ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE
CONCENTRATION TECHNOLOGIES DE L'INFORMATION
M. Sc. A.

PAR
Tarek SLAIMIA

DÉTECTION DES ROOTKITS NIVEAU NOYAU BASÉE SUR LTTNG

MONTRÉAL, LE 06 AOÛT 2015

PRÉSENTATION DU JURY

CE RAPPORT DE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

M. Abdelouahed Gherbi, directeur du mémoire
Département de génie logiciel et TI à l'École de Technologie Supérieure

M. Chamseddine Talhi, codirecteur du mémoire
Département de génie logiciel et TI à l'École de Technologie Supérieure

M. Alain April, président du jury
Département de génie logiciel et TI à l'École de Technologie Supérieure

M. Makan Pourzandi, examinateur externe
Expert en sécurité TI, Ericsson

ELLE A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

J'exprime mes profonds remerciements à mon directeur de recherche, le professeur Abdelouahed Gherbi pour l'aide compétente qu'il m'a apportée, pour sa patience, son encouragement, son soutien contenu et son œil critique qui m'a été très précieux pour structurer le travail.

J'adresse toute ma gratitude à mon codirecteur de recherche, le professeur Chamseddine Talhi pour les appuis scientifiques qu'il m'a procurés, sa disponibilité et ses conseils précieux qui m'ont permis d'améliorer la qualité de mon travail.

Je tiens à exprimer mes remerciements et mon respect au président de jury, le professeur Alain April et au membre externe M. Makan Pourzandi d'avoir accepté d'évaluer mon projet de recherche.

Je tiens aussi à remercier mes collègues du Laboratoire de Génie logiciel : Abdelfattah Amamra et Iheb Abdellatif.

Je dédie ce mémoire principalement à mes parents Borni et Sabra, mes sœurs Imen, Sihem, Houda et mon frère Ahmed. Je remercie également ma fiancée Amira pour son amour et son soutien.

J'exprime mes remerciements les plus sincères à toute personne qui m'a aidé et qui a contribué à ce projet de recherche.

DÉTECTION DES ROOTKITS NIVEAU NOYAU BASÉE SUR LTTNG

Tarek SLAIMIA

RÉSUMÉ

Les mécanismes de détection des logiciels malveillants, basés sur le traçage des activités de l'espace utilisateur, peuvent être facilement contournés par les rootkits niveau du noyau. Ce mémoire propose une solution de détection des techniques d'attaques utilisées par ce type de rootkits en se basant sur le traçage de l'activité du noyau Linux avec l'outil LTTng. Cette solution est basée sur l'analyse des données statiques et dynamiques du noyau Linux. Elle est conçue suivant deux axes : le premier est l'intégration des techniques de détection de rootkits dans les modules noyaux du traceur LTTng et le deuxième est l'utilisation des traces de LTTng contenant les données du noyau dans l'approche de détection basée sur des techniques d'apprentissage automatique. Ce deuxième axe a subi une optimisation des valeurs des vecteurs d'entrée correspondants aux différentes attaques et a subi encore le paired T-test pour la sélection du meilleur classificateur d'apprentissage. La validation de cette solution est faite sur un environnement de test conçu spécialement pour ce type de rootkits.

Mots clés : rootkit niveau noyau, apprentissage automatique, LTTng, noyau Linux.

KERNEL LEVEL ROOTKIT DETECTION BASED ON LTTNG

Tarek SLAIMIA

ABSTRACT

The malware detection mechanisms, based on tracing the user space activities, can be easily bypassed by the kernel level rootkits. This thesis proposes a solution to detect attack techniques used by this type of rootkits using the tracing of linux kernel activities with the LTTng tool. This solution is based on the analysis of static and dynamic data of the linux kernel. It is designed along two axes: the first one is the integration of rootkit detection techniques in the kernel modules of the LTTng tracer and the second one is the use of the generated traces of LTTng containing the kernel data in the detection approach based on machine learning technique. The second axis has undergone optimization of input vectors corresponding to the different valuesof attacks and still underwent a paired t-test for selecting the best machine learning classifier. The validation of this solution was made on a test environment specifically designed for this type of rootkits.

Keywords: kernel level rootkit, machine learning, LTTng, linux kernel, security, intrusion detection

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

LKM	Linux Kernel Module
VFS	Virtual File System
NOP	No Operation
LTTNG	Linux Trace Toolkit Next Generation
LTP	Linux Test Project
CTF	Common Trace Format
HPC	Hardware Performance Counters
SVM	Support Vector Machine
TMF	Tracing and Monitoring Framework also known as “Trace Compass”
ARFF	Attribute-Relation File Format
GPL	GNU Public Licence
kNN	k -Nearest Neighbors
MLP	Multi Layer Perceptron
IDT	Interrupt Descriptor Table

INTRODUCTION

1.1 Contexte

Au cours des dernières années, les attaques informatiques utilisent une variété de méthodes sophistiquées pour accéder au système, incluant les rootkits. L'objectif principal des rootkits est de cacher les preuves de présence des activités d'intrusion dans le système. La plupart de ces rootkits ont besoin des privilèges root pour s'installer dans le système. Habituellement, les attaquants exploitent certaines vulnérabilités du système pour obtenir les privilèges de niveau administrateur nécessaires. Il sera difficile pour un utilisateur ordinaire de détecter la présence d'un rootkit puisqu'il ne remarque pas une différence dans le comportement du système même si ce dernier est infecté par un rootkit.

Avec l'utilisation croissante des systèmes d'exploitation dans les téléphones intelligents et bien d'autres appareils embarqués, les menaces posées par les rootkits deviennent de plus en plus importantes. Les premières générations de rootkits étaient plus faciles à détecter, car ils visaient principalement à modifier les programmes de niveau utilisateur (les fichiers journaux, les fichiers binaires des applications, etc.). Des méthodes telles que la vérification de la somme de contrôle pourraient facilement détecter ces types d'infections. Les rootkits sont apparus à la fin des années 80 comme méthode utilisée pour cacher les fichiers journaux. Actuellement, ils constituent une menace sérieuse pour l'industrie informatique. Par conséquent, une multitude de travaux de recherche sont de plus en plus consacrés pour la détection des rootkits.

Par le passé, les administrateurs système se basaient essentiellement sur les utilitaires systèmes tels que ls, ps et bien d'autres pour détecter la présence de rootkits. Mais l'apparition de nouvelles générations de rootkits, qui peuvent facilement cacher leur présence à ces utilitaires, rend leur utilisation pour la détection obsolète. Les rootkits, de niveau noyau, modifient des sections critiques du système d'exploitation tel que son noyau et les pilotes de

périphériques. La difficulté à les détecter découle du fait qu'ils fonctionnent au même niveau de sécurité que le système d'exploitation (Joy et John, 2011).
Le travail présenté dans ce mémoire est réalisé dans le cadre de la détection des rootkits installés au niveau du noyau du système d'exploitation Linux. Ce mémoire de recherche focalise sur l'identification des mesures du noyau qui peuvent être utiles dans la détection des rootkits noyaux et ceci en étudiant leurs natures et en analysant leurs comportements dans le système.

1.2 Définition du problème

La problématique principale dans notre travail est : comment arriver à détecter la présence du rootkit niveau noyau à partir des structures, des fonctions et des appels systèmes du noyau du système d'exploitation et des autres mesures disponibles en utilisant le traceur LTTng?

Pour répondre à cette question de recherche, trois étapes successives d'études ont été effectuées :

- **Étape 1 :** Une revue de littérature qui traite de deux aspects principaux. Le premier aspect est une étude en ce qui a trait à la définition, à l'identification des différents mécanismes liés au noyau Linux, et, aux rootkits et à l'apprentissage automatique. Le deuxième est une étude qui vise l'état de l'art des différents techniques d'analyse qui peuvent être appliquées dans notre cas d'étude ainsi que les différentes approches de détection des techniques d'attaques utilisées par les rootkits;
- **Étape 2 :** Une identification de la méthode d'analyse la plus adéquate ainsi que la sélection des approches de détection de rootkits qui vont être proposées et validées par des expérimentations des attaques étudiées;
- **Étape 3 :** Une recherche en ce qui a trait à la représentation de meilleures approches de détection de ces attaques qui permettrait de tenir compte des différentes mesures utilisées.

Ces étapes de la recherche ont permis d'identifier des problématiques secondaires. Ces problématiques sont présentées avec plus de détail dans les trois sous-sections qui suivent :

1.2.1 Étape 1 :

a) Quels sont les mécanismes liés au noyau, au rootkit et à l'apprentissage automatique?

La première étape de notre recherche présentée dans ce mémoire focalise principalement sur l'identification, la documentation et la définition des différents mécanismes qui interagissent avec le noyau Linux ainsi que les mesures qui leurs sont associées, aussi les différents techniques et classes des rootkits et enfin les différents techniques liées à l'apprentissage automatique et plus précisément l'apprentissage supervisé. Vu la grande diversité des rootkits, la complexité des techniques utilisées par ces derniers devient de plus en plus importante. Par conséquent, la complexité des méthodes de détection augmente. L'une des méthodes les plus utilisées pour la détection d'activité malveillante est l'apprentissage automatique. Dans ce mémoire, nous présentons une introduction au domaine d'apprentissage automatique. Les techniques et les méthodes d'évaluation liées à l'apprentissage automatique sont variées et il faut fixer les mesures, les méthodes d'évaluations et d'optimisation des données utilisées, et par la suite, il faut également choisir la méthode de comparaison des algorithmes de classification qui va valider l'algorithme le plus adéquat.

b) Quelles sont les méthodes d'analyse et les approches de détections des rootkits?

Le deuxième aspect traité dans cette étape consiste à définir les différentes méthodes possibles pour analyser des malwares, à introduire l'aspect traçage et l'outil LTTng ainsi qu'énumérer les différentes approches possibles appliquées dans le domaine de détection de malwares.

1.2.2 Étape 2 : Comment sélectionner la méthode d'analyse et de détection de rootkits adéquate à l'outil LTTng ?

Dans cette étape, il est nécessaire de procéder à l'identification de la méthode d'analyse la mieux adaptée à l'ensemble des fonctionnalités offertes par LTTng. Cette méthode devra être capable de fournir l'ensemble des données et des mesures du noyau nécessaires pour la détection des rootkits. Par la suite, on effectue une identification des approches de détection à intégrer à LTTng. L'ensemble de ces approches doivent être bien conçues afin de respecter l'architecture du traceur LTTng. Les autres expérimentations sont faites à base des rootkits récents compatible avec les nouvelles versions du noyau Linux. Les scripts d'automatisation des expérimentations et les outils de déclenchement des appels systèmes sont spécifiés, et ceci, après avoir installé LTTng. La trace récupérée est ensuite analysée et formatée selon le format des données sélectionné et selon le format d'entrée de l'outil d'apprentissage automatique qui sera choisi. Donc, le besoin de développer un outil qui effectue ces tâches s'avère indispensable.

Ce mémoire propose la méthode d'analyse ainsi que les différentes approches de détection qui vont être intégrées au traceur LTTng et le rôle de ce traceur dans la détection par apprentissage automatique. Les attaques étudiées sont les suivantes :

- La modification des appels systèmes;
- Le détournement des appels systèmes;
- La dissimulation de processus;
- La duplication des tables systèmes;
- L'utilisation des appels systèmes obsolètes pour le contrôle des rootkits;
- L'activation du mode *promiscuous* de l'interface réseau.

Pour les différents scénarios d'attaque, les approches conçues pour être intégrées au traceur ainsi que les approches reliées à l'apprentissage automatique sont spécifiées. Les expérimentations et leur environnement ainsi que les différents outils sont mis en place pour être vérifiés et validés.

1.2.3 Étape 3 : Quel serait l'algorithme d'apprentissage le plus efficace dans la détection des attaques spécifiées par les expérimentations?

Le choix du meilleur algorithme d'apprentissage pour la détection de ces attaques vient après avoir fait des interprétations à plusieurs niveaux :

- Le premier niveau est d'évaluer la performance de chaque algorithme après l'avoir testé avec un nombre variable d'instances.
- Le deuxième niveau est d'évaluer sa performance après avoir appliqué à ces attributs la méthode d'optimisation à choisir et voir s'il y a un ensemble d'attributs minimal qui peut être pris en compte à la place de l'ensemble initial.
- Le troisième niveau est d'appliquer une méthode de comparaison standard entre les différents résultats de l'ensemble des expérimentations afin de choisir l'algorithme le mieux adapté.

Ainsi, les classificateurs, les méthodes d'optimisation et les méthodes de comparaison des classificateurs et leurs valeurs par défaut devraient, d'une manière ou d'une autre, dépendre de la technique d'apprentissage choisie. De ce fait, les outils d'évaluation et de validation sont soit spécifiques à la technique d'apprentissage à titre de méthode d'optimisation et de comparaison, soit d'ordre général, comme le fait de varier le nombre d'instances pour évaluer la performance du classificateur.

Lors de cette étape, il est nécessaire de faire l'étude, l'expérimentation et la proposition du meilleur algorithme de classification, parmi notre choix limité, pour la détection des techniques d'attaques effectuées lors des expérimentations. Finalement, nous discutons de la nécessité ou non d'appliquer la méthode d'optimisation par sélection des attributs choisie. Il est nécessaire d'écrire également la performance des différents classificateurs en variant son nombre d'instances.

1.3 Questions de recherche

Ce mémoire, présente une approche basée sur une diversité de techniques de détection des rootkits du niveau noyau en utilisant le minimum des ressources logicielles possible. On étudie aussi la possibilité d'intégrer ces différentes techniques dans l'outil de traçage LTTng. L'objectif visé par notre approche est de répondre aux questions suivantes :

1. Quelles sont les types de rootkits noyau et qu'elles sont les techniques qu'ils utilisent?
2. Quelles sont les mesures du noyau qui peuvent être utilisées pour la détection des rootkits?
3. Quelles sont les techniques d'analyse adéquate à la détection des rootkits noyau?
4. Peut-on intégrer des mécanismes de détection des techniques d'attaques utilisées par les rootkits dans le traceur LTTng à partir des données statiques et des données dynamiques du noyau Linux?
5. Peut-on détecter les techniques d'attaques utilisées par les rootkits avec l'aide de technique d'apprentissage automatique basé sur les données dynamiques du noyau et ce avec une haute performance? Si oui, comment alors établir cette approche et comment choisir le classificateur le mieux adapté?

Pour répondre à ces questions une méthodologie est mise en place et elle sera décrite à la section suivante.

1.4 Méthodologie de recherche

La méthodologie de recherche adoptée tout au long de cette recherche est présentée par le biais du cadre de Basili (Basili et Selby, 1991; Basili, Selby et Hutchens, 1986; Bourque, 2000) adapté au domaine de l'ingénierie logicielle. Ce projet de recherche a été élaboré en suivant les quatre phases suivantes :

- **Phase 1 – La définition :** dans cette phase on procède à l'identification, la précision de la problématique et la documentation des questions de recherche. Aussi à choisir la méthodologie expérimentale de recherche en élaborant les scénarios de test, les étapes de conception de solutions et les méthodes de validation de ces solutions proposées.

- **Phase 2 – La planification :** cette phase est basée sur plusieurs revues de littérature qui portent sur les sujets suivants :
    - Les mécanismes, les composants et les mesures du noyau Linux;
    - Les classes des rootkits et les techniques qu'ils utilisent pour contrôler le système;
    - Les types d'apprentissages automatiques, les mesures d'évaluation des classificateurs, les méthodes de sélection d'attributs et les méthodes de comparaison entre les algorithmes de classification.
    - Les techniques d'analyse des rootkits et le traçage;
    - les approches de détection des rootkits.

  L'objectif de cette phase est d'identifier les mesures du noyau, les techniques d'analyse et les approches de détection à utiliser.

- **Phase 3 – Le développement et opération :** cette phase contient la conception des modules de détection à intégrer à LTTng, la configuration et le développement des outils de filtrages et de conversion de traces et enfin la conception des expérimentations pour valider l'approche de détection basée sur l'apprentissage automatique. Ces différentes étapes sont faites à base des mesures et des techniques d'analyse fixée dans la phase précédente.

- **Phase 4 – L'interprétation :** Les résultats des expérimentations sont interprétés au cours de cette phase. Aussi, la réalisation d'un retour sur les questions de recherche définis dans section précédente a été faite ainsi que l'identification des contributions

de cette recherche. Et ceci, sans oublier de mentionner certaines perspectives de futures recherches.

1.5 Organisation du rapport

Ce mémoire se compose de quatre chapitres :

1) Dans le premier chapitre, nous présentons les notions de base de l'apprentissage machine, des composants du noyau du système d'exploitation Linux ainsi que des classes et des types des rootkits;
2) Le deuxième chapitre est consacré à la présentation des différentes approches d'analyse et de détection des rootkits;
3) Le troisième chapitre présente des approches de détection de rootkits, basées sur l'analyse de donnée du noyau, dont une partie sera intégrée dans le traceur LTTng et l'autre est validée par les expérimentations des techniques d'attaques utilisées par les rootkits;
4) Le quatrième, et dernier, chapitre est consacré à la validation de l'approche par apprentissage automatique et le choix du meilleur algorithme de classification, parmi notre liste, pour la détection des attaques étudiées.

Finalement, une conclusion de ce mémoire est présentée et une présentation de la synthèse des principales limites de notre approche et les perspectives de son extension dans un futur travail de recherche sont présentées.

CHAPITRE 1

LES ASPECTS FONDAMENTAUX LIÉS À LA DÉTECTION DES ROOTKITS

1.1 Introduction

La détection des rootkits est un domaine de recherche qui a subi beaucoup d'évolution ces dernières années. Ceci s'explique principalement par l'évolution des techniques d'infiltration et de communication utilisés par les rootkits qui ne sont pas détectés par les outils de détections présentes sur le marché. C'est donc une course entre les développeurs des logiciels malveillants et les développeurs des outils de détections qui s'est transformée en un vrai un défi (Preda et al., 2007).

Ce premier chapitre effectue une synthèse de la présentation des aspects fondamentaux portant sur les différents mécanismes constituant le noyau Linux, les différentes techniques et classes des rootkits ainsi que les différents aspects d'apprentissage machine et plus précisément la technique d'apprentissage supervisé.

1.2 Le système d'exploitation Linux

Cette section présente une synthèse de la littérature ciblée sur les différents aspects fondamentaux du système Linux et plus précisément ceux qui constituent son noyau, à savoir : la gestion de la mémoire, la gestion des interruptions, la gestion des processus, le système de fichier, les appels systèmes, etc. Finalement, cette section contient un descriptif de différentes mesures de performance du noyau Linux.

1.2.1 Le noyau Linux

Le noyau Linux a été initialement créé par Linus Torvalds en 1991. Linux est un système d'exploitation basé sur la famille des processeurs x86 d'Intel. Actuellement, Linux est un système d'exploitation en logiciel libre publié sous licence GPL et disponible sur plusieurs

plateformes matérielles. Il est développé par plusieurs groupes de personnes. D'autre part, le projet GNU fournit les applications liées au noyau et les programmes qui rendent le noyau Linux utilisable, tels que les systèmes de fichiers, les compilateurs, les binaires d'administration système, les environnements graphiques, les éditeurs, etc. Le noyau Linux moderne possède une architecture monolithique complétée par son soutien de modules, tel que le support de multiples systèmes de fichiers, et un modèle de processus multitâches léger implémenté sur un noyau non préventif.

