Detection externalisée de vulnerabilités pour la plateforme Android à l'aide du langage OVAL

Gaëtan Hurel

UNIVERSITÉ DE STRASBOURG

Master 2
Réseaux Informatiques et Systèmes Embarqués

Présenté par

Gaëtan Hurel
gaetan.hurel@[etu.unistra.fr|inria.fr]

Détection externalisée de vulnérabilités pour la plateforme Android à l'aide du langage OVAL

Encadré par

Rémi Badonnel & Olivier Festor
remi.badonnel@loria.fr - olivier.festor@inria.fr

Au sein de

Équipe Madynes - LORIA/INRIA Grand-Est

Juin 2013

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Inria
INVENTEURS DU MONDE NUMÉRIQUE

Préface

Ce rapport est le résultat du travail réalisé lors de mon stage de fin d'études au sein de l'équipe Madynes [1] du LORIA [2], au centre de recherche INRIA [3] Grand-Est. Ce stage a été réalisé dans le cadre du cursus de deuxième et dernière année du Master RISE [4] (promotion 2012/2013) proposé par le département Mathématiques-Informatique de l'Université de Strasbourg.

L'INRIA est un établissement publique de recherche comprenant huit centres à travers la France et rassemblant 1 800 chercheurs, en plus de 1 600 universitaires ou chercheurs d'autres organismes. Le LORIA est un laboratoire de recherche rattaché notamment au CNRS [5], à l'Université de Lorraine et au centre INRIA Grand-Est. Il compte 500 membres et 27 équipes, dont l'équipe Madynes. Celle-ci se compose d'une vingtaine de doctorants, chercheurs et ingénieurs se concentrant principalement sur la recherche appliquée et expérimentale pour la gestion de l'Internet du futur. Plus précisément, l'équipe s'intéresse à l'étude et la conception de nouvelles approches pour la configuration, la surveillance et la sécurité des protocoles de communications et des services du futur.

Je profite de ces quelques lignes pour remercier Olivier Festor et Rémi Badonnel - sans oublier Stéphane Cateloin - de m'avoir donné l'opportunité de réaliser ce stage. Je remercie également tous les membres de l'équipe Madynes pour l'expérience humaine et scientifique que j'ai pu acquérir à leurs côtés, et particulièrement Martín Barrère pour sa disponibilité et son aide fort précieuse.

1. MAnaging DYnamic NEtworks and Services
2. Laboratoire Lorrain de Recherche en Informatique et ses Applications
3. Institut National de Recherche en Informatique et Automatique
4. Réseaux Informatiques et Systèmes Embarqués
5. Centre National de la Recherche Scientifique

Table des matières

Chapitre 1

Introduction

L'utilisation des systèmes mobiles a évolué de façon exponentielle ces dernières années. Cela s'explique par leur capacité à intégrer de nouvelles technologies d'une part, et par l'étoffement des services qu'ils proposent aux utilisateurs d'autre part. Cette pluralité des services a engendré leur popularisation auprès du grand public (*e.g.* smartphones). Aussi, il est prévu que le nombre de systèmes mobiles dépasse celui de la population mondiale d'ici la fin de l'année 2013 [19]. Notre travail se focalise principalement sur la plateforme Android [9], aujourd'hui leader sur le marché des smartphones et tablettes tactiles à travers le monde. L'omniprésence de ces systèmes permet aux utilisateurs de réaliser de nombreuses activités, impliquant possiblement des montants considérables de données sensibles. Néanmoins, les applications, services, et systèmes d'exploitation de ces terminaux exposent ces utilisateurs à des menaces de sécurité variées. Les milliers d'applications disponibles en téléchargement depuis le marché officiel [1] (ou autres) souffrent d'un manque de filtrage évident, entraînant la présence de programmes malicieux capables d'exploiter les faiblesses des systèmes. De plus, le cycle de correction de vulnérabilités (patching) pour la plateforme Android est plus long que la moyenne, augmentant ainsi la période d'exposition de ses terminaux.

Les smartphones et autres stations mobiles proposent diverses applications en plus de la téléphonie classique, comme la navigation web, les jeux vidéos et la gestion de données personnelles. Des services sont aussi exécutés en tâche de fond afin notamment de contrôler le comportement global de chaque système. Toutes ces activités engendrent une consommation de ressources non négligeable sur les stations mobiles. Cette charge de travail doit être minimisée afin de maximiser leurs performances et leur réactivité. L'utilisation excessive de ressources par une application ou un service peut entraîner la désactivation volontaire de celui-ci par l'utilisateur. Cette limite représente un point bloquant que nous désirons prendre en compte. Le déploiement à grande échelle des systèmes mobiles, combiné à leurs problèmes de sécurité actuels et leurs restrictions de ressources, forme en effet un challenge de taille qui doit être négocié efficacement. Un tel contexte soulève et accentue le besoin de solutions légères et transparentes pour la détection de vulnérabilités sur les systèmes, afin d'augmenter leur sécurité en diminuant leur exposition aux menaces.

Nous proposons dans ce rapport une approche novatrice pour analyser les systèmes Android et détecter leurs vulnérabilités de façon légère. Cette approche regroupe les principales composantes du processus d'analyse sous forme de service externalisé que les clients mobiles peuvent ensuite exploiter à l'aide d'un agent minimal. Le langage OVAL [2] [11] est utilisé comme support pour la description et l'analyse de vulnérabilités. En configurant la fréquence des analyses et le pourcentage de vulnérabilités à traiter au cours de chacune d'entre elles, l'approche proposée permet de limiter l'allocation de ressources côté client et de transférer les différents traitements sur des serveurs distants. La stratégie employée consiste à partager et distribuer les analyses à travers le temps pour réduire significativement l'activité sur les systèmes mobiles, tout en assurant un traitement de la totalité des vulnérabilités connues dans un laps de temps fini. De cette méthodologie résulte un processus d'analyse orienté cloud plus léger et plus rapide, pouvant limiter de façon significative la consommation de ressources et d'énergie côté client.

1. Google Play Store
2. Open Vulnerability and Assessment Language

Contributions

Les contributions principales de ce travail sont :

1. un modèle mathématique supportant l'approche proposée ;
2. un framework d'analyses basé sur le langage OVAL pour la détection de vulnérabilités sur les systèmes Android de façon peu coûteuse pour ces derniers ;
3. un prototype d'implémentation du framework et de nombreuses expérimentations montrant la faisabilité de notre solution.

Structure du rapport

La suite de ce document se présente comme suit : le chapitre 2 donne une vue générale des systèmes mobiles, de leurs limites actuelles en termes de sécurité, et des possibles exploitations du langage OVAL dans ce contexte. Le chapitre 3 présente le modèle mathématique de l'approche probabiliste que nous proposons pour la détection de vulnérabilités, ainsi que l'architecture du framework exploitant ce modèle. Le chapitre 4 décrit les détails d'implémentation du prototype du framework et fournit une analyse de ses performances, extraites par le biais de nombreuses expérimentations. Finalement, le chapitre 5 donne une conclusion de ce travail et ouvre de nouvelles perspectives quant à nos travaux futurs.

Chapitre 2

État de l'art

Sommaire

Ce chapitre décrit le contexte scientifique et les prérequis du travail qui a donné lieu à ce rapport. L'environnement mobile et ses spécificités sont abordés dans la première section. La seconde section de ce chapitre présente les aspects sécuritaires qui peuvent s'appliquer aux plateformes mobiles, en particulier pour la gestion et la détection de leurs vulnérabilités.

2.1 Avènement des systèmes mobiles

Par le biais de nouvelles technologies couplées à l'évolution constante des besoins utilisateurs, les systèmes mobiles ont pris une importance majeure dans notre vie quotidienne durant ces dernières années. Cette section présente les principales caractéristiques de l'environnement mobile, ainsi que les différentes méthodologies d'attaques auxquelles les systèmes de ce type s'exposent. La plateforme Android, sur laquelle notre choix s'est porté comme support d'étude, est finalement introduite.

2.1.1 Environnement mobile

Cette sous-section donne une définition générale des systèmes mobiles dans un premier temps, et des smartphones dans un second temps. Les technologies développées pour la transmission de données et les communications entre ces systèmes sont ensuite brièvement détaillées.

Caractérisation générale

Du fait de leur évolution constante, caractériser les systèmes mobiles de façon exacte et complète s'apparente à une tâche peu aisée. En [7], un ensemble de critères est cependant donné offrant ainsi un référentiel de comparaison avec une station non mobile :

- **Taille réduite** : par nature, un terminal mobile est destiné à changer fréquemment d'emplacement. Il adopte une taille réduite pour faciliter cette tendance ;
- **Connectivité** : un système mobile dispose d'une interface réseau au minimum. Cette interface peut utiliser les technologies sans-fil traditionnelles (*e.g.* Wi-FI), les réseaux cellulaires ou d'autres technologies pour intégrer le système dans les infrastructures lui offrant une connectivité réseau ;

- **Disques et mémoire intégrés** : les support principaux de mémoire pour le système (*e.g.* RAM) ou le stockage de données (*e.g.* disques durs) sont généralement intégrés au terminal, et donc non changeables ;
- **OS** [1] **embarqués** : les systèmes d'exploitation équipant les terminaux sont spécialement adaptés aux plateformes mobiles. Ils se montrent plus légers que les systèmes d'exploitation d'ordinateurs de bureau par exemple ;
- **Synchronisation native** : les systèmes mobiles sont couramment dotés de fonctionnalités leur permettant de se synchroniser avec des stations distantes (*e.g.* ordinateur personnel, serveurs d'opérateur mobiles, serveurs tierce-partie) ;
- **Ressources limitées** : les terminaux bénéficient de ressources (CPU, RAM, autres) relativement restreintes, à cause notamment de leur taille réduite. Leur autonomie peut aussi se montrer limitée car il est fréquent qu'ils fonctionnent sur batterie. Ces caractéristiques demeurent une différence critique avec les stations de travail non mobiles. Les ressources des systèmes mobiles sont peu adaptées pour l'exécution de tâches lourdes (*e.g.* HIDS [2]) et doivent être gérées efficacement pour favoriser leur durée de vie et leur réactivité.

Caractérisation des smartphones

Le contexte de ce travail se limite principalement aux smartphones, des téléphones portables évolués offrant de nombreuses fonctionnalités (*e.g.* web, multimédia, jeux vidéos) autres que l'émission et la réception d'appels ou de messages. Les smartphones ont connu un fort succès ces dernières années et constituent une sous-catégorie majeure des systèmes mobiles actuels, à laquelle s'ajoutent les caractéristiques suivantes :

- **Forte mobilité** : comme tous téléphones portables, les smartphones sont pensés pour équiper leur utilisateur de manière quasi-continue ;
- **Forte connectivité** : les interfaces réseaux des smartphones sont capables d'exploiter les réseaux cellulaires, les technologies Wi-Fi ou autres. Il en résulte que les systèmes disposent d'une forte connectivité. Ils s'intègrent de plus parfaitement aux réseaux IP traditionnels ;
- **Forte personnalisation** : le propriétaire d'un smartphone en est généralement le seul utilisateur. Un smartphone est donc enclin à contenir de nombreuses données plus ou moins privées relatives à son possesseur ;
- **Déploiement d'applications** : l'utilisation de systèmes d'exploitation permet l'ajout de couches d'abstraction pour le développement d'applications ; ces applications sont installées sur les systèmes par différents moyens (*e.g.* incluses de base avec l'OS, ou disponibles depuis les marchés d'applications) ;
- **Multimédia** : les smartphones disposent de périphériques spécifiques aux activités multimédia, tels qu'un écran, une carte son, un microphone et une caméra intégrés ;
- **Traçabilité** : le système de localisation GPS [3] est très fréquemment inclus de base dans les smartphones actuels, permettant ainsi leur traçabilité à différents degrés.

Télécommunications et téléphonie mobile

Les smartphones et autres systèmes mobiles disposent de plusieurs technologies et standards pour communiquer et transférer des données [49]. Parmi ces technologies :

- **GSM** : *Global System for Mobile communications* (GSM, 1990) est un standard de communications utilisé dans les réseaux mobiles de deuxième génération (2G). Il permet la création de réseaux cellulaires basiques où les téléphones mobiles communiquent entre eux par l'intermédiaire de BTS [4], et autres équipements réseaux spécifiques assurant le relais et la correspondance des données. GSM est le premier standard permettant l'introduction de nouveaux services simples orientés données (*e.g.* SMS [5], fax, mails) ;

1. Operating System
2. Host Intrusion Detection System
3. Global Positioning System
4. Base Transceiver Station
5. Short Message Service

- **GPRS/EDGE** : *General Packet Radio Service* (GPRS, fin des années 1990 - 2.5G) et *Enhanced Data rates for GSM Evolution* (EDGE, 2000 - 2.75G) sont deux standards qui étendent l'architecture de GSM et améliorent ses performances, en termes de débit notamment. Premier standard de téléphonie mobile compatible IP, GPRS utilise des techniques de commutation de paquets pour permettre le transfert de données entre utilisateurs, permettant le déploiement de services multimédia tels que le WAP [6] ou le MMS [7]. EDGE à quant à lui été développé ultérieurement pour améliorer les fonctionnalités offertes par GPRS, en augmentant le débit et la stabilité réseau.
- **UMTS** : *Universal Mobile Telecommunications System* (UMTS, 2002) est un standard utilisé par les réseaux mobiles 3G. Il supporte les commutations de circuits ou de paquets, permettant d'intégrer à son architecture les standards plus anciens (GSM, GPRS, EDGE) avec lesquels il se montre compatible. À la différence d'EDGE et de GPRS, UMTS nécessite de nouveaux type d'antennes radio (Node B) et de nouvelles bandes de fréquence. Le cœur du réseau se base en grande partie sur IP (*e.g.* IMS [8]) pour offrir aux utilisateurs divers types de nouveaux services (*e.g.* multimédia), et différentes classes de services (conversation, streaming, attente, autres..) en fonction de leur activité.
- **LTE/LTE Advanced** : *Long Term Evolution* (LTE, de 2004 à 2006 - 3.9G) est un standard de téléphonie mobile compatible GSM/GPRS/UMTS conçu pour améliorer les performances de ces derniers, mais aussi pour simplifier leur cœur de réseau. Longtemps baptisé standard 4G, la norme souffre de quelques manques (*e.g.* débit trop faible) invalidant cette appellation. Son évolution, la norme *LTE Advanced* (2011), améliore ces points sombres et se voit ainsi baptisée premier standard 4G. Les réseaux LTE Advanced utilisent, comme le LTE, de nouvelles bandes de fréquence, un système radio évolué (evolved Node B), et un cœur de réseau basé intégralement sur les protocoles IP (Evolved Packet Core) pour la signalisation et le transport de la voix et des données. Les réseaux mobiles 4G sont actuellement en cours de déploiement.

Technologies réseau

En parallèle des standards propres à la téléphonie sans-fil, certaines technologies réseau peuvent aussi être utilisées par les systèmes mobiles. Ces technologies ne sont pas toutes spécifiques à l'environnement. Parmi elles se trouvent :

- **Wireless LAN IEEE 802.11 (Wi-Fi)** : IEEE 802.11 est une famille de standards pour les réseaux WLAN [9] comprenant un ensemble de protocoles de communications sans-fil. Les standards IEEE 802.11 peuvent être utilisés sous deux modes différents, i) le mode infrastructure, où une station (point d'accès) est chargée de coordonner les clients sans-fil et leur accès au réseau, et ii) le mode ad-hoc, où les clients ont à leur charge la gestion de l'accès au réseau. Les protocoles Wi-Fi les plus répandus sont 802.11b (11Mbps), 802.11g (54Mbps) et dernièrement 802.11n (300Mbps grâce aux techniques MIMO [10]).
- **IEEE 802.15.1 - Bluetooth** : Bluetooth est un standard conçu pour l'interconnexion de clients mobiles et l'échange de données dans un rayon réduit. Il permet la création de réseaux de type WPAN [11] sécurisés ou non (3 modes de sécurité sont supportés par le standard). Les transmissions radios Bluetooth opèrent sur la même bande de fréquence que le Wi-Fi (2.4 GHz). Elles sont de faibles portées (1-100m) et peu coûteuses en termes de consommation énergétique.

Par ailleurs, d'autres techniques de transmissions de données telles que le RFID [12] ou le NFC [13] peuvent aussi être utilisées par les smartphones et autres terminaux mobiles. Ces technologies ne sont pas abordées en détail dans ce document du fait notamment de leur relative immaturité.

6. Wireless Application Protocol
7. Multimedia Messaging Service
8. IP Multimedia Subsystem
9. Wireless Local Area Network
10. Multiple-Input Multiple-Output
11. Wireless Personnal Area Network
12. Radio-Frequency IDentification
13. Near Field Communication

2.1.2 Attaques sur les systèmes

La convergence des technologies sans-fil et l'intégration totale des nouveaux réseaux mobiles aux réseaux IP a entraîné une augmentation du nombre d'attaques sur les systèmes. Ceux-ci se montrent en effet accessibles par de nombreux moyens, et peuvent être plus ou moins exposés en fonction des données qu'ils transportent (*e.g.* smartphones). Les attaques sur les systèmes mobiles peuvent se diviser en six catégories principales [58], dont toutes ne sont pas spécifiques à l'environnement. Cette sous-section introduit chacune d'entre elles.

Attaques sur le médium sans-fil

Les systèmes mobiles transmettent et reçoivent généralement des données par l'usage de canaux hertziens. Ces derniers utilisent l'air comme support de propagation, et peuvent donc être écoutés par n'importe quelle antenne opérant sur la bande de fréquence adéquate. De nombreux types d'attaques exploitent cette faiblesse intrinsèque [48], en particulier pour le vol de données sensibles, telles que les identifiants ou mots de passe utilisateurs. Des techniques de spoofing sur les identifiants matériels (*e.g.* adresse MAC) peuvent également permettre à un attaquant d'usurper l'identité d'une station, et ainsi d'intercepter, modifier et redistribuer les communications à son gré. Enfin, la saturation des canaux par le biais d'interférences peut aussi être un cas d'attaque sur le médium sans-fil [72].

