observer un *malware*, on s'intéresse au déclenchement et à l'observation du code malveillant et les approches de la littérature utilisent souvent des méthodes d'analyse dynamique pour focaliser l'analyse sur des parties du code qui sont suspicieuses. La table 3.1 présente une synthèse de ces outils et l'on peut lire une comparaison plus détaillée dans [150]. Une des premières approches, AASandBox, a été proposée par Bläsing *et al.* [104]. Dans un premier temps, AASandBox cherche des motifs de programmation tels que des appels JNI, l'exécution de binaires ou l'utilisation de code réflexif. Dans un second temps, l'exécution dynamique est surveillée au travers des appels systèmes. Cependant, les deux étapes ne coopèrent pas, c'est-à-dire que l'analyse dynamique ne tire pas profit de l'étape d'analyse statique. Ces premiers outils utilisaient le Monkey [132] comme stimulateur d'interface graphique, ce qui en réduisait l'efficacité puisque les évènements étaient aléatoires.

Les approches qui sont apparues ensuite étaient plus ou moins intrusives du point de vue de l'application ou du système d'exploitation. Une approche notable peu intrusive est DroidScope. DroidScope [180] surveille depuis une interface de virtualisation le code natif et le code s'exécutant dans la machine virtuelle Dalvik. Il n'est alors pas possible de manipuler l'interface graphique ou les composants de l'application. Cependant, Tam *et al.* ont proposé CopperDroid [160, 168] qui surveille par l'interface de virtualisation de Qemu le malware mais propose de reconstruire la sémantique des actions et des objets utilisés dans la machine virtuelle ou dans le processus natif. Ainsi, l'approche n'est plus sensible à des évolutions du code de l'OS. La majorité des autres approches préfèrent modifier le noyau, la machine virtuelle ou l'application elle-même afin de pouvoir extraire ou injecter des informations lors de l'exécution.

La plupart des outils proposés réalisent une analyse statique plus ou moins approfondie. Puis, lors de l'analyse dynamique, soit l'interface graphique n'est tout simplement pas stimulée (DroidRanger [184], DroidScope [180]), soit elle l'est en utilisant le Monkey [132] (Andrubis [176], Crowdroid [110], Mobile SandBox [167], Vetdroid [181]) ou des langages de script (ANANAS [122]). Le résultat étant souvent décevant, d'autres outils proposent une stimulation plus intelligente du malware comme AppsPlayground [158] qui stimule l'interface et les composants systèmes (service, *receivers*) ou SmartDroid [182] qui tente de ne parcourir que les activités utiles pour atteindre

le code malveillant. Enfin, Andrubis, SmartDroid et Mobile SandBox utilisent des résultats de l'analyse statique pour aider l'analyse dynamique. Il peut s'agir d'identifier la bonne séquence d'activités ou le bon composant système qu'il faut déclencher pour atteindre le code malveillant.

Enfin, certains outils utilisent des approches centrées sur l'utilisateur. CrowDroid [110] permet de rejouer des interactions collectées par des enregistrements utilisant des utilisateurs humaines (*crowdsourcing*). PuppetDroid [128] permet de rejouer automatiquement des enregistrements en reconnaissant des applications qui sont graphiquement similaires.

Les informations collectées sont de différentes natures et dépendent des méthodes utilisées [150]. Il peut s'agir du *taint tracking* c'est-à-dire de la surveillance d'une marque associée à un fichier ou une information utilisateur [91, 124, 144]. Il peut s'agir des ressources du téléphone, par exemple les accès aux fichiers ou au réseau. De nombreuses approches tracent les appels systèmes, les APIs et les méthodes exécutées dans l'application [178, 160, 81, 111]. Toutes ces informations permettent alors de caractériser les agissements du *malware*.

Il faut noter que peu de plate-formes dédiées à l'exécution de malware ont été réalisées ces dernières années [167, 176, 168]. La plus aboutie est sans doute Andrubis qui permet d'alimenter un cluster d'émulateurs et qui a réussi à analyser 1 million d'applications [145], dont 40% détectées comme malveillantes. Le service n'est plus disponible et les résultats scientifiques ont été injectés dans la start-up LastLine.

2.1.2 Exemples de malware protégeant leur comportement malveillant

Comme nous l'annoncions en introduction, les *malware* modernes cherchent à échapper à l'analyse statique et dynamique. Ils utilisent donc des techniques comme l'obfuscation, le chiffrement de code ou de données, ou tentent de détecter des potentiels observateurs. A titre d'exemple, on peut imaginer une protection simple qui se base sur l'appel à la méthode statique *isOnEmulator()*. Si celle-ci est vraie, alors un *malware* ne devrait sans doute pas exécuter son code malveillant. Une telle méthode peut être implémentée de bien des façons. Par exemple, on peut chercher le terme "SDK" dans le modèle de l'OS (astuce de certains développeurs : *android.os.Build.MODEL.contains("SDK");}*. D'un point de vue du code, pour un *malware* dont l'objectif serait d'envoyer des SMS frauduleux, on obtiendrait quelque chose qui ressemblerait à :

```
if (isOnEmulator())
   return; // Branch 1
else
   manager = SmsManager.getDefault(); // Branch 2
```

Listing 3.7 – Exemple de code avec une condition

```
   $z0 = staticinvoke <DummyClass: boolean isOnEmulator()>();
   if $z0 != 0 goto label1;
   return; // Branch 1
label1: // Branch 2
   $r6 = staticinvoke <SmsManager: SmsManager getDefault()>();
```

Listing 3.8 – Même code, en Jimple

Pour les *malware* réels que l'on trouve sur les différents *markets*, de nombreuses conditions peuvent être utilisées pour protéger le code malveillant. Nous en avons étudié 7 en détail afin d'en faire un jeu de référence où les conditions de déclenchement de chaque *malware* deviennent ainsi connues et documentées [140]. Parmi ces conditions on peut trouver l'attente d'un délai, l'attente d'une commande provenant d'un serveur, la disponibilité du réseau, le redémarrage du téléphone.

Pour résoudre ce problème appelé le *triggering*, les travaux antérieurs se focalisent sur les *input* extérieurs et ont donc une approche en boite noire. Andrubis [176], SmartDroid [182] et A3E [95] essaient d'envoyer tous les *Intents* (message inter applications) ou évènements graphiques possibles, détectés à travers l'analyse du Manifest de l'application. DynoDroid [146] réalise une boucle "observation-sélection-exécution" et possède différentes stratégies pour la phase de sélection. Cependant, DynoDroid se focalise sur l'interface, les *Intents* et les services et ne traite pas spécifiquement de la problématique du code malveillant et des conditions protégeant son déclenchement.

A notre connaissance, seul TriggerScope [127], une contribution postérieure à nos travaux, répond à la problématique que nous nous posons. TriggerScope propose une méthode d'analyse statique qui combine une exécution symbolique et une analyse des chemins d'exécution pour déterminer les variables qui impactent l'exécution du code malveillant. Lors de l'analyse symbolique, des prédicats sont créés et annotent les *basic blocks* afin de décrire les conditions qui contrôlent l'atteinte de ce *basic block*. Ainsi, le long d'un chemin, on peut reconstruire le prédicat global contrôlant l'atteinte d'un *basic block*. L'évaluation de l'outil sur un grand ensemble d'applications du *market* Android est impressionnante : Fratantonio *et al.* [127] montrent qu'avec une telle approche, on peut identifier des bombes logiques basées sur le temps, la localisation et la réception de SMS. Avec 14 *malware* cachés dans un pool de 9 313 applications, ils obtiennent un taux de faux positif de 0.38% et retrouvent l'ensemble des conditions de *triggering* pour les 14 bombes logiques. Une analyse fine sur 11 *malware* montre de bons résultats sur la sémantique des conditions de *triggering* extraites. Cependant, les auteurs ne vérifient jamais, car ce n'est

Table 3.2 – Score de risque pour chaque catégorie de classes

Catégorie	Classes associées	Score
SMS	android.telephony.SmsManager	50
Telephony	android.telephony.TelephonyManager	20
Binary	java.lang.Runtime java.lang.Process java.lang.System	10
Dynamic	dalvik.system.BaseDexClassLoader dalvik.system.PathClassLoader dalvik.system.DexClassLoader dalvik.system.DexFile	10
Reflection	java.lang.reflect.*	3
Crypto	javax.crypto.* java.security.spec.*	3
Network	java.net.Socket java.net.ServerSocket java.net.HttpURLConnection java.net.JarURLConnection	3

pas leur but, si l'extraction de ces conditions de *triggering* permettent de réellement réussir à exécuter le code malveillant.

De manière similaire à Fratantonio *et al.*, notre travail s'est focalisé sur les conditions qui protègent le code malveillant. Notre but n'est pas réellement d'extraire la sémantique des conditions protégeant le code malveillant mais de s'assurer qu'à l'exécution, celui-ci est bien exécuté.

2.1.3 Attaque du flot de contrôle d'un malware

Pour résoudre le problème évoqué précédemment, nous avons réalisé un outil permettant de "corriger" le flot de contrôle afin de pousser le flot d'exécution vers le code malveillant. Cet outil est une des briques du logiciel GroddDroid, présenté dans la section 4. Il se base sur l'utilisation de Soot qui permet d'analyser et de modifier le bytecode d'une archive APK [93]. Si l'on reprend l'exemple du code du listing 3.8, nous souhaitons forcer le flot d'exécution vers la seconde branche. C'est ce qui est réalisé par l'insertion d'une attaque au niveau du test de la variable $z0$. Le code attaqué est donné en listing 3.9 et montre que le test est simplement supprimé.

```
  $z0 = staticinvoke <DummyClass: boolean isOnEmulator()>();
  goto label1; // Forced branch 2
  return; // Branch 1
label1: // Branch 2
  $r6 = staticinvoke <SmsManager: SmsManager getDefault()>();
```

Listing 3.9 – Same sample code with forced control flow

La difficulté de l'approche réside dans le calcul du chemin d'exécution entre un point d'entrée du programme et le code malveillant. En effet, sous Android, les points d'entrée sont multiples : chaque activité, service, événement peut être considéré comme un point d'entrée. De plus, pour forcer le flot de contrôle vers le bon nœud du graphe de flot de contrôle, il faut identifier le code malveillant, ce qui n'est pas trivial.

2.1.4 Identification du code potentiellement malveillant

L'identification du code malveillant est réalisée par une heuristique de *scoring* qui s'appuie sur des statistiques d'utilisation de *malware* de l'API Android [81]. En effet, Aafer *et al.* ont remarqué que les *malware* utilisent plus fréquement certaines APIs, comme par exemple *android . telephony . SmsManager* qui permet d'envoyer des SMS à des numéros surtaxés. D'autres *malware* tentent de masquer leurs activités en chargeant dynamiquement du code ce qui impose de manipuler un chargeur de classe (*BaseDexClassLoader*). Enfin, l'utilisation de code réflexif est suspicieux puisqu'il peut s'agir d'une tentative pour tromper des algorithmes d'analyse statique.

A partir de ce *scoring* arbitraire donné en table 3.2 mais que l'on peut adapter, nous calculons un score de risque pour chaque méthode de l'application. Ce score ne sert pas à distinguer si une application est malveillante ou non (nous savons déjà que l'application est malveillante), mais sert uniquement à discriminer les méthodes entre elles pour identifier les méthodes malveillantes. Ainsi, les méthodes ayant le plus haut score sont celles que nous considérons comme potentiellement malveillantes. Par exemple, pour le *malware* SaveMe [140], les méthodes malveillantes sont identifiées si l'on ne prend pas en compte la réflexion qui perturbe la qualité du

Table 3.3 – Scoring pour le malware SaveMe

Method	Nb Units	Score
com.savemebeta.GTSTSR : void CHECK()	2	100
com.savemebeta.SCHKMS : void fetchContacts()	2	100
com.savemebeta.Scan : int uploadFile(java.lang.String)	12	36
com.savemebeta.Analyse : void onCreate(android.os.Bundle)	4	32
com.savemebeta.CHECKUPD : void onCreate()	4	32
com.savemebeta.CO : void onCreate()	2	16

Table 3.4 – Scoring pour le dataset Kharon [140]

Malware	Highest ranked method	Successful Targeting	Most scored method
BadNews	80	ok	gathers user information (phone number, IMEI, ...)
Cajino	200	ok	sends SMS with parameters from a C&C server
DroidKungFu	50	ok	starts a binary containing the exploit *udev*
MobiDash	147	wrong	gathers user information for legitimate use
SaveMe	100	ok	sends SMS with parameters from a C&C server
SimpleLocker	-	crash	-
WipeLocker	150	ok	sends SMS

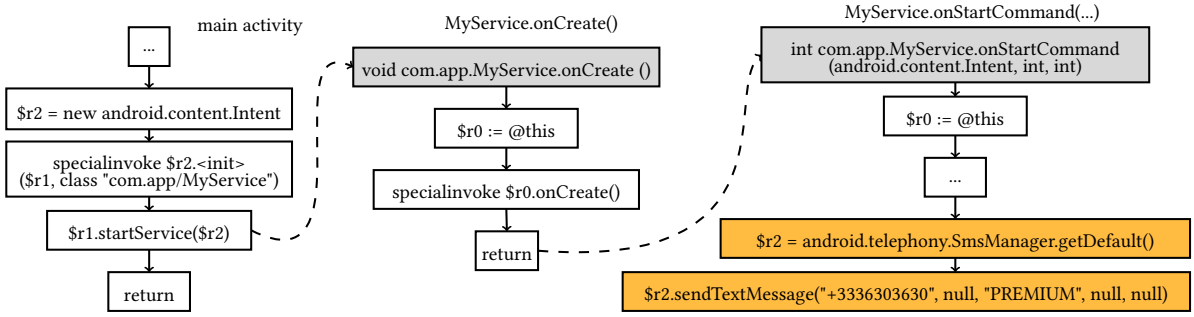


Figure 3.1 – Exemple de reconstruction de CFG pour une activité démarrant un service

résultat. Comme montré en table 3.3, les méthodes ciblées seront *CHECK()* et *fetchContacts ()*. Nous avons testé la qualité du ciblage de l'heuristique sur un nouveau dataset que nous avons créé pour l'occasion, appelé *Kharon dataset* [140]. A la différence des larges dataset comme le *Genome Project* [183], Contagio mobile [148] ou *AndroZoo* [87] pour lesquels les parties du code qui sont malveillantes ne sont pas identifiées, le dataset Kharon fournit une telle information. Comme reporté en table 3.4, excepté sur deux *malware* où l'analyse du bytecode lève une exception ou fait un faux positif, la méthode ayant le score le plus haut correspond à chaque fois à une partie du code malveillante du *malware*.

2.1.5 Reconstruction du graphe de flot de contrôle (CFG)

Pour attaquer le contrôle de flot du *malware* il est nécessaire de construire le CFG du *malware*. La difficulté de ce calcul réside dans l'architecture des applications Android. En effet, le flot d'exécution est contrôlé par la machine virtuelle Dalvik (ou son pendant lorsque l'application est compilée par Art) qui exécute différents morceaux applicatifs dans le thread principal. Par exemple, lorsqu'un évènement graphique survient (un clic de bouton), la machine virtuelle exécute la *callback* associée. Les appels de procédure peuvent donc être réalisés depuis la machine virtuelle, ce qui les rend difficiles à voir lorsqu'on analyse statiquement le code de l'application. Certains travaux se sont penchés sur l'analyse de ces appels implicites au framework Android [111], et nous évoquons ces pistes dans les perspectives de ce chapitre.

Pour résoudre partiellement ce problème, nous avons réalisé la détection des appels explicites (les appels de méthodes) et de certains appels implicites qui sont très bien documentés [133] et utilisés fréquemment par les développeurs d'applications et donc de *malware*. Les appels implicites pris en compte sont les suivants :

- Démarrage d'une activité, par l'appel statique *startActivity ()* ;
- Démarrage d'un service, par l'appel statique *startService ()* ;
- Liaison à un service, par l'appel statique à *bindService ()* ;
- Les appels provoqués par la navigation graphique de l'utilisateur dans l'application.

Par exemple, la figure 3.1 montre un CFG partiel décomposé au niveau de l'instruction du bytecode et com-

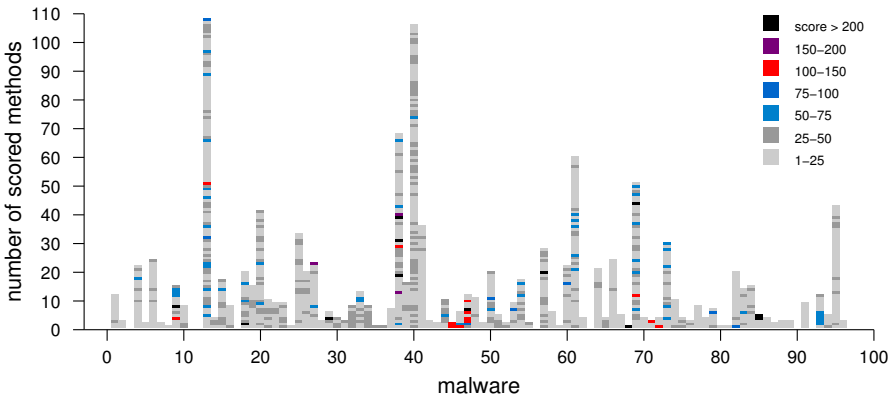


FIGURE 3.2 – Score des méthodes ciblées pour chaque malware

portant deux arcs représentant un flot implicite. Le service est démarré par un appel à *startService(r2)*. Cet appel provoque l'exécution implicite de la méthode *onCreate()*, puis dans un second temps l'exécution dans un thread indépendant de la méthode *onStartCommand()*. Dans cet exemple, le code malveillant d'envoi de SMS est localisé dans cette dernière méthode. Pour les appels implicites liés à la navigation graphique, ceux-ci sont générés par le framework Android lui-même lorsque l'utilisateur interagit avec le téléphone. Les méthodes correspondants à des évènements graphiques sont donc d'autres points d'entrée du programme qu'il convient de prendre en compte. Le travail sur la prise en compte de l'ensemble des appels implicites se poursuit actuellement grâce au travail de thèse de Mourad Leslous.

2.1.6 Résultats expérimentaux

L'approche proposée a été évaluée sur un dataset de 100 *malware* pris au hasard dans le *genome project* [183]. En figure 3.2, le scoring de chaque *malware* est donné par un empilement en colonne du score de chacune de ces méthodes. Certains malware peuvent comporter jusqu'à 110 méthodes ayant un score, mais en moyenne, 12 méthodes seulement obtiennent un score. Pour chaque malware, nous avons sélectionné la méthode ayant le score le plus haut et qui constitue la cible d'exécution. Si cette méthode est exécutée, nous considérons que nous avons réussi à exécuter le malware, sinon c'est un échec. La figure 3.3 présente le type des appels détectés dans les classes de score de la table 3.4 et donne ainsi une visualisation spatiale des "gènes" de chacun des 100 *malware*. On retrouve le plus souvent des appels à l'API de téléphonie (lecture de l'IMEI) et de l'envoi de SMS. Les appels au chargement dynamique de code sont très rares.

Pour ce dataset, nous avons utilisé trois méthodes d'analyse dynamique :

1. Monkey [132] : l'outil fourni par Google qui réalise des interactions graphiques aléatoires ;
2. GroddDroid [84] : le gorille aux pouvoirs psychiques qui réalise un parcours exhaustif de l'interface graphique ;
3. GroddDroid + ForceCFI : le gorille utilisant ses pouvoirs psychiques pour forcer le flot de contrôle afin d'exécuter le code malveillant.

Les résultats comparatifs obtenus sont donnés en table 3.5. On remarque que le taux de couverture des méthodes et des différentes branches est faible, mais que le code malveillant ciblé est plus haut : environ 20% du temps avec le Monkey. En utilisant GroddDroid, on obtient 4% de code malveillant exécuté en plus et en forçant le flot, encore 4% supplémentaires. Le tour de force réside dans le rapport entre l'augmentation du code malveillant exécuté et le nombre de branches du CFG supplémentaires : en exécutant seulement 0.41% de branches supplémentaires, on gagne 4% de couverture de code malveillant.

Néanmoins, la couverture globale de 28% reste faible et nous pensons qu'il est largement possible d'améliorer ce résultat. En effet, il est illusoire de réussir à obtenir une couverture de 100% : de nombreux malware ne sont plus fonctionnels, soit parce qu'ils sont trop vieux ou ne l'ont jamais été. Pour les *malware* contrôlés par l'attaquant, ceux-ci utilisent un serveur distant qui peut ne plus être en ligne. D'autre part, seuls quelques flots implicites sont pris en compte pour l'instant, ce qui peut empêcher un calcul de chemin correct dans le CFG.

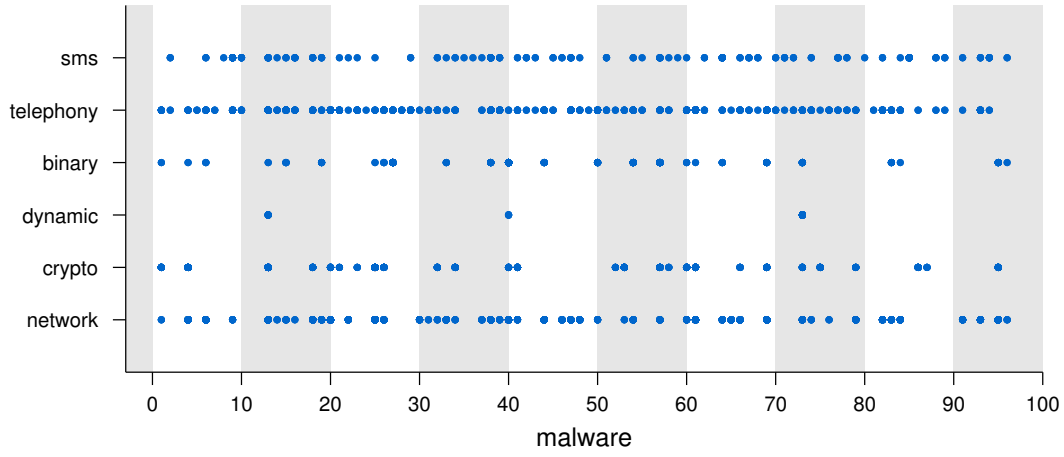


FIGURE 3.3 – Génome du dataset avec les classes de la table 3.2

TABLE 3.5 – Comparaison des performances d'exécution des *malware*

	Couverture		
Approches	Méthodes	Branches	Méthodes scorées
Monkey	14.99 %	9.35 %	20 %
GroddDroid	16.31 %	10.07 %	24 %
GroddDroid + ForceCFI	16.50 %	10.48 %	28 %

2.2 Attaques contre les codes de carte à puce

Le deuxième cas d'usage que nous abordons dans cette section concerne les attaques physiques contre les cartes à puce. Nous nous focalisons sur des attaques réalisées par des moyens physiques comme des perturbations de l'horloge [96], du courant électrique [90], du champ magnétique [120] pendant l'exécution de programmes natifs. Dans la littérature [141], on trouve notamment les attaques utilisant des impulsions laser [166] qui permettent de contrôler finement la position de l'attaque sur le circuit de la carte et le moment où la perturbation est effectuée. La précision temporelle de l'attaque est intéressante pour se synchroniser sur des algorithmes de cryptographie et pour réaliser plusieurs attaques en séquences (plusieurs tirs laser [175]) afin de synchroniser chaque perturbation avec le code qui s'exécute, par exemple lorsqu'on considère une boucle dans un algorithme cryptographique. Combinées avec l'observation des canaux auxiliaires et des techniques de cryptanalyse [135, 129], ces attaques peuvent devenir très dangereuses car elles permettent, par exemple, d'extraire des secrets cryptographiques.

2.2.1 Conséquences des attaques physiques sur les codes de carte à puce

Les conséquences d'une attaque physique sur l'exécution d'un programme sont multiples. Une attaque peut corrompre l'état d'un registre [99], l'état mémoire de la RAM ou de l'EEPROM [166], les données transmises sur le bus de données ou de code [98]. Si l'on considère l'impact de ces attaques à haut niveau, les critères communs définissent les impacts principaux en trois catégories [80] : la modification d'une variable en mémoire ; l'altération de la qualité d'un générateur de nombre aléatoire ; la modification du flot de contrôle. Le lien entre l'effet à bas niveau et la conséquence au niveau du programme exécuté n'est pas évident. Nous avons montré ce lien dans [102] et nous donnons les éléments principaux de réflexion ci-dessous.

Une attaque physique peut impacter un registre ou une valeur de la RAM ou de l'EEPROM. La valeur du registre ou de la variable peut aussi être impactée lors d'un transfert sur le bus de donnée. Nous dénommons une telle attaque, une attaque de donnée. De manière complémentaire, nous dénommons une attaque de code une attaque physique qui modifie le code exécuté, par exemple lorsque celle-ci impacte le bus d'instructions.

Une attaque de donnée qui impacte un registre ou une valeur de la RAM va impacter soit la valeur d'une variable du programme, soit le flot du programme si le registre ou la variable sont utilisés ultérieurement pour

```
char example(char u)
{
  char res, b;
  c = u + 5;
  b = c < 10;
  if (b){
   res = c + 1;
  }
  else{
   res = 0;
  }
  return res;
}
```

```
_example:
   mov   r2,dpl    // load the parameter in r2
   mov   a,#0x05   // put 5 into a
   add   a,r2      // compute u + 5 in a
   mov   _c,a      // store c into RAM from a
   clr   c         // clear the carry
   subb  a,#0x0A    // computes b i.e. c-10
   jnc   00102$    // jumps to 102
                   // if carry is not set (else)
/* @ATTACK_ADDR1 */
   mov   a,_c      // load c into a
   inc   a         // a++ i.e. c + 1
   mov   r2,a      // r2 stores a (res = c + 1)
   sjmp  00103$    // jump over else
00102$:
   mov   r2,#0x00  // r2 stores 0 (res = 0)
00103$:
   mov   dpl,r2     // push r2 on the stack
   ret              // returns
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

FIGURE 3.4 – Exemple de code C et le code assembleur 8051C correspondant

modifier le flot, soit les deux. Une attaque de code impacte un *opcode* ou une opérande soit les deux. Quel que soit le jeu d'instructions considéré, une instruction va enregistrer un résultat dans un registre ou mettre à jour une valeur à une adresse mémoire ou impacter le flot de contrôle. Pour le cas particulier des registres, ceux-ci sont utilisés pour stocker des résultats intermédiaires, comme des adresses mémoire ou des valeurs manipulées par le programme. Ils sont utilisés plus loin par le programme (sinon ils sont inutiles). Lors de leur utilisation, ils mettent à jour une case mémoire de la RAM utilisée comme valeur du programme ou adresse de case mémoire, ou bien sont utilisés pour modifier le flot de contrôle. Ainsi, une attaque de code ou une attaque de donnée vont soit impacter le contrôle de flot ou bien la valeur d'une variable ou bien les deux.

Par exemple, dans la figure 3.4, un effet sur la valeur de *c* peut être obtenu si l'on modifie la valeur du registre *r2*. De même, si l'instruction *mov a,#0x05* de la ligne 3 est remplacée par une instruction *NOP*, alors l'opération *add a, r2* sera impactée. On a donc de multiple façons d'impacter *c*, ce qui revient d'un point de vue du programme source à faire $c = u + ?$. D'un point de vue du flot de contrôle, plusieurs attaques peuvent impacter la construction *it−then−else* de la figure 3.4. Par exemple, l'instruction *subb* calcule le résultat de la variable *b*, en ligne 7, i.e. *c<10*. Ce registre de statut est utilisé ensuite en ligne 8. Ainsi, si une attaque remplace l'instruction *subb* par un *NOP* (ou toute autre instruction n'affectant pas le registre de retenue), le flot de contrôle est forcé pour prendre le saut de la ligne 8.

Etant donné que deux effets différents sont possibles lorsqu'une attaque a lieu (mauvaise valeur d'une variable ou flot de contrôle altéré), les contremesures à élaborer sont de natures différentes. Dans un cas, il faut s'assurer que les valeurs des variables sont correctement calculées alors que dans l'autre cas, il faut s'assurer que le *program counter* ne dévie pas du flot possible et reste en accord avec les variables contrôlant le flot. Nous nous sommes donc focalisés sur le flot de contrôle, notamment parce que le contrôle de la valeur des variables est plus difficile à réaliser avec un tir laser. En effet, la littérature montre que les attaques de code qui transforment des instructions en *NOP* donnent de meilleurs résultats pour l'attaquant que la mutation vers des *opcode* particuliers [169], ce qui revient à réaliser un saut inconditionnel sur le code. De plus, des travaux récents de Timmers *et al.* [170] montrent comment attaquer des instruction afin d'influer sur la valeur du *program counter* sur un processeur ARM. Cette hypothèse d'attaque est aussi considérée comme importante à traiter pour le *bytecode* Java dont le code est directement lu depuis la mémoire RAM [163, 172, 107].

2.2.2 Attaques physiques contre le flot de contrôle

Lorsqu'on s'intéresse aux attaques impactant le contrôle de flot, la difficulté technique de réalisation est liée aux paramètres expérimentaux de l'attaque : puissance et temps du laser sur le circuit, localisation spatiale. Lorsqu'on connait les paramètres optimaux pour s'attaquer à un branchement, Barbu *et al.* ont montré que l'on peut atteindre un taux de succès d'au moins 70% sur la reproductibilité de l'attaque [97]. En fait, certains programmes sont déjà vulnérables lorsqu'il existe des valeurs d'entrée qui font dévier le contrôle de flot vers des chemins que le programmeur n'avait pas imaginé. Certaines approches cherchent à déterminer ces valeurs d'entrées, comme Chucky [179] ou TDS [112], afin d'atteindre des portions de codes jugées dangereuses. Avec des attaques physiques, l'atteinte de ces portions de codes peut être plus aisée mais certaines valeurs de variables peuvent être corrompues puisque l'on viole la sémantique du programme.