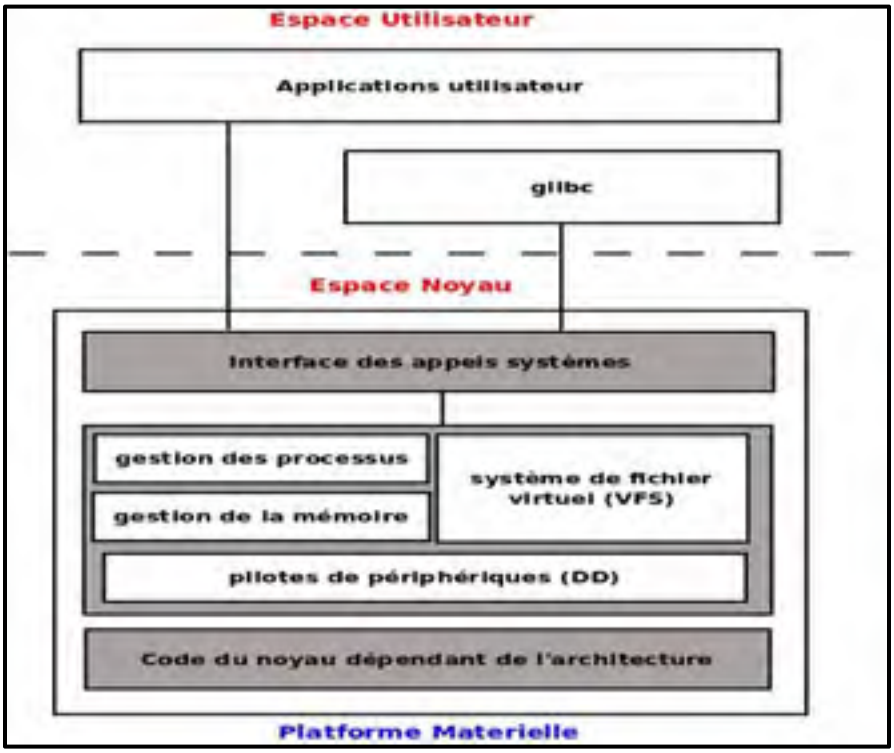

Figure 1.1 Architecture du système d'exploitation Linux
Tirée de Jones (2007)

Le noyau est l'élément responsable de la gestion des ressources matérielles du système (voir figure 1.1). Il effectue plusieurs tâches :

- La gestion de la mémoire : contrôle à la fois les sous-systèmes de mémoire virtuelle et physique. Les capacités de mise en cache du noyau sont cruciales pour la performance du système.

- La gestion des processus : y compris les deux modes d'exécution utilisateur et noyau. Les transitions entre ces deux modes et le modèle de processus de signalisation et d'autres mécanismes de communication interprocessus.

- La gestion de système de fichiers : y compris le système de fichiers virtuel, une couche d'abstraction et les implémentations réelles du système de fichiers comme : ext2, ext3, UFS, ISO9660.

- Les pilotes de périphériques : sont responsables de l'interaction avec chaque périphérique matériel comme le clavier, la souris, l'écran, les cartes réseau, les disques, etc. Le noyau devrait synchroniser toutes les interruptions reçues de la part de tous les composants du système.

- La pile réseau : implémente tous les protocoles, principalement dans le modèle TCP/IP, de la couche physique à la couche protocole (celle de TCP/UDP).

Ces tâches démontrent que le noyau est totalement dépendant du matériel. Il est seulement capable d'exploiter les fonctionnalités matérielles disponibles. La plupart du code du noyau Linux a été programmé en langage C, mais il y a de petites portions du code dépendant du processeur qui sont programmées en code assembleur (Pelaez, 2004).

1.2.2 Les modules du noyau Linux

Les modules noyaux sont des composants qui peuvent être chargés et déchargés dans le noyau sur demande. Ils étendent la fonctionnalité du noyau sans avoir besoin de redémarrer le système. Un exemple de module noyau est le pilote de périphérique qui permet au noyau d'y accéder. Pour étendre les fonctionnalités du noyau sans utiliser les modules, il faut construire des noyaux monolithiques et ajouter les nouvelles fonctionnalités directement dans l'image du noyau. Ceci rend la taille du noyau plus grande. En plus de la taille, il est

nécessaire de reconstruire et de redémarrer le noyau à chaque fois que nous voulons ajouter de nouvelles fonctionnalités (Salzman, Burian et Pomerantz, 2001).

1.2.3 Les appels systèmes

Les applications à l'espace utilisateur peuvent demander des ressources du système et ceci via les appels systèmes. Ces derniers sont des routines noyaux qui sont exécutés suite à une invocation d'un programme utilisateur et qui sollicitent des fonctions du noyau. Ils jouent le rôle d'un intermédiaire, car il n'y a pas de lien direct entre l'espace utilisateur et l'espace noyau. Cette interface de liaison fournit des APIs selon les différentes interactions qu'un programmeur souhaite implémenté. Des tables systèmes sont aussi utilisées pour établir ce lien entre les deux espaces utilisateur et noyau.

Suite à une demande d'un appel système de la part d'un programme utilisateur, l'interruption logicielle 80 est invoquée. L'application enregistre le numéro de l'appel système dans le registre eax tandis que les autres paramètres sont enregistrés dans d'autres registres du processeur. La routine associée à l'interruption 80 est exécutée à partir de la table des interruptions IDT que son adresse est sauvegardée aussi dans un registre (Yao, 2009). Cette routine est le gestionnaire des appels systèmes qui utilise le numéro de l'appel système pour déclencher la routine associée à cet appel système à partir de la table des appels systèmes SCT (MARTIN, PAULIAT et PELLETIER, 2005). La figure 1.2 présente le chemin d'exécution d'un appel système de l'espace utilisateur jusqu'à l'espace noyau :

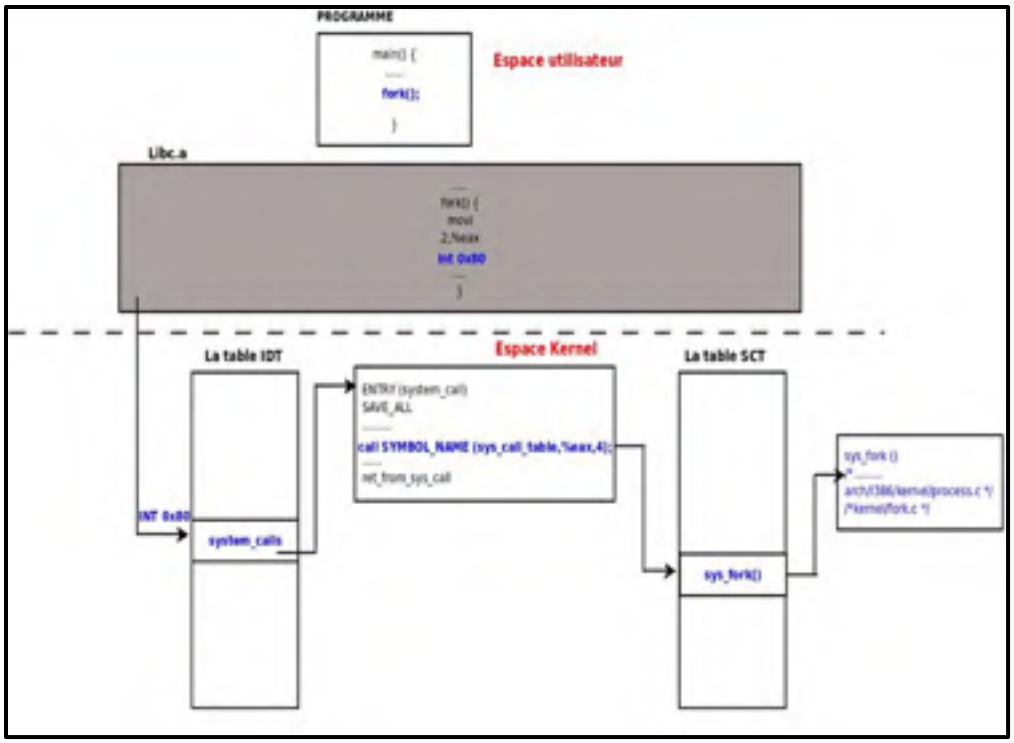

Figure 1.2 Déclenchement de l'appel système
Adaptée de YAO (2009)

1.2.4 Les kprobes

Kprobes est un mécanisme développé par les chercheurs d'IBM et a été introduit à la version du noyau Linux 2.6.9. Ce mécanisme permet l'exécution d'un gestionnaire du code arbitraire à n'importe quel point, durant l'exécution du noyau. Depuis que les gestionnaires des sondes sont écrits dans les modules du noyau, le privilège root s'avère indisponible pour exploiter ce service. Ce mécanisme est extrêmement utile pour le débogage de certaines parties spécifiques du noyau. De cette manière, le débogage sera effectué sans avoir besoin de recompiler et de redémarrer le système. Ceci présente un gain considérable de temps et du coût du développement. En plus, kprobes offre un meilleur support pour la traçabilité du noyau et une meilleure évaluation de ses performances (Ramaswamy, 2008).

Kprobes faisait partie de l'outil Dprobes qui est un outil de traçage de haut niveau. Ce mécanisme constitue une interface qui permet l'insertion dynamique des points d'instrumentation dans le noyau lors de son exécution.

Le principe de kprobes consiste à associer un événement à une adresse dans le code du noyau. Une fois l'exécution atteint cette adresse, l'événement sera déclenché. Kprobes permet d'associer une fonction à un événement. Cette fonction s'exécute à chaque déclenchement de l'événement qui lui est associé. kprobes présente trois types de points de traçage du noyau : les kprobes ordinaires, les kretprobes et les jprobes.

Les kprobes ordinaires s'insèrent dans n'importe quelle routine du noyau et elles peuvent associer deux fonctions pour chaque évènement. La première fonction est exécutée avant l'instruction, qui se situe à l'adresse du point d'instrumentation, elle est appelée pre_handler. La deuxième fonction est exécutée après cette instruction, cette fonction est appelée post_handler. Quand les deux fonctions sont appelées, une structure de type pt_regs, contenant une copie de l'état de tous les registres au moment où l'événement est déclenché, leur est passée. Ce type de structure change de contenu quand l'architecture change.

La spécificité des kretprobes est que ce type de kprobes est enregistré aux points de retour des fonctions.

Les jprobes, à l'inverse des kretprobes, ont la possibilité de s'associer aux points d'entrée des fonctions du noyau. Ceci a pour but de collecter les arguments de ces fonctions. Plusieurs outils récents, de traçage noyau, utilisent les kprobes tels que : Ftrace, LTTng et SystemTap (Fahem, 2012).

1.2.5 La gestion des processus sous Linux

La gestion des processus est considérée comme l'une des tâches les plus importantes d'un système d'exploitation. Son implémentation pour Linux est similaire à celle d'Unix. Elle inclut l'ordonnancement des processus, la gestion des interruptions, la gestion des signaux, la gestion des priorités des processus, leurs états et la gestion de leurs mémoires. Cette section, présente plus de détails concernant la gestion des processus sous Linux et son impact sur la performance du système (Ciliendo et Kunimasa, 2007).

1.2.5.1 C'est quoi un processus ?

Le processus est une instance d'exécution qui s'exécute sur un processeur. Il utilise toutes les ressources que le noyau Linux peut gérer pour achever sa tâche. Tous les processus qui s'exécutent sous Linux sont gérés par la structure de task_struct (voir figure 1.3). Cette structure contient toutes les informations nécessaires pour qu'un processus s'exécute, tels que : l'identification du processus, de ses attributs et des ressources qui le construisent (Ciliendo et Kunimasa, 2007).

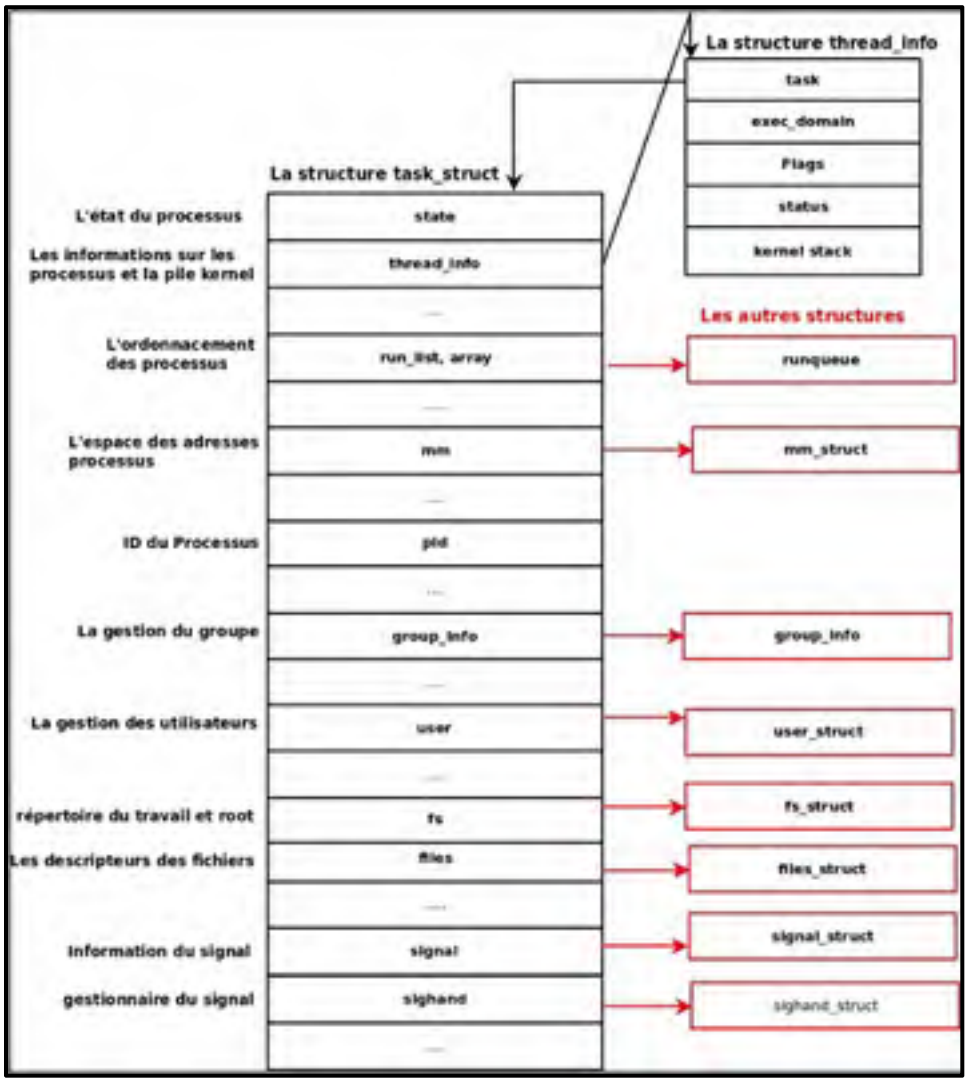

Figure 1.3 Schéma des structures de données reliées au processus
Tirée de Ciliendo et Kunimasa (2007)

1.2.5.2 Cycle de vie d'un processus

Chaque processus a son propre cycle de vie tel que la création, l'exécution, la terminaison et la suppression. Ces phases seront répétées des millions de fois tant que le système est en cours d'exécution. Le processus père déclenche un appel système fork () pour créer un processus fils. Quand un appel système fork () est exécuté, il obtient un descripteur du processus fils crée et détermine son id. Il copie les valeurs de descripteur du processus père au processus fils. L'espace d'adresse du processus père n'est pas entièrement copié. L'appel système exec() copie le nouveau programme à l'espace d'adresse du processus fils. Puisque les deux processus père et fils partagent le même espace d'adresse, une écriture des données du nouveau programme cause une exception de défaut de page. À ce stade, le noyau attribue la nouvelle page physique au processus fils. Après la terminaison de l'exécution du programme, le processus fils se termine avec un appel système exit(). Cet appel système libère la plupart des structures de données du processus fils et notifie le processus père de sa terminaison par un signal. Le processus à cet état est appelé processus zombie. Le processus fils ne sera pas complètement éliminé jusqu'à ce que le processus père soit notifié de la terminaison de tous ses fils. Ceci est accompli à l'aide de l'appel système wait(). Une fois qu'il sera notifié de la terminaison de tous ses fils il supprime toute la structure de données du processus fils et libère le descripteur de processus (Ciliendo et Kunimasa, 2007).

1.2.6 Le gestionnaire des interruptions

La gestion des interruptions est une des tâches les plus prioritaires d'un système d'exploitation. Les interruptions sont générées par les périphériques d'entrées/sorties du système. Le gestionnaire d'interruption informe le noyau Linux de l'interception d'un événement comme la saisie au clavier, arrivée de trame Ethernet et autres. Il indique au noyau d'interrompre l'exécution des processus et d'effectuer la gestion des interruptions aussi rapidement que possible parce que certains dispositifs nécessitent une réactivité rapide. Cela est essentiel pour la stabilité du système. Quand un signal d'interruption est intercepté, par le noyau, ce dernier doit passer du processus en cours d'exécution à un nouveau processus pour gérer cette interruption. Cela signifie que les interruptions provoquent le changement de

contexte. Un grand nombre d'interruptions pourraient causer une dégradation des performances du système. Il existe deux types d'interruptions dans les implémentations Linux. Le premier type est les interruptions matérielles. Une interruption matérielle est générée pour les dispositifs qui exigent une réactivité (E / S disque interruption, interruption de la carte réseau, clavier interruption, interruption de la souris). Les informations des interruptions matérielles sont localisées sous /proc/interrupts. Le deuxième type est les interruptions logicielles. Une interruption logicielle est utilisée pour des tâches dont le traitement peut être différé (c.à.d comme les opérations TCP / IP, les opérations du protocole SCSI, etc.). Dans un environnement multiprocesseur, les interruptions sont gérées par chaque processeur. Cette approche améliore la performance du système (Ciliendo et Kunimasa, 2007).

1.2.7 L'architecture mémoire Linux

Le noyau Linux attribue une partie de la zone mémoire à un processus pour son exécution. Ce dernier utilise la zone mémoire pour son espace de travail. Le nombre de processus en cours d'exécution atteint parfois des dizaines de milliers et la taille de mémoire est généralement limitée. Pour cela le noyau Linux doit gérer la mémoire de façon efficace (Ciliendo et Kunimasa, 2007).

1.2.7.1 Le gestionnaire de la mémoire virtuelle

L'architecture de la mémoire physique d'un système d'exploitation est généralement transparente pour l'application et l'utilisateur parce que les systèmes d'exploitation mappent tout genre de mémoire en mémoire virtuelle. Les applications n'allouent pas la mémoire physique, mais ils demandent une mise en correspondance de la mémoire d'une certaine taille, au noyau Linux, et ils reçoivent en échange une mise en correspondance dans la mémoire virtuelle. La mémoire virtuelle ne fournit pas nécessairement une correspondance dans la mémoire physique (Ciliendo et Kunimasa, 2007).

1.2.8 Les systèmes de fichiers Linux

Un des grands avantages de Linux est qu'il offre aux utilisateurs une variété de systèmes de fichiers supportés. Les noyaux Linux modernes peuvent soutenir presque tous les types de systèmes de fichier du système du fichier basique FAT aux systèmes de fichiers de haute performance comme JFS. Les systèmes de fichiers Ext2, Ext3 et ReiserFS sont des systèmes de fichiers Linux natifs pris en charge par la plupart des distributions Linux. Cette recherche se concentrera sur le VFS car in présente une des cibles principales des rootkits (Ciliendo et Kunimasa, 2007).

1.2.8.1 Le système de fichier virtuel VFS

Le système de fichier virtuel (VFS) est une couche qui constitue une interface d'abstraction entre les processus utilisateurs et les différents types d'implémentations de système de fichiers Linux. Le système de fichiers virtuel fournit des modèles communs d'objets (tels que les i-node, les fichiers objets, les pages de cache, etc.) et des méthodes pour accéder aux objets du système de fichiers. Il cache les différences de chaque implémentation de système de fichiers au processus utilisateur (voir figure 1.5). Les processus utilisateurs n'ont pas besoin de savoir quel système de fichiers à utiliser ou quel appel système devrait être délivré pour chaque système de fichiers (Ciliendo et Kunimasa, 2007).

