Étude de la sécurité sous Android

Thomas Duboucher

Rapport de stage

Étude de la sécurité sous Android

Stagiaire :
Duboucher Thomas

Encadrement :
Bernard Mathieu

Mastère recherche informatique – parcours *Sécurité des contenus et des infrastructures informatiques*

27 *Janvier* 2010

Chapitre 1

Introduction

Aujourd'hui, la convergence des technologies a transformé ce qui était autrefois un simple téléphone portable en un véritable ordinateur de poche, au propre comme au figuré.

Ce que dans nos poches nous appelons un *smartphone* ou « ordiphone » mérite bien son nom. Tout d'abord, dans l'usage que l'on en fait, que ce soit pour jouer, consulter ses mails ou encore naviguer sur le web. Du point de vue conception aussi, puisqu'ils mettent maintenant à profit des processeurs qui sont aujourd'hui aussi présents dans les *netbooks*. Ils disposent même maintenant de systèmes d'exploitations complexes en lieu et place du *firmware*.

Cependant, le téléphone portable est aussi un peu plus que cela. Il est présent physiquement avec nous toute la journée, suit nos discussions et nos habitudes et peut aussi proposer des services payants.

C'est dans cet univers là qu'est sorti en 2008 Android, un système d'exploitation libre basé sur Linux, pour équiper nos smartphones. Son code source ouvert et la disponibilité de téléphones relativements ouverts lui ont valu un succès rapide parmi les développeurs

Le ministère de la Défense s'est alors naturellement intéressé aux possibilités offertes par Android, en particulier concernant la sécurisation des équipements mobiles. Durant ce stage, nous avons souhaité vérifier la possibiliter de mettre à profit ce système ouvert afin d'utiliser des politiques de sécurité multi-niveaux pour garantir la confidentialité de l'information.

Ce rapport est organisé de la façon suivante. Le chapitre 2 offrira un état de l'art des mécanismes de sécurité présents dans les équipements mobiles ainsi que plus particulièrement sous Android. Le chapitre 3 explicitera les objectifs du stage avant de présenter le modèle qui aura été développé au cours du stage dans le chapitre 4. Enfin, le chapitre 5 donnera les détails de l'implémentation et les résultats avant de conclure.

Table des matières

Chapitre 2

État de l'art

Ce premier chapitre présente un état de l'art de la sécurité sur les systèmes informatiques, ainsi que l'état de l'emploi de ces systèmes existants sur des terminaux mobiles. Au cours des trois premières parties, nous nous concentrerons successivement sur l'utilisation de politiques de sécurité, sur les différentes techniques de virtualisation existante ainsi que l'informatique de confiance. Nous étudions dans la dernière partie le modèle de sécurité existant sur Android et nous y présentons certains travaux existant visant à renforcer ce modèle.

2.1 Politiques de sécurité

Aujourd'hui, les Operating Systems (OSes) sur plate-formes PC, comme GNU/-Linux utilisent des politiques de sécurité de type *Discretionary Access Control* (DAC) [HRU76]. Dans ce modèle, un ensemble de sujets du système se voient fournir sur un ensemble d'objets du système des permissions sous la forme d'une matrice de contrôle d'accès $\mathcal{A}$, où $\mathcal{A}(s_i, o_j)$ représente les droits que possède le sujet s_i sur l'objet o_j. Le but de cette politique est de contrôler que les sujets effectuent sur les objets des actions qui sont spécifiquement autorisées.

Dans le cadre des systèmes Unix, les sujets sont des utilisateurs du système, les objets des fichiers présents dans le *Virtual File System* (VFS) [1] et les droits se limitent à `rwx` [2] [3].

Un autre type de politique de sécurité répandu est le modèle *Mandatory Access Control* (MAC). Celui-ci cherche à contrôler la façon dont circule l'information afin de détecter les flots suspects. Dans le cas de la confidentialité, c'est à dire de vérifier qu'une information confidentielle ne circule pas par un canal non-autorisé, le modèle utilisé est celui de Bell et LaPadula [LB96].

Ici, les sujets reçoivent des niveaux d'habilitation et les objets des niveaux de classification. La politique interdit ensuite la lecture si le niveau d'habilitation du sujet est inférieur au niveau de classification de l'objet ; similairement, l'écriture requiert un niveau d'habilitation inférieur au niveau de classification. Ainsi, l'information ne peut circuler que dans le sens du niveau de classification croissant.

1. dans la philosophie Unix, tout est fichier
2. `read`, `write`, `execute`
3. les politiques DAC sur les systèmes récents sont plus raffinées, introduisant par exemple la notion de groupes

De la même façon, il est possible d'utiliser des modèles MAC pour assurer l'intégrité. Le modèle de Biba [BM77] permet de contrôler qu'aucune information de faible intégrité ne soit mélangée avec une information de forte intégrité. Pour cela, on inverse les contraintes du modèle de Bell et LaPadula ; l'écriture n'est autorisée que vers des objets d'intégrité moindres et la lecture seulement depuis les objets de plus forte intégrité.

À ce jour, plusieurs *frameworks* fournissant des politiques MAC sont disponibles sous Linux. On pourra citer à titre d'exemple SELinux, Tomoyo ou encore Smack, tous trois basés sur le *framework Linux Security Modules* (LSM).

2.1.1 SELinux

SELinux permet la mise en place de plusieurs types de politiques de sécurité, parmis lesquelles la plus classique est *Domain Type Enforcement* (DTE). Dans ce modèle, les éléments du système reçoivent un identifiant supplémentaire orthogonal à la politique DAC déjà présente ; on appelle *domaine* un identifiant qui porte sur un sujet (processus) et *type* un identifiant qui porte sur un objet (fichier). La façon dont les identifiants sont données est donnée à l'aide des fichiers de configuration de SELinux.

Par défaut, toute tentative de la part d'une entité d'accéder à un objet est bloquée. Celle-ci doit être explicitement autorisée par la politique. Néanmoins, pour ajouter de la souplesse au système, il est possible d'effectuer une transition d'un domaine à un autre. Celle-ci peut se faire soit implicitement, dans ce cas elle est prévue dans la politique, soit explicitement.

Par exemple, lors du démarrage d'un serveur HTTP, une transition se fait implicitement vers le domaine `http_d` sous lequel s'exécute alors le serveur. La configuration de SELinux donne le type `http_conf_t` aux fichiers de configurations du serveur présents dans `/etc/httpd/` et le type `http_t` aux fichiers présents dans `/srv/http/`. Pour finir, le domaine `http_d` a les droits en lecture sur les types `http_conf_t` et `http_t` pour pouvoir lire sa configuration et le contenu des sites qu'il met à disposition.

SELinux présente cependant le désavantage d'avoir été conçu pour des plateformes PC ; son *overhead* et son exhaustivité sont peu adaptés à des équipements mobiles sous Linux. Plusieurs projets, comme Tomoyo [4] ou Smack [5], se reposent sur le *framework* LSM introduit par SELinux pour proposer une alternative légère pour ces environnements.

2.1.2 MSSF

[Kas] présente *Mobile Simplified Security Framework* (MSSF), un *framework* complet pour équipements mobiles, plus large que du contrôle d'accès, mais qui implémente un système léger basé sur LSM. La partie *contrôle d'accès* étend la politique DAC du système en ajoutant des *resource tokens*.

Chaque resource devant être protégée est déclarée avec un ou plusieurs *resource tokens* ; De la même façon, chaque application devant accéder à une resource spécifique doit le déclarer. Cette déclaration se fait par le biais d'un manifeste (2.1, 2.2) présent lors de l'installation de l'application. Il est par ailleurs possible de créer des *resource tokens* qui seront interne à une application.

4. `http://tomoyo.sourceforge.jp/index.html.en`
5. `http://www.schaufler-ca.com/`

```
<mssf>
  <provide>
    <credential name="UserData" />
  </provide>
</mssf>
```

Listing 2.1 – MSSF Manifest File - Server

```
<mssf>
  <request>
    <credential name="server-pkg::UserData" />
    <credential name="Cellular" />
    <credential name="CAP::net_admin" />
    <for path="/usr/bin/userdatamanager"/>
    <for path="/usr/bin/userdataclient"/>
  </request>
</mssf>
```

Listing 2.2 – MSSF Manifest File - Client

L'équipement mobile dispose d'une *device security policy* qui permet d'associer des informations permettant de vérifier l'origine d'une application grâce à une signature avec une liste de *resource tokens*. Lors de l'installation, la signature de l'application est vérifie ; En cas de succès, L'intersection du manifeste avec la *device security policy* permet de mettre à jour la politique de sécurité globale de l'équipement mobile.

Le contrôle d'accès se fait au niveau de la communication inter-processus au travers de *sockets* Unix. Lorsque l'application cliente tente de contacter, le *socket* qui est utilisé est étiqueté de façon transparente par *framework*.

L'application serveur a à sa disposition un ensemble de primitives pouvant être utilisées pour vérifier si la communication est effectuée à partir d'une application cliente autorisée (2.3).

```
creds_value_t value;
creds_type_t type;

fd = accept(sockfd, &cli_addr, &clilen);

require_type = creds_str2creds("UserData", &require_value);
ccreds = creds_getpeer(fd);
allow = creds_have_p(ccreds, require_type, require_value);

if (allow) write(fd, MESSAGE("GRANTED\n"));
else write(fd, MESSAGE("DENIED\n"));
```

Listing 2.3 – Vérifications côté serveur

Cette technique est à la fois discrétionaire et intrusive ; Elle demande que chaque application fournissant des services et devant être protégée soit modifiée pour s'adapter à MSSF. Dans la pratique, cela va demander à ce que chaque application que l'on souhaite protéger soit relue et modifiée à la main. Cette approche est difficilement concevable pour un système complexe tel qu'Android.

2.2 Virtualisation

La virtualisation consiste à faire exécuter un système prévu pour une architecture donnée sur une autre architecture ; ce système est dit virtualisé. On distingue la couche applicative virtualisante comme *hôte* du système virtualisé comme *invité*.

2.2.1 Hyperviseurs

Un *Virtual Machine Monitor* (VMM) ou hyperviseur est une couche logicielle qui permet l'exécution concurrente de plusieurs OS sur une même machine physique [Gol74]. Elle fourni en particulier

- une abstraction de la machine physique sous la forme d'une Machine Virtuelle *(Virtual Machine)* (VM) ;
- une isolation entre les VMs.

Un VMM aura pour tâche au minimum de gérer le démarrage des VMs et le partage des resources du système. En particulier, la configuration de la *Memory Management Unit* (MMU) permet la séparation des espaces mémoires des OSes ; La répartition des interruptions est aussi nécessaire en particulier pour le partage du temps. Enfin, l'ensemble réalise aussi la répartition des resources physiques du système, en particulier les périphériques d'entrée-sortie.

On distingue deux type de virtualisations avec hyperviseur

- la *paravirtualisation*, dans laquelle l'invité est modifié pour dialoguer avec son hôte par le biais d'*hypercalls* ;
- la virtualisation complète ou *hardware-assisted virtualization*, qui requiert des mécanismes de protection matériels entre les VMs.

Les deux méthodes nécessitent un matériel spécifique avec un mode d'exécution privilégié pour le VMM ainsi que des instructions privilégiées associées.

2.2.2 Noyau en espace utilisateur

Un noyau en espace utilisateur ou *user-space kernel* est un noyau modifié pour être exécuté dans l'espace utilisateur d'un autre OS. Du fait de sa présence en espace utilisateur, le noyau invité abandonne tout contrôle de l'espace noyau à son hôte et fourni en échange des interfaces avec le noyau hôte. Ainsi, une application du système invité devient à l'exécution un processus du système hôte ; la communication entre l'application et le système invité est émulée à l'aide d'*Inter Process Communication* (IPC) du système hôte.

Un exemple de noyau en espace utilisateur est *L4Linux*[6], un noyau Linux modifié s'exécutant au dessus d'un micro-noyau L4[7]. Un des principaux avantages d'une telle technique étant de réduire à son strict minimum le code devant être vérifié [KEH+09].

On peut citer comme autre exemple User-Mode Linux (UML)[8] qui permet d'exécuter Linux dans un Linux.

2.2.3 Virtualisation par le système d'exploitation

La virtualisation par le système d'exploitation *(Operating System-level Virtualization)*, plus souvent évoquée sous le terme de virtualisation légère, abandonne tout concept de machine virtuelle pour se concentrer uniquement sur la séparation des espaces utilisateurs tout en ayant recours qu'à un seul noyau. Dans un tel système, un noyau unique attribue à chaque ressource un label correspondant à une instance d'espace utilisateur. Ainsi, chaque espace utilisateur est alors clairement séparé des autres et s'exécute souvent sans aucune connaissance du fait qu'il partage ses ressources.