Attaques par exploitation locale

Les attaques par exploitation locale sont essentiellement utilisées pour élever les privilèges d'un intrus ayant déjà un point d'entrée sur le système cible, ou d'un des programmes s'exécutant sur ce dernier [64]. Des techniques bas-niveau (*e.g.* débordement de tampons ou attaques de type *string format*) y sont notamment employées pour corrompre la mémoire sur la machine afin d'y injecter et exécuter du code malicieux. Ces attaques sont généralement réalisées de façon ponctuelle, et ne constituent qu'une étape intermédiaire dans l'exploitation d'un système. En effet, elles permettent surtout à l'attaquant d'obtenir l'accès et les droits nécessaires à l'installation de programmes malveillants (*e.g.* porte dérobée, virus) sur la machine victime. Une attaque de ce type réussie offre ainsi un moyen efficace d'obtenir un contrôle distant et continu sur le système infecté.

Attaques sur l'infrastructure réseau

Les services principaux (émission et réception d'appels, de SMS, web, autres) offerts par les terminaux mobiles actuels se basent tous sur une infrastructure sous-jacente où relais-antennes, switches, routeurs, serveurs et autres composants forment l'ensemble du réseau mobile. Des dénis de service aux attaques par canaux auxiliaires, les conséquences d'attaques sur de telles infrastructures peuvent s'avérer lourdes et de large portée, comme le montrent de nombreux cas de mise en pratique au cours de ces dernières années [36],[67]. Du fait de leur intégration dans les réseaux IP, qui s'avèrent relativement vulnérables, les infrastructures GPRS et UMTS sont les plus enclines à pâtir de ce type d'attaque [42]. Le déploiement en cours de la 4G laisse présager les mêmes déconvenues pour le standard LTE Advanced [52].

Attaques orientées vers

Les systèmes mobiles, et plus particulièrement les smartphones, sont couramment équipés de plusieurs interfaces réseaux, proposant de multiples connexions et par conséquent de nombreuses routes pour être accessible. Les vers sont des programmes malicieux capables d'attaquer et se répandre de système en système en exploitant la pluralité de ces connexions. Ils peuvent par ailleurs épuiser les ressources des systèmes de par leur activité. La première infection d'un système par un ver peut être réalisée par le biais de nombreuses attaques indirectes, comme le téléchargement de fichiers malicieux depuis le web, ou via une synchronisation avec une station corrompue par exemple. Une fois sur un système, les vers spécifiques à l'environnement mobile utilisent notamment le Bluetooth (et dans une moindre mesure le RFID [61] et le NFC [50]) ou les SMS/MMS pour se répandre [32]. Ces méthodes rendent les vers qui les utilisent particulièrement virulents, du fait que ces derniers peuvent se dupliquer sur les systèmes sans même bénéficier d'une connexion à l'Internet, par l'intermédiaire d'attaques qualifiées de proximité [60].

Attaques orientées botnet

Durant les premières générations de réseaux mobiles, les terminaux se montraient relativement isolés de l'Internet. Par conséquent, peu de mécanismes de sécurité ont été élaborés pour les protéger des attaquants désireux de créer des réseaux de machines zombies, ou botnets. Les menaces traditionnelles de l'Internet sont maintenant susceptibles de concerner les systèmes mobiles, du fait que ceux-ci disposent de moyens de connectivité variés. En l'occurrence, les botnets de systèmes mobiles peuvent être construits et contrôlés (Command & Control) par SMS, Bluetooth, mais aussi via HTTP, P2P, ou autres protocoles applicatifs qui ont fait leurs preuves dans l'Internet [37]. Des cas concrets de botnets dans l'environnement mobiles [51] ont par ailleurs prouvé que la combinaison de ces techniques (*e.g.* HTTP-SMS/P2P) entraînent une grande puissance de contrôle côté attaquant. À l'image des vers, l'activité engendrée par un botnet côté client peut être préjudiciable en termes de ressources consommées.

Attaques par ingénierie sociale

Cette catégorie rassemble toutes les attaques non techniques qui permettent toutefois d'exploiter les systèmes mobiles. Mettre à disposition un panel de fonctionnalités sécuritaires sur les terminaux ne les protège cependant pas totalement. À de nombreuses reprises sur l'Internet, l'usage d'ingénierie sociale a prouvé que le maillon faible de la sécurité d'une station reste très souvent son utilisateur [44]. Cette vérité s'applique également aux réseaux et systèmes mobiles. La plupart des malwares actuels n'utilisent pas d'exploit pour se répandre, mais préfèrent abuser de la confiance de l'utilisateur. Celui-ci, par inattention ou incompréhension, peut par exemple modifier les mécanismes de sécurité présents sur son terminal afin qu'un programme malveillant puisse s'installer et fonctionner correctement à ses dépends.

2.1.3 Android, un OS pour smartphones comme cas d'étude

Android est un système d'exploitation libre pour smartphones, tablettes tactiles, PDA et autres terminaux mobiles. Il bénéficie de son aspect open-source et d'une communauté importante de développeurs pour étendre les fonctionnalités des terminaux par l'intermédiaire d'applications. Ainsi, Android est la plateforme numéro un pour les smartphones et tablettes dans le monde depuis 2010. Les prévisions [20] [21] pour les années à venir confirment cette domination mondiale malgré la concurrence d'Apple (IOS) et l'arrivée de Windows (Windows Phone) sur le marché. Pour ces raisons, cette plateforme a été choisie comme support d'étude dans ce travail. Son architecture générale et ses problèmes de sécurité sont abordés dans cette sous-section.

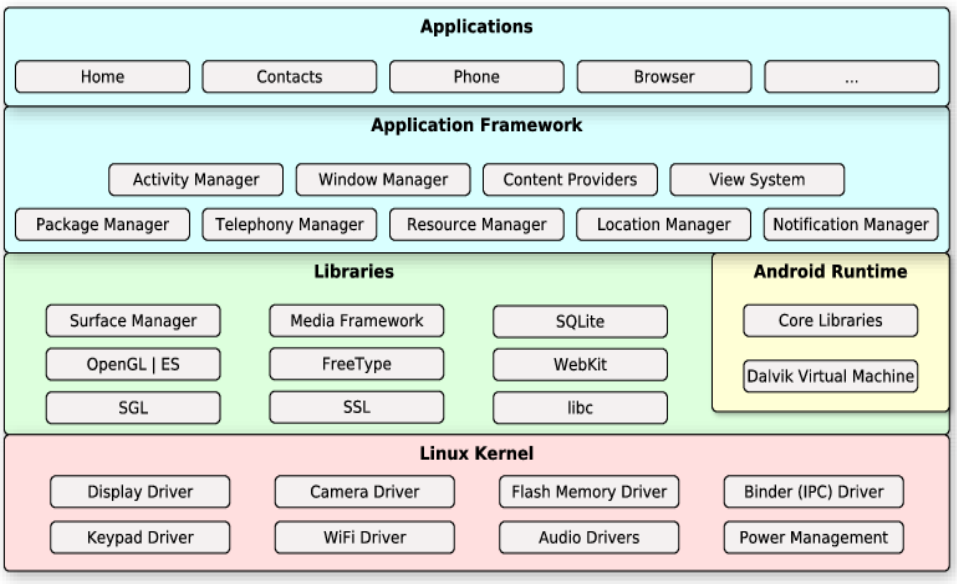

FIGURE 2.1 – Architecture de l'OS Android [3]

Architecture générale

Android se base sur le noyau Linux (version 2.6 ou 3.x depuis la version 4.0 '*Ice Cream Sandwich*') et utilise un ensemble de middlewares et librairies écrits en langage C/C++ dans les couches basses de son architecture, présentée par la figure 2.1. Les couches supérieures sont écrites en Java et permettent essentiellement de fournir un framework modulable pour le développement et l'exécution d'applications utilisateurs, développées en Java également. Android se caractérise notamment par l'utilisation automatique de la machine virtuelle Dalvik [8] pour exécuter les applications utilisateurs. Cette machine se rapproche de la VM Java standard, utilisée dans la plupart des plateformes telles que Linux, Mac OS ou Windows. Elle présente toutefois d'importantes différences (structure des registres, jeu d'instructions, mécanisme de référencements..) afin de s'adapter au mieux à la plateforme mobile. Pour exécuter une application, la VM Dalvik traduit d'abord le bytecode Java correspondant (fichiers .class) et le retranscrit dans dans son propre format (Dalvik Executable format, fichiers .dex), générant des exécutables avec du code réduit et optimisé pour l'architecture matérielle des terminaux (ARM généralement, ou x86, MIPS et I.MX).

Augmentation spectaculaire du nombre de malwares

La plateforme Android souffre de nombreuses vulnérabilités [5] [43] qui peuvent être exploitées par la majorité des types d'attaques présentés en 2.1.2. Entre autres, des bogues d'implémentation (*e.g.* possibilité d'un User ID commun à plusieurs applications, engendrant des fuites dans les IPC [14]) et des failles inhérentes au système (*e.g.* débordement mémoire dans la librairie C *libsysutils.so*) engendrent l'exposition des terminaux et de leurs utilisateurs à des programmes malveillants [65]. Cette exposition est de plus accentuée par la lenteur du processus de patching spécifique à la plateforme, en raison notamment des différentes architectures matérielles utilisées par les terminaux.

Le Google Play Store et les autres marchés alternatifs constituent un important vecteur d'infections, car les mécanismes de filtrage qu'ils mettent en place pour réguler l'ensemble des applications tierces proposées en téléchargement sont bien trop laxistes [1] [41], si ce n'est inexistant. Un système de permissions est instauré pour exprimer de façon explicite les droits nécessaires à chaque application disponible sur le marché, telles que l'accès au réseau, au carnet de contacts, au stockage interne ou externe, ou encore la permission d'émettre des appels et d'envoyer des messages. Néanmoins, au moment de télécharger une application, il est rare qu'un utilisateur prenne garde de ces informations. Le nombre de programmes malveillants dissimulés dans des applications légitimes a ainsi rapidement augmenté durant ces dernières années [38] [74], comme le montre la figure 2.2. Des simples malwares aux botnets les plus complexes [57], ces programmes profitent des vulnérabilités de la plateforme et des permissions qui leurs ont été accordées pour infecter les terminaux avec des charges utiles visant principalement la numérotation vers des numéros surtaxés, le spamming et les vols de données personnelles [65] [68]. Il est nécessaire et urgent de déployer des mesures de sécurité sur la plateforme pour contrecarrer ces tendances.

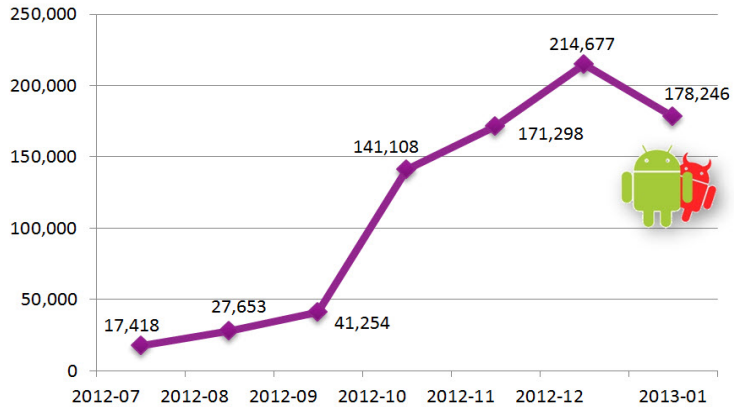

FIGURE 2.2 – Évolution des malwares Android au cours des derniers mois [25]

14. Inter-Process Communication

2.2 Sécurité appliquée aux plateformes mobiles

La sécurité informatique comprend un nombre important de composantes afin de garantir la sécurité des machines et des systèmes d'information. Le travail décrit par ce document se porte essentiellement sur la détection de vulnérabilités sur les systèmes informatiques. La détection de vulnérabilités se présente comme une étape intermédiaire dans le processus complet de gestion des vulnérabilités, dont le but primaire est de prévenir les attaques sur les systèmes en corrigeant les faiblesses de ces derniers. Des mécanismes additionnels pour la détection ou la prévention d'attaques (IDS/IPS) peuvent également être incorporés de façon complémentaire dans cette optique.

2.2.1 Gestion de vulnérabilités

La gestion de vulnérabilités permet de limiter l'exposition des systèmes aux différentes attaques exploitant leurs faiblesses. Une vulnérabilité est une faille sur un système permettant à un attaquant de porter atteinte à l'intégrité de celui-ci (*e.g.* pénétration du système, exécution de code). Elle peut être la conséquence directe d'erreurs ou d'approximations dans la conception ou l'utilisation d'un composant matériel ou logiciel sur le système. Ce travail se limite aux vulnérabilités de types logicielles, qui sont les plus répandues et les plus exploitées actuellement.

Cycle de vie d'une vulnérabilité

Depuis son apparition jusqu'à sa correction effective, le cycle de vie d'une vulnérabilité peut se décomposer en six étapes représentées par le diagramme 2.3. Après la découverte (*discovery)* et l'automatisation de son exploitation (*exploit*), la vulnérabilité est publiée ouvertement (*disclosure*). Des mesures d'atténuations ou de contournement sont ensuite proposées (*countermeasures*) afin de limiter son impact, en attendant la parution d'un correctif intégral (*patch available*). Au final, le laps de temps écoulé entre la sortie de l'exploit et l'application du correctif - qui peut survenir bien après sa parution - est généralement long, de l'ordre de plusieurs mois, voire plusieurs années. Cette exposition offre une opportunité durable aux attaquants et pirates pour abuser des systèmes vulnérables. La portée de ce travail se limite à la détection de vulnérabilités, *i.e.* l'analyse d'un système pour déterminer la présence ou l'absence de vulnérabilités. Le processus de détection d'une vulnérabilité intervient typiquement après le moment où celle-ci est découverte (*discovery*) et publiquement connue (*disclosure*).

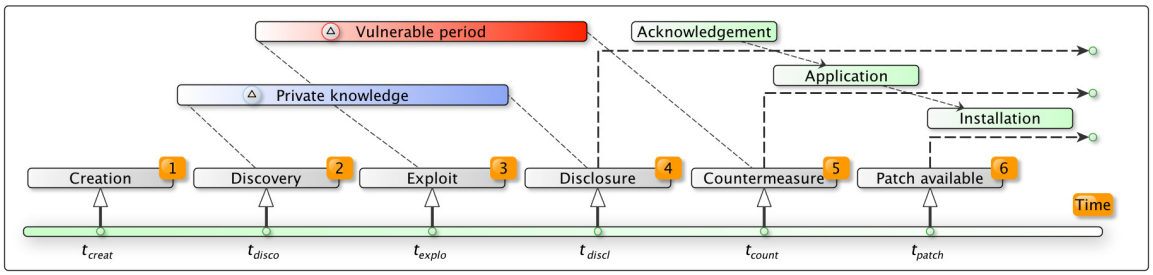

FIGURE 2.3 – Cycle de vie d'une vulnérabilité [28]

Approches de découverte de vulnérabilités

Durant la période de création d'un logiciel, il est fort probable que des erreurs de programmation soient créées involontairement par les développeurs travaillant sur le projet. La découverte de vulnérabilités sur un logiciel consiste à analyser le comportement de ce dernier afin d'en extraire les bogues relatifs à ces erreurs, et leurs éventuelles exploitations. Cette découverte peut être conduite par l'intermédiaire de deux types d'analyse notamment [46] :

- **analyse statique** : une analyse de ce type consiste à analyser le code (source ou binaire) d'un programme sans l'exécuter pour autant, l'objectif primaire étant d'étudier son comportement, et dans une moinde mesure d'y découvrir des failles. Des analyses statiques de teintes (*i.e.* analyses sur la portée des données utilisateur) peuvent être employées comme en [47], tout comme

des techniques de model-checking (*i.e.* vérification automatique des systèmes à états finis) ou des analyses par interprétation abstraite (*i.e.* approximation du comportement du système). Les analyses statiques sont coûteuse en termes de complexité, et découlent fréquemment sur des problèmes indécidables (NP-hard).
- **analyse dynamique** : l'analyse dynamique d'un programme consiste à étudier son exécution dans l'optique de détecter des comportements anormaux. Des techniques de fuzzing comme en [33] ou d'analyses de traces d'exécution comme en [40] peuvent notamment y être utilisées. Le principe du fuzzing est de passer des données mal formées en entrées du programme pour essayer de faire boguer celui-ci. Bien que moins complexe qu'une analyse statique, une analyse dynamique induit des coups d'exécutions non négligeables, et il est difficile de parcourir l'ensemble des chemins d'exécutions possibles. L'idée est donc de déterminer quelles sont les données entrantes à tester en priorité.

Approches de détection de vulnérabilités

Une fois découverte et publiquement connue, une vulnérabilité sur un système peut être corrigée par l'application de patch spécifique, où contournée temporairement par la mise en place de contre-mesures. Néanmoins, cela requiert dans un premier temps de déterminer si le système est vulnérable ou non. Dans cette optique, il est nécessaire de savoir comment l'analyser pour détecter la présence de la vulnérabilité. En fonction du degré de connaissance obtenue concernant l'état (*e.g.* configuration, programmes installés) d'un système cible, plusieurs approches peuvent être utilisées pour l'analyser [63], à savoir (1) l'approche par boîte noire (utilisée notamment par Nessus et Nmap, et dans [34]) si la connaissance est nulle, (2) l'approche par boîte blanche à la manière de [53] si la connaissance est totale, et (3) l'approche par boîte grise si la connaissance est partielle. Nous utilisons dans ce travail une approche par boîte grise, où la connaissance partielle de l'état d'un système est acquise à partir de ses différents fichiers de configuration. Les caractéristiques des vulnérabilités à traiter sont quant à elles connues par l'intermédiaire de leur description. Une description de vulnérabilité renseigne les informations nécessaires à son analyse et sa détection sur un système. L'usage d'un format standard et interprétable par une machine pour rédiger ces descriptions permet d'augmenter grandement leur portée à travers tout le spectre d'outils de sécurité et de détection. Cependant, la majorité des solutions de détection actuelles utilisent des formats qui leur sont propres pour décrire les vulnérabilités, réduisant ainsi les chances de pouvoir échanger ces connaissances.