```
237  void aes_addRoundKey_cpy(uint8_t *
         buf, uint8_t *key, uint8_t *
         cpk)
238  {
239    register uint8_t i = 16;
240
241    while (i--)
242    {
243      buf[i] ^= key[i];
244      cpk[i] = key[i];
245      cpk[16+i] = key[16+i];
246    }
247    ;
248  }
```

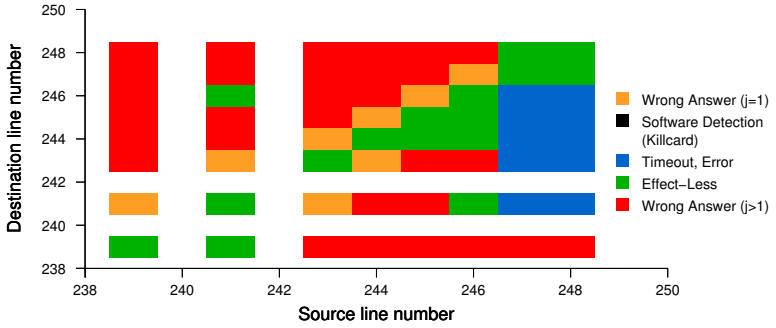


FIGURE 3.5 – Evaluation des conséquences des attaques par saut sur la fonction aes_addRoundKey_cpy

Afin de comprendre l'impact d'une attaque physique sur le contrôle de flot, il est naturel de remonter à la sémantique originelle du programme, c'est-à-dire son code source. Plusieurs approches complémentaires se sont donc intéressées à la simulation des attaques en considérant le code source de l'application et les conséquences d'une attaque physique qui perturberaient celui-ci. Potet *et al.* proposent ainsi dans [155] de simuler les fautes provoquant des inversions de test au niveau d'un bloc *if-then-else*. L'outil proposé, Lazart, permet ensuite de déterminer, à l'aide d'une méthode symbolique, une combinaison multiple de fautes qui permettent d'atteindre la cible de l'attaque, par exemple une portion de code critique. Dans [156], une approche orientée test injecte la faute à l'exécution à l'aide d'un débugueur. A l'intérieur d'une fonction, les auteurs proposent de sauter une ligne C du code originel afin de vérifier l'impact produit à l'exécution. De manière similaire à ces approches, nous avons proposé une simulation exhaustive des attaques du flot de contrôle en exploitant le modèle d'attaque du flot de contrôle au niveau C [102], comme décrit en section 1.3.

Des approches hybrides sont possibles. Dans [161], Riviere *et al.* proposent de simuler les attaques à haut niveau, dans le code source, mais réalisent la simulation à l'intérieur de la carte à puce. On est alors au plus près de la simulation d'une attaque physique réelle, mais l'approche requiert de disposer d'un mécanisme de débugging qui peut agir sur le programme dans la carte. De plus, les temps de simulation s'allongent à cause de la vitesse limitée de la carte.

2.2.3 Simulation d'attaques du flot de contrôle

Pour évaluer l'effet d'une attaque simulée au niveau du code source, nous utilisons un oracle dont le rôle est de distinguer une exécution fautée ayant réussi à donner un avantage à l'attaquant (fuite d'un secret, autorisation erronée, etc.). Nous avons repris la terminologie introduite par Goloubeva *et al.* [131] :

- *Wrong answer* (WA) : pendant l'exécution, un bénéfice est obtenu par l'attaquant ;
- *Effect-less* (EL) : le comportement de l'application reste inchangé ;
- *Error* or *Timeout* (TO) : le programme ne termine pas ou est terminé par le système d'exploitation ou bien se termine avec un message d'erreur, un signal (SIGSEGV, SIGBUS, ...) ;
- *Software detection* or *killcard* (SD) : une contremesure a détecté l'attaque et réveille une protection de sécurité pour traiter l'attaque, comme l'extinction de la carte ou sa destruction physique.

Un exemple de résultat est présenté en figure 3.5 pour la méthode *aes_addRoundKey_cpy_datu_heatmap* de l'algorithme AES. Pour la catégorie *Wrong answer*, nous avons choisi une surapproximation : dès que l'algorithme AES renvoie une valeur différente de la valeur originale, nous supposons qu'un gain est acquis par l'attaquant. Ainsi, dans la figure 3.5 de nombreuses attaques réussissent, surtout si le saut est un saut avant. Les sauts arrière provoquent quant à eux plus d'erreurs. Une évaluation plus quantitative de la simulation de ces attaques sur AES est donnée plus tard en section 3.3 en comparant leurs effets en présence des contremesures proposées.

Les deux cas d'étude précédents mettent en lumière toute l'importance de considérer les attaques contre le flot de contrôle. Dans la suite de ce chapitre, nous nous intéressons aux contremesures qui permettent de sécuriser le flot de contrôle travaillant au niveau du code source. Nous proposons en particulier une méthode de sécurisation adaptée pour le cas des attaques physiques qui ciblent les cartes à puce.

3 Contremesures pour la sécurité du flot de contrôle

Les attaques et les simulateurs d'attaques présentés précédemment sont utilisés pour mettre à l'épreuve les méthodes de sécurisation proposées par leurs auteurs. Dans cette section, nous passons en revue les principales techniques pour sécuriser le flot de contrôle et nous portons une attention particulière aux techniques dédiées au sauts provoqués par des attaques physiques contre les cartes à puces. Puis, nous proposons une approche pour sécuriser le flot d'un code source C embarqué dans une carte à puce et nous donnons une évaluation expérimentale de celle-ci [142].

3.1 Sécurisation du flot de contrôle

3.1.1 Intégrité du flot contrôle

La sécurisation du contrôle de flot est une vieille thématique de la littérature, étudiée dans un cadre plus large lorsqu'une erreur dans l'interprétation du code survient. Dans de nombreux systèmes industriels, des fautes d'interprétation du code ou de la réception de celui-ci pourrait induire de graves accidents. En 1970, la technique du *monoprocesseur codé* est proposée par Daniel Dollé pour corriger une erreur à l'exécution : N bits sur 32 sont utilisés en redondance pour corriger une possible erreur. Utilisée dans le développement du RER A de Paris, il décode les informations de vitesse et de position, avant d'envoyer ces informations au processeur de calcul [139]. Cette seconde unité de calcul utilise une autre technique pour se prémunir d'une défaillance : le code est exécuté deux fois, ce qui impacte les performances. Ces techniques peuvent être utilisées pour garantir la bonne exécution du contrôle de flot, face à des défaillances aléatoires. Cependant, pour des attaques physiques contrôlées, elles deviennent inefficaces.

En 2005, Abadi *et al.* [82, 83] introduisent l'intégrité du contrôle de flot comme une technique de sécurisation à part entière. Les auteurs définissent l'intégrité du flot de contrôle comme la garantie de l'exécution du flot vis-à-vis d'une politique de sécurité qui exprime comment le flot de contrôle doit s'exécuter dans les conditions nominales. Dans ce travail, Abadi *et al.* proposent une analyse binaire statique pour extraire la politique de flot. Puis, le programme est instrumenté pour introduire des identifiants et des vérifications lorsque le flot de contrôle saute d'une source à une destination. Ainsi, avant le saut, le programme vérifie que l'identifiant de la destination est correct avant d'effectuer le saut depuis la source. Avec une telle technique, il n'y a plus besoin d'avoir un processeur particulier ou des nouvelles instructions de contrôle. L'*overhead* est très variable, proche de 0% pour bzip2 qui est très calculatoire mais de 45% pour crafty, un programme de SPEC CPU 2000. Suite à ce travail d'Abadi *et al.*, de nouvelles techniques d'attaques utilisant le flot de contrôle ont été proposées : *Return-Oriented Programming attacks* [165] (ROP) et *Jump-Oriented Programming attacks* (JOP) [152, 92]. Ces attaques, qui réutilisent des portions de code déjà existant pour construire le code malveillant, peuvent être traitées par la méthode proposée par Abadi *et al.* Bletsch *et al.* proposent en 2011 une méthode dédiée pour les attaques par réutilisation de code [105] : avant un transfert du flot de contrôle, un morceau de code gèle une valeur en mémoire, la dégèle et la vérifie après le saut. Les performances sont légèrement meilleures que les techniques antérieures. Cette technique ressemble à la technique des *cookies* ou *canaries* qui permet d'éviter l'écrasement d'une partie de la pile qui contient une adresse de retour [117]. En 2007, Petroni *et al.* [153] introduit une variante de l'intégrité du contrôle de flot comme défini par Abadi *et al.* La vérification de l'intégrité du flot est faite au travers de la vérification d'un état, lorsque plusieurs étapes d'exécution se sont écoulées et que le système entre dans un nouvel état. Évidemment, si l'attaquant réussit à réaliser une modification, puis à l'effacer, ce contrôle d'état ne détecte pas l'attaque.

D'autres approches supposent que le matériel peut être modifié [125, 113, 118, 177, 116] ou repose sur des capacités spéciales du matériel [126]. Ces solutions permettent de réduire l'*overhead* de la surveillance du flot. En s'inspirant de ces travaux, des solutions purement logicielles ont vu le jour [151, 114, 130, 131]. Oh *et al.* [151] réalisent des vérification de signatures aux destinations des *basic block* à l'aide d'un *watchdog thread* supplémentaire. La vérification consiste à calculer la signature du *basic block* afin de garantir son intégrité. L'approche YACCA [130, 131] détecte les attaques à l'aide d'instructions supplémentaires pour enregistrer des valeurs à la frontière des *basic block* et vérifier leur intégrité lorsque l'instruction *test* est rencontrée.

Cependant, l'ensemble de ces approches se concentre sur la sécurisation du contrôle de flot originel, c'est-à-dire ne suppose pas que le modèle d'attaque puisse générer une rupture dans le flot au milieu d'un *basic block*. De plus, lorsqu'un test dans une construction *if-then-else* est effectué, ces approches garantissent que le saut ira soit dans la branche *then* soit dans la branche *else* mais pas que le saut sera cohérent avec la valeur du test.

3.1.2 Intégrité du flot de contrôle au sein d'un basic block

Pour gérer les ruptures de flot au sein d'un *basic block*, les approches de la littérature implémentent des contre-mesures au niveau du code assembleur du programme originel [101, 100, 149]. De manière synthétique, les instructions du code sont dupliquées afin de garantir que l'une des deux instructions est réellement faite ou bien de détecter que l'une des deux exécutions qui sont normalement identiques est différente. Néanmoins, ces approches nécessitent d'analyser finement le code assembleur et de disposer de registres supplémentaires disponibles [100, 149]. Barry *et al.* proposent une passe de compilation pour résoudre ce problème [101].

Ces techniques sont principalement utilisées pour déjouer des sauts de courte portée, qui sautent quelques instructions assembleurs. Elles pourraient être utilisées pour des sauts plus grands à l'intérieur d'un *basic block* mais auraient un *overhead* beaucoup plus important.

3.1.3 Intégrité du flot de contrôle pour la carte à puce

L'ensemble des approches précédentes ne peuvent s'appliquer directement à du logiciel embarqué dans une carte à puce. Il est tout d'abord difficile de modifier le matériel sous-jacent, à moins de revoir complètement l'architecture du processeur utilisé. D'autre part, la modification du code assemblé n'est pas une chose aisée dans le processus de développement : cela rend plus difficile les phases de test car l'environnement de développement ne peut plus être en phase avec le débugueur si le binaire est modifié.

La duplication du code est une solution coûteuse en terme de performance. Elle est particulièrement efficace pour les attaques sautant peu d'instructions [149] mais serait lourde à mettre en œuvre pour de grands sauts.

Enfin, ce ne sont pas les sauts d'une fonction vers du code d'une autre fonction qui sont les plus dangereux (et qui sont utilisés pour les ROP attaques) mais plutôt les sauts à l'intérieur d'une fonction qui désactivent une portion de code, par exemple un round d'AES. Dès lors, les approches travaillant sur les bornes de *basic block* où les appels explicites de fonctions deviennent inutiles.

3.2 Contremesures pour l'intégrité du flot de contrôle de code C

Le principe de sécurisation proposée dans cette section repose sur l'intégrité d'un compteur dédié à chaque bloc d'instructions C du code source. Le compteur est incrémenté entre chaque instruction C du code source originel. Avant chaque incrémentation il est vérifié et en cas d'anomalie, la fonction *killcard()* est symboliquement appelée pour représenter la fin du programme, par exemple la désactivation de la carte à puce pour une carte bancaire. La difficulté de cette approche est de gérer les structures conditionnelles et les boucles.

3.2.1 Sécurisation d'instructions séquentielles et d'appels de fonctions

Pour implémenter une contremesure basée sur ce principe pour un code linéaire, la solution est triviale. Afin de bien vérifier aux bornes d'un appel de fonction que le transfert de flot s'effectue correctement, nous utilisons deux compteurs différents et les vérifications et déclarations sont imbriquées. La figure 3.6 présente le principe de sécurisation avec deux fonctions f et g et leurs compteurs respectifs *cnt_f* et *cnt_g*. Les macros utilisées sont définies dans la figure 3.7. La vérification de l'intégrité de ces deux compteurs est implémentée à l'aide de la macro *CHECK_INCR_FUNC* qui vérifie leur cohérence au retour de la fonction. On retrouve dans cette vérification quelque chose de similaire aux approches de la littérature. Néanmoins, la différence réside dans les vérifications successives à l'intérieur de la fonction g, idée déjà évoquée dans un papier non publié et antérieur [85].

Les valeurs des compteurs sont statiques et calculées à l'avance, par analyse du code source C. Cela suppose que le code source est bien formé, c'est-à-dire qu'il n'y a pas de rupture de séquence volontaire, comme par exemple une instruction *goto*. La difficulté réside alors dans l'adaptation de ce principe de sécurisation simple pour du code linéaire et des appels de fonctions à des imbrications plus complexes de structures conditionnelles et itératives.

3.2.2 Sécurisation de structures conditionnelles et de boucles

La sécurisation de structures conditionnelles et de boucles est représentée en figures 3.8 et 3.9. Le nombre de compteurs est plus important car il en faut un par sous bloc : *cnt_then* pour le bloc *then*, *cnt_else* pour le bloc *else* et *cnt_while* pour le bloc *while*, en plus de *cnt* pour le corps de la fonction. Pour vérifier l'intégrité d'une branche de la conditionnelle ou d'un tour de boucle, on utilise une variable b qui stocke le résultat de la condition. Ainsi, il est possible, en sortant d'une des deux branches de la conditionnelle, ou en sortant de la boucle ou bien

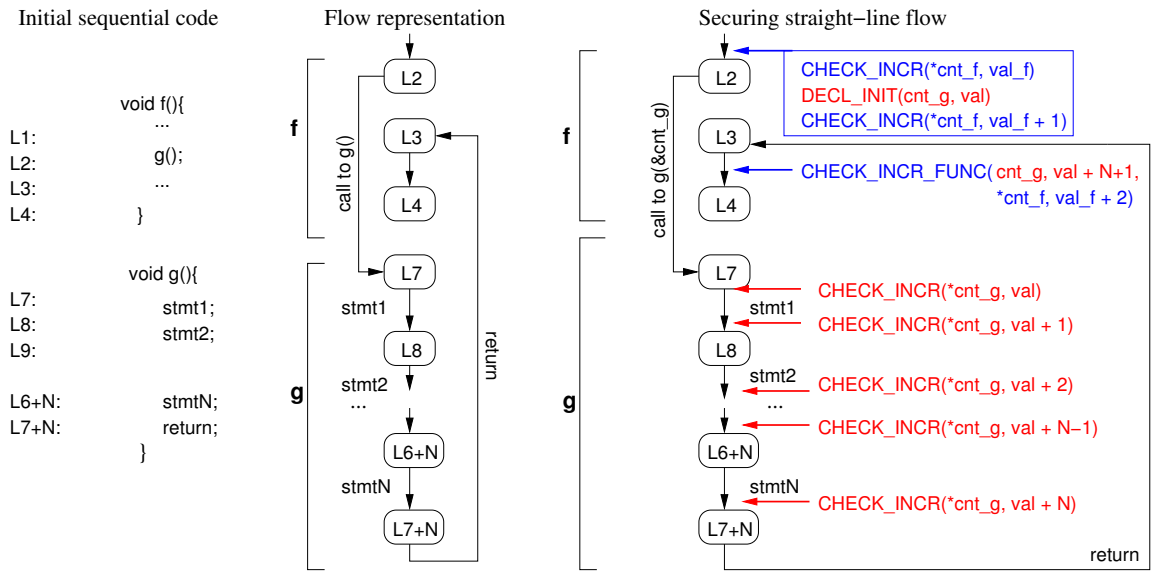


FIGURE 3.6 – Sécurisation d'instructions séquentielles et d'appels de fonctions

```
// STRAIGHT−LINE FLOW
#define DECL_INIT(cnt,val) unsigned short cnt = val;
#define CHECK_INCR(cnt,val) cnt = (cnt == val ? cnt + 1 : killcard ());

// AFTER A FUNCTION CALL
#define CHECK_INCR_FUNC(cnt1, x1, cnt2, x2) cnt1 = ((cnt1 == x1) && (cnt2 == x2) ? cnt1 + 1 : killcard ());

// IF
#define CHECK_END_IF_ELSE(cnt_then, cnt_else, b, x, y) \
    if ( ! ( (cnt_then == x && cnt_else == 0 && b) || (cnt_else == y && cnt_then == 0 && !b) ) ) killcard ();
#define CHECK_END_IF(cnt_then, b, x) \
    if (! ( (cnt_then == x && b) || (cnt_then == 0 && !b) ) ) killcard ();

// WHILE
#define CHECK_INCR_COND(b, cnt, val, cond) (b = (((cnt)++ != val) ? killcard () : cond))
#define CHECK_END_LOOP(cnt_loop, b, val) if ( ! (cnt_loop == val && !b) ) killcard ();
#define CHECK_LOOP_INCR(cnt, val, b) cnt = (b ? (cnt + 1) : 0);
#define RESET_CNT(cnt_while, val) cnt_while = !(cnt_while == 0 || cnt_while == val) ? killcard () : 0;

// For the late detection
#define INCR(cnt) cnt = cnt + 1;
#define INCR_COND(b, cnt, cond) (++cnt && (b = (cond)))
```

FIGURE 3.7 – Macros utilisées pour la sécurisation du flot de contrôle

en rebouclant une nouvelle fois, de tester la valeur de b qui doit être cohérent avec les autres compteurs. Par exemple, si b est vrai, alors *cnt_then* doit valoir 4 et *cnt_else* 0 (selon l'exemple de la figure 3.8), puisque la valeur de b signifie que l'on est passé dans la branche *then*. On notera l'utilisation de la macro *RESET_CNT* ainsi que le remplacement de l'instruction *while* par une combinaison *if-goto*, que le compilateur aurait fait de toute manière.

3.2.3 Autres instructions

Les autres constructions du C, comme le *switch case* et le *break* peuvent se réécrire sous forme de *if-then-else*. Ils peuvent donc se traiter de manière similaire. L'utilisation de multiple *return* peut aussi être traité. Cela nécessiterait d'adapter la macro *CHECK_INCR_FUNC* pour inclure plusieurs valeurs possibles lors du retour de fonction ainsi qu'une variable pour indiquer quel *return* a été utilisé.

A l'inverse, l'instruction *goto* est difficile à traiter, surtout si ce *goto* change de niveau de bloc par exemple en sortant d'un block *then*. Cependant, un bon programme n'étant pas censé comporter une telle instruction, ce point n'a pas été traité.

Enfin, l'utilisation de pointeurs de fonction est aussi une difficulté non traitée par les contremesures proposées. C'est pourtant un cas classique pour les codes de cartes à puce qui sélectionnent le code fonctionnel

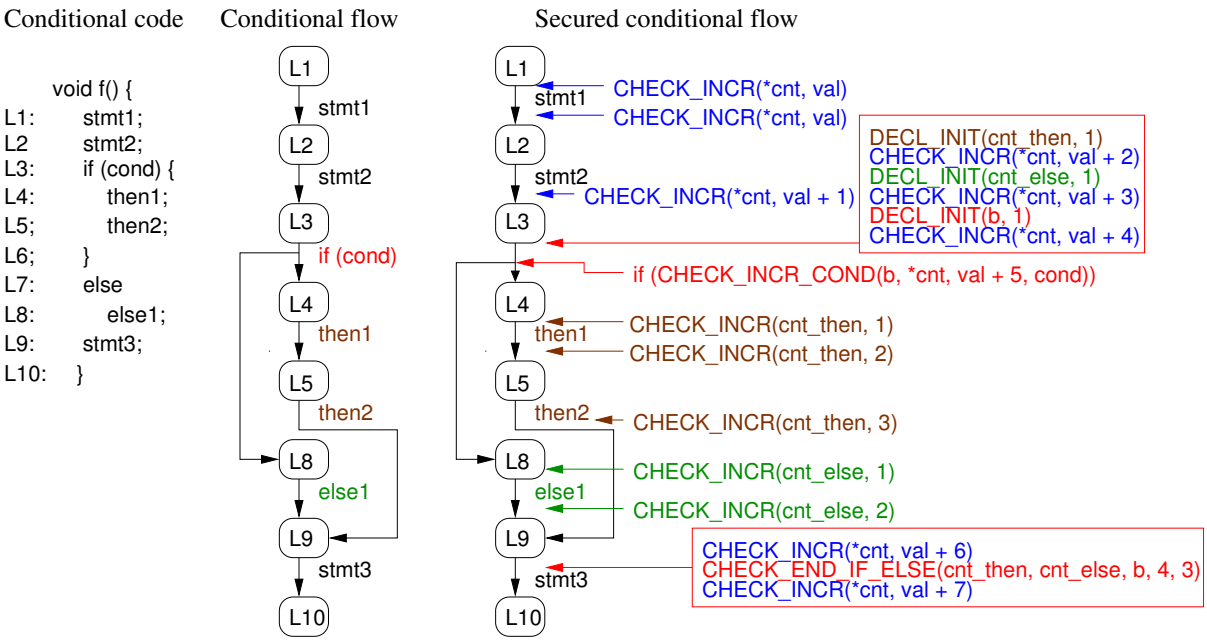


FIGURE 3.8 – Sécurisation d'un bloc conditionnel

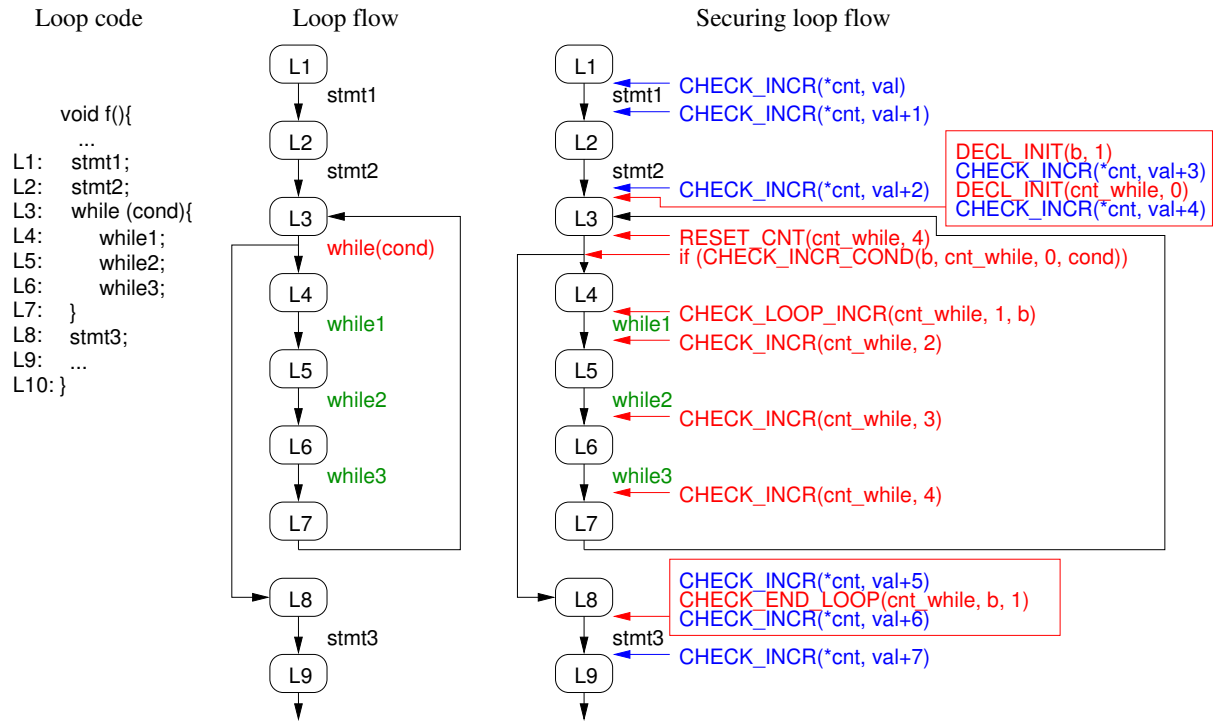


FIGURE 3.9 – Sécurisation d'une boucle

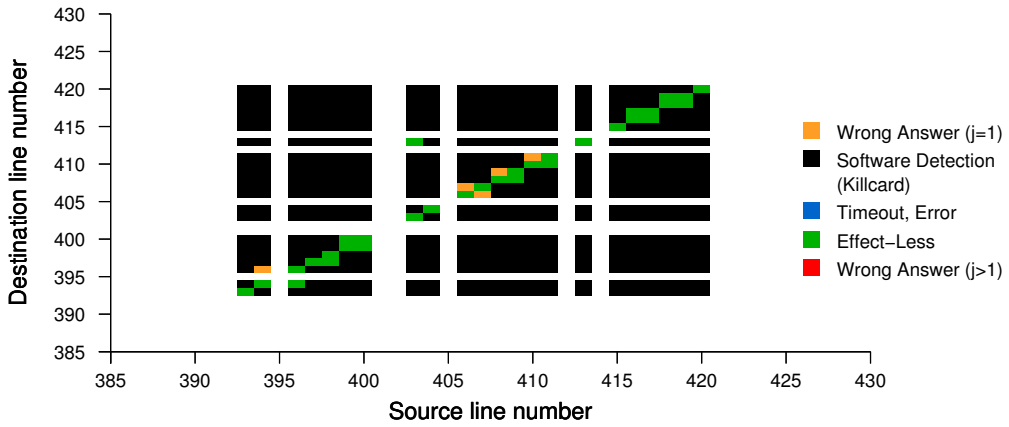


FIGURE 3.10 – Evaluation des conséquences des attaques par saut sur la fonction sécurisée d'aes_addRoundKey_cpy

à lancer en fonction des premières interactions avec le terminal. Comme expliqué par Arthur *et al.* [92], il est toujours possible de réécrire le code pour traduire les appels de pointeurs de fonctions en un nombre d'appels explicitement écrits dans le code, si l'on connait l'ensemble des appels possibles à l'avance.

3.2.4 Détection précoce ou tardive

Nous avons aussi proposé une détection tardive pour alléger le coût de ces contremesures. En effet, au lieu de vérifier la valeur du compteur à chaque incrémentation lors de l'appel à *CHECK_INCR*, on peut déléguer la vérification du compteur aux bornes du *basic block* lors de l'appel à *CHECK_INCR_FUNC* ou *CHECK_END_IF_ELSE* ou *CHECK_END_LOOP*. Dans ce cas, au lieu d'appeler la macro *CHECK_INCR* (resp. *CHECK_INCR_COND*) on utilise *INCR* (resp. *INCR_COND*). La détection du saut sera donc plus tardive mais les performances sont meilleures, comme présenté plus loin en section 3.3.2.

3.3 Evaluation des contremesures

Pour tester l'approche proposée nous avons automatisé l'injection des contremesures sur un code C standard. En reprenant le simulateur de fautes et la classification des conséquences des attaques présentés en section 2.2.3, il devient alors possible de comparer l'effet des attaques entre la version avec et sans contremesures. La figure 3.10 présente la fonction *aes_addRoundKey_cpy* après sécurisation. On constate que l'intégralité des sauts de plus de 2 lignes sont désormais classés dans la catégorie *Software detection*. Quelques sauts sont sans effets et quelques rares sauts d'une ligne unique de code C provoquent un effet de bord. Ces résultats, très intéressants sur cette fonction sont confirmés quantitativement dans les expériences qui suivent.

3.3.1 Efficacité

La table 3.6 présente les résultats de simulation de fautes sur trois codes AES [143], SHA [134] et Blowfish [134]. Pour chaque programme, nous classifions l'impact des fautes simulées suivant la classification introduite en section 2.2.3. Pour les fautes *Wrong answer*, nous distinguons celles qui provoquent des sauts de plus d'une ligne C ($size > 1$) et celles qui sautent exactement 1 ligne C ($size = 1$). Pour chaque programme, la première ligne donne le résultat de classification sans contremesure. Pour AES par exemple, 7 835 attaques sautant au moins 2 lignes perturbent la sortie, tandis que 17 323 attaques sont sans effet. Puis, nous donnons les résultats avec contremesures précoces (+CMED pour CounterMeasures with Early Detection) ou tardives (+CMED pour CounterMeasures with Deferred Detection).

Comme attendu le nombre d'attaques sautant au moins 2 lignes (*Wrong answer* $size > 1$) devient nul. Sur les attaques sautant 1 ligne, le nombre diminue et la proportion sur le nombre d'attaques total devient très faible, par exemple 0.2% pour AES. Nous avons aussi testé l'effet des contremesures lorsque l'on réalise des attaques au

TABLE 3.6 – Effets des attaques par saut pour la version originale, avec détection précoce (+CMED) et détection tardive (+CMDD)

	WA SIZE > 1		WA SIZE = 1		EL		SD: KILLCARD		TO		TOTAL
C JUMP ATTACKS	Attacking all functions at C level for all transient rounds										
AES	7835	29 %	1104	4.2 %	17 323	65 %	0		108	0.4 %	26 370
AES + CMED	0		603	0.2 %	18 339	6.7 %	255 614	93 %	1	0.0 %	274 557
AES + CMDD	0		603	0.2 %	37 334	13 %	236 619	86 %	1	0.0 %	274 557
SHA	32 811	75 %	1527	3.5 %	8531	19 %	0		405	0.9 %	43 274
SHA + CMED	0		1143	0.3 %	5015	1.3 %	368 941	98 %	290	0.1 %	375 389
SHA + CMDD	0		1144	0.3 %	5015	1.3 %	368 736	98 %	494	0.1 %	375 389
Blowfish	70 318	32 %	3553	1.7 %	134 360	62 %	0		5490	2.6 %	213 721
Blowfish + CMED	0		2468	0.2 %	312 745	25 %	887 364	73 %	6852	0.6 %	1 209 429
Blowfish + CMDD	0		2467	0.2 %	312 745	25 %	869 690	71 %	24 527	2.0 %	1 209 429

TABLE 3.7 – Attaques par saut au niveau assembleur pour la version originale, avec détection précoce (+CMED) et détection tardive (+CMDD)

	WA SIZE > 1		WA SIZE = 1		EL		SD: KILLCARD		TO		TOTAL
ASM JUMP ATTACKS	Attacking function at ASM level for the first transient round										
aes_encrypt	1566	82.8 %	36	1.9 %	179	9.5 %			111	5.9 %	1892
aes_encrypt + CMED	482	0.3 %	24	0.01 %	19 895	11.1 %	155 639	87.3 %	2466	1.3 %	178 506
aes_encrypt + CMDD	355	0.4 %	24	0.02 %	6088	6.5 %	82 902	88.8 %	3961	4.2 %	93 330
aes_mixcolumn	2042	86.8 %	45	1.9 %	200	8.5 %			65	2.8 %	2352
aes_mixcolumn + CMED	834	1.3 %	51	0.08 %	7592	11.6 %	56 547	86.6 %	256	0.4 %	65 280
aes_mixcolumn + CMDD	615	1.9 %	29	0.09 %	7329	23 %	22 012	69.1 %	1877	5.9 %	31 862

niveau de l'instruction assembleur, c'est-à-dire une granularité plus fine que l'instruction C. La table 3.7 présente ces résultats pour 2 fonctions d'AES. On constate que l'ensemble des attaques possibles ne peut bien sûr pas être réduit à zéro mais baisse significativement. La proportion des attaques qui réussissent sur la fonction *aes_encrypt* atteint 0.4% lorsque la détection est précoce, contre 82% sans contremesures.

3.3.2 Overhead

Les overheads de nos contremesures sont reportés en table 3.8 pour nos 3 programmes de test compilés sur une architecture x86 et un processeur cortex-M3, plus proche de ce que l'on trouve dans une carte à puce. Le temps de simulation est aussi donné à titre indicatif.

Une détection précoce impacte considérablement les performances, par exemple jusqu'à $+115\%$ pour AES sur un x86 et $+219\%$ sur un ARM. Le processeur ARM est pénalisé par son incapacité à paralléliser les instructions et à réaliser de la prédiction de branchement. Lorsque la détection est tardive, le pire cas pour le processeur ARM est de $+110\%$, ce qui peut s'avérer acceptable si l'on active la contremesure que sur certaines portions de code qui sont critiques (par exemple, l'authentification).

TABLE 3.8 – Taille et overhead (ohd) pour la version originale, avec détection précoce (+CMED) et détection tardive (+CMDD)

	x86					ARM-V7M			
	SIMULATION TIME	SIZE BYTES	SIZE OHD	EXECUTION TIME TIME	EXECUTION TIME OHD	SIZE BYTES	SIZE OHD	EXECUTION TIME TIME	EXECUTION TIME OHD
AES	22m	18 016		0.77 ms		4224		62.18 ms	
AES + CMED	8h 40m	38 552	+113 %	1.83 ms	+138 %	13 472	+219 %	186.7 ms	+200 %
AES + CMDD	6h 05m	30 360	+68 %	1.39 ms	+81 %	9732	+110 %	131 ms	+110 %
SHA	58m	13 528		1.04 µs		3184		504.3 µs	
SHA + CMED	12h 34m	21 776	+60 %	2.10 µs	+102 %	7032	+121 %	851.36 µs	+69 %
SHA + CMDD	9h 37m	17 680	+30 %	1.79 µs	+72 %	5272	+66 %	730.4 µs	+45 %
Blowfish	4h 39m	30 352		0.67 µs		5546		4.55 ms	
Blowfish + CMED	2d 0h 44m	46 784	+54 %	1.05 µs	+56 %	14 668	+164 %	14.46 ms	+217 %
Blowfish + CMDD	1d 14h30	38 592	+32 %	0.89 µs	+41 %	10 820	+95 %	13.01 ms	+186 %

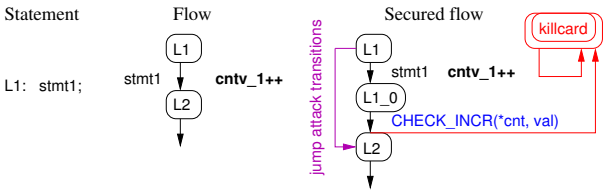


FIGURE 3.11 – Principes des systèmes de transitions $M(c)$ et $CM(c)$

3.4 Vérification formelle des contremesures

L'ajout de contremesures au code source ou bien au code compilé pourrait avoir des conséquences sur celui-ci. Il pourrait, d'une part, ne plus se dérouler comme initialement et d'autre part, manquer certaines attaques à cause d'une erreur de conception. Comme Abadi *et al.* [82], nous proposons dans cette partie une vérification formelle de nos contremesures. Pour l'instrumentation de binaire, Abadi *et al.* modélisent un programme formellement en représentant les instructions et en exprimant ce que les instructions modifient du point de vue de la mémoire, des registres, et du *program counter*. L'attaquant est alors modélisé par la possibilité de modifier arbitrairement les données de la mémoire et les registres. Enfin, l'étape de vérification consiste à évaluer si, quelles que soient les règles du modèle utilisé, le *program counter* passe d'un état à un autre en suivant une transition valide prévue dans le programme originel ou bien ne change pas malgré l'utilisation d'une règle d'attaque. Notre approche est similaire à celle d'Abadi *et al.* mais notre modélisation diffère pour l'attaquant car nous introduisons de nombreuses possibilités de sauts entre états pour représenter les attaques.

3.4.1 Systèmes de transitions pour modéliser un programme

Nous introduisons deux systèmes de transitions $M(c)$ et $CM(c)$ pour représenter le code source originel et avec contremesures. Chaque système de transition est caractérisé par $TS = \{S, T, S_0, S_f, L\}$ où S est l'ensemble des états, T l'ensemble des transitions $T : S \rightarrow S$, S_0 et S_f des sous-ensembles de S des états initiaux et finaux. Sur chaque transition entre états, nous introduisons un compteur d'exécution d'une d'instruction appelé plus simplement compteur d'instruction et noté $cntv\_\alpha i$, où α représente la fonction considérée. Ce compteur d'instruction permet de représenter l'exécution ou non de l'instruction *stmt_i* qui lui correspond. Dans la figure 3.11, ce compteur est nommé $cntv\_1$. Un état de S est défini, par le *program counter*, la valeur des compteurs $cntv\_\alpha i$ et les valeurs des variables permettant de calculer les conditions (*e.g.* les compteurs d'itérations ou les conditions pour les *if*). Enfin, L est l'ensemble des labels possibles pour le *program counter*, dans notre cas, les numéros de lignes du code source.

L'état initial correspond à un *program counter* pointant sur la première ligne du programme. Lorsqu'une transition de T a lieu, celle-ci correspond à l'exécution de l'instruction $stmt\_i$, pointée par le *program counter*. Dans ce cas, dans le modèle, le compteur d'instruction $cntv\_\alpha i$ est incrémenté pour représenter l'exécution de l'instruction. Cependant, la réalité de l'opération effectuée par l'instruction (opération arithmétique, changement d'un registre, etc.) est oublié car le système de transition ne cherche à représenter que l'exécution ou la non exécution de $stmt\_i$ et le déplacement du *program counter*. La modélisation d'une attaque du flot de contrôle sautant plus de 2 lignes de code source est alors simple : une attaque affecte directement le *program counter*. Un exemple d'une telle attaque sautant de la ligne L1 à la ligne L2 est représentée en violet sur la figure 3.11. Dans $CM(c)$, des transitions supplémentaires doivent être ajoutées à cause des contremesures. Comme montré en rouge sur la figure 3.11, la macro *CHECK_INCR* envoie dans un état spécial appelé "killcard" qui est un état puits pour le système.

3.4.2 Propriétés à vérifier sur $M(c)$ et $CM(c)$

La vérification de nos contremesures se base sur le principe de l'*equivalence-checking* : $M(c)$ est le modèle de référence auquel on compare le modèle $CM(c)$. Ainsi, nous utilisons un *model checker* pour construire le produit des deux modèles et nous lui demandons de vérifier des formules CTL représentant l'équivalence d'exécution, d'un point de vue du contrôle de flot, des deux modèles. Pour le cas du flot de contrôle linéaire et de l'appel de fonctions que nous traitons intégralement ici, quatre propriétés seront nécessaires. Nous donnons en figure 3.13 l'intégralité des modèles $M(c)$ et $CM(c)$ pour ce cas ainsi que les variables $cntv\_\alpha i$ utilisées. Les cas du *if-then-else* et du *while* sont donnés en annexe 2.

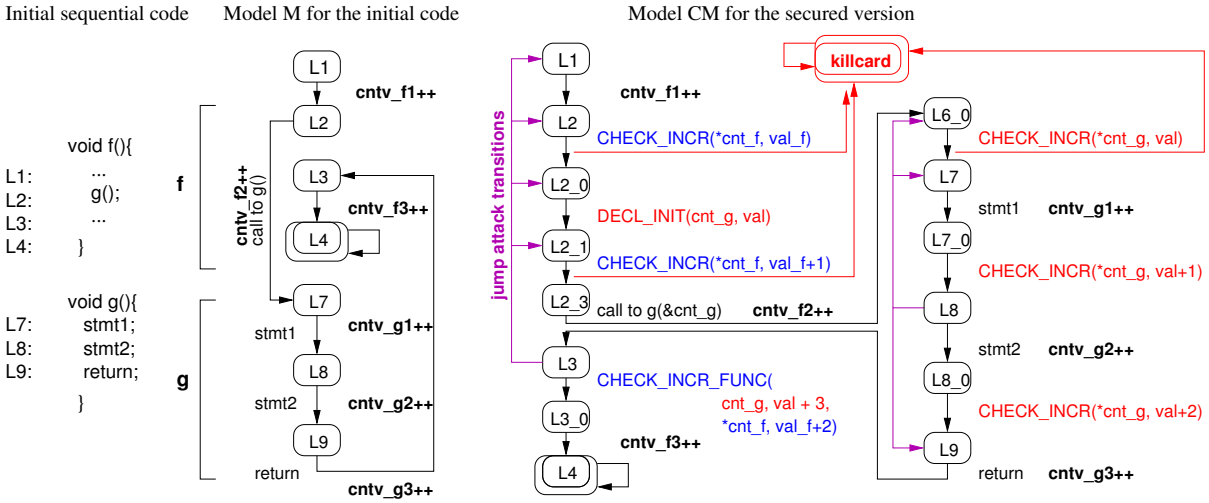


FIGURE 3.12 – Représentation compacte de TS pour le code linéaire et l'appel de fonction