6. http://os.inf.tu-dresden.de/L4/LinuxOnL4/
7. http://os.inf.tu-dresden.de/L4/
8. http://user-mode-linux.sourceforge.net/

Linux supporte nativement ce mécanisme à l'aide de *conteneurs*[9] ou encore à l'aide du *patch VServer*[10].

2.3 Architectures de confiance

La sécurité informatique se base principalement sur trois concepts :

- la *confidentialité*, qui assure qu'une information ne soit connue que par des personnes légitimes ;
- l'*authenticité*, qui prouve un lien entre une information et une « *personne*[11] » ;
- l'*intégrité*, qui montre la non-altération d'une information.

Ces concepts ont par exemple permis l'essors de méthodes formelles de vérifications de protocoles. Cependant, cette approche théorique oublie que la pratique pourrait se résumer à un utilisateur et à son ordinateur. L'ordinateur est toujours disponible pour rendre service à son propriétaire ; En revanche, son propriétaire n'est pas en mesure d'appréhender la complexité de l'ordinateur. *Mon ordinateur fait-il réellement ce qu'il me dit faire ? Puis-je avoir confiance en ce qu'il fait ?*

Ceci a conduit à l'apparition de l'*informatique de confiance*, ou *trusted computing*. Dans ce domaine, le concept central de *confiance*, que l'on peut résumer à « la machine fait ce qu'elle est censé faire, et uniquement cela », a été résolu à l'aide de l'intégrité. Il est impossible de montrer de façon raisonnable à un instant donné l'affirmation précédente ; On peut néanmoins considérer que c'est bien le cas dans un état initial. Dans ce cas là, prouver qu'elle est de confiance revient alors à montrer son intégrité.

En pratique, les approches se restreignent souvent à considérer que le matériel est de confiance ; Seul le logiciel peut avoir été modifié. Cela revient à considérer que le pouvoir de l'attaquant est dénué de toute présence physique. Ce type d'attaquant se manifeste par exemple sous la forme d'un cheval de Troie.

Créer un logiciel capable de prouver qu'il est de confiance est un casse tête insolvable. Si celui-ci est de confiance, il affirmera toujours qu'il l'est ; Si il ne l'est pas, il affirmera quand même qu'il l'est. Il est néanmoins possible de contourner cette difficulté en ayant recours à un élément matériel [PB03] sous la forme d'un tiers de confiance.

L'informatique de confiance fait usage du concept de *Trusted Computing Base* (TCB) et utilise le concept de *trusted chain* ainsi que le *trusted boot*. La TCB peut se résumer comme étant l'ensemble des parties qui composent un système et dont une modification met en danger la sécurité de l'ensemble du système. La *trusted chain* est une relation transitive de confiance entre les composants du système[12].

La TCB n'a pas besoin d'être la plus large possible ; Au contraire, il est intéressant de limiter la TCB au strict minimum, la *trusted chain* pouvant garantir l'intégrité du des autres composants.

Le *trusted boot* est quand à lui l'opération qui consiste à démarrer le système en partant de la TCB et en construisant une *trusted chain*. Celle-ci peut être obtenue de diverse manières, souvent en ayant recours à la cryptographie. Deux méthodes simples pour un élément de confiance consistent en vérifier un *hash* préalablement calculé ou vérifier une signature.

9. http://lxc.sourceforge.net/
10. http://linux-vserver.org/Welcome_to_Linux-VServer.org
11. physique ou non, comme une machine
12. en réalité, la structure obtenue est un arbre, cependant les cas pratiques simplifient cette structure

2.3.1 Trusted platform module

Le *Trusted Platform Module* (TPM) est une spécification [TPMa], écrite par le *Trusted Computing Group* (TCG), pour assurer des fonctions liées à la sécurité d'un système, plus spécifiquement son intégrité et s'intéresse en particuliers aux plate-formes PC. Le résultat est un composant discret présent qui est aujourd'hui disponible par exemple sur certains ordinateurs vendus au public.

Un TPM est en mesure de réaliser un certain nombre de primitives cryptographiques [TPMb], certaines étant imposées par la spécification. Parmis ces primitives, on pourra noter la possibilité d'effectuer des opérations de hachage, de chiffrement et de déchiffrement symétrique ou asymétrique, ainsi que la génération de nombres pseudo-aléatoires. Il est également possible d'effectuer les opérations nécessaires à la création et la vérification de signature, et tout particulièrement la génération et la gestion de clefs. Toutes les opérations disponbiles sont définies en détail dans [TPMc]. Enfin, un TPM dispose de mémoires internes, volatiles et non-volatiles. L'ensemble se présente sous la forme d'un composant discret conçu pour assurer la sécurité des données qu'il traite ; La réalisation sous forme *hardware* est ici une condition afin de résister à un ensemble large d'attaques *software*.

Un élément notable d'un TPM est la présence de registres spécifiques, les *Platform Configuration Registers* (PCRs), qui sont un composant important de l'opération de *trusted boot*. Les PCRs peuvent être lus à tout moment à l'aide de l'opération `TPM_PCRRead` mais ne peuvent pas être librement modifiés ; ceux-ci sont réinitialisés lors du démarrage et peuvent être *étendus* à l'aide de l'opération `TPM_Extend` [TPMc]. L'extension est une opération à sens unique qui met à jour le contenu d'un PCR.

$$Extend(PCR_N,\ value) = \mathcal{H}(PCR_N \mid\mid value)$$

La concaténation seule suivie du hachage est appelée *mesure*. Sous réserve que la fonction de hachage $\mathcal{H}$ soit sûre [13], on comprend que le contenu d'un PCR ne peut être obtenu que par une suite d'opérations d'extension données ; Le contenu du PCR peut alors être utilisé à des fins d'audit de la plate-forme. C'est sur cette propriété que repose le *trusted boot*.

Lors du démarrage, le BIOS réalise une mesure d'une partie du système ainsi que du *bootloader* ; Le *bootloader* réalise une mesure du noyau, et ainsi de suite. Ceci crée la *chaine de confiance* qui est en pratique une mesure du système. Intuitivement, on comprend que se pose la question de l'initialisation. Le TCG défini un *Core Root of Trust Measurement* (CRTM) [TCG], un noyau de composants du système dont l'intégrité est critique. Sur une plate-forme PC, ces composants sont par exemple la mémoire centrale, le processeur, le BIOS ainsi que certains bus.

Les PCRs sont par la suite utilisés de différentes façons pour assurer l'intégrité du système. Lire et comparer l'état d'un PCR à son état attendu n'est pas suffisant, puisque si le système a été modifié, il est vraisemblable que l'attaquant puisse aussi modifier l'état qui a été enregistré. Le TPM propose un concept de *scellés* avec le couple de commandes `TPM_Seal` et `TPM_Unseal` [TPMc]. La première permet au TPM de chiffrer une information à l'aide d'une clef et de l'état de un ou plusieurs PCRs ; La seconde réalise l'opération de déchiffrement, qui ne peut être effectuée par le TPM que si les PCRs sont dans le même état que lors de la phase de chiffrement. D'une manière similaire, un TPM est capable de prouver à distance l'intégrité du système en signant l'état de ses PCRs.

13. La spécification utilise pour l'instant `SHA1`

2.3.2 Mobile trusted module

Les *Mobile Trusted Modules* (MTMs) [MPWb] sont le pendant des TPMs pour les équipements mobiles et sont vu de loin très similaires. Cependant, du fait qu'ils cherchent à sécuriser des plate-formes différentes, ils présentent logiquement des différences en particulier dans leurs usages.

Tout d'abord, le concept de *secure boot* est ajouté en plus de celui de *trusted boot*. Celui-ci permet au MTM d'arrêter le démarrage de la plate-forme en cas d'échec dans la vérification de l'intégrité de celle-ci. Ce mécanisme a été ajouté pour répondre au cas des équipements mobiles dont l'usage est réglementé.

Ensuite, à l'inverse des TPMs, la spécification n'impose pas la présence d'un composant discret. Cette obligation avait conduit à des recherches en particuliers sur l'utilisation de la virtualisation et des sécurités matérielles associées [PSvD] dans le cadre des plate-formes PC. Ceci est cohérent avec l'univers des équipements mobiles où les constructeurs préféreraient ajout un MTM au sein d'un *System in Package* (SiP) ou d'un *System on Chip* (SoC), plutôt que de placer un composant discret coûteux en volume.

Enfin, la spécification propose la possibilité de faire cohabiter en parallèle plusieurs MTMs sur le même équipement (2.1). Un équipement mobile peut être en pratique la propriété de plusieurs acteurs, par exemple son propriétaire physique et un opérateur de téléphonie mobile. L'implémentation de référence [MPWa] prévoit la présence de deux MTMs correspondant respectivement aux acteurs précédemment cités, le *Mobile Local owner Trusted Module* (MLTM) et le *Mobile Remote owner Trusted Module* (MRTM).

Il faut noter qu'aucune MTM n'est disponible sur des mobiles Android à ce jour.

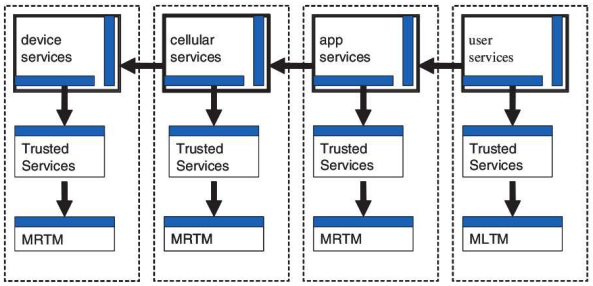

FIGURE 2.1 – *Mobile Trusted Module*

*
**

[SKK08] propose une première implémentation de MTM sous forme purement logicielle ; Celle-ci part du principe que toutes les services fournis par un MTM peuvent l'être par un TPM.

Faire cohabiter l'ensemble des MTMs d'une plate-forme au sein d'un unique TPM ou d'une émulation logicielle de MTM n'est en pratique pas acceptable, puisque cela expose les secrets des multiples acteurs à l'ensemble des acteurs ; Un acteur pourrait alors être en mesure par exemple de récupérer des clefs qui ne sont pas les siennes.

Pour parer à ce risque, [SKK08] utilise une architecture reposant sur un ensemble de compartiments séparés par une couche de virtualisation comme décrit dans [PSvD]. Cette solution permet de déployer un MTM par machine virtuelle, la sécurité pouvant être assurée par un seul TPM ou MTM présent sur la plate-forme.

L'implémentation se base sur Turaya [14], une plate-forme compatible avec l'*European Multilaterally Secure Computing Base* (EMSCB) [15] au dessus d'un micro-noyau de la famille L4 [16]. Le système d'exploitation original est virtualisé et s'exécute sans aucune modifications, à l'exception d'un pilote pour les MTMs virtuels permettant d'effectuer des appels vers l'hyperviseur. Chaque MTM est conçu autour d'une instance de L4Linux [17] du micro-noyau et peut ainsi agir de façon isolée des autres MTMs. Un service du micro-noyau, le *MTM proxy* assure la communication par IPC entre le système d'exploitation original et une instance de MTM virtuel, ou entre un MTM virtuel et le micro-noyau. Enfin, le micro-noyau dispose de son driver propre pour accéder au TPM ou au MTM physiquement présent sur la plate-forme.

2.3.3 ARM TrustZone

ARM a produit pour ses gammes de processeurs dédiés aux terminaux mobiles un ensemble d'extensions [AF04][AF05], la *TrustZone*, afin d'assurer des fonctions de sécurité au niveau *hardware*. Celles-ci se concentrent principalement sur la gestion de l'accès aux fonctionalités du processeur ainsi qu'au bus AMBA en introduisant une séparation de privilèges.

Les bases de la *TrustZone* reposent sur la séparation lors de l'exécution en deux mondes distincts, dénommés *secure world* et *non-secure world*. Ces deux mondes forment un concept orthogonal aux concept déjà existant *privileged/unprivileged* qui correspondent aux modes d'exécution utilisateur/noyau. Ces mécanismes de séparations de privilège se reposent sur deux éléments matériel. Dans un premier temps, le changement explicite de mode n'est possible que vers le mode de privilège moindre. Dans un second temps, l'accès à la configuration de la *TrustZone* ne se fait que dans le mode de privilège le plus élevé.