2.2.2 Le langage de descriptions et d'analyses OVAL

Dans un effort de standardisation, plusieurs langages tels que VulnXML [24] et AVDL [15] [2] ont été conçus pour normaliser la manière de représenter les vulnérabilités. Cependant, ces langages se focalisent principalement sur les applications web, et ne couvrent ainsi qu'un sous-ensemble limité des vulnérabilités existantes. Open Vulnerability and Assessment Language (OVAL), proposé par MITRE corporation [15], est devenu un standard international de sécurité visant à promouvoir la diffusion publique de contenu sécuritaire et à normaliser la façon d'échanger ces informations. OVAL inclut son propre langage pour standardiser les trois étapes principales du processus d'analyse, à savoir (1) représenter l'état et la configuration du système à analyser, (2) analyser ce système pour détecter la présence d'états spécifiques (vulnérabilité, configuration, patch, etc.), et (3) retranscrire le résultat de l'analyse.

Structure du langage OVAL

Les membres de la communauté OVAL ont développé trois schémas en XML pour définir le vocabulaire et la structure du langage. Ces schémas correspondent directement aux trois étapes du processus d'analyse :

- Le schéma *OVAL System Characteristics* définit un format XML standard pour représenter les informations de configuration et d'état d'un système. Ces informations comprennent entre autres les paramètres du système d'exploitation et des programmes installées, mais aussi les

15. Application Vulnerability Description Language

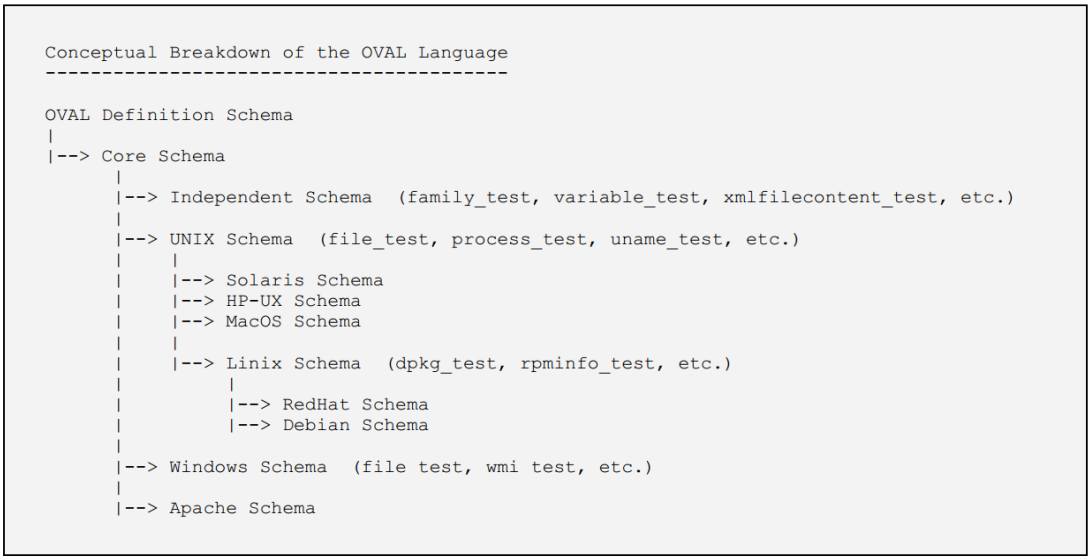

FIGURE 2.4 – Schémas noyau et auxiliaires *OVAL Definitions* [23]

propriétés de l'état courant comme les ports ouverts ou les processus en cours d'exécution. Ce schéma fournit l'ensemble des caractéristiques et propriétés devant être comparées à une définition OVAL (état spécifique) durant l'analyse d'un système. Cette standardisation du format permet à de nombreux outils de sécurité d'avoir une base commune pour l'échange d'information sur les états des systèmes.

- Le schéma *OVAL Definition* est utilisé pour écrire en XML (1) une définition de vulnérabilité décrivant les conditions qui doivent exister sur un système pour qu'une vulnérabilité spécifique soit présente, (2) une définition de correctif permettant de déterminer si un patch spécifique doit être appliqué ou non sur un système, et (3) une définition de conformité indiquant les conditions devant être respectées par un système pour que celui-ci soit conforme à une politique de sécurité donnée ou une configuration requise. Une définition OVAL contient un ensemble de tests OVAL permettant d'indiquer la façon de vérifier les propriétés spécifiques sur les systèmes, pour déterminer s'ils respectent ou non les conditions requises exprimées dans la définition.

- Le schéma *OVAL Result* donne un format XML standard pour représenter les résultats d'analyse d'un système. Les données retranscrites contiennent notamment les résultats de comparaison de l'état courant du système à une ou plusieurs définitions OVAL. Le schéma *OVAL Result* permet à des applications auxiliaires d'interpréter les informations rapportées pour déterminer les actions nécessaires à appliquer sur un système, comme la correction d'une vulnérabilité par l'application d'un correctif spécifique, ou la modification de certains paramètres de configuration pour être conforme à une politique requise.

Ces trois schémas fournissent la totalité du vocabulaire du langage OVAL. Chacun d'entre eux se compose d'un schéma noyau et de plusieurs schémas auxiliaires qui viennent étendre ce dernier. Les schémas noyaux permettent de spécifier la syntaxe et la structure générale attendue pour chaque type de document. Les schémas auxiliaires adaptent et étendent ces information pour chaque plateforme supportée par le standard. La figure 2.4 montre la hiérarchie des schémas compris dans le schéma *OVAL Definitions*. Le schéma noyau donne la structure générale d'une définition OVAL et des métadonnées (*e.g.* identifiants CVE [16], plateformes affectées) pouvant s'y intégrer, alors que les schémas auxiliaires définissent les tests spécifiques à chaque plateforme pour pouvoir utiliser ces définitions.

Fonctionnement du langage OVAL

Le diagramme donné par la figure 2.5 montre toutes les étapes du processus d'analyse d'un système dont celles impliquant les trois schémas OVAL. Ce processus démarre avec la diffusion

16. Common Vulnerabilities and Exposures

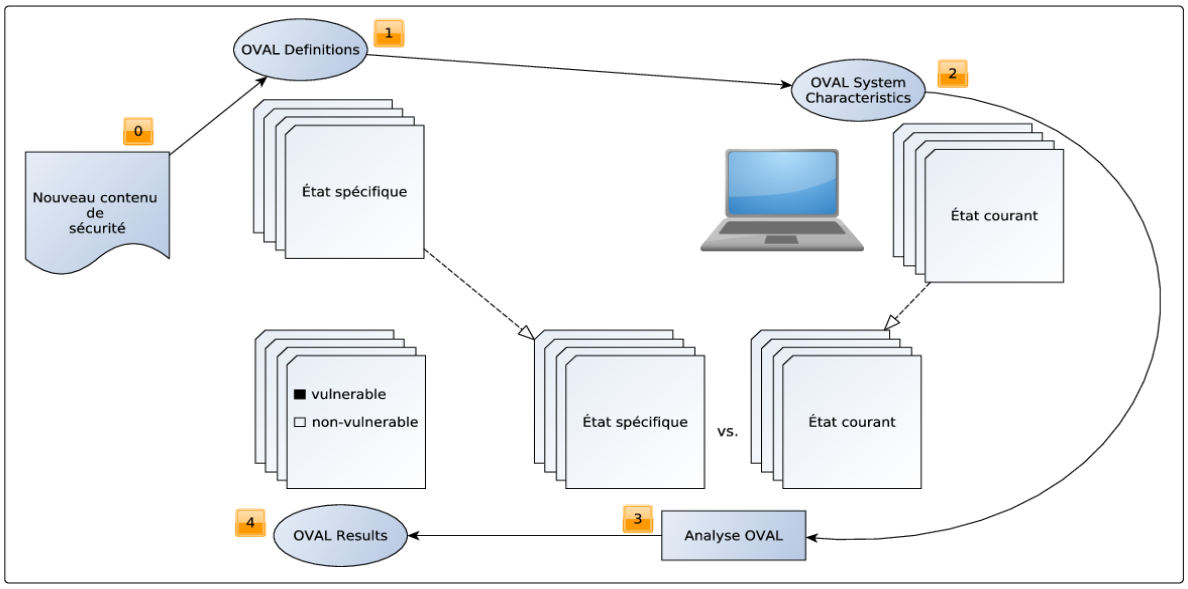

FIGURE 2.5 – Cycle de fonctionnement du langage OVAL [23]

publique de nouveau contenu sécuritaire (étape 0), comme la publication de descriptions de nouvelles vulnérabilités par un vendeur, ou de nouvelles configurations requises par une entreprise tierce par exemple. Les états spécifiques qui y sont décrits sont encodés et retranscrits en définitions OVAL (étape 1). Ces définitions sont structurées de sorte à pouvoir indiquer quelles propriétés doivent être controlées sur l'état des systèmes à analyser. Ainsi, les propriétés concernées sont dans un premier temps collectées sur le système cible, et regroupées dans un ou plusieurs documents *OVAL System Characteristics* (étape 2). Après la collecte de données, l'analyse est réalisée (étape 3) : elle consiste à comparer le ou les fichiers *OVAL System Characteristics* (état du système analysé) aux définitions OVAL (états attendus) pour savoir si le système est vulnérable, ou si sa configuration satisfait les critères requis. Des fichiers *OVAL Results* sont générés à la fin de l'analyse (étape 4), rapportant les résultats de ces dernières. Les étapes 2, 3 et 4 sont typiquement réalisées par un interpréteur OVAL tel qu'Ovaldi [14] ou XOvaldi [29], capable de déterminer les actions à réaliser en fonction des documents qui lui sont passés en paramètres, et de générer les fichiers correspondants (documents *OVAL System Characteristics* ou *OVAL Results*).

Exemple de définition

Afin de comprendre le langage, un exemple de définition OVAL est donné par la figure 2.6. Pour des raisons d'affichage, ce document est incomplet mais conserve cependant tous les éléments pertinents. La première partie du document spécifie les différents schémas XML utilisés, et les méta-informations associées à la génération du document. Une document *OVAL Definition* peut contenir une ou plusieurs définitions (balise *definitions*), chacune disposant d'un identifiant unique. La seule définition (balise *definition*) de ce document décrit une vulnérabilité de type débordement de mémoire sur les systèmes Android. Deux conditions représentées par deux tests OVAL (balises *criterion*) doivent être vérifiées sur l'état de la machine cible. L'attribut *operator* (*AND*) de la balise *criteria* indique que ces deux conditions doivent être respectées pour que le système soit vulnérable.

Tous les tests OVAL (balise *tests*) ont un identifiant unique, et se composent obligatoirement d'un objet et d'un état OVAL. Le couple objet-état forme l'empreinte et l'unicité d'un test. Le premier test (balise *sys_details_test*) consiste à sonder la version d'Android sur le terminal, alors que le second (balise *file_test*) teste la présence d'une librairie. Ces informations sont données par les objets (balise *objects*) qui leur sont associés. Les deux états correspondant (balise *states*) indiquent respectivement les versions d'Android attendues (balise *os_version_number*, expression régulière représentant les versions 2.2 à 2.2.2 ou 2.3 à 2.3.6), et le chemin et le nom (balises *path* et *filename*) de la librairie requise.

```
<?xml version="1.0" encoding="UTF-8"?>
<oval_definitions xsi:schemaLocation= [...]
 xmlns=[...]
 xmlns:oval=[...]
 xmlns:oval-def=[...]>
 <generator>
  <oval:product_name>[...]</oval:product_name>
  <oval:schema_version>[...]</oval:schema_version>
  <oval:timestamp>[...]</oval:timestamp>
 </generator>

 <definitions>
  <definition id="oval:fr.inria.madynes:def:1"
      class="vulnerability">
   <metadata>
        <title>Libsysutils stack overflow</title>
        <affected family="android"/>
        <reference ref_id="" source=""/>
        <description>[...]</description>
   </metadata>
   <criteria operator="AND">
    <criterion test_ref="oval:inria.madynes:tst:1"/>
    <criterion test_ref="oval:inria.madynes:tst:2"/>
   </criteria>
  </definition>
 </definitions>

 <tests>
  <sys_details_test id="oval:inria.madynes:tst:1">
   <object object_ref="oval:fr.inria.madynes:obj:1"/>
   <state state_ref="oval:fr.inria.madynes:ste:1"/>
  </sys_details_test>
  <file_test id="oval:fr.inria.madynes:tst:2">
   <object object_ref="oval:fr.inria.madynes:obj:2"/>
   <state state_ref="oval:fr.inria.madynes:ste:2"/>
  </file_test>
 </tests>

 <objects>
  <sys_details_object id="oval:inria.madynes:obj:1">
   <os_version_number></os_version_number>
  </sys_details_object>
  <file_object id="oval:inria.madynes:obj:2">
   <path operation="equals">/system/lib/</path>
   <filename operation="equals">
    libsysutils.so
   </filename>
  </file_object>
 </objects>

 <states>
  <sys_details_state id="oval:inria.madynes:ste:1">
   <os_version_number operation="pattern_match">
    2\.2|2\.2\.[1-2]|2\.3|2\.3\.[1-6]
   </os_version_number>
  </sys_details_state>
  <file_state id="oval:inria.madynes:ste:2">
   <path operation="equals">/system/lib/</path>
   <filename operation="equals">
    libsysutils.so
   </filename>
  </file_state>
 </states>
</oval_definitions>
```

FIGURE 2.6 – Exemple de définition OVAL

Standards et langages complémentaires

OVAL tire en grande partie sa connaissance du dictionnaire CVE [4], une base publique de descriptions de vulnérabilités maintenue également par MITRE corporation. Un billet CVE est une entrée du dictionnaire dans laquelle sont résumées les informations systèmes caractéristiques d'une machine exposée à une vulnérabilité donnée. En traduisant ces billets CVE en définitions OVAL, les membres de la communauté alimentent une vaste base de définitions, offrant l'opportunité d'effectuer des analyses spécifiques en fonction des vulnérabilités choisies et de la plateforme du système. La communauté met elle-même à disposition du public cette base de définitions sur son site officiel ("The OVAL repository" [18]), et de nombreuses autres organisations tierces ont suivi cette démarche. OVAL fait par ailleurs partie intégrante du protocole SCAP [17] [26], introduit par le NIST [18] [17] afin de fournir des méthodes standards pour la gestion et la correction des vulnérabilités de manière autonome. Le protocole SCAP inclut d'autres standards ouverts tels que XCCDF [19] [75] et CVSS [20] [6]. XCCDF est un langage conçu pour la correction automatique des vulnérabilités ou défauts de configuration sur les systèmes, alors que CVSS fournit un moyen normalisé d'affecter un score à chaque vulnérabilité connue en fonction de sa dangerosité et de son impact.

2.2.3 Limites des travaux actuels

L'attention des pirates informatiques s'est naturellement portée sur la plateforme Android à mesure que celle-ci devenait populaire. Aujourd'hui leader mondiale sur le marché des smartphones et tablettes, elle affiche de nombreuses limites de sécurité qui entraînent l'exposition de ses utilisateurs aux menaces de l'Internet ou autres. La sécurisation de la plateforme représente un point critique qui demeure au centre de ses enjeux.

Mécanismes de sécurité natifs

Le système Android inclut de base un nombre conséquent de mécanismes de sécurité afin de protéger les données utilisateur et système. Il reprend par exemple les fonctionnalités les plus robustes du noyaux Linux, telles que son modèle de droits et de permissions pour définir une séparation stricte des privilèges en fonction de l'utilisateur. Les processus en mémoire sont isolés les uns des

17. Security Content Automation Protocol
18. National Institute of Standards and Technology
19. eXtensible Configuration Checklist Description Format
20. Common Vulnerability Scoring System

autres et les mécanismes de communications qui leur sont propres (IPC) bénéficient d'une sécurité accrue. Android tire profit d'une telle approche pour forcer chaque application à s'exécuter dans une sandbox, en leur attribuant à chacune un processus dédié et un identifiant utilisateur unique (User ID ou UID) avec des droits restreints. Des mécanismes bas-niveau additionels sont par ailleurs incorporés à la plateforme, en parti pour prévenir au mieux les corruptions de mémoire (protection de la pile avec SSP [21], ASLR [22] à partir de la version 4.0, autres). La couche applicative propose principalement des utilitaires de chiffrement (fichiers et flux réseau), des mécanismes de protection par mot de passe et des outils de sauvegarde sécurisée pour les couples d'identifiants. Un inventaire intégral de la sécurité dans Android est disponible dans sa documentation officielle [16] et en [35]. En dépit de cet éventail de fonctionnalités, les mécanismes de sécurité natifs dans Android ne permettent pas de protéger totalement les terminaux des attaques et programmes malveillants.

Travaux de recherche et outils actuels

Afin de limiter l'impact de l'augmentation de malwares exploitant les faiblesses de la plateforme (sous-section 2.1.3), plusieurs travaux ont été réalisés sur l'analyse et la classification des applications Android disponibles sur les différents marchés [39] [45]. D'autres se sont portés sur l'élaboration de solutions pour la détection et le filtrage des programmes malicieux qui y sont injectés [66], [69] et [71]. Leurs contributions peuvent augmenter de façon significative la sécurité des terminaux Android. Cependant, aucun moyen effectif de détecter les vulnérabilités ou nettoyer les infections sur des systèmes n'est proposé. De nombreux logiciels sont disponibles pour renforcer la sécurité intrinsèque des systèmes Android, tels que des antivirus (Lookout, Norton), antirootkits (Sophos) et autres scanners de vulnérabilités (X-Ray, eEye). Ces solutions sont généralement privées et ne fournissent pas de moyens standards pour augmenter ou échanger leurs connaissances. L'usage de bases de connaissances publiques et du langage OVAL a déjà fait l'objet d'études sur l'analyse de vulnérabilités dans des réseaux à grande échelle [56] [27]. Actuellement, le support d'Android par le standard est encore au stade expérimental [13], et très peu de recherches ont été effectuées concernant les mécanismes de détection de vulnérabilité pour cette plateforme. Dans notre travail précédent [30], nous avions proposé une solution basée sur OVAL pour standardiser la détection de vulnérabilités sur les systèmes Android. Bien que concluante, cette approche engendrait une charge système non négligeable sur les dispositifs durant les analyses.