```
; P1 : final state reachability in M and CM
AG(AF(M.pc = L4))
AG(AF(CM.pc = L4 + CM.pc = killcard))
; P2 : right statement execution counts in CM and M when execution reaches a correct final state
  in both models
AG( ((M.pc = L4) . (CM.pc = L4)) => (M.cntv_f1 = CM.cntv_f1) . (M.cntv_f2 = CM.cntv_f2)
                                  . (M.cntv_f3 = CM.cntv_f3) . (M.cntv_g1 = CM.cntv_g1)
                                  . (M.cntv_g2 = CM.cntv_g2) . (M.cntv_g3 = CM.cntv_g3))

; Macro Before:
Before(cntA, cntB) ≡ (cntA=cntB + cntA=cntB+1)

; P3 on M : right order of statement execution in M
AG( Before(M.cntv_f1, M.cntv_f2) . Before(M.cntv_f2, M.cntv_f3)
  . Before(M.cntv_g1, M.cntv_g2) . Before(M.cntv_g2, M.cntv_g3)
  . Before(M.cntv_f2, M.cntv_g1) . Before(M.cntv_g3, M.cntv_f3))

; P3' on CM : right order of statement execution in CM or attack detection
AG( (Before(CM.cntv_f1, CM.cntv_f2) . Before(CM.cntv_f2, CM.cntv_f3)
   . Before(CM.cntv_g1, CM.cntv_g2) . Before(CM.cntv_g2, CM.cntv_g3)
   . Before(CM.cntv_f2, CM.cntv_g1) . Before(CM.cntv_g3, CM.cntv_f3))
  + AF(CM.pc = killcard))
```

FIGURE 3.13 – Propriétés du *model checker* pour le code linéaire et l'appel de fonction

Les quatre propriétés sont données en figure 3.13.

P1 : P1 exprime le fait que le programme doit terminer dans un des états puits, ici L4, l'état normal de fin du programme et éventuellement killcard qui est aussi un état normal pour le modèle $M(c)$ (l'attaque a été capturée).

P2 : P2 exprime qu'à l'issue des exécutions des modèles dans l'état puits normal (L4), les compteurs d'instructions $cntv\_\alpha i$ doivent être identiques entre ceux du modèles originel $M(c)$ et ceux du modèle avec contremesure $CM(c)$. Ainsi, cela garantit que les instructions des deux programmes ont été exécutées le même nombre de fois.

P3 : P3 exprime l'ordre sur l'incrémentation des compteurs d'instructions $cntv\_\alpha i$ du modèle $M(c)$ à l'aide de la macro *Before*. Deux compteurs consécutifs $cntv\_\alpha i$ et $cntv\_\alpha i + 1$ doivent obligatoirement être égaux ou à une différence de 1 (i.e. $cntv\_\alpha i$ = $cntv\_\alpha i + 1$ + 1) quel que soit l'état considéré. Ainsi, cette propriété représente le fait que deux instructions successives ont été exécutées le même nombre de fois ou que dans cet état particulier du système, l'instruction i vient d'être exécutée alors que l'instruction $i + 1$ ne l'a pas encore été.

P3' : P3' est le corollaire de P3 pour le modèle $CM(c)$. Un cas particulier de cette propriété est le fait que si des compteurs successifs ne sont pas distants d'une valeur de 1, alors *killcard* doit être atteint dans un futur état.

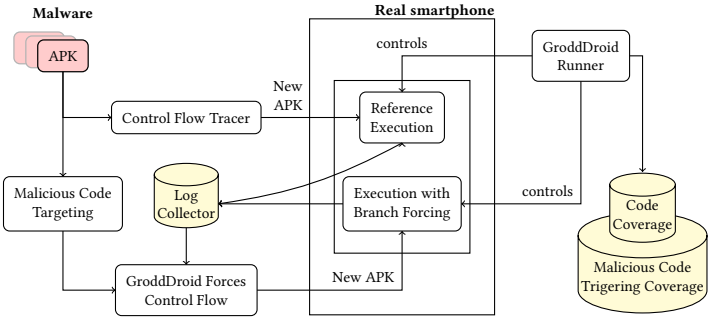


FIGURE 3.14 – *Framework* GroddDroid

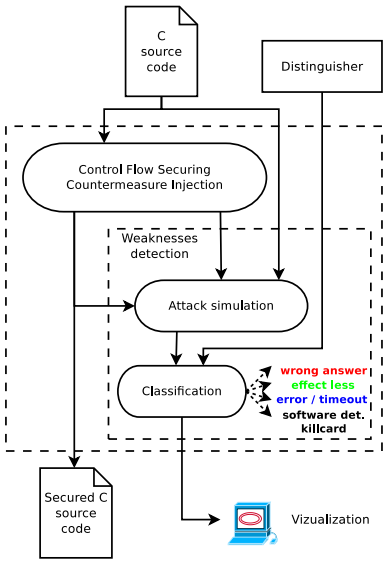


FIGURE 3.15 – *Framework* cfi-c

Avec ces quatre propriétés, à tout moment le long d'un chemin parcouru par le *model checker*, les compteurs $cntv\_\alpha i$ et $cntv\_\alpha(i+1)$ pour deux instructions du code adjacentes seront distants de 1 ou égaux, ce qui réduit le produit des deux modèles et modélise une exécution correcte du programme, du point de vue du flot de contrôle. A la fin, les compteurs $cntv\_\alpha i$ de $M(c)$ et $CM(c)$ doivent être égaux, ce qui garantit que les deux exécutions sont équivalentes. Ces propriétés doivent rester valides quelles que soient les transitions d'attaques empruntées dans $CM(c)$ (ici violettes dans la figure 3.12 pour les sauts arrière depuis L3).

Nous avons utilisé Vis [109] comme *model checker* pour vérifier ces propriétés sur nos trois types de contremesures (flot linéaire et appel de fonction, if-then-else, boucles). Plusieurs fois, Vis a trouvé des chemins ne respectant pas les propriétés, mettant en lumière une attaque non prise en compte par les premières versions de nos contremesures. Ces attaques étaient aussi observables dans nos essais expérimentaux sur AES. Cette phase de vérification est donc particulièrement utile pour les contremesures complexes à mettre en œuvre, notamment pour la boucle *while*.

4 Logiciels

Lors des travaux présentés dans ce chapitre, deux logiciels ont été réalisés : GroddDroid pour l'automatisation de l'exécution de *malware* Android et cfi-c pour la simulation d'attaques physiques et l'injection de contremesures.

4.1 GroddDroid

GroddDroid [84], en référence au gorille Grodd [1] aux pouvoirs psychiques de DC Comics, est un logiciel open source disponible sur le site du projet Kharon (`http://kharon.gforge.inria.fr/grodddroid.html`). Il a été développé conjointement avec CentraleSupélec et Inria (Adrien Abraham, Radoniaina Andriatsimandefitra, Adrien Brunelat, Mourad LesLous, Valérie Viet Triem Tong et de nombreux étudiants ingénieurs). Il manipule des programmes Android (fichiers APK) qu'il déploie sur un téléphone fraichement réinstallé, et qu'il manipule comme si un utilisateur réel l'utilisait. Comme représenté sur la figure 3.14, deux modes d'exécutions sont possibles :

— l'exécution de référence qui permet de tracer ce qui est exécuté grâce au *Control Flow Tracer* qui a préalablement injecté du code dans l'APK.
— l'exécution avec *Branch Forcing* qui permet d'attaquer le flot de contrôle pour le pousser vers le code malveillant.

Le *Runner* est modulaire afin de pouvoir comparer celui de GroddDroid au Monkey [132] par exemple.

1. `https://fr.wikipedia.org/wiki/Gorilla_Grodd`

4.2 cfi-c

Cfi-c est un logiciel open source disponible à http://cfi-c.gforge.inria.fr/. Il a été développé conjointement avec l'Université Pierre et Marie Curie (Karine Heydemann) et l'INSA Centre Val de Loire (Pascal Berthomé). Cfi-c manipule le code C d'un programme pour injecter les fautes simulées et/ou les contremesures logicielles au niveau du flot de contrôle. La figure 3.15 montre les interactions possibles avec les autres briques logicielles. Après une campagne de simulation d'attaques, un classifieur permet de trier les résultats d'attaque en fonction de l'effet sur le programme. Enfin, un logiciel de visualisation a été développé. Il permet de parcourir l'ensemble des attaques et visualiser les sauts concernés sur le code source du programme considéré.

Notre outil a été utilisé pour enrichir un benchmark récent dédié aux cartes puces : FISSC (*a Fault Injection and Simulation Secure Collection*) [121]. Pour certains exemples de code de la carte contenu dans ce benchmark, nous avons réalisé la version sécurisée de ceux-ci à l'aide de nos contremesures. Ces versions sécurisées ont été ajoutées au benchmark FISSC.

5 Bilan et perspectives de recherche

Ce chapitre s'est focalisé sur la sécurité du flot de contrôle. Les deux cas d'étude proposés sont très différents : la carte à puce subit des attaques contre son flot de contrôle et il est important de les prévenir. A l'inverse, nous utilisons des attaques de flot de contrôle pour étudier des *malware* Android. Sur ces deux exemples, les perspectives sont nombreuses et ne se limitent pas au flot de contrôle.

La gestion des attaques multiples commence à être traitée par des travaux récents [156, 121] puisque réaliser des attaques combinant plusieurs fautes est une réalité [175]. Même si ces attaques sont difficiles à réaliser, d'un point de vue du flot, elles permettraient de sortir du flot à un point donné et de revenir plus tard pour reprendre le flot initial. Si les contremesures ne s'activent qu'aux bornes d'un *basic block*, cela permettrait de contourner les contremesures de la littérature. Il reste à évaluer la faisabilité de telles attaques, surtout si des contremesures sécurisant le code pour des fautes simples sont déjà présentes.

Nous n'avons traité que l'impact sur le contrôle de flot. Certaines approches traitent l'impact sur les variables ou les registres en dupliquant les instructions du code au niveau du code assembleur [149, 100]. Ainsi, les sauts de quelques instructions assembleur impactant des valeurs peuvent être détectés. Cependant, il faut manipuler le code compilé ce qui peut être gênant pour un industriel. Les solutions proposées pourraient être adaptées au niveau du langage de programmation utilisé. Une combinaison entre la duplication à grain fin (niveau assembleur) et à gros grain (niveau langage) permettrait d'assurer une sécurité homogène sur les valeurs quelle que soit la conséquence d'une attaque (un saut long ou court). Ces pistes restent à explorer. Par ailleurs, en implémentant des contremesures à haut niveau, les industriels pourraient plus facilement certifier leurs programmes en évitant la modification de la chaine de compilation. En effet, en utilisant un compilateur lui-même certifié (comme Keil [137]), un industriel peut plus facilement faire certifier le code embarqué développé.

Concernant l'analyse de *malware*, nous poursuivons nos efforts pour étudier et comprendre les comportements malveillants. Un de nos objectifs à court terme est de mettre en place une plate-forme d'analyse de *malware* où tout un chacun pourrait soumettre un *malware* et récupérer un résultat d'analyse. Cela représente un moyen de collecter des échantillons et d'avoir une étude statistique plus large sur un ensemble conséquent de *malware*. Par ailleurs, comprendre le fonctionnement d'un *malware* et analyser son exécution nous aide à élaborer des solutions de détection. Si une exécution devient observable, alors les méthodes de détection n'utilisent plus la sémantique du *malware* mais l'analyse des observations de ses actions. Les premiers résultats déjà obtenus sont encourageants. Pour obtenir de meilleurs résultats, il faut encore améliorer le déclenchement de comportements malveillants. De nombreuses choses sont à prendre en compte : une meilleure manipulation de l'interface graphique, les *inputs* textuels de l'utilisateur, la génération de données conformes à ce qui est attendu par le code malveillant, la simulation de protocole lorsque le *malware* se connecte à un serveur. Nos efforts se portent aussi sur la prise en compte des appels implicites, c'est-à-dire quand le *framework* Android décide d'exécuter une portion de code d'une application, et donc génère une exécution à partir d'un nouveau point d'entrée. L'ensemble de ces problématiques est à traiter indépendamment et requiert des solutions de recherche dédiées.

La visualisation de la sécurité devient aussi petit à petit une partie importante de mes travaux de recherche. Pour les carte à puce, l'outil de visualisation permet de classer les attaques par type et d'en sélectionner une visuellement pour obtenir sa représentation sur le code source. Concernant les *malware* Android, je travaille activement sur la visualisation des données d'exécution récoltées, à la fois au niveau système d'exploitation mais aussi au niveau du *bytecode* exécuté. Le but est de pouvoir rejouer l'exécution du *malwarea posteriori* et de représenter cette exécution conjointement à ces deux niveaux.

Souvent, on considère un modèle d'attaque où l'attaquant est extérieur et le programme cherche à se protéger, sans s'appuyer sur des mécanismes bas niveau particuliers. Mais si l'attaque provient du programme lui-même, c'est-à-dire que l'on n'a pas confiance dans l'origine et les intentions de ce programme et que l'on dispose éventuellement de son code source, on peut imaginer vouloir renforcer le contrôle de celui-ci en lui injectant des modifications permettant de l'empêcher de réaliser des opérations jugées dangereuses. Cette hypothèse peut paraître surprenante mais elle peut prendre son sens dans des cas particuliers, par exemple, dans un contexte militaire où le niveau de sécurité est tel que l'on dispose des codes sources des applications et que l'on doute de leurs intentions. Sans aller jusqu'à une telle hypothèse, on peut imaginer tisser des aspects à un programme compilé pour un téléphone mobile, afin de lui ajouter du code, pour par exemple l'instrumenter afin de surveiller son fonctionnement.

De manière plus générale, je crois qu'il est important de continuer à élaborer des mécanismes d'auto-protection à l'intérieur des codes sources des applications en minimisant l'impact et les modifications nécessaires à l'écosystème de celui-ci. Il s'agit donc bien "d'auto-protection" : le programme doit, par lui-même, chercher à contrecarrer les vecteurs d'attaques, qu'ils soient internes ou externes.

6 Références

[3] X. Kauffmann-Tourkestansky. **Analyses sécuritaires de code de carte à puce sous attaques physiques simulées**. Thèse de doctorat. Université d'Orléans, novembre 2012 (cf. p. 5, 37).

[80] **Application of Attack Potential to Smartcards**. Rapport technique March. Common Criteria, 2009 (cf. p. 44).

[81] Y. Aafer, W. Du et H. Yin. **DroidAPIMiner : Mining API-Level Features for Robust Malware Detection in Android**. Dans : *Security and Privacy in Communication Networks* 127 (2013). Sous la dir. de T. Zia, A. Zomaya, V. Varadharajan et M. Mao, p. 86–103. doi : 10.1007/978-3-319-04283-1_6 (cf. p. 40, 41).

[82] M. Abadi, M. Budiu, Ú. Erlingsson et J. Ligatti. **Control-flow integrity**. Dans : *12th ACM conference on Computer and communications security*. Alexandria, USA : ACM Press, novembre 2005, p. 340–353. doi : 10.1145/1102120.1102165 (cf. p. 33, 35, 36, 47, 53).

[83] M. Abadi, M. Budiu, Ú. Erlingsson et J. Ligatti. **Control-flow integrity principles, implementations, and applications**. Dans : *ACM Transactions on Information and System Security* 13.1 (octobre 2009), p. 1–40. doi : 10.1145/1609956.1609960 (cf. p. 47).

[84] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande et V. Viet Triem Tong. **GroddDroid : a Gorilla for Triggering Malicious Behaviors**. Dans : *10th International Conference on Malicious and Unwanted Software*. Fajardo, Puerto Rico : IEEE Computer Society, octobre 2015, p. 119–127. doi : 10.1109/MALWARE.2015.7413692 (cf. p. 6, 38, 39, 43, 55).

[85] M.-l. Akkar, L. Goubin et O. Ly. *Automatic Integration of Counter-Measures Against Fault Injection Attacks*. 2003 (cf. p. 48).

[86] F. E. Allen. **Control flow analysis**. Dans : *SIGPLAN Notices* 5.7 (1970), p. 1–19. doi : 10.1145/390013.808479 (cf. p. 33).

[87] K. Allix, T. F. Bissyandé, J. Klein et Y. Le Traon. **AndroZoo : collecting millions of Android apps for the research community**. Dans : *13th International Workshop on Mining Software Repositories*. Austin, USA : ACM Press, mai 2016, p. 468–471. doi : 10.1145/2901739.2903508 (cf. p. 42).

[88] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine et A. M. Memon. **Using GUI ripping for automated testing of Android applications**. Dans : *27th IEEE/ACM International Conference on Automated Software Engineering*. Essen, Germany : ACM Press, septembre 2012, p. 258–261. doi : 10.1145/2351676.2351717 (cf. p. 39).

[89] S. Anand, M. Naik, M. J. Harrold et H. Yang. **Automated concolic testing of smartphone apps**. Dans : *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. Cary, USA : ACM Press, novembre 2012. doi : 10.1145/2393596.2393666 (cf. p. 39).

[90] R. Anderson et M. Kuhn. **Tamper resistance - a cautionary note**. Dans : *Second USENIX Workshop on Electronic Commerce*. Oakland, USA : USENIX Association, novembre 1996, p. 1–11 (cf. p. 33, 37, 44).

[91] R. Andriatsimandefitra, S. Geller et V. Viet Triem Tong. **Designing information flow policies for Android's operating system**. Dans : *IEEE International Conference on Communications*. Ottawa, Canada : IEEE Computer Society, juin 2012, p. 976–981. doi : 10.1109/ICC.2012.6364161 (cf. p. 38, 40).

[92] W. Arthur, B. Mehne, R. Das et T. Austin. **Getting in control of your control flow with control-data isolation**. Dans : *13th Annual {IEEE/ACM} International Symposium on Code Generation and Optimization*. San Francisco, CA, USA : IEEE Computer Society, 2015, p. 79–90. doi : 10.1109/CGO.2015.7054189 (cf. p. 47, 51).

[93] S. Arzt, S. Rasthofer et E. Bodden. **Instrumenting Android and Java Applications as Easy as abc**. Dans : *Fourth International Conference on Runtime Verification*. T. 8174. LNCS. Rennes, France : Springer Berlin Heidelberg, septembre 2013, p. 364–381. doi : 10.1007/978-3-642-40787-1_26 (cf. p. 41).

[94] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau et P. McDaniel. **FlowDroid : Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps**. Dans : *ACM SIGPLAN Conference on Programming Language Design and Implementation*. T. 49. 6. Edinburgh, UK : ACM Press, juin 2014, p. 259–269. doi : 10.1145/2666356.2594299 (cf. p. 38).

[95] T. Azim et I. Neamtiu. **Targeted and Depth-first Exploration for Systematic Testing of Android Apps**. Dans : *ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. T. 48. 10. Indianapolis, USA : ACM Press, 2013, p. 641–660. doi : 10.1145/2544173.2509549 (cf. p. 40).

[96] J. Balasch, B. Gierlichs et I. Verbauwhede. **An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs**. Dans : *The 8th Workshop on Fault Diagnosis and Tolerance in Cryptography*. Sous la dir. de L. Breveglieri, S. Guilley, I. Koren, D. Naccache et J. Takahashi. Nara, Japan : IEEE Computer Society, 2011, p. 105–114. doi : 10.1109/FDTC.2011.9 (cf. p. 44).

[97] G. Barbu, G. Duc et P. Hoogvorst. **Java card operand stack : fault attacks, combined attacks and countermeasures**. Dans : *10th Smart Card Research and Advanced Application IFIP Conference*. T. 7079. LNCS. Leuven, Belgium : Springer Berlin / Heidelberg, septembre 2011, p. 297–313. doi : 10.1007/978-3-642-27257-8_19 (cf. p. 36, 45).

[98] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall et C. Whelan. **The sorcerer's apprentice guide to fault attacks**. Dans : *Proceedings of the IEEE* 94.2 (février 2006), p. 370–382. doi : 10.1109/JPROC.2005.862424 (cf. p. 33, 44).

[99] A. Barenghi, L. Breveglieri, I. Koren et D. Naccache. **Fault injection attacks on cryptographic devices : Theory, practice, and countermeasures**. Dans : *Proceedings of the IEEE* 100.11 (2012), p. 3056–3076. doi : 10.1109/JPROC.2012.2188769 (cf. p. 44).

[100] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi et F. Regazzoni. **Countermeasures against fault attacks on software implemented AES**. Dans : *5th Workshop on Embedded Systems Security*. Scottsdale, USA : ACM Press, octobre 2010, p. 1–10. doi : 10.1145/1873548.1873555 (cf. p. 48, 56).

[101] T. Barry, D. Couroussé et B. Robisson. **Compilation of a Countermeasure Against Instruction-Skip Fault Attacks**. Dans : *Third Workshop on Cryptography and Security in Computing Systems*. Prague, Czech Republic : ACM Press, 2016, p. 1–6. doi : 10.1145/2858930.2858931 (cf. p. 48).

[102] P. Berthome, K. Heydemann, X. Kauffmann-Tourkestansky et J.-F. Lalande. **High Level Model of Control Flow Attacks for Smart Card Functional Security**. Dans : *Seventh International Conference on Availability, Reliability and Security*. Prague, Czech Republic : IEEE Computer Society, août 2012, p. 224–229. doi : 10.1109/ARES.2012.79 (cf. p. 6, 44, 46).

[103] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky et J.-F. Lalande. **Attack model for verification of interval security properties for smart card C codes**. Dans : *5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. Toronto Canada : ACM, juin 2010, p. 1–12. doi : 10.1145/1814217.1814219 (cf. p. 6, 37).

[104] T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe et S. Albayrak. **An Android Application Sandbox system for suspicious software detection**. Dans : *5th International Conference on Malicious and Unwanted Software*. Fajardo, USA : IEEE Computer Society, octobre 2010, p. 55–62. doi : 10.1109/MALWARE.2010.5665792 (cf. p. 39).

[105] T. Bletsch, X. Jiang et V. Freeh. **Mitigating code-reuse attacks with control-flow locking**. Dans : *The 27th Annual Computer Security Applications Conference*. Orlando, Florida, USA : ACM Press, décembre 2011, p. 353–362. doi : 10.1145/2076732.2076783 (cf. p. 47).

[106] D. Boneh, R. A. DeMillo et R. J. Lipton. **On the importance of checking cryptographic protocols for faults**. Dans : *16th annual international conference on Theory and application of cryptographic techniques*. Konstanz, Germany : Springer, mai 1997, p. 37–51. doi : 10.1007/3-540-69053-0_4 (cf. p. 36).

[107] G. Bouffard, M. Lackner, J.-l. Lanet et J. Loinig. **Heap . . . Hop ! Heap is also Vulnerable**. Dans : *13th Smart Card Research and Advanced Application Conference*. T. 8968. LNCS. Paris, France : Springer Berlin / Heidelberg, novembre 2014, p. 18–31. doi : 10.1007/978-3-319-16763-3_2 (cf. p. 35, 45).

[108] G. Bouffard et J.-L. Lanet. **The ultimate control flow transfer in a Java based smart card**. Dans : *Computers & Security* 50 (2015), p. 33–46. doi : 10.1016/j.cose.2015.01.004 (cf. p. 36).

[109] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Staple, G. Swamy et T. Villa. **VIS : A system for verification and synthesis**. Dans : *Computer Aided Verification*. Sous la dir. de R. Alur et T. Henzinger. T. 1102. Lecture Notes in Computer Science. New Brunswick, USA : Springer, Heidelberg, 1996, p. 428–432. doi : 10.1007/3-540-61474-5_95 (cf. p. 55).

[110] I. Burguera, U. Zurutuza et S. Nadjm-Tehrani. **Crowdroid : Behavior-Based Malware Detection System for Android**. Dans : *1st ACM workshop on Security and privacy in smartphones and mobile devices*. Chicago, USA : ACM Press, octobre 2011, p. 15–26. doi : 10.1145/2046614.2046619 (cf. p. 39, 40).

[111] Y. Cao, Y. Fratantonio, A. Bianchi et M. Egele. **EdgeMiner : Automatically Detecting Implicit Control Flow Transitions through the Android Framework**. Dans : *22nd Annual Network and Distributed System Security Symposium*. February. San Diego, USA : The Internet Society, février 2015, p. 8–11 (cf. p. 40, 42).

[112] D. Ceară, L. Mounier et M. L. Potet. **Taint Dependency Sequences : A characterization of insecure execution paths based on input-sensitive cause sequences**. Dans : *3rd International Conference on Software Testing, Verification, and Validation Workshops*. Paris, France : IEEE Computer Society, avril 2010, p. 371–380. doi : 10.1109/ICSTW.2010.28 (cf. p. 45).

[113] H. Chen et B. Zang. **CFIMon : Detecting violation of control flow integrity using performance counters**. Dans : *IEEE/IFIP International Conference on Dependable Systems and Networks*. Boston, USA : IEEE Computer Society, juin 2012, p. 1–12. doi : 10.1109/DSN.2012.6263958 (cf. p. 47).

[114] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha et M. H. Jakubowski. **Oblivious hashing : A stealthy software integrity verification primitive**. Dans : *5th International Workshop on Information Hiding*. Noordwijkerhout, The Netherlands : Springer-Verlag Berlin Heidelberg, octobre 2003, p. 400–414. doi : 10.1007/3-540-36415-3_26 (cf. p. 47).

[115] S. R. Choudhary, A. Gorla et A. Orso. *Automated Test Input Generation for Android : Are We There Yet ?* 2015. arXiv : arXiv:1503.07217v1 (cf. p. 39).

[116] R. de Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. D. Bosschere, B. Preneel, B. D. Sutter et I. Verbauwhede. **SOFIA : Software and control flow integrity architecture**. Dans : *Design, Automation & Test in Europe*. Sous la dir. de L. Fanucci et J. Teich. Dresden, Germany : IEEE Computer Society, mars 2016, p. 1172–1177 (cf. p. 47).

[117] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle et E. Walthinsen. **Protecting systems from stack smashing attacks with StackGuard**. Dans : *Linux Expo*. 1999 (cf. p. 47).

[118] J.-L. Danger, S. Guilley, T. Porteboeuf, F. Praden et M. Timbert. **HCODE : Hardware-Enhanced Real-Time CFI**. Dans : *4th Program Protection and Reverse Engineering Workshop*. Sous la dir. de M. D. Preda et J. T. McDonald. New Orleans, USA : ACM Press, janvier 2014. doi : 10.1145/2689702.2689708 (cf. p. 47).

[119] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger et A.-R. Sadeghi. **MoCFI : A Framework to Mitigate Control-Flow Attacks on Smartphones**. Dans : *19th Annual Network and Distributed System Security Symposium*. San Diego, California, USA : The Internet Society, février 2012 (cf. p. 38).

[120] A. Dehbaoui, A.-p. Mirbaha, N. Moro, J.-m. Dutertre et A. Tria. **Electromagnetic Glitch on the AES Round Counter**. Dans : *4th International conference on Constructive Side-Channel Analysis and Secure Design*. T. 7864. Paris, France : Springer Berlin / Heidelberg, mars 2013, p. 17–31. doi : 10.1007/978-3-642-40026-1_2 (cf. p. 36, 44).

[121] L. Dureuil, G. Petiot, M.-l. Potet, A. Crohen et P. D. Choudens. **FISSC : a Fault Injection and Simulation Secure Collection**. Dans : *International Conference on Computer Safety, reliability and Security*. T. 9922. LNCS. Trondheim, Norway : Springer Berlin / Heidelberg, 2016, p. 3–11. doi : 10.1007/978-3-319-45477-1_1 (cf. p. 56).

[122] T. Eder, M. Rodler, D. Vymazal et M. Zeilinger. **ANANAS - A Framework For Analyzing Android Applications**. Dans : *Eighth International Conference on Availability, Reliability and Security*. Regensburg, Germany : IEEE Computer Society, juillet 2013, p. 711–719. doi : 10.1109/ARES.2013.93. arXiv : 1307.5410 (cf. p. 39).

[123] M. Egele, T. Scholte, E. Kirda et C. Kruegel. **A survey on automated dynamic malware-analysis techniques and tools**. Dans : *ACM Computing Surveys* 44.2 (2012). doi : 10.1145/2089125.2089126 (cf. p. 38).

[124] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel et A. N. Sheth. **TaintDroid : an information-flow tracking system for realtime privacy monitoring on smartphones**. Dans : *9th USENIX Symposium on Operating Systems Design and Implementation*. Vancouver, BC, Canada : USENIX Association, octobre 2010, p. 393–407 (cf. p. 38, 40).

[125] A. Fiskiran et R. Lee. **Runtime execution monitoring (REM) to detect and prevent malicious code execution**. Dans : *IEEE International Conference on Computer Design : VLSI in Computers and Processors*. San Jose, California : IEEE Computer Society, octobre 2004, p. 452–457. doi : 10.1109/ICCD.2004.1347961 (cf. p. 47).

[126] A. Francillon, D. Perito et C. Castelluccia. **Defending embedded systems against control flow attacks**. Dans : *First ACM workshop on Secure execution of untrusted code*. Chicago, USA : ACM Press, novembre 2009, p. 19. doi : 10.1145/1655077.1655083 (cf. p. 35, 36, 47).

[127] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel et G. Vigna. **TriggerScope : Towards Detecting Logic Bombs in Android Applications**. Dans : *37th IEEE Symposium on Security and Privacy*. San Jose, USA : IEEE Computer Society, mai 2016, p. 1–33. doi : 10.1109/SP.2016.30 (cf. p. 40).

[128] A. Gianazza, F. Maggi, A. Fattori, L. Cavallaro et S. Zanero. *PuppetDroid : A User-Centric UI Exerciser for Automatic Dynamic Analysis of Similar Android Applications*. Février 2014. arXiv : 1402.4826 (cf. p. 40).

[129] C. Giraud. **DFA on AES**. Dans : *AES Conference 2004*. Sous la dir. de H. Dobbertin, V. Rijmen et A. Sowa. T. 3373. LNCS. Bonn, Germany : Springer Berlin Heidelberg, mai 2004, p. 27–41. doi : 10.1007/11506447_4 (cf. p. 44).

[130] O. Goloubeva, M. Rebaudengo, M. S. Reorda et M. Violante. **Soft-error detection using control flow assertions**. Dans : *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. Boston, MA, USA : IEEE Computer Society, novembre 2003, p. 581–588. doi : 10.1109/DFTVS.2003.1250158 (cf. p. 33, 47).

[131] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda et M. Violante. **Improved software-based processor control-flow errors detection technique**. Dans : *Annual Reliability and Maintainability Symposium*. Alexandria, VA, USA : IEEE Computer Society, janvier 2005, p. 583–589. doi : 10.1109/RAMS.2005.1408426 (cf. p. 46, 47).

[132] Google. *UI/Application Exerciser Monkey* (cf. p. 39, 43, 55).

[133] M. Grace, Y. Zhou, Q. Zhang, S. Zou et X. Jiang. **RiskRanker : Scalable and Accurate Zero-day Android Malware Detection**. Dans : *10th International Conference on Mobile Systems, Applications, and Services*. Low Wood Bay, Lake District, UK : ACM Press, juin 2012, p. 281–294. doi : 10.1145/2307636.2307663 (cf. p. 38, 42).

[134] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge et R. B. Brown. **MiBench : A free, commercially representative embedded benchmark suite.** Dans : *IEEE 4th Annual Workshop on Workload Characterization*. Austin, TX : IEEE Computer Society, décembre 2001, p. 3–14. doi : 10.1109/WWC.2001.990739 (cf. p. 51).

[135] E. Hess, N. Janssen, B. Meyer et T. Schütze. **Information Leakage Attacks Against Smart Card Implementations of Cryptographic Algorithms and Countermeasures–a Survey**. Dans : *Eurosmart Security Conference*. T. 130. Juin 2000, p. 55–64 (cf. p. 44).

[136] J. Iguchi-cartigny et J.-l. Lanet. **Evaluation of Countermeasures Against Fault Attacks on Smart Cards**. Dans : *International Journal of Security and Its Applications* 5.2 (2011), p. 49–60 (cf. p. 36).

[137] Keil. *Keil uVision for ARM processors*. 2012 (cf. p. 56).

[138] V. Kiriansky, D. Bruening et S. Amarasinghe. **Secure execution via program shepherding**. Dans : *USENIX Security Symposium*. San Francisco, USA : USENIX Association, août 2002 (cf. p. 33).

[139] C. Kirkland. *Engineering the Channel Tunnel*. 1995 (cf. p. 47).

[140] N. Kiss, J.-F. Lalande, M. Leslous et V. Viet Triem Tong. **Kharon dataset : Android malware under a microscope**. Dans : *The LASER Workshop : Learning from Authoritative Security Experiment Results*. San Jose, United States : USENIX Association, mai 2016, p. 1–12 (cf. p. 6, 40–42, 69).

[141] O. Kömmerling et M. G. Kuhn. **Design principles for tamper-resistant smartcard processors**. Dans : *USENIX Workshop on Smartcard Technology*. USENIX Association, mai 1999, p. 2 (cf. p. 44).

[142] J.-F. Lalande, K. Heydemann et P. Berthomé. **Software Countermeasures for Control Flow Integrity of Smart Card C Codes**. Dans : *19th European Symposium on Research in Computer Security*. Sous la dir. de M. Kutylowski et J. Vaidya. T. 8713. LNCS. Wroclaw, Pologne : Springer International Publishing, septembre 2014, p. 200–218. doi : 10.1007/978-3-319-11212-1_12 (cf. p. 6, 38, 47).

[143] I. Levin. *A byte-oriented AES-256 implementation*. 2007 (cf. p. 51).

[144] L. Li, A. Bartel, T. F. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau et P. McDaniel. **IccTA : Detecting Inter-Component Privacy Leaks in Android Apps**. English. Dans : *IEEE/ACM 37th IEEE International Conference on Software Engineering*. Firenze, Italy : IEEE, mai 2015, p. 280–291. doi : 10.1109/ICSE.2015.48 (cf. p. 40, 83).

[145] M. Lindorfer et M. Neugschwandtner. **ANDRUBIS - 1,000,000 Apps Later : A View on Current Android Malware Behaviors**. Dans : *3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. San Jose, CA, USA : IEEE Computer Society, septembre 2014. doi : 10.1109/BADGERS.2014.7 (cf. p. 39, 40).

[146] A. Machiry, R. Tahiliani et M. Naik. **Dynodroid : an input generation system for Android apps**. Dans : *9th Joint Meeting on Foundations of Software Engineering*. Saint Pertersburg, Russia : ACM Press, août 2013, p. 224–234. doi : 10.1145/2491411.2491450 (cf. p. 40).

[147] R. Mahmood, N. Mirzaei et S. Malek. **EvoDroid : segmented evolutionary testing of Android apps**. Dans : *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China : ACM Press, novembre 2014, p. 599–609. doi : 10.1145/2635868.2635896 (cf. p. 39).

[148] Mila Parkour. *Contagio mobile*. 2012 (cf. p. 42).

[149] N. Moro, K. Heydemann, E. Encrenaz, B. Robisson, N. Moro et E. Encrenaz. **Formal verification of a software countermeasure against instruction skip attacks**. Dans : *Journal of Cryptographic Engineering* 4.3 (2014), p. 1–12. doi : 10.1007/s13389-014-0077-7 (cf. p. 36, 48, 56).

[150] S. Neuner, V. V. D. Veen et M. Lindorfer. **Enter Sandbox : Android Sandbox Comparison**. Dans : *3rd IEEE Mobile Security Technologies Workshop*. San Jose, CA, mai 2014. arXiv : arXiv:1410.7749 (cf. p. 39, 40).

[151] N. Oh, P. Shirvani et E. McCluskey. **Control-flow checking by software signatures**. Dans : *IEEE Transactions on Reliability* 51.1 (mars 2002), p. 111–122. doi : 10.1109/24.994926 (cf. p. 36, 47).

[152] M. Payer, A. Barresi et T. R. Gross. **Fine-Grained Control-Flow Integrity Through Binary Hardening**. Dans : *12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. T. 9148. LNCS. Milan, Italy : Springer Berlin / Heidelberg, 2015, p. 144–164. doi : 10.1007/978-3-319-20550-2_8 (cf. p. 47).

[153] N. L. Petroni et M. Hicks. **Automated detection of persistent kernel control-flow attacks**. Dans : *14th ACM conference on Computer and communications security*. Alexandria, USA : ACM Press, octobre 2007, p. 103. doi : 10.1145/1315245.1315260 (cf. p. 47).

[154] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel et G. Vigna. **Execute this ! analyzing unsafe and malicious dynamic code loading in android applications**. Dans : *Network and Distributed System Security Symposium*. T. 14. San Diego, California, USA : The Internet Society, février 2014, p. 23–26 (cf. p. 38).

[155] M.-L. Potet, L. Mounier, M. Puys et L. Dureuil. **Lazart : A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections**. Dans : *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. Cleveland, USA : IEEE, mars 2014, p. 213–222. doi : 10.1109/ICST.2014.34 (cf. p. 46).

[156] M. Puys, L. Rivière, J. Bringer et T. Le. **High-Level Simulation for Multiple Fault Injection Evaluation**. Dans : *3rd International Workshop on Quantitative Aspects in Security Assurance*. T. 8872. LNCS. Wroclaw, Poland : Springer Berlin / Heidelberg, 2015, p. 293–308. doi : 10.1007/978-3-319-17016-9_19 (cf. p. 46, 56).

[157] V. Rajesh, J. Hayes et B. Murray. **Low-cost on-line fault detection using control flow assertions**. Dans : *9th IEEE On-Line Testing Symposium*. Kos Island, Greece : IEEE Computer Society, juillet 2003, p. 137–143. doi : 10.1109/OLT.2003.1214380 (cf. p. 36).

[158] V. Rastogi, Y. Chen et W. Enck. **AppsPlayground : Automatic Security Analysis of Smartphone Applications**. Dans : *Third ACM conference on Data and application security and privacy*. San Antonio, TX, USA : ACM Press, 2013, p. 209. doi : 10.1145/2435349.2435379 (cf. p. 39).

[159] V. Rastogi, Y. Chen et X. Jiang. **Catch Me If You Can : Evaluating Android Anti-Malware Against Transformation Attacks**. Dans : *IEEE Transactions on Information Forensics and Security* 9.1 (janvier 2014), p. 99–108. doi : 10.1109/TIFS.2013.2290431 (cf. p. 38).

[160] A. Reina, A. Fattori et L. Cavallaro. **A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors**. Dans : *6th European Workshop on System Security*. Prague, Czech, avril 2013 (cf. p. 39, 40).

[161] L. Riviere, J. Bringer, T.-H. Le et H. Chabanne. **A novel simulation approach for fault injection resistance evaluation on smart cards**. English. Dans : *6th international Workshop on Security Testing*. Sectest. Graz, Austria : IEEE Computer Society, avril 2015, p. 1–8. doi : 10.1109/ICSTW.2015.7107460 (cf. p. 46).

[162] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi et T. Holz. **Counterfeit object-oriented programming : On the difficulty of preventing code reuse attacks in C++ applications**. Dans : *IEEE Symposium on Security and Privacy*. San Jose, USA : IEEE Computer Society, mai 2015, p. 745–762. doi : 10.1109/SP.2015.51 (cf. p. 33).

[163] A. a. Sere, J. Iguchi-Cartigny et J.-L. Lanet. **Automatic detection of fault attack and countermeasures**. Dans : *4th Workshop on Embedded Systems Security*. New York, New York, USA : ACM Press, 2009, p. 1–7. doi : 10.1145/1631716.1631723 (cf. p. 36, 45).

[164] SFR. **(U)SIM Java Card Platform Protection Profile Basic and SCWS Configurations-Evolutive Certification Scheme for (U)SIM cards**. Rapport technique. 2010 (cf. p. 33).

[165] H. Shacham. **The geometry of innocent flesh on the bone**. Dans : *14th ACM conference on Computer and communications security*. Alexandria, USA : ACM Press, octobre 2007, p. 552–561. doi : 10.1145/1315245.1315313 (cf. p. 33, 47).

[166] S. P. Skorobogatov et R. J. Anderson. **Optical Fault Induction Attacks Sergei**. Dans : *4th International Workshop on Cryptographic Hardware and Embedded Systems*. T. 2523. LNCS. Redwood Shores, USA : ACM Press, août 2002, p. 2–12. doi : 10.1007/3-540-36400-5 (cf. p. 37, 44).

[167] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck et J. Hoffmann. **Mobile-sandbox : Having a Deeper Look into Android Applications Michael**. Dans : *28th Annual ACM Symposium on Applied Computing*. Coimbra, Portugal : ACM Press, mars 2013, p. 1808–1815. doi : 10.1145/2480362.2480701 (cf. p. 39, 40).

[168] K. Tam, S. Khan, A. Fattori et L. Cavallaro. **CopperDroid : Automatic Reconstruction of Android Malware Behaviors**. Dans : *22nd Annual Network and Distributed System Security Symposium*. San Diego, California, USA : The Internet Society, février 2015 (cf. p. 39, 40).

[169] P. Teuwen. **How to Make Smartcards Resistant to Hackers' Lightsabers ?** Dans : *Foundations for Forgery-Resilient Cryptographic Hardware*. Sous la dir. de Jorge Guajardo and Bart Preneel and Ahmad-Reza Sadeghi and Pim Tuyls. Dagstuhl, 2010, p. 1–8 (cf. p. 45).

[170] N. Timmers, A. Spruyt et M. Witteman. **Controlling PC on ARM using Fault Injection**. Dans : *Fault Diagnosis and Tolerance in Cryptography*. Santa Barbara, USA : IEEE Computer Society, août 2016 (cf. p. 45, 67).

[171] I. Verbauwhede. **The fault attack jungle - A classification model to guide you**. Dans : *Fault Diagnosis and Tolerance in Cryptography*. Tokyo, Japan : IEEE Computer Society, septembre 2011, p. 3–8. doi : 10.1109/FDTC.2011.13 (cf. p. 33, 35, 36).

[172] E. Vetillard et A. Ferrari. **Combined attacks and countermeasures**. Dans : *The 9th Smart Card Research and Advanced Application IFIP Conference*. Limoges, France : Springer Berlin / Heidelberg, avril 2010, p. 133–147. doi : 10.1007/978-3-642-12510-2_10 (cf. p. 36, 45).

[173] T. Vidas et N. Christin. **Evading Android Runtime Analysis via Sandbox Detection**. Dans : *9th ACM Symposium on Information, Computer and Communications Security*. Kyoto, Japan : ACM Press, juin 2014, p. 447–458. doi : 10.1145/2590296.2590325 (cf. p. 38).

[174] D. Wagner. **Cryptanalysis of a provably secure CRT-RSA algorithm**. Dans : *11th ACM conference on Computer and communications security*. Chicago, USA : ACM Press, octobre 2004, p. 92–97. doi : 10.1145/1030083.1030097 (cf. p. 36).

[175] B. Wang, L. Liu, C. Deng, M. Zhu, S. Yin et S. Wei. **Against Double Fault Attacks : Injection Effort Model, Space and Time Randomization Based Countermeasures for Reconfigurable Array Architecture**. Dans : *IEEE Transactions on Information Forensics and Security* 11.6 (juin 2016), p. 1151–1164. doi : 10.1109/TIFS.2016.2518130 (cf. p. 44, 56).

[176] L. Weichselbaum. **Andrubis : Android Malware Under The Magnifying Glass**. Rapport technique. 2014 (cf. p. 39, 40).

[177] M. Werner, E. Wenger et S. Mangard. **Protecting the Control Flow of Embedded Processors against Fault Attacks**. Dans : *14th International Conference Smart Card Research and Advanced Applications*. Sous la dir. de N. Homma et M. Medwed. T. 9514. LNCS. Bochum, Germany : Springer Berlin / Heidelberg, novembre 2015, p. 161–176. doi : 10.1007/978-3-319-31271-2_10 (cf. p. 47).

[178] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee et K.-P. Wu. **DroidMat : Android Malware Detection through Manifest and API Calls Tracing**. Dans : *Seventh Asia Joint Conference on Information Security*. Tokyo, Japan : IEEE Computer Society, août 2012, p. 62–69. doi : 10.1109/AsiaJCIS.2012.18 (cf. p. 40).

[179] F. Yamaguchi, C. Wressnegger, H. Gascon et K. Rieck. **Chucky : exposing missing checks in source code for vulnerability discovery**. Dans : *ACM Conference on Computer and Communications Security*. Sous la dir. d'A.-R. Sadeghi, V. D. Gligor et M. Yung. Berlin, Germany : ACM Press, novembre 2013, p. 499–510. doi : 10.1145/2508859.2516665 (cf. p. 45).

[180] L. K. Yan et H. Yin. **DroidScope : seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis**. Dans : *USENIX Security Symposium*. Bellevue, USA : USENIX Association, août 2012, p. 29 (cf. p. 39).

[181] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang et B. Zang. **Vetting undesirable behaviors in android apps with permission use analysis**. Dans : *ACM SIGSAC conference on Computer & communications security*. Berlin, Germany : ACM Press, novembre 2013, p. 611–622. doi : 10.1145/2508859.2516689 (cf. p. 39).

[182] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han et W. Zou. **SmartDroid : an automatic system for revealing UI-based trigger conditions in android applications**. Dans : *Second ACM workshop on Security and privacy in smartphones and mobile devices*. Raleigh, NC, USA : ACM Press, octobre 2012, p. 93–104. doi : 10.1145/2381934.2381950 (cf. p. 39, 40).

[183] Y. Zhou et X. Jiang. **Dissecting Android Malware : Characterization and Evolution**. Dans : *IEEE Symposium on Security and Privacy*. 4. San Jose, USA : IEEE Computer Society, mai 2012, p. 95–109. doi : 10.1109/SP.2012.16 (cf. p. 42, 43, 69).

[184] Y. Zhou, Z. Wang, W. Zhou et X. Jiang. **Hey , You , Get Off of My Market : Detecting Malicious Apps in Official and Alternative Android Markets**. Dans : *19th Annual Network and Distributed System Security Symposium*. 2. The internet society, février 2011 (cf. p. 39).

Chapitre 4

Protection des données personnelles