Dans le cadre de la *TrustZone*, certains des registres processeurs sont dupliqués afin d'être librement disponibles dans les deux mondes, chaque monde disposant de sa copie propre. Les autres registres, en particulier les registres de configuration sont accessibles en lecture et en écriture par le monde sécurisé ; les accès du monde non-sécurisé dépendent de la configuration du processeur. Afin de fournir une interface entre les deux mondes, un nouveau mode d'exécution, le *« monitor mode »* , est disponible pour le monde sécurisé. Dans ce mode, le monde sécurisé est en mesure de modifier la configuration des extensions *TrustZone* du processeur, ainsi que de basculer dans le mode privilégié d'un monde ou de l'autre ; le retour vers le mode moniteur peut se faire en utilisant l'instruction `SMC`. Enfin, les exceptions au niveau du processeur [18] peuvent être configurée pour être routées vers le mode privilégié de l'un des deux mondes ou vers le mode moniteur.

Enfin, un processeur utilisant la *TrustZone* dispose aussi d'un mécanisme de *trusted boot*. À l'inverse des concepts proposés par le TCG [MPWa] [MPWb], la *TrustZone* se base sur un mécanisme de signature à clefs asymétriques. Le processeur démarre à partir d'une ROM interne, implicitement de confiance et dispose de la partie publique d'une paire de clef inscrite définitivement à l'aide de fusibles ; Chaque élément successivement chargé en mémoire au cours du démarrage et signé avec la partie privée

14. `http://www.emscb.com/content/pages/turaya.htm`
15. `http://www.emscb.com`
16. `http://os.inf.tu-dresden.de/L4/`
17. `http://os.inf.tu-dresden.de/L4/LinuxOnL4/`
18. celles-ci sont désigné IRQ, FIQ, *external data abort* et *external prefetch abort* dans les processeurs ARM

de la clef. Sa signature vérifiée par un co-processeur avant exécution et le démarrage est interrompu si la signature n'est pas valide, ce qui permet d'établir une chaîne de confiance au démarrage. On s'assure ainsi que les équipements qui ont été compromis au sein d'un réseau mobile soient réduits au silence et ne puissent pas être utilisés pour attaquer le réseau ou utiliser le réseau pour effectuer des attaques.

[EB09] propose une implémentation logicielle de MRTM basé sur TI-M-Shield [AF], un ensemble de technologies reposant en partie sur ARM *TrustZone* et présentes dans certains SoC de TI, en particulier les ARM 9. La principale contrainte de ces composants est la taille très faible de la mémoire volatile présente dans le *Trusted Execution Environnement* (TrEE) ; Ceci est d'autant plus visible que cette mémoire est prévue pour recevoir une partie du système d'exploitation de la plate-forme. La solution envisagée s'inspire de [KS] et de [STP08].

En supposant que la plate-forme fournisse les mécanismes suivants

- une mémoire non-volatile pour le stockage du code dont l'intégrité pourra être vérifiée à l'aide d'une signature ;
- une mémoire volatile protégée à l'intérieur du SoC ;
- un secret spécifique à chaque équipement permettant de l'identifier de façon unique.

Le MRTM est décomposé en plusieurs parties

- le code est séparé en plusieurs collections de fonctions suffisamment petites pour tenir dans le TrEE ;
- l'état interne du MTM est séparé de la même manière.

Ainsi, le code émulant le MTM est chargé sur demande ; Son intégrité et son authenticité sont assurées par la signature du code fournie par la *TrustZone*. Implicitement, la signature assure que le code exécuté fait bien partie de la TCB. L'état interne est quand à lui stocké à l'extérieur de la TrEE sous forme de *blobs*[19] chiffrés à l'aide d'un secret présent dans la mémoire volatile à l'intérieur de la TrEE. Lors du redémarrage de la plate-forme, les anciens blobs ne sont pas réutilisables et le MTM retourne bien à son état initial.

2.4 Android

Android[20] est un environnement de développement complet développé par l'*Open Handset Alliance* (OHA) et destiné en particulier à s'adresser au marché de la téléphonie mobile. Il se compose entre autres d'un système d'exploitation basé sur le noyau Linux et d'une machine virtuelle Java nommée *Dalvik*.

19. *binary large object*, un aggrégat de données binaires
20. `http://developer.android.com`

Le noyau Linux fourni au système Android l'ensemble des primitives de base, tout particulièrement la gestion de la mémoire, la gestion des processus, un ensemble de pilotes ainsi que la gestion du réseau. Le noyau a par ailleurs reçu un certain nombre de *patches* afin d'améliorer le support des plate-formes mobiles. L'autre élément principal du système Android est Dalvik, la machine virtuelle Java qui permet l'exécution des application à l'intérieur de *sandboxes*.

Une application Android est sous la forme d'une archive au format `.apk` qui regroupe de façon assez similaire au format `.jar` ses constituants. L'exécutable est lui-même sous la forme d'un fichier au format `.dex` conçu avec un soucis de compacité. L'exécution est séparée en plusieurs classes, les *activities* qui correspondent à ce qui sera affiché à l'écran, et les *providers* qui fourniront des services aux autres applications.

Enfin, une application Android comprend aussi un fichier `AndroidManifest.xml` (2.4) qui la décrit. On y retrouve la liste des *activities* et *providers*, mais aussi les droits que l'application requiert [21] ainsi qu'une liste d'*intents*, des services que l'application est prête à satisfaire dans le cadre des communications inter-processus.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    ↪package="com.android.notepad">
  <application android:icon="@drawable/app_notes" android:label="
      ↪@string/app_name">
    <provider class=".NotePadProvider" android:authorities="com.google.
        ↪provider.NotePad" />
    <activity class=".NotesList" android:label="@string/
        ↪title_notes_list">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
```

Listing 2.4 – AndroidManifest.xml

Au dessus du noyau Linux, un certains nombre de bibliothèques natives, écrites en C ou C++, fournissent des services aux applications présentes dans le système Android. Celles-ci fournissent en particulier des interfaces natives Java pour être utilisables au sein de la machine virtuelle Java. À côté, les *core libraries* ainsi que le *application framework layer*, écrits tous les deux en Java, ajoutent respectivement un sous-ensembles des fonctionnalités de la Java 5 SE ainsi qu'un certains nombre d'extensions.

D'autres applications coexistent dans le système. Le processus `init` lancé en premier par le noyau Linux assure le démarrage du système. Le `zygote`, une application Java, agit comme application maître et prend en charge la création d'une instance de Dalvik lors du démarrage d'une application Java. Le `system_server`, une application Java démarrée par le `zygote`, met à disposition des autres machines virtuelles Java un ensemble de services. Enfin, d'autres applications natives sont présentes, comme `mountd` qui prend en charge la gestion du montage des périphériques blocs, `installd` qui assure l'instalation des applications à partir des fichiers `.apk`, ou `rild` qui assure la communication avec la partie radio de l'appareil mobile.

Lors de l'installation d'une application, le système fourni à l'application les permissions qu'elle demande dans la mesure ou l'utilisateur les acceptes [22]. Dans un

21. ou plutôt demande

22. les permissions demandées forment un tout et il n'est pas possible à l'utilisateur de restreindre une application

premier temps, un utilisateur Unix unique est crée. C'est sous cet utilisateur non-privilégié que sera exécutée l'application ; Il est aussi le propriétaire des fichiers relatifs à l'application. Cela permet la mise à jour de la politique DAC du système. Dans un second temps, l'application se voit offrir les permissions spécifiques au système Android. Parmis ces permissions, on peut trouver la possibilité d'installer une application (`install_packages`), d'utiliser SIP (`use_sip`), d'envoyer des SMS (`send_sms`), ou encore de « *bricker* » l'équipement [23] (`brick`). Les permissions Android nécessaires sont listées dans le manifeste. Lors de l'installation, le système demande à l'utilisateur son accord pour certaines permissions ; Ceci se fait en « tout ou rien » et l'utilisateur n'est pas en mesure d'affiner sa politique autrement qu'en refusant de poursuivre l'installation.

Les permissions sont classées en 4 catégories

- *normal* qui ne requiert pas d'autorisation explicite de l'utilisateur ;
- *dangerous* qui requiert une autorisation explicite ;
- *signature* qui demande une signature de l'application délivrée par la même entité que celle de l'application qui fourni le service demandé par la permission ;
- *signature-or-system* qui n'existe que par compatibilité ascendante.

Enfin, les applications Android sont signées. Cependant, à l'inverse d'autres plateformes où une signature provenant d'une entitée certifiée est indispensable, celle-ci ne sert ici qu'à authentifier l'auteur de l'application. Ceci permet par exemple de donner le même utilisateur Unix à deux applications crées par la même personne afin que ces application puissent partager explicitement des fichiers.

La sécurité se base en pratique sur un système de réputations ; Les auteurs des applications étant authentifié par leurs signatures, un utilisateur est en mesure de connaître les avis d'autres utilisateurs sur une application et de partager le sien sur des dépôts [24].

*
**

Sous sa forme actuelle, Dalvik, la machine virtuelle Java semble être un choix sain en matière de sécurité. Lors de l'installation d'une application, un nouvel utilisateur Unix est crée pour y être associé. Le démarrage d'une application se fait ensuite grâce au `zygote`. Celui-ci ouvre un *socket* en écoute local et attend pour traiter les requêtes envoyées par ce biais. Lorsqu'un message y est envoyé pour démarrer une application, il *fork*. L'instance Dalvik nouvellement crée prend alors en charge le chargement de l'application, puis effectue une descente de privilèges, en particulier en donnant au processus l'`UID` et le `GID` prévus lors de l'instalation. Ceci respecte le principe de séparation des privilèges ; Les applications sont confinées et les interférences entre les machines virtuelles sont ainsi réduites.

La communication inter-processus entre les machines virtuelles Java s'effectue par le biais du *binder*, un périphérique fourni par le noyau et présent dans le VFS sous la forme du `/dev/binder`. Lorsqu'une application Java souhaite dialoguer avec une autre, celle-ci génère un *intent*, un message qui contient en particulier une action suivie d'une *Uniform Resource Identifier* (URI). Lorsque le message est envoyé au *binder*, le noyau cherche à l'aide des *intent filters* déclarés à l'instalation par les manifestes une [25] application capable de recevoir le message et la démarre si besoin

23. le transformer en brique *(brick)* en détruisant le *firmware*
24. Google propose par exemple *Android Market*
25. ou plusieurs

est. Le *binder* permet aussi de gérer l'accès aux interfaces du système obtenues via les permissions Android lors de l'installation.

Dalvik fourni aussi un outil pour les appels de méthodes inter-processus. Plus particulièrement, il permet de définir une interface de communication qui sera utilisé entre deux processus suivant un modèle client-serveur. L'interface est décrite à l'aide du langage Android Interface Description Language (AIDL) (2.5) et permet d'instancier les deux implémentations jouant le rôle de client et de serveur.