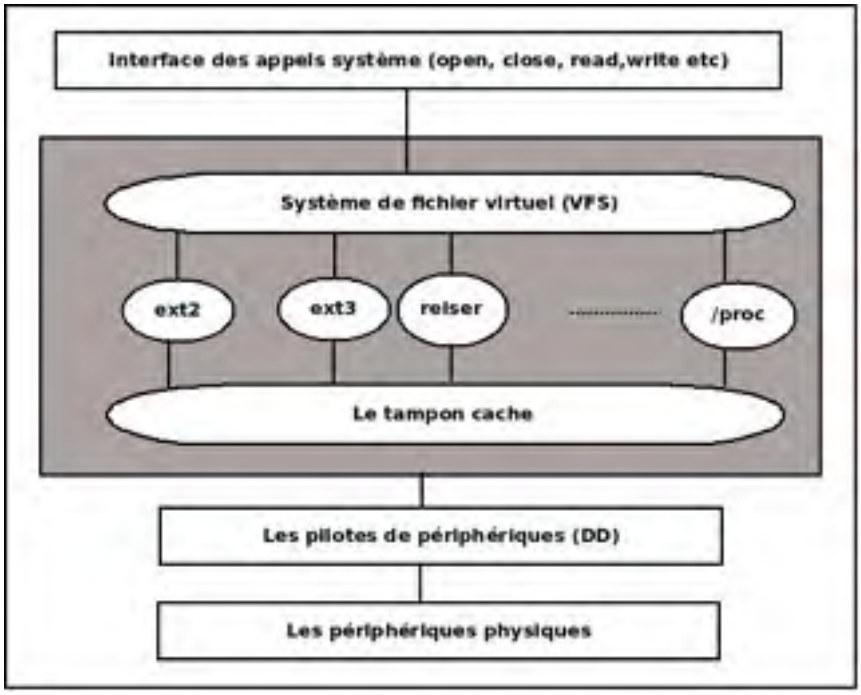

Figure 1.4 Le système de fichier VFS
Tirée de Jones (2007)

1.2.9 Les mesures de performances Linux

Un grand nombre d'outils de mesure de la performance sont disponibles sous Linux. Le choix d'un outil parmi eux dépendra des préférences de l'utilisateur, de la quantité de données et des détails souhaités.

Tous les outils de mesures de performances mesurent les mêmes caractéristiques. Dans les parties qui suivent, les détails des mesures les plus importantes du système sont présentés (Ciliendo et Kunimasa, 2007).

1.2.9.1 Les mesures de performances du processeur

Tableau 1.1 Les mesures de performances du processeur
Adapté de Ciliendo et Kunimasa (2007)

L'utilisation du CPU	Ce paramètre décrit l'utilisation globale du processeur.
Le temps utilisateur	Ce paramètre représente le pourcentage de CPU passé sur les processus de l'utilisateur.
Le temps système	Ce paramètre représente le pourcentage de CPU consacré aux opérations du noyau, dont le temps des IRQ et des softirq.
Waiting	La valeur totale du temps CPU passé en attendant qu'une opération d'E / S se produire.
Le temps Idle	Ce paramètre représente le pourcentage de CPU pendant lequel le système est en état idle en attente des tâches.
Le temps Nice	Ce paramètre représente le pourcentage de CPU passé sur le re-nicing des processus qui changent l'ordre d'exécution et la priorité des processus.
Load average	La moyenne du load n'est pas un pourcentage, c'est la moyenne du rolling de la somme de ce qui suit : - Le nombre de processus en file en attente d'être traité - Le nombre de processus en attente de tâche non interruptible à se terminer Autrement dit c'est la moyenne de la somme des processus TASK_RUNNING et TASK_UNINTERRUPTIBLE.
Runable processes	Cette valeur représente les processus qui sont prêts à être exécutés.
Blocked	Les processus qui ne peuvent pas s'exécuter pendant qu'ils sont en attente d'une opération d'E / S pour se terminer.
Context switch	Le taux de commutations entre les threads qui se produisent dans le système. Un grand nombre de changements de contexte avec un grand nombre d'interruptions peuvent signaler des problèmes de pilote ou d'application.
Interrupts	La valeur d'interruption contient les interruptions matérielles et softwares.

1.2.9.2 Les mesures de performances de la mémoire

Les mesures principales de la mémoire ainsi que leurs descriptions sont représentées au tableau 1.2 :

Tableau 1.2 Les mesures de performances de la mémoire
Adapté de Ciliendo et Kunimasa (2007)

Free memory	Le noyau Linux alloue de la mémoire non utilisée comme cache de fichiers systèmes. Donc, soustraire la quantité de tampons et le cache de la mémoire utilisée pour déterminer la mémoire libre.
Swap usage	Cette valeur représente la quantité d'espace swap utilisé. L'utilisation du swap indique seulement si Linux gère la mémoire de façon efficace.
Buffer and cache	Cache alloué comme système de fichiers et cache de périphérique bloc.
Slabs	Ce paramètre décrit l'utilisation de la mémoire du noyau.
Active versus inactive memory	Ce paramètre fournit des informations sur l'utilisation active de la mémoire système.

1.2.10 Le traçage

Le traçage est une technique de débogage qui consiste à enregistrer plusieurs types d'événements durant l'exécution d'un programme. Le développeur examine les informations contenues dans les événements collectés afin d'en déduire la cause du problème. Chaque évènement peut être constitué de plusieurs champs comportant chacune un type d'information spécifique comme : un libellé qui présente une information sur l'état actuel du système, l'heure à laquelle l'événement est survenu, des informations supplémentaires appelées aussi les arguments.

Les traceurs utilisent des approches de plus en plus optimisées pour le soutien continu de très hauts débits d'événements de l'ordre de milliers d'événements par seconde ou mêmes plus. Leur haute performance leur permet également l'enregistrement des événements de bas niveau à titre d'entrées dans les gestionnaires d'interruptions ou des appels systèmes. Les trois modules suivants forment le mécanisme de traçage : les fournisseurs de données, la collecte des données et l'analyse des données (Fournier, 2009).

Choix des outils de traces au lieu des débogueurs pour la détection des malwares :

Les logiciels malveillants sont devenus de plus en plus sophistiqués et dangereux. Ainsi l'analyse du programme binaire ou la rétro ingénierie est une tâche incontournable, mais très complexe en vue de comprendre le programme, afin de déterminer et de développer des contre-mesures. Cependant, de nombreux les logiciels malveillants mettent actuellement en œuvre des protections qui empêche la détection et l'analyse. Parmi ces protections, il y a ceux qui protège contre l'analyse statique tel que la technique de Binary code encryption et aussi contre l'analyse dynamique en utilisant des techniques telle que Breakpoints detection, Ptrace détection et timing check. Dans le cas d'analyse dynamique, l'utilisation des débogueurs pour surveiller les malwares est devenue de plus en plus obsolète, ce qui accélère l'intérêt pour les nous pousse à utiliser les traceurs pour effectuer cette tâche (Desfossez, Dieppedale et Girard, 2011).

L'obstacle majeur pour l'adoption d'un traceur est son impact en termes de performance. Le traceur LTTng satisfait les propriétés de faible impact sur la mise à l'échelle, le débit et la latence moyenne du système d'exploitation. Il est caractérisé aussi par son impact déterministe sur la réponse temps-réel, sa portabilité vers des architectures variées et un haut niveau de réentrance. Plusieurs outils de visualisation de la trace (peut être des Gigabits de données) sont développés pour la trace LTTng (Bornhofen, 2013; Desnoyers, 2009). La section suivante présente son architecture, ses avantages et les différents outils de visualisation de trace qui lui sont associés.

1.2.10.1 L'outil de traçage LTTng

LTTng a été écrit par Mathieu Desnoyers suite à la maintenance du logiciel prédécesseur appelé "Linux Trace Toolkit". Cet outil offre à la fois la possibilité de tracer l'espace utilisateur et l'espace noyau. Il prétend tracer avec un très faible impact sur le système. Les données de traçage peuvent être enregistrées directement sur le disque ou envoyées par le réseau. Le schéma suivant présente une vue d'ensemble de l'architecture de l'outil LTTng :

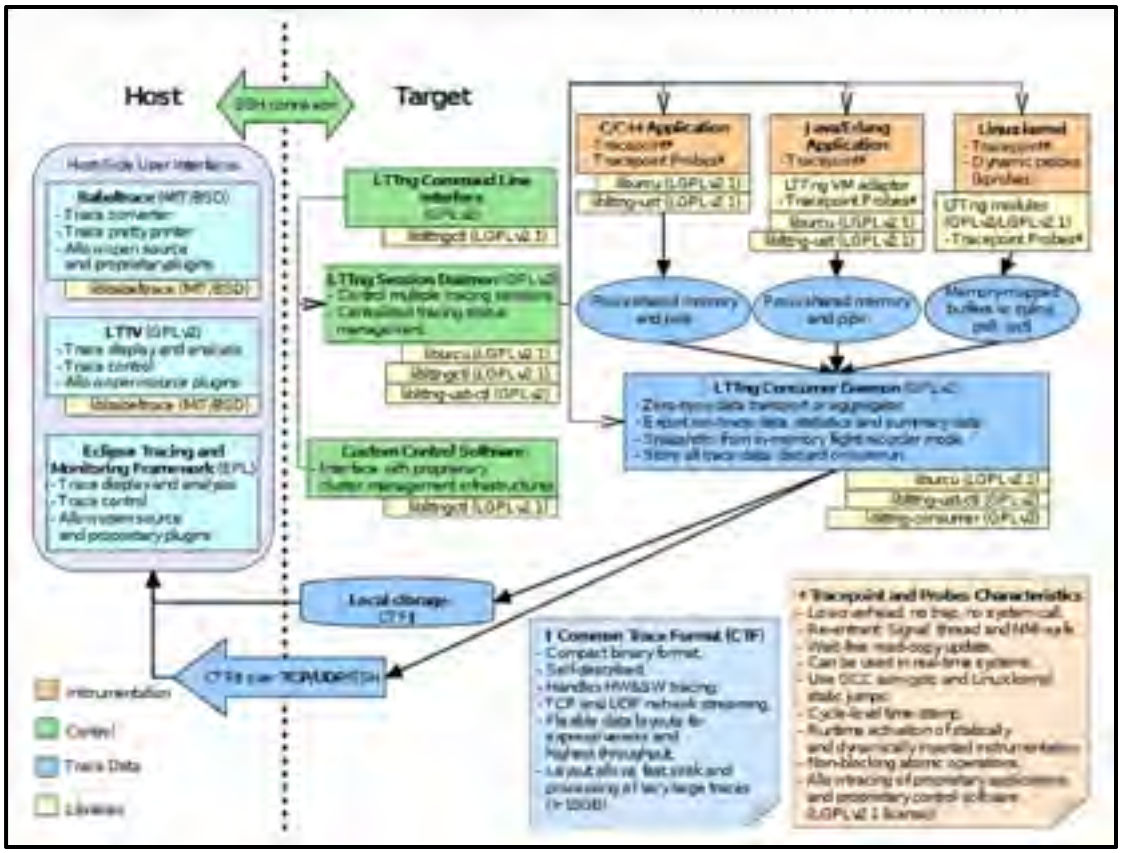

Figure 1.5 L'architecture de l'outil LTTng
Tirée de Vocking (2012)

Domaine d'utilisation :

LTTng est uniquement disponible pour Linux, il y a également un support complet pour les architectures : x86-32, x86-64, PowerPC 32/64, ARMv7, OMAP3, MIPS, sh, sparc64, s390 avec une précision du timestamp de 1 ns. D'autres architectures Linux sont supportées aussi, mais LTTng pourrait seulement avoir une précision du timestamp limitée sur ces derniers. Les Langages de programmation supportés par LTTng sont C et C ++, mais il peut être utilisé avec n'importe quel autre langage qui peut appeler du code C, par exemple il peut être utilisé avec Java ou Erlang via l'adaptateur VM de LTTng.

Fonctionnement de l'outil LTTng :

LTTng se compose de trois parties : la partie noyau qui contrôle le traçage de l'activité du noyau, l'application de ligne de commande de l'espace utilisateur nommé lttctl et le démon de l'espace utilisateur nommé lttd qui attend les données à écrire sur le disque. L'outil LTTng est un outil modulaire, il est constitué de cinq modules :

1. ltt-heartbeat qui génère des événements périodiques pour augmenter la base du temps monotonique.
2. ltt-facilities est une collection de types d'événements.
3. ltt-statedump génère des événements pour décrire l'état du noyau au moment du démarrage.
4. ltt-core gère un certain nombre d'événements de contrôle LTTng et contrôle donc ltt-heartbeat mentionné précédemment, ltt-facilities, et LTT-statedump.
5. ltt-base qui est un objet du noyau construit (built in kernel object en anglais) qui contient des symboles et des structures de données.

Tous ces modules travaillent ensemble pour assurer la surveillance du système en générant des données importantes pour le traçage. Le traçage est un moyen efficace de contrôler l'exécution d'un programme et d'enregistrer les données observées. Nous observons en particulier les événements du noyau du système d'exploitation tel que :

- Les appels systèmes
- Les demandes d'interruption
- L'ordonnancement des activités, et
- Les activités du réseau

Les événements peuvent avoir un nombre arbitraire d'arguments, donc la taille d'événements peut différer. Tous les événements sont ordonnés de façon précise sauf si le matériel rend cet

ordonnancement impossible. Les données capturées enregistrent l'événement ainsi que ses attributs et son *timestamp*. Il y a plusieurs façons de le faire, en voici une liste de trois :

1) Les tracepoints statiques :

Ils sont situés au niveau du code source et donc câblés dans un fichier binaire compilé d'un programme ou dans le noyau. Ils doivent être activés avant l'exécution.

2) Les points d'arrêt dynamiques :

Ils peuvent être insérés sans être recompilé via un appel système, un trap ou un saut dynamique (en anglais *jump*) et ajouté dans le noyau Linux via kprobe. Un point d'arrêt d'interruption provoque de gros surcouts. LTTng utilise les tracepoints pour enregistrer les données brutes qui sont analysées dans l'étape de post-traitement pour maintenir un *overhead* faible lors de l'exécution.

3) Les compteurs de performances :

Les compteurs de performances peuvent être activés via le paramètre contexte associé aux *tracepoints* du LTTng. Ce dernier implémente ces compteurs en utilisant l'API du noyau de perf nommée perf_event (Weaver, 2013). Il y a deux types de compteurs disponibles : les compteurs de performances matérielles et logicielles. La figure1.7 illustre l'architecture du perf_event (Gregg, 2015)

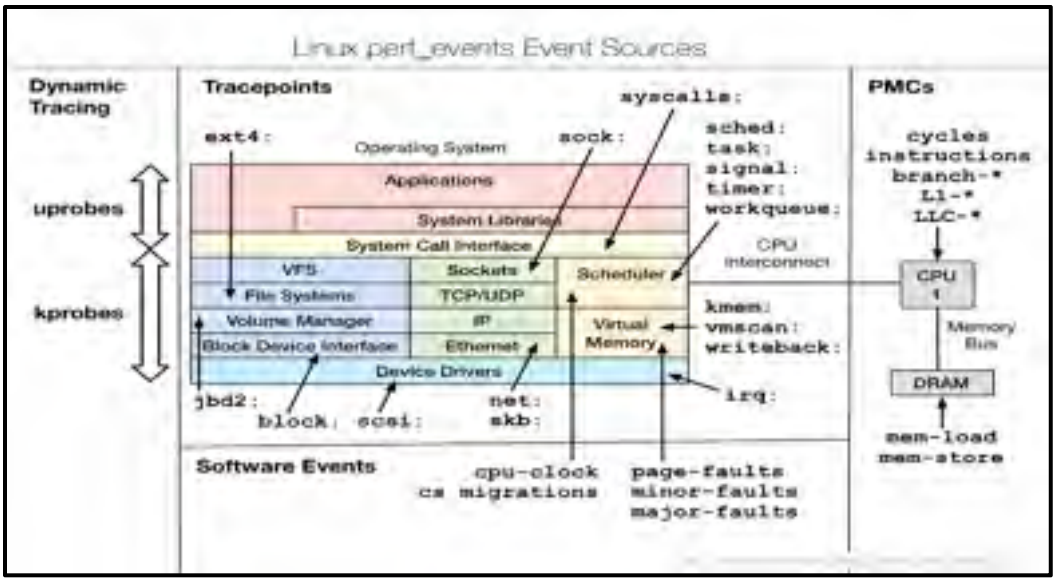

Figure 1.6 L'architecture de perf_event
Tirée de Gregg (2015)

Le tableau 1.3 présente une brève description des quelques compteurs de performances :

Tableau 1.3 Descriptif de quelques compteurs de performances
Adapté de Gregg (2015)

Nom de l'événement	**Description**	**Type**
Emulation faults	Nombre de *emulation faults* générées à cause des instructions non implémentées	logiciel
Alignment faults	Nombre d'*alignement faults :* lors d'accès à une *unaligned memory*	Logiciel
Cpu migrations	Nombre de migrations de CPU	Logiciel
Context switches	Nombre de commutation du contexte	Logiciel
Page faults, major faults, minor faults	Interruptions : nombre de défauts	Logiciel
Task clock, cpu clock	Temps total passé dans une tâche ou le CPU	Logiciel
Bus cycles	Nombre de cycles de bus	Matériel
Branch misses	Nombre de mauvaise prédiction de branchement	Matériel
Branches	Nombre de branchement	Matériel
Instructions	Nombre d'instructions	Matériel

Analyse et post-traitement de la trace :

La *tracedump* ne doit pas nécessairement être affichée dans le même environnement où les données ont été enregistrées. La sortie de trace est un fichier binaire autodescriptif et donc portable.

Babeltrace

Babeltrace offre des bibliothèques pour analyser des traces et l'afficher en texte lisible par l'utilisateur. Il est distribué par EfficiOS qui est géré par Mathieu Desnoyers, l'auteur et le mainteneur de LTTng. Babeltrace fonctionne avec des logs en format CTF et il n'offre pas d'interface graphique. Il est utilisé en la ligne de commande. Pour une représentation visuelle de la trace, l'outil LTT Viewer peut être utilisé.

LTT Viewer

Le LTT Viewer, ou tout simplement lttv, peut être utilisé pour afficher des traces enregistrées soit par le traceur de l'espace du noyau ou la trace de l'espace utilisateur. Il est écrit en langage C en utilisant glib et GTK. Cet outil est indépendant du traceur LTTng. Les développeurs peuvent facilement modifier lttv en développant des plugins. L'interface graphique de lttv offre des organigrammes de flux de contrôle et l'affichage des ressources.

Trace Compass

Il était précédemment connu sous le nom du plugin TMF. Via ce plugin, les traces peuvent être visualisées et analysées directement dans Eclipse. Ce plugin fait partie du projet Linux tools et peut être installé en utilisant le *built in* du gestionnaire de software dans éclipse. Il nécessite l'installation de la bibliothèque de lecture de trace LTTng. Il offre les fonctions suivantes :

- Flux de contrôle : visualise la transition d'état des processus.
- Ressources : visualise la transition d'état de ressources système.
- Statistiques : fournis des statistiques sur l'occurrence des événements.

L’impact de l’outil LTTng sur la performance

L'impact sur la performance de LTTng est raisonnablement petit. Dans un système chargé, le temps CPU ajouté par le traçage sous une moyenne et une forte charge varie de 1,54% à 2,28%. Seulement sous une très haute charge, l'impact est d’environ 9,46%.

Les domaines d'application

L’outil LTTng est utilisé par une variété d'entreprises pour le suivi de performances et du débogage. IBM l’utilise pour résoudre les problèmes dans les systèmes de fichiers distribués, Autodesk, pour résoudre les problèmes temps réel lors de développement d’application. Siemens l’utilise aussi pour le débogage interne et le suivi de performance de leurs systèmes. En outre, LTTng est inclus dans les packages de nombreuses distributions Linux comme Ubuntu, Montavista, Wind River, STLinux et Suse (Vöcking, 2012).