La protection des données personnelles est une thématique qui prend de l'importance à la fois dans la société civile et dans de nombreux domaines de la recherche en informatique. Les données des utilisateurs qui sont manipulées sur le web, dans les téléphones mobiles, représentent une source de revenus pour les éditeurs de logiciels et de services qui peuvent y accéder. Des entreprises de vente de données personnelles se développent, comme DataCoup [1]. Avec ces services, le revenu que l'on peut tirer des données personnelles est difficile à évaluer. Pour se donner une idée concrète, on peut adopter l'approche inverse : évaluer le prix que l'utilisateur est prêt à payer pour obtenir un service en échange du libre accès à ses données personnelles. Bauer *et al.* [191] montrent ainsi que 48% des utilisateurs de Facebook ne verseraient pas un euro pour conserver l'accès au service. Pour les autres, suivant la méthode d'analyse utilisée, les utilisateurs déclareraient accepter de verser de 5 à 15 euros. Il n'est donc pas encore clair de savoir s'il sera possible, dans un futur proche, de monnayer ses données personnelles tout en contrôlant leur diffusion et les traitements qu'elles subissent.

Les applications web de type réseau social sont les premières à manipuler de larges quantités de données personnelles. Elles se nourrissent à la fois des données que l'utilisateur consent à divulguer à des fins de partage mais croisent aussi les profils des utilisateurs avec leurs habitudes : historique de navigation, comportement de consommation, etc. Le deuxième lieu de stockage de nos données est sans doute nos ordinateurs personnels. Cependant, les logiciels qui y sont installés sont hétérogènes et les utilisateurs sont davantage sensibilisés à la collecte de données, qui semble difficile à automatiser sur ce type de systèmes. C'est pourquoi les téléphones mobiles représentent un meilleur vecteur pour collecter des données structurées et personnelles, que l'on peut ensuite croiser avec les données de réseaux sociaux. A la différence d'un ordinateur, un téléphone mobile comporte un *framework* unique de gestion des données : contacts, calendriers, localisation, comptes de services en ligne. Cette structuration facilite le développement d'applications accédant à ces données, au prix d'une plus grande facilité à faire fuiter ces informations.

Dans sa vie quotidienne, un utilisateur peut accéder à de nombreux services en ligne grâce à son téléphone mobile. D'un point de vue de la sécurité de ses données personnelles, les menaces proviennent alors de deux vecteurs : une attaque provenant d'un logiciel malveillant de son téléphone ou une attaque du service en ligne lui-même, au travers du réseau. En effet, le fournisseur du service, s'il est déloyal, peut chercher à récupérer plus d'informations que ce qu'il déclare collecter ou bien, il peut avoir été compromis par un attaquant qui cherche alors à attaquer tous les clients du fournisseur de service. Les façons dont un attaquant peut parvenir à de telles situations diffèrent, techniquement parlant. Pour compromettre le téléphone, on cherchera plutôt à attaquer les applications de celui-ci, afin de créer des logiciels malveillants que l'on cherchera à distribuer aux utilisateurs. Pour attaquer le fournisseur de service, il faut prendre le contrôle d'un des serveurs fournissant le service ou bien être capable de surveiller les interactions au sein du protocole liant le fournisseur de service à ses utilisateurs.

Dans ce chapitre, je présente les contributions qui permettent de protéger la vie privée de l'utilisateur dans deux scénarios différents : les attaques que l'on peut subir à cause de *malware* exploitant des canaux auxiliaires, et les attaques pouvant être orchestrées par un fournisseur de service en charge d'un réseau de transport cherchant à espionner ces utilisateurs.

Je commence par introduire la problématique générale de la sécurité pour les téléphones mobiles en donnant une vue interne de la sécurité d'un téléphone mobile (section 1). Je montre les interactions qui existent entre les composants matériels d'un téléphone et un fournisseur de service qui communique avec une application du

1. https://datacoup.com/

téléphone. J'explique en particulier les spécificités des éléments d'Android qui permettent d'assurer la sécurité du téléphone et de ses données. Ces éléments sont utiles pour la section qui suit où j'aborde les canaux auxiliaires qui permettent de contourner ces éléments de sécurité.

Je me penche ensuite sur la problématique des attaques basées sur des canaux auxiliaires (section 2). Ces attaques permettent de construire des canaux de communication afin de faire transiter des informations, par exemple des données personnelles, entre deux systèmes ou logiciels contrôlés par l'attaquant. Ce type d'attaque a été particulièrement étudié pour les protocoles réseaux, dans lesquels il devient possible de cacher des informations à des mécanismes de surveillance inspectant le protocole. Dans le cas des téléphones mobiles, je montre comment ces canaux auxiliaires peuvent être exploités pour contourner de sécurité d'Android, notamment les mécanismes de protection qui implémentent les politiques représentées par les permissions des applications. L'étude et la conception de nouvelles attaques de ce type permet de poser les bases à l'élaboration de contremesures.

Ainsi, je présente deux méthodes de détection de ces canaux (section 3). Une première méthode se base sur l'analyse et la corrélation de l'activité CPU des processus qui coopèrent au travers du canal auxiliaire. La seconde méthode propose une détection des anomalies de la consommation énergétique du système lorsque celui-ci héberge deux processus qui communiquent par un canal auxiliaire.

Enfin, je présente comment se protéger d'un fournisseur de service qui chercherait à surveiller les trajets de ses utilisateurs, dans le cas d'un système de transport où les utilisateurs se servent de leur téléphone mobile pour ouvrir les portiques (section 4). Ce cas d'utilisation très précis est en lien avec l'ANR Lyrics dans lequel nous avons travaillé sur la sécurité des services mobiles sans contact. L'idée du scénario est qu'un téléphone mobile peut servir à valider son titre de transport sur une borne NFC d'un portique d'accès au métro, train, etc. La difficulté dans la validation du titre de transport est de réussir à s'assurer que l'utilisateur possède bien un titre de transport valide, tout en évitant de permettre au fournisseur de service de pouvoir reconnaître cet utilisateur ou bien de pouvoir lier plusieurs trajets successifs. Pour résoudre ce problème, nous avons proposé un protocole de validation qui surpasse ceux de la littérature existante, au prix d'une certaine complexité calculatoire mais tout en préservant une efficacité de mise en œuvre.

Je termine le chapitre en décrivant brièvement les logiciels développés relatifs à ces contributions et je donne quelques perspectives de recherche.

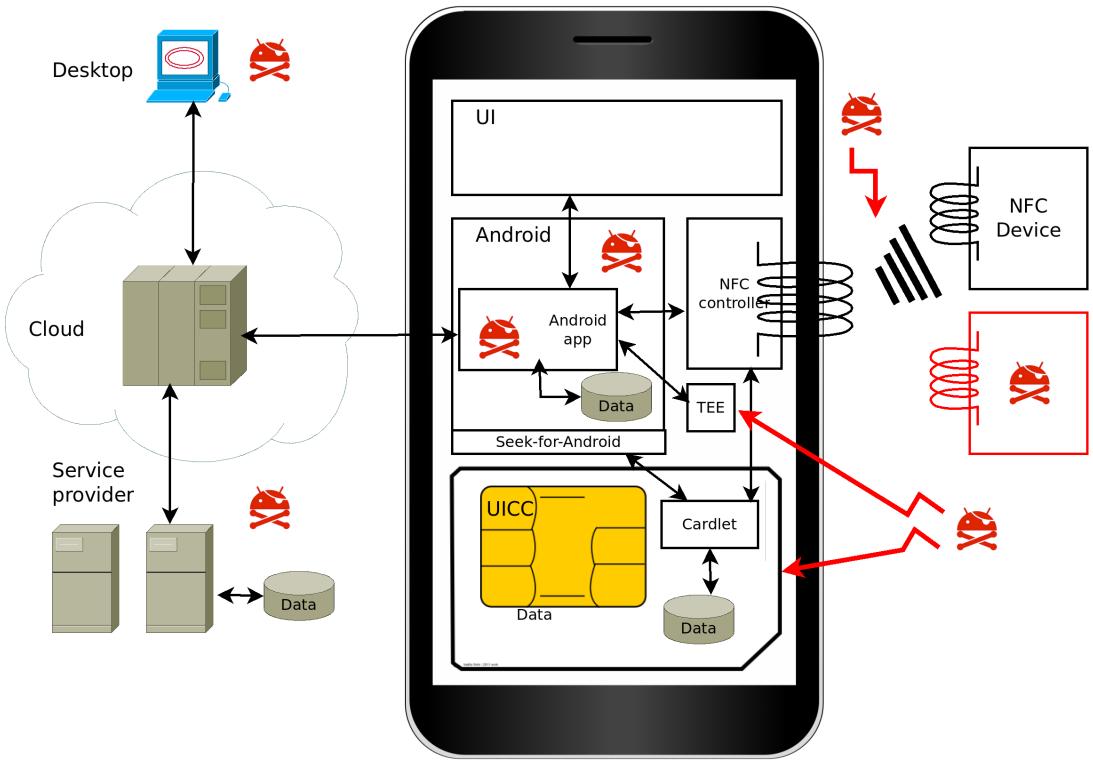


FIGURE 4.1 – Architecture globale d'un service mobile et possibilités d'attaque

1 Introduction aux architectures de services mobiles

Le développement d'applications mobiles est une tâche devenue complexe étant donné la diversité des architectures réseaux, des logiciels et des matériels qui sont impliqués. Dans la figure 4.1, nous donnons un aperçu d'une architecture typique pour une application d'une entreprise qui délivre un service à l'utilisateur. Dans cet aperçu, nous nous focalisons sur les composants internes à un téléphone mobile que l'on peut exploiter à des fins de sécurité et nous présentons leurs interactions avec l'extérieur.

Dans ce scénario global, une application Android tourne au-dessus du système d'exploitation Android et peut accéder à des données applicatives et informations personnelles de l'utilisateur. Cette application peut se connecter au fournisseur de service au travers d'internet et peut utiliser des technologies de Cloud, par exemple fournies par Google, pour être capable de gérer des millions d'utilisateurs. Ainsi, le fournisseur de service possède lui aussi des données personnelles relatives à l'utilisateur. Souvent, le fournisseur de service permet à l'utilisateur d'accéder à une version web de son application en utilisant un ordinateur standard. L'application peut éventuellement interagir en champ proche grâce à la technologie NFC avec des composants passifs ou actifs. Cela permet par exemple de construire des applications d'authentification interagissant avec une borne NFC et réalisant une vérification en ligne jusqu'au fournisseur de service [233, 219, 231]. Enfin, les derniers composants importants de ce scénario global sont les éléments sécurisés disponibles : il peut s'agir d'un UICC (*Universal Integrated Circuit Card*) c'est-à-dire une carte à puce permettant d'héberger plusieurs applications ou un TEE (*Trusted Execution Environment*) [210, 188]. Dans les deux cas, l'objectif est de permettre de réaliser des traitements avec un haut niveau de sécurité.

Dans cette architecture, les vecteurs d'attaque sont nombreux. Au niveau d'Android, on peut supposer l'application corrompue par l'attaquant, lui permettant d'avoir les mêmes privilèges que l'application. Si celui-ci a été capable de corrompre le système d'exploitation, l'intégralité des données du système lui sont accessibles. Pour des applications critiques comme le paiement bancaire ou l'authentification, ce vecteur d'attaque gêne l'émergence de ces types de service. L'utilisation d'un élément sécurisé tiers devient obligatoire : en s'appuyant sur le haut niveau de sécurité offert par un UICC ou TEE, on peut préserver l'intégrité du service embarqué et de ses données. Cependant, comme évoqué dans le chapitre 3, l'attaquant peut réaliser des attaques physiques contre l'UICC et donc aussi le TEE [170] puisque ces composants sont inaccessibles d'un point de vue logiciel. Même si ces attaques semblent difficiles à mettre en œuvre sans avoir un accès physique direct au téléphone

d'un utilisateur, cela reste possible sur un téléphone volé. De plus, de l'extérieur du téléphone, l'attaquant peut réaliser des attaques au niveau des communications avec une borne NFC ou réaliser une borne NFC pirate [226]. Dans ce cas, la sécurité des interactions repose sur la robustesse du protocole utilisé.

Du côté du fournisseur de service toutes les hypothèses sur les capacités de l'attaquant sont possibles. Le fournisseur peut avoir été attaqué par un tiers et voir ses services compromis, sans pour autant le savoir. Au pire, on peut même considérer le cas où le fournisseur de service est lui-même un attaquant, c'est-à-dire qu'il cherche à collecter les données personnelles à l'insu de l'utilisateur, par exemple sa localisation, tout en honorant le service qu'il fournit. Ce type d'attaquant dit "honnête mais curieux" est une hypothèse moins extrême que de supposer qu'un attaquant tiers a pénétré les systèmes du fournisseur de service. De plus, une telle hypothèse est maintenant prise en compte par les entreprises car la défiance des utilisateurs vis-à-vis des fournisseurs de service augmente considérablement [203]. Aussi, la capacité d'un fournisseur de service à mettre en avant son honnêteté peut avoir des incidences commerciales fortes. Elle devient donc un enjeu de communication majeur et toute révélation la contredisant apporte un préjudice économique.

1.1 Modèle de sécurité sous Android

La sécurité d'Android est basée sur la réutilisation des mécanismes de sécurité fournis par les distributions Linux [207]. Les objectifs du système sont les suivants :

— Empêcher les attaques contre le système ;
— Prévenir les escalades de privilèges.
— Cloisonner les applications ;
— Contrôler les accès des applications aux données et ressources.

A cette fin, Google utilise désormais SELinux comme politique de contrôle d'accès obligatoire pour protéger les services systèmes des applications utilisateurs. En effet, une application s'exécutera dans un contexte dont la partie objet sera *untrusted_app:process* alors que par exemple zygote (le processus qui lance toutes les applications) possède pour partie objet *zygote:process*. Ainsi, les applications utilisateur ne peuvent interagir avec les objets ou les services du système puisque la politique appliquée à un sujet *untrusted_app:process* ne le permet pas. Par ailleurs, une escalade de privilège limiterait la portée de l'attaquant puisque seule la politique appliquée au contexte courant serait obtenue. Ainsi, avant l'intégration de SELinux, il était possible de compromettre tout le système en exploitant une vulnérabilité [202].

Malgré l'utilisation de SELinux, le cloisonnement des applications est réalisée avec une politique DAC. En effet, si l'on devait utiliser une politique SELinux pour cloisonner chaque application, il faudrait reconfigurer la politique dynamiquement puisque l'on ne sait pas à l'avance quelles applications seront installées. Android détourne l'utilisation normale des UID sous Linux, réservés à la gestion des utilisateurs, pour cloisonner les applications. Chaque application et chaque objet manipulé par l'application possède un UID différent, démarrant à 10 000. A l'inverse, le services systèmes ont des UID démarrant à 1 000 et quelques démons tournent sous l'UID 0. Ainsi, la non interférence sur les fichiers utilisateurs est gérée en empêchant la lecture pour le groupe ou *others*. Concernant l'accès aux ressources, Android utilise encore une fois les permissions DAC. L'accès au réseau est contrôlé par l'ajout ou non au groupe *inet*. Ainsi, on bénéficie de la sécurité assurée par le noyau en réutilisant les mécanismes Unix de contrôle des accès. L'ensemble du contrôle de l'accès aux ressources est la plupart du temps encodé avec des groupes Unix, mais certaines vérifications sont parfois faites au niveau de la machine virtuelle qui exécute le *bytecode de l'application*.

L'ensemble des autorisations que l'on peut appliquer à une application est exprimée pour chaque application dans son *Manifest*. Dans le monde Android, on appelle cette politique les permissions de l'application. Sans vouloir décrire toutes les permissions, quelques exemples importants pour la suite sont à noter. La permission *INTERNET* donne un accès complet au réseau, qu'il soit local ou qu'il sorte vers le réseau Internet. La permission *READ_CONTACT* autorise l'application à lire des informations de contact, quelle que soit la source de ces contacts (SIM, compte Google ou tout autre application qui fournit des contacts). D'autres permissions expriment des autorisations pour lire ou écrire des informations personnelles. Certaines permissions autorisent des opérations d'administration : installer des applications, gérer les processus, etc. Enfin, on peut aussi créer ses propres permissions afin de contrôler la coopération entre deux applications [232].

A partir de la version 6 d'Android, la déclaration dans le Manifest d'une permission de niveau *dangerous* ne suffit plus pour l'obtenir : l'utilisateur doit explicitement accorder la permission dans les réglages de l'application. S'il ne le fait pas, l'application ne l'obtient pas et doit donc ajouter du code en conséquence pour prévenir l'utilisateur. Ce retour à une sorte de politique discrétionnaire permet de limiter les abus, c'est-à-dire d'empêcher des applications de demander un trop large ensemble de permissions et d'abuser de cette largesse pour récupérer

Table 4.1 – Vecteurs et types d'attaques

	Vecteur d'attaque	Confidentialité	Intégrité	Disponibilité
Attaquant tiers	APK	Ransomware Spyware	Adware Data Eraser	Phone blocker
	Système d'exploitation			Remote Administration Tool
Fournisseur de service honnête et curieux	APK	Espionnage		
	Protocole	Espionnage		

des informations. Avant que cette possibilité n'existe dans Android, Bartel *et al.* [190] et nous-même [193] avons proposé une méthode pour modifier les applications afin de contrôler l'accès aux données de l'utilisateur puisque le système ne le permettait pas. Désormais, l'utilisateur peut contrôler lui-même ces accès dans la configuration système de l'application.

1.2 La communication sous Android

Quelques précisions sont données dans cette partie sur les moyens de communiquer pour les logiciels embarqués sur un téléphone mobile Android.

Au sein du système Android, Google a défini un moyen de faciliter et contraindre les communications entre les applications. Il s'agit de la classe *Intent* qui contient un type d'action, une catégorie et des données. Ces *Intents* permettent de faire coopérer des applications ou bien d'envoyer des messages d'une application à l'autre. Cette classe est tellement fondamentale qu'elle est le support de nombreux contenus classiques : emails, SMS, appels téléphoniques, hyperliens. Pour la sécurité, cela devient un point faible. Certains *malware* capturent les SMS en filtrant les *Intents* appropriés et annulent la propagation de l'*Intent* en question aux autres applications de messagerie. Certaines attaques qui inondent le système de certains types d'*Intents* arrivent à faire redémarrer le téléphone [216].

Comme montré en figure 4.1, d'autres moyens de communication sont disponibles. Une application peut interagir avec un élément sécurisé comme un TEE ou une carte SIM. Dans ce cas, nous faisons l'hypothèse que le programme est déjà provisionné dans la carte ou le TEE. Un programme qui coopère avec une *cardlet* dans une carte SIM doit utiliser des APDUs, comme un terminal bancaire qui authentifie un utilisateur. Cela rend plus difficile le développement d'applications de ce type, notamment à cause des contraintes de taille et de bande passante de ce type de communication. La carte SIM peut aussi être interrogée avec ce même protocole depuis l'extérieur du téléphone depuis un système équipé de NFC. Dans ce cas, les APDUs sont routés directement au travers du contrôleur NFC vers la SIM, sans qu'il y ait besoin de faire travailler le système d'exploitation. La communication est même possible si le téléphone est éteint ou n'a plus d'énergie grâce au champ électromagnétique qui alimente l'antenne NFC et par rebond, la SIM. Par ailleurs, la SIM n'est pas forcément un élément passif que l'on interroge depuis l'extérieur : elle peut interroger un tag ou un système NFC extérieur au téléphone. Le support du NFC autorise donc les interrogations dans tous les sens possibles. Une application Android peut elle aussi être interrogée ou interroger en NFC mais il existe pour l'instant peu d'applications commerciales déployées utilisant de tels protocoles et les industriels se limitent souvent à l'interrogation de tags passifs afin de capturer une information statique depuis l'extérieur du téléphone afin de paramétrer l'application associée dans le téléphone. Enfin, le fonctionnement des communications opérées par un TEE est encore plus difficile à décrire, étant donné que la technologie n'est pas encore mature et déployée massivement. Le principe général est de pouvoir partager des zones mémoires (ou bien les synchroniser) entre le système d'exploitation du TEE et Android [188].

1.3 Hypothèses d'attaque

Pour contourner les mécanismes de sécurité implantés dans Android, les attaquants abusent la plupart du temps des permissions demandées. Ainsi, des attaquants tiers ou des fournisseurs de service malhonnêtes n'ont pas besoin d'obtenir des privilèges *root* pour attaquer un téléphone mobile. Comme montré en table 4.1, l'utilisation d'un APK demandant les permissions appropriées permet de réaliser différents types de *malware* [183]. Nous avons exploré l'ensemble de ces catégories de *malware* pour comprendre leur fonctionnement interne [140]. Par exemple, un *malware* du type *Phone blocker* peut tout à fait rendre le téléphone inutilisable en lançant régulièrement une application plein écran empêchant l'utilisateur de se rendre dans un menu. De plus, il peut tuer certaines applications cibles (s'il possède la permission pour cela), afin par exemple d'empêcher de le désinstaller ou d'installer un antivirus. Si le *malware* compromet le système d'exploitation, il peut bien évidemment réaliser

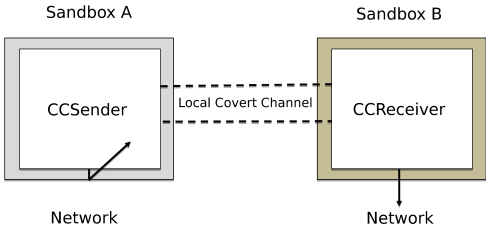


FIGURE 4.2 – Canal auxiliaire entre deux applications locales

les mêmes attaques qu'avec un APK malveillant. Cependant, bien souvent, ce type d'attaque sert à installer un RAT (*Remote Administration Tool*) pour réaliser d'autres tâches ultérieures : installation d'autres *malware*, attaques distribuées, etc. Pour l'attaquant tiers, la difficulté pour réaliser l'attaque est d'arriver à installer un APK contenant la charge sur le téléphone de l'utilisateur. Le vecteur préféré des attaquants est de *repackager* des applications du *Google Play Store* et de les diffuser sur des *markets* alternatifs qui comportent 5 à 10% d'applications repackagées [246].

Si l'attaque ne provient pas d'un tiers mais d'un fournisseur de service honnête mais curieux, il est raisonnable de considérer que les attaques ciblent les données personnelles de l'utilisateur. En effet, le fournisseur de service prendrait trop de risque pour d'autres types d'attaques. Deux moyens sont à sa disposition : comme l'attaquant tiers, il peut utiliser l'APK de son application pour parvenir à ses fins. En effet, il maitrise totalement le code de son application et a donc toutes les facilités pour implanter des mécanismes de surveillance à l'encontre de l'utilisateur. Cependant, il prend le risque d'être découvert si les utilisateurs analysent le *bytecode* de l'application. Si le fournisseur de service souhaite rester discret, il peut concevoir un protocole entre ses serveurs et l'application lui permettant d'inférer des informations intéressantes. L'attaque considérée relève donc plus du détournement du protocole que d'une attaque directe via un logiciel. Dans tous les cas, les données collectées peuvent être vendues par le fournisseur de service ou bien lui servent à améliorer sa stratégie marketing.

Dans la suite, nous allons nous intéresser à un type d'attaque particulier : les canaux auxiliaires. Le but de ces attaques est de contourner les mécanismes de cloisonnement mis en place par Android afin de faire fuiter de l'information. Ces mécanismes permettent d'envisager l'émergence de nouveaux types de *ransomware* ou *spyware*, dont les actions deviendraient alors difficiles à détecter.

2 Attaques par canaux auxiliaires

Les attaques par canaux auxiliaires (*covert channel*) sont des transferts d'informations entre deux entités qui n'utilisent pas un canal de communication normal [222]. A la différence des attaques par canaux cachés (*side channel*) qui permettent de capturer des informations induites par le fonctionnement du système [208], un canal auxiliaire est créé délibérément par l'attaquant pour faire communiquer deux parties du système. L'enjeu d'un canal auxiliaire est de réussir à contourner les sécurités des systèmes impliqués et de rester indétectable pendant que les deux systèmes communiquent. Leur utilisation peut permettre par exemple d'échapper à la censure [244] mais sert plutôt à des fins malveillantes, par exemple pour contrôler un ensemble de *botnets* [223].

Les systèmes concernés par les canaux auxiliaires sont nombreux et ont été étudiés au fil du temps dans de nombreuses publications. Les systèmes concernés peuvent être des systèmes locaux (sans réseau) [243], reliés par un réseau TCP IPv4 [235] ou IPv6 [225]. Il peut aussi s'agir de systèmes plus exotiques comme les *workflow* de processus métiers [185] où des informations peuvent fuiter à cause d'un défaut de conception.

2.1 Les canaux auxiliaires sous Android

Le premier travail sur canaux auxiliaires sous Android a été proposé par Schlegel *et al.*. Les auteurs ont développé Soundcomber [238], un *malware* qui peut, par exemple, capturer les touches numériques d'un smartphone en utilisant le son de celui-ci. Soundcomber implémente plusieurs types de canaux auxiliaires : l'utilisation des applications, par exemple le navigateur web, les fonctions du téléphone comme le paramètre de vibration ou de volume, et l'état de l'écran.

TABLE 4.2 – Canal de contrôle/données et permissions pour nos canaux auxiliaires

COVERT CHANNEL TYPE	CONTROL CHANNEL	DATA CHANNEL	REQUIRED PERMISSION
CC#1 : Task list/screen	screen state	task list	GET_TASK
CC#2 : Process prio./screen	screen state	process priority	
CC#3 : Process priorities		process priority	
CC#4 : Pure screen-based		screen based	WAKE_LOCK

En 2012, Marforio *et al.* présentent une étude bien plus exhaustive en brossant l'ensemble des possibilités de canaux cachés dans un téléphone Android [227]. Ils présentent comment deux applications peuvent communiquer au sein d'un téléphone en utilisant plusieurs types de canaux auxiliaires. Le principe de l'architecture utilisée est montrée en figure 4.2 : une application (*CCSender*) ne possède pas le droit de joindre le réseau alors qu'une autre le peut (*CCReceiver*). En coopérant au travers d'un canal auxiliaire, *CCSender* peut réussir à faire fuiter des informations collectées dans le téléphone en s'appuyant sur l'accès au réseau de *CCReceiver*. De surcroit, *CCReceiver* peut utiliser des techniques pour masquer la communication vers l'extérieur en utilisant des canaux auxiliaires orientés réseau ou du chiffrement. Marforio *et al.* explorent les différents types de canaux auxiliaires qui peuvent être utilisés. Il peut s'agir de déduire des informations de la signalisation réseau, par exemple le type des *Intents* qui sont transmis ou le fait qu'un port soit ouvert ou non. Il peut s'agir de métriques systèmes, par exemple l'espace disque ou l'espace mémoire disponible, le nombre de *threads* qui sont lancés par un processus, les informations disponibles dans */proc/stat*. Enfin, Marforio *et al.* expliquent comment s'appuyer sur le matériel pour construire un canal. En mesurant le temps pour réaliser une tâche, on peut déduire la charge du processeur. Ainsi, l'émetteur du message peut encoder une information en chargeant artificiellement celui-ci. Cependant, ce type de canal peut être plus difficile à gérer en fonction du matériel sur lequel il s'exécute et suivant la configuration du noyau qui peut avoir différentes stratégies d'ordonnancement des tâches.

Marforio *et al.* s'intéressent aux performances des différents types de canaux cachés en terme de quantité de données transmises et de temps de synchronisation. La synchronisation est le temps requis par la partie du protocole qui permet de démarrer une transmission : les deux applications doivent se mettre d'accord sur la date la transmission démarre, ce qui prend un certain temps. Ce temps est en général inférieur à $500\,ms$ mais dépasse $50\,s$ pour les canaux auxiliaires basés sur la charge du processeur par exemple. Les quantités de données transmises varient de $0.47\,bps$ à $4324\,bps$. La meilleure bande passante est obtenue pour les types d'*Intents* ou un Galaxy S transmet $4324\,bps$ et un Nexus One $3350\,bps$.

2.2 Conception de canaux auxiliaires discrets

Marforio *et al.* [227] ne prennent pas en compte la discrétion du canal auxiliaire dans leur évaluation. En fait, la capacité du canal à être discret est directement liée aux méthodes de détection que l'on peut mettre en œuvre, comme discuté plus tard en section 3. Typiquement, en exploitant un canal auxiliaire à sa bande passante maximum on induit une charge excessive du processeur. Dans certains cas, on peut même provoquer un *reboot* du système par inadvertance : le *flood* d'*Intents* est utilisé comme déni de service dans certaines attaques [216].

De plus, nous pensons que les *malware* modernes exploiteront leur capacité à se déclencher plus tard, comme expliqué en chapitre 3 section 2.1.2, pour échapper à d'éventuelles contremesures. Par ailleurs, si l'utilisateur constate un comportement suspect, par exemple le ralentissement de son système, la surcharge de son disque, il pourrait éventuellement suspecter une utilisation frauduleuse des ressources.

2.2.1 Principes

Nous avons proposé dans [221] quatre canaux auxiliaires dont le but est double : se focaliser sur la discrétion vis-à-vis de l'utilisateur et minimiser les permissions requises. En effet, pour les canaux auxiliaires de la littérature, certaines permissions sont souvent nécessaires. Si la permission demandée à l'utilisateur est suspicieuse, par exemple, l'accès à des paramètres de volume (*MODIFY_AUDIO_SETTINGS*) pour une application de calendrier, alors l'utilisateur peut suspecter un abus. Ainsi, le canal auxiliaire idéal est un canal qui ne requiert aucune permission. En table 4.2, nous détaillons les méthodes utilisées pour chaque type de canal auxiliaire (CC pour *Covert Channel*). Dans les deux premiers cas, nous utilisons l'extinction de l'écran comme évènement de synchronisation : à la différence de Marforio *et al.* qui développent un protocole à part entière pour la synchronisation de la transmission, nous basons cette synchronisation sur le fait que l'utilisateur quitte la session de son

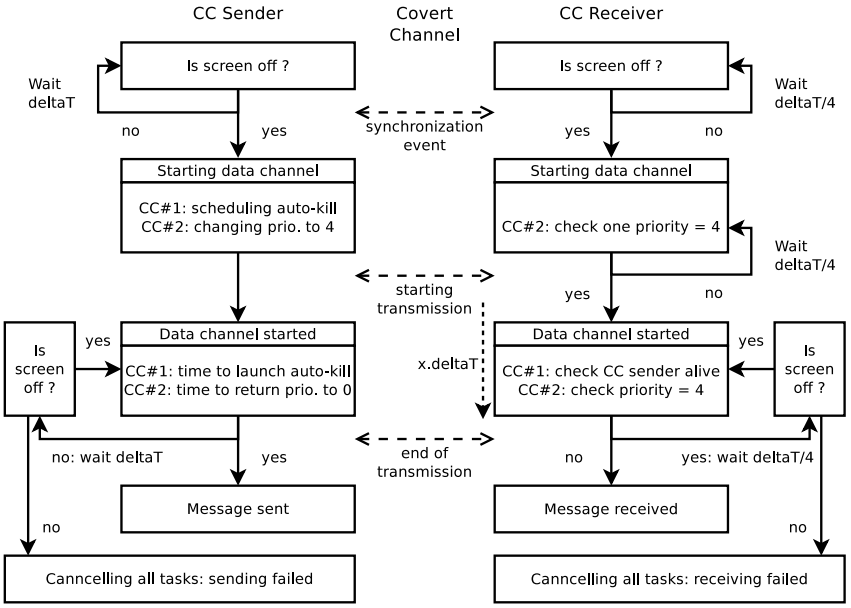


FIGURE 4.3 – Workflow de fonctionnement pour les canaux auxiliaires CC#1 et CC#2

téléphone et que son écran s'éteint. Ensuite, une fois le transfert débuté, nous utilisons plusieurs méthodes pour la transmission des données :

- la liste des applications actives (nécessite une permission pour CCReceiver) ;
- la priorité des processus ;
- l'état de l'écran (nécessite une permission pour CCSender).

Le principe général est identique quel que soit le canal de données utilisé. Comme expliqué en figure 4.3, à partir de l'instant de synchronisation (l'extinction de l'écran), le CCSender modifie un élément du système, puis patiente un temps $x * \Delta T$ pour encoder x. ΔT est un paramètre fixé à l'avance qui permet de faire varier la bande passante et la qualité de la transmission. Pour CC#1, le CCSender doit rester en avant plan pendant la transmission, puis, au bout d'un temps $x * \Delta T$ l'application se tue. Il en résulte que le CCReceiver voit cette tâche disparaitre dans la liste des tâches actives. Pour CC#2 et CC#3, CCSender modifie la priorité de son processus à une valeur inhabituelle pendant un temps $x * \Delta T$. CCReceiver peut alors scanner l'ensemble des numéros de processus possibles et récupérer leur priorité afin de détecter la valeur de priorité inhabituelle. Enfin CC#4 n'utilise que l'écran en jouant avec l'extinction automatique de celui-ci : il suffit de compter le temps entre deux allumages/extinctions pour retrouver x.

2.2.2 Consommation énergétique

Un autre objectif de cette étude est de minimiser l'impact sur la consommation énergétique. En effet, il n'y a rien de plus suspicieux qu'une application qui vide la batterie d'un téléphone. Nous avons mesuré la consommation énergétique de ces différents canaux auxiliaires, comme reporté dans la figure 4.4. Nous avons constaté que la méthode basée sur les priorités est très énergivore : en effet, le scan de l'ensemble des processus pour attendre la synchronisation est très coûteux en temps processeur. A l'inverse, la méthode CC#2 est plus efficace car la synchronisation se fait grâce à l'extinction de l'écran et le scan de priorité trouve très vite le processus cible. Cette différence montre bien qu'avec une combinaison de techniques, on peut significativement réduire la consommation énergétique d'un canal auxiliaire. Cet aspect des choses, la consommation énergétique du canal auxiliaire, est important à noter et à garder en mémoire, car nous évoquerons en section 3.4 une méthode de détection basée sur la mesure de la consommation d'énergie du système.

2.2.3 Performances

La mesure de la bande passante de CC#1 et CC#2 dépend directement des interactions de l'utilisateur. Si celui-ci utilise son smartphone, aucune transmission ne s'opère. Nous avons donc simulé un faux utilisateur qui

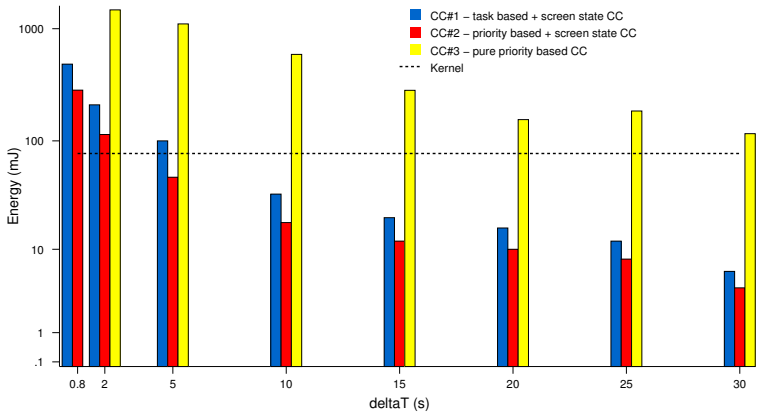


FIGURE 4.4 – Énergie consommée pendant une période d'une minute de transmission

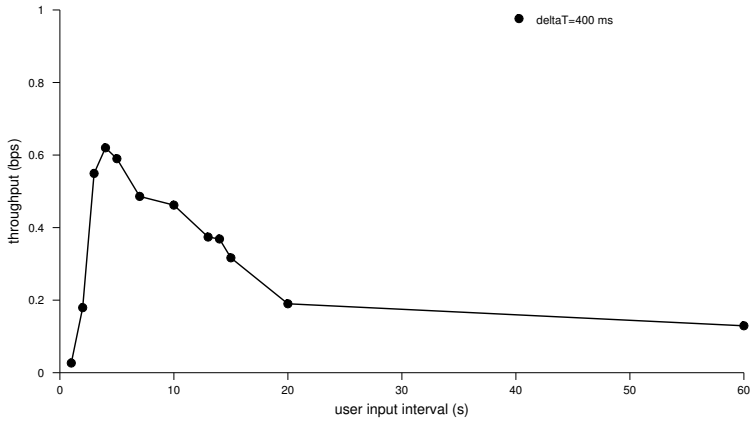


FIGURE 4.5 – Bande passante de CC#2 en fonction des interactions utilisateur et pour $\Delta T = 400\,ms$

réveille son smartphone à intervalles réguliers ΔU. La figure 4.5 montre la bande passante obtenue pendant une expérience de 3h20 pour CC#2 (les résultats pour CC#1 sont très similaires). Si ΔU est faible, il interrompt fréquemment la communication et fait donc chuter la bande passante. A l'inverse, si l'utilisateur ne réveille son smartphone que peu fréquemment, il espace l'évènement de synchronisation qu'attendent CCReceiver et CCSender, à savoir l'extinction de l'écran : la bande passante moyenne chute. On observe sur la figure 4.5 une bande passante maximale pour $\Delta U = 4\,s$.

Il faut noter que la bande passante optimale reste cependant très faible pour cette expérience : environ $0.6\,bps$, ce qui est très loin des résultats de Marforio *et al.* Cependant, notre objectif était de travailler sur la discrétion du canal auxiliaire et sa faible consommation énergétique ce qui explique ces différences. De plus, l'ensemble des symboles transmis (l'ensemble des lettres minuscules et majuscules) fait baisser la bande passante. Avec un jeu de symboles plus restreint ou en compressant ces données, on obtiendrait de meilleurs résultats.

Pour CC#3, avec $\Delta T = 800\,ms$, la bande passante mesurée est de $17.6\,bps$, ce qui est bien supérieur au deux canaux auxiliaires précédents. Il est possible d'augmenter ce débit en décrémentant ΔT mais des problèmes de concurrence apparaissent si le scan de priorités est trop fréquent, provoquant des arrêts de CCReceiver. Concernant CC#4, la bande passante mesurée est de $0.22\,ms$ avec $\Delta T = 800\,ms$, ce qui est faible mais s'explique par le délai d'attente pour que l'écran s'éteigne automatiquement.

Dans la suite de ce chapitre, nous explorons deux types de contremesures contre les attaques provenant d'un tiers (section 3) ou du fournisseur de service (section 4). Pour faire suite à la description des attaques par canaux auxiliaires, nous présentons donc naturellement les deux méthodes de détection de ces canaux.

3 Détection de canaux auxiliaires

Pour ne pas subir d'attaques par canal auxiliaire, l'idéal est de concevoir un système ne comportant aucun moyen de construire un tel canal. Un des premiers travaux à proposer une méthode de conception permettant de réduire les possibles canaux auxiliaires est l'utilisation de la *Shared Resource Matrix Methodology* [217]. Dans ce travail de 1983, Kemmerer propose d'identifier dans une matrice les permissions entre les objets du systèmes (processus et fichiers) ainsi que les primitives permettant de réaliser des opérations, puis d'analyser la matrice en croisant permissions et primitives afin de déterminer si une fuite d'information peut se produire. Cette approche a été plus tard étendue pour automatiser le croisement entre permissions et primitives [234]. En 1993, Kang *et al.* proposent *The Pump*, un *buffer* de message permettant de faire communiquer deux entités et de diviser par n (la taille de ce buffer) le débit des canaux auxiliaires basés sur le temps. Toutes ces approches ne peuvent pas totalement éradiquer la présence d'un canal auxiliaire. Il faut donc alors se tourner vers des solutions qui tentent de les détecter plutôt que de les prévenir.

Les travaux de recherche qui ont suivi se sont donc intéressés aux canaux auxiliaires purement réseaux, en particulier les méthodes manipulant les entêtes de paquets et les intervalles de temps entre les paquets (*timing channels*) [244] alors que les premiers travaux se concentraient sur les fuites dues à la manipulation de fichiers (*storage channels*). Le principe des *timing channels* est simple : le processus qui envoie les données fait varier l'intervalle de temps entre les paquets afin d'encoder des 0 et des 1. Pour détecter de tels canaux, Berk *et al.* [192] proposent par exemple d'analyser statistiquement ces délais inter paquets : dans une communication normale, ils doivent être centrés sur une valeur alors que ce n'est pas le cas si un canal auxiliaire est utilisé. Ce genre de méthode nécessite donc une surveillance temps réel des paquets réseaux ce qui a un coût non négligeable. Pour le cas des canaux auxiliaires locaux à un téléphone, ces méthodes ont donc peu d'intérêt.

Nous avons proposé deux méthodes de détection basées sur la collecte de données au niveau du noyau Android. Ces méthodes sont le fruit d'une collaboration internationale avec W. Mazurczyk (Warsaw University of Technology), L. Caviglione et M. Gaggero (National Research Council of Italy) et ont mené à plusieurs publications [197, 242, 198]. Dans un cas, le principe consiste à mesurer les activités des processus au niveau noyau afin de repérer une corrélation forte entre deux processus. Dans l'autre cas, nous proposons de détecter un canal auxiliaire en analysant l'énergie consommée par l'ensemble du système et en détectant une anomalie dans la prédiction du comportement de consommation normale de cette énergie. Dans la suite de cette section, nous décrivons la problématique de la détection des canaux auxiliaires dans les téléphones mobiles, avant de proposer deux nouvelles méthodes dont nous évaluons l'efficacité expérimentalement.

3.1 La détection de canaux auxiliaires pour les téléphones mobiles

La corrélation de données du système pour détecter une attaque est une méthode classique pour réaliser de la détection d'intrusion ou d'anomalie. Le principe est d'arriver à différencier des données nominales par rapport à une anomalie dans ces données. De nombreuses méthodes ont été proposées pour la détection d'autres types d'attaque, par exemple une intrusion réseau, la détection de *malware*, mais ces méthodes ne peuvent s'appliquer à des canaux auxiliaires. Ainsi, peu de méthodes de détection sont dédiées aux canaux auxiliaires locaux.

Un des seuls travaux dédiés à cette question a été porté par Hansen *et al.* [211] qui proposent une méthode pour détecter trois types de canaux auxiliaires : ceux basés sur la vibration, le volume et la capacité de réveiller l'écran. La méthode proposée se base sur le décompte des évènements et par la levée d'une alarme si un nombre trop important d'évènements est observé pendant une fenêtre temporelle glissante. La difficulté pour utiliser la méthode réside dans le seuil choisi, auquel un *malware* pourrait s'adapter pour ne pas être détecté.

Pourtant, dans leur travail initial sur les canaux auxiliaires, Marforio *et al.* [227] évoquent plusieurs pistes de recherche qui semblent intéressantes. Une première piste consiste à limiter au maximum l'accès des applications aux APIs quand celles-ci ne sont pas nécessaires. En effet, comme montré avec le canal basé sur les priorités des processus, la plupart des applications n'ont pas besoin de cette information. Si l'accès à ces APIs persistent, des permissions doivent être créées pour en restreindre l'accès. Nous avons testé cette idée en utilisant la notion de valeurs barrières [221]. Dans le canal auxiliaire basé sur les priorités (CC#2 et CC#3), nous renvoyons une valeur erronée avec une probabilité de $\frac{bs}{1000+bs}$. Le paramètre bs (*barrier size*) contrôle la fréquence pour laquelle on cherche à perturber la lecture des priorités. Expérimentalement, avec $bs < 160$, on obtient 1 octet faux sur 5 environ. Quand bs augmente à plus de 200, on s'approche d'un message perturbé à 50%. Évidemment, le problème d'une telle méthode est que l'on risque de perturber le bon fonctionnement des applications ayant besoin des bonnes valeurs de priorité. Pour d'autre types de canaux auxiliaires comme ceux basés sur l'allumage de l'écran ou le volume de la sonnerie, une telle méthode serait difficile à mettre en œuvre.

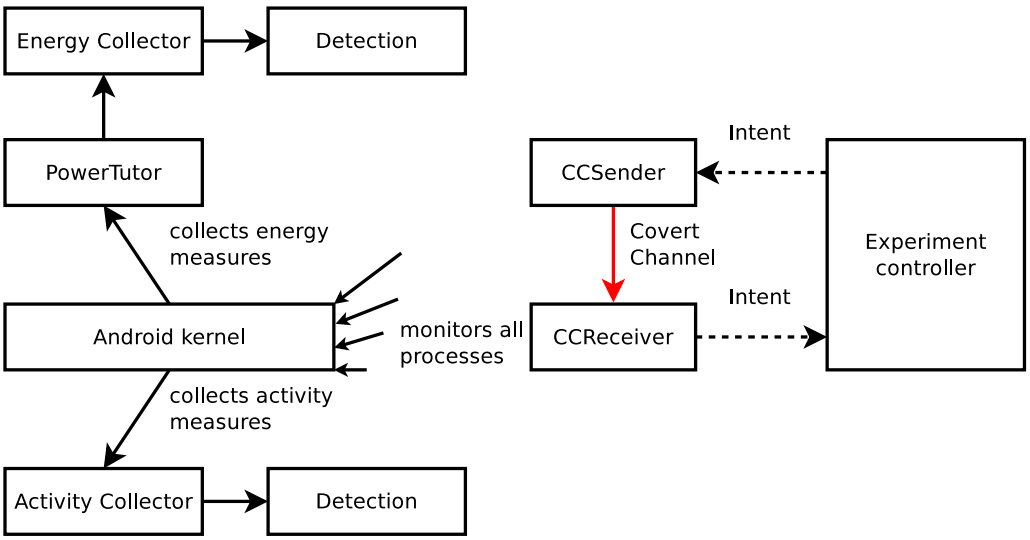


FIGURE 4.6 – Architecture de collecte de données

De même, Marforio *et al.* mentionnent que la limitation de la programmation d'applications pouvant s'exécuter en concurrence permet de réduire ce type d'attaques. L'évolution des processeurs multi cœurs et la programmation fréquentes de tâches asynchrones ne va pas dans ce sens. Une seconde piste qu'il faut mentionner consiste à restreindre l'accès aux mécanismes de programmation bas niveau comme les *sockets*, *fork* de processus, etc. Ces mécanismes n'ont pas de raison d'être pour des applications Java classiques mais restent cependant accessibles pour des applications ayant des parties natives. Ces mécanismes favorisent l'émergence d'attaques par canaux auxiliaires. Enfin, les auteurs évoquent la possibilité de ne pas remonter au niveau applicatif des informations précises, notamment sur le temps qui s'écoule, afin de perturber les transmissions par canaux auxiliaires. En effet, nous avons observé expérimentalement que lorsqu'un téléphone mobile est en mode *idle*, son noyau entre dans un mode de fonctionnement permettant d'économiser de l'énergie au prix d'une précision plus faible des évènements programmés à des dates précises. On peut observer plusieurs secondes de décalage dans le déclenchement de l'évènement lorsque le téléphone est dans ce mode, ce qui perturbe considérablement les programmes réalisant la transmission sur un canal auxiliaire. Cet effet de bord, dont le but initial est d'économiser la batterie, pourrait être utilisé pour contrecarrer des transmissions illicites.

Globalement, il n'y a donc que peu de méthodes existantes dans la littérature pour traiter le problème des canaux auxiliaires sur téléphone mobile. Nous avons donc travaillé sur ce problème avec une approche expérimentale : nous avons collecté des données à partir d'implémentations de canaux auxiliaires, comme ceux évoqués en section 2.2. A partir de ces données, nous avons cherché à réaliser des méthodes de détection dont les calculs sont raisonnables pour la puissance d'un téléphone mobile.