Sécurité mobile à faibles charges

Du fait de leur aspect embarqué, les systèmes Android fonctionnent généralement sur batterie et disposent de ressources limitées. La gestion de ces dernières et de leur consommation d'énergie demeure une composante critique qui doit être négociée de façon efficace. Dans le contexte de ce travail, le déport de l'analyse sur des serveurs distants peut être un moyen d'atteindre cet objectif en allégeant la charge sur les stations mobiles. L'utilisation du cloud pour les plateformes mobiles a déja été expérimentée à plusieurs reprises, notamment pour l'externalisation de tâches [70],[54] et pour la mise en place de services distants [62],[55]. Les résultats de ces travaux laissent à penser qu'une telle stratégie peut s'avérer prometteuse, en particulier pour allonger la durée d'autonomie des terminaux. Elle favorise également le déploiement de services transparents, qui étendent les fonctionnalités des systèmes sans pour autant perturber leur comportement. En outre, un tel modèle peut permettre d'instaurer une centralisation à différents degrés des données et de l'intelligence nécessaire au service proposé [73]. Appliqué à la détection de vulnérabilités, il peut faciliter l'ordonnancement des analyses et l'exploitation de leurs résultats. L'établissement d'historiques ou l'extraction de statistiques sur l'évolution des terminaux et de leurs vulnérabilités peuvent être des exemples d'applications directes.

Dans ce contexte, notre nouvelle approche [31] vise à réduire les ressources affectées par le processus de détection tout en garantissant une précision suffisante à travers le temps, par le biais d'algorithmes probabilistes pour gouverner les analyses des systèmes, et d'un modèle orienté cloud pour le déport de celles-ci.

21. Stack-Smashing Protector
22. Address Space Layout Randomization

Chapitre 3

Détection probabiliste de vulnérabilités

Sommaire

Dans ce chapitre, nous décrivons l'approche utilisée pour réaliser les analyses sur les systèmes cibles en vue de détecter les vulnérabilités qu'ils présentent. Le modèle probabiliste élaboré pour cette approche est détaillé dans la première section. La seconde section présente l'architecture de notre solution exploitant ce modèle, ainsi que la stratégie mise en place pour externaliser les analyses.

3.1 Modélisation mathématique

Les ressources limitées des systèmes mobiles sont une caractéristique devant être prise en compte au moment de développer des applications ou services qui leur sont destinés. Dans ce contexte, le modèle proposé dans cette section vise à minimiser la consommation de ressources (*e.g.* batterie, taux processeur, mémoire) sur les systèmes tout en maximisant la précision de leurs analyses. Il utilise pour cela une stratégie probabiliste afin de distribuer les analyses efficacement à travers le temps.

3.1.1 Principes du modèle

Cette sous-section donne une vue d'ensemble du modèle proposé. Après avoir spécifié la manière de représenter un test et une vulnérabilité, nous comparons brièvement notre approche à une approche traditionnelle concernant la façon de les analyser. Les mécanismes de sélection de tests à travers le temps sur les systèmes cibles sont également abordés, tout comme les moyens de paramétrisation des analyses.

Tests et vulnérabilités

Une vulnérabilité peut être décrite comme une combinaison logique de propriétés système (*e.g.* la version de l'OS est égale à X ET la librairie Y est installée). Dans ce travail, nous utilisons les définitions OVAL comme support pour décrire les vulnérabilités. Chaque définition combine logiquement un ensemble de tests OVAL (*e.g.* $v = t_1 \wedge t_2$) qui vont permettre de vérifier l'état du système cible. Un test OVAL peut être utilisé dans plusieurs définitions, et se compose d'un objet et d'un état OVAL. Un objet OVAL permet de représenter la propriété à analyser sur le système, alors

qu'un état OVAL exprime l'état attendu pour cette propriété. Évaluer un test consiste à comparer l'état de la propriété collectée sur le système (représenté par l'objet OVAL) à l'état attendu par ce test (représenté par l'état OVAL). Le résultat d'une évaluation est un booléen à $vrai$ si les deux états sont semblables, $faux$ sinon. Analyser une vulnérabilité sur un système consiste à évaluer ses tests pour résoudre la combinaison logique qui la caractérise dans sa définition. Si cette combinaison est vérifiée par l'état du système sondé, alors celui-ci est vulnérable.

Soit $T = \{t_1, t_2, ..., t_n\}$ l'ensemble des tests OVAL possibles. L'ensemble V des vulnérabilités connues peut alors être construit de la manière suivante :

i) si $t_i \in T$, alors $t_i \in V$

ii) si $\alpha, \beta \in V$, alors $(\alpha \diamond \beta) \in V$ avec $\diamond \in \{\wedge, \vee\}$

iii) si $\alpha \in V$, alors $(\neg\alpha) \in V$

Approche traditionnelle et probabiliste

Les mécanismes traditionnels d'analyse d'un système traitent généralement toutes les vulnérabilités de V au cours d'une séance de détection. Une telle stratégie est certes précise, mais se révèle coûteuse pour les systèmes cibles en matière de charge de travail et de consommation énergétique. L'approche que nous proposons tire son origine de ce constat, et adopte un modèle probabiliste afin de partager et distribuer à travers le temps l'analyse de V dans sa totalité. L'objectif principal consiste à réduire les charges engendrées par les analyses tout en maintenant ces dernières à un niveau de précision acceptable, tenant compte des propriétés des vulnérabilités à couvrir.

La figure 3.1 compare les deux approches sur l'analyse d'un ensemble de vulnérabilités impliquant huit tests différents à travers quatre séances de détection. La méthode traditionnelle traite à chaque période la totalité des vulnérabilités, évaluant ainsi l'ensemble des tests disponibles. L'approche probabiliste ne sélectionne quant à elle qu'un sous-ensemble de tests à évaluer au cours de chaque période, et ne couvre ainsi qu'un sous-ensemble de vulnérabilités à chaque fois. Les probabilités de sélection des tests sont calculées à partir de (1) leur utilité respective dans le jeu de vulnérabilités à traiter, et (2) le temps écoulé depuis leur dernière évaluation. En suivant cette méthodologie, l'approche probabiliste réduit significativement la charge et l'allocation de ressources sur les systèmes cibles durant les analyses, tout en présentant un temps de convergence satisfaisant pour assurer le traitement de V dans sa totalité.

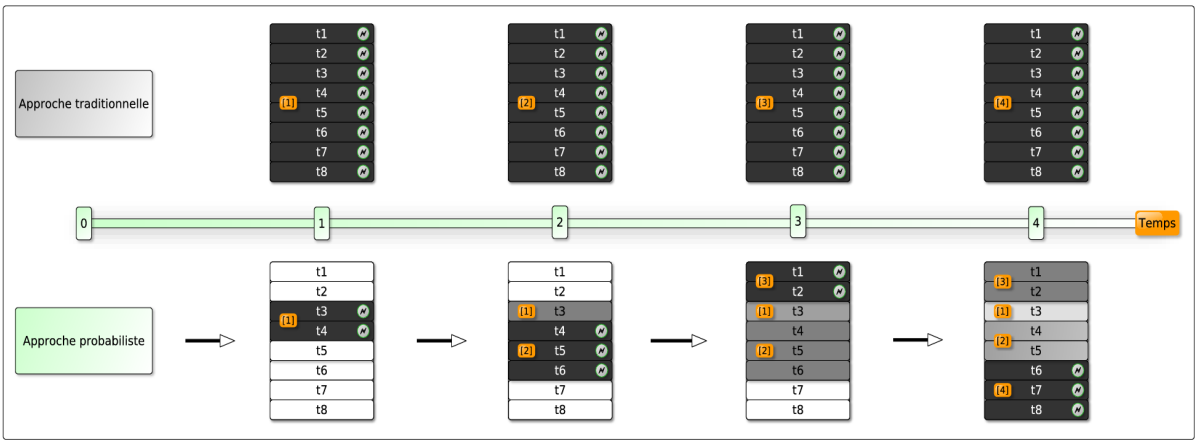

FIGURE 3.1 – Approche traditionnelle vs. approche probabiliste

Sélection de tests et distribution temporelle

Un exemple du processus de sélection de tests est présenté dans la figure 3.1. Quatre analyses sont nécessaires pour traiter la totalité des vulnérabilités comprises dans V. Les tests t_3 et t_4 sont évalués au cours de la première période. Les tests t_5 et t_6 sont évalués durant la seconde période, mais

aussi t_4 à nouveau. Cela peut s'expliquer par l'utilité élevée de ce test dans le jeu de vulnérabilités. Au cours de la troisième période, les tests t_1 et t_2 sont exécutés. L'évaluation de t_3 devient ancienne de deux périodes, celles de t_4, t_5 et t_6 d'une seule. Les tests restant non évalués (t_7,t_8) le sont au cours de la période 4, tout comme t_6. Là encore, cette redondance peut s'expliquer entre autres par l'utilité de t_6 dans V. Cette dernière période permet de compléter l'évaluation des tests de T, et ainsi des vulnérabilités de V.

À mesure que les périodes passent, les probabilités de sélection des tests les plus anciennement évalués (*e.g.* t_3) augmentent. Ces derniers se montrent ainsi compétitifs en termes de sélectivité malgré leur utilité relativement faible dans V par rapport à celle d'autres tests (*e.g.* t_4 et t_6). L'idée principale est de favoriser de façon incrémentale les tests les moins importants pour s'assurer qu'ils seront évalués tôt ou tard. Cette priorisation par le temps d'ancienneté permet de prévenir d'éventuelles famines de tests, des scénarios où certains tests ne sont jamais exécutés du fait de leur faible utilité. La prévention de famine de tests permet de garantir une convergence finie sur l'évaluation de V dans sa totalité.

Paramétrisation du modèle

Le modèle considère deux paramètres principaux qui permettent à l'utilisateur de l'adapter à ses besoins spécifiques. Le premier paramètre λ indique le seuil requis de couverture de V, *i.e.* le pourcentage de vulnérabilités à traiter au cours d'une séance d'analyse. Le second paramètre δ donne le laps de temps (*e.g.* un nombre de périodes) qui séparent deux séances. L'idée derrière cette paramétrisation consiste à évaluer itérativement un nombre limité de tests durant chacune des analyses, effectuées à la fréquence δ. La sélection des tests dans ce processus itératif est en parti influencée par leurs statistiques respectives concernant leur utilité dans V, mais aussi par le temps écoulé depuis leur dernière évaluation.

Au cours d'une itération, les probabilités de sélection des tests encore non évalués sont dans un premier temps calculées. En appliquant ces probabilités, un test est ensuite sélectionné par tirage au sort puis évalué, permettant d'analyser de façon totale ou partielle un sous-ensemble de vulnérabilités. Ce processus se répète jusqu'à ce que le seuil λ de vulnérabilités à traiter soit atteint. La sélection de tests demeure un processus critique pour assurer la réduction de la charge sur les systèmes sondés. Deux composantes clefs doivent y être garanties : la première est de pouvoir sélectionner les tests les plus utiles le plus souvent possible, la seconde est de s'assurer que tout test peut éventuellement être évalué indépendamment de son importance. Les méthodes permettant d'appliquer ces deux concepts sont présentées dans les sous-sections suivantes.

3.1.2 Quantification de l'utilité d'un test

Dans cette sous-section, nous décrivons comment déterminer l'utilité d'un test dans un jeu de vulnérabilités donné. Le concept de réduction de vulnérabilité est abordé dans un premier temps, alors que les méthodes mathématiques pour quantifier l'utilité d'un test sont données dans un second temps. Enfin, nous spécifions le critère statistique choisi pour classer les tests en fonction de leur utilité.

Réduction de vulnérabilité

L'utilité d'un test dans un ensemble V de vulnérabilités est déterminée en fonction de deux facteurs qui sont (1) sa capacité à réduire les vulnérabilités de V quand la valeur du test est connue, et (2) l'importance des vulnérabilités dans lesquelles le test est impliqué. Le concept de réduction de vulnérabilités représente la réduction de leur équation logique respective, en y remplaçant le littéral exprimé par le test par la valeur de son résultat. Plus l'équation logique d'une vulnérabilité est réduite, plus le résultat de son analyse est proche d'être connu. Une vulnérabilité est traitée complètement lorsque son équation est résolue (réduction maximale). Afin de faciliter les réductions de vulnérabilités, les équations logiques qui les représentent sont exprimées sous forme normale conjonctive (FNC). Une vulnérabilité exprimée en FNC est une conjonction de clauses où chaque

clause est une disjonction de tests, représentée de la manière suivante :

$$v_i^{CNF} = \bigwedge(\bigvee(t_j|\neg t_j)) \qquad t_j \in T, v_i \in V \tag{3.1}$$

Soit par exemple v une vulnérabilité décrite par l'équation $v = t_1 \wedge (t_2 \vee t_3)$. Si la valeur de t_1 est connue et évaluée à $faux$, alors l'équation de v est réduite au maximum et donc résolue avec cette seule connaissance. Il n'y a dans ce cas là aucun besoin d'évaluer t_2 ou t_3, car t_1 a éliminé tous les tests de l'équation de v. Si $t_1 = vrai$, alors un ou deux tests supplémentaires au maximum doivent être évalués. Si en revanche, seul t_2 est connu, alors un autre test dans le meilleur des cas (t_1 si t_2 est $vrai$) doit être exécuté, et deux dans le pire des cas (si $t_2 = faux$ et $t_3 = vrai$, t_1 doit être évalué). Ce phénomène reste le même si t_3 est évalué en premier. Par conséquent, l'utilité de t_1 est plus importante que celle de t_2 ou t_3. Après le processus de réduction, l'équation de v sera $v = faux$ si $t_1 = faux$ (analyse de v terminée), ou $v = vrai \wedge (t_2 \vee t_3) = (t_2 \vee t_3)$ si $t_1 = vrai$, auquel cas l'analyse devra continuer sur les tests t_2 voire t_3 afin de pouvoir la résoudre.

Calcul de l'utilité

Afin de calculer l'utilité U d'un test dans V quand sa valeur est connue, nous donnons la formule suivante :

$$U(t, val, V) = \frac{\sum_{i=1}^{|V|}(\frac{testRed(v_i,t,val)\times I(v_i)}{totalTests(v_i)})}{|V|} \qquad t \in T, val \in \{vrai, faux\}, v_i \in V \tag{3.2}$$

La fonction $totalTests$ retourne le nombre de tests formant l'équation de v_i. La fonction I retourne une valeur numérique représentant le score de la vulnérabilité en fonction de sa dangerosité et de son impact (*e.g.* score CVSS). La fonction *testRed* permet de calculer le nombre de tests éliminés dans l'équation en FNC de v_i par la connaissance de t à val. Un test éliminé par t dans v_i est un test dont le résultat est connu (t s'élimine lui-même) ou n'a plus aucune utilité pour la réduction de l'équation de v, car la connaissance qu'il permettrait d'apporter est déjà couverte par l'évaluation de t.

Le fonctionnement de la méthode *testRed* est décrit par l'algorithme 1, où C_v représente l'ensemble des clauses contenues dans l'équation en FNC de v, alors que T_c représente l'ensemble des tests combinés dans la clause c. VA_t^c dénote la valeur attendue de t dans c. Soit par exemple $c = (\neg t_1 \vee t_2)$, alors $VA_{t_1}^c = faux$, et $VA_{t_2}^c = vrai$. Enfin, la fonction $totalTestsC$ retourne le nombre de tests contenus dans la clause c.

Algorithme 1: Fonctionnement de la méthode *testRed*

Input : FNCeq v , Test t, Booléen val
Output : Entier nb_elim_test

```
1  nb_elim_test ← 0;
2  for ∀ c ∈ C_v telle que t ∈ T_c do
3  |  if (totalTestsC(c) = 1 AND val = VA_t^c ) then
4  |  |  nb_elim_test ← nb_elim_test + 1;
5  |  else if (totalTestsC(c) = 1 AND val != VA_t^c ) then
6  |  |  nb_elim_test ← totalTests(v);
7  |  else if (totalTestsC(c) > 1 AND val = VA_t^c ) then
8  |  |  nb_elim_test ← nb_elim_test + totalTestsC(c);
9  |  else
10 |  |  // (totalTestsC(c) > 1 AND val != VA_t^c )
11 |  |  nb_elim_test ← nb_elim_test + 1;
12 |  end
13 |  if (nb_elim_test = totalTests(v)) then break;
14 end
15 return nb_elim_test;
```

Utilité moyenne

La formule U donnée par l'équation 3.2 est employée pour calculer l'utilité d'un test dans V quand le résultat de son évaluation est connu. Cette utilité détermine en partie sa probabilité d'être choisi lors du processus de sélection au cours de l'analyse. Puisque, par définition, le test n'est pas encore évalué au moment du calcul de son utilité, sa valeur n'est donc pas encore connue. Par conséquent, nous définissons une fonction U_M permettant de calculer l'utilité moyenne d'un test dans un ensemble V comme suit :

$$U_M(t, V) = \frac{U(t, vrai, V) + U(t, faux, V)}{2} \qquad t \in T \tag{3.3}$$

Bien que triviale, cette formule permet de classer les tests en fonction de leur importance moyenne, et de construire ainsi une liste ordonnée $L_U = \{t_x, t_k, ..., t_n\}$ utilisée par le processus de sélection durant les analyses. Elle fournit, pour chaque test encore non évalué, les informations statistiques permettant de déterminer en partie sa probabilité d'être choisi. À ces données vient se greffer la composante temporelle (*i.e.* temps écoulé depuis la dernière exécution) complétant l'aspect probabiliste du processus de sélection, détaillé dans la sous-section suivante.

3.1.3 Sélection probabiliste des tests

Le paramètre-seuil α permet de ne traiter qu'un sous-ensemble de vulnérabilités de V durant une séance d'analyse. Tous les tests de T ne sont donc pas évalués au cours d'une même période. En considérant que l'état de la machine sondée soit constant, si les tests sélectionnés sont toujours les plus utiles, alors certains tests peuvent n'être jamais évalués. Ce phénomène est dénommé famine de tests signifiant que certains tests peuvent ne jamais être exécutés à cause de leur trop faible utilité dans V. Une conséquence directe peut être la famine de vulnérabilités, signifiant que certaines vulnérabilités n'auront jamais l'occasion d'être traitées. Afin de prévenir ce problème, nous considérons deux facteurs pour diriger le comportement du processus de sélection de tests.