```
1 package com.android.sample;

  interface intfSample
  {
5   int get ();
    void set (in int value);
    //
```

Listing 2.5 – AIDL

Par défaut, seuls les types primitifs peuvent être utilisés directement dans ces interfaces. Les objets doivent impérativement implémenter l'interface `Parcelable` qui ressemble à l'interface `Serializable`. Celle-ci permet de transtyper un objet vers ou depuis une *parcel*, une représentation de l'objet qui réside dans une zone mémoire partagée entre les deux processus. À l'inverse des techniques de sérialisation habituelle, cette interface impose de définir explicitement cette transformation. Ceci permet d'éviter les risques existants dans l'utilisation d'objets sérialisables, en particulier les attaques utilisant un objet sérialisé mal formé [26].

Le choix de Java comme langage est cohérent avec la politique de sécurité retenue. La présence d'un vérificateur de *bytecode* assurant le respect des contraintes de typage du modèle objet ainsi que la protection mémoire assurée par la machine virtuelle Java permettent de réduire la surface d'attaque sur Android par le biais de Dalvik.

*
**

[SFK+10] propose une analyse assez large des menaces face auxquelles un équipement mobile sous Android est confronté ainsi qu'une revue des mécanismes de sécurité et leur efficacité face à ces attaques. Ces attaques sont classifiées en cinq catégories

- celles exploitant les mécanismes de permissions ;
- celles exploitant une faille dans le noyau Linux ;
- celles s'attaquant aux informations stockées sur l'équipement mobile ;
- celles s'attaquant à la consommation d'énergie ;
- celles s'attaquant au réseau mobile.

Plusieurs solutions sont proposées pour apporter une meilleure sécurité du système. Une première méthode serait d'effectuer une certification des applications qui sont proposées. Relire et tester chaque application aurait un coût élevé et le mécanisme de réputation semble plus adapté. Cependant, il est tout à fait pensable de le faire pour une sous-partie des applications. Une seconde méthode serait de permettre à l'utilisateur lors de l'installation de choisir de manière sélective quelles sont les permissions qu'il souhaite accorder à une application. Pour l'instant, ceci n'est pas possible dans Android, et l'utilisateur doit prendre une décision entre refuser des permissions qu'il juge abusives ou bien estimer qu'il peut prendre le risque. Enfin, un

26. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5353

troisième méthode consisterait en déployer SELinux pour contrôler en partie l'activité des applications.

Dalvik n'est pas une pièce centrale dans la sécurité d'Android. Cependant, certains choix montrent un soucis de propreté dans la conception. La communication entre deux applications différentes est correctement contrôlée ; En particulier, on notera la présence de l'interface `Parcelable` qui remplace de manière avantageuse les objets sérialisables.

On peut regretter néanmoins l'impossibilité de choisir avec plus de précision les droits que l'on accorde à une application, ainsi que l'impossibilité de mettre en place un contrôle obligatoire des signatures des applications à l'installation.

2.4.1 SELinux

[SFE09] propose un exemple d'utilisation de SELinux dans Android. Après avoir activé LSM dans le noyau, la configuration de SELinux a été recrée à partir de rien car

- la configuration par défaut est trop large ;
- Android ne respecte pas la hiérarchie standard de VFS de Linux.

La politique résultante s'exécute avec un *overhead* très faible. Cependant, deux problèmes ont été rencontrés lors de la configuration de SELinux. Tout d'abord `yaffs2`, le système de fichier d'Android, ne supporte pas les attributs étendu `xattrs` dont a besoin SELinux pour placer des identifiants sur les objets. Cette limitation a été contournée en utilisant le mécanisme de contextes qui permet de retrouver ces identifiants dynamiquement. L'autre difficultée est qu'une application Android est du point de vue de Linux un fichier qui est lu par une application native, la machine virtuelle Java. Aussi il n'est pas possible de donner en l'état actuel un domaine précis à Dalvik en fonction de l'application qu'elle est en train d'exécuter. Ceci pourrait être résolu en modifiant le zygote de sorte qu'il effectue explicitement la transition de domaine nécessaire avec la descente de privilèges.

2.4.2 Mesure d'intégrité

Récement [NKZS10], des travaux ont démarrés afin d'adapter le système Android aux architectures de confiance. Ces travaux s'orientent spécifiquement autour de la capacité d'installer facilement des applications avec tous les risques que cela apporte. En particulier, ils s'intéressent à la possibilité de prouver l'intégrité du système tout en permettant l'ajout de ces applications.

L'architecture du système se base sur les concepts du *Trusted Computing Group - Mobile Workgroup* (TCG-MW) [MPWb][MPWa], à savoir la mesure d'une empreinte cryptographique du système. Cependant, les MTM n'étant pas encore à l'heure actuelle disponibles sur les systèmes Android, la solution retenue consiste pour l'instant à utiliser un émulateur comme celui décrit dans [EB09].

La *trusted chain* permettant de prouver l'intégrité du système a ensuite été modifiée afin de s'adapter à la présence de la machine virtuelle Java. En plus du *Stored Measurement Log* (SML), qui contiendra ici par exemple une mesure de Dalvik ou de bibliothèques, un nouveau journal, appelé *Android Measurement Log* (AML), est chargé d'une façon similaire au SML de conserver les mesures des applications chargées dans les instances de la machine virtuelle Java. Ces mesures se font de deux manières. La première, au niveau de l'application, consiste à effectuer une mesure de fichier `.apk` correspondant lors du démarrage de celle-ci. La seconde, plus fine,

consiste quand à elle à installer un *hook* sur le chargeur de classes, assurant ainsi que la totalité des classes chargées dans la machine virtuelle Java aient bien été mesurées.

Les premiers résultat montrent un *overhead* assez important sur le temps de chargement total d'une application pouvant aller de l'ordre de la seconde pour la première méthode. Ceci est très supérieur au temps de réaction de l'être humain et affecte l'ergonomie du système. Un autre aspect considéré est la taille en mémoire de l'AML ; Dans le premier cas, celui-ci reste de l'ordre du kB, cependant l'approche fine génère un journal de l'ordre de la centaine de kB. L'aspect consommation d'énergie est esquivé, les calculs se faisant exclusivement sur le GPP en des temps négligeables vis-à-vis de la batterie.

Enfin, un dernier point abordé est la faiblesse du système à ce jour. L'absence d'une MTM physique remplacée par un émulateur est en effet contraire au concept des plate-formes de confiance et rend la sécurité de l'ensemble faible. Ceci pourrait être corrigé par l'arrivée de systèmes Android équipés de MTM.

*
**

D'autres travaux concernant Android n'ont pas été présentés ici.

[Win08] donne un autre exemple d'architecture proposasnt une MTM obtenue à partir de *TrustZone*.

[MSS+08] combine l'intégrité du système obtenue à l'aide de l'informatique de confiance avec des modèles DAC d'intégrité utilisant SELinux.

[Cha09] propose de modéliser les communications entre les applications à l'aide de sémantique opérationnelle. L'utilisation conjointe avec un système de types permet de vérifier que les interactions entre les machines virtuelles se font en respectant des contraintes données.

[Hil09] propose une implémentation d'Android, puis de Dalvik seule [27] sur le micro-noyau OKL4 [28].

27. http://www.ertos.nicta.com.au/software/androidokl4/
28. http://www.ok-labs.com/

Chapitre 3

Objectifs

Les travaux qui auront été effectués au cours de stage ont pour objectif d'étudier la faisabilité de l'intégration d'une politique de sécurité multi-niveau telle que celle présentée par Bell et Lapadula au sein d'Android. Plus particulièrement, nous nous intéresserons à des outils légers déjà présent dans Linux pour assurer l'isolation nécessaire à une telle politique et contrôler les flux d'infomartion. Nous veillerons tout particulièrement à satisfaire deux contraintes qui nous ont semblé pertinentes.

La première de ces contraintes est l'impact en terme de volume de modifications apportées. Android est un système jeune qui, lui même ainsi que les projets qui le composent, évolue très rapidement et des modifications importantes sont naturellement difficiles à maintenir dans le temps sur de tels projets.

La seconde contrainte vient de l'aspect « système embarqué ». Les plate-formes sur lesquelles évolue Android étant souvent restreintes en terme de temps processeur disponible, de mémoire et d'énergie, et devant parfois répondre à des contraintes de temps-réel, il est donc important de s'assurer que l'impact sur le système reste minial.

De cette première contrainte, nous nous sommes attardé plus particulièrement sur les nombreux outils déjà présents dans Linux, ce qui nous assure la disponibilité à moyen et long terme. La seconde contrainte donnait un critère objectif de choix à priori et d'évaluation à posteriori.

Nous chercherons donc à présenter un modèle léger qui sera par la suite validé par l'expérimentation sur une plate-forme réelle.

Chapitre 4

Isolation sous Android

Ce chapitre se consacre à l'étude qui aura été réalisée au cours de ce stage.

Dans une première partie, nous détaillerons les différents scenarii proposés et auxquels nous tenterons de répondre. Nous présenterons dans la partie qui suit les solutions techniques qui auront été retenues. Enfin, dans la dernière partie, nous expliciterons le modèle que nous proposons.

4.1 Confidentialité sous Android

Afin de répondre à une nécessité forte de confidentialité, nous souhaitons ajouter à Android le modèle de sécurité multi-niveau décrit par Bell et Lapadula [LB96]. Du fait que nous nous plaçons dans un système embarqué mono-utilisateur [1], nous ne cherchons pas à utiliser le modèle exhaustif ; nous réduisons donc le poset des habilitations à deux éléments qui seront nommés par la suite « haut » et « bas ».

Nous décrivons maintenant trois scenarii qui décrivent une action légitime selon les deux propriétés énoncées pour le modèle.

Remontée d'une donnée

« *L'utilisateur veut transformer le niveau de classification d'une information de bas vers haut (write-up).* »

Déclassification d'une donnée

« *L'utilisateur veut transformer le niveau de classification d'une information de haut vers bas (write-down).* »

Cette action est en théorie illicite du point de vue du modèle ; Cependant si l'information est chiffrée et que les sujets de niveau de classification bas ne sont pas en mesure de la déchiffrer, alors le modèle est respecté.

4.2 Contexte technique

Linux étant relativement fréquemment utilisé dans le domaine de la recherche pour son aspect logiciel libre ainsi que sa communauté importante, celui-ci dispose

1. entendre un seul utilisateur humain, mais toujours plusieurs utilisateurs Unix

d'un panel large d'implémentations de diverses mécanismes de sécurité. En particulier, la présence des LSM permet l'implémentation ou une utilisation facile d'une politique DAC. Cependant, l'ajout de ces mécanismes qui ne sont pas présents sous forme de *Loadable Kernel Module* (LKM) a tendance à augmenter l'espace disque utilisé par le noyau, parfois dans des proportions très importantes.

Malheureusement, la place offerte au noyau Linux d'Android est parfois très réduite [2] et en pratique beaucoup de noyaux ne rentrerons tout simplement pas après l'ajout d'une option dans sa configuration avant compilation. Nous devions donc prendre en compte cette contrainte dans le choix des mécanismes que nous allions utiliser.

4.2.1 Conteneurs Linux

Linux dispose à partir des versions 2.6.15 et ultérieures d'un mécanisme de virtualisation légère appelé « conteneurs Linux » avec les outils en espace utilisateurs *lxc* [3]. Il s'agit d'un projet à long terme qui est à ce jour toujours en cours d'implémentation et d'inclusion dans la branche principale de Linux [4].

Les conteneurs reposent sur les sous-systèmes du noyau et les groupes de contrôle qui sont deux mécanismes de labélisation pour les tâches et les ressources du système respectivement.

Un groupe de contrôle, ou *cgroup*, est un ensembles de tâches du système d'exploitation. Chacun de ces ensembles de tâches peut être lié à un ou plusieurs sous-systèmes assurant la configuration de l'accès aux ressources fournies par le noyau aux tâches de cet ensemble. Les groupes de contrôle sont par ailleurs organisés de manière hiérarchique sous la forme d'un arbre. Enfin, comme toute tâche fait partie d'un unique groupe de contrôle, celles-ci sont liées par défaut à la racine de cet arbre [5] ou au groupe de contrôle du processus parent si celui-ci en dispose d'un.