3.2 Architecture de collecte de données

Notre plate-forme expérimentale est représentée en figure 4.6. Nous collectons deux types de données au niveau du noyau Android : les activités des processus et l'énergie consommée par l'ensemble du système. L'activité des processus est mesuré dans les fichiers */proc/[pid]/stat* dans lesquels sont collectés les temps CPU alloués à chaque processus depuis son lancement. L'énergie est mesurée en utilisant PowerTutor [245], comme présenté par Hoffmann *et al.* [213]. PowerTutor est un outil qui calcule l'énergie consommée à partir d'une modélisation du matériel et les valeurs logiciels d'utilisation de ce matériel (consommation réseau, luminosité de l'écran, état du CPU, etc.). Nous avons automatisé le contrôle de l'envoi d'un message utilisant une des implémentation de canaux auxiliaires. Pour ce faire, nous enregistrons les valeurs discutées précédemment et nous déclenchons l'émission d'un message de longueur et contenu aléatoires à une date elle aussi aléatoire (quelques secondes à quelques minutes après le début de la mesure). Puis, nous attendons un temps assez grand et nous mettons fin à la collecte de mesures. Ce procédé a été répété de nombreuses fois afin de générer un grand ensemble de données. Nous avons aussi réalisé une campagne identique sans qu'aucune transmission n'ait lieu afin d'avoir des données témoins représentant l'absence de transmission par canal auxiliaire.

TABLE 4.3 – Exemple de calcul de $T_C(x)$ pour les processus a, b, c pendant 5 mesures consécutives

MEASURE	PROCESS A	PROCESS B	PROCESS C
M_1	Active	Active	Not active
M_2	Not active	Active	Not active
M_3	Active	Active	Active
M_4	Not active	Active	Active
M_5	Active	Active	Active
$T_C(x)$	$T_C(a) = 3$	$T_C(b) = 5$	$T_C(c) = 3$

TABLE 4.4 – Exemple de calcul du facteur d'activité $A_F(x, y)$ pour les process a, b et c

PAIR	ACTIVITY	$A_F(x, y)$ [%]
a, b	Active during 1, 3, 5	$A_F(a, b) = \frac{3}{(3+5-3)} \cdot 100 = 60$
a, c	Active during 3, 5	$A_F(a, c) = \frac{2}{(3+3-2)} \cdot 100 = 50$
b, c	Active during 3, 4, 5	$A_F(b, c) = \frac{3}{(3+5-3)} \cdot 100 = 60$

Les types de canaux auxiliaires que nous avons implémentés sont les suivants :

— Type d'*Intent* [227] : les données sont encodées dans le type d'*Intent* utilisé et non pas dans les données de l'*Intent* lui-même.

— Paramètre de vibration ou de volume [238] : comme proposé dans Soundcomber [238], le canal auxiliaire est créé en modifiant un paramètre du téléphone ; ici plusieurs paramètres de sonnerie sont utilisés (volume pour les messages, les appels, etc.) ;

— Verrou sur des fichiers [238] : le principe est de poser un verrou sur le même fichier : si le receveur du message n'y parvient pas, il infère un 1, sinon un 0.

— Charge du système [227] : l'envoi d'un 1 est réalisé par l'envoyeur en chargeant le CPU pendant un cours instant. Pour lire la donnée, le receveur compte combien l'envoyeur a reçu de temps processeur pendant un intervalle de temps. Si cette valeur dépasse un certain seuil, il infère 1, sinon 0.

— Découverte de Socket Unix [227] : les données sont encodées dans l'état de la *socket* : 1 quand elle est ouverte, 0 sinon.

A cette occasion, nous avons proposé deux implémentation supplémentaires de canaux auxiliaires :

— Taille des fichiers : l'émetteur envoie des données en modifiant la taille d'un fichier et à l'inverse, le récepteur qui lit les données déduit l'information de la taille.

— Charge de la mémoire : l'émetteur mesure initialement la charge de la mémoire RAM, puis la surveille. Pour émettre un 1, le processus émetteur alloue des données dans la mémoire et les désalloue pour émettre un 0.

3.3 Détection de canaux auxiliaires basée sur l'activité des processus

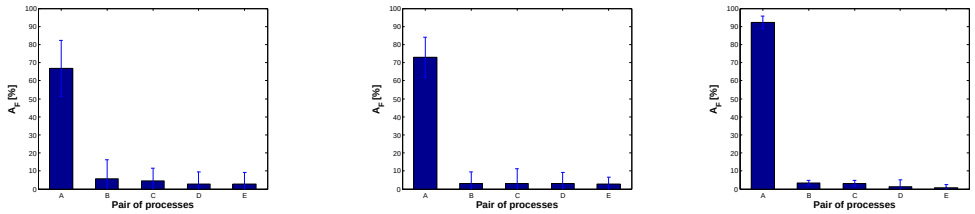
Pour détecter la présence d'un canal auxiliaire, nous avons proposé une méthode corrélant l'activité des processus [242]. Intuitivement, deux processus Android, notamment deux applications de l'utilisateur, ne doivent pas avoir une activité corrélée dans une longue période de temps, à moins qu'ils ne soient en train de réaliser une tâche ensemble. Pour calculer cette corrélation, nous utilisons les données précédemment collectées et nous avons introduit trois règles de décision utilisant M_i, le nombre de cycles alloués au processus entre l'instant $i-1$ et i.

1. règle *threshold* : le processus est considéré actif si M_i dépasse le seuil T.
2. règle *relative* : le processus est considéré actif si M_i dépasse le nombre moyen de cycle de l'ensemble des processus.
3. règle *sliding window* : le processus est considéré actif si la moyenne des N dernières mesures (i.e., M_{i-N-1}, $M_{i-N-2}, \ldots, M_{i-1}$) est plus bas que la dernière mesure M_i.

Nous définissons alors $T_C(x)$ comme le nombre total de fois où le processus x est actif. Un exemple de valeurs de $T_C(x)$ est donné en table 4.3. Nous définissons $A_C(x, y)$ le nombre de mesures pour lesquels x et y sont actifs en même temps. Enfin, nous calculons un facteur d'activité $A_F(x, y)$ qui va représenter la corrélation d'activité de x et y :

$$A_F(x, y) = \frac{A_C(x, y)}{T_C(x) + T_C(y) - A_C(x, y)} \times 100$$

Un exemple de résultat pour $A_F(x, y)$ est donné en table 4.4. Ainsi, $A_F(x, y)$ donne une vue condensée de l'activité "concurrente" de deux processus et sera utilisé pour juger si des applications travaillent de concert au travers d'un canal auxiliaire.



(a) Type d'Intent (b) Volume (c) Verrou sur fichier

FIGURE 4.7 – $A_F(x, y)$ pour la paire du canal auxiliaire et les 4 autres paires les plus actives avec $T = 1$

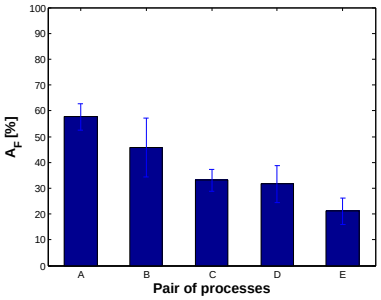


FIGURE 4.8 – $A_F(x, y)$ pour la paire du canal auxiliaire et les 4 autres paires les plus actives avec $T = 1$ pour le canal auxiliaire de type verrou sur un fichier

Scénario sans utilisateur. Pour évaluer l'efficacité du facteur d'activité, nous avons réalisé un premier scénario où l'utilisateur n'utilise pas son téléphone mobile. Dans notre esprit, cela correspond aux attaques décrites en section 2.2 où les logiciels malveillants attendent que l'utilisateur soit absent de son téléphone pour agir. Les résultats expérimentaux sont donnés en figure 4.7 pour le paramètre $T = 1$ (les résultats sont similaires pour d'autres valeurs de T). Dans chaque graphique, la première barre A représente l'activité de la paire de processus utilisant le canal auxiliaire. Les autres barres sont les paires de processus ayant le plus grand facteur d'activité. On distingue clairement la différence entre la paire impliquée dans le canal auxiliaire et les autres paires. Les autres types de règle de décision (règles "*relative*" et "*sliding window*") donnent des résultats semblables. Lorsque l'on compare l'ensemble de ces résultats, on remarque que le facteur d'activité est toujours supérieur à 60% ce qui donne une valeur expérimentale de T permettant de toujours détecter le canal auxiliaire.

Scénario avec utilisateur. Dans ce scénario de 15 minutes, nous avons réalisé les opérations manuellement. Pendant les 5 premières minutes, l'utilisateur regarde une vidéo, puis pendant 5 minutes il utilise l'application Facebook et envoie des messages, et enfin les 5 dernières minutes sont consacrées à la navigation sur le web. Compte tenu de la lourdeur de l'expérience (il faut répéter plusieurs fois l'expérience afin d'avoir une moyenne représentative), nous n'avons évalué que le canal auxiliaire utilisant un verrou sur les fichiers. La figure 4.8 montre les résultats obtenus, toujours avec la paire de processus du canal auxiliaire en colonne A. Les résultats sont clairement plus mitigés : les autres paires actives atteignent un facteur d'activité de plus de 40%. Ce résultat montre bien la limite de la méthode : lorsqu'on introduit un troisième processus actif dans le système, il est beaucoup plus difficile de repérer la corrélation entre les deux processus du canal auxiliaire.

3.4 Détection de canaux auxiliaires basée sur l'énergie consommée

Avec la méthode précédente, un inconvénient notable est qu'il faut surveiller l'ensemble des processus du système. Nous avons donc cherché à réaliser la détection sur une donnée plus globale, afin de simplifier l'algorithme de détection et l'impact sur les ressources du téléphone. En s'inspirant de la littérature, nous avons travaillé sur la détection basée sur la consommation énergétique, en sachant qu'il n'était pas certain que l'on pourrait extraire assez d'information pour détecter quelque chose.

3.4.1 Le problème de la détection d'attaques utilisant les mesures d'énergie

Dans la littérature, les méthodes de détection d'anomalie basées sur les mesures d'énergie peuvent être classées en plusieurs grandes familles : les méthodes basées sur la consommation du système, sur la consommation des applications, sur les habitudes utilisateurs et sur la consommation particulière des attaques.

Les méthodes basées sur la consommation du système mesure l'énergie globale ou de certaines parties du système. Par exemple Jacoby *et al.* [215] montrent comment détecter des attaques réseaux avec un IDS qui analyse la consommation de parties matérielles et logicielles du système. Pour des attaques qui ciblent un système disposant d'une énergie limitée, ces méthodes sont particulièrement efficaces. Nash *et al.* [230] montrent comment construire une combinaison linéaire de différentes mesures énergétiques du système (CPU, écran, réseau, etc.) afin de construire une règle de détection de ce type d'attaque. Liu *et al.* [224] utilisent une approche comparative entre la consommation normale d'un téléphone sous Symbian et la consommation lorsqu'une attaque se produit, ce qui lève une alerte. Par la suite, Hoffmann *et al.* [213] comparent des profils énergétiques et se concentrent sur les communications wifi et 3G. Les auteurs concluent qu'il serait difficile de repérer des comportements malveillants, étant donné la précision dans les mesures énergétiques, combiné à la grande variabilité des comportements des utilisateurs. Les résultats obtenus par Liu *et al.* sont même donnés comme non reproductibles. Il faut donc être raisonnable dans les capacités d'un système de détection basé sur l'énergie : toutes les malveillances ne sont sans doute pas détectables et l'utilisateur est un facteur important. Nous avons donc restreint nos propres travaux aux hypothèses suivantes : les canaux auxiliaires considérés utiliseront une bande passante conséquente et l'utilisateur sera supposé absent.

Les attaques essayant de vider la batterie du téléphone ont été particulièrement étudiées [218, 204, 205, 199, 201, 229]. Pour ce type d'attaque, on peut concevoir des contremesures qui s'appuient sur l'analyse des applications ou des habitudes de l'usager. Kim *et al.* [218] proposent de construire des signatures applicatives en utilisant l'énergie consommée par chaque application. Une fois les signatures acquises, la détection s'effectue en utilisant la distance du χ^2. L'intérêt de la méthode est de pouvoir détecter des virus non connus réalisant les mêmes types d'attaques que ceux contenus dans la base de signature, dans ce cas précis, des attaques pour vider la batterie. Évidemment, le type d'attaque considéré aide grandement la méthode proposée. Dixon *et al.* [204, 205] montrent qu'il existe une corrélation forte entre la localisation de l'usager et la consommation énergétique de son téléphone. Ainsi, il devient possible de déterminer la consommation moyenne pour une localisation particulière et d'exploiter ce fait pour détecter des anomalies. Curti *et al.* [201] se concentrent sur les attaques de déni de service, par exemple une attaque à l'aide de simples *ping* depuis un PC contre une application du téléphone. Par la suite, Merlo *et al.* [229] complètent cette étude en montrant que la détection de ce type d'attaque ne peut se faire qu'en capturant les mesures énergétiques à bas niveau, au plus proche des composants. Cette conclusion n'est valable que pour ce type d'attaque qui n'impacte que les services réseaux. Ainsi, l'ensemble de ces contributions montrent que la détection basée sur l'énergie dépend étroitement du type d'attaque considéré, son contexte de mise en œuvre, notamment la présence de l'utilisateur, et de la façon de mesurer l'énergie.

3.4.2 Détection de canaux auxiliaires basée sur l'énergie consommée

Pour réaliser nos systèmes de détection, nous avons exploré une approche similaire à Nash *et al.* [230] en modélisant le comportement standard de l'énergie du système, afin de pouvoir détecter une déviance par la suite [197]. Deux grandes familles de méthodes ont été considérées : les méthodes basées "régression" qui utilisent les valeurs acquises dans le passé afin de prédire les valeurs à venir ; les méthodes basées "classification" qui utilisent des caractéristiques du système pour classer comportement normal et anormal. Les modèles de régression permettent de prédire des valeurs réelles, et donc dans notre cas, la consommation énergétique du téléphone. Les méthodes de classification projettent les valeurs d'entrée dans des classes et seront utilisées dans notre cas pour classer parmi les classes "avec/sans" canal auxiliaire. Pour ces deux ensembles de méthodes, une phase d'apprentissage est nécessaire pour paramétrer correctement les modèles. Concrètement, nous avons utilisé deux méthodes classiques de l'Intelligence Artificielle en mode régression et classification, ce qui fait quatre méthodes au total :

- réseaux de neurones (régression et classification) avec un apprentissage utilisant l'algorithme Levenberg-Marquardt [228] ;
- arbres de décisions binaires (classification et classification) où l'apprentissage cherche à minimiser la prédiction d'erreur avec les données d'entrainement [200].

Pour entrainer ces modèles, nous avons réalisé 5 000 séances de mesures d'énergie, avec ou sans canal auxiliaire actif. Certains paramètres sont choisis manuellement : ils permettent d'optimiser les résultats de détection afin d'obtenir les meilleurs résultats pour chaque méthode testée. Les résultats obtenus sont présentés dans la

Table 4.5 – Average percentages of correct detection for the different detection methods and covert channels.

	Régression		Classification	
Canal Auxiliaire	Réseau de neurones	Arbre de décision	Réseau de neurones	Arbre de décision
Type d'Intent	74.3	73.1	90.8	86.7
Paramètre de volume	87.1	86.0	97.6	93.5
Verrou sur fichier	65.9	86.2	96.5	96.2
Charge du système	90.4	94.2	93.3	91.1
Découverte de socket	75.4	74.2	94.2	91.4
Taille des fichiers	65.3	68.6	88.9	80.5
Charge de la mémoire	71.5	72.4	84.6	79.0

table 4.5. Les canaux auxiliaires basés sur la charge du système et le paramètre de volume sont les plus faciles à détecter, ce qui sous entendrait que ce sont les méthodes qui consomment le plus d'énergie. A l'inverse, les canaux auxiliaires utilisant la taille d'un fichier ou la charge mémoire sont les plus difficiles à détecter. Nous pensons que cela est lié à la façon dont l'énergie est mesurée : en la mesurant à haut niveau (au niveau de l'ensemble des logiciels de l'espace utilisateur), nous ne mesurons pas l'énergie que le noyau consomme pour réaliser les opérations d'allocation mémoire. Au global, les méthodes basées sur la régression ont en moyenne un taux de détection de 65% (35% de faux positifs) et les méthodes basées sur la classification de 85% (15% de faux positifs).

Les résultats obtenus sont particulièrement encourageants. Ils montrent que l'on peut détecter un canal auxiliaire en mesurant uniquement l'énergie globale du système. Les méthodes proposées pour traiter cette mesure sont raisonnables en temps de traitement et il est complètement envisageable de les intégrer au système d'exploitation. L'approche suppose cependant que l'utilisateur n'intervient pas pendant que le canal auxiliaire se déclenche. L'extension naturelle de ce travail consiste à intégrer à l'expérience un utilisateur et à mesurer l'impact sur la méthode.

Dans la suite de chapitre, nous nous focalisons sur les attaques qui peuvent provenir du fournisseur de service. Nous présentons un cas d'usage précis, le transport, et nous montrons comment un protocole peut être conçu pour respecter la vie privée des utilisateurs et rendre caduque toute attaque du fournisseur de service.

4 Conception d'un service de transport anonyme et intraçable

Dans cette partie, nous proposons de concevoir un système d'authentification pour l'accès à un service de transport anonyme et intraçable. Dans ce scénario, les utilisateurs accèdent au service de transport en utilisant un téléphone mobile NFC et s'authentifient donc à un portique d'accès comme dans le métro. La phase d'authentification est une phase de validation du titre de transport, plutôt qu'une phase cherchant à vérifier l'identité de l'utilisateur. Cependant, dans de nombreux systèmes, la validation du titre revient à vérifier son identité et l'association entre l'identité et le titre de transport associé. De plus, le titre de transport considéré dans cette partie sera un ticket digital, plutôt qu'un *pass* de transport donnant droit à un nombre illimité de voyages. Nous avons aussi contribué sur le cas du *pass* de transport anonyme et intraçable [186], mais le cas du ticket digital anonyme et intraçable [189] est plus intéressant à présenter, car plus complexe à réaliser.

Pour ce cas d'étude, l'hypothèse d'attaque est que le fournisseur de service est honnête mais curieux et qu'il cherche donc à tracer les utilisateurs du service de transport afin de connaitre leurs habitudes d'accès au service. Dans la suite, avant de proposer un protocole permettant de protéger la vie privée de l'utilisateur, nous donnons les contributions de la littérature s'étant intéressés à cette problématique dans le cas du transport.

4.1 Vie privée et tickets de transport digitaux

Les premiers à s'intéresser aux questions d'anonymat dans les transport publics sont Heydt-Benjamin *et al.* [212]. Ils définissent les enjeux et donnent les principales propriétés qu'un système de transport auquel on accède avec un ticket digital doit comporter. Les hypothèses données sur l'infrastructure sont toujours d'actualité. Les portiques pour l'accès au transport sont supposés reliés au réseau avec un débit suffisant. A l'inverse, le moyen utilisé par l'utilisateur pour transporter son ticket digital est un système informatique à ressources limitées : la puissance de calcul est limité, et les moyens de communication le sont aussi. Cette hypothèse est toujours vraie avec un téléphone mobile qui peut ne pas disposer d'une connexion réseau, notamment dans le métro. Heydt-Benjamin *et al.* discutent des technologies de communication entre le système transporté par l'utilisateur et la borne. Que la technologie soit du RFID ou maintenant du NFC, la problématique globale ne

Table 4.6 – Comparaison des solutions de ticketing

Auteurs	Anonymat	Non traçabilité	Dépendance matériel	Validation $< 300\,ms$
Heydt-Benjamin *et al.* [212]	✓	✓	Mobile	?
Ekberg et Tamrakar [206, 241]	T	T	TEE	✓
Sadeghi *et al.* [237]	T	T	TEE	✓
Isern-Deyà *et al.* [214]	✓	✓	Mobile	†
Rupp *et al.* [236]	✓	†	tag RFID	†
Blass *et al.* [194]	✓	✓	tag RFID	?

✓ : présent – T : garanti vis-à-vis d'un tiers – † : absent

change pas : les communications se font à distances courtes et le débit de communication est limité. A l'époque, il devenait naturel d'envisager le téléphone mobile comme moyen de transporter des titres de transport digitaux et de passer le portique en présentant son titre au portique. De tels services ont été déployés en Asie, en utilisant des cartes sans contacts : dans les transport de Séoul en 1995 et à Honk Kong avec la carte Octopus [240] en 1997. La principale difficulté technique pour de tels services embarqués sur des cartes ou des mobiles à faibles capacités de calcul, est de concevoir un protocole qui respecte une contrainte temporelle forte : la validation doit être effectuée en moins de $300\,ms$.

D'un point de vue de la sécurité et de la vie privée, un tel système de transport digital doit garantir un certain nombre de propriétés. Il doit rendre les tickets digitaux impossibles à forger, non duplicables, mais éventuellement fournir une solution de sauvegarde de ceux-ci [212] et permettre de transférer un ticket à un tiers. Pour la protection de la vie privée, il faut ajouter en plus l'anonymat et la non traçabilité du ticket, notamment vis-à-vis d'un tiers ou du transporteur. En effet, personne ne doit être en mesure de pouvoir faire le lien entre un ticket digital utilisé à un portique et l'identité de l'utilisateur. Si cette propriété est naturelle quand on pense à un tiers, elle ne l'est pas forcément lorsque l'on évoque le fournisseur de service.

En effet, les premières contributions traitant de la préservation de la vie privée s'intéressent à un modèle d'attaquant tiers [206, 241, 237]. Ekberg et Tamrakar [206, 241] proposent une solution permettant de recharger le service mobile avec des tickets générés de manière coopérative par le transporteur et un programme du téléphone protégé par un TEE. A la validation, le portique vérifie que le ticket est valide, c'est-à-dire qu'il a été généré correctement vis-à-vis de la clef privée du transporteur, et il vérifie que le téléphone est capable de signer correctement un challenge envoyé par le portique avec une clef ayant participé à la génération des tickets et que le transporteur peut tester avec la partie publique. En combinant ces deux vérifications, on s'assure que l'utilisateur possède un ticket qu'il a acheté précédemment, et qu'il s'agit bien de l'utilisateur original (pas d'un attaquant rejouant un ticket par exemple). Dans ce protocole, les auteurs vérifient que l'anonymat de l'utilisateur est préservé vis-à-vis d'un attaquant extérieur. De plus, on ne peut tracer un téléphone mobile en récupérant plusieurs validations de tickets différents. On a donc des garanties fortes, anonymat et non traçabilité, mais la solution reste transparente vis-à-vis du transporteur. D'autres solutions ayant les mêmes propriétés ont été proposées [237].

A partir de 2012, les modèles d'attaquant se sont alors renforcés et ont inclu le transporteur de service comme possible espion dans le système [214, 236]. Avec une telle hypothèse, concevoir un système de transport anonyme et intraçable pour le fournisseur du service est un vrai défi. De plus, en cas de problème grave où la police ou la justice souhaiterait connaitre l'identité des utilisateurs, de telles propriétés empêcheraient le transporteur d'accéder à ces demandes. Ainsi, dans les solutions de la littérature, il faut introduire une ou plusieurs autorités qui ont le pouvoir de désanonymiser un ticket. On appelle ce type d'anonymat, l'anonymat révocable. Pour réaliser cela, on utilise une exponentation modulaire de l'identifiant de l'utilisateur afin d'obtenir un pseudonyme de cet identifiant [214]. Le transporteur peut alors vérifier une preuve à divulgation de connaissance nulle de l'identifiant de l'utilisateur à partir de ce pseudonyme [239]. Par ailleurs, le second principe de ces protocoles, consiste à réaliser des signatures de groupes [214] pour vérifier la validité du titre de transport pour obtenir des garanties d'anonymat et de non traçabilité. Ces solutions sont proches des systèmes de paiements anonymes et intraçables. Certaines contributions s'appuient sur la manipulation de pièces pour donner de la flexibilité au système en ne payant de manière anonyme que ce qui est nécessaire en fonction de la distance parcourue [236, 194].

Nous donnons une synthèse des propriétés d'anonymat et de non traçabilité dans la table 4.6. Il faut noter l'absence de non traçabilité de la solution de Rupp *et al.* [236] pour laquelle une attaque a été identifiée lors du travail de thèse de Ghada Arfaoui [1]. La difficulté est de réussir à obtenir l'anonymat et la non traçabilité avec un délai de calcul raisonnable sur des environnements aux ressources contraintes.

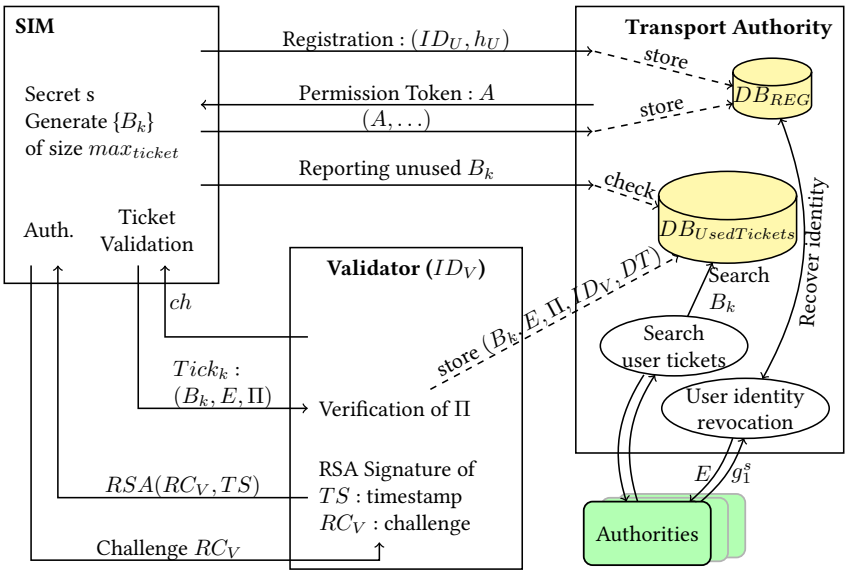


FIGURE 4.9 – Le protocole de ticketing

4.2 Protocole de ticketing anonyme et intraçable

Nous proposons de donner une vue globale du protocole de *ticketing* anonyme et intraçable que nous avons réalisé et implémenté sur une carte à puce [189]. Notre objectif est de décrire les grandes étapes du protocole et de donner l'intuition des mécanismes qui garantissent les propriétés de sécurité et d'anonymat. Les propriétés visées sont présentées informellement ci-dessous :

- efficacité : la contrainte d'une validation en moins de $300\,ms$ doit être respectée ;
- versatilité : le téléphone et le portique de validation ne sont pas connectés à un serveur en *back-end* au moment de la validation du ticket ;
- les tickets doivent être non forgeables : un utilisateur ne peut pas forger plus de nouveaux tickets que prévus dans le protocole et un attaquant extérieur ne peut en générer non plus ;
- les tickets doivent être non traçables : il doit être impossible de lier des tickets utilisés entre eux, ou de lier des tickets utilisés avec l'identité de l'utilisateur ou avec les tickets non utilisés ;
- le système doit garantir la non répudiation de l'utilisateur : il doit être impossible d'accuser un utilisateur honnête de fraude ;
- l'anonymat d'un ticket doit pouvoir être révoqué, en s'appuyant sur des autorités indépendantes ;
- la duplication malveillante de tickets doit pouvoir être détectée.

Les grandes étapes du protocole sont données ci-dessous et représentées en figure 4.9.

Enregistrement de l'utilisateur L'identité de l'utilisateur ID_U est enregistrée dans la base du transporteur, ainsi que sa clef publique h_U lui permettant de réaliser des signatures de type Schnorr [239]. Il prouve au passage qu'il possède bien des clefs valides en signant un challenge.

Négociation du jeton de permission L'utilisateur et le transporteur construisent un jeton qui repose sur une signature de groupe de type Boneh-Boyen [195, 196]. Cette signature de groupe signe le secret de l'utilisateur s (une clef de groupe). A l'issue du protocole, des valeurs sont renvoyées au transporteur et qui sont utilisées par les autorités en cas de fraude ou de levée de l'anonymat.

Authentification du portique L'utilisateur authentifie le portique en proposant un challenge au portique qui répond avec une signature RSA.

Validation du ticket Le ticket est représenté par un numéro unique B_k qui est une signature de groupe à partir du secret de l'utilisateur s, de l'indice du ticket dans un intervalle $[1, max]$, une preuve de connaissance Π à divulgation nulle que :

TABLE 4.7 – Temps de validation d'un ticket incluant l'authentification du portique, en ms (écart type)

	Validator authentication	Card Signature + NFC connection	Verification by PC (1) without pairing	Verification by PC (2) with pairing	Total (1)	Total (2)
Battery-On	56, 98 (0.70)	123.01 (3.24)	4.43 (1.32)	12.19 (3.20)	184.25 (3.43)	191.80 (4.73)
Battery-Off	76.55 (7.46)	185.28 (18.68)			266.52 (17.91)	272.55 (25.73)

— A est correctement calculé vis-à-vis de s ;
— B_k est correctement calculé avec s ;
— k est dans l'intervalle $[1, max]$ afin de limiter le nombre de tickets générés.

Ainsi, sans connaitre s, l'autorité de transport peut recevoir ce numéro de ticket B_k sans savoir quel utilisateur l'a généré mais en sachant que l'utilisateur possède une signature qu'il a approuvé sur A et qu'il a calculé ce numéro en utilisant s et A. Ce numéro de ticket, ainsi que les preuves correspondantes sont enregistrées dans la base de données du transporteur.

D'un point de vue des propriétés de sécurité énoncées précédemment, nous donnons quelques éléments donnant l'intuition des preuves sous-jacentes. Les tickets sont non forgeables car générés à partir du secret s que le transport a signé et pour lequel l'utilisateur prouve posséder une signature valide. De plus, si un utilisateur légitime génère trop de tickets, il aura un indice k plus grand que max ce qui sera détecté dans la vérification de sa preuve à divulgation de connaissance nulle. On ne peut lier deux tickets grâces aux propriétés des signatures de groupe : deux signatures de groupe ne peuvent être liées à la clef d'un utilisateur membre du groupe.

Enfin, le protocole prévoit la levée de l'anonymat des utilisateurs grâce aux informations stockées dans les différentes bases et à un protocole dédié à l'enregistrement de l'utilisateur avec les autorités. Il est pourtant peu probable d'avoir à utiliser cette fonctionnalité, par exemple en cas de ticket clonés. En effet, il serait très difficile d'arriver à cloner des tickets car cela supposerait d'arriver à extraire d'une carte SIM le secret s et le jeton A. Une hypothèse plus réaliste pourrait être que l'utilisateur force sa carte SIM à utiliser plusieurs fois le même ticket k, par exemple avec une attaque physique. Malgré ces hypothèses d'attaques très difficiles à mettre en œuvre, il est possible de détecter des tickets clonés ou utilisés plusieurs fois puisque l'on doit divulguer le numéro du ticket. On peut alors lever l'anonymat du ou des utilisateurs concernés.

4.3 Fonctionnalités supplémentaires et efficacité du protocole

L'originalité du protocole proposé réside aussi dans sa capacité à implémenter un paiement postérieur à l'utilisation des tickets. Le jeton d'autorisation A sert à donner un droit à l'utilisateur tout en le limitant ce droit à la génération de max tickets. Une fois ce nombre de tickets dépensés, l'utilisateur doit renouveler le jeton A et donc payer les tickets utilisés. Pour rester anonyme, l'utilisateur divulgue au transporteur les tickets non-utilisés et le transporteur en déduit le nombre de tickets qu'il a utilisés, sans pour autant connaitre les numéros en question. Si l'utilisateur ne réalise jamais cette phase, c'est-à-dire la remontée des tickets non utilisés et le renouvellement de A, le transporteur peut charger financièrement et par défaut max tickets à l'utilisateur.

La seconde originalité de la solution est qu'il est possible de valider un de ses tickets sans que le téléphone soit allumé. Grâce à l'alimentation du circuit NFC du téléphone grâce au champ magnétique du portique, la carte SIM est alimentée en mode dégradé et peut répondre au portique.

Ainsi, nous donnons en table 4.7 les temps pour la validation d'un ticket ainsi que l'écart type entre parenthèses. Si le portique (représenté par un ordinateur dans l'expérience) est capable de réaliser des calculs de *pairing* pour la vérification des signatures de groupe, la validation s'effectue en $184.25\,ms$ en moyenne. Dans le cas contraire, seulement $7\,ms$ supplémentaires sont nécessaires au portique pour exécuter la partie de notre protocole qui prend ce cas de figure en compte. Quand le téléphone est éteint, la validation est plus lente mais s'effectue tout de même en $272.55\,ms$ dans le pire des cas, ce qui respecte la contrainte des $300\,ms$.

5 Logiciels

Lors des travaux présentés dans ce chapitre, nous avons développé une plate-forme d'expérimentation pour les canaux auxiliaires appelée SteganoCC. Cette plate-forme permet de réutiliser les implémentations de tous les canaux auxiliaires testés et d'expérimenter la collecte des données énergétiques à partir d'une version modifiée de PowerTutor. L'objectif est de garantir la reproductibilité des expériences menées dans nos travaux d'élaboration de canaux auxiliaires [221] et de détection [197].

Nos travaux sur les protocoles pour les systèmes de transport utilisant un téléphone mobile NFC ont conduit à plusieurs développements de démonstrateurs, dans le cadre de l'ANR Lyrics. Même si l'ensemble de ces démonstrateurs appartient au consortium et n'est pas distribué sous licence libre, nous avons réalisé de nombreuses démonstrations à des manifestations scientifiques [209, 220, 187] et salons industriels.

6 Bilan et perspectives de recherche

Les travaux présentés dans ce chapitre se sont focalisés sur la protection de la vie privée de l'utilisateur. Notre approche duale s'est focalisée d'une part sur les attaques par canaux auxiliaires et leur détection, puis sur les attaques d'un fournisseur de service honnête mais curieux et les protocoles répondant à ce problème.

Nos contributions sur les canaux auxiliaires montrent que de nombreuses façons permettent d'extraire des données en contournant les mécanismes de sécurité d'Android. La bande passante de ces canaux auxiliaires ne nous parait pas un critère déterminant. La discrétion de ces canaux, notamment les permissions qu'ils requièrent et l'énergie qu'ils consomment, nous parait un nouveau critère à prendre en compte pour caractériser ces malveillances. Cet aspect des choses rejoint les travaux menés dans le chapitre 3 où l'on considère des *malware* qui cherchent à échapper à l'analyse dynamique. Les malveillances cherchent désormais à être discrètes, plutôt qu'à maximiser leur efficacité. Dans le cas de l'extraction de données volées, cela signifie qu'il faut concentrer l'effort de recherche sur les aspects de masquage des attaques plutôt que sur la bande passante lors de l'extraction.

Sur les aspects de détection de canaux auxiliaires, nous avons montré qu'avec l'hypothèse d'un utilisateur absent, ces communications malveillantes pouvaient être facilement détectées. Il faut donc se concentrer sur la recherche de nouvelles méthodes pour masquer les canaux auxiliaires avec l'activité de l'utilisateur. Ce type de recherche est particulièrement difficile puisqu'il faut à la fois concevoir la méthode, puis démontrer au moins expérimentalement qu'un canal auxiliaire qui chercherait à opérer de manière corrélée à l'utilisateur est plus difficile à détecter. Même si intuitivement, cette thèse est raisonnable, la méthode de quantification est à élaborer spécifiquement. D'autre part, il faut noter que nous avons alimenté le sujet controversé de la détection de *malware* en utilisant des mesures énergétiques. Nous montrons que cela est possible, même à haut niveau, mais avec des hypothèses bien identifiées et pour un périmètre d'attaque borné. Ce sujet est donc encore ouvert pour de nombreux scénarios d'attaques encore non étudiés.

Le cas du canal auxiliaire entre deux processus contrôlés par l'attaque est une architecture d'attaque qu'il faut aussi remettre en cause. En effet, il n'est pas forcément aisé pour l'attaquant d'avoir deux applications sous son contrôle. Nous pensons donc qu'il faudrait étudier l'hypothèse où l'attaquant ne contrôle qu'une application et cherche à abuser d'une autre application du téléphone, afin de faire transiter des informations entre ces deux applications. Si cette seconde application communique ensuite les données sur un serveur distant, alors il serait possible d'extraire des données avec une seule application malveillante. Si cela était possible, alors on obtiendrait quelque chose de similaire aux fuites d'informations qui peuvent survenir entre des composants logiciels [144], mais qui cette fois serait intentionnelles et contrôlées par un attaquant.

Enfin, nos travaux sur les services de transport ont contribué à développer des protocoles pour l'accès à des services de transport à l'aide de tickets digitaux. Ces protocoles garantissent un anonymat fort et pourtant révocable si des circonstances exceptionnelles l'exigent. Pour ce type de contributions, la complexité des calculs cryptographiques rend l'élaboration des preuves de sécurité longues et fastidieuses. Cette complexité s'explique par la puissance limitée de la carte SIM qui embarque une partie du code, l'autre étant gérée par le téléphone. Avec de nouvelles architectures matérielles comme les TEE, nous pourrions simplifier les protocoles proposés et gagner en lisibilité.

7 Références

[1] G. Arfaoui. **Conception de protocoles cryptographiques préservant la vie privée pour les services mobiles sans contact**. Thèse de doctorat. Université d'Orléans, novembre 2012 (cf. p. 5, 80).

[185] R. Accorsi et C. Wonnemann. **InDico : Information Flow Analysis of Business Processes for Confidentiality Requirements**. Dans : *6th International Workshop on Security and Trust Management*. T. 6710. LNCS. Athens, Greece : Springer Berlin / Heidelberg, septembre 2010, p. 194–209. doi : 10.1007/978-3-642-22444-7_13 (cf. p. 70).

[186] G. Arfaoui, G. Dabosville, S. Gambs, P. Lacharme et J.-F. Lalande. **A Privacy-Preserving NFC Mobile Pass for Transport Systems**. Dans : *EAI Endorsed Transactions on Mobile Communications and Applications* 14.5 (décembre 2014), e4. doi : 10.4108/mca.2.5.e4 (cf. p. 6, 79).

[187] G. Arfaoui, G. Dabosville, S. Gambs, P. Lacharme et J.-F. Lalande. **Un pass de transport anonyme et intraçable pour mobile NFC**. Dans : *Atelier sur la Protection de la Vie Privée 2014*. Cabourg, France, juin 2014 (cf. p. 83).

[188] G. Arfaoui, S. Gharout et J. Traoré. **Trusted Execution Environments : A look under the hood**. Dans : *The International Workshop on Trusted Platforms for Mobile and Cloud Computing*. Oxford, UK : IEEE Computer Society, avril 2014, p. 259–266. doi : 10.1109/MobileCloud.2014.47 (cf. p. 67, 69).

[189] G. Arfaoui, J.-F. Lalande, J. Traoré, N. Desmoulins, P. Berthomé et S. Gharout. **A Practical Set-Membership Proof for Privacy-Preserving NFC Mobile Ticketing**. Dans : *Proceedings on Privacy Enhancing Technologies* 2015.2 (janvier 2015), p. 25–45. doi : 10.1515/popets-2015-0019 (cf. p. 6, 79, 81).

[190] A. Bartel, J. Klein, M. Monperrus, K. Allix et Y. L. Traon. *In-Vivo Bytecode Instrumentation for Improving Privacy on Android Smartphones in Uncertain Environments*. 2012. arXiv : arXiv:1208.4536v1 (cf. p. 69).

[191] C. Bauer, J. Korunovska et S. Spiekermann. **On the Value of Information - What Facebook Users are Willing to Pay**. Dans : *20th European Conference on Information Systems*. Barcelona, Spain, juin 2012 (cf. p. 65).

[192] V. H. Berk, A. Giani et G. V. Cybenko. **Detection of Covert Channel Encoding in Network Packet Delays**. Rapport technique. Department of Computer Science - Dartmouth College, 2005 (cf. p. 74).

[193] P. Berthomé, T. Fécherolle, N. Guilloteau et J.-F. Lalande. **Repackaging Android applications for auditing access to private data**. Dans : *First International Workshop on Security of Mobile Applications*. Prague, Czech Republic : IEEE Computer Society, août 2012, p. 388–396. doi : 10.1109/ARES.2012.74 (cf. p. 6, 69).

[194] E.-O. Blass, A. Kurmus, R. Molva et T. Strufe. **PSP : Private and secure payment with RFID**. Dans : *Computer Communications* 36.4 (2013), p. 468–480. doi : 10.1016/j.comcom.2012.10.012 (cf. p. 80).

[195] D. Boneh et X. Boyen. **Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups**. English. Dans : *Journal of Cryptology* 21.2 (2008), p. 149–177. doi : 10.1007/s00145-007-9005-7 (cf. p. 81).

[196] J. Camenisch et A. Lysyanskaya. **Signature Schemes and Anonymous Credentials from Bilinear Maps**. Dans : *Advances in Cryptology*. Sous la dir. de M. Franklin. T. 3152. LNCS. Santa Barbara, USA : Springer Berlin Heidelberg, août 2004, p. 56–72. doi : 10.1007/978-3-540-28628-8_4 (cf. p. 81).

[197] L. Caviglione, M. Gaggero, J.-F. Lalande, W. Mazurczyk et M. Urbanski. **Seeing the Unseen : Revealing Mobile Malware Hidden Communications via Energy Consumption and Artificial Intelligence**. Dans : *IEEE Transactions on Information Forensics and Security* 11.4 (2016), p. 799–810. doi : 10.1109/TIFS.2015.2510825 (cf. p. 6, 74, 78, 82).

[198] L. Caviglione, J.-F. Lalande, W. Mazurczyk et S. Wendzel. **Analysis of Human Awareness of Security and Privacy Threats in Smart Environments**. Dans : *3rd International Conference on Human Aspects of Information Security, Privacy and Trust*. Sous la dir. de T. Tryfonas et I. G. Askoxylakis. T. 9190. LNCS. Los Angeles, USA : Springer Berlin / Heidelberg, août 2015, p. 165–177. doi : 10.1007/978-3-319-20376-8_15. arXiv : 1502.00868 (cf. p. 6, 74).

[199] L. Caviglione et A. Merlo. **The energy impact of security mechanisms in modern mobile devices**. Dans : *Network Security* 2012.2 (2012), p. 11–14. doi : 10.1016/S1353-4858(12)70015-6 (cf. p. 78).

[200] T. Cover et T. Joy. *Elements of Information Theory*. Wiley, 1991 (cf. p. 78).

[201] M. Curti, A. Merlo, M. Migliardi et S. Schiappacasse. **Towards energy-aware intrusion detection systems on mobile devices**. Dans : *International Conference on High Performance Computing and Simulation*. Helsinki, Finland : IEEE Computer Society, juillet 2013, p. 289–296. doi : 10.1109/HPCSim.2013.6641428 (cf. p. 78).

[202] L. Davi, A. Dmitrienko, A. Sadeghi et M. Winandy. **Privilege escalation attacks on android**. Dans : *13th Information Security Conference*. T. 6531. LNCS. Boca Raton, USA : Springer Berlin / Heidelberg, octobre 2011. doi : 10.1007/978-3-642-18178-8_30 (cf. p. 68).

[203] M. De Saint Léger, S. Gambs, B. Juanals, J.-F. Lalande et J.-L. Minel. **Privacy and Mobile Technologies : the Need to Build a Digital Culture**. Dans : *Digital Intelligence*. Nantes, France : Université de Nantes, septembre 2014, p. 100–105 (cf. p. 6, 68).

[204] B. Dixon, Y. Jiang, A. Jaiantilal et S. Mishra. **Location based power analysis to detect malicious code in smartphones**. Dans : *1st Workshop on Security and Privacy in Smartphones and Mobile Devices*. Chicago, Illinois, USA : ACM Press, 2011, p. 27–32. doi : 10.1145/2046614.2046620 (cf. p. 78).