1.3 Les rootkits

Les logiciels malveillants, ou les malwares, sont classés en plusieurs types. Une des classifications intéressantes est celle proposée par Joanna Rutkowska (de Almeida, 2008). Elle a utilisé la taxonomie qui va être détaillée, par la suite, pour expliquer les différentes techniques des rootkits et leurs contremesures. Au lieu de classer les logiciels malveillants par catégorie, ce qui est généralement fait dans les différentes études de sécurité, (telles que les virus, vers, logiciels espions, les portes dérobées, etc.) elle a décidé de créer une classification imagée de malware présentée comme suit (de Almeida, 2008) :

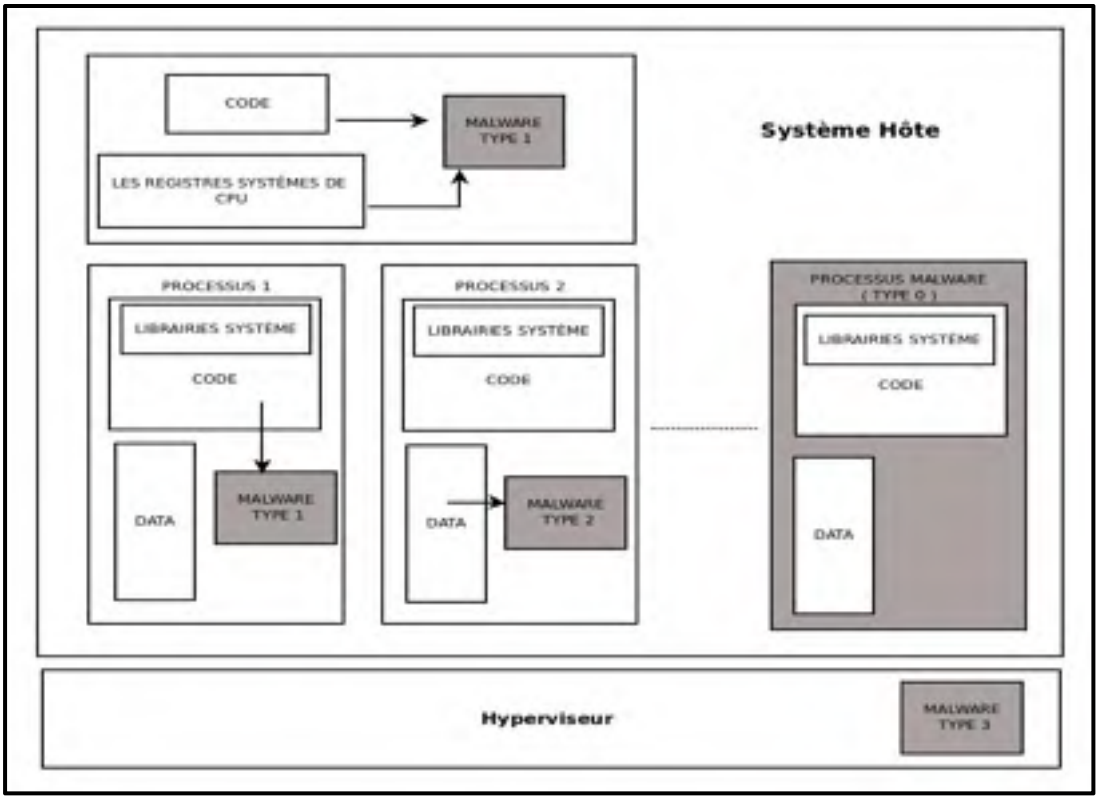

Figure 1.7 Classification des malwares
Tirée de Almeida (2008)

Les différents types de malwares sont décrits dans la figure 1.8. Ces types de malwares sont :

Malware type 0 : ce type de malware ne change pas le comportement d'aucune application exécuté dans le système. Ce type de malware ne peut être considéré comme rootkit que s'il utilise une des techniques d'infiltration utilisée par les rootkits pour cacher son activité.

Malware type 1 : ce type de malware change des sections de code comme celle contenue dans les fichiers exécutables ou dans le code du bios. Due à la nature de ces sections touchées, l'approche de vérification d'intégrité est la technique de détection la plus adaptée à ce type de malware.

Malware type 2 : ce type de malware modifie des sections dynamiques des processus exécutés et du noyau (fichier de configuration, les registres de données et les sections des données relatives aux processus exécutés, etc.). Ce type de malware est difficile à détecter.
Malware type 3 : ce type de malware qui est connu sous le nom rootkit hyperviseur représente une des technologies avancées des rootkits. Il ne change rien dans le système d'exploitation. Il opère comme une machine hôte et utilise les fonctionnalités de virtualisations du processeur. En interceptant tous les appels matériels. Ce type de rootkit est aussi très difficile à détecter (de Almeida, 2008).

1.3.1 C'est quoi un rootkit

Les rootkits sont définis comme un ensemble d'outils développés et utilisés par un attaquant une fois qu'une infiltration au système est faite et que l'attaquant dispose des privilèges nécessaires pour accomplir sa tâche (Elhadi, Maarof et Barry, 2013). Ils sont aussi définis comme un ensemble de modifications qui assurent le contrôle d'une machine hôte pour une période du temps nécessaire afin d'effectuer une tâche bien déterminée. Cet ensemble de modifications constituent le cœur de la différence entre un malware et un rootkit et ceci en changeant des composants du système tels que les programmes utilisateurs ou bien les fonctions et les structures du noyau (Lacombe, 2009).

1.3.2 La notion du ring

Les accès vers la mémoire et les ressources des systèmes d'exploitation sont effectués selon deux principaux modes : le mode utilisateur et le mode noyau. Le niveau du ring 3 contient les applications qui s'exécutent en mode utilisateur. Le niveau du ring 0 contient les fonctionnalités principales du noyau comme la gestion du processeur, de la mémoire et des entrées/sorties. Les processeurs récents permettent la virtualisation. Ce niveau est le niveau d'opération du mode hyperviseur. Toute application opérant dans ce niveau est décrite comme application appartenant au ring -1. Le plus haut degré de privilèges est accordé à ce niveau où il est possible d'exercer un contrôle complet des applications et des systèmes d'exploitation installés sur le PC (MARTIN, PAULIAT et PELLETIER, 2005).

1.3.3 Le cycle de vie d'un rootkit

Le cycle de vie d'un rootkit se résume généralement en quatre étapes essentielles décrites comme suit :

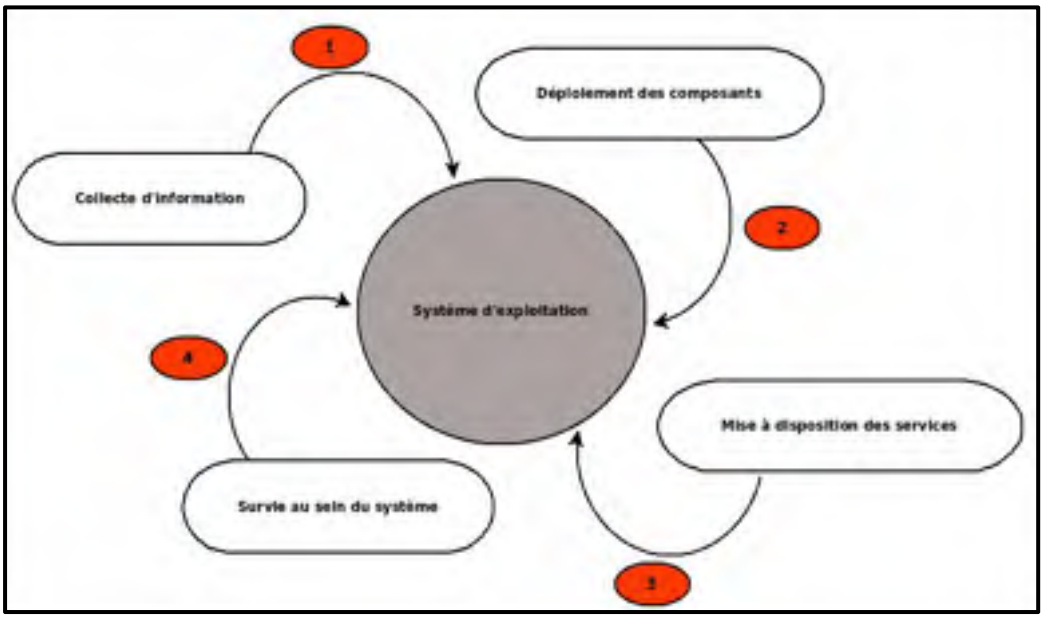

Figure 1.8 Cycle de vie d'un rootkit
Adaptée de Martin, Pauliat et Pelletier (2005)

1. La collecte d'information : Avant de débuter ces tâches, le rootkit procède à la collecte des informations qui lui seront utiles pour son exécution. Essentiellement ces informations dépendent des adresses des données qu'il souhaite modifier (les tables systèmes ou des routines spécifiques) ;
2. Le déploiement des composantes : Cette phase est marquée principalement par le chargement du rootkit dans le système soit par la technique LKM ou par injection directe de son code dans la mémoire. À ce stade, le rootkit effectue les modifications qu'il désire apporter au système hôte ;
3. La mise à disposition de services : Une fois les modifications sont apportées, les services qu'un attaquant souhaite activer sont alors disponibles et contrôlés par le biais des modifications apportées lors de l'étape précédente. Ces services sont soit

passifs (écoute, etc.) ou actifs (modification des données de la victime, etc.), tout dépend des tâches qu'un attaquant souhaite faire ;

4. La survie au sein du système compromis : Une fois exécuté sur la machine cible, le rootkit souhaite aussi persister, après redémarrage de la machine, et ceci en utilisant plusieurs techniques. L'une de ces techniques est de charger un script qui sera exécuté lors du redémarrage de la machine (MARTIN, PAULIAT et PELLETIER, 2005).

1.3.4 Les classes des rootkits

Les rootkits peuvent opérer principalement à cinq niveaux :

1) niveau application;
2) niveau librairie;
3) niveau noyau;
4) niveau firmware; et
5) niveau machine virtuelle.

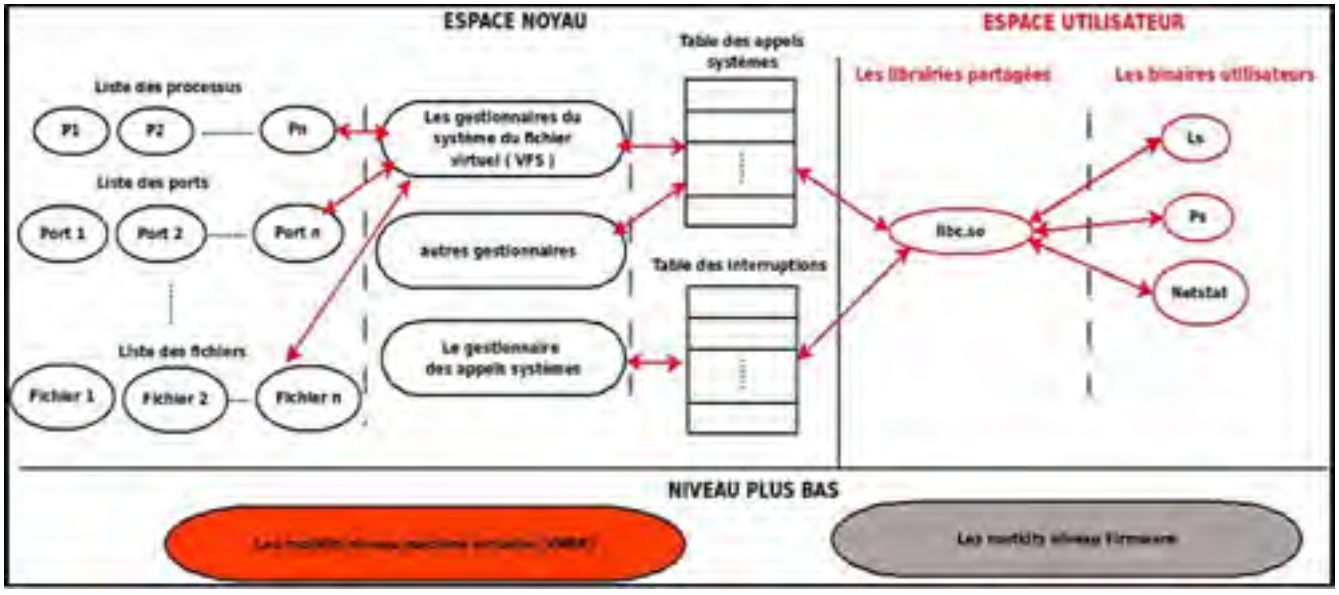

Figure 1.9 Les niveaux d'opérations des rootkits
Adaptée de Baliga (2009)

1) Rootkit niveau application (ring3)

Ce type de rootkit opère en remplaçant les binaires systèmes par d'autres recompilés avec du code malicieux qui permettant à un attaquant d'effectuer les taches qu'il désire faire (Shields, 2008). Un cas classique des rootkit niveau utilisateur est les utilitaires tels que ls ou ps modifiés qui permettent de cacher la présence de certains processus et fichiers. Les processus cachés sont généralement des malwares. Ce type des rootkits sont facilement détectés en utilisant les outils de vérification d'intégrité du système de fichiers (David et al., 2008).

2) Rootkit niveau librairie (ring3)

Les rootkits niveau librairie consistent à modifier ou remplacer les appels des librairies systèmes. Bien que ces rootkits résident en espace utilisateur, ils ne sont pas classés comme rootkits niveau utilisateur, et ceci à cause de la technique différente qu'ils utilisent pour effectuer l'attaque. À base du code source de certaines librairies, l'attaquant modifie ces sources dans le but de créer des versions de ces librairies qui sont adaptées à ses objectifs. Ces rootkits affectent plusieurs librairies bien que les modifications qu'ils apportent touchent seulement quelques librairies. Ces rootkits cachent la présence des malwares et de certains fichiers en modifiant les résultats des appels aux librairies systèmes (David et al., 2008).

3) Rootkit niveau noyau (ring0)

Les rootkits niveau noyau sont beaucoup plus difficile à détecter que les rootkits niveau utilisateur et librairie. Ce type de rootkit utilise des techniques telles que les modules noyau changeables et l'injection directe en mémoire noyau pour modifier les données du noyau du système d'exploitation. Ces données sont principalement les tables systèmes, les handlers des fonctions de ces tables et les données des structures systèmes. Quelques rootkits de ce type peuvent être détectés par les outils de vérification de la signature du code et des structures ou des données critiques du noyau. Ce type de détection peut être facilement neutralisé par l'utilisation de

nouvelle technique. L'une de ces techniques est l'utilisation des rootkits qui changent fréquemment leurs signatures (David et al., 2008).

4) Les rootkits niveau firmware

Suite à l'amélioration continue des techniques de détection des rootkits niveau noyau, les développeurs des rootkits ont commencé à explorer d'autres pistes (David et al., 2008). Une de ces pistes est l'installation du rootkit au niveau firmware. Ce genre de rootkit est implémenté au niveau matériel. En modifiant directement le code sur le matériel, un attaquant peut facilement embarquer un bout de code qui effectue certaines tâches qui ne seront pas faciles à détecter. Les cibles de ce genre de rootkit incluent les périphériques matériels, les contrôleurs de disques, les processeurs et la mémoire du firmware. L'idée derrière ce type de rootkit est que le firmware peut être modifié à partir du système d'exploitation directement. Cette idée est très intéressante puisque les rootkits firmware sont exécutés au démarrage avant l'exécution du système d'exploitation. Ce qui permet de faire l'accrochage des interruptions appelées par le système. Des actions comme la réinstallation du système d'exploitation ou le formatage du disque dur ne permettent pas d'enlever ce genre de rootkit (Shields, 2008).

5) Les rootkits basés sur la machine virtuelle (ring -1)

Cette technique représente une évolution et un passage de la complexité des rootkits à un niveau plus élevé. Ce genre de rootkit utilise la propriété des moniteurs de la machine virtuelle, qui est la transparence envers les systèmes d'exploitation invités, pour se cacher. Cette technique consiste à exécuter le système d'exploitation hôte existant dans une machine virtuelle. Cette virtualisation permet au rootkit de se cacher et d'être protégé contre les outils de détection installée sur la machine hôte. SubVirt a été le premier prototype qui démontre ce concept. Le support matériel pour la virtualisation, tels que Intel VT et la technologie SVM d'AMD, peut être aussi utilisé pour la construction des rootkits. Le rootkit Blue Pill exploite le support

matériel de la virtualisation pour améliorer sa dissimulation et réduire l'*overhead* au niveau de la performance et de l'empreinte mémoire. Néanmoins, plusieurs approches de détections basées sur les écarts entre le matériel virtuel et matériel physique ont été proposées pour révéler la présence de VMM et VMBRs (David et al., 2008).

1.3.5 Les méthodes d'injection en espace noyau

Utilisation des modules noyaux

Cette méthode est utilisée pour l'insertion des modules noyaux dans l'espace noyau. Elle nécessite que le support LKM par le noyau soit activé (Lacombe, Raynal et Nicomette, 2007). Ce support permet l'insertion et la suppression des modules noyaux en cours d'exécution, ce qui permet aux utilisateurs de modifier les fonctionnalités du noyau sans avoir besoin de recompiler et de redémarrer le système.

Au début de sa création, cette technique avait pour but de faciliter la manipulation des pilotes de périphériques, des pilotes du système de fichier, les appels systèmes, etc. Les modules noyaux sont insérés, en utilisant la commande insmod qui est un utilitaire qui insère les modules noyaux, peu importe sa localisation ou l'utilisation de la commande modprobe. Cette dernière nécessite que les modules à insérer soient localisés dans le répertoire / lib/modules. Les attaquants ont bénéficié de cette simplicité d'utilisation du support LKM pour cacher leurs activités malicieuses, mais avant ça ils doivent obtenir le privilège root.

La nécessité de localisation des modules noyaux dans le répertoire /lib/modules constitue le point faible de l'utilisation de l'utilitaire modprobe. Ces modules seront facilement détectés. Pour cette raison, les attaquants préfèrent l'utilisation de l'utilitaire de l'insmod. D'autres utilitaires du support LKM peuvent divulguer des informations importantes sur le système. Lsmod liste les modules noyaux en cours d'exécution sur un système (Vandeven, 2014).

Accès à la mémoire noyau via le périphérique virtuel /dev/kmem et via dev/mem

Cette méthode consiste à utiliser le fichier spécial /dev/kmem pour l'injection des rootkits noyaux. Ce fichier pointe vers une image de l'espace mémoire du kernel en cours d'exécution. Un attaquant peut modifier en utilisant cette technique le noyau en cours d'exécution une fois que le support /dev/kmem est activé (Vandeven, 2014). Ce genre de technique ne persiste pas lors d'un redémarrage du système. Cette technique peut être efficace dans des systèmes où leurs redémarrages sont peu fréquents comme les serveurs. Un des exemples de rootkits le plus connu, qui utilise cette technique, est le rootkit SuckIT (Super user control kit). Ce rootkit remplace la référence à la table des appels systèmes par une autre qui pointe vers une nouvelle table qui contient les appels systèmes contenant le code malicieux. La table originale des appels système reste chargée en mémoire sans subir aucune modification. Cette approche rend la détection de ce genre de rootkit très difficile (Shah et Giffin, 2008). L'utilisation d'accès à la mémoire physique dev/mem, pour charger le rootkit, est similaire à celui de /dev/kmem. Mais ce fichier spécial représente non seulement la mémoire du noyau, mais il représente aussi l'image entière de la mémoire physique (Lineberry, 2009; Vandeven, 2014).

Exploitations des failles du noyau

Certaines failles du noyau permettent l'injection des rootkits pour détourner les mécanismes de sécurités du noyau. Mais cette approche reste dépendante de la version du noyau affecté par la faille exploitée (Lacombe, Raynal et Nicomette, 2007).