```
/cgroup/cpu/
           /high/
                /cpu.shares
                /tasks
           /low/
               /cpu.shares
               /tasks
           /cpu.shares
           /tasks
```

Listing 4.1 – Exemple de listing du pseudo-système de fichiers `cgroup` (`cpu`)

Comme indiqué précédemment, un sous-système assure la configuration de l'accès à un type de ressource pour les tâches présentes dans un groupe de contrôle. Un groupe de contrôle peut se voir associer un ou plusieurs sous-systèmes différents ; Le comportement par défaut si le sous-système n'est pas présent étant de fournir un accès sans restriction à la ressource associée.

Par exemple, nous décidons d'offrir à certaines applications plus de temps processeur qu'à d'autres. Nous utilisons le sous-système `cpu` (4.1) qui permet de répartir le temps processeur et créons deux groupes, *high* et *low*. Les processus font partie d'un *cgroup* quand ils sont listés dans le pseudo-fichier `tasks`. Chaque *cgroup* se voit

2. sur les modèles de téléphones qui auront été utilisés lors du stage, la partition `boot` de la NAND qui contient le noyau et le *ramdisk* initial ne mesure que 2.4MB
3. http://lxc.sourceforge.net/
4. la version 2.6.39 sortie en Mai de cette année intègre les *user namespaces*
5. ce sera forcément le cas pour `/sbin/init`

nom	type	fonction
`checkpoint`	*special*	sauvegarde/reprise d'états
`cpu`	ressources	limitation du temps processeur
`cpuset`	isolation	affinité processeur
`cpuacct`	ressources	comptabilité du temps processeur
`device`	isolation	restriction d'accès aux périphériques
`diskio`	ressources	limitation de la bande passante entrées/sorties
`freezer`	*special*	gel des processus
`memory`	ressources	limitation de la mémoire disponible
`namespace`	isolation	labélisation des ressources et processus
`net`	ressources	limitation de la bande passante réseau

TABLE 4.1 – Liste des sous-systèmes présents dans le noyau Linux

alors accorder une fraction de temps processeur indiquée dans les pseudo-fichiers `cpu.shares` respectivement, qui est elle-même une fraction du temps processeur du *cgroup* parent.

Les sous-systèmes (4.1) se décomposent en deux catégories distinctes.

- Les contrôleurs de ressources fournissent des limites par groupe aux ressources, par exemple le sous-système `cpu` permet de donner une limite haute au temps processeur fourni à un ensemble de tâches [6] ;
- Les contrôleurs d'isolation permettent une séparation de l'accès aux ressources, par exemple le sous-système `cpuset` permet de définir sur quels processeurs pourront être ordonnancé un ensemble de tâches.

Un conteneur Linux est une configuration donné d'un ensemble de sous-systèmes permettant par la suite d'instancier un espace utilisateur complet isolé du reste du système. Cet espace utilisateur peut être utilisé de deux façons différentes. On peut dans un premier temps choisir d'y placer explicitement une ou plusieurs applications. Cependant, l'usage habituel consiste à y démarrer un système GNU/Linux complet ce qui permet de faire cohabiter de façon isolée plusieurs OS GNU/Linux partageant le même noyau.

4.2.2 TOMOYO Linux

TOMOYO Linux [7] est une implémentation de politique de sécurité MAC se reposant sur les LSM introduits par SELinux. Il se distingue des autres implémentations en offrant des mécanismes simples, le rendant particulièrement propice pour de l'apprentissage automatisé de politique et un usage sur un système embarqué.

Dans un système Unix standard, la méthode usuelle pour démarrer un programme consiste à utiliser un `fork+execve`. Ainsi, tout processus est explicitement démarré par le dédoublement d'un autre processus, le parent, puis le remplacement du programme exécuté par le double ainsi crée. On obtient donc une transition explicite d'un exécutable, celui du parent, vers un autre, celui de l'enfant. TOMOYO Linux profite de ce comportement pour identifier des domaines dans lesquels vont s'exécuter chaque processus.

6. Android fait déjà usage des sous-systèmes `cpuacct` et `cpuctl` pour garantir un temps processeur minimum à l'application en premier plan

7. http://tomoyo.sourceforge.jp/index.html.en

Pour TOMOYO Linux, chaque processus fait partie d'un domaine. Celui-ci représente un historique ou un trace des domaines précédement atteints. Ainsi, lorsqu'un processus est démarré, TOMOYO Linux va faire transiter ce processus vers son domaine associé en apposant au domaine original le chemin absolu de l'application correspondante.

Par exemple, le domaine correspondant au noyau est par défaut `<kernel>`. Lorsque le programme d'initialisation est démarré, la tâche raçine du noyau réalise un `fork+execve` pour démarrer `/sbin/init`. TOMOYO Linux va alors associer au processus nouvellement crée le domain `<kernel> /sbin/init` résultant de la concaténation du domaine du noyau avec le chemin absolu du programme `init`.

À l'inverse de SELinux, TOMOYO Linux n'utilise pas de méta-données pour identifier les objets et ne les définit pas explicitement non plus. Les objets sont dans la pratique identifié par leurs chemins absolu, ce chemin pouvant être rendu générique à l'aide d'un modèle [8].

TOMOYO Linux associe ensuite à chaque domaine une *Access Control List* (ACL) regroupant l'ensemble des actions permises. Ces actions sont sous la forme d'une permission suivie d'un chemin absolu ou d'un modèle de chemin absolu vers une ressource du système.

Enfin, deux modifications dans le comportement de la transition de domaines permettent de factoriser les ACL. Il est tout d'abord possible de conserver le domaine du processus parent, le processus enfant s'exécutant alors avec exactement la même ACL que son parent. Ensuite, il est possible d'indiquer qu'un programme lors de son exécution va générer un nouveau domaine en oubliant l'historique des domaines précédemment traversés [9] ; De cette manière, il est possible d'attribuer une ACL à un processus indépendamment de la manière dont il a été exécuté [10].

TOMOYO Linux opère sous quatre profiles [11] :

- *disabled* : application de la politique DAC uniquement ;
- *learning* : toute action illicite sera utilisée pour l'apprentissage ;
- *permissive* : toute action illicite sera journalisée ;
- *enforce* : toute action illicite sera bloquée.

L'avantage de TOMOYO Linux réside dans le second profil, *learning*, sous lequel TOMOYO Linux ajoute lors de violations de la politique de sécurité les ACL nécessaires en s'aidant d'une liste d'exceptions crée par l'utilisateur.

4.3 Modèle proposé

Le but principal du stage est l'évaluation des mécanismes existant afin de fournir à un terminal sous Android une approche *Multi-Level Security* (MLS) légère et suffisante pour un usage dans un terminal mobile. Pour cela, nous proposons (4.5) d'établir deux conteneurs séparés afin d'assurer l'isolation des niveaux haut et bas.

8. un expression rationnelle, comme `/dev/ttyS\$` qui va correspondre à l'ensemble des ports série
9. c'est à dire que le domaine sera obtenu en concaténant directement le domaine du noyau avec le chemin absolu du programme
10. un service aura la même ACL qu'il soit démarré par `upstart` ou redémarré explicitement par l'utilisateur
11. les profiles sont en réalité appliqués par domaine et sont hérités lorsque le domaine a été nouvellement crée

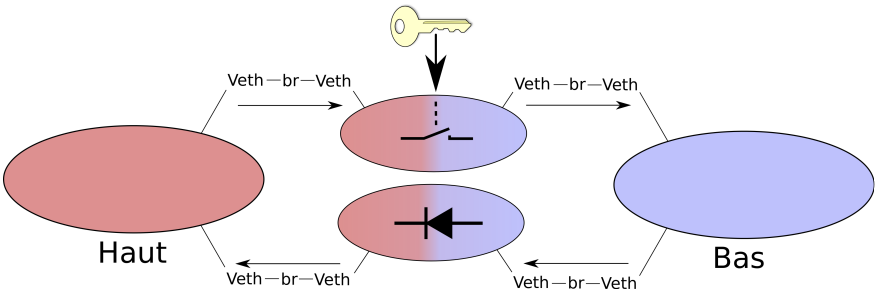

FIGURE 4.1 – Schéma de principe du modèle recherché

La communication entre les niveaux est assurée par plusieurs réseaux virtuels IP pour lesquels le sous-système `net` assure que l'information ne puisse pas être lue ou modifiée [12]. Enfin, le respect des propriétés énoncées par le modèle de BELL et LAPADULA est assuré par des conteneurs intermédiaires, l'un assurant la fonction de diode, l'autre la fonction de dé-classification de l'information. Au sein de ces conteneurs, nous ferons usage de TOMOYO Linux pour assurer la classification des données.

4.3.1 Organisation des conteneurs

Les conteneurs Linux ne sont en soit qu'un moyen et il est donc nécessaire de choisir la manière dont ceux-ci pourront être mis à contribution sous Android. Pour cela, plusieurs choix s'offrent à nous.

Une première approche consiste à organiser les conteneurs en suivant une hiérarchie (4.2). Cette hiérarchie s'établit explicitement par l'ordre dans lequel les conteneurs sont démarrés, et implicitement par la relation d'ordre qui existe entre les conteneurs au niveau des privilèges accordés. Dans ce modèle, l'hôte dispose de la totalité des droits, et les invités ne disposent que de droits restreints.

Implicitement, ce modèle s'apparente au modèle de BIBA [BM77] ou de BELL et LAPADULA où chaque niveau de conteneur correspondrait à un niveau de confidentialité/intégrité. En effet, un sujet instancié dans un conteneur ne peut lire une information que dans son conteneur ou un conteneur fils mais pas dans son parent. La comparaison s'arrête là puisque par exemple l'écriture suit les même règles, ce qui sous entend que l'écriture vers un niveau plus élevé se fait à la discrétion d'un sujet du conteneur de ce niveau et que l'écriture vers un niveau plus bas est autorisée.

12. même si le réseau présente un routage, les paquets sont en réalité « déplacés » par le noyau directement sur l'interface de destination

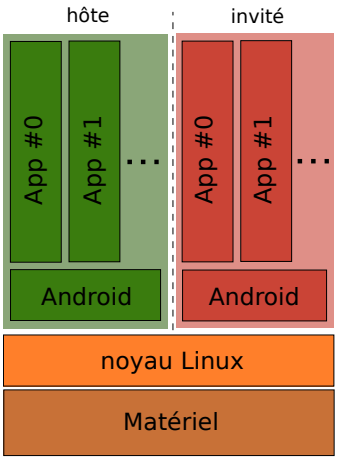

FIGURE 4.2 – Approche par encapsulation « *matriochkas* »

Une seconde approche (4.3) consiste à faire cohabiter des conteneurs séparés. Dans ce modèle beaucoup plus simple, l'hôte s'assure du démarrage et de la configuration des conteneurs avec les privilèges nécessaires pour leur exécution. L'hôte s'assure au besoin de la communication entre les conteneurs.

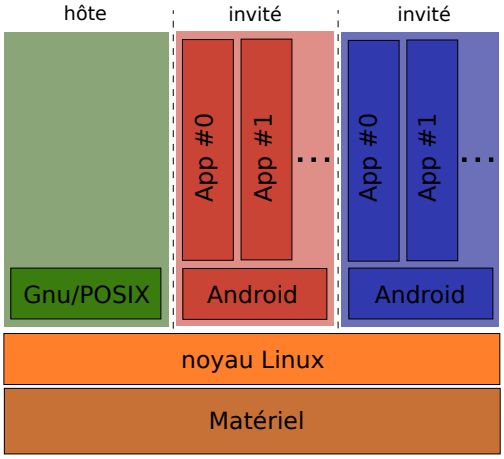

FIGURE 4.3 – Approche par séparation complète

Une dernière approche (4.4) consiste à ne pas utiliser les conteneurs pour exécuter un environnement complet, mais seulement des instances de la machine virtuelle Dalvik.

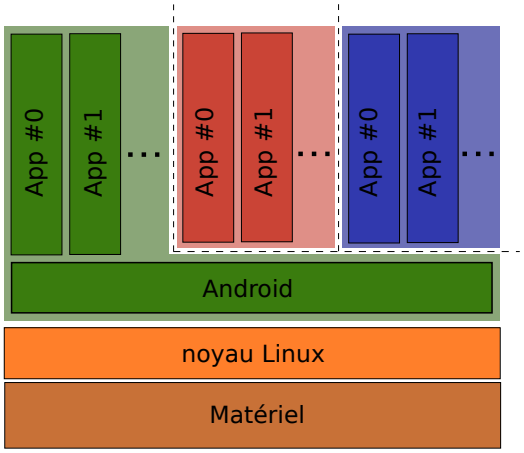

FIGURE 4.4 – Approche par isolation des machines Dalvik

Enfin, toute combinaison de ces trois approches est possible. Il n'y a en effet pas de limite au nombre et à l'organisation des conteneurs [13] et l'on peut imaginer plusieurs façons d'envisager le découpage du système.

4.3.2 Modèle retenu

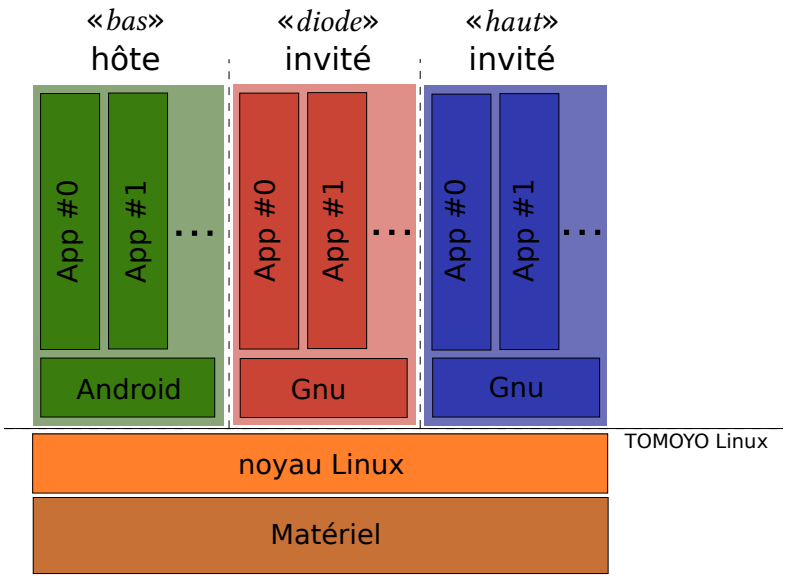

FIGURE 4.5 – Organisation des conteneurs

Après plusieurs essais, il nous est apparu qu'il était difficile de placer Android dans un conteneur sans effectuer de modifications lourdes. En effet, beaucoup de

13. cette limite existe, mais n'est en pratique pas atteignable

composants sont inter-dépendants, en particulier entre le noyau et l'espace utilisateur. Nous avons donc dans un premier temps choisi de conserver Android dans le conteneur hôte.

Le modèle retenu (4.5) instancie un système Android dans le conteneur hôte. Celui-ci devra naturellement manipuler des objets qui sont classifiées bas ; Nous l'appellerons par la suite le conteneur bas. Nous utilisons alors TOMOYO Linux afin de cloisonner le conteneur hôte lors de l'exécution de sorte que les actions qu'il est en mesure d'entreprendre ne puisse en aucune manière avoir d'impact sur les conteneurs invité.

À côté de cela, plusieurs conteneurs invités sont ensuites crées pour contenir des systèmes GNU/Debian [14]. L'un de ces conteneurs aura pour tâche de traiter les objets qui sont classifiés hauts et sera donc nommé par la suite le conteneur haut. Un autre conteneur fera office de diode ; Un dernier conteneur servira à la dé-classification.

14. nous avons eu recours directement aux binaires pour l'architecture ARM little-endian « *armel* » qui sont compatibles avec le noyau Android

Chapitre 5

Implémentation

Dans ce chapitre, nous détaillerons la méthode appliquée pour créer une plate-forme de démonstration à partir du modèle présenté auparavant ainsi que réaliser son évaluation. Dans une première partie, nous verrons comment sont instanciés les conteneurs invités. Puis, dans une second partie, nous montrerons comment a été utilisé TOMOYO Linux pour isoler le conteneur hôte des conteneurs invités ainsi que la manière dont nous avons réalisés la diode ou de façon plus générale comment nous contrôlons le flot d'information entre les conteneurs haut et bas. Enfin, dans une dernière partie, nous évaluerons l'impact de ces nouveaux mécanismes sur la plate-forme de démonstration.

5.1 Création des conteneurs

Instancier un conteneur se résume dans la pratique à installer un système d'exploitation complet, moins son noyau. Pour cela, nous avons eu recours à l'outil `debootstrap` qui rempli parfaitement cette fonction, suivi d'une phase de configuration manuelle.

La configuration des conteneurs se fait grâce à l'outil `lxc`. Nous veillons à respecter le principe du moindre privilège, plus particulièrement en ayant recours aux directives `lxc.cgroup.devices.allow` et `lxc.cap.drop` qui vont permettre respectivement d'autoriser explicitement l'accès aux périphériques depuis le conteneur et l'abandon des permissions qui ne sont pas strictement nécessaires [1], en particulier l'administration et le contournement des LSM (5.1).