[205] B. Dixon et S. Mishra. **Power based malicious code detection techniques for smartphones**. Dans : *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. Melbourne, Australia : IEEE Computer Society, 2013, p. 142–149. doi : 10.1109/TrustCom.2013.22 (cf. p. 78).

[206] J. Ekberg et S. Tamrakar. **Mass transit ticketing with NFC mobile phones**. Dans : *The Third International Conference on Trusted Systems*. Sous la dir. de L. Chen, M. Yung et L. Zhu. T. 7222. Beijing, China : Springer Berlin / Heidelberg, 2012, p. 48–65. doi : 10.1007/978-3-642-32298-3_4 (cf. p. 80).

[207] N. Elenkov. *Android Security Internals*. No Starch Press, 2014 (cf. p. 68).

[208] J. Fan, X. Guo, E. D. Mulder, P. Schaumont, B. Preneel et I. Verbauwhede. **State-of-the-art of Secure ECC Implementations : A Survey on Known Side-channel Attacks and Countermeasures**. Dans : *IEEE International Symposium on Hardware-Oriented Security and Trust*. Anaheim Convention Center, California, USA : IEEE Computer Society, juin 2010, p. 76–87. doi : 10.1109/HST.2010.5513110 (cf. p. 70).

[209] S. Gambs, J.-F. Lalande et J. Traoré. **ANR LYRICS : Cryptographie pour la protection de la vie privée, optimisée pour les services mobiles sans contact**. Dans : *Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. Troyes, France, mai 2015 (cf. p. 83).

[210] GlobalPlatform. **The Trusted Execution Environment : Delivering Enhanced Security at a Lower Cost to the Mobile Market**. Rapport technique February. Global Platform, 2011, p. 1–26 (cf. p. 67).

[211] M. Hansen, R. Hill et S. Wimberly. **Detecting covert communication on Android**. Dans : *37th Annual IEEE Conference on Local Computer Networks*. Clearwater, Florida, USA : IEEE Computer Society, octobre 2012, p. 300–303. doi : 10.1109/LCN.2012.6423634 (cf. p. 74).

[212] T. S. Heydt-Benjamin, H.-J. Chae, B. Defend et K. Fu. **Privacy for Public Transportation**. Dans : *6th International Conference on Privacy Enhancing Technologies*. Sous la dir. de G. Danezis et P. Golle. T. 4258. LNCS. Cambridge, UK : Springer Berlin Heidelberg, 2006, p. 1–19. doi : 10.1007/11957454_1 (cf. p. 79, 80).

[213] J. Hoffmann, S. Neumann et T. Holz. **Mobile Malware Detection Based on Energy Fingerprints—A Dead End ?** Dans : *16th International Symposium on Research in Attacks, Intrusions and Defenses*. T. 8145. LNCS. Rodney Bay, St. Lucia : Springer Berlin Heidelberg, octobre 2013, p. 348–368. doi : 10.1007/978-3-642-41284-4_18 (cf. p. 75, 78).

[214] A. P. Isern-Deya, A. Vives-Guasch, M. Mut-Puigserver, M. Payeras-Capella et J. Castella-Roca. **A Secure Automatic Fare Collection System for Time-Based or Distance-Based Services with Revocable Anonymity for Users**. Dans : *The Computer Journal* 56.10 (avril 2012), p. 1198–1215. doi : 10.1093/comjnl/bxs033 (cf. p. 80).

[215] G. Jacoby, N. J. Davis IV et al. **Battery-based intrusion detection**. Dans : *IEEE Global Telecommunications Conference*. Dallas, USA : IEEE Computer Society, novembre 2004, p. 2250–2255. doi : 10.1109/GLOCOM.2004.1378409 (cf. p. 78).

[216] R. Johnson, M. Elsabagh, A. Stavrou et V. Sritapan. **Targeted DoS on Android : How to Disable Android in 10 Seconds or Less**. Dans : *10th International Conference on Malicious and Unwanted Software*. Fajardo, Puerto Rico : IEEE Computer Society, octobre 2015, p. 136–143. doi : 10.1109/MALWARE.2015.7413694 (cf. p. 69, 71).

[217] R. A. Kemmerer. **Shared resource matrix methodology : an approach to identifying storage and timing channels**. Dans : *ACM Transactions on Computer Systems* 1.3 (1983), p. 256–277. doi : 10.1145/357369.357374 (cf. p. 74).

[218] H. Kim, J. Smith et K. G. Shin. **Detecting energy-greedy anomalies and mobile malware variants**. Dans : *6th international conference on Mobile systems, applications, and services*. Breckenridge, USA : ACM Press, juin 2008, p. 239–252. doi : 10.1145/1378600.1378627 (cf. p. 78).

[140] N. Kiss, J.-F. Lalande, M. Leslous et V. Viet Triem Tong. **Kharon dataset : Android malware under a microscope**. Dans : *The LASER Workshop : Learning from Authoritative Security Experiment Results*. San Jose, United States : USENIX Association, mai 2016, p. 1–12 (cf. p. 6, 40–42, 69).

[219] P. Kulkarni, D. R. Kalbande, S. Warrier et N. Gulrajani. **Smartcard based Android Application for Public Transport Ticketing System**. Dans : *International Journal of Computer Applications* 60.11 (2012), p. 29–32 (cf. p. 67).

[220] J.-F. Lalande. **Un titre de transport sur mobile NFC respectueux de la vie privée**. Dans : *Colloque International sur la Sécurité des Systèmes d'Information*. Kénitra, Morocco, mars 2014 (cf. p. 83).

[221] J.-F. Lalande et S. Wendzel. **Hiding privacy leaks in Android applications using low-attention raising covert channels**. Dans : *First International Workshop on Emerging Cyberthreats and Countermeasures*. Regensburg, Germany : IEEE Computer Society, septembre 2013, p. 701–710. doi : 10.1109/ARES.2013.92 (cf. p. 6, 71, 74, 82).

[222] B. W. Lampson. **A Note on the Confinement Problem**. Dans : *Communications of the ACM* 16.10 (1973), p. 613–615. doi : 10.1145/362375.362389 (cf. p. 70).

[144] L. Li, A. Bartel, T. F. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau et P. McDaniel. **IccTA : Detecting Inter-Component Privacy Leaks in Android Apps**. English. Dans : *IEEE/ACM 37th IEEE International Conference on Software Engineering*. Firenze, Italy : IEEE, mai 2015, p. 280–291. doi : 10.1109/ICSE.2015.48 (cf. p. 40, 83).

[223] Z. Li, A. Goyal et Y. Chen. **Honeynet-based Botnet Scan Traffic Analysis**. Dans : *Botnet Detection*. Sous la dir. de W. Lee, C. Wang et D. Dagon. Springer Berlin Heidelberg, 2008, p. 25–44. doi : 10.1007/978-0-387-68768-1_2 (cf. p. 70).

[224] L. Liu, G. Yan, X. Zhang et S. Chen. **Virusmeter : Preventing your cellphone from spies**. Dans : *Recent Advances in Intrusion Detection*. T. 5758. LNCS. Saint-Malo, France : Springer Berlin Heidelberg, 2009, p. 244–264. doi : 10.1007/978-3-642-04342-0_13 (cf. p. 78).

[225] N. B. Lucena, G. Lewandowski et S. J. Chapin. **Covert Channels in IPv6**. Dans : *5th International Workshop on Privacy Enhancing Technologies*. T. 3856. LNCS. Cavtat, Croatia : Springer Berlin Heidelberg, juin 2005, p. 147–166. doi : 10.1007/11767831_10 (cf. p. 70).

[226] G. Madlmayr, J. Langer, C. Kantner et J. Scharinger. **NFC Devices : Security and Privacy**. Dans : *Third International Conference on Availability, Reliability and Security*. Barcelona, Spain : IEEE Computer Society, mars 2008, p. 642–647. doi : 10.1109/ARES.2008.105 (cf. p. 68).

[227] C. Marforio, H. Ritzdorf, A. Francillon et S. Capkun. **Analysis of the communication between colluding applications on modern smartphones**. Dans : *28th Annual Computer Security Applications Conference*. Orlando, Florida, USA : ACM Press, décembre 2012, p. 51–60. doi : 10.1145/2420950.2420958 (cf. p. 71, 74, 76).

[228] D. W. Marquardt. **An algorithm for least-squares estimation of nonlinear parameters**. Dans : *Journal of the Society for Industrial & Applied Mathematics* 11.2 (1963), p. 431–441. doi : 10.1137/0111030 (cf. p. 78).

[229] A. Merlo, M. Migliardi et P. Fontanelli. **On Energy-Based Profiling of Malware in Android**. Dans : *9th International Workshop on Security and High Performance Computing Systems*. Bologna, Italy : IEEE Computer Society, juillet 2014, p. 535–542. doi : 10.1109/HPCSim.2014.6903732 (cf. p. 78).

[230] D. C. Nash, T. L. Martin, D. S. Ha et M. S. Hsiao. **Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices**. Dans : *International Workshop on Pervasive Computing and Communication Security*. Kauai Island, USA : ACM Press, mars 2005, p. 141–145. doi : 10.1109/PERCOMW.2005.86 (cf. p. 78).

[231] S. M. Nasution, E. M. Husni et A. I. Wuryandari. **Prototype of train ticketing application using Near Field Communication (NFC) technology on Android device**. Dans : *International Conference on System Engineering and Technology*. Bandung, West Java, Indonesia : IEEE Computer Society, septembre 2012, p. 1–6. doi : 10.1109/ICSEngT.2012.6339362 (cf. p. 67).

[232] C. Orthacker, P. Teufl, S. Kraxberger, A. Marsalek, J. Leibetseder et O. Prevenhueber. **Android Security Permissions – Can we trust them ?** Dans : *3rd International ICST Conference on Security and Privacy in Mobile Information and Communication Systems*. T. 94. LNCS. Aalborg, Denmark : Springer Berlin Heidelberg, mai 2011, p. 40–51. doi : 10.1007/978-3-642-30244-2_4 (cf. p. 68).

[233] S. B. Patela, N. K. Jainb et V. V. Nagar. **Near Field Communication (NFC) based Mobile Phone Attendance System for Employees**. Dans : *International Journal of Engineering Research & Technology* 2.3 (2013), p. 1–8 (cf. p. 67).

[234] P. A. Porras et R. A. Kemmerer. **Covert Flow Trees : A Technique for Identifying and Analyzing Covert Storage Channels**. Dans : *IEEE Computer Society Symposium on Research in Security and Privacy*. Oakland, CA : IEEE Computer Society, mai 1991, p. 36–51. doi : 10.1109/RISP.1991.130770 (cf. p. 74).

[235] C. H. Rowland. **Covert Channels in the TCP/IP protocol suite**. Dans : *First Monday* 2.5 (mai 1997) (cf. p. 70).

[236] A. Rupp, G. Hinterwälder, F. Baldimtsi et C. Paar. **P4R : Privacy-Preserving Pre-Payments with Refunds for Transportation Systems**. English. Dans : *Financial Cryptography and Data Security*. Sous la dir. d'A.-R. Sadeghi. T. 7859. LNCS. Okinawa, Japan : Springer Berlin Heidelberg, 2013, p. 205–212. doi : 10.1007/978-3-642-39884-1_17 (cf. p. 80).

[237] A. Sadeghi, I. Visconti et C. Wachsmann. **User Privacy in Transport Systems Based on RFID E-Tickets**. Dans : *International Workshop on Privacy in Location-Based Applications*. Sous la dir. de C. Bettini, S. Jajodia, P. Samarati et X. S. Wang. T. 397. Malaga, Spain : CEUR, octobre 2008 (cf. p. 80).

[238] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia et X. Wang. **Soundcomber : A Stealthy and Context-Aware Sound Trojan for Smartphones**. Dans : *Network and Distributed System Security Symposium*. San Diego, California, USA : The Internet Society, février 2011 (cf. p. 70, 76).

[239] C. Schnorr. **Efficient Signature Generation by Smart Cards**. English. Dans : *Journal of Cryptology* 4.3 (1991), p. 161–174. doi : 10.1007/BF00196725 (cf. p. 80, 81).

[240] Sony Corp. Info. *Mobile payment services using NFC SIMs equipped with Sony FeliCa™ technology to begin in Hong Kong*. News Releases. Octobre 2013 (cf. p. 80).

[241] S. Tamrakar et J. Ekberg. **Tapping and Tripping with NFC**. Dans : *6th International Conference on Trust & Trustworthy Computing*. T. 7904. London, United Kingdom : Springer Berlin / Heidelberg, 2013, p. 115–132. doi : 10.1007/978-3-642-38908-5_9 (cf. p. 80).

[170] N. Timmers, A. Spruyt et M. Witteman. **Controlling PC on ARM using Fault Injection**. Dans : *Fault Diagnosis and Tolerance in Cryptography*. Santa Barbara, USA : IEEE Computer Society, août 2016 (cf. p. 45, 67).

[242] M. Urbanski, W. Mazurczyk, J.-F. Lalande et L. Caviglione. **Detecting Local Covert Channels Using Process Activity Correlation on Android Smartphones**. Dans : *International Journal of Computer Systems Science and Engineering* 32.2 (2017) (cf. p. 6, 74, 76).

[243] J. C. Wray. **An Analysis of Covert Timing Channels**. Dans : *Journal of Computer Security* 1 (3-4 1992), p. 219–232 (cf. p. 70).

[244] S. Zander, G. Armitage et P. Branch. **A survey of covert channels and countermeasures in computer network protocols**. Dans : *IEEE Communications Surveys & Tutorials* 9.3 (2007), p. 44–57. doi : 10.1109/COMST.2007.4317620 (cf. p. 70, 74).

[245] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao et L. Yang. **Accurate online power estimation and automatic battery behavior based power model generation for smartphones**. Dans : *International conference on Hardware/software codesign and system synthesis*. University of Leipzig, Germany : IEEE/ACM/IFIP, octobre 2010, p. 105–114. doi : 10.1145/1878961.1878982 (cf. p. 75).

[246] W. Zhou, Y. Zhou, X. Jiang et P. Ning. **Detecting repackaged smartphone applications in third-party android marketplaces**. Dans : *Second ACM conference on Data and Application Security and Privacy*. Sous la dir. d'E. Bertino et R. S. Sandhu. San Antonio, TX, USA : ACM Press, février 2012, p. 317–326. doi : 10.1145/2133601.2133640 (cf. p. 70).

[183] Y. Zhou et X. Jiang. **Dissecting Android Malware : Characterization and Evolution**. Dans : *IEEE Symposium on Security and Privacy*. 4. San Jose, USA : IEEE Computer Society, mai 2012, p. 95–109. doi : 10.1109/SP.2012.16 (cf. p. 42, 43, 69).

Chapitre 5

Conclusion

Ce manuscrit couvre dix années de recherche en sécurité informatique. J'ai couvert un domaine varié de systèmes : les clusters de calcul, les cartes à puce, les téléphones mobiles. Les problématiques sont souvent différentes mais les technologies sous-jacentes sont très proches. Je consacre actuellement la majorité de mon temps de recherche à la sécurité dans les téléphones mobiles. Ce thème est désormais très présent dans les conférences majeures en sécurité : on trouve presque à chaque fois un *track* sur la sécurité mobile dans ces conférences, notamment dans les quatre conférences majeures. L'avenir de ce thème est donc assuré, avec une compétition qui ne cesse de se renforcer : 16% de taux de sélection pour ACM CSS 2016 avec 831 travaux soumis.

Concernant la thématique des logiciel embarqués dans des éléments sécurisés, des mutations technologiques sont en cours. L'émergence des TEE remet en cause l'utilisation des cartes à puce pour embarquer des logiciels autres que ceux qui servent à s'enregistrer sur le réseau téléphonique. Pourtant les cartes à puce servent déjà à déployer de tels logiciels dans d'autres contextes que le téléphone mobile. Il n'est sans nul doute pas facile de trouver un modèle économique satisfaisant pour les opérateurs téléphoniques. Cependant, les TEE peinent à s'imposer sur le marché. L'articulation entre les applications du système d'exploitation normal et du TEE offre de nouveaux problèmes de recherche intéressants. L'enjeu est de proposer un nouveau service à l'utilisateur, éventuellement partiellement hébergé comme une application normale du téléphone, tout en s'appuyant sur le haut niveau de sécurité offert par le TEE. En plus de la problématique du déploiement de telles applications [247], il faut définir les propriétés de sécurité attendues et s'assurer que les attaques possibles, depuis le système d'exploitation riche ou depuis l'extérieur du téléphone sont contenues. Il n'est pas certain que les contributions espérées seront utilisées. Dans le contexte du monde PC, l'implantation de TPM n'a été que très marginalement utilisé. Cependant, je pense que l'essor de l'internet des objets va renforcer le besoin et que les applications seront de plus en plus bicéphales : une partie hébergée dans un environnement sécurisé et l'autre non.

Concernant la thématique des *malware* en environnement mobile, de nombreuses questions sont encore ouvertes. L'analyse fine des comportements malveillants est un problème difficile parce qu'il est plus ambitieux qu'un problème de décision. De nombreux papiers proposent des méthodes de classification, mais décrire les actions opérées par une des classes de *malware* apporte des informations plus précises pour l'analyste en sécurité. De surcroît, un *malware* est un logiciel qui se périme vite, notamment à cause des montées de version du système d'exploitation. La plupart des *malware* ayant plus de 2 ans et cherchant à exploiter des vulnérabilités ne sont plus efficaces. D'une part la vulnérabilité n'est plus présente, mais les nouvelles défenses embarquées dans Android, par exemple SELinux, rendent les attaques inopérantes. Il est donc prévisible que la sophistication des attaques va considérablement augmenter. Cependant, les attaques qui ne compromettent pas l'intégrité du système sont celles dont l'efficacité n'est pas réellement freinée avec les changements dans le cœur d'Android. Les applications repackagées qui contiennent une partie de code malveillant abusent des permissions de l'application originelle, par exemple pour exfiltrer des données. Ces attaques sont les plus nombreuses et les plus difficiles à analyser et méritent donc un effort de recherche.

Malgré ces mutations et l'effort qu'il faut continuer de fournir pour la compréhension des attaques, plusieurs axes personnels de recherche sont à développer dans les années à venir.

Visualisation de *malware* Android La visualisation des caractéristiques statiques et dynamiques des *malware* est un point crucial à développer rapidement. Il n'existe pas encore de contribution majeure sur cet objectif mettant en avant les caractéristiques particulières des applications Android sous une forme visuelle qui permette de représenter une malveillance. Cependant, le sujet a été largement couvert pour les *malware* PC [252]. L'as-

pect dynamique d'une malveillance est aussi à prendre en compte, notamment lorsqu'on s'intéresse à un malware dont le comportement n'est observable qu'à l'exécution. Une représentation efficace doit permettre d'augmenter la productivité d'un analyste et de dégager des méthodes d'investigation pour explorer des archétypes de malveillances. L'effort de recherche doit porter sur la combinaison de méthodes statiques et dynamiques et doit lever le verrou de la complexité d'un code malveillant caché dans une application complexe repackagée. De plus, la qualité de la visualisation d'un *malware* doit être quantifiée rigoureusement, malgré l'absence d'outils existants comme points de comparaison.

Coopération avec des éléments sécurisés A moyen terme, la généralisation d'éléments sécurisés embarqués dans les téléphones ouvre de nouvelles perspectives pour sécuriser des applications. L'enjeu est de définir quelle sécurité peut être fournie depuis un élément sécurisé à une application qui évolue dans un système d'exploitation qui ne l'est pas. Cette idée a déjà été explorée pour la carte à puce [249] mais les résultats restent limités par les capacités de la carte. Une piste intéressante de recherche exploitant cette idée, consiste à héberger une application "personalisée" pour l'utilisateur et surveillée par un autre programme embarqué dans un élément sécurisé. L'application personalisée pourrait être une application diversifiée [248] c'est-à-dire compilée de sorte à la rendre unique pour cet utilisateur, mais rendant un service équivalent. En contrôlant le processus de diversification et en le mariant avec un élément sécurisé qui se charge de surveiller cette version diversifiée, on empêcherait un attaquant de pouvoir appliquer une même attaque sur plusieurs applications de différents utilisateurs. Cette piste de recherche est particulièrement intéressante pour Android, dont le processus de déploiement d'une application utilise une étape de compilation *ahead-of-time*, lors de l'installation.

Auto-protection des applications En attendant l'émergence d'éléments sécurisés, l'industrie cherche des solutions pour intégrer des défenses à l'intérieur même des applications. Les acteurs économiques développant des applications n'a pas le pouvoir d'influer sur la sécurité au niveau du système d'exploitation. Aussi, fournir des outils de sécurité au niveau applicatif est un thème encore émergent de la recherche pour les applications mobiles. En 2014, Zhou et al.[253] proposent de personnaliser le jeu d'instructions utilisé par l'application afin d'éviter une décompilation trop aisée du bytecode. Cependant, cette solution nécessite de modifier la machine virtuelle Dalvik pour fonctionner. En 2015, Falsina et al. [250] proposent un système pour charger du code dynamiquement dans une application et proposent un framework pour réaliser des vérifications sur le code chargé. L'idée est d'éviter de subir des attaques lorsqu'une application doit charger du code dynamiquement. Pour cette contribution, on s'affranchit de la nécessité de modifier le système. Récemment, Sun et al. [251] introduisent la *randomization* de la mémoire afin d'éviter qu'un attaquant puisse réutiliser des gadgets de code situés en mémoire à des endroits prévisibles. L'idée sous-jacente à l'ensemble de ces travaux est que la variabilité de l'exécution d'une application permet de se prémunir d'attaques qui se basent sur la connaissance de l'architecture logicielle : le jeu d'instructions utilisé, la façon de charger du code dynamique, la localisation mémoire des bibliothèques. Dans cet esprit, de larges perspectives de recherche existent pour apporter de la sécurité en exploitant la variabilité dans l'exécution du code. La variabilité peut être introduite directement à l'exécution, ou bien à la phase de compilation de l'application. Un tel mécanisme permet de renforcer l'auto-protection des applications parce qu'il n'est pas nécessaire de changer le système d'exploitation sous-jacent pour cela. Sur cet axe de travail, il faut quantifier le gain obtenu, question à laquelle les travaux mentionnés précédemment n'apportent pas encore de réponse claire.

Références

[247] G. Arfaoui, J.-F. Lalande, S. Gharout et J. Traoré. **Practical and Privacy-Preserving TEE Migration**. Dans : *9th IFIP WG 11.2 International Conference on Information Security Theory and Practice*. Sous la dir. de R. N. Akram et S. Jajodia. T. 9311. LNCS. Heraklion, Greece : Springer, août 2015, p. 153–168. doi : 10.1007/978-3-319-24018-3_10 (cf. p. 6, 89).

[248] B. Baudry et M. Monperrus. **The Multiple Facets of Software Diversity : Recent Developments in Year 2000 and Beyond**. Dans : *ACM Computing Survey* 48.1 (2015), p. 1–26. doi : 10.1145/2807593 (cf. p. 90).

[249] S. Chaumette, O. Ly et R. Tabary. **Automated software protection through program externalization on memory-limited secure devices**. Dans : *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. Hong Kong, China : IEEE Computer Society, décembre 2010, p. 777–784. doi : 10.1109/EUC.2010.122 (cf. p. 90).

[250] L. Falsina, Y. Fratantonio, S. Zanero et C. Kruegel. **Grab'n Run : Secure and Practical Dynamic Code Loading for Android Applications**. Dans : *Annual Computer Security Applications Conference*. Los Angeles, USA, décembre 2015 (cf. p. 90).

[251] M. Sun, J. C. S. Lui et Y. Zhou. **Blender : Self-randomizing Address Space Layout for Android Apps**. Dans : *19th International Symposium on Research in Attacks, Intrusions and Defenses*. T. 9854. LNCS. Paris, France : Springer Berlin / Heidelberg, septembre 2016. doi : 10.1007/978-3-319-45719-2_21 (cf. p. 90).

[252] M. Wagner, F. Fischer, R. Luh, A. Haberson, A. Rind, D. A. Keim et W. Aigner. **A Survey of Visualization Systems for Malware Analysis**. Dans : *EuroVis* (2015). doi : 10.2312/eurovisstar.20151114 (cf. p. 89).

[253] W. Zhou, Z. Wang, Y. Zhou et X. Jiang. **DIVILAR : diversifying intermediate language for anti-repackaging on android platform**. Dans : *4th ACM Conference on Data and Application Security and Privacy*. San Antonio, TX, USA : ACM Press, 2014, p. 199–210. doi : 10.1145/2557547.2557558 (cf. p. 90).

Bibliographie personnelle

Articles dans des journaux internationaux

[186] G. Arfaoui, G. Dabosville, S. Gambs, P. Lacharme et J.-F. Lalande. **A Privacy-Preserving NFC Mobile Pass for Transport Systems**. Dans : *EAI Endorsed Transactions on Mobile Communications and Applications* 14.5 (décembre 2014), e4. doi : 10.4108/mca.2.5.e4 (cf. p. 6, 79).

[189] PoPETS
G. Arfaoui, J.-F. Lalande, J. Traoré, N. Desmoulins, P. Berthomé et S. Gharout. **A Practical Set-Membership Proof for Privacy-Preserving NFC Mobile Ticketing**. Dans : *Proceedings on Privacy Enhancing Technologies* 2015.2 (janvier 2015), p. 25–45. doi : 10.1515/popets-2015-0019 (cf. p. 6, 79, 81).

[12] FGCS
M. Blanc et J.-F. Lalande. **Improving Mandatory Access Control for HPC clusters**. Dans : *Future Generation Computer Systems* 29.3 (avril 2013), p. 876–885. doi : 10.1016/j.future.2012.03.020 (cf. p. 6, 17).

[19] J. Briffaut, J.-F. Lalande et C. Toinard. **Formalization of security properties : enforcement for MAC operating systems and verification of dynamic MAC policies**. Dans : *International journal on advances in security* 2.4 (2009), p. 325–343 (cf. p. 6, 24, 26).

[20] JCP
J. Briffaut, J.-F. Lalande et C. Toinard. **Security and Results of a Large-Scale High-Interaction Honeypot**. Dans : *Journal of Computers* 4.5 (2009), p. 395–404. doi : 10.4304/jcp.4.5.395-404 (cf. p. 6, 13).

[197] TIFS
L. Caviglione, M. Gaggero, J.-F. Lalande, W. Mazurczyk et M. Urbanski. **Seeing the Unseen : Revealing Mobile Malware Hidden Communications via Energy Consumption and Artificial Intelligence**. Dans : *IEEE Transactions on Information Forensics and Security* 11.4 (2016), p. 799–810. doi : 10.1109/TIFS.2015.2510825 (cf. p. 6, 74, 78, 82).

[254] IJNSA
J.-F. Lalande, D. Rodriguez et C. Toinard. **Security properties in an open peer-to-peer network**. Dans : *International Journal of Network Security & Its Applications* 1.3 (avril 2010), p. 73–89. arXiv : 1004.0772 (cf. p. 6).

[66] FGCS
W. W. Smari, P. Clemente et J.-F. Lalande. **An extended attribute based access control model with trust and privacy : Application to a collaborative crisis management system**. Dans : *Future Generation Computer Systems* 31.- (février 2014), p. 147–168. doi : 10.1016/j.future.2013.05.010 (cf. p. 6, 21).

[242] M. Urbanski, W. Mazurczyk, J.-F. Lalande et L. Caviglione. **Detecting Local Covert Channels Using Process Activity Correlation on Android Smartphones**. Dans : *International Journal of Computer Systems Science and Engineering* 32.2 (2017) (cf. p. 6, 74, 76).

[255] S. Wendzel, L. Caviglione, W. Mazurczyk et J.-F. Lalande. **Network Information Hiding and Science 2.0 : Can it be a Match ?** Dans : *International Journal of Electronics and Telecommunications* 63.2 (2017), p. 217–222. doi : 0.1515/eletel-2017-0029.

Chapitres de livres

[256] S. Alouf, E. Altman, J. Galtier, J.-F. Lalande et C. Touati. **Quasi-Optimal Resource Allocation in Multi-Spot MFTDMA Satellite Networks**. Dans : *Combinatorial Optimization in Communication Networks*. Sous la dir. de D.-Z. D. Maggie Xiaoyan Cheng, Yingshu Li. Combinatorial Optimization. Springer Berlin Heidelberg, 2006, p. 325–365. doi : 10.1007/0-387-29026-5_13.

[16] J. Briffaut, P. Clemente, J.-F. Lalande et J. Rouzaud-Cornabas. **Honeypot forensics for system and network SIEM design**. Dans : *Advances in Security Information Management : Perceptions and Outcomes*. Sous la dir. de G. S. de Tangil et E. Palomar. Computer Networks and Computer Science, Technology and Applications. Nova Science Publishers, 2013. Chap. 8, p. 181–216 (cf. p. 6, 14).

[257] L. Caviglione, M. Gaggero, J.-F. Lalande et W. Mazurczyk. **Understanding Information Hiding to Secure Communications and to Prevent Exfiltration of Mobile Data**. Dans : *Adaptive Mobile Computing : Advances in Processing Mobile Data Sets*. Sous la dir. de M. Migliardi, A. Merlo et S. Al-Haj Baddar. Elsevier, 2017, To appear.

Articles dans des conférences internationales

[84] **MALWARE 2015**
A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande et V. Viet Triem Tong. **GroddDroid : a Gorilla for Triggering Malicious Behaviors**. Dans : *10th International Conference on Malicious and Unwanted Software*. Fajardo, Puerto Rico : IEEE Computer Society, octobre 2015, p. 119–127. doi : 10.1109/MALWARE.2015.7413692 (cf. p. 6, 38, 39, 43, 55).

[258] **INFOCOM 2005**
S. Alouf, E. Altman, J. Galtier, J.-F. Lalande et C. Touati. **Quasi-optimal bandwidth allocation for multi-spot MFTDMA satellites**. Dans : *IEEE Conference on Computer Communications*. T. 1. Miami États-Unis : IEEE Computer Society, mars 2005, p. 560–571. doi : 10.1109/INFCOM.2005.1497923.

[259] **MobiCASE 2013**
G. Arfaoui, S. Gambs, P. Lacharme, J.-F. Lalande, L. Roch et J.-C. Paillès. **A Privacy-Preserving Contactless Transport Service for NFC Smartphones**. Dans : *Fifth International Conference on Mobile Computing, Applications and Services*. Sous la dir. de G. Memmi et U. Blanke. T. 130. LNICST. Paris, France : Springer Berlin / Heidelberg, novembre 2013, p. 282–285. doi : 10.1007/978-3-319-05452-0_24 (cf. p. 6).

[247] **WISTP 2015**
G. Arfaoui, J.-F. Lalande, S. Gharout et J. Traoré. **Practical and Privacy-Preserving TEE Migration**. Dans : *9th IFIP WG 11.2 International Conference on Information Security Theory and Practice*. Sous la dir. de R. N. Akram et S. Jajodia. T. 9311. LNCS. Heraklion, Greece : Springer, août 2015, p. 153–168. doi : 10.1007/978-3-319-24018-3_10 (cf. p. 6, 89).

[102] **AReS 2012**
P. Berthome, K. Heydemann, X. Kauffmann-Tourkestansky et J.-F. Lalande. **High Level Model of Control Flow Attacks for Smart Card Functional Security**. Dans : *Seventh International Conference on Availability, Reliability and Security*. Prague, Czech Republic : IEEE Computer Society, août 2012, p. 224–229. doi : 10.1109/ARES.2012.79 (cf. p. 6, 44, 46).

[260] **HPCS 2010**
M. Blanc et J.-F. Lalande. **Mandatory Access Control for shared HPC clusters : Setup and performance evaluation**. Dans : *International Conference on High Performance Computing & Simulation*. Caen, France : IEEE Computer Society, juin 2010, p. 291–298. doi : 10.1109/HPCS.2010.5547118 (cf. p. 6).

[261] **SIROCCO 2003**
M. Bouklit, D. Coudert, J.-F. Lalande, C. Paul et H. Rivano. **Approximate Multicommodity Flow for WDM Networks Design**. Dans : *Colloquium on Structural Information and Communication Complexity*. Umeä (Sweden) : Carleton Scientific, juin 2003, p. 43–56.

[14] SECRYPT 2011
A. Bousquet, P. Clemente et J.-F. Lalande. **SYNEMA : visual monitoring of network and system security sensors**. Dans : *International Conference on Security and Cryptography*. Sous la dir. de J. Lopez et P. Samarati. Seville, Spain : SciTePress, juillet 2011, p. 375–378. doi : 10.5220/0003516203750378 (cf. p. 6, 15, 27).

[17] SECURWARE 2009
J. Briffaut, X. Kauffmann-Tourkestansky, J.-F. Lalande et W. Smari. **Generation of role based access control security policies for Java collaborative applications**. Dans : *Third International Conference on Emerging Security Information, Systems and Technologies*. Athens/Glyfada, Greece : IEEE Computer Society, juin 2009, p. 224–229. doi : 10.1109/SECURWARE.2009.41 (cf. p. 6, 21, 28).

[18] SECURWARE 2008
J. Briffaut, J.-F. Lalande et W. Smari. **Team-based MAC policy over Security-Enhanced Linux**. Dans : *Second International Conference on Emerging Security Information, Systems and Technologies*. Cap Esterel France : IEEE Computer Society, août 2008, p. 41–46. doi : 10.1109/SECURWARE.2008.35 (cf. p. 6, 21).

[262] SECURWARE 2009
J. Briffaut, J.-F. Lalande, C. Toinard et M. Blanc. **Enforcement of Security Properties for Dynamic MAC Policies**. Dans : *Third International Conference on Emerging Security Information, Systems and Technologies*. Athens/Glyfada, Greece : IEEE Computer Society, juin 2009, p. 114–120. doi : 10.1109/SECURWARE.2009.25 (cf. p. 6).

[198] HAS 2015
L. Caviglione, J.-F. Lalande, W. Mazurczyk et S. Wendzel. **Analysis of Human Awareness of Security and Privacy Threats in Smart Environments**. Dans : *3rd International Conference on Human Aspects of Information Security, Privacy and Trust*. Sous la dir. de T. Tryfonas et I. G. Askoxylakis. T. 9190. LNCS. Los Angeles, USA : Springer Berlin / Heidelberg, août 2015, p. 165–177. doi : 10.1007/978-3-319-20376-8_15. arXiv : 1502.00868 (cf. p. 6, 74).

[23] SECRYPT 2012
P. Clemente, J.-F. Lalande et J. Rouzaud-Cornabas. **HoneyCloud : elastic honeypots - On-attack provisioning of high-interaction honeypots**. Dans : *International Conference on Security and Cryptography*. Rome, Italy : SciTePress, juillet 2012, p. 434–439. doi : 10.5220/0004129604340439 (cf. p. 6, 14).

[203] DI 2014
M. De Saint Léger, S. Gambs, B. Juanals, J.-F. Lalande et J.-L. Minel. **Privacy and Mobile Technologies : the Need to Build a Digital Culture**. Dans : *Digital Intelligence*. Nantes, France : Université de Nantes, septembre 2014, p. 100–105 (cf. p. 6, 68).

[142] ESORICS 2014
J.-F. Lalande, K. Heydemann et P. Berthomé. **Software Countermeasures for Control Flow Integrity of Smart Card C Codes**. Dans : *19th European Symposium on Research in Computer Security*. Sous la dir. de M. Kutylowski et J. Vaidya. T. 8713. LNCS. Wroclaw, Pologne : Springer International Publishing, septembre 2014, p. 200–218. doi : 10.1007/978-3-319-11212-1_12 (cf. p. 6, 38, 47).

Articles dans des workshops internationaux

[193] IWSMA 2012
P. Berthomé, T. Fécherolle, N. Guilloteau et J.-F. Lalande. **Repackaging Android applications for auditing access to private data**. Dans : *First International Workshop on Security of Mobile Applications*. Prague, Czech Republic : IEEE Computer Society, août 2012, p. 388–396. doi : 10.1109/ARES.2012.74 (cf. p. 6, 69).

[103] **PLAS 2010**
P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky et J.-F. Lalande. **Attack model for verification of interval security properties for smart card C codes**. Dans : *5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. Toronto Canada : ACM, juin 2010, p. 1–12. doi : 10.1145/1814217.1814219 (cf. p. 6, 37).

[263] **POLICY 2006**
M. Blanc, J. Briffaut, J.-F. Lalande et C. Toinard. **Distributed control enabling consistent MAC policies and IDS based on a meta-policy approach**. Dans : *Seventh IEEE International Workshop on Policies for Distributed Systems and Networks*. University of Western Ontario, London Canada : IEEE Computer Society, juin 2006, p. 153–156. doi : 10.1109/POLICY.2006.15 (cf. p. 6).

[264] **COLSEC 2006**
M. Blanc, J. Briffaut, J.-F. Lalande et C. Toinard. **Collaboration between MAC policies and IDS based on a meta-policy approach**. Dans : *Workshop on Collaboration and Security*. Sous la dir. de W. W. Smari et W. McQuay. Las Vegas : IEEE Computer Society, mai 2006, p. 48–55. doi : 10.1109/CTS.2006.25 (cf. p. 6).

[265] **COLSEC 2009**
M. Blanc, K. Guérin, J.-F. Lalande et V. Le Port. **Mandatory access control implantation against potential NFS vulnerabilities**. Dans : *Workshop on Collaboration and Security*. Sous la dir. de Waleed W. Smari et William McQuay. Baltimore États-Unis : IEEE Computer Society, mai 2009, p. 195–200. doi : 10.1109/CTS.2009.5067481 (cf. p. 6).

[266] **SHPCS 2008**
J. Briffaut, J.-F. Lalande et C. Toinard. **A proposal for securing a large-scale high-interaction honeypot**. Dans : *Workshop on Security and High Performance Computing Systems*. Sous la dir. de R. K. Guha et L. Spalazzi. Cyprus Chypre : IEEE Computer Society, juin 2008, p. 206–212 (cf. p. 6).

[140] **LASER 2016**
N. Kiss, J.-F. Lalande, M. Leslous et V. Viet Triem Tong. **Kharon dataset : Android malware under a microscope**. Dans : *The LASER Workshop : Learning from Authoritative Security Experiment Results*. San Jose, United States : USENIX Association, mai 2016, p. 1–12 (cf. p. 6, 40–42, 69).

[267] **SERTIF 2016**
J.-F. Lalande. **Contre-mesures logicielles contre les fautes induisant des sauts**. Dans : *Workshop SERTIF : Simulation pour l'Evaluation de la RobusTesse des applications embarquées contre l'Injection de Fautes*. Grenoble, France, octobre 2016.

[268] **METHOD 2012**
J.-F. Lalande et D. Rodriguez. **Protecting resources in an open and trusted peer-to-peer network**. Dans : *The 1st IEEE International Workshop on Methods for Establishing Trust with Open Data*. Izmir, Turkey : IEEE Computer Society, juillet 2012, p. 140–143. doi : 10.1109/COMPSACW.2012.35 (cf. p. 6).

[221] **ECTCM 2014**
J.-F. Lalande et S. Wendzel. **Hiding privacy leaks in Android applications using low-attention raising covert channels**. Dans : *First International Workshop on Emerging Cyberthreats and Countermeasures*. Regensburg, Germany : IEEE Computer Society, septembre 2013, p. 701–710. doi : 10.1109/ARES.2013.92 (cf. p. 6, 71, 74, 82).

Conférences internationales invitées

[220] **CISSI 2014**
J.-F. Lalande. **Un titre de transport sur mobile NFC respectueux de la vie privée**. Dans : *Colloque International sur la Sécurité des Systèmes d'Information*. Kénitra, Morocco, mars 2014 (cf. p. 83).

[269] **CISSI 2015**
J.-F. Lalande. **Sécurité Android : exemples de malware**. Dans : *Colloque International sur la Sécurité des Systèmes d'Information*. Kénitra, Morocco, mars 2015.

[270] CISSI 2016
J.-F. Lalande. **Malware à base de canaux auxiliaires**. Dans : *Colloque International sur la Sécurité des Systèmes d'Information*. Kénitra, Morocco, décembre 2016.

Communications sans actes

[271] RESSI 2015
R. Andriatsimandefitra Ratsisahanana, T. Genet, L. Guillo, J.-F. Lalande, D. Pichardie et V. Viet Triem Tong. **Kharon : Découvrir, comprendre et reconnaitre des malware Android par suivi de flux d'information**. Dans : *Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. Troyes, France, mai 2015.

[187] APVP 2014
G. Arfaoui, G. Dabosville, S. Gambs, P. Lacharme et J.-F. Lalande. **Un pass de transport anonyme et intraçable pour mobile NFC**. Dans : *Atelier sur la Protection de la Vie Privée 2014*. Cabourg, France, juin 2014 (cf. p. 83).

[272] APVP 2014
G. Arfaoui et J.-F. Lalande. **A Privacy Preserving Post-Payment Mobile Ticketing Protocol for Transport Systems**. Dans : *Atelier sur la Protection de la Vie Privée 2014*. Cabourg, France, juin 2014.

[273] 3SL 2011
P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky et J.-F. Lalande. **Attaques physiques à haut niveau pour le test de la sécurité des cartes à puce**. Dans : *Journée Sécurité des Systèmes & Sûreté des Logiciels*. Saint-Malo, France, mai 2011, p. 13–14.

[274] e-Smart 2011
P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky et J.-F. Lalande. **Simulating physical attacks in smart card C codes : the jump attack case**. Dans : *e-Smart : The Future of Digital Security Technologies*. Nice - Sophia Antipolis, France, septembre 2011.

[275] APVP 2012
P. Berthomé et J.-F. Lalande. **Comment ajouter de la privacy after design pour les applications Android ?** Dans : *Atelier Protection de la Vie Privée*. Ile de Groix, France, juin 2012.

[209] RESSI 2015
S. Gambs, J.-F. Lalande et J. Traoré. **ANR LYRICS : Cryptographie pour la protection de la vie privée, optimisée pour les services mobiles sans contact**. Dans : *Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*. Troyes, France, mai 2015 (cf. p. 83).

[276] COSADE'2013
J.-F. Lalande et S. Wendzel. **Attacking Smartphone Privacy Using Local Covert Channels**. Dans : *Fourth International Workshop on Constructive Side-Channel Analysis and Secure Design*. Paris, France, mars 2013.

Articles dans des conférences nationales

[277] AlgoTel 2003
M. Bouklit, D. Coudert, J.-F. Lalande et H. Rivano. **Approximation Combinatoire de Multiflot Factionnaire : Améliorations**. Dans : *5ièmes Rencontres Francophones sur les Aspects ALGOrithmiques des TELécommunications*. Banyuls-sur-Mer (France), mai 2003.

[278] ROADEF 2003
J.-F. Lalande, S. Pérennes et M. Syska. **Groupage dans les réseaux dorsaux WDM**. Dans : *Congrès annuel de la Société française de recherche opérationnelle et d'aide à la décision*. Avignon France : Université d'Avignon et des Pays de Vaucluse, février 2003, p. 254–255.

[279] ROADEF 2005
J.-F. Lalande, M. Syska et Y. Verhoeven. **Arrondi aléatoire et protection des réseaux WDM**. Dans : *Congrès annuel de la Société française de recherche opérationnelle et d'aide à la décision*. Sous la dir. de Jean-Charles Billaut et Carl Esswein. Tours France : Tours : Presses universitaires François Rabelais, février 2005, p. 241–242.

Articles dans des magazines

[280] V. Viet Triem Tong, J.-F. Lalande et M. Leslous. **Challenges in Android Malware Analysis**. Dans : *ERCIM News*. Special Theme : Cybersecurity 106 (2016), p. 42–43.

Thèse de doctorat

[281] J.-F. Lalande. **Conception de réseaux de télécommunications : optimisation et expérimentations**. Thèse de doctorat. Université de Nice Sophia-Antipolis, 2004 (cf. p. 5).

[282] J.-F. Lalande. **Vers la sécurité mobile : caractérisation des attaques et contremesures**. Habilitation à diriger des recherches. Université d'Orléans, novembre 2016.

Rapports techniques ou de recherche

[283] S. Alouf, E. Altman, J. Galtier, J.-F. Lalande et C. Touati. **Un algorithme d'allocation de bande passante satellitaire**. Rapport technique. Sophia-Antipolis : INRIA, 2004.

[284] P. Berthomé, J.-F. Lalande et V. Levorato. **Implementation of exponential and parametrized algorithms in the AGAPE project**. Rapport technique. Université d'Orléans - ENSI de Bourges, 2012.

[285] J. Briffaut, P. Clemente, J.-F. Lalande et J. Rouzaud-Cornabas. **From Manual Cyber Attacks Forensic to Automatic Characterization of Attackers' Profiles**. Rapport technique. Université d'Orléans, 2011.

[286] J.-F. Lalande, M. Syska et Y. Verhoeven. **Mascopt - A Network Optimization Library : Graph Manipulation**. Rapport technique. Sophia-Antipolis : INRIA, 2004.

Liste des figures

Liste des tableaux

Liste des listings

Annexes

1 Attaque transiente implémentée en assembleur

Réalisation d'une attaque sur un code assembleur consistant à sauter d'une ligne A à une ligne B du code. Contrainte : L'attaque doit être réalisée une unique fois lors du x-ième passage sur le hack et être complètement inopérante sur les passages précédents et suivants. Choix du numéro de passage : quand attaquer lors de la comparaison dans le hack (ici 4ème passage) : *cmpl $4, %eax*.