Accès au contrôleur de la Mémoire physique sans intervention du processeur

Cette technique exploite les périphériques qui accèdent directement aux contrôleurs de la mémoire sans l'intervention du processeur (l'accès DMA). Un exemple de périphérique qui utilise cette technique est le bus firewire qui peut lire ou injecter les données en mémoire (Lacombe, Raynal et Nicomette, 2007).

1.3.6 Les méthodes du détournement

La duplication des tables systèmes

L'attaquant duplique l'une des tables système (SCT, IDT ou autre) dans l'espace mémoire du noyau. Ensuite, il la modifie en y incluant les fonctionnalités malveillantes qu'il souhaite installer au système. Puis il change l'adresse d'appel de la table originale avec la nouvelle adresse de sa copie.

Les méthodes de détection qui se basent sur les données statiques ne sont pas efficaces pour la détection de ce type d'attaque. Dans le cas de la table des interruptions, l'adresse de la copie est chargée auprès du processeur. Tandis que pour l'adresse de la table des appels systèmes, elle est chargé à partir du gestionnaire des appels systèmes dans la table des interruptions (Lacombe, Raynal et Nicomette, 2007).

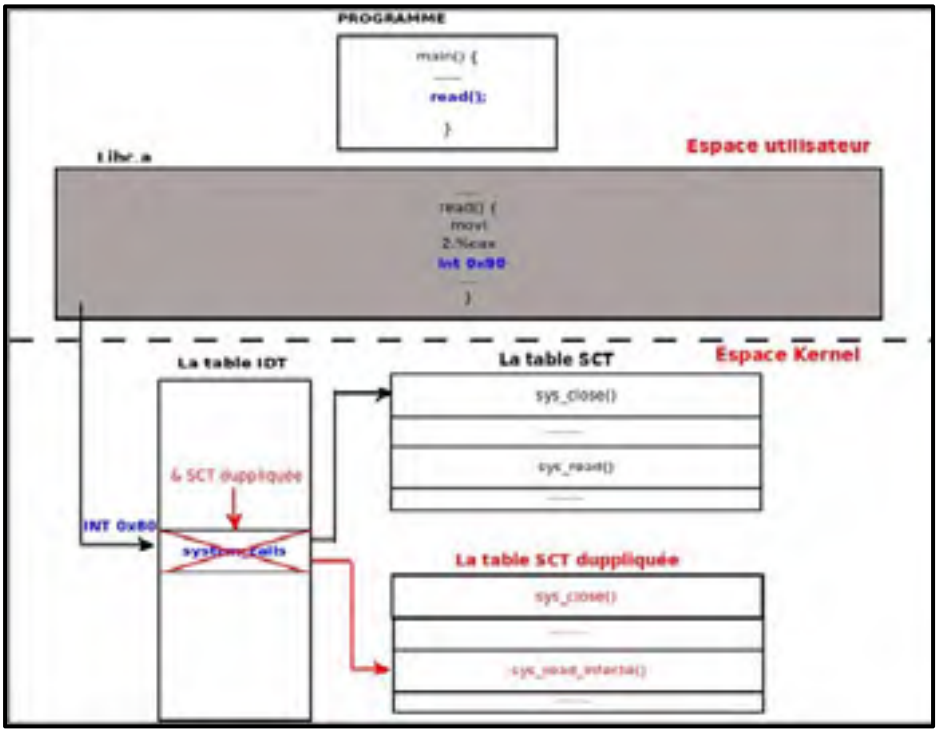

Figure 1.10 Duplication de la table système SCT
Tirée de Yao (2009)

Le détournement des appels systèmes

Après que le rootkit soit injecté en espace noyau, il change les adresses des appels systèmes dans la table SCT par les adresses des fonctions malveillantes. Le même comportement est effectué avec la table IDT pour les attaques qui visent les interruptions des adresses des fonctions originales sont réécrites à nouveau dans les tables correspondantes une fois que la fonctionnalité malveillante est exécutée (MARTIN, PAULIAT et PELLETIER, 2005).

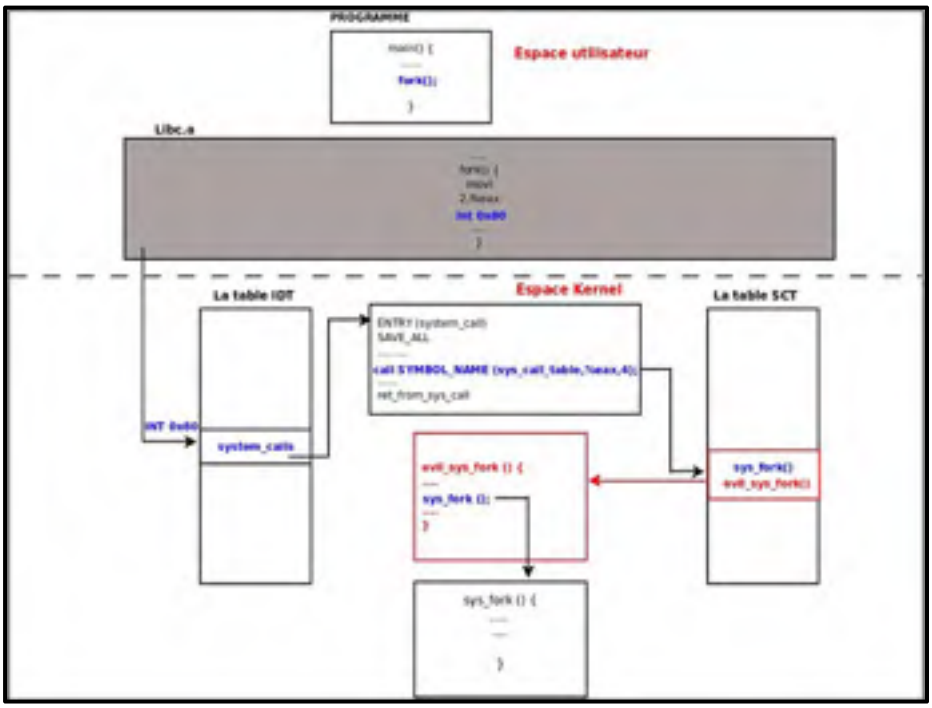

Figure 1.11 Détournement des appels systèmes
Tirée de Yao (2009)

La modification des pointeurs de fonctions du VFS

Les rootkits visant le système des fichiers virtuels cherchent à cacher des processus ou des fichiers spécifiques aux utilisateurs du système d'exploitation.

Cette attaque est faite en changeant les routines responsables de lister les fichiers systèmes ou les processus avec leurs propres routines. Le rootkit adore-ng appartient à cette famille de rootkits.

Il redirige la référence d'appel de la fonction proc_root _lookup à un appel de fonction malicieuse qu'il crée lui-même (Shah et Giffin, 2008) .

La modification des structures de données systèmes

La structure de données la plus visée par les rootkits est la structure task_struct. Chaque processus dans le système est représenté par cette structure. Pour faire l'ordonnancement des processus, deux listes chainées doivent être manipulées qui sont all_tasks list et run_list. La liste all_tasks est la liste de tous le processus système. Elle est alimentée lors de la création d'un nouveau processus. Quand un processus sera prêt à s'exécuter, sa structure task_struct correspondante est placée à la liste run_list. L'ordonnanceur sélectionne périodiquement des processus à partir de la liste run_list et les ordonnances afin d'être exécutés. Une fois qu'un processus finit son exécution, l'ordonnanceur le supprime de la liste run list. Dans l'espace utilisateur de Linux, des utilitaires comme ps consultent la liste all_tasks pour identifier la liste des processus systèmes. Ce qui n'est pas le cas pour l'ordonnanceur, ce dernier utilise la liste run_list. Pendant l'exécution, un processus qui est un élément de la liste d'exécution run_list aura également une entrée correspondante dans la liste des all_tasks. En conséquence, tous les processus en cours d'exécution seront également affichés dans la sortie de la commande ps (Pelaez, 2004; Riley, Jiang et Xu, 2009).

L'écart entre la vue de l'ordonnanceur et la vue de l'espace utilisateur des processus peut être exploité pour cacher des processus malveillants en espace utilisateur de Linux. Ce type de système d'exploitation à une architecture extensible dans lequel les modules noyau et les pilotes de périphériques ont un accès illimité à l'écriture et lecture des structures de données du noyau. Petroni et al. (2006) ont utilisé cette spécificité pour l'implémentation d'un rootkit qui se charge tel que module noyau. Ce module crée un nouveau processus et l'insère

seulement dans la liste run_list. Aucune entrée correspondante à ce processus ne sera créée dans la liste all_tasks. Ce processus est ordonnancé pour l'exécution, mais il ne sera pas visible par les outils qui affichent la liste des processus dans l'espace utilisateur. Ce processus est dédié à effectuer des activités malveillantes (Pelaez, 2004; Riley, Jiang et Xu, 2009).

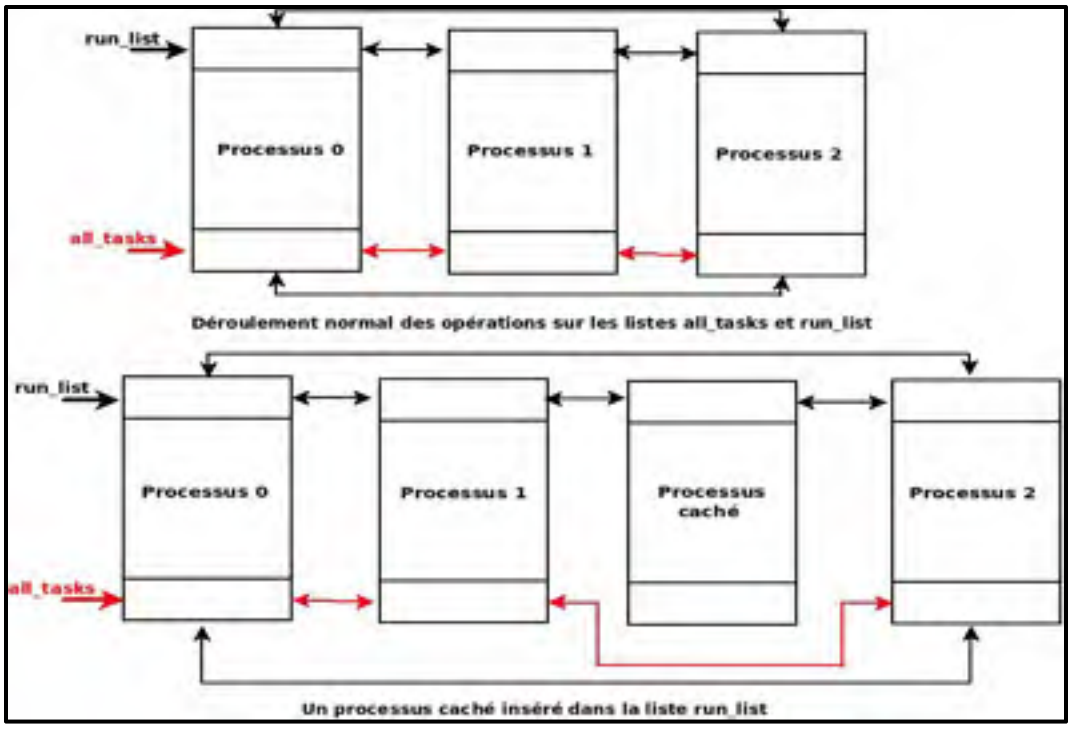

Figure 1.12 Illustration de l'état normal et de l'état contenant un processus caché
Adaptée de Petroni Jr et al. (2006)

La technique de modification des appels systèmes

Cette technique consiste à réécrire une partie du code de l'appel système continu dans la table des appels systèmes. Cette dernière ne subit aucune modification. Le rootkit utilisant cette technique réécrit quelques premières instructions de l'appel système en mettant une instruction jump (comme on peut utiliser aussi un push puis un ret ou un call) qui redirige l'exécution à un code malveillant. Cette technique peut être détectée en comparant les octets d'opcode courant avec les octets opcode normaux de l'appel système (Levine, Grizzard et Owen, 2006).

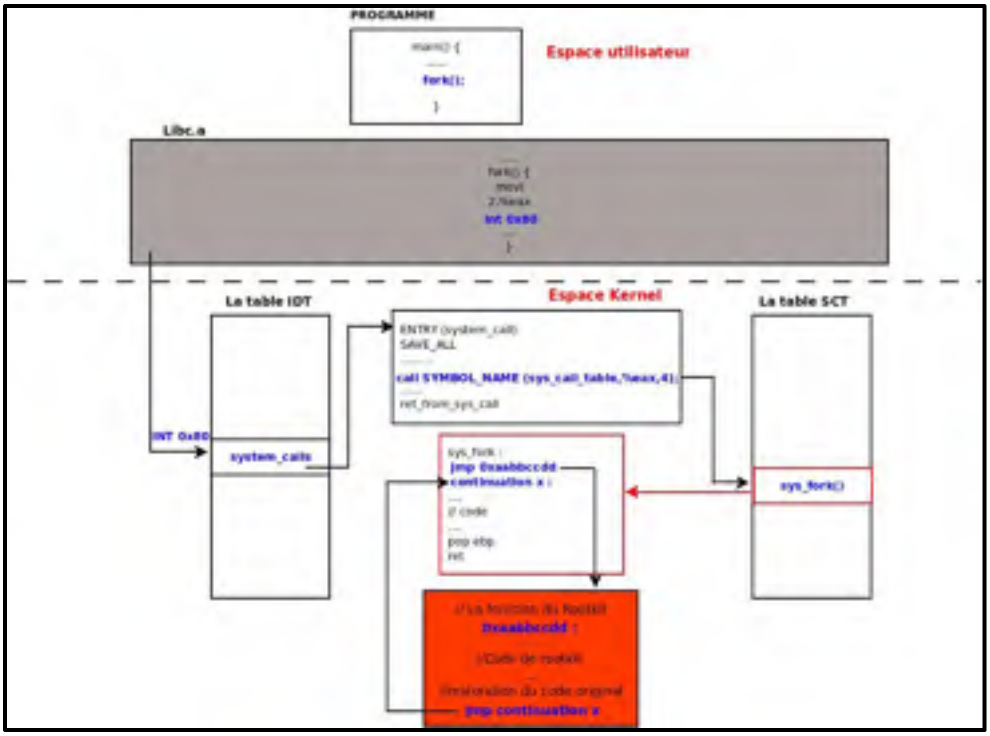

Figure 1.13 Modification d'un appel système
Tirée de Yao (2009)

L'implantation des rootkits au niveau du secteur de démarrage de la machine

À ce niveau, l'attaquant prend le contrôle au démarrage du matériel avant le système d'exploitation. Ce type d'attaque est effectué via des rootkits tel que BootRoot et Bootkit (Lacombe, Raynal et Nicomette, 2007).

L'implantation des rootkits au niveau hyperviseur

Les rootkits installés au niveau hyperviseur reposent sur les technologies de virtualisation matérielles. Le premier exemple de rootkit hyperviseur est BluePill qui attaque les processeurs AMD. Ce rootkit utilise l'extension matérielle de virtualisation de ce type du processeur. Dans ce cas le système d'exploitation installé dans le matériel est censé ne pas utiliser la virtualisation matérielle. Le rootkit à ce niveau va prendre le rôle d'hyperviseur auprès du processeur et placer le système d'exploitation sur la machine virtuelle pour le contrôler (Lacombe, Raynal et Nicomette, 2007).

1.3.7 Méthodes de communication avec un rootkit

Quand un attaquant effectue une intrusion, il a tout intérêt à établie une communication avec le système cible. Cette communication peut être faite à trois phases principales de l'attaque. La première phase est quand l'attaquant réussit à compromettre la sécurité du système cible. La deuxième phase est celle du transfert du rootkit pour l'installer. La troisième phase est celle de l'envoi des instructions de la part de l'attaquant et de la récupération de leurs résultats (Lacombe, Raynal et Nicomette, 2007).

1.4 Les machines d'apprentissage

L'apprentissage machine a toujours été connu comme un ensemble de techniques puissantes de fouille de données et de découverte de la connaissance. Cet ensemble de techniques recherche et décrit des patterns structurels utiles dans les données. En 1992, Shi a défini l'apprentissage machine comme l'étude qui permet aux machines d'acquérir de nouvelles connaissances, de nouvelles compétences et de réorganiser les connaissances existantes. Une machine d'apprentissage a la capacité d'apprentissage automatique à partir des expériences et de l'amélioration de sa base de connaissances. L'apprentissage machine possède une vaste gamme d'applications tels que les moteurs de recherche, les diagnostics médicaux, la reconnaissance de l'écriture manuscrite, le marketing et les diagnostics des ventes, etc. En 1994, l'apprentissage machine a été utilisé pour la classification des flux internet dans le cadre de la détection d'intrusion. L'apprentissage machine prend en entrée la forme d'un ensemble d'instances de données (ou exemples). Chaque instance est caractérisée par les valeurs de leurs attributs qui mesurent les différents aspects d'une instance. L'ensemble de données est finalement présentés en une matrice d'occurrences par rapport à des caractéristiques. Le résultat de la machine d'apprentissage est la description de la connaissance qui a été apprise. Le résultat spécifique du processus d'apprentissage est représenté (la syntaxe et la sémantique) dépend en grande partie de l'approche d'apprentissage utilisée (Nguyen et Armitage, 2008).

1.4.1 Les types d'apprentissages

Witten et Frank ont défini quatre types basiques d'apprentissages :

• La classification (ou apprentissage supervisé) : L'apprentissage par classification implique un apprentissage machine à partir d'un ensemble pré-étiqueté d'exemples, à partir duquel il construit un ensemble de règles de classification (appelé aussi un modèle) pour classer les exemples invisibles ;
• L'apprentissage non supervisé (ou apprentissage par clustering) : L'apprentissage non supervisé est le regroupement des instances qui ont des caractéristiques semblables en groupes, sans aucune orientation préalable ;
• L'apprentissage par association : Dans l'apprentissage par association, n'importe quelle association entre les caractéristiques est demandée ;
• L'apprentissage par prévision numérique : En prévision numérique, le résultat à prédire n'est pas classe discrète, mais une quantité numérique (Nguyen et Armitage, 2008).

1.4.2 L'apprentissage supervisé

L'apprentissage supervisé crée des structures de connaissances qui ont pour tâche de classer les nouvelles instances dans les classes prédéfinies (Barros et al., 2013). La machine d'apprentissage est fournie avec une collection d'instances de l'échantillon qui est préclassifié en classes. Le résultat du processus d'apprentissage est un modèle de classification qui est construit en examinant et en généralisant les données fournies. En effet, l'apprentissage supervisé se concentre sur la modélisation des relations d'entrée / sortie. Son objectif est d'identifier une correspondance à partir des caractéristiques d'entrée à une classe de sortie.
Les connaissances acquises peuvent être présentés sous forme d'organigramme, un arbre de décision, des règles de classification qui peuvent être utilisé plus tard pour classer une nouvelle instance invisible. Il y a deux grandes phases en apprentissage supervisé :

• La phase du *training* (ou formation) : C'est la phase d'apprentissage qui examine les données fournies (appelé l'ensemble de données de formation) et construit un modèle de classification ;
• La phase du *test* (ou classification) : C'est la phase où le modèle qui a été construit dans la phase du *training* est utilisé pour classer les nouvelles instances invisibles. Par exemple, TS un ensemble de données de *training* constitué d'un ensemble de paires d'entrée / sortie $TS = \{\langle x1|y1\rangle, \langle x2|y2\rangle, \dots, \langle xn|yn\rangle\}$ Où xi est le vecteur des valeurs des caractéristiques d'entrée correspondant à i^{ème} instance, et yi représente la valeur de la classe de sortie.