```
## Network
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br0
lxc.network.name = eth0
lxc.network.ipv4 = 192.168.0.1

## Devices
lxc.cgroup.devices.deny = a

lxc.cgroup.devices.allow = c 1:3 rwm     # /dev/null
lxc.cgroup.devices.allow = c 1:5 rwm     # /dev/zero
lxc.cgroup.devices.allow = c 1:7 rwm     # /dev/full
```

1. Linux implémente les « POSIX *capabilities* »

```
lxc.cgroup.devices.allow = c 1:8 rwm     # /dev/random
lxc.cgroup.devices.allow = c 1:9 rwm     # /dev/urandom
lxc.cgroup.devices.allow = c 4:1 rwm     # /dev/tty1
lxc.cgroup.devices.allow = c 5:0 rwm     # /dev/tty
lxc.cgroup.devices.allow = c 5:1 rwm     # /dev/console
lxc.cgroup.devices.allow = c 5:2 rwm     # /dev/ptmx
lxc.cgroup.devices.allow = c 136:* rwm   # dev/pts/*

## Capabilities
lxc.cap.drop = audit_control
lxc.cap.drop = audit_write
lxc.cap.drop = kill
lxc.cap.drop = lease
lxc.cap.drop = linux_immutable
lxc.cap.drop = mac_admin
lxc.cap.drop = mac_override
lxc.cap.drop = sys_boot
lxc.cap.drop = sys_chroot
lxc.cap.drop = sys_pacct
lxc.cap.drop = sys_ptrace
```

Listing 5.1 – Extrait de configuration d'un conteneur

5.2 TOMOYO Linux

5.2.1 Introduction

À titre d'exemple introductif, nous donnons ici une politique de sécurité dans le cas simple d'un service présent sur un des conteneurs.

Le scénario est le suivant. Un serveur SSH, `dropbear`, ainsi qu'un serveur HTTP, `boa`, sont tous deux disponibles et en écoute sur l'interface réseau d'un conteneur et permettent donc une interaction avec ce conteneur. L'utilisateur unique, `root`, doit être en mesure de s'authentifier via SSH pour pouvoir accéder au conteneur sur un terminal. De là, il doit être en mesure d'influer sur le serveur HTTP en arrêtant ou en démarrant le service associé. Tout autre action, comme par exemple démarrer une autre application que le terminal via SSH, n'est pas autorisée.

Nous définissons plusieurs règles d'initialisation de domaines pour les domaines suivant :

- `/etc/init.d/boa` ;
- `/etc/init.d/dropbear` ;
- `/usr/sbin/boa` ;
- `/usr/sbin/dropbear`.

Ces domaines sont ensuite placés sous le profil d'apprentissage avant de procéder à l'exécution du scénario précédemment décrit. L'utilisateur `root` se connecte sur le conteneur via SSH puis redémarre le serveur HTTP.

```
root@host:~ # ssh root@tier1
root@tier1 s password:
root@lxc-tier1:~# service boa restart
Restarting HTTP server: boa... done.
```

L'arbre de transitions de domaines obtenu (5.2) donne une image de l'enchaînement des applications qui ont été exécutées pour accomplir cette tâche. On observe par exemple que l'exécution de `/etc/init.d/dropbear` (correspondant au domaine `<kernel> /etc/init.d/dropbear`) va provoquer une série de transitions

jusqu'au domaine `<kernel> /etc/init.d/dropbear /sbin/start-stop-daemon /usr/sbin/dropbear`. Cette dernière transition correspond à une initialisation de domaine vers le domaine `<kernel> /usr/sbin/dropbear`. Ainsi, que le serveur HTTP ait été démarré par le processus d'initialisation ou redémarré par l'utilisateur `root`, la politique qui lui est appliquée est la même.

```
  0:  0     <kernel>
  1:  1  *      /etc/init.d/boa
  2:  1             /sbin/start-stop-daemon
                        /usr/sbin/boa ( -> 287 )

103:  1  *      /etc/init.d/dropbear
104:  1             /bin/sleep
105:  1             /sbin/start-stop-daemon
                        /usr/sbin/dropbear ( -> 288 )

287:  1  *      /usr/sbin/boa
288:  1  *      /usr/sbin/dropbear
289:  1             /bin/bash
290:  3                 /usr/bin/id
291:  3                 /usr/bin/mesg
292:  1                 /usr/sbin/service
293:  1                     /usr/bin/basename
294:  1                     /usr/bin/env
                                /etc/init.d/boa ( -> 1 )
```

Listing 5.2 – Extrait de l'arbre de transitions de domaines obtenu

Un extrait des permissions nécessaires pour `dropbear` (5.3) nous montre que celui-ci a, entre autres, besoin, et y aura droit, d'accéder à ses pseudo-terminaux, au générateur de nombres pseudo-aléatoires `/dev/urandom` ainsi qu'aux clefs de chiffrement permettant au serveur de s'authentifier.

```
4: allow_chgrp       /dev/pts/\$
5: allow_chmod       /dev/pts/\$
6: allow_chown       /dev/pts/\$
7: allow_ioctl       /dev/pts/\$
8: allow_read/write  /dev/pts/\$
9: allow_read/write  /dev/tty
10: allow_read       /dev/urandom
11: allow_read       /etc/dropbear/dropbear_dss_host_key
12: allow_read       /etc/dropbear/dropbear_rsa_host_key
```

Listing 5.3 – Extrait de politique appliqué au serveur SSH `dropbear`

Celles-ci demandent par la suite un post-traitement à la main pour raffiner les permissions, le mode d'apprentissage de TOMOYO n'étant qu'une aide. En particulier, nous veillerons à :

- Factoriser les permissions, comme les accès aux fichiers qui ne sont pas préexistants mais respectent des conventions de nommage ;
  Les pseudo-terminaux sont tous nommés sur le VFS sous la forme `/dev/pts/<id>` et l'on peut donc simplifier les règles en utilisant le modèle `/dev/pts/\$`.
- Supprimer les permissions qui ne sont pas strictement nécessaires ou ne sont pas acceptables.
  L'accès en lecture/écriture sur le journal du système n'est pas nécessaire, et en pratique ne doit pas être autorisé car elle empêche de pouvoir auditer le système. Cependant, TOMOYO apprend la permission `allow_read/write /var/log/lastlog` lorsque `dropbear` met à jour la dernière date de connexion de l'utilisateur `root` puisque le fichier est ouvert en mode `rw`. Cependant, cette

permission n'est pas nécessaire et il est possible de la restreindre à `allow_read /var/log/lastlog`.

Pour finir, nous testons cette nouvelle politique afin de vérifier trois points :

- l'apprentissage doit avoir été exhaustif pour éviter que TOMOYO interdise une action légitime ;
- le post-traitement effectué ne doit pas interdire une action légitime ;
- la politique doit être efficace.

Bien que les deux premiers points semblent triviaux à vérifier, le troisième ne l'est pas. Sans chercher à définir de manière précise ce que peut être l'efficacité d'une politique [2], l'approche DAC par liste blanche [3] obligerait à tester tous les cas possibles, ou tout du moins à trouver une approche pour tester des ensembles de cas.

Nous nous restreignons donc à tester une action légitime, c'est à dire celle qui correspond au scénario, puis à tenter une action qui n'est pas légitime.

```
root@host:~ # ssh root@tier1
root@tier1 s password:
root@lxc-tier1:~# service boa restart
Restarting HTTP server: boa... done.
root@lxc-tier1:~# lastlog
-bash: /usr/bin/lastlog: Operation not permitted
```

L'utilisateur `root` a été capable de se connecter au conteneur pour redémarrer le serveur HTTP (5.2.1) ; Cependant, lorsqu'il a souhaité accéder aux journaux des dernières connexions, cela lui a été explicitement interdit. Un évènements correspondant est par ailleurs ajouté aux journaux du système (5.2.1).

```
[663109.469024] TOMOYO-WARNING: Access  execute(do_execve) /usr/bin/
    ↪lastlog  denied for /bin/bash
```

5.2.2 Isolation d'Android

Maintenant que nous avons présenté un cas simple pour débuter sur TOMOYO, nous allons ensuite nous attacher à procéder à l'apprentissage de la politique de sécurité pour Android. La couche applicative Android se situant dans le conteneur hôte, il est donc possible à n'importe quelle application d'avoir accès à des informations d'un niveau de sensibilité haut [4], ce qui n'est évidemment pas acceptable. Nous proposons d'utiliser TOMOYO Linux afin de cloisonner l'exécutif Android pour en interdire implicitement l'accès aux données hautes.

Pour cela, nous activons TOMOYO Linux au démarrage du système en passant au noyau le paramètre `--security=tomoyo` [5].

Nous plaçons ensuite le domaine `<kernel>` en mode apprentissage puis nous redémarrons le téléphone. La configuration processus de démarrage `init` a du être modifié en deux points. Tout d'abord, une partie du contenu du VFS a été déplacé de la mémoire interne [6] vers la mémoire externe du téléphone [7] afin de s'affranchir de la contrainte de taille et aussi de permettre un accès en lecture et en écriture, ce qui

2. on pourra voir cela comme la capacité à autoriser une action légitime ou à l'inverse interdire une action illégitime
3. en explicitant les autorisations
4. Android étant l'hôte, il s'agit en pratique du conteneur qui a accès à l'intégralité du VFS
5. le noyau est compilé avec le paramètre `DEFAULT_SECURITY_POLICY=DAC` pour conserver la politique de sécurité DAC Unix au démarrage pour faciliter la récupération du téléphone en cas d'erreur de manipulation
6. les partitions NAND `boot` et `system`
7. une carte SDHC

permet d'obtenir la persistance de politique entre chaque redémarrage. Enfin, avant de commencer le démarrage des services, on procède au chargement de la politique qui a été sauvegardée sur cette mémoire persistante.