```
/*** P. Berthome - 14/04/11 */
.LCJUMP:
.string "Jumping to: %i\n"
.text
.globl errhack
.type errhack, @function
errhack:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $.LCJUMP, %edx
movl stderr, %eax
movl $1, 8(%esp)
movl %edx, 4(%esp)
movl %eax, (%esp)
call fprintf
leave
ret
.size .errhack, .-errhack
/*** Fin integration erreur ***/
/*** Integration variables globales ***/
.globl CPT /* Compteur de passage dans le hack */
.bss
.align 4
.type CPT, @object
.size CPT, 4
CPT:
.zero 4 /* Initialise a 0 */
.globl TAMPON /* Sauvegarde du registre eax */
.align 4
.type TAMPON, @object
.size TAMPON, 4
TAMPON:
.zero 4
.section
.rodata
/*** Fin integration variables globales ***/
.hack:
/******** DEBUT HACK LINEAIRE **********/
movl %eax, TAMPON /* On stocke dans le TAMPON ce qui est dans eax */
movl CPT, %eax /* On met CPT (0) dans eax */
addl $1, %eax /* On ajoute 1 a eax */
movl %eax, CPT /* On remet eax dans CPT qui a donc ete incremente */
cmpl $4, %eax /* la valeur choisie est comparee a eax, c'est ici que l'on decide
lors de quel passage l'attaque est jouee */
jne
.rejeu /* Si le tour ne correspond pas, l'attaque n'est pas executee, lors du
passage voulu, on ne jump pas */
call errhack
movl TAMPON, %eax /* on restitue eax comme au debut */
jmp .desthack
.rejeu:
movl TAMPON, %eax /* on restitue eax comme au debut */
jmp .finhack
/******* FIN HACK LINEAIRE ************/
```

2 Systèmes de transitions pour le if-then-else et le while

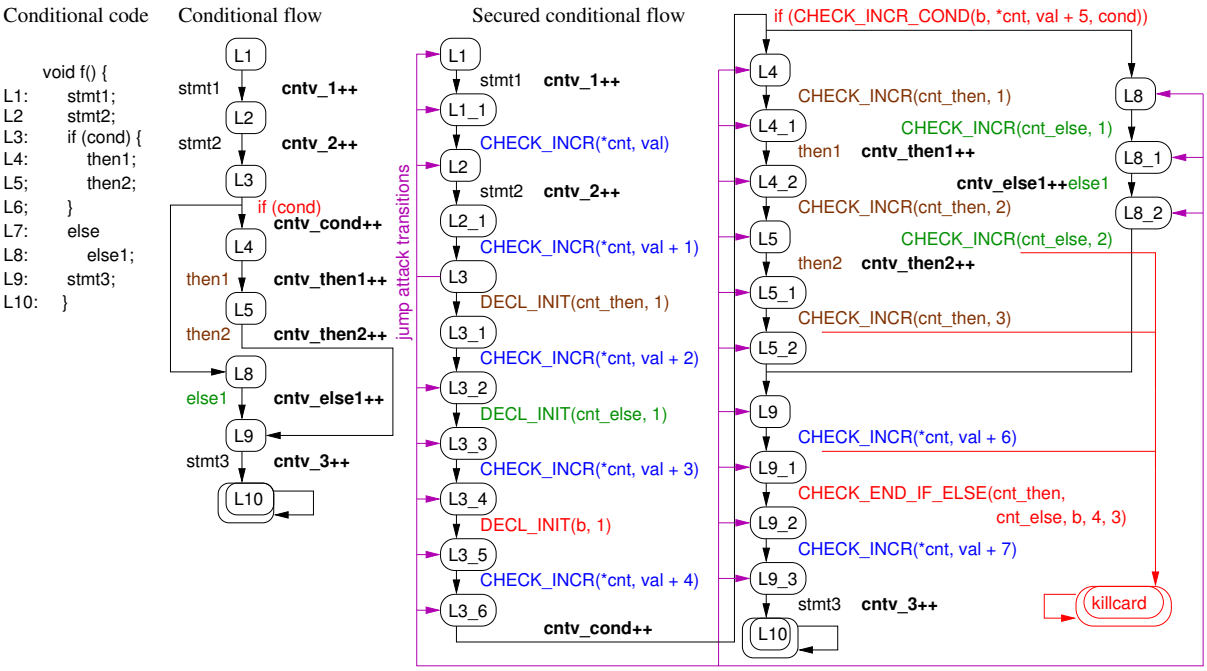


FIGURE 5.1 – Représentation compacte de TS pour une conditionelle if-then-else