L'objectif de la classification peut être formulé comme suit : à partir d'un ensemble de données de formation TS, trouver une fonction $f(x)$ des caractéristiques d'entrée qui fournit les meilleures prédictions des résultats de la classe de sortie Y pour toutes les nouvelles valeurs invisibles de x. La sortie prend sa valeur à partir d'un ensemble discret $\{y1, y2, \dots, yn\}$ qui se compose de toutes les valeurs prédéfinies de la classe. La fonction $f(x)$ est la base du modèle de classification. Le modèle créé pendant la phase du training (la formation) est amélioré si nous fournissons simultanément les exemples d'instances qui appartiennent à la classe d'intérêt et les instances connues pour ne pas être membres de la classe d'intérêt. Cela permettra d'améliorer la capacité du modèle à identifier les instances appartenant à la classe d'intérêt.

Il existe un grand nombre d'algorithmes de classification de l'apprentissage supervisé et chacun d'entre eux diffère dans la façon de construire de son modèle de classification et de l'algorithme d'optimisation utilisé dans la recherche d'un modèle fidèle à la réalité (Nguyen et Armitage, 2008).

1.4.3 Les mesures d'évaluation et de la classification

Les différentes mesures d'évaluations sont dérivées de la matrice de confusion. Ci-dessous, un tableau représentant la matrice de confusion correspondant à un problème de classification binaire (les deux classes sont une classe positive et autre négative) :

Tableau 1.4 La matrice de confusion à deux classes
Tirée de Sayad (2010)

Matrice de confusion		Cible			
		Positive	Négative		
Modèle	Positive	a	b	Positive predictive value	a / (a + b)
	Négative	c	d	Negative predictive value	d / (c + d)
		Sensitivity	Specificity	**Accuracy** = (a + b) / (a + b + c + d)	
		a / (a + c)	d / (b + d)		

Une manière populaire de caractériser la précision d'un classificateur est d'utiliser des paramètres connus sous le nom de False Positives ou False Negatives, True Positives et True Negatives (qui sont respectivement les valeurs c, d, a et b dans la matrice de confusion représentée ci-dessus). Ces mesures sont définies comme suit :

• False Negatives (FN) : Pourcentage des membres de la classe X classé par erreur comme membres n'appartenant pas à cette classe.

• False Positives (FP) : Pourcentage de membres des autres classes classés comme membres appartenant à la classe X.

• True Positives (TP) : Pourcentage des membres de la classe X correctement classé comme appartenant à la classe X (équivalent à 100% - FN).

• True Negatives (TN) : Pourcentage de membres des autres classes correctement classé comme membres n'appartenant pas à la classe X (équivalent à 100% - FP).

Un bon classificateur vise à minimiser FN et FP. Certains travaux utilisent le paramètre précision (*Accuracy* en anglais) comme une mesure d'évaluation. La précision est généralement définie comme le pourcentage d'instances correctement classées parmi le nombre total d'instances. La littérature du domaine utilise souvent deux mesures supplémentaires appelées Recall, F-measure et Precision. Ces paramètres sont définis comme suit:

• F-measure= (2 × recall × precision)/ (recall + precision)

• Sensitivity × specificity = (TP / (TP + FN)) × (TN / (FP + TN))

• Recall : Pourcentage des membres de la classe X correctement classé comme appartenant à la classe X.

• Precision : Pourcentage des instances qui ont vraiment la classe X parmi toutes les instances qui sont classées dans la classe X.

Si tous les indicateurs sont considérés comme allant de 0 (très mauvais) à 100% (optimal), on peut voir que Recall est équivalent à TP.

La courbe ROC fournit une mesure intéressante de la performance d'un classificateur. Ce graphique présente une mesure d'estimation qui permet, par sa simple analyse du graphique, la lecture des valeurs du recall, de precision et du Fscore (Nguyen et Armitage, 2008; Rioult, 2011).

• Aire sous la courbe ROC : Pour mesurer l'aire sous la courbe ROC, des mesures normalisées par les instances des classes sont utilisées. Ces mesures sont les vrais et les faux positifs. L'aire sous la courbe ROC indique la probabilité qu'un classificateur fournisse un score d'appartenance à une classe X supérieur à celui d'une classe Y.(Rioult, 2011)

1.4.4 Évaluation des algorithmes d'apprentissages supervisés

Un bon classificateur permettrait d'optimiser le paramètre *recall* et *precision*. Cependant, il peut y avoir des compromis entre ces perspectives. Pour décider lequel parmi ces deux paramètres est le plus important ou qui devrait être plus prioritaire, il faut prendre en compte le coût de la prise des décisions erronées ou des mauvaises classifications. La décision dépend donc d'un contexte d'application spécifique et selon les priorités commerciales et

opérationnelles recherchées. Il existe plusieurs outils qui permettent le support à la prise de décision dans ces cas. La courbe ROC fournit un moyen de visualisation des compromis entre TP et FP en traçant le nombre de TP en fonction du nombre de FP (tous les deux sont respectivement exprimés en pourcentage du total des TP et des FP). Un grand défi qui se présente lors de l'utilisation d'algorithmes d'apprentissage supervisé est que les phases de formation et de test doivent être effectuées en utilisant des ensembles de données qui ont déjà été classées et vérifiés (marqué). Idéalement, il faudrait avoir un grand ensemble de données d'apprentissage (pour l'apprentissage optimal et la création de modèles) et un grand ensemble de données de tests pour évaluer correctement la performance de l'algorithme.

En pratique, nous sommes souvent confrontés à une quantité limitée de données expérimentales pré-étiquetés. Une procédure simple, pour résoudre ce problème, consiste à mettre une partie de ces données pré-étiquetées de côté pour la formation (par exemple, les deux tiers) et le reste (par exemple, un tiers) pour les tests. Une autre procédure qui peut être utilisée dans ce cas, est la validation croisée. Cette procédure consiste à diviser l'ensemble des données en N partitions approximativement égales. Chaque partition (1/N) à tour de rôle est utilisée pour les tests tandis que le reste des partitions ((N - 1) / N) seront utilisées pour le training (formation). La procédure est répétée N fois de telle sorte qu'en fin de compte, chaque exemple serait utilisé une seule fois pour le test. Le *Recall* et la *Precision* sont calculés à partir de la moyenne des *Recalls* et *Precisions* mesurées à partir de tous les tests de N. Il a été proposé que N = 10 fournit une bonne estimation de la performance de classification (Nguyen et Armitage, 2008).

1.4.5 La sélection des caractéristiques d'un vecteur d'entrée

Les méthodes "*filter*" et "*wrapper*" représentent les deux groupes principaux des méthodes de sélection des caractéristiques.

1.4.5.1 Les méthodes de filtrage (ou *filter*)

Afin d'effectuer le filtrage des caractéristiques qui contiennent peu d'information, ce type de méthodes utilise des mesures statistiques calculées en fonction de ces caractéristiques. L'application des algorithmes de classification ne peut être faite qu'à la suite de cette opération. Les principaux avantages de ce type de méthode sont l'efficacité calculatoire et la robustesse face au sur-apprentissage.

Ce type de méthode présente aussi des inconvénients tels que le fait de ne pas tenir compte des interactions entre les caractéristiques et d'effectuer la sélection des caractéristiques comportant des données redondantes au lieu de chercher ceux qui ont des informations complémentaires. Ce type de méthode ne prend pas en considération la méthode de classification qui va être utilisée (Chouaib, 2011).

1.4.5.2 Les méthodes enveloppantes (ou *wrapper*)

En vue de chercher un sous-ensemble de caractéristiques optimales, cette approche procède à l'exploration de l'espace des caractéristiques pour l'algorithme de classification utilisée. Les sous-ensembles sélectionnés sont adaptés à cet algorithme et peuvent ne pas rester valides si un autre algorithme de classification est utilisé.

Le temps de calcul dépend principalement de la complexité de l'algorithme d'apprentissage utilisé. Les méthodes enveloppantes sont considérées meilleures que celles du filtrage dû à plusieurs critères. Un de ces critères est que ce type de méthode est capable de faire la sélection des sous-ensembles de caractéristiques performants pour le classificateur utilisé et en plus de tailles plus petites que celles des méthodes du filtrage (Chouaib, 2011) .

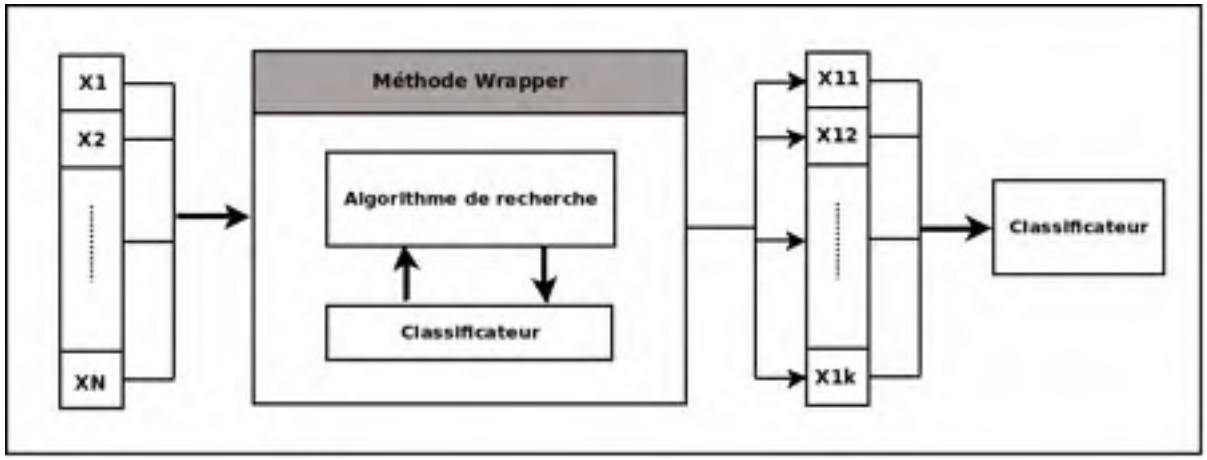

Figure 1.14 Méthode de sélection enveloppante
Tirée de Chouaib (2011)

1.4.6 Les méthodes de comparaison entre différents algorithmes de classifications

Plusieurs familles de tests statistiques peuvent être effectués afin d'effectuer la comparaison entre deux ou plusieurs classificateurs utilisés sur multiples ensembles de données. Citant par exemple une première famille de ces tests qui sont les tests paramétriques. Cette famille est composée de plusieurs méthodes telles que paired T-test et ANOVA. Un autre exemple de famille est les tests non paramétriques tels que Wilcoxon et le test de Friedman. Ce dernier exemple de famille est aussi des tests non paramétriques, mais cette famille a pour caractéristique qu'elle n'assume aucune proportionnalité des résultats. Finalement, le test du signe est une technique appartenant à cette famille. Dans cette recherche le choix porte sur la famille des tests paramétriques et plus précisément la technique paired T-test (Demšar, 2006; Khorasgan, 2010).

La technique paired T-test

Le paired T-test est utilisé fréquemment pour comparer deux populations. Cela signifie que les observations, dans un échantillon, peuvent être jumelées avec les observations dans l'autre échantillon. Les exemples de situations où cela pourrait se produire sont :

• Avant et après les observations sur les mêmes sujets ;

• Une comparaison de deux méthodes différentes de mesure ou deux traitements différents où les mesures / traitements sont appliquées sur les mêmes sujets (Paired t-tests, 2004).

Comparaison des significations statistiques

Le paired T-test est une méthode de test d'hypothèse statistique pour comparer le résultat de mesure d'un groupe deux fois. En prenant en compte à la fois de la moyenne et de la variance des différences entre ces deux mesures sur plusieurs exécutions, le paired T-test calcule la valeur tvalue. En utilisant cette valeur et le niveau de signification désirée (normalement de 5%), la probabilité que ces deux mesures soient nettement différentes peut être obtenue en consultant la table t-distribution. Par conséquent nous pouvons dire que ces classificateurs sont significativement différents ou non avec un certain degré de confiance (100 - niveau de signification). Le paired T-test fait une fausse supposition que ces différences de précision entre les deux classificateurs sont indépendantes et ils ont donc une distribution normale (qui n'est en fait pas vrai parce que les ensembles de tests et les ensembles de formation se chevauchent). Le ré échantillonnage du paired T-test corrigé stimule le paired T-test en utilisant la fraction des données utilisées pour les tests et la formation dans la formule de calcul t. Le ré échantillonnage du paired T-test corrigé est effectué à l'aide de l'outil d'analyse de la machine d'apprentissage WEKA afin de prétendre lequel des classificateurs appliqués est mieux (Khorasgan, 2010).

1.5 Conclusion

Au cours de ce chapitre, il y a eu la description des différentes parties touchées par nos expérimentations et notre étude. Il a débuté par présenter un descriptif du noyau du système d'exploitation Linux et les différentes parties qui le constituent. Aussi, il a décrit les différents types de rootkits et les différentes techniques utilisées pour que les personnes malveillantes s'infiltrent dans le système d'exploitation Linux. Enfin, les différentes techniques reliées à l'apprentissage automatique supervisé ainsi que les différentes méthodes

et mesures utilisées pour l'optimisation et la comparaison des classificateurs, ont été présentées.

CHAPITRE 2

REVUE DE LITTÉRATURE SUR LES MÉTHODES D'ANALYSE ET DE DÉTECTION DES ROOTKITS

2.1 Introduction

L'analyse des logiciels malveillants est indispensable pour leur détection sur un système d'exploitation. Les différentes approches de détection disposent de plusieurs techniques d'analyse qui peuvent être utilisés.

Dans ce chapitre, on considère les questions suivantes :

1. Comment analyser les rootkits dans le système?
2. Comment détecter la présence des rootkits dans le système?

Afin de répondre à ces questions, cette étape de la recherche a été orientée en suivant deux axes, nécessitant deux revues de littéraires :

2. Revue littéraire concernant les techniques d'analyse des logiciels malveillants;
3. Revue littéraire concernant l'étude de différentes approches de détection des logiciels malveillants.

2.2 Méthodes d'analyse

Les outils utilisés pour l'analyse des malwares peuvent être répartis en deux catégories principales : statique et dynamique.

2.2.1 Analyse statique

Généralement, l'analyse statique est plus sécurisée que l'analyse dynamique parce qu'elle est effectuée sans que le binaire soit réellement exécuté sur la machine. Elle fournit un ensemble d'information sur le flux de données et les caractéristiques statistiques du programme (le

format exécutable, les instructions opcode et les signatures des librairies). Pour faire une analyse statique détaillée de la logique du logiciel malicieux, on est amené à utiliser un désassembleur pour analyser son code assembleur. Des outils comme File finger printing,Virus Scanning,Packer Detection, Strings peuvent être utilisés pour faire l'analyse statique (Alzarooni, 2012; Kendall et McMillan, 2007).

2.2.2 Analyse dynamique

L'analyse dynamique nécessite l'exécution du malware sur une machine réelle ou virtuelle et l'observation de son comportement. Pour cela, ce type d'analyse doit se faire dans un environnement de test isolé. La surveillance du programme et de ces interactions avec les fichiers systèmes, les registres, les autres processus et le réseau nous permet de construire une image assez claire de son comportement.

Ce type d'analyse permet d'acquérir de l'expérience dans l'analyse les logiciels malveillants, et permet aussi de développer un filtre cognitif en fonction de l'intuition de l'analyseur pour déterminer quel comportement est normal (Alzarooni, 2012; Kendall et McMillan, 2007).

La localisation de données, ciblées par les logiciels malveillants, dans la mémoire du noyau permet de déterminer en l'analysant la présence des logiciels malveillants. On peut classifier ces localisations en deux principales catégories :

- La localisation statique où les données ont une localisation fixe déterminée dès la compilation comme init_task_union qui est une structure task déclaré statiquement pour le premier processus;
- La localisation dynamique qui est déterminée lors de l'exécution comme les structures des tâches reliées aux processus crées durant l'exécution du système (Rhee et al., 2009).

2.3 Méthodes de détection

2.3.1 Détection basée sur les signatures

La détection basée sur la signature fonctionne comme une empreinte digitale. Ce concept consiste à la comparaison d'une séquence d'octets du programme principal avec une autre séquence appartenant au programme malveillant. Ce concept est adopté par la plupart des logiciels anti-virus bien qu'il est faible et non efficace contre les logiciels malveillants inconnus. Dans la plupart des cas, ce concept est appliqué sur le système de fichier. Un rootkit peut facilement s'introduire au système sans être détecté par ce genre de concept et ceci en utilisant plusieurs techniques. Pour une meilleure efficacité contre les rootkits, la mémoire du noyau doit aussi avoir des signatures. Cette méthode peut nous fournir d'excellents résultats de détection des rootkits. Les différentes signatures sont mises dans une base de données qui peut être utilisée par les outils de détection quand ils effectuent l'opération d'analyse du système. Plusieurs outils populaires de détection utilisent cette technique (Arnold, 2011; de Almeida, 2008; Idika et Mathur, 2007).

2.3.2 Détection basée sur le comportement

Les techniques de détection basées sur le comportement visent à réduire le taux de faux positifs générés lors de l'étape de la surveillance du système à protéger. Au cours de la phase d'apprentissage, un détecteur basé sur le comportement est fourni avec un ensemble de règles qui spécifient tous les comportements acceptables de toute application qui peut se présenter au sein du système à protéger. L'inconvénient majeur de la détection basée sur le comportement est la difficulté de déterminer l'ensemble des comportements sécuritaires qu'un programme peut présenter lors de son exécution au sein du système à protéger (Alzarooni, 2012).

Rabek et al. (2003) présentent une méthode de détection de logiciels malveillants masqués pouvant s'injecter et se générer dynamiquement lors de l'exécution. Le détecteur utilise une technique d'analyse statique pour obtenir des détails sur tous les appels système pertinents

intégrés dans le code, comme les noms de fonction, les adresses et l'adresse d'instruction suivie par chaque appel système. Le détecteur conserve un enregistrement des adresses de retour pour les appels de système dans le code. Puis, quand un programme suspect est exécuté, le détecteur surveille le comportement de l'exécutable et s'assure que tous les appels aux services du système lors de l'exécution sont les mêmes que ceux enregistrés lors de la première étape. Les auteurs ont conclu en se basant sur une étude de preuve de concept que leur technique assure que tout code malveillant injecté et généré peut être détecté quand il fait des appels système inattendu. Un inconvénient majeur de cette technique est lors de l'insertion de certains appels d'APIs non pertinents dans un code malveillant, le détecteur peut ne pas réussir à correspondre le nouveau comportement malicieux avec le comportement déjà enregistré (Rabek et al., 2003).

La recherche de Wang et Karri (2013) présente l'outil NumChecker, un nouveau moniteur de machine virtuelle (VMM) fondé pour détecter le flot de contrôle de modification des rootkits noyau dans la machine virtuelle hôte (VM). NumChecker détecte les modifications malicieuses d'un appel système dans l'invité VM en vérifiant le nombre de certains événements matériels qui se produisent durant l'exécution de l'appel système. Pour compter automatiquement ces événements, NumChecker s'appuie sur les compteurs de performance matérielle qui sont principalement le nombre total d'instructions, les branches, les *returns* et les opérations des points flottants. En utilisant ces paramètres, le coût de vérification est considérablement réduit et la protection est renforcée. Et outil qui a montré qui est pratique et efficace. Dans cette étude, les rootkits utilisés sont SuckIT qui remplace la table des appels systèmes avec sa propre copie et il l'utilise pour la redirection vers les appels systèmes malicieux. Un autre rootkit utilisé est Adore-ng qui manipule les pointeurs de fonctions à la couche VFS pour la redirection du flot d'exécution des routines malicieuses qui cachent des informations en filtrant les données systèmes. Un taux de déviation de 5 % est fixé comme valeur au-dessus de laquelle un appel système est considéré comme modifié. Les appels systèmes supervisés sont sys_open, sys_close, sys_read, sys_getdents64, sys_stat64 (Wang et Karri, 2013; 2014).