```
0: 1 <kernel>

294: 1 /init
295: 1 /sbin/adbd
296: 1 /system/bin/logcat
297: 1 /system/bin/akmd
298: 1 /system/bin/app_process
299: 1 /system/bin/app_process
300: 1 /system/bin/bluetoothd
301: 1 /system/bin/bootanimation
302: 1 /system/bin/brcm_patchram_plus
303: 1 /system/bin/dbus-daemon
304: 1 /system/bin/debuggerd
305: 1 /system/bin/dspcrashd
306: 1 /system/bin/handle_compcache

316: 1 /system/bin/installd
317: 1 /system/bin/keystore
318: 1 /system/bin/mediaserver
321: 1 /system/bin/netd
323: 1 /system/bin/rild
324: 1 /system/bin/sdptool
325: 1 /system/bin/servicemanager
326: 1 /system/bin/vold
328: 1 /system/bin/wpa_supplicant
```

Listing 5.4 – Extrait de l'arbre de transition de domaines pour Android

L'arbre des transitions de domaines obtenu (5.4) est relativement petit, même si pour des raisons de compacité certains domaines ont été omis [8].

```
329: 1 * /sbin/adbd
330: 1 /system/bin/logcat

432: 1 * /system/bin/app_process
```

Pour la suite, l'arbre est découpé en deux parties distinctes. Une première partie va comprendre l'ensemble des démons du système dont le rôle dans le système est clairement défini, c'est à dire que leur comportement ne va pas varier au cours de la durée de vie du téléphone. Ces processus comprenent entre autres la gestion du *Bluetooth*, `bluetoothd`, ou de la partie radio du téléphone, `rild`. Les applications qui font partie de ces domaines n'ont donc besoin que d'un apprentissage rapide, et une fois les autorisations nécessaires connues, il est possible d'appliquer la politique de sécurité obtenue.

L'autre partie va comprendre les applications riches (5.2.2), c'est à dire tout particulièrement le serveur de débogage `adbd` ainsi que les machines virtuelles Dalvik. `adbd` devant être utilisé pour travailler sur le téléphone et n'étant à priori pas démarré [9], nous nous sommes concentré sur la machine virtuelle qui présente un comportement riche. Cependant, dans la pratique, l'ACL obtenue pour Dalvik se factorise facilement (5.5). On notera en particulier la présence du modèle `/system/lib/\@.so`

8. parmis ces domaines on retrouve ceux qui correspondent aux programmes inclus dans `toolbox` ou de manière plus générale les scripts

9. il serait inutile de sécuriser un système embarqué si l'on conserve une porte dérobée

pour identifier l'ensemble des librairies natives installées ; la suppression de ce modèle permet de restreindre les librairies natives auxquelles Dalvik peut se lier dynamiquement.

```
<kernel> /system/bin/app_process

use_profile 3
allow_create /data/data/\@.\@.\@/\*/\*
allow_ioctl /data/data/\@.\@.\@/\*/\*
allow_ioctl socket:[\$]
allow_mkdir /data/data/\@.\@.\@/\*/
allow_read /system/app/\@.apk
allow_read /system/fonts/\@.ttf
allow_read /system/framework/android.policy.jar
allow_read /system/framework/core.jar
allow_read /system/framework/ext.jar
allow_read /system/framework/framework.jar
allow_read /system/framework/services.jar
allow_read/write /acct/uid/\$/tasks
allow_read/write /data/data/\@.\@.\@/\*/\*
allow_read/write /dev/ashmem
allow_read/write /dev/binder
allow_read /system/lib/\@.so
allow_rename /data/data/\@.\@.\@/\*/\* /data/data/\@.\@.\@/\*/\*
```

Listing 5.5 – Extrait de la politique finale appliqué à Dalvik

5.2.3 Implémentation de la diode

Nous allons montrer dans cette dernière partie de quelle façon nous pouvons implémenter la diode ainsi que d'une manière plus générale le flot d'information dans les conteneurs qui servent de tampon entre les conteneurs haut et bas.

TOMOYO Linux ne permettant pas de labéliser des objets [10] comme le fait SELinux [11], nous ne pouvons pas simplement marquer une information et la suivre. Cependant, TOMOYO Linux identifie les objets par leurs chemins absolu ; la solution consiste donc à associer le chemin absolu à la classification de l'information.

Pour cela, nous créons donc deux répertoires dans le VFS de la diode, `/var/high` et `/var/low`, associés respectivement aux classifications d'information haute et basse. Nous ajoutons ensuite dans TOMOYO Linux deux modèles (5.6) pour identifier les fichiers présents dans ces répertoires.

```
file_pattern    /var/high/\*
file_pattern    /var/low/\*
```

Listing 5.6 – Identification des objets hauts et bas

Enfin, pour assurer le transfert de l'information, nous avons besoin de trois services. Un récepteur *(receiver)* qui sera chargé d'écouter sur l'interface virtuelle `Veth` connecté au conteneur bas et de copier l'information reçu dans `/var/low/`. Un émetteur *(emitter)* qui sera chargée d'envoyer l'information présente dans `/var/high/` vers le conteneur haut. Enfin, la diode à proprement parler qui sera chargée de transférer l'information de `/var/low/` vers `/var/high/`. Les ACL correspondantes (5.7,5.8 et 5.9) comprendront au minimum les permissions pour faire circuler cette informa-

10. ni opérer sur des flux réseau
11. et encore moins de suivre un flot d'information

tion. Ces applications font usage de l'interface `inotify` de Linux [12] afin d'opérer ces transferts sans aucune intervention de l'utilisateur ou utilisation d'un *job* `cron`.

```
1 allow_write   /var/low/\*
```

Listing 5.7 – ACL pour l'application `receiver`

```
1 allow_write   /var/high/\*
  allow_read    /var/low/\*
```

Listing 5.8 – ACL pour l'application `diode`