```
; P1 : final state reachability in M and CM
AG(AF(M.pc = L10));
AG(AF(CM.pc = L10 + CM.pc = killcard));

; P2 : equal statement execution counts in CM and M when reaching a correct final state
stmt_exec_count_eq :=
      (M.cntv_1 = CM.cntv_1) . (M.cntv_2 = CM.cntv_2) . (M.cntv_cond = CM.cntv_cond)
    . (M.cntv_then1 = CM.cntv_then1) . (M.cntv_then2 = CM.cntv_then2)
    . (M.cntv_else1 = CM.cntv_else1) . (M.cntv_3 = CM.cntv_3) ;
AG(((M.pc = L10) . (CM.pc = L10)) -> stmt_exec_count_eq = 1);

; P3': In each state of the M model, control flow is correct
right_flow_c :=
     Before(M.cntv_1, M.cntv_2) . Before(M.cntv_2, M.cntv_cond) // P3_1
   . Before(M.cntv_cond, M.cntv_then1) . Before(M.cntv_then1, M.cntv_then2) // P3_2 for the then branch
   . Before(M.cntv_cond, M.cntv_else1) // P3_2 for the else branch
   . ((M.cntv_3 = 0) + ((cond = 0) . (M.cntv_else1 = 1))^((cond = 1).(M.cntv_then2 = 1))) // P3_3
AG(right_flow_c = 1);

; P3: In each state of the CM model, control flow is correct or an attack will be detected
right_flow_cm :=
    Before(CM.cntv_1, CM.cntv_2) . Before(CM.cntv_2, CM.cntv_cond) // P3_1
   . Before(CM.cntv_cond, CM.cntv_then1) . Before(CM.cntv_then1, CM.cntv_then2) // P3_2 for the then branch
   . Before(CM.cntv_cond, CM.cntv_else1) // P3_2 for the else branch
   . ((CM.cntv_3 = 0) + ((cond = 0) . (M.cntv_else1 = 1))^((cond = 1).(M.cntv_then2 = 1))) // P3_3
AG(right_flow_cm = 1 + AG(AF(CM.pc = killcard)));
```

FIGURE 5.2 – Propriétés du *model checker* pour une conditionelle if-then-else

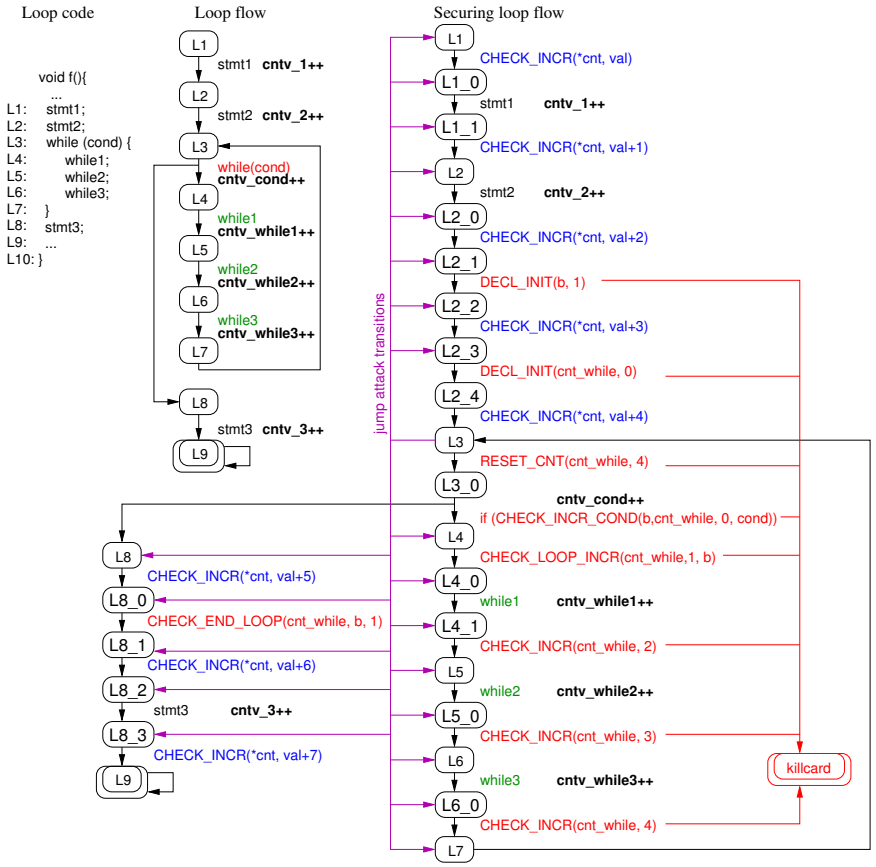


FIGURE 5.3 – Représentation compacte de TS pour une boucle

```
; P1 : final state reachability
AG(AF(M.pc=L9))
AG(AF(CM.pc=L9 + CM.pc=killcard))

; P2 : right statement execution counts in CM and M when reaching a correct final state
stmt_exec_count_eq :=
      (M.cntv_1 = CM.cntv_1) . (M.cntv_2 = CM.cntv_2) . (M.cntv_3 = CM.cntv_3)
    . (M.cntv_cond = CM.cntv_cond) . (M.cntv_while1 = CM.cntv_while1)
    . (M.cntv_while2 = CM.cntv_while2) . (M.cntv_while3 = CM.cntv_while3)
AG( ((M.pc = L9) . (CM.pc = L9)) -> stmt_exec_count_eq = 1);

; P3: The flow is correct at any time during execution in M
right_flow_c :=
     Before(M.cntv_1, M.cntv_2) . Before((M.cntv_2, M.cntv_3) // P3_1 stmt1/2 before stmt2/3
    . ((M.cntv_cond = 0) + (M.cntv_2 = 1)) // P3_2 stmt2 before cond
    . ((M.cntv_cond > 0) + (M.cntv_3 = 0))) // P3_3 cond before stmt3
    . ((M.cntv_3 = 0) + (M.cntv_cond = M.cntv_while1+1).(cond = 0)) // P3_3 stmt3 after cond and the end
      of loop
    . Before(M.cntv_cond, M.cntv_while1) // P3_4 while1 after cond
    . Before(M.cntv_while1, M.cntv_while2) // P3_4 while2 after while1
    . Before(M.cntv_while2, M.cntv_while3) // P3_4 while3 after while2
AG(right_flow_c = 1);

; P3': The flow is correct at any time during execution in CM or an attack will be detected
right_flow_cm :=
     Before(CM.cntv_1, CM.cntv_2) . Before((CM.cntv_2, CM.cntv_3) // P3_1/2 stmt1 before stmt2/3
    . ((CM.cntv_cond = 0) + (CM.cntv_2 = 1)) // P3_2 stmt2 before cond
    . ((CM.cntv_cond > 0) + (CM.cntv_3 = 0))) // P3_3 cond before stmt3
    . ((CM.cntv_3 = 0) + (CM.cntv_cond = CM.cntv_while1+1).(cond = 0)) // P3_3 stmt3 after cond and the
      end of loop
    . Before(CM.cntv_cond, CM.cntv_while1) // P3_4 while1 after cond
    . Before(CM.cntv_while1, CM.cntv_while2) // P3_4 while2 after while1
    . Before(CM.cntv_while2, CM.cntv_while3) // P3_4 while3 after while2
AG(right_flow_cm = 1 + AG(AF(CM.pc = killcard)));
```

FIGURE 5.4 – Propriétés du *model checker* pour une boucle