Dans la même logique, l'outil patchfinder proposé par Rutkowski (2002) utilise la technique d'analyse de chemin d'exécution pour la détection des rootkits. Ceci en comptant les nombres d'instructions exécutés sur un processeur en mode single step. Une information de débogage est générée après chaque exécution d'instruction. La vulnérabilité principale de cette technique est que le comptage et l'analyse peuvent être manipulés par les rootkits modernes ayant un haut privilège et un accès total à la mémoire noyau (Rutkowski, 2002).

Une autre technique de détection utilisée pour identifier les rootkits niveau noyau est la technique de récupération du temps d'exécution des appels systèmes à l'état clean et à l'état infecté (Rutkowski, 2002). Cette mesure permet de différentier entre le comportement normal ou malicieux des appels systèmes par la récupération de sa valeur après un certain nombre de répétitions (Brodbeck, 2012).

2.3.3 Détection basée sur la vérification d'intégrité

Cette technique est considérée comme l'une des techniques les plus efficaces pour la détection des malwares du type 1. Elle consiste principalement à vérifier si certaines régions du système des fichiers et de la mémoire sont identiques à une base de confiance contenant des valeurs connues de ces régions. Les régions qui doivent être analysés sont les plus susceptibles d'être modifiés comme les sections du code en mémoire dans l'espace noyau ou l'espace utilisateur et les fichiers systèmes. Ce type de bases de données doit être sécurisé (de Almeida, 2008).

Cette technique calcule les fonctions de *hashs* cryptographiques pour les fichiers du système d'exploitation critiques et inchangeables et les compare aux valeurs connues qui sont stockées dans une base de données. Typiquement, cette base de données est générée avec une version propre du système d'exploitation. Alors quand une variation est détectée, on constate qu'un fichier a été modifié (probablement par un logiciel malveillant). Cette technique fonctionne bien contre la dissimulation des fichiers par des rootkits, mais les auteurs des rootkits les adaptent rapidement afin qu'ils utilisent des techniques de détournement à sa

place. En conséquence, la vérification de l'intégrité des fichiers n'est pas largement utilisée comme une méthode de détection pour les systèmes anti-rootkit modernes (Arnold, 2011).

L'étude de Malone, Zahran et Karri (2011) consiste à détecter la modification des programmes au cours de leur chargement ou pendant leur exécution respectivement par la vérification statique et dynamique d'intégrité en se basant sur les compteurs de performances matérielles (ou HPC). Ces derniers sont un ensemble de registres spéciaux intégrés dans presque tous les processeurs. L'avantage principal de la vérification d'intégrité basée sur les HPCs est que son coût matériel est négligeable. La méthode de vérification statique collecte les valeurs de ces compteurs pour les programmes cibles et les enregistre hashés et cryptés sur le disque. Tandis que la méthode de vérification dynamique collecte ces valeurs dans des petits intervalles de temps durant l'exécution du programme cible et cherche par la suite une relation mathématique approximative entre ces compteurs. Les HPCs utilisés dans cette recherche sont INS, BR, WR, IO, FP et FN.

Cette étude a montré qu'en utilisant la vérification statique d'intégrité on arrive à détecter les modifications. Le compteur IO présente les meilleurs résultats de détection puisqu'il est le plus sensible à ce genre de modification. En utilisant la vérification dynamique, les compteurs INS et WR ont montré qu'ils ont une relation linéaire. Les expérimentations sont faites à base de ces deux paramètres. Le taux de déviation de 5% est fixé comme taux au-dessus duquel le programme est considéré malicieux (Malone, Zahran et Karri, 2011).

2.3.4 Détection basée sur le *crossview*

La détection basée sur *Crossview* (ou *X-View*) compare une vue de haut niveau avec une vue de bas niveau du système. Si la vue de haut niveau ne montre pas le même contenu que la vue du bas niveau, cela signifie que quelque chose a été caché et le système a été modifié. *X-View* peut être utilisé pour détecter les fichiers cachés, les processus, les clés de registre et les connexions réseau. La vue de haut niveau du système est obtenue en utilisant les fonctions API communes fournies par le système d'exploitation. Dans le cas d'un rootkit, la vue de bas

niveau devrait être obtenue sans être modifiée et c'est la partie la plus difficile lors de la conception d'un *X-View*. L'obtention de la vue de bas niveau dépend de ce qu'on est en train d'analyser. Pour mettre en œuvre l'approche *X-View* dans une analyse du système de fichiers, une bonne approche pour obtenir une vue de bas niveau serait d'accéder aux secteurs de fichiers du disque et les analyser en fonction de la mise en page NTFS. Un rootkit intelligent fera la modification de la fonction qui lit les secteurs de fichiers et fournit à l'anti-rootkit les mauvaises informations sur le système de fichier. Cela signifie qu'un anti-rootkit *X-View* fiable peut être très difficile à mettre en œuvre (de Almeida, 2008; Rutkowska, 2005).

2.3.5 Détection basée sur la sémantique

La détection de logiciels malveillants basés la sémantique est une nouvelle approche qui peut surmonter les faiblesses des méthodes de détection heuristiques ou à base de signatures en intégrant la sémantique de l'énoncé de programme (instructions) plutôt que les propriétés syntaxiques du code. L'approche basée sur la sémantique a la capacité d'identifier le comportement malveillant d'un programme caché et peut améliorer la détection des futures variantes inconnues de logiciels malveillants (Alzarooni, 2012; Preda et al., 2007).

2.3.6 Détection basée sur les heuristiques

Les approches de détection de logiciels malveillants basée sur les heuristiques déploient plusieurs techniques, comme l'apprentissage automatique, dans le but de rechercher des attributs et des caractéristiques spécifiques nécessaires à l'identification de variantes de logiciels malveillants. La plupart des systèmes de détection de logiciels malveillants qui utilisent les techniques heuristiques n'ont pas besoin de créer ou de maintenir des signatures. Cette approche détecte habituellement une anomalie dans le programme testé ou dans le système hôte où le programme sera exécuté.

La détection des programmes malveillants en utilisant cette approche est accomplie en deux phases :

- La première phase est la phase du *training* (voir section 1.4.2). Un système de détection doit être formé avec des données d'entrée afin de capturer les caractéristiques d'intérêt;
- La deuxième phase est la phase de surveillance ou de détection. Dans cette phase, le détecteur formé prend des décisions intelligentes sur de nouveaux échantillons basés sur des données du *training*.

En outre, il existe deux méthodes déployées dans la phase du *training* :

- La première méthode utilise deux catégories de données, à savoir les données normales et anormales;
- La deuxième méthode utilise une seule catégorie de données. Dans ce cas, les détecteurs de logiciels malveillants sont formés avec une seule classe (normale ou anormale). Cela signifie que le système sera formé seulement avec l'activité normale du système, ce qui lui permet d'identifier la présence d'une activité anormale (Alzarooni, 2012; Arnold, 2011; de Almeida, 2008).

Diverses approches d'apprentissage automatique comme les règles d'association, le Support Vector Machine, les arbres de décision, le Random Forest, le naive bayes et la mise en cluster ont été proposées pour la détection et la classification des malwares. La classification est appliquée sur des échantillons inconnus en famille de malwares connus ou aussi souligner les échantillons qui présentent un comportement invisible pour une analyse détaillée. Les algorithmes d'apprentissage les plus utilisés dans la littérature sont discutés dans cette section (Gandotra, Bansal et Sofat, 2014).

Schultz et al. (2001) ont été les premiers à introduire le concept de la fouille de données pour la détection des malwares. Ils ont utilisé trois caractéristiques statiques différentes pour la classification des logiciels malveillants : Portable Executable (PE), les chaînes et les séquences d'octets. Dans l'approche PE, les caractéristiques sont extraites à partir des informations de DLL dans les fichiers PE. Les chaînes sont extraites des exécutables sur la

base des chaînes de texte qui sont codés dans les fichiers programmes (program files). L'approche de la séquence d'octets utilise des séquences de n octets extraites d'un fichier exécutable. Ils ont utilisé dans leur étude un ensemble de données qui se composait de 4 266 fichiers incluant 3265 programmes malveillants et 1001 programmes bénins. Un algorithme de règle d'induction appelé Ripper (Cohen, 1995) a été appliqué pour trouver des modèles dans les données de DLL. L'algorithme d'apprentissage Naive Bayes a été utilisé pour trouver des modèles dans les chaines de données et les n-grammes de séquences d'octets. Ces données sont utilisées comme données d'entrée pour l'algorithme multinomial de Bayes Naive. L'algorithme naive bayes prend les chaînes de données comme entrée et il donne la plus grande précision de classification de 97,11%. Les auteurs ont mis en évidence que le taux de détection des malwares utilisant la méthode basée sur la fouille de données est de deux fois de plus que la méthode basée sur les signatures. Leurs résultats ont été améliorés plus tard par Kolter et al. (Kolter et Maloof, 2004). Ils ont utilisé des n-grammes (au lieu de séquence d'octets) et la méthode de fouille de données pour détecter les exécutables malicieux. Ils ont utilisé différents classificateurs comme Naive-Bayes, Support Vector Machine, les arbres de décision et leurs versions boostés. Ils ont conclu que l'arbre de décision boostée donne les meilleurs résultats de classification (Gandotra, Bansal et Sofat, 2014; Schultz et al., 2001).

Tian et al. (2008) ont utilisé la fréquence de longueur de fonctions pour classer les trojans. La longueur de fonctions est mesurée par le nombre d'octets dans le code. Leurs résultats indiquent que la longueur de fonctions ainsi que leur fréquence sont importantes dans l'identification des familles de logiciels malveillants et peuvent être combinés avec d'autres caractéristiques de classification des logiciels malveillants rapide et évolutif. Ils ont utilisé des algorithmes d'apprentissage machine disponibles dans la bibliothèque de WEKA pour classer les malwares (Gandotra, Bansal et Sofat, 2014; Hall et al., 2009; Tian, Batten et Versteeg, 2008).

Santos et al. (2011) ont mis en évidence que l'apprentissage supervisé nécessite une quantité importante d'exécutables marqués en tant qu'ensemble de données appartenant soit à la classe

malveillante ou à la classe bénigne. Ils ont proposé une approche d'apprentissage semi-supervisé pour détecter les malwares inconnus. Cette approche est conçue pour construire un classificateur d'apprentissage automatique en utilisant un grand nombre d'instances étiquetées et non étiquetées. L'algorithme semi-supervisé LLGC est utilisé par cette approche. Cet algorithme est capable d'apprendre à partir des données étiquetées et non étiquetées et de fournir une solution par rapport à la structure intrinsèque affichée par les deux types d'instances étiquetés et non étiquetés. Les exécutables sont représentés en utilisant la technique de distribution n-gram. Ils déterminent et évaluent aussi le nombre optimal d'instances étiquetées et l'effet de ce paramètre sur la précision du modèle. La contribution principale de cette recherche est de réduire le nombre nécessaire d'instances marquées tout en conservant une haute précision (Gandotra, Bansal et Sofat, 2014; Santos, Nieves et Bringas, 2011).

Rieck et al. (2011) ont proposé une *framework* d'analyse automatique du comportement des programmes malveillants en utilisant l'apprentissage automatique. Ce *framework* collecte un grand nombre d'échantillons de logiciels malveillants et surveille leur comportement en utilisant un environnement *sandbox*. Ils appliquent les algorithmes d'apprentissage en intégrant le comportement observé dans un espace vectoriel. Ils utilisent l'apprentissage non supervisé pour identifier les nouvelles classes de logiciels malveillants ayant un comportement similaire. L'affectation de ces logiciels malveillants inconnus à ces classes découvertes se fait par la classification. Une approche incrémentale basée sur le regroupement et la classification sont utilisées pour l'analyse comportementale. Cette approche est capable de traiter le comportement de milliers de binaires malveillants sur une base quotidienne (Gandotra, Bansal et Sofat, 2014; Rieck et al., 2011).

Anderson et al. (2011) ont présenté un algorithme de détection de logiciels malveillants basé sur l'analyse des graphiques. Ces graphiques sont construits à partir des traces d'instruction collectées dynamiquement. Une version modifiée du *framework* d'analyse des logiciels malveillants Ether (Dinaburg et al., 2008) est utilisée pour collecter des données. Cette méthode utilise 2-grams pour conditionner les probabilités de transition de la une chaîne de

Markov. Le mécanisme de graphiques du noyau est utilisé pour construire une matrice de similarité entre les instances de l'ensemble de formation. La matrice du noyau est construite à l'aide de deux mesures distinctes de similarité : le noyau gaussien qui mesure la similarité locale entre les contours du graphique et le noyau spectrale qui mesure la similarité globale entre les graphiques. Le SVM est formé à partir de la matrice du noyau pour classer les données de test. La performance des multiples méthodes d'apprentissage du noyau utilisée dans ce travail est démontrée par la discrimination de différentes instances de logiciels malveillants et bénins. La limitation de cette approche est que la complexité de calcul est très élevée (Anderson et al., 2011; Gandotra, Bansal et Sofat, 2014).

Tian et al. (2010) ont utilisé un outil automatisé d'extraction de séquences d'appels des API depuis les exécutables. Cette extraction est faite au moment où ces exécutables sont en cours d'exécution dans un environnement virtuel. Ils ont utilisé les classificateurs disponibles dans la bibliothèque WEKA (Hall et al., 2009) pour distinguer les fichiers malveillants des fichiers propres et pour la classification des malwares dans leurs familles. Ils ont utilisé un ensemble de données de 1368 *malwares* et 456 *cleanwares* pour démontrer leur travail et atteindre une précision de plus de 97% (Gandotra, Bansal et Sofat, 2014; Tian et al., 2010).

Park et al. (2010) ont proposé une méthode de classification de logiciels malveillants qui est basée sur la détection du composant maximale du sous-graphe. Après l'exécution des échantillons de logiciels malveillants dans un environnement *sandbox*, les appels système ainsi que les valeurs de leurs paramètres sont capturés et un graphe orienté est généré à partir de ces traces. Le sous-graphe commun maximal est calculé pour comparer deux programmes. L'inconvénient de cette méthode est qu'il y a des échantillons de logiciels malveillants connus qui parviennent à obtenir des privilèges en mode noyau sans faire usage de l'interface de l'appel système. Ceci permet d'échapper de la méthode d'analyse (Gandotra, Bansal et Sofat, 2014; Park et al., 2010).

Firdausi et al. (2010) ont présenté une preuve de concept d'une méthode de détection des malwares. Tout d'abord, le comportement des échantillons de logiciels malveillants est

analysé dans un environnement *sandbox* en utilisant Anubis. Les rapports générés sont prétraités dans les modèles de *sparse vector models* pour la classification en utilisant l'apprentissage machine. La comparaison des performances des cinq classificateurs différents à savoir KNN, Naive Bayes, l'arbre de décision J48 , le SVM, et le MLP se fait sur un petit ensemble de données de 220 échantillons malveillants et de 250 échantillons bénignes avec et sans la fonction sélection. Les résultats obtenus représentés montrent que la meilleure performance est obtenue par l'algorithme d'arbre de décision J48 avec un rappel de 95,9%, un taux de faux positif de 2,4%, une *precision* de 97,3%, et une *accuracy* de 96,8% (Firdausi et al., 2010; Gandotra, Bansal et Sofat, 2014).

Santos et al. (2013) ont proposé un détecteur hybride de logiciels malveillants inconnus appelé OPEM. Ce détecteur utilise un ensemble de caractéristiques obtenues de l'analyse statique et dynamique du code malveillant. Les caractéristiques statiques sont obtenues par la modélisation d'un exécutable en tant que séquence de codes opérationnels. Tandis que les caractéristiques dynamiques sont obtenues par la surveillance des appels systèmes, des opérations et des exceptions soulevées. L'approche est ensuite validée avec deux ensembles de données différentes en utilisant différents algorithmes d'apprentissage comme les classificateurs d'arbre de décision, KNN, le réseau bayésien et le SVM. Il a été constaté que cette approche hybride améliore les performances des deux approches statique et dynamique lorsqu'ils sont exécutés séparément (Gandotra, Bansal et Sofat, 2014; Santos et al., 2013).

Dans la recherche de Demme et al. (2013), la méthodologie utilisée était d'étudier la faisabilité d'un détecteur matériel de rootkit qui se base sur les HPCs. Ils appliquent les algorithmes de classification comme KNN, les arbres de décisions, FANN, *Random forest* et Tensor pour détecter les variantes connues de rootkit. Ce travail montre qu'on utilisant cette approche on peut détecter les malwares avec une précision proche de 90% et 3% comme taux de false positive. Les rootkits utilisés dans ce travail sont : average coder qui est un rootkit niveau noyau et Jynx2 qui est un rootkit niveau librairie. Cette étude se base sur l'étude de performances des processus ps, ls, netstat et who (Demme et al., 2013).

L'étude de Quinn (2012) consiste à utiliser le paramètre de l'utilisation du cache L2 du CPU pendant la phase de test dans un environnement clean et dans un autre après installation de deux types de rootkits. Les paramètres sont les cycles de transferts des données vers le core, les lignes du cache expulsé et les requêtes du cache. Les deux tests qui sont en premier lieu le faite de lister les répertoires ayant un long chemin et en deuxième lieu on liste les répertoires contenant un grand nombre de fichiers. Dans cette étude il applique trois algorithmes de classification qui sont Mahalanobis Distance, One-class SVM, KNN pour la détection de rootkits sur plusieurs systèmes d'exploitation. Les résultats des expérimentations faites sur Linux montrent que la détection des deux rootkits Adore et Enyelkm n'est assuré que par les paramètres PMC 0x23 et PMC 0x2E. Le paramètre PMC 0x25 ne permet pas leur détection.

L'étude Revathi et Malathi (2014) montre l'utilisation de ces algorithmes Multi-layer perceptron, J48, Random forest, JRIP et Navie Bayes dans la détection des attaques U2R qui contient des attaques comme Buffer_overflow, Rootkit , Loadmodule , Perl , SQLattack , Xterm et ps. Selon cette étude, le meilleur algorithme pour la détection des attaques U2R est MLP. Les valeurs d'évaluation pour la détection des rootkits par MLP sont : 0.889 pour la précision, 0.8 pour la valeur du recall et 0.842 pour la valeur du Fvalue (Revathi et Malathi, 2014).

2.4 Conclusion

Ce chapitre a présenté les principales méthodes d'analyse des rootkits et a discuté de différentes approches de détection utilisées. Cette revue de littérature permet de répondre aux différentes questions énoncées en début de chapitre :

1) Comment analyser les rootkits dans le système?

Les rootkits peuvent toucher diverses parties du noyau du système d'exploitation (voir section 1.3) d'où la nécessité de l'analyse dynamique. Cette méthode nous permet d'avoir plusieurs mesures importantes sur le comportement et 'état du système (voir section2.2).

Le traceur LTTng est notre outil d'analyse dynamique qui va nous fournir les mesures nécessaires pour la détection de rootkits.

2) Comment détecter la présence des rootkits dans le système?

À la lumière de la revue de littérature présentée à la section 2.3, nous avons pu identifier les approches de détection qui peuvent se baser sur le traceur LTTng. Pour la détection des rootkits qui modifient les données statiques du noyau, la meilleure approche à utiliser est la détection comportementale. Tandis que pour la détection des rootkits qui modifient les données dynamiques du noyau, les approches adéquates à utiliser sont la détection basée sur les heuristiques (et plus précisément la technique d'apprentissage automatique), la détection basée sur le CrossView et la détection basée sur le balayage de la structure des tâches.

CHAPITRE 3

APPROCHES D'ANALYSE ET DE DÉTECTION DES ROOTKITS BASÉES SUR LTTNG

3.1 Introduction

Cette recherche vise, entre autres, à répondre à la problématique générale suivante :

- Comment détecter en utilisant LTTng les rootkits du ring 0 en se basant sur les données statiques et dynamiques de l'espace noyau?
- Comment l'architecture et les options du traceur LTTng peuvent-elles aider à y intégrer des mécanismes de détection des rootkits ou à être une source de données pour ces derniers ?

Pour répondre à ces questions, nous avons analysé l'architecture du traceur LTTng pour l'intégration des modules de détections et aussi mettre en œuvre un environnement d'expérimentation fiable.