```
1 allow_read    /var/high/\*
```

Listing 5.9 – ACL pour l'application `transmitter`

Pour le conteneur chargé de la dé-classification de l'information par chiffrement, la configuration se fait *mutatis mutandis*.

5.3 Évaluation des performances

L'ajout de mécanismes de sécurité sur une architecture existante se fait au détriment de son utilisabilité. Ceci se traduit dans la pratique par deux observations :

- une perte de fonctionalités ;
- une consomation supplémentaire de ressources.

Le premier élément se comprend intuitivement ; Les mécanismes ajoutés restreignent voire interdisent des fonctions autrefois offertes par le système. Dans le cas où ces fonctions n'avaient pas lieu d'être, l'impact est nul et la sécurité du système peut se voir renforcée par le respect du principe de moindres privilège. À l'inverse, lorsque celles-ci sont nécessaire à son fonctionnement, cela résulte d'une mauvaise définition des périmètres à couvrir et peut contrevenir au principe de disponibilité.

Le second élément se comprend tout aussi intuitivement ; l'ajout de ces mécanismes implique une charge supplémentaire au système. Ceci pourra se traduire en pratique par :

- non-respect de contraintes temporelles sur un système temps-réel ;
- perte d'autonomie sur un système embarqué ;
- limitation du temps utile de processeur disponible.

Nous chercherons dans la partie suivante à évaluer la charge supplémentaire induite par les ajouts effectués pour le besoin du stage, plus particulièrement en terme de consommation de temps processeur.

5.3.1 Considérations

L'évaluation de la charge supplémentaire induite sur la consommation du temps processeur, plus courament appelée *overhead*, est un exercice périlleux à plus d'un titre. Tout d'abord, il est difficile d'établir un étalon en raison de la disparité des systèmes. Ensuite, il n'est pas forcément aisé de placer un système existant dans un état neutre propice aux mesures. Enfin, les résultats pouvant trouver leurs explications en allant du matériel au logiciel, l'interprétation des mesures peut se révéler complexe à plus d'un titre.

L'*overhead* se mesure à l'aide de tests de performances ou *benchmarks*. Ceci se subdivisent en deux catégories :

12. cette interface permet à un processus d'être notifié par le noyau lorsqu'un changement est effectué sur un inode du VFS

· des *micro-benchmarks* ;
· des *macro-benchmarks* ;

Les micro-benchmarks se concentrent sur la mesure d'un élément en essayant d'obtenir la granularité la plus fine possible. Ces tests ne sont pas capables de fournir une représentation du comportement macroscopique d'un système. Néanmoins, ils sont utiles pour s'en faire une idée à priori ainsi que l'expliquer au besoin. Les macro-benchmarks quand à eux observent le système dans son ensemble. Ils donnent une vision plus précise de l'impact sur le système, au dépens de la possibilité d'être capable d'en expliquer les résultats.

5.3.2 Performances des appels systèmes

Nous mesurons ici dans un premier temps les performances des appels systèmes ; Les mécanismes ajoutés, en particulier LSM, y étant directement liés, on obtient ainsi une vue très fine de l'impact sur le système.

Les mesures ont été réalisées à l'aide de l'outil `lmbench` fourni par le projet Debian. Plusieurs cas ont été considérés :

· *vanilla* : système Android original sur le noyau modifié, politique de sécurité DAC ;
· *TOMOYO (empty)* : système Android original sur le noyau modifié, politique de sécurité MAC vide ;
· *TOMOYO* : système Android original sur le noyau modifié, politique de sécurité MAC en cours d'apprentissage (~300 domaines) ;
· *TOMOYO+lxc (empty)* : système Debian dans un conteneur Linux, politique de sécurité MAC vide ;
· *TOMOYO+lxc* : système Debian dans un conteneur Linux, politique de sécurité MAC en cours d'apprentissage (idem).

Un ensemble de 20 jeux ont été lancé pour chaque cas sur le système Android modifié et au repos et tournant sur un processeur Snapdragon QSD8250 [13].

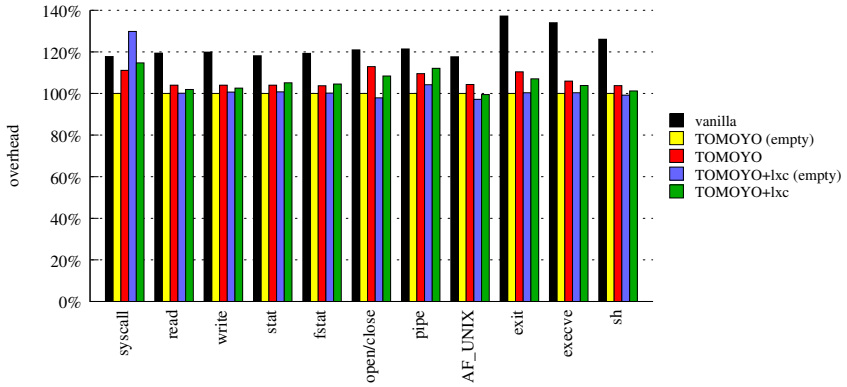

FIGURE 5.1 – Comparaison du temps d'exécution des appels système (%)

On observe tout d'abord que le système original est quasi-systématiquement plus lent sur chacun des tests ; Ce résultat est bien évidement contraire à ce à quoi l'on pourrait s'attendre et n'a pas pu être expliqué. Les mesures de charge supplémentaire

13. architecture ARMv7

	TOMOYO (empty)	TOMOYO	TOMOYO+lxc (empty)	TOMOYO+lxc
syscall	.3226	.3584 (+11.10%)	.4188 (+29.82%)	.3701 (+14.72%)
read	.7339	.7631 (+3.98%)	.7347 (+.11%)	.7481 (+1.93%)
write	.6406	.6661 (+3.98%)	.6449 (+.67%)	.6572 (+2.59%)
stat	4.6605	4.8472 (+4.01%)	4.6980 (+.80%)	4.9000 (+5.14%)
fstat	.9254	.9597 (+3.71%)	.9272 (+.19%)	.9673 (+4.53%)
open/close	7.0475	7.9577 (+12.92%)	6.8979 (-2.12%)	7.6413 (+8.43%)
pipe	12.5385	13.7343 (+9.54%)	13.0670 (+4.22%)	14.0570 (+12.11%)
AF_UNIX	22.1425	23.1014 (+4.33%)	21.5187 (-2.82%)	22.0259 (-.53%)
exit	725.0326	800.4245 (+10.40%)	727.4403 (+.33%)	775.9279 (+7.02%)
execve	796.5801	843.9424 (+5.95%)	799.4588 (+.36%)	827.2453 (+3.85%)
sh	3942.9166	4092.8500 (+3.80%)	3910.9000 (-.81%)	3990.6833 (+1.21%)

TABLE 5.1 – Comparaison du temps d'exécution des appels système (ms)

se sont donc faites en prenant pour étalon le système original avec une politique MAC vide.

Deux constatations ressortent de ces mesures. Tout d'abord on observe comme attendu une croissance dans le temps d'exécution des appels systèmes lorsque la politique MAC se complexifie. Cela s'observe par une augmentation de la latence comprise entre 3% et 13%. Deuxièmement, l'utilisation de conteneurs n'a pas d'impact notable sur le temps d'exécution ; on obtient même en pratique un gain sur certains appels systèmes. On pourra aussi constater, si l'on regarde par exemple les appels système `read` et `write`, que ces latences ne sont pas cumulatives.

5.3.3 Performance des connexions réseau

Nous nous intéressons ici dans un deuxième temps aux performances des communications inter-processus. Plus particulièrement, nous allons observer l'impact sur la bande passante maximale sur des sockets lors d'une communication sur un réseau virtuel qui relie l'hôte aux conteneurs.

De même, plusieurs cas pouvant correspondre à des cas pratiques ont été évoqués :

- *hors-conteneur* : communication entre deux applications en dehors de tout conteneur ;
- *intra-conteneur* : communication entre deux applications situées dans le même conteneur ;
- *inter-conteneur* : communication entre deux applications situées dans deux conteneurs séparés ;
- *conteneur-hôte* : communication entre une application située dans un conteneur et une application hors-conteneur ;

Pour se faire, nous réalisons un réseau virtuel interne au système. Celui-ci se compose :

- d'une interface pont virtuel sur la machine hôte ;
- d'une paire d'interfaces réseau virtuel, une extrémité étant liée à l'interface pont virtuel de l'hôte.

Les mesures ont été réalisées à l'aide de l'outil `iperf` sur un système au repos.

	TCP	UDP
hors-conteneur	964	598
intra-conteneur	948	575
conteneur-hôte	395	266
inter-conteneur	387	219

TABLE 5.2 – Comparaison de la bande passante maximale possible en TCP et UDP (Mbps)

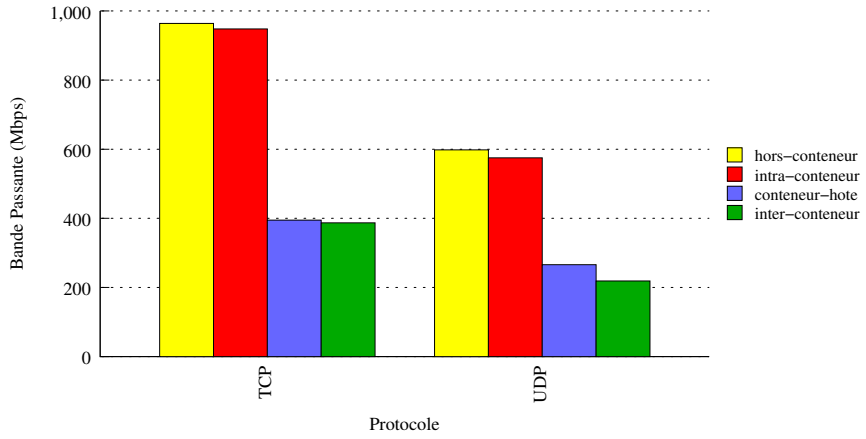

FIGURE 5.2 – Comparaison de la bande passante maximale possible en TCP et UDP (Mbps)

Ces mesures donnent lieu à deux observations [14]. Tout d'abord, l'écart en bande passante entre les mesures « hors-conteneur » et « intra-conteneur » est faible, ce qui confirme que la surcharge due à l'utilisation des conteneurs est légère. Ensuite, la présence du réseau virtuel a un coût significatif.

5.3.4 Conclusions

L'ajout de mécanismes de sécurité sur un système impacte toujours celui-ci. Nous avons observé qu'à petite échelle l'utilisation d'une politique MAC concurremment avec des conteneurs n'avait qu'un effet limité sur le temps d'exécution des appels système. Nous avons vu ensuite qu'à priori ceux-ci ne devrait pas avoir plus d'effet à grand échelle, ce qui s'est confirmé en mesurant la bande passante sur un réseau virtuel.

14. Le fait que la bande passante en UDP soit inférieure à celle en TCP peut surprendre le lecteur. Ceci est du en pratique à une taille de buffer différente entre les deux protocoles. Pour ces mesures, les tailles de buffer par défaut ont été conservées.

Chapitre 6

Conclusion

Au cours de ce stage, nous avons montré qu'il était possible de créer une politique de sécurité multi-niveau adaptée à un système embarqué sous Android sans opérer de changements importants. Pour cela, nous avons croisés plusieurs mécanismes de sécurité déjà existants sous Linux.

Nous avons montré dans un premier de quelle façon nous souhaitions adapter le modèle de Bell et Lapadula à Android. Nous avons montré ensuite que nous pouvions implémenter ce modèle à l'aide en particulier des conteneurs Linux et de TOMOYO Linux. Enfin, nous avons montré que cela n'avait eu qu'un impact faible sur Android en termes de consommation de ressources.

Cette approche, bien que fonctionnelle, n'est cependant pas nécessairement suffisante. En effet, nous nous sommes limités dans notre étude à un seul domaine avec deux niveaux de classifications ; L'utilisation de plus de domaines et de niveaux de classification ne devrait pas donner des résultats aussi favorables, en particulier en termes d'espace de stockage nécessaire pour chacun des conteneurs.

Chapitre 7

Travaux futurs

Ayant à notre disposition encore quelques mois de stage, deux pistes sont à ce jour évoquées quand aux travaux futurs.

La première piste consisterait à implémenter un LSM propre à Android car nous n'avons pas trouvé l'approche par chemins absolus de TOMOYO Linux entièrement satisfaisante. En effet, il n'est pas possible de labéliser explicitement un fichier, seulement implicitement par son emplacement dans le VFS et il n'est pas non plus possible de labéliser explicitement un processus.

Nous réfléchissons spécifiquement à une méthode pour labéliser les processus et capable de travailler sur des machines virtuelles qui serait couplé à une classification des fichiers par chemins absolu couplé à un suivi à chaud.

La seconde piste s'éloigne de ce qui a été traité au cours de ce stage pour s'intéresser aux possibilités offertes par une MTM logicielle qui nous devrions avoir à disposition sous peu, en particulier l'utiliser avec les mesures d'intégrité EVM et IMA présents dans Linux qui font usage de TPM.

Acronymes

ACL	*Access Control List*
AIDL	Android Interface Description Language
AML	*Android Measurement Log*
CRTM	*Core Root of Trust Measurement*
DAC	*Discretionary Access Control*
DTE	*Domain Type Enforcement*
EMSCB	*European Multilaterally Secure Computing Base*
IPC	*Inter Process Communication*
LKM	*Loadable Kernel Module*
LSM	*Linux Security Modules*
MAC	*Mandatory Access Control*
MLS	*Multi-Level Security*
MLTM	*Mobile Local owner Trusted Module*
MMU	*Memory Management Unit*
MRTM	*Mobile Remote owner Trusted Module*
MSSF	*Mobile Simplified Security Framework*
MTM	*Mobile Trusted Module*
OHA	*Open Handset Alliance*
OS	*Operating System*
PCR	*Platform Configuration Register*
SiP	*System in Package*
SML	*Stored Measurement Log*
SoC	*System on Chip*
TCB	*Trusted Computing Base*
TCG	*Trusted Computing Group*
TCG-MW	*Trusted Computing Group - Mobile Workgroup*
TPM	*Trusted Platform Module*
TrEE	*Trusted Execution Environnement*
UML	User-Mode Linux
URI	*Uniform Resource Identifier*
VFS	*Virtual File System*
VM	Machine Virtuelle *(Virtual Machine)*
VMM	*Virtual Machine Monitor*

Bibliographie

[AF] J. Azema and G. Fayad, *M-Shield mobile security technology : making wireless secure*, Texas Instruments Whitepaper.

[AF04] T. Alves and D. Felton, *Trustzone : Integrated hardware and software security*, ARM white paper (2004).

[AF05] ———, *Building a Secure System using TrustZone Technology*, ARM white paper (2005).

[BM77] K.J. Biba and MITRE CORP BEDFORD MA, *Integrity considerations for secure computer systems*.

[Cha09] A. Chaudhuri, *Language-based security on Android*, Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, ACM, 2009, pp. 1–7.

[EB09] J.E. Ekberg and S. Bugiel, *Trust in a small package : minimized MRTM software implementation for mobile secure environments*, Proceedings of the 2009 ACM workshop on Scalable trusted computing, ACM, 2009, pp. 9–18.

[EOM09] W. Enck, M. Ongtang, and P. McDaniel, *Understanding android security*, Security & Privacy, IEEE 7 (2009), no. 1, 50–57.

[Gol74] R.P. Goldberg, *Survey of virtual machine research*, IEEE Computer 7 (1974), no. 6, 34–45.

[Hil09] M. Hills, *Native OKL4 Android Stack*.

[HRU76] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman, *Protection in operating systems*, Communications of the ACM **19** (1976), no. 8, 461–471.

[Kas] D. Kasatkin, *Mobile Simplified Security Framework*.

[KEH+09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al., *seL4 : Formal verification of an OS kernel*, Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM, 2009, pp. 207–220.

[KS] K. Kursawe and D. Schellekens, *Flexible μTPMs through Disembedding*, status : published.

[LB96] L.J. LaPadula and D.E. Bell, *Secure computer systems : A mathematical model*, Journal of Computer Security **4** (1996), 239–263.

[Mor10] B. Morin, *Modèle de sécurité d'Android*, MISC (2010), 25–29.

[MPWa] TCG MPWG, *TCG Mobile Reference Architecture*, TCG specification Version **1**.

[MPWb] _____, *The TCG mobile trusted module specification*, TCG specification version 0.9 revision **1**.

[MSS+08] D. Muthukumaran, A. Sawani, J. Schiffman, B.M. Jung, and T. Jaeger, *Measuring integrity on mobile phone systems*, Proceedings of the 13th ACM symposium on Access control models and technologies, ACM, 2008, pp. 155–164.

[NKZS10] M. Nauman, S. Khan, X. Zhang, and J.P. Seifert, *Beyond Kernel-Level Integrity Measurement : Enabling Remote Attestation for the Android Platform*, Trust and Trustworthy Computing (2010), 1–15.

[PB03] S. Pearson and B. Balacheff, *Trusted computing platforms : TCPA technology in context*, Prentice Hall PTR, 2003.

[PSvD] R. Perez, R. Sailer, and L. van Doorn, *vTPM : virtualizing the trusted platform module.*

[SFE09] A. Shabtai, Y. Fledel, and Y. Elovici, *Securing Android-Powered Mobile Devices Using SELinux*, IEEE Security and Privacy (2009).

[SFK+09] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev, *Google Android : A State-of-the-Art Review of Security Mechanisms*, Arxiv preprint arXiv :0912.5101 (2009).

[SFK+10] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, *Google Android : A comprehensive security assessment*, Security & Privacy, IEEE **8** (2010), no. 2, 35–44.

[SKK08] A.U. Schmidt, N. Kuntze, and M. Kasper, *On the deployment of mobile trusted modules*, Wireless Communications and Networking Conference, 2008. WCNC 2008. IEEE, IEEE, 2008, pp. 3169–3174.

[STP08] D. Schellekens, P. Tuyls, and B. Preneel, *Embedded trusted computing with authenticated non-volatile memory*, Trusted Computing-Challenges and Applications (2008), 60–74.

[TCG] PC TCG, *Client Specific Implementation Specification for Conventional BIOS*, TCG specification Version.

[TPMa] TCG TPM, *TPM Main Specification Level 2 Version 1.2, Part 1 - Design Principles*, TCG specification Version **1**.

[TPMb] _____, *TPM Main Specification Level 2 Version 1.2, Part 2 - Structure of the TPMs*, TCG specification Version **1**.

[TPMc] _____, *TPM Main Specification Level 2 Version 1.2, Part 3 - Commands*, TCG specification Version **1**.

[Win08] J. Winter, *Trusted computing building blocks for embedded linux-based ARM trustzone platforms*, Proceedings of the 3rd ACM workshop on Scalable trusted computing, ACM, 2008, pp. 21–30.

[ZAS07] X. Zhang, O. Acıiçmez, and J.P. Seifert, *A trusted mobile phone reference architecture via secure kernel*, Proceedings of the 2007 ACM workshop on Scalable trusted computing, ACM, 2007, pp. 7–14.