Probabilité de sélection et temps d'ancienneté d'un test

Le premier facteur est une probabilité de sélection ρ attribuée à chaque test en fonction de son utilité, afin d'instaurer la propriété suivante : même si un test a toujours la plus grande utilité à travers le temps et l'état du système, d'autres tests moins important peuvent quand même être sélectionnés à sa place. Une telle approche se montre moins élitiste, mais offre un compromis intéressant entre efficacité (les tests les plus utiles sont les plus susceptibles d'être évalués) et équité (les tests peu utiles ont toujours l'opportunité d'être sélectionnés). Pour calculer la probabilité ρ d'un test donné, nous définissons la formule 3.4. Les valeurs sont normalisées pour chaque test afin de respecter le caractère probabiliste du facteur.

$$\rho(t, V, L_U) = \frac{U_M(t, V)}{\sum_{i=1}^{|L_U|} U_M(t_i, V)} \qquad t, t_i \in L_U \tag{3.4}$$

Le second facteur utilisé par le processus de sélection est le laps de temps τ déterminé pour chaque test, exprimant le temps écoulé depuis leur dernière évaluation respective. Cette durée est calculée trivialement comme suit (x représente le temps courant) :

$$\tau(t, x) = x - dateDernEval(t) \qquad t \in L_U, x \in [0..\infty] \tag{3.5}$$

Plus le τ d'un test est élevé, plus celui-ci a des chances d'être sélectionné et évalué au cours de l'analyse. Cette priorisation par le temps, couplée au caractère probabiliste de la sélection, permet de garantir la prévention de famine de tests. Le temps τ d'un test doit être corrélé avec sa probabilité ρ, notamment afin de respecter les données statistiques extraites concernant son utilité, comme expliqué dans le paragraphe suivant.

Valeur de sélectivité d'un test

La combinaison des deux facteurs introduits ci-dessus permet de calculer une valeur de sélectivité (ou probabilité finale de sélection) pour chaque test compris dans L_U, *i.e.* les tests non évalués. Nous définissons cette valeur pour un test t à un temps x donné grâce à la formule 3.6. L'idée principale de cette formule est de prioriser les tests bénéficiant d'une utilité élevée, représentée par leur probabilité ρ, en favorisant simultanément les tests moins importants à mesure que leur delta de temps τ s'accroît. Le processus de sélection de test se base sur les valeurs de sélectivité pour élire par tirage au sort le prochain test à réaliser.

$$S(t, x, V, L_U) = \rho(t, V, L_U) * \tau(t, x) \qquad t \in L_U, x \in [0..\infty) \tag{3.6}$$

Compte-tenu de cet ensemble de facteurs, le comportement du processus de sélection est illustré par la figure 3.2, où cinq tests constituant les vulnérabilités de $\mathrm{V} = \{[v_1 = t_1],[v_2 = t_2 \wedge (t_3 \vee t_4)],[v_3 = t_1 \wedge t_5]\}$ sont évalués sur dix périodes de temps ($\delta = 1$, $\lambda = 1/3$). Les tests représentés en ordonnée sont triés en ordre décroissant en fonction de leur utilité ; le premier test (le plus en bas) est ainsi le plus important. Il est exécuté sept fois sur les dix possibles, un ratio bien plus élevé que ceux des autres tests, moins utiles que lui. Cependant, ces derniers sont toutefois évalués au moins une fois durant les dix périodes, à une fréquence plus faible. Il peut également être dénoté que le test t_3 est exécuté deux fois plus que t_4, alors que ces deux tests présentent la même utilité. Cette tendance est une conséquence intéressante de la nature probabiliste du processus de sélection.

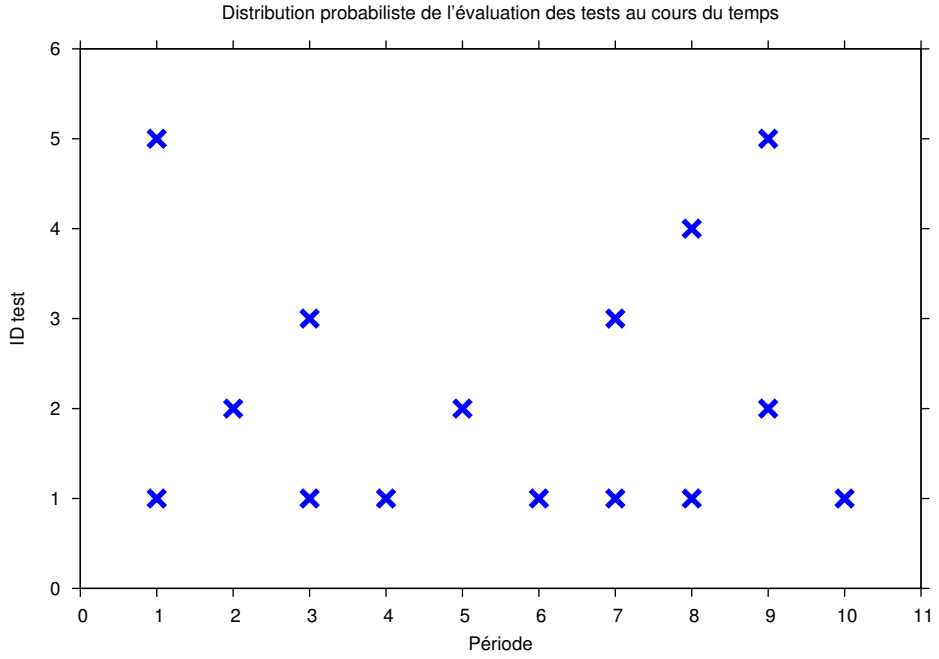

FIGURE 3.2 – Exemple de distribution d'évaluations de tests

3.2 Ovaldroid, un framework probabiliste pour la détection de vulnérabilités

Dans cette section, nous présentons Ovaldroid, un framework probabiliste de détection de vulnérabilités à l'aide du langage OVAL. Ce framework intègre le modèle proposé dans la section précédente afin d'augmenter efficacement la sécurité des systèmes Android. L'architecture du framework est détaillée dans sa globalité, tout comme la stratégie d'analyses externalisées qu'il met en place dans le but de limiter au maximum les charges engendrées sur les systèmes analysés.

3.2.1 Architecture du framework

L'architecture d'Ovaldroid (figure 3.3) a été pensée comme un service centralisé capable d'analyser à distance les vulnérabilités sur les systèmes Android. Elle se compose de deux blocs principaux,

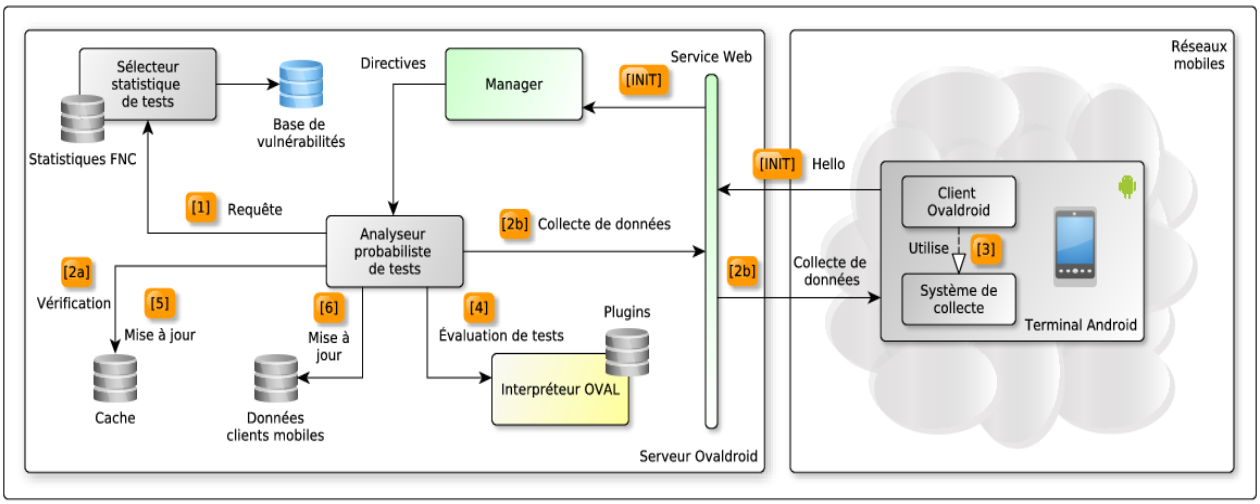

FIGURE 3.3 – Architecture d'Ovaldroid

à savoir un serveur qui gère intégralement le processus d'analyse, et un client implanté dans les systèmes mobiles afin de pouvoir exploiter le service de détection.

Interactions client-serveur

Les clients questionnent périodiquement le serveur pour se synchroniser avec lui au cas où un évènement (*e.g.* analyse, changement de paramètre) soit ordonnancé et en attente. Ces communications sont initiées par le client à l'aide de message *Hello*, envoyés au service web mis en place côté serveur. En se basant sur l'historique associé au système client et sur sa fréquence d'analyse (δ), le serveur est capable de déterminer via son module manager si une nouvelle séance de détection est à effectuer ou non. Il transmet cette information dans sa réponse au client par l'intermédiaire du service web. Si une analyse est programmée, le module manager envoie des directives spécifiques au module d'analyse probabiliste, chargé de conduire l'analyse. Ce dernier exécute itérativement une séquence de tests OVAL pour ce client jusqu'à ce que le pourcentage requis de vulnérabilités traitées (λ) soit atteint.

Application de la sélection probabiliste

Afin de sélectionner le test à réaliser au cours d'une itération, l'analyseur probabiliste utilise les services du module de sélection statistique. Celui-ci récupère le jeu d'équations FNC courant pour extraire les statistiques individuelles de chaque test encore non évalué. Il construit la liste ordonnée L_U en appliquant la formule 3.3 pour calculer leur utilité respective sur le jeu de vulnérabilités courant. L'analyseur se base en partie sur cette liste pour choisir le prochain test à réaliser. Il combine pour chacun des tests leur valeur dans la liste et le temps d'ancienneté (connu dans l'historique associé au client) afin d'en extraire une probabilité finale, exprimée par la formule 3.6. Ces probabilités sont utilisées dans le tirage au sort servant à choisir le prochain test. Les tests les plus utiles sont ainsi les plus susceptibles d'être sélectionnés, et les tests moins importants gardent néanmoins une chance d'être exécutés.

Cache de tests et d'objets

Une fois qu'un test a été sélectionné pour être évalué sur l'état d'un système client, l'analyseur vérifie dans un premier temps s'il n'existe pas d'entrée valide dans le cache qu'il maintient (étape 2a). Si ce dernier contient une entrée pour le test et le client en question, alors celle-ci est utilisée par l'analyseur pour connaître directement le résultat du test, sans avoir à interagir avec le client. Le cache stocke également les objets OVAL, du fait que ceux-ci peuvent être utilisés par plusieurs tests différents. Ainsi, si aucune entrée donnant le résultat du test n'est trouvée, l'analyseur sonde le cache pour demander cette fois l'objet associé à ce test. Si une entrée pour cet objet existe, l'analyseur détermine alors le résultat du test en comparant l'objet à l'état requis, évitant encore une fois

d'interagir avec le système client. Si en revanche, aucune entrée dans le cache n'est trouvée ni pour le test ni pour l'objet associé, alors l'analyseur interagit directement avec le client pour collecter les informations nécessaires au test (construction de l'objet). À partir de ces données collectées, le test est évalué par l'analyseur côté serveur.

Collecte de données et évaluation des tests

La collecte de données est réalisée côté client par l'intermédiaire d'une application Android faisant office d'agent (étape 3). Celui-ci se veut transparent et peu complexe, la majeure partie de l'intelligence du framework étant située côté serveur. Une fois les données récupérées et transmises au serveur, l'analyseur utilise un interpréteur OVAL pour évaluer le test associé (étape 4). En fonction de la nature des vulnérabilités, différents types de tests peuvent être utilisés pour leur description (*e.g.* test sur la présence d'un fichier, d'un processus, test de version de l'OS, etc.). Dans ce contexte, l'interpréteur OVAL utilise un ensemble de plugins pour chaque type de test. Chaque plugin permet de savoir comment collecter et analyser les informations nécessaires au type de test pour lequel il a été créé. Après évaluation, l'objet construit et le résultat du test sont enregistrés dans le cache pour une utilisation future (étape 5). Finalement, le résultat du test et sa nouvelle date de dernière évaluation sont aussi retranscrits dans l'historique du client, maintenu côté serveur (étape 6). Le processus d'analyse boucle sur les étapes 1 à 6 jusqu'à ce que le pourcentage de couverture requis (λ) soit atteint. Les résultats finaux d'analyses sont de même enregistrés dans l'historique du client (*e.g.* pour établissement de statistiques).

3.2.2 Stratégie d'analyse et d'externalisation

Dans cette sous-section, nous résumons la stratégie d'analyse utilisée dans Ovaldroid en spécifiant son algorithme. Après une explication détaillé de son comportement, nous proposons un protocole simple pour l'échange d'informations entre les clients mobiles et le serveur, afin de pouvoir réaliser les analyses à distance.

Algorithme pour l'analyse des systèmes

Les étapes suivies au cours d'une analyse sont décrites par l'algorithme 2, qui prend en entrée le seuil de couverture λ associé au client et le jeu complet de vulnérabilités en FNC (ce jeu est dupliqué intégralement depuis la base de connaissances du serveur à chaque fois qu'une nouvelle analyse démarre). Les résultats de l'analyse sont donnés en tant que sortie de l'algorithme.

Le principe de l'analyse consiste à sélectionner et évaluer les tests sur le système cible jusqu'à ce que le seuil de couverture soit atteint (ligne 2). À chaque itération, un test est choisi comme expliqué en sous-section 3.1.3, en considérant son utilité pour atteindre le seuil de couverture requis, l'impact des vulnérabilités dans lesquelles il est impliqué et le temps écoulé depuis sa dernière évaluation (ligne 3). L'algorithme cherche en premier lieu une entrée dans le cache donnant le résultat du test choisi (ligne 4). Si un résultat est trouvé, il est directement utilisé (ligne 5). S'il n'existe pas, l'objet référencé par ce test est cherché dans le cache (ligne 8). Si l'objet est trouvé (ligne 9), il est directement utilisé. En cas contraire, le processus de collecte de données est lancé sur le système client (ligne 11) afin de pouvoir construire l'objet. Après un cache-hit ou la collecte de données, le test est évalué (ligne 13) et le cache est mis à jour, en y ajoutant l'objet (si construit) et le résultat du test (ligne 14). Le résultat de l'évaluation s'intègre au résultat global de l'analyse courante, contenant notamment les résultats des tests déjà évalués au cours des itérations précédentes (ligne 16). En considérant ces résultats, les vulnérabilités sont réduites comme expliqué en sous-section 3.1.2, en remplaçant les résultats de tests connus dans leur équation FNC respective (ligne 17). Finalement, la couverture de vulnérabilités atteinte en ce point est mise à jour (ligne 18). L'algorithme termine quand cette couverture égale ou dépasse le seuil requis. Cette stratégie est appliquée à chaque fois que le serveur Ovaldroid considère qu'une analyse est requise sur un système.

Externalisation des analyses

Bien qu'ordonnancées par le serveur, les analyses sont initialement déclenchées par le client, qui contacte périodiquement le serveur avec des messages *Hello* pour se manifester. Comme illustré par

Algorithme 2: Analyse probabiliste

Input : FNCVulnList $vulnList$, SeuilCouverture λ
Output : ResultatsAnalyse $resultats$

```
1  couverture ← 0;
2  while couverture < λ do
3  |  test ← selectionProbabilisteTest(vulnFNCliste);
4  |  if existeEntreeCache(test) then
5  |  |  resultat_test ← resultatCache(test);
6  |  else
7  |  |  ref_objet ← extraireRefObjet(test);
8  |  |  if existeEntreeCache(ref_objet) then
9  |  |  |  objet ← objetCache(ref_objet);
10 |  |  else
11 |  |  |  objet ← collecteObjetSurClient(ref_objet);
12 |  |  end
13 |  |  resultat_test ← evaluer(test, objet);
14 |  |  updateCache(test, objet, resultat_test);
15 |  end
16 |  updateResultatsAnalyse(resultats, resultat_test);
17 |  reductionFNCVulnList(vulnList, resultat_test);
18 |  updateCouverture(couverture, vulnList, test, resultats);
19 end
```

la figure 3.4, les communications se composent d'un ensemble de messages envoyés par le client et de réponses fournies par le serveur. Ces réponses sont interprétées par le client pour déterminer les actions devant être réalisées. Elles peuvent exprimer le déclenchement d'une nouvelle analyse (DebutAnalyse), la mise à jour de la politique de sécurité du client (Update), la survenue d'une erreur (ERREUR), ou l'absence d'action particulière (OK). Des techniques de piggybacking sont employées dans les communications pour réduire le montant de messages à envoyer. Si une nouvelle analyse est requise (*e.g.* en se basant sur le paramètre δ du client, ou si de nouvelles descriptions de vulnérabilités sont disponibles), le serveur envoie la réponse appropriée ainsi que la description du premier objet OVAL à collecter. Tout le processus d'analyse est géré côté serveur, seule la collecte de données est à la charge du client. Celui-ci envoie l'objet collecté dans un nouveau message (PostObjet) destiné au serveur. À la réception de l'objet, le serveur évalue le test et réduit le jeu de vulnérabilités. Il répond ensuite au client en lui indiquant la description d'un nouvel objet à collecter (SuiteAnalyse), ou en le notifiant que l'analyse est terminée (FinAnalyse).

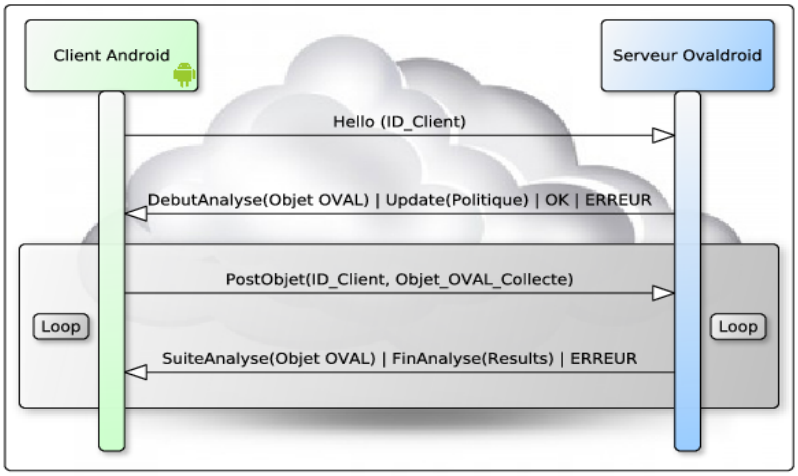

FIGURE 3.4 – Communication client-serveur pour l'externalisation d'analyse

Du point de vue client, ce dernier entre dans une boucle tant que le serveur envoie des réponses SuiteAnalyse, *i.e.* jusqu'à ce que l'analyse se termine (ou qu'une erreur survienne). La collecte d'objet est peu complexe comparé à l'évaluation d'un test, ce qui permet d'économiser des ressources sur les systèmes mobiles. L'analyse se réalise ainsi à distance, de façon interactive par l'intermédiaire de seulement deux requêtes HTTP invoquées par le client. Cependant, d'intéressants protocoles de management tels que NETCONF [22] existent déjà et leur usage peut être envisagé dans un futur proche, lorsque leur lien avec OVAL et le protocole SCAP bénéficiera d'une maturité plus prononcée. L'utilisation de tels protocoles pourrait notamment permettre le déport de la construction d'objets côté serveur, qui sonderait seulement le système cible pour collecter les informations de façon granulaire.