Cet environnement nous a permis de tester et de valider l'approche de détection des rootkits qui est basée sur l'apprentissage automatique. Les approches et les expérimentations sont détaillées dans les sections qui suivent.

3.2 Méthode d'analyse

Dans notre travail, on s'est basé sur l'utilisation de l'analyse dynamique pour la détection des rootkits noyau (voir la section 2.4). Comme mentionné dans la section 2.2.2, l'analyse dynamique peut nous fournir deux types de localisation de données. Notre choix de ces données est dépendant de la nature des rootkits étudiés :

- Les données statiques intéressantes sont les adresses des tables systèmes et leurs entrées;
- Les données dynamiques intéressantes sont les compteurs de performances matérielles et logicielles ainsi que les structures des données du noyau. Les compteurs de performances du système et plus précisément les compteurs de performances matérielles (ou *HPC*) ont montré leur efficacité dans la détection de rootkits (voir section 2.3). Dans notre étude, on a utilisé les compteurs de performances matérielles et logicielles pour améliorer la détection des rootkits.

3.3 Les approches de détection

Les approches de détection de rootkits détaillées dans cette section se reposent sur l'outil LTTng et sur les méthodes d'analyse des données statiques et dynamiques décrites dans la section précédente.

La première approche est la détection par apprentissage automatique, cette approche ne nécessite pas la modification de l'outil LTTng. Elle se sert des traces LTTng comportant les compteurs de performances des appels systèmes comme une entrée à la machine d'apprentissage. Les compteurs de performances matérielles et logicielles que l'outil LTTng fournit sont ceux du mode basé sur les interruptions qui est le mode basique des compteurs de performances (Xia et al., 2012).

La deuxième approche nécessite la modification de l'outil LTTng pour effectuer la détection : soit en modifiant les modules existants ou en ajoutant des modules supplémentaires. Cette approche se repose sur les données statiques (comme les adresses des appels systèmes, etc.) et dynamiques (comme la structure des taches, etc.) du noyau Linux.

3.3.1 Détection par algorithmes d'apprentissage automatique

Dans cette section, on décrit la logique suivie pour l'élaboration des expérimentations à base des compteurs de performance des appels systèmes. La figure 3.1 décrit la vue d'ensemble de la méthodologie qu'on a suivie :

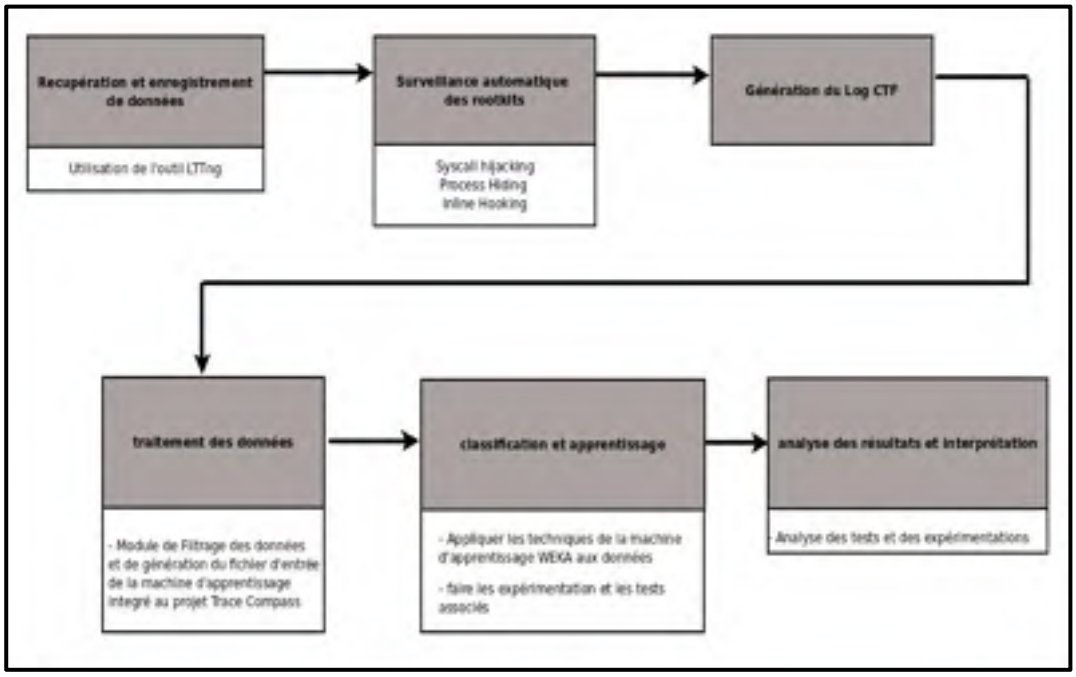

Figure 3.1 Vue d'ensemble de la méthodologie suivie
Adaptée d'Asmitha et Vinod (2014)

Les expérimentations, qui sont mises en place pour valider l'approche, se basent sur les techniques d'attaque des appels systèmes qui sont : le détournement des appels systèmes, la modification des appels systèmes et la dissimulation de processus. Ces attaques sont effectuées par Suterusu et Kbeast. Les deux premières attaques, qui sont la modification des appels systèmes et la dissimulation de processus, sont effectués par Suterusu. Tandis que le détournement des appels système est effectué par Kbeat. Les deux sous sections suivantes détaillent ces deux rootkits.

Le rootkit *Suterusu*

La plupart des rootkits effectuent traditionnellement la falsification de l'adresse de l'appel système en échangeant des pointeurs de fonction dans la table des appels système, mais cette technique est bien connue et trivialement détectable par des détecteurs de rootkit intelligents. Des nouveaux rootkits comme Suterusu utilise une technique différente et effectue l'attaque en modifiant le prologue de la fonction cible pour transférer l'exécution à la routine de remplacement. Ceci peut être observé en examinant les quatre fonctions suivantes :

- hijack_start();
- hijack_pause();
- hijack_resume();
- hijack_stop().

Ces fonctions effectuent cette technique d'attaque à travers une liste chaînée de structs sym_hook, définies comme suit :

```
struct sym_hook {
   void *addr;
   unsigned char o_code[HIJACK_SIZE];
   unsigned char n_code[HIJACK_SIZE];
   struct list_head list;
};
LIST_HEAD(hooked_syms);
```

Les pages de texte du noyau sont marquées en lecture seule, essayant d'écraser un prologue de la fonction dans cette région de la mémoire on va produire un kernel oops. Cette protection peut être contournée en mettant le bit de WP dans le registre de cr0 à 0 ce qui engendre la désactivation de la protection en écriture sur le CPU (voir tableau 3.1).

Tableau 3.1 Descriptif du bit de WP dans le registre cr0
Tirée de Sun et al. (2014)

Bits	Nom	Nom complet	Description
16	WP	Write Protect	Il détermine si le CPU peut écrire dans les pages en lecture seule

Une des choses les plus élémentaires qu'un rootkit doit faire, c'est de cacher des processus et des objets du système de fichiers qui peuvent être faits avec la même technique de base. Dans le noyau Linux, une ou plusieurs instances de la structure file_operations sont associées à chaque système de fichiers pris en charge. Ces structures contiennent des pointeurs vers les routines associées aux opérations de fichiers différents, par exemple la lecture, l'écriture, la mise en correspondance en utilisant mmap, la modification des autorisations, etc. Suterusu utilise le détournement des routines readdir et filldir de /proc afin de cacher les processus désirés (Sun et al., 2014).

Le rootkit *Kbeast*

Kbeast est un rootkit mode noyau qui se charge comme un module du noyau. Il a également une composante de l'espace utilisateur qui permet l'accès à distance à l'ordinateur. Cette porte dérobée est cachée par les autres applications du l'espace utilisateur par le module noyau. *Kbeast* cache également des fichiers, des répertoires et des processus qui commencent par un préfixe défini par l'utilisateur. *KBeast* assure son contrôle sur un ordinateur en modifiant la table des appels système et les structures utilisées pour l'implémentation de l'interface netstat en espace utilisateur (Pauna, 2012; Vandeven, 2014).

3.3.1.1 Outils constituant l'environnement d'expérimentation

Des scripts shell sont développés pour organiser le lancement, la terminaison et la configuration des différents outils et rootkits formant notre environnement de test. En plus des scripts, nous avons développé un plugin de filtrage et de conversion des traces sous le

projet trace Compass d'éclipse pour l'évaluation des expérimentations. Dans cette section, on détaille les spécificités des outils formant notre environnement de tests.

Plugin de filtrage CTF et de conversion en format ARFF du projet Trace Compass :

Les traces capturées par LTTng lors des expérimentations sont enregistrées sur le disque en format CTF (Desnoyers, 2011; Marangozova-Martin et Pagano, 2013). Pour arriver à analyser ces traces, il faut qu'on ait les outils nécessaires pour les filtrer et dessellé les informations pertinentes qui nous servira par la suite d'identifier la présence des rootkits dans notre système.

Pour toutes ces raisons, nous avons opté à développer des filtres au sein du projet Linux tools d'éclipse sous le plugin TMF qui est maintenant connu sous le nom du projet Trace Compass. Le projet Trace Compass est un outil Java pour la visualisation et l'analyse de tout type de log ou de trace. Il a pour objectif de fournir des vues, des graphiques, des indicateurs et d'extraire des informations utiles à partir des traces (Gebai, Giraldeau et Dagenais, 2014; Laperle, 2015).

Nous avons utilisé les APIs fournies par ce projet pour développer nos filtres CTF et aussi nos convertisseurs en format ARFF qui est le format d'entrée de la machine d'apprentissage WEKA (Bouckaert et al., 2010; Hall et al., 2009; Sigillito et al., 1989).

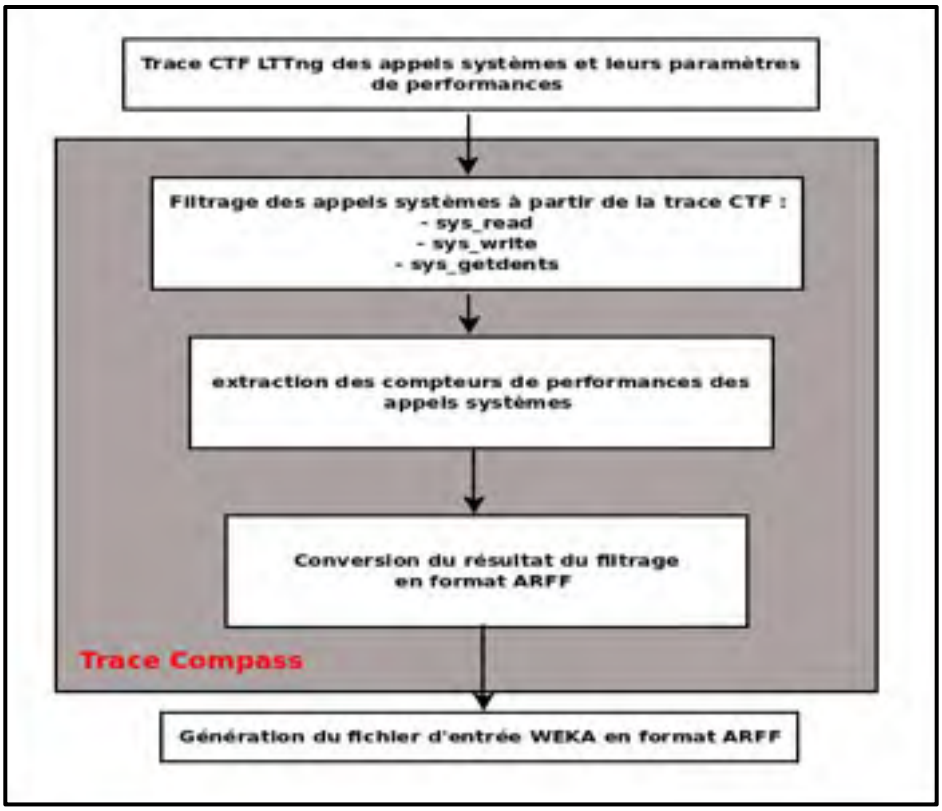

Figure 3.2 Parseur CTF et convertisseur ARFF implémentés
dans le projet Trace Compass

L'outil LTP :

L'outil Linux Test Project est un projet qui a commencé par SGI et il était développé et maintenu par IBM, Cisco, Fujitsu, SUSE, Red Hat et d'autres. Cet outil est conçu dans le but de fournir des suites de tests à la communauté open source pour valider la fiabilité, la robustesse et stabilité du noyau Linux. La suite de tests LTP contient une collection d'outils pour tester le noyau Linux et d'autres caractéristiques qui lui sont liées(LTP-developers, 2012).

Le script runltp, qui est un des outils de test du noyau contenu dans LTP, est utilisé pour l'exécution de la plupart des tests. Il exécute les tests de systèmes de fichiers, les tests des entrées/sorties du disque, les tests de gestion de la mémoire, les tests de communications interprocessus, les tests d'ordonnancement, les tests de vérification fonctionnelle des appels

systèmes et bien d'autres tests. Ce script peut être facilement modifié pour des tests personnalisés. Il doit être configuré en mentionnant les options nécessaires pour chaque type de test. Un exemple des options utilisées :

- -f CMDFILES : où on doit spécifier le fichier de commande contenant les suites de tests qu'on souhaite lancer.
- -t DURATION : où on spécifie la durée de l'exécution de la suite de test (à titre d'exemple : -t 60s pour 60 seconds, -t 45m pour 45 minutes, -t 24h pour 24 heures, -t 2d pour 2 jours)
- -T REPETITION : où on spécifie le nombre de répétitions qu'on souhaite appliquer sur la suite de test.

Le fichier de commande contenant la suite de test qu'on souhaite lancer, qui est dans notre cas les appels systèmes étudiés, est spécifié pour le script runltp. Ce fichier contient deux champs : le premier est *tag* où on spécifie le nom de test et le deuxième est *test case* où on spécifie les tests à appliquer avec leurs arguments. Ci-dessous un exemple de fichier de commande :

```
#Tag       Test case
#-----------------------------
read01      read01
write01     write01 -T 1000
#-----------------------------
```

L'outil WEKA :

WEKA est un logiciel développé par le groupe de machine d'apprentissage de l'université Waikato. Ce logiciel intègre plusieurs techniques standards de machine d'apprentissage. En utilisant cet outil, des spécialistes et des chercheurs dans divers domaines d'expertise, ainsi que des industriels peuvent tirer des connaissances utiles à partir d'un grand nombre de

données qu'ils ne peuvent pas les analyser manuellement (Bouckaert et al., 2010; Hall et al., 2009).

Pour utiliser cette machine d'apprentissage et bénéficier des différentes fonctionnalités qu'elle contient, il faut lui donner comme entrée le fichier comportant toutes les données collectées lors des expérimentations. Ces données doivent être formatées en respectant le format ARFF. C'est le format d'entrée de la machine d'apprentissage WEKA. Les fichiers ARFF contiennent deux sections distinctes. La première section comporte les informations du *HEADER* comme le nom de la relation, la liste des attributs et leurs types. Ci-dessous un exemple du contenu de la section *HEADER* :

@RELATION SysReadTraces

@ATTRIBUTE	*page_fault*	*integer*
@ATTRIBUTE	*task_clock*	*integer*
@ATTRIBUTE	*cpu_clock*	*integer*
@ATTRIBUTE	*bus_cycles*	*integer*
@ATTRIBUTE	*branch_misses*	*integer*
@ATTRIBUTE	*branches*	*integer*
@ATTRIBUTE	*instructions*	*integer*
@ATTRIBUTE	*type*	$\{normal, intrusion\}$

La deuxième section comporte les données collectées lors des expérimentations. Ci-dessous un exemple de cette section :

@Data

62, 3113779900, 3113502095, 42604517, 1478673, 485171073, 1775880718, *normal*

62, 3113800900, 3113523081, 42606615, 1478872, 485174089, 1775903768, *normal*

302, 16028878372, 16028306660, 17989408, 1767873, 16471876, 105080407, *intrusion*

59, 16036043994, 16035513750, 39804447, 2874512, 60602260, 396309776, *intrusion*

L'outil LTTng :

Ce traceur est configuré pour créer des sessions pour les appels systèmes avec leurs compteurs de performances activées et pour activer des évènements spécifiques selon le scénario de test qu'on souhaite appliqué. La section2.3.1 contient plus de détails sur cet outil.

3.3.1.2 Environnement d'expérimentation

Les expérimentations sont faites sur un poste Linux Ubuntu 14.04 équipé d'un processeur intel i7 64 bits. On a installé LTTng et on l'a configuré pour détecter les événements et les appels systèmes de l'espace noyau. L'outil LTP est aussi installé pour déclencher l'exécution des appels systèmes modifiés par les rootkits. Des scripts de configuration et de paramétrage des outils LTTng, LTP et des rootkits sont développés. Les filtres et les convertisseurs en format ARFF qu'on a développés génèrent les fichiers ARFF contenant les données résultantes des expérimentations qui servent comme entrée aux algorithmes de classifications de la machine d'apprentissage WEKA.

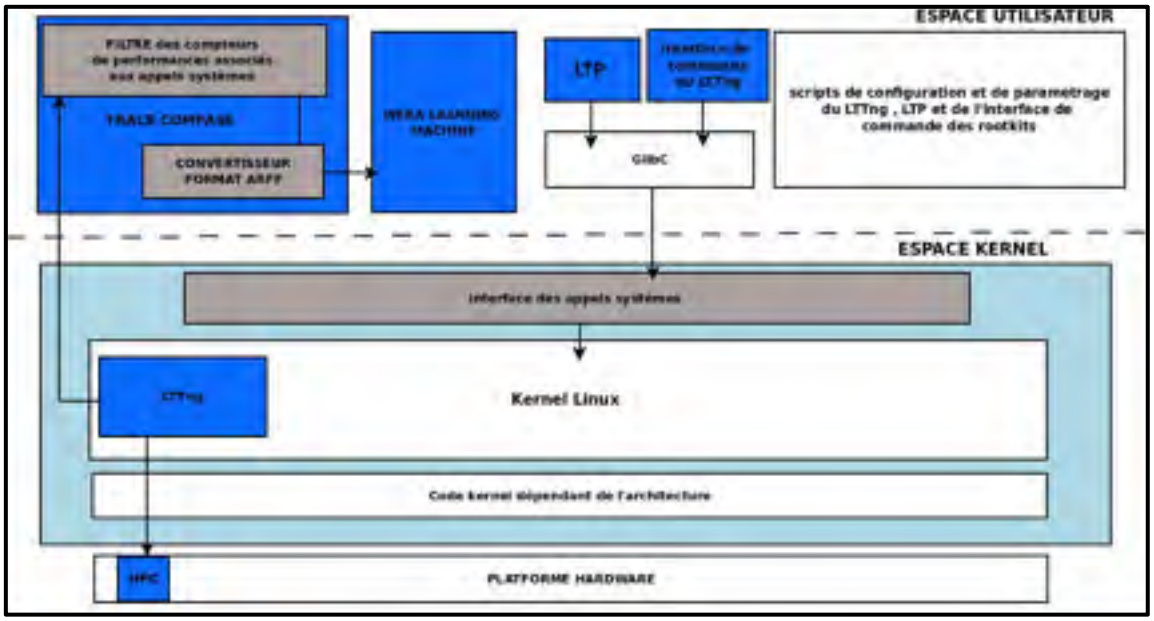

Figure 3.3 Schéma de l'environnement d'expérimentation

1) Expérimentation basée sur l'attaque de modification des appels systèmes

Cette expérimentation consiste à installer le rootkit Suterusu qui doit être compilé avec l'option de modification des appels systèmes (*inline hooking* en anglais). Une fois qu'on installe le module noyau du rootkit, on lance l'outil LTP pour déclencher les appels systèmes modifiés afin de les analyser. LTTng capte les traces des appels systèmes exécutés avec leurs différents compteurs de performances et les sauvegardes sous format CTF.