Chapitre 4

Prototypage et évaluation

Sommaire

Afin de tester concrètement l'approche proposée dans le chapitre précédent, nous avons développé un prototype du framework Ovaldroid. Nous avons également conduit un ensemble d'expériences pour évaluer le comportement et les performances de notre solution. Ce chapitre présente les choix et détails de l'implémentation du prototype d'Ovaldroid ainsi que les résultats et les analyses de son expérimentation.

4.1 Choix d'implémentation

Ovaldroid à été conçu en se basant sur une architecture client-serveur. Afin de disposer d'un support d'expérimentation stable dans le temps imparti, certaines fonctionnalités auxiliaires du framework ont été omises dans le prototype, tant du côté client (*e.g.* changement de politique de sécurité) que du côté serveur (*e.g.* alimentation automatisée de la base de définitions depuis la base publique OVAL). En revanche, tous les points clefs du modèle (*e.g.* sélection et évaluation de test, distribution des analyses à travers le temps, cache) ont été implantés afin de pouvoir évaluer précisément l'approche de notre solution.

4.1.1 Prototype serveur

Tous les modules représentés dans le bloc serveur de la figure 3.3 ont été intégralement développés en Java. L'importation de connaissances en matière de vulnérabilités (*i.e.* définitions OVAL) est réalisée en externe, par l'usage d'un outil auxiliaire développé à cette fin. Les vulnérabilités pour la plateforme Android sont décrites et retranscrites en définitions OVAL par le biais des schémas fournis dans le projet OVAL Sandbox [13]. Pour chaque vulnérabilité importée dans la base du serveur, la combinaison logique qui représente cette vulnérabilité est d'abord extraite depuis sa définition. Cette combinaison est ensuite transformée en FNC grâce au convertisseur logique disponible dans le projet Aima [12].

Service web

Les clients Android interagissent avec le serveur par l'intermédiaire de son service web. Celui-ci a été développé en utilisant la technologie Jersey, l'implémentation standard des services web

d'architecture REST [1] pour Java. À la différence du modèle SOAP [2] qui se bâtit sur du XML (WSDL [3]) et s'oriente RPC [4], l'architecture REST est peu complexe et n'ajoute qu'une fine couche d'abstraction pour être fonctionnelle. Les services web de ce type étendent simplement les méthodes du protocole HTTP [5] (*e.g.* GET,POST) pour mettre en place leurs fonctionnalités, accessibles par des URIs spéciales que les clients utilisent pour indiquer leurs requêtes. Cette simplicité se répercute directement sur le montant de ressources allouées lors des communications, un avantage certain pour les systèmes Android utilisant les services.

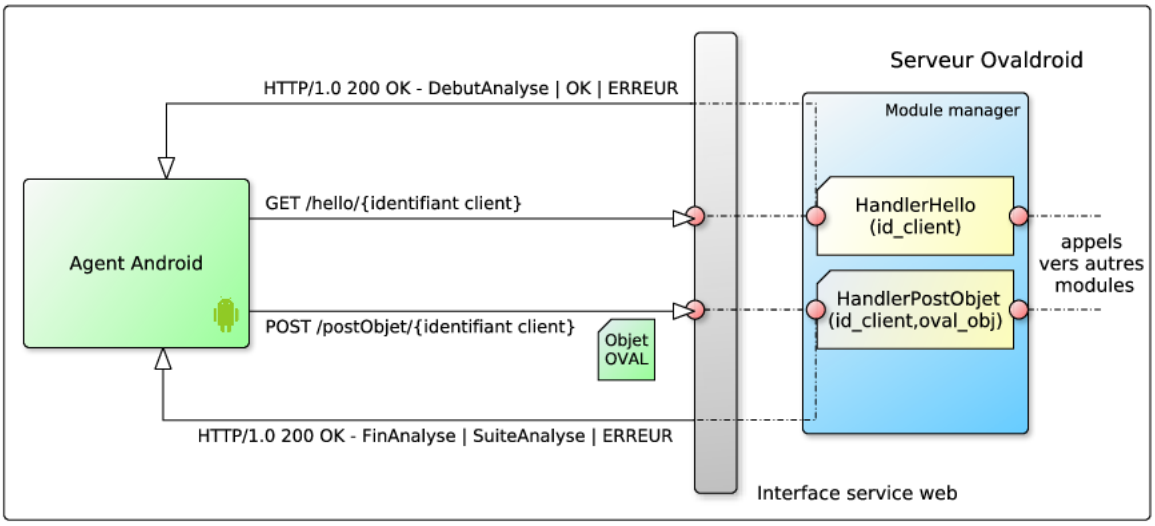

FIGURE 4.1 – Service web REST du prototype serveur Ovaldroid

Dans le prototype d'Ovaldroid, seulement deux URIs proposées par le service web suffisent pour que les clients exploitent pleinement le service d'analyse. La première d'entre elle utilise la méthode GET, et permet aux clients d'envoyer les messages *Hello* au serveur en spécifiant leur identifiant. La seconde URI se base sur la méthode POST, et permet aux client d'envoyer les objets OVAL au serveur durant les analyses. Ces deux types de requêtes sont prises en charge côté serveur par le module manager, comme illustré par la figure 4.1. Celui-ci détermine l'action à réaliser en fonction de la requête et du client, et appelle si besoin d'autre modules du serveur pour compléter le travail ou la réponse à retourner. Afin de respecter la philosophie du standard OVAL, le format des réponses envoyées par le service web se base sur un schéma XML qui lui est propre, intégré au code Java grâce à la technologie JAXB [6]. Néanmoins, l'usage du format JSON peut tout à fait être envisagé pour faciliter le parsing et optimiser l'économie de ressources côté client.

Gestion et manipulation des données

Le bloc serveur interagit fréquemment avec une base de données (BDD) locale, notamment pour y stocker ou retrouver les données concernant les vulnérabilités connues, les clients (*e.g.* paramètres, historique) et l'ensemble du mécanisme de cache. Cette base de données est implémentée en utilisant MySQL, un SGBD [7] libre et largement répandu dans la communauté informatique. Les modules du bloc serveur en interaction avec la base (*i.e.* le sélecteur statistique de tests et l'analyseur) accèdent directement à cette dernière en utilisant le driver JDBC [8] pour établir la connexion et réaliser les requêtes SQL.

1. REpresentational State Transfer
2. Simple Object Access Protocol
3. Web Services Description Language
4. Remote Procedure Call
5. Hyper Text Transfer Protocol
6. Java Architecture for XML Binding
7. Système de Gestion de Base de Données
8. Java DataBase Connectivity

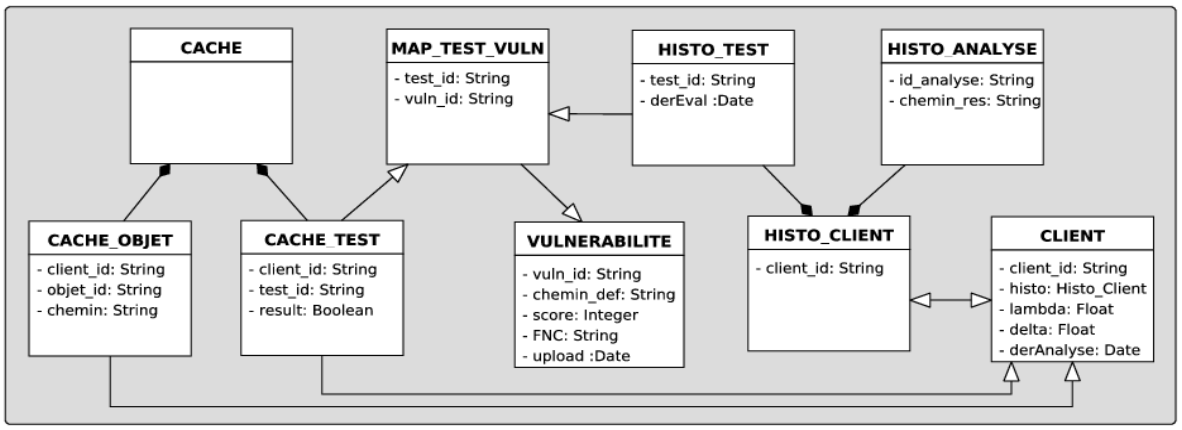

FIGURE 4.2 – Modélisation UML des données côté serveur

Les différentes données utilisées dans le prototype serveur pour les analyses des clients sont représentées par le modèle UML décrit par la figure 4.2. La classe *Vulnérabilité* permet de gérer les informations sur les vulnérabilités. Elle donne pour chacune d'entre elles l'identifiant unique (*e.g.* identifiant de définition OVAL), le chemin vers sa définition (enregistrée sur le disque), la date à laquelle elle a intégré la base, son facteur d'importance et son équation en FNC. La classe *Client* permet de manipuler les données relatives à chaque client connu. Elle renseigne leur identifiant, leur paramètre λ et δ, leur date de dernière analyse et leur historique. L'historique d'un client se compose d'une part des résultats de ses analyses précédentes (fichiers *OVAL Results* stockés sur le disque), d'autre part des dates de dernière évaluation de chaque test. Le cache (représenté par la classe éponyme) comprend un ensemble de tests et d'objets, respectivement modélisés par les classes *Cache_Test* et *Cache_Objet*. Ces deux classes ont une structure similaire, et renseignent pour chaque entrée l'identifiant du client concerné, celui du test ou de l'objet, le résultat (test) ou le chemin vers le fichier correspondant (objet), et leur date de création, permettant de vérifier leur validité. Enfin, la classe *Map_Test_Vuln* indique pour chaque test connu une vulnérabilité dans laquelle il est employé. Cette dernière classe permet notamment de fournir le savoir nécessaire à la génération de définition mono-test pour l'interpréteur, comme expliqué dans le paragraphe suivant.

Évaluation de tests

Les tests sont évalués côté serveur grâce à l'interpréteur OVAL multi-plateformes XOvaldi [29]. Celui-ci est appelé par l'analyseur probabiliste avec une définition mono-test et un fichier *OVAL System Characteristics* (contenant l'objet OVAL construit par le client) en entrée, et donne en sortie le résultat de l'analyse de la définition, *i.e.* le résultat du test sous forme de fichier *OVAL Result*, comme représenté par la figure 4.3.

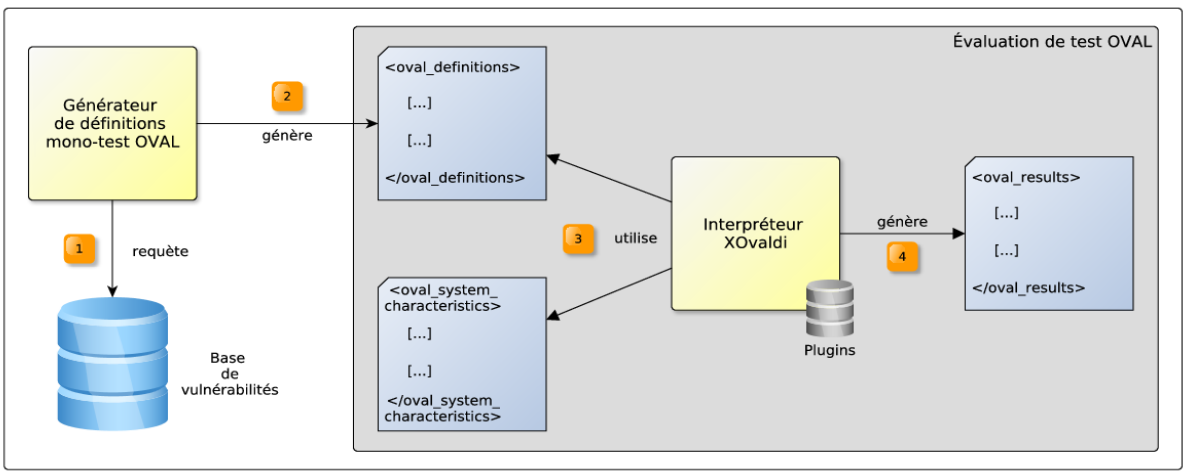

FIGURE 4.3 – Évaluation de tests sur le prototype serveur Ovaldroid

Au cours d'une itération, après la collecte de l'objet requis, une définition mono-test représentant le test à évaluer est forgée en mémoire vive juste avant l'appel à l'interpréteur. Un module auxiliaire (non représenté sur la figure 3.3), est chargé de mener à bien cette tâche. Pour ce faire, le module récupère dans un premier temps une définition contenant ce test depuis la base de données du serveur. Il sait quelle définition récupérer en sondant les correspondances représentées par la classe *Map_Test_Vuln* du modèle UML de la figure 4.2. Puis, par l'intermédiaire de requêtes XPath, il extrait de la définition les données (*e.g.* objet et état OVAL associés) du test, pour les retranscrire ensuite dans une définition vierge. Le module rajoute finalement les méta-données caractéristiques des définitions OVAL pour compléter la génération de la définition temporaire. Lorsque la génération est terminée, la définition forgée est transmise à l'analyseur, qui la passe en paramètre à XOvaldi lors de son appel. L'interpréteur dispose d'un ensemble de plugins et exécute celui qui est spécifique au type de test pour réaliser l'analyse. Une fois l'évaluation terminée, la définition mono-test est supprimée. L'analyseur récupère la valeur du test en faisant une requête XPath spécifique sur le fichier résultat OVAL généré par l'interpréteur.

4.1.2 Prototype client

Le prototype client Ovaldroid pour les systèmes mobiles est simple et léger. Il se compose de deux activités facultatives (écran d'accueil, écran de paramètres/préférences), d'un service central orchestrant les actions à réaliser, d'une librairie externe pour la collecte de données, et de tâches asynchrones pour les communications réseau. Ce prototype a été développé pour être supporté par les versions Android égales ou supérieures à 2.3.3, représentant grossièrement 80% de l'ensemble des smartphones et tablettes commercialisés jusqu'à présent.

Communications réseau asynchrones

L'ensemble des communications avec le serveur sont exécutées en tâches asynchrones, afin d'éviter tout blocage (ANR [9]) du service et de l'agent en cas d'incident non prévu, dû à l'instabilité de la connectivité réseau par exemple. Cette pratique est instaurée de force depuis les versions Android 3.X *Honeycomb* et supérieures, où des exceptions sont levées à chaque fois qu'une opération réseau est effectuée dans le thread principal d'une application. Bien que ces exceptions n'existent pas pour les versions antérieures à *Honeycomb*, l'usage de tâches asynchrones pour les communications réseau est néanmoins tout à fait supporté. Par ailleurs, les communications avec le service web sont principalement implémentées à l'aide de l'API Apache HttpCore [10], intégralement supportée par la plateforme Android. Le parsing des réponses XML du serveur est quant à lui géré par deux classes Java générées à l'aide de JAXB à partir du schéma des réponses.

Service central

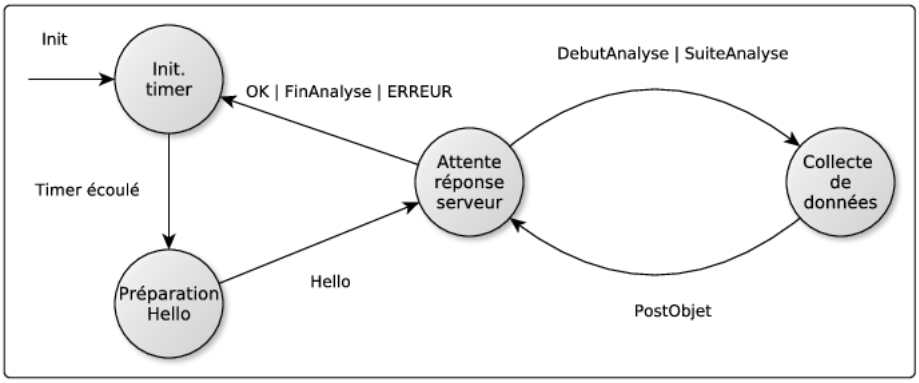

FIGURE 4.4 – Automate à états finis du prototype client Ovaldroid

Le service central de l'agent est lancé explicitement (activité écran d'accueil) ou implicitement (boot terminé) sur le système. Il est chargé de sonder périodiquement le serveur à l'aide de message

9. Application Not Responding

Hello, et d'orchestrer si besoin les actions spécifiques sur le système en fonction des réponses reçues. Lorsque le service est lancé, les différents états dans lesquels le client peut se trouver sont donnés par l'automate à états finis représentés par la figure 4.5. Seulement quatre états suffisent pour résumer son activité. À son démarrage (action *Init*), le service programme en premier lieu un timer sur le client (état *Init. timer*). Lorsque le timer est écoulé, un message Hello doit être envoyé au serveur (état *Préparation Hello*). Une fois ce message envoyé, le service se met en attente de la réponse du serveur (état *Attente réponse serveur*). Si celle-ci contient le message OK ou Erreur, le timer est reprogrammé et marque le début d'un nouveau cycle (état *Init. timer*). Si en revanche, la réponse du serveur contient le message DebutAnalyse, alors le service prépare la collecte de données (état *Collecte de données*), détaillée dans le paragraphe suivant. Il rentre dans la boucle spécifique à l'analyse, et les objets OVAL demandés sont transmis au serveur tant que celui-ci envoie des messages SuiteAnalyse. Lorsque le serveur répond par un message FinAnalyse ou Erreur, le timer est reprogrammé pour le début d'un nouveau cycle (état *Init. timer*).

Collecte d'informations et construction d'objets OVAL

Le processus de construction d'objets OVAL est exécuté côté client par XOvaldi4Android, une extension pour Android de l'interpréteur XOvaldi. XOvaldi4Android se présente sous forme de librairie (.jar) de 94KB fournissant l'intégralité du sous-système de collecte de données et de construction d'objets OVAL. Il est appelé automatiquement par le service central de l'agent pendant les analyses de vulnérabilités, pour créer les objets demandés par le serveur. Dans ce contexte, le service central récupère dans un premier temps la description de l'objet transmise dans la réponse du serveur. Dans le prototype actuel, une description d'objet est en fait représentée par une définition mono-test, forgée côté serveur comme expliqué en sous-section 4.1.1. Cette définition est directement intégrée dans le corps de la réponse du service web. Le service de l'agent l'extrait et la retranscrit dans un fichier temporaire, qu'il passe en paramètre lors de l'appel à XOvaldi4Android. Celui-ci analyse la définition pour déterminer quel objet doit être construit. À la manière de XOvaldi, il dispose d'un ensemble de plugins spécifiques à chaque type de test, qu'il utilise pour construire l'objet approprié. Le fichier *OVAL System Characteristics* contenant cet objet est généré à la fin du processus de collecte. Ce document est par la suite envoyé au serveur via son service web et l'URI prévue à cet effet (postObjet).

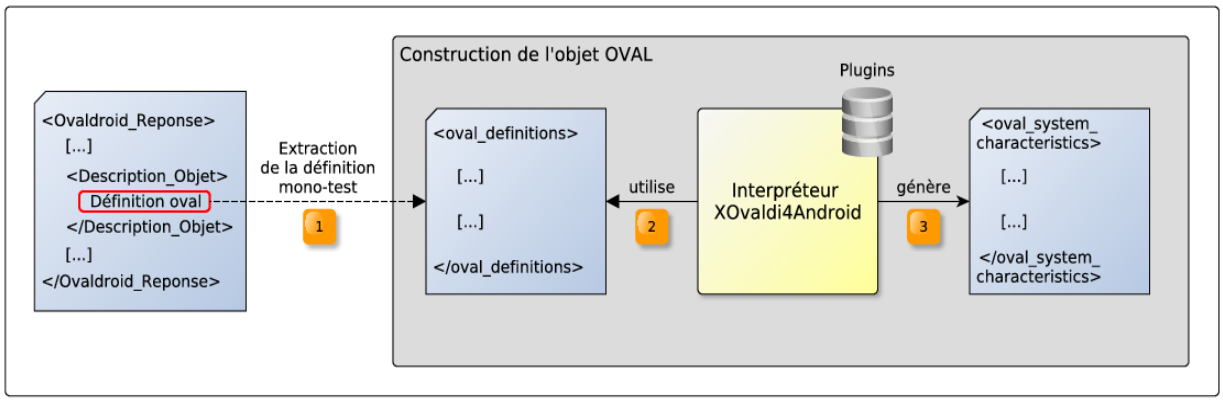

FIGURE 4.5 – Génération d'objets OVAL sur le prototype client Ovaldroid

4.2 Expérimentations du framework

Cette section présente les expériences que nous avons réalisées pour évaluer le comportement et les performances du prototype d'Ovaldroid. Le serveur a été déployé sur un ordinateur portable hébergeant une distribution Linux standard (kernel 3.7.9), et disposant d'un processeur Intel Core i7 2.20 Ghz ainsi que de 8 GB de mémoire vive. Le client a quant à lui été déployé sur un smartphone Samsung I9300 Galaxy S III (Quad-core 1.4 GHz, 4 GB de RAM) utilisant la version Android 4.1.0. Nous détaillons l'environnement des expérimentations et les résultat obtenus concernant les

performances de l'heuristique statistique, la convergence de la couverture, les fréquences d'analyses des vulnérabilités, et la réduction de charge engendrée côté client.

Contexte des expérimentations

Le jeu de vulnérabilités utilisé dans les expériences - exceptée la première, qui a été réalisée à une période différente - a été construit en utilisant des descriptions de vulnérabilités réelles pour la plateforme Android. Dans l'optique d'évaluer le passage à l'échelle, nous avons dupliqué leur structure pour créer de nouvelles descriptions et ainsi augmenter le nombre de définitions disponibles. En moyenne, les définitions utilisent deux tests pour décrire leur vulnérabilité. Les définitions créées représentent d'un point de vue sémantique la même vulnérabilité que leur définition mère. Cependant, d'un point de vue technique, elles utilisent des identifiants (identifiants de vulnérabilité, définition, test, objet et état OVAL) qui leur sont propres pour se différencier, et sont donc considérées comme des définitions distinctes à part entière. En suivant cette méthodologie, nous avons pu construire un jeu de vulnérabilités contenant jusqu'à 500 définitions différentes.

Concernant le paramétrage d'Ovaldroid, nous avons expérimenté le framework avec différentes valeurs pour le seuil de couverture périodique λ, alors que le laps de temps δ séparant deux analyses est resté fixé à 1. La période de validité du cache (test et objet) a été fixée à 3 périodes, signifiant que les objets et résultats de tests qui y sont enregistrés expirent au bout de trois analyses. Toute les expériences décrites dans les sous-sections suivantes ont été répétées un nombre fixé de fois afin de lisser au maximum les mesures. Les prises de mesures ont à chaque fois été effectuées côté serveur, celui-ci disposant de l'ensemble des données nécessaires pour profiler le framework.

4.2.1 Analyse de l'heuristique statistique

Notre première expérience profile le comportement des analyses lorsque le processus de sélection côté serveur ne considère que les données statistiques des tests pour choisir lesquels évaluer. Dans ce scénario, le premier test de la liste L_U (sous-section 3.1.2), *i.e.* celui avec l'utilité moyenne la plus élevée, est sélectionné à chaque itération au cours d'une analyse. Il peut s'avérer intéressant d'isoler et d'évaluer dans cette configuration les performances de l'heuristique statistique, caractérisée par la formule 3.3 permettant de classer les tests dans la liste L_U en fonction de leur utilité moyenne. Notre expérience se porte sur le nombre requis de tests pour atteindre un pourcentage de couverture donné au cours d'une analyse sur un système. Le cache (objets et tests) a été désactivé.

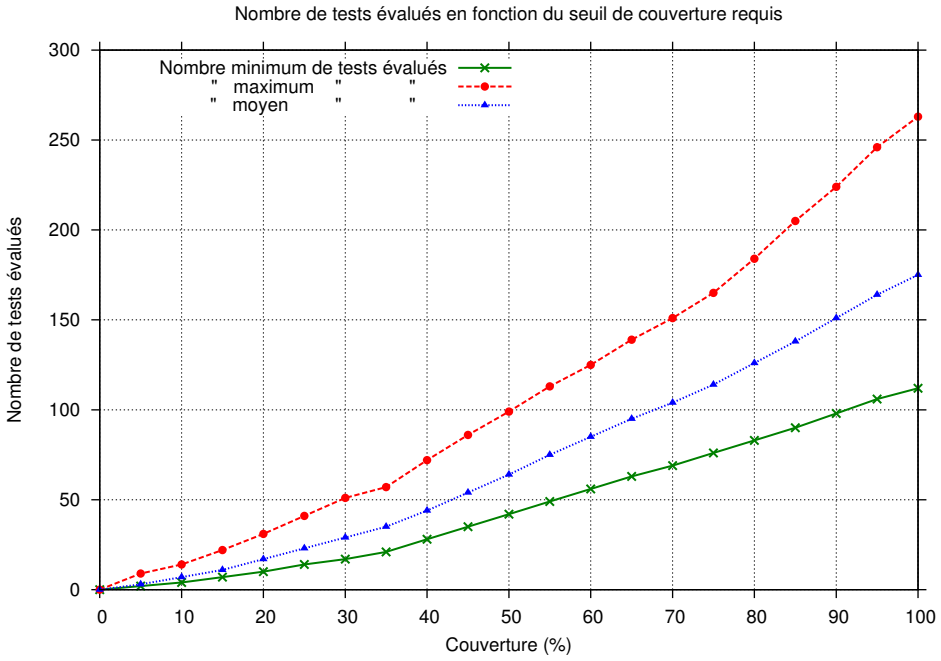

FIGURE 4.6 – Analyses par sélection statistique

La figure 4.6 donne les résultats de cette expérience, où un jeu spécifique de 350 tests permet de décrire 500 vulnérabilités (certains tests sont donc utilisés dans plusieurs définitions). Trois courbes y sont représentées pour exprimer le nombre minimal (points en croix), moyen (points en triangle) et maximal (points en ronds) de tests devant être réalisés pour atteindre le pourcentage de couverture indiqué en abscisse. Les disparités de valeurs entre les trois courbes s'expliquent par les variations aléatoire de l'état du système cible au cours du temps. Nous pouvons observer qu'environ un tiers des tests suffisent dans le meilleur des cas pour couvrir 100% des vulnérabilités de la base. À l'inverse, il faut évaluer environ trois quarts des tests pour atteindre cette couverture dans le pire des cas. Une caractéristique intéressante est la non nécessité d'évaluer la totalité des tests dans tous les cas. Cela s'explique par la notion de tests éliminés durant la réduction des vulnérabilités (sous-section 3.1.2). Bien que satisfaisantes en termes d'efficacité, les analyses de ce scénario ne se présentent jamais en pratique. En effet, elles sont fortement enclines à engendrer des famines de tests, puisque seules les données statistiques sont prises en comptes pour sélectionner ces derniers. Nous intégrons et évaluons dans nos expériences suivantes l'aspect probabiliste et temporel qui permet d'annuler ce problème.

4.2.2 Convergence de la couverture

Notre seconde expérience montre comment l'approche probabiliste converge vers la couverture complète du jeu de vulnérabilités à travers le temps. Cette convergence est atteinte lorsque toute les vulnérabilités de la base ont été analysées au moins une fois sur le système client. Elle représente un point clef du modèle, l'une des principales caractéristiques d'Ovaldroid étant sa capacité à distribuer la charge à travers plusieurs périodes d'analyses. En donnant une priorité plus importante aux tests les plus utiles tout en évitant la famine de tests (sous-section 3.1.3), la convergence sur l'ensemble des vulnérabilités est toujours finie et atteinte comme le montre la figure 4.7.

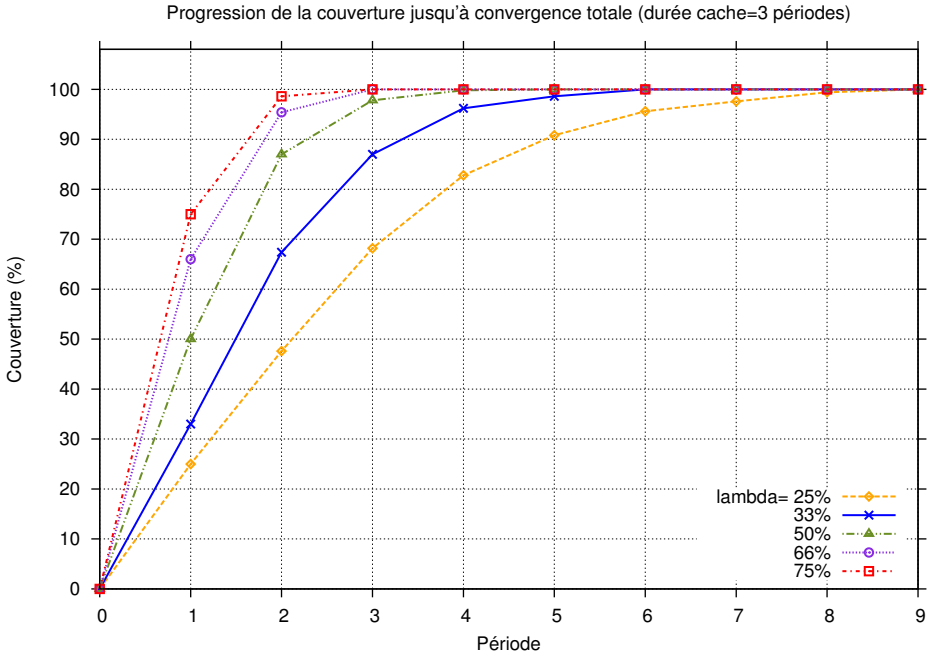

FIGURE 4.7 – Convergence de la couverture sur la totalité des vulnérabilités

En fonction des valeurs du seuil de couverture périodique λ, les résultats obtenus sont plus ou moins probants. Les différentes convergences associées s'étalent sur un intervalle allant de trois ($\lambda = 75\%$) à neufs ($\lambda = 25\%$) périodes d'analyse. Comme attendu, le temps de convergence croît à mesure que λ décroît. Un compromis efficace entre distribution et convergence peut être trouvé lorsque $\lambda = 33\%$. En considérant que chaque période est espacée d'un jour, toutes les vulnérabilités de la base sont alors analysées au moins une fois en moins d'une semaine (6 jours). Si la base reste la même, les vulnérabilités sont re-évaluées durant les périodes suivantes à une fréquence proportionnelle à leur impact et leur importance, en assurant toujours l'évaluation des moins utiles également. Si

de nouvelles vulnérabilités sont ajoutées à la base, une priorité élevée leur est temporairement attribuée tant qu'elles n'ont pas été évaluées. Ce processus de re-évaluation lisse la charge sur les systèmes cibles tout en produisant des résultats précis et complets. Il s'adapte aussi efficacement aux changements de la base de vulnérabilités.

4.2.3 Fréquences d'analyses des vulnérabilités

Dans l'expérience précédente, toutes les vulnérabilités ont le même score (formule 3.2, fonction I) concernant leur impact et leur importance. Afin d'évaluer la fréquence à laquelle chaque vulnérabilité est analysée à travers le temps en fonction de son score, nous avons réalisé une expérience complémentaire où un score aléatoire est attribuée à chacune d'entre elle.

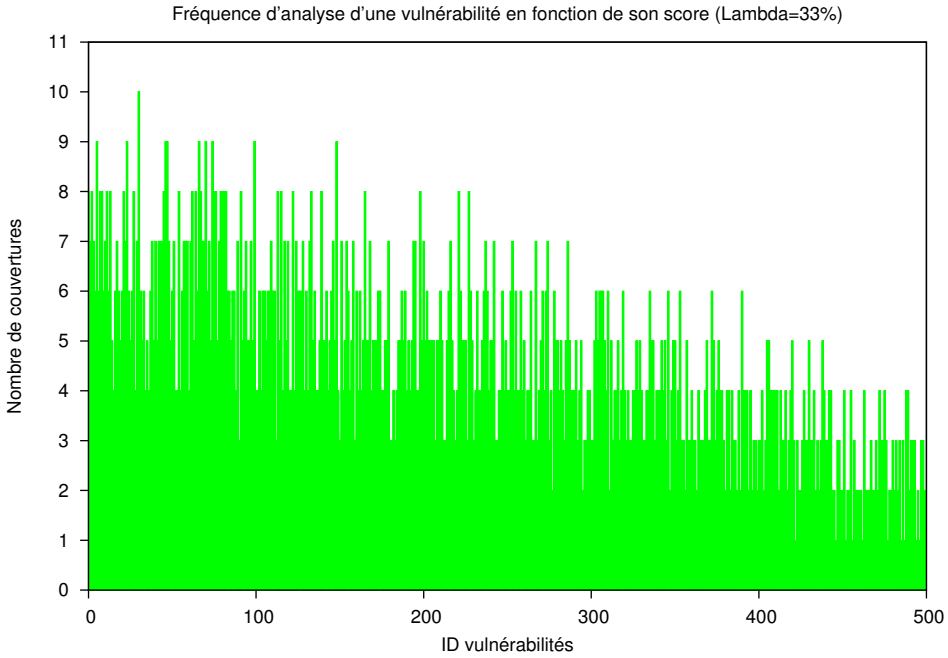

FIGURE 4.8 – Distribution des analyses en fonction des score des vulnérabilités

La figure 4.8 donne les résultats de cette expérience considérant quatorze périodes d'analyses (λ=33%). En abscisse, les vulnérabilités sont classées en ordre décroissant en fonction de leur score. Comme attendu, les vulnérabilités les plus importantes sont évaluées le plus fréquemment. Cependant, les vulnérabilités avec un impact moindre sont également évaluées plusieurs fois, prouvant que le problème de famine est aussi résolu dans cette configuration. Néanmoins, il est important de noter que la présence de disparités dans les scores des vulnérabilités peut influer sur le temps de convergence, notamment en fonction de l'intervalle des valeurs attribuées aux scores. En ajustant la valeur des scores entre 0 et 10 (nombres entiers), nous avons pu ainsi dénoter une augmentation moyenne d'environ 1.5 à 2 périodes d'analyse pour atteindre la convergence. L'efficacité de la normalisation des valeurs de score et de leur pondération pour atténuer ce phénomène fait partie des points sur lesquels nous comptons travailler prochainement.

4.2.4 Réduction de charge sur les clients

Notre dernière expérience porte sur la réduction de charge côté client, et permet d'analyser les variations de l'activité sur les systèmes mobiles en termes de nombre d'objets collectés par les agents. Une comparaison est dressée entre l'approche standard analysant toutes les vulnérabilités de la base à chaque période, et l'approche utilisée par Ovaldroid distribuant l'activité d'analyse à travers plusieurs périodes (λ=33%). Le jeu de vulnérabilités utilisé dans cet expérience se compose de 500 définitions contenant chacune deux tests uniques, entraînant un total de 1000 tests et 1000 objets.

La figure 4.9 décrit le comportement des deux approches en indiquant deux facteurs pour chacune d'entre elle, à savoir le nombre d'objets collectés périodiquement (courbes continues), et le

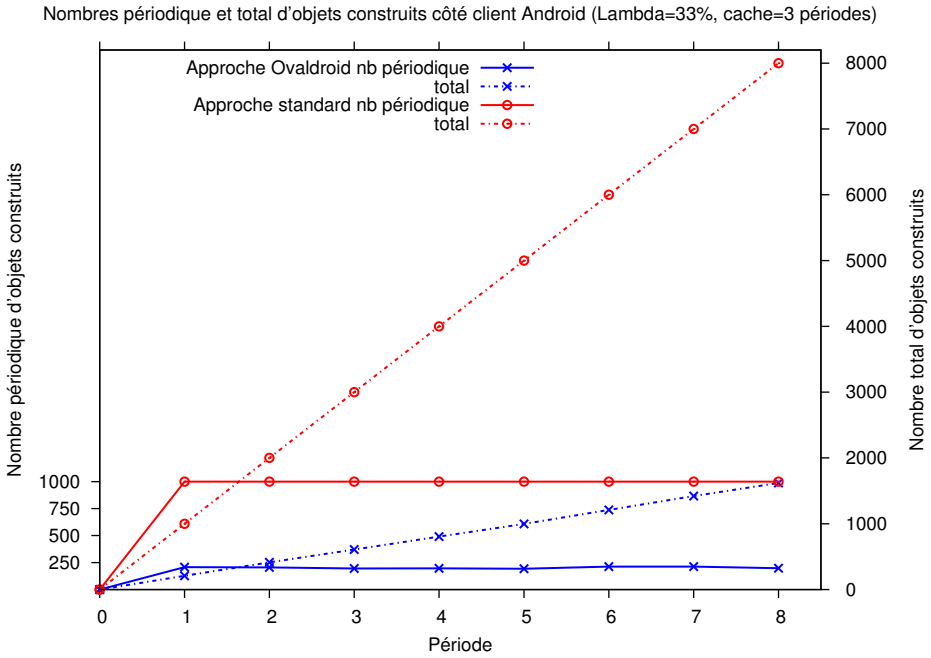

FIGURE 4.9 – Comparaison du nombre de collectes d'objet côté client

nombre total d'objets collectés depuis la première période (courbes en pointillés). L'approche standard réalise les mêmes opérations à chaque séance d'analyse et collecte 1000 objets par période. L'approche d'Ovaldroid évolue au cours du temps et collecte environ 200 à 250 objets par période, représentant seulement 20 à 25% du montant de l'approche standard. Bien que l'approche d'Ovaldroid soit plus lente que l'approche standard en termes de temps de convergence, la réduction de charge qu'elle permet d'engendrer sur les systèmes mobiles est significativement importante, contribuant grandement à l'efficacité et la réactivité des dispositifs contenant l'agent. Les courbes représentant les totaux respectifs des deux approches montrent clairement le gain apporté par l'approche d'Ovaldroid.

Chapitre 5

Conclusion

Les smartphones et autres terminaux portables, de par l'évolution de leurs technologies et l'abondance des services qu'ils fournissent, ont révolutionné l'usage et les bénéfices de l'informatique mobile. Néanmoins, les utilisateurs exploitant ces systèmes font aussi face à des problèmes de sécurité variés, du fait des nombreuses vulnérabilités que ces systèmes présentent (applications, services et systèmes d'exploitation). De plus, les terminaux mobiles affichent des ressources limitées, et ne peuvent donc pas bénéficier des mêmes mécanismes de défense que ceux rencontrés dans l'informatique traditionnelle. Ces ressources doivent être gérées intelligemment afin de pouvoir utiliser pleinement les terminaux, tout en assurant leur sécurité de façon optimale.

Bilan du travail

Dans ce travail, nous avons proposé une nouvelle approche pour détecter de manière légère et précise les vulnérabilités présentes sur les systèmes Android. Pour ce faire, nous avons présenté des méthodes statistiques afin d'optimiser les analyses, et un modèle probabiliste pour assurer leur précision et leur complétude sur l'ensemble des vulnérabilités connues à travers le temps. Nous avons élaboré une stratégie d'externalisation des analyses sur des serveurs distants pour réduire la charge de travail sur les stations mobiles durant les séances de détection. Nous avons également proposé un framework d'analyses basé sur le langage OVAL reprenant cette approche pour réduire globalement la consommation de ressources sur les systèmes cibles. Finalement, nous avons présenté un prototype d'implémentation du framework et procédé à de nombreuses expérimentations pour évaluer ses performances. Ces dernières montrent la faisabilité et les bénéfices de notre solution, en termes d'efficacité des analyses notamment, et de ressources économisées côté client mobile également.

Travaux futurs

Pour nos travaux futurs, nous prévoyons dans un premier temps des études supplémentaires sur les algorithmes statistiques et probabilistes du modèle, afin d'optimiser encore plus les analyses des systèmes. Nous envisageons également d'implémenter un mécanisme de *lookahead* pour réaliser efficacement des séquences de tests au lieu d'un seul par échange client-serveur. Cela permettra de réduire considérablement le montant de messages transmis sur le réseau au cours d'une analyse. À ce mécanisme de *lookahead* peut se greffer celui d'agrégation de données, à l'instar du protocole épidémique décrit en [59], pour transmettre intelligemment et efficacement les informations diverses entre les mobiles. Orienté comme un service distant, le framework Ovaldroid doit bénéficier de plus de sécurité pour être protégé des hostilités spécifiques à l'environnement du cloud : nous souhaitons implémenter les mécanismes nécessaires à la prévention d'attaques côté serveur (*e.g.* déni de service), et sécuriser les communications avec les clients mobiles (*e.g.* via SSH). Enfin, la détection de vulnérabilités n'est qu'une première étape dans le processus complet de gestion des vulnérabilités. Aussi, nous souhaitons implémenter les mécanismes de remédiation associés pour étendre notre solution.

Bibliographie

[1] Analyse du mécanisme de filtrage Google Bouncer. http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/. Dernière visite en juin 2013.

[2] Application Vulnerability Description Language (AVDL). https://www.oasis-open.org/. Dernière visite en mai 2013.

[3] Architecture d'Android. http://developer.android.com/about/versions/index.html. Dernière visite en mai 2013.

[4] CVE - Common Vulnerabilities and Exposures. http://cve.mitre.org/. Dernière visite en mai 2013.

[5] CVE pour Android. http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html. Dernière visite en mai 2013.

[6] CVSS, Common Vulnerability Scoring System. http://www.first.org/cvss/. Dernière visite en mai 2013.

[7] Guidelines for Managing and Securing Mobile Devices in the Enterprise. http://csrc.nist.gov/publications/drafts/800-124r1/draft_sp800-124-rev1.pdf. Dernière visite en juin 2013.

[8] La Machine Virtuelle Dalvik. http://www.dalvikvm.com. Dernière visite en mai 2013.

[9] La plateforme Android. http://www.android.com/. Dernière visite en mai 2013.

[10] L'API Apache HttpCore. http://hc.apache.org/httpcomponents-core-ga/httpcore/apidocs/. Dernière visite en juin 2013.

[11] Le langage OVAL. http://oval.mitre.org/language/. Dernière visite en juin 2013.

[12] Le Projet AIMA. http://code.google.com/p/aima-java/. Dernière visite en juin 2013.

[13] Le Projet OVAL Sandbox (support expérimental pour Android). https://github.com/OVALProject/Sandbox. Dernière visite en mai 2013.

[14] L'interpréteur standard Ovaldi. http://oval.mitre.org/language/interpreter.html. Dernière visite en mai 2013.

[15] MITRE Corporation. http://www.mitre.org/. Dernière visite en mai 2013.

[16] Mécanismes de sécurité dans Android. http://source.android.com/tech/security/. Dernière visite en mai 2013.

[17] NIST, National Institute of Standards and Technology. http://www.nist.gov/. Dernière visite en mai 2013.

[18] The OVAL Repository. http://oval.mitre.org/repository/. Dernière visite en mai 2013.

[19] Prévisions de Cisco sur l'environnement mobile. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html. Dernière visite en mai 2013.

[20] Prévisions des parts de marché des smartphones Android pour 2017. http://www.idc.com/getdoc.jsp?containerId=prUS23523812. Dernière visite en mai 2013.

[21] Prévisions des parts de marché des tablettes Android pour 2017. http://www.idc.com/getdoc.jsp?containerId=prUS24002213. Dernière visite en mai 2013.

[22] RFC du protocole NETCONF. http://www.faqs.org/rfcs/rfc4741.html. Dernière visite en juin 2013.

[23] Une introduction au langage OVAL. http://oval.mitre.org/documents/docs-06/an_introduction_to_the_oval_language.pdf. Dernière visite en mai 2013.

[24] VulnXML Vision Document. http://xml.coverpages.org/VulnXMLVisionDocument.pdf.

[25] Évolution des malwares Android au cours des derniers mois. https://blog.commtouch.com/cafe/wp-content/uploads/Android-malware-Jan-2013.jpg. Dernière visite en juin 2013.

[26] J. Banghart and C. Johnson. The Technical Specification for the Security Content Automation Protocol (SCAP) http://csrc.nist.gov/publications/nistpubs/800-126/sp800-126.pdf. 2009.

[27] M. Barrère, R. Badonnel, and O. Festor. Supporting Vulnerability Awareness In Autonomic Networks and Systems with OVAL. In *Proceedings of the seventh International Conference on Network and Service Management (CNSM 2011)*, pages 1–8, 2011.

[28] M. Barrère, R. Badonnel, and O. Festor. Vulnerability Management and Past Experience in Autonomic Networks and Services. Rapport de recherche INRIA (http://hal.inria.fr/hal-00747660), September 2012.

[29] M. Barrère, G. Betarte, and M. Rodriguez. Towards Machine-assisted Formal Procedures for the Collection of Digital Evidence. In *Proceedings of the Ninth Annual International Conference on Privacy, Security and Trust (PST 2011)*, pages 32–35, 2011.

[30] M. Barrère, G. Hurel, R. Badonnel, and O. Festor. Increasing Android Security using a Lightweight OVAL-based Vulnerability Assessment Framework. In *Proceedings of the IEEE 5th Symposium on Security Analytics and Automation (SafeConfig 2012)*.

[31] M. Barrère, G. Hurel, R. Badonnel, and O. Festor. A Probabilistic Cost-efficient Approach for Mobile Security Assessment. *Soumis dans une conférence internationale*, 2013.

[32] A. Bose and K.G. Shin. On Mobile Viruses Exploiting Messaging and Bluetooth Services. In *Proceedings of the Securecomm and Workshops 2006*, pages 1–10, 2006.

[33] H. Dai, C. Murphy, and G. Kaiser. Configuration Fuzzing for Software Vulnerability Detection. In *Proceedings of the fifth International Conference on Availability, Reliability, and Security (ARES'10)*, pages 525–530, 2010.

[34] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the State : A State-Aware Black-Box Vulnerability Scanner. In *Proceedings of the USENIX Security Symposium (USENIX)*, Bellevue, WA, August 2012.

[35] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *Security Privacy, IEEE*, 7(1) :50–57, 2009.

[36] W. Enck, P. Traynor, P. Mcdaniel, and T. La Porta. Exploiting Open Functionality in SMS-capable Cellular Networks. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, pages 393–404. ACM Press, 2005.

[37] M. Eslahi, R. Salleh, and N.B. Anuar. MoBots : A New Generation of Botnets on Mobile Devices and Networks. In *Proceedings of the IEEE Symposium on Computer Applications and Industrial Electronics (ISCAIE 2012)*, pages 262–266, 2012.

[38] A.P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)*, pages 3–14. ACM, 2011.

[39] M. Ghorbanzadeh, Y. Chen, Z. Ma, T.C. Clancy, and R. McGwier. A Neural Network Approach to Category Validation of Android Applications. In *Proceedings of the International Conference on Computing, Networking and Communications (ICNC 2013)*, pages 740–744, 2013.

[40] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong. CRAX : Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations. In *Proceedings of IEEE Sixth International Conference on Software Security and Reliability (SERE 2012)*, pages 78–87, 2012.

[41] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou. Analysis of Android Applications' Permissions. In *Proceedings of the IEEE Sixth International Conference on Software Security and Reliability Companion (SERE-C 2012)*, pages 45–46, 2012.

[42] G. Kambourakis, C. Kolias, S. Gritzalis, and J.H. Park. DoS Attacks Exploiting Signaling in UMTS and IMS. *Computer Communications vol. 34*, pages 226–235, 2011.

[43] S. Khan, M. Nauman, A.T. Othman, and S. Musa. How Secure is Your Smartphone : An Analysis of Smartphone Security Mechanisms. In *Proceedings of the International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec 2012)*, pages 76–81, 2012.

[44] L. Laribee, D.S. Barnes, N.C. Rowe, and C.H. Martell. Analysis and Defensive Tools for Social-Engineering Attacks on Computer Systems. In *Proceedings of the IEEE Information Assurance Workshop 2006*, pages 388–389, 2006.

[45] S. Li. Juxtapp and DStruct : Detection of Similarity Among Android Applications. In *Master's thesis, EECS Department, University of California, Berkeley.*, May 2012.

[46] B. Liu, L. Shi, Z. Cai, and M. Li. Software Vulnerability Discovery Techniques : A Survey. In *Fourth International Conference on Multimedia Information Networking and Security (MINES 2012)*, pages 152–156, 2012.

[47] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[48] M. Maxim and D. Pollino. *Wireless Security.* The McGraw-Hill Companies, 2002.

[49] C. Mulliner. Security of Smart Phones. In *Master's thesis, University of California, Santa Barbara.*, June 2006.

[50] C. Mulliner. Vulnerability Analysis and Attacks on NFC-Enabled Mobile Phones. In *Proceedings of the Forth International Conference on Availability, Reliability and Security (ARES 2009)*, pages 695–700, 2009.

[51] C. Mulliner and J.-P. Seifert. Rise of the iBots : Owning a Telco Network. In *Proccedings of 5th International Conference on Malicious and Unwanted Software (MALWARE 2010)*, pages 71–80, 2010.

[52] M. Nagy and M. Kotocova. An IP based Security Threat in Mobile Networks. In *Proceedings of the 35th International Convention (MIPRO 2012)*, pages 667–670, 2012.

[53] M. Nita and D. Notkin. White-box Approaches for Improved Testing and Analysis of Configurable Software Systems. In *ICSE Companion*, pages 307–310, 2009.

[54] M. Nkosi and F. Mekuria. Improving the Capacity, Reliability and Life of Mobile Devices with Cloud Computing. In *Proceedings of the IST-Africa Conference 2011*, pages 1–9, 2011.

[55] C. Oriaku, N. Alwan, and I.A. Lami. The Readiness of Mobile Operating Systems for Cloud Computing Services. In *Proceedings of the 4th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT 2012)*, pages 49–55, 2012.

[56] X. Ou and S. Govindavajhala. Mulval : A Logic-Based Network Security Analyzer. In *Proceedings of the 14th USENIX Security Symposium (Usenix 2005)*, pages 113–128, 2005.

[57] H. Pieterse and M.S. Olivier. Android Botnets on the Rise : Trends and Characteristics. In *Information Security for South Africa (ISSA 2012)*, pages 1–5, 2012.

[58] M. La Polla, F. Martinelli, and D. Sgandurra. A Survey on Security for Mobile Devices. *IEEE Communications Surveys Tutorials vol. 15*, (1) :446–471, 2013.

[59] A.G. Prieto and R. Stadler. A-GAP : An Adaptive Protocol for Continuous Network Monitoring with Accuracy Objectives. *IEEE Transactions on Network and Service Management*, 4(1) :2–12, 2007.

[60] C.J. Rhodes and M. Nekovee. The Opportunistic Transmission of Wireless Worms between Mobile Devices. *CoRR*, abs/0802.2685, 2008.

[61] M.R. Rieback, B. Crispo, and A.S Tanenbaum. RFID Malware : Truth vs. Myth. *IEEE Security Privacy vol. 4*, (4) :70–72, 2006.

[62] M. Rodriguez-Martinez, J. Seguel, M. Sotomayor, J.P. Aleman, J. Rivera, and M. Greer. Open911 : Experiences with the Mobile Plus Cloud Paradigm. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD 2011)*, pages 606–613, 2011.

[63] N. Shrestha. Security Assessment via Penetration Testing : A Network and System Administrator's Approach. In *Master's thesis, Oslo University College, Norway*, June 2012.

[64] X. Song, M. Stinson, R. Lee, and P. Albee. A Qualitative Analysis of Privilege Escalation. In *IEEE International Conference on Information Reuse and Integration*, pages 363–368, 2006.

[65] D. Votipka T. Vidas and N.Christin. All Your Droid are Belong to Us : A Survey of Current Android Attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies (WOOT'11)*, pages 10–10. USENIX Association, 2011.

[66] W.B. Tesfay, T. Booth, and K. Andersson. Reputation Based Security Model for Android Applications. In *Proceedings of the IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2012)*, pages 896–901, 2012.

[67] P. Traynor, M. Lin, M. Ongtang, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta. On Cellular Botnets : Measuring the Impact of Malicious Devices on a Cellular Network Core. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, pages 223–234. ACM, 2009.

[68] T.-E. Wei, A.-B. Jeng, H.-M. Lee, C.-H. Chen, and C.-W. Tien. Android Privacy. In *Proceedings of the International Conference on Machine Learning and Cybernetics (ICMLC 2012)*, volume 5, pages 1830–1837, 2012.

[69] T.-E. Wei, C.-H. Mao, A.-B. Jeng, H.-M. Lee, H.-T. Wang, and D.-J. Wu. Android Malware Detection via a Latent Network Behavior Analysis. In *Proceedings of the IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2012)*, pages 1251–1258, 2012.

[70] Y. Wen, W. Zhang, and H. Luo. Energy-optimal Mobile Application Execution : Taming Resource-poor Mobile Devices with Cloud Clones. In *Proceedings of the IEEE INFOCOM 2012*, pages 2716–2720, 2012.

[71] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. DroidMat : Android Malware Detection through Manifest and API Calls Tracing. In *Proceedings of the Seventh Asia Joint Conference on Information Security (Asia JCIS 2012)*, pages 62–69, 2012.

[72] W. Xu, Y. Zhang, and T. Wood. The Feasibility of Launching and Detecting Jamming Attacks in Wireless Networks. In *Proceedings of the ACM MOBIHOC 2005*, pages 46–57, 2005.

[73] L.C. Yarter. Private Cloud Delivery Model for Supplying Centralized Analytics Services. *IBM Journal of Research and Development*, 56(6) :10 :1–10 :6, 2012.

[74] Y. Zhou and X. Jiang. Dissecting Android Malware : Characterization and Evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP 2012)*, pages 95–109, 2012.

[75] N. Ziring and S. D. Quinn. Specification for the Extensible Configuration Checklist Description Format (XCCDF), Mars 2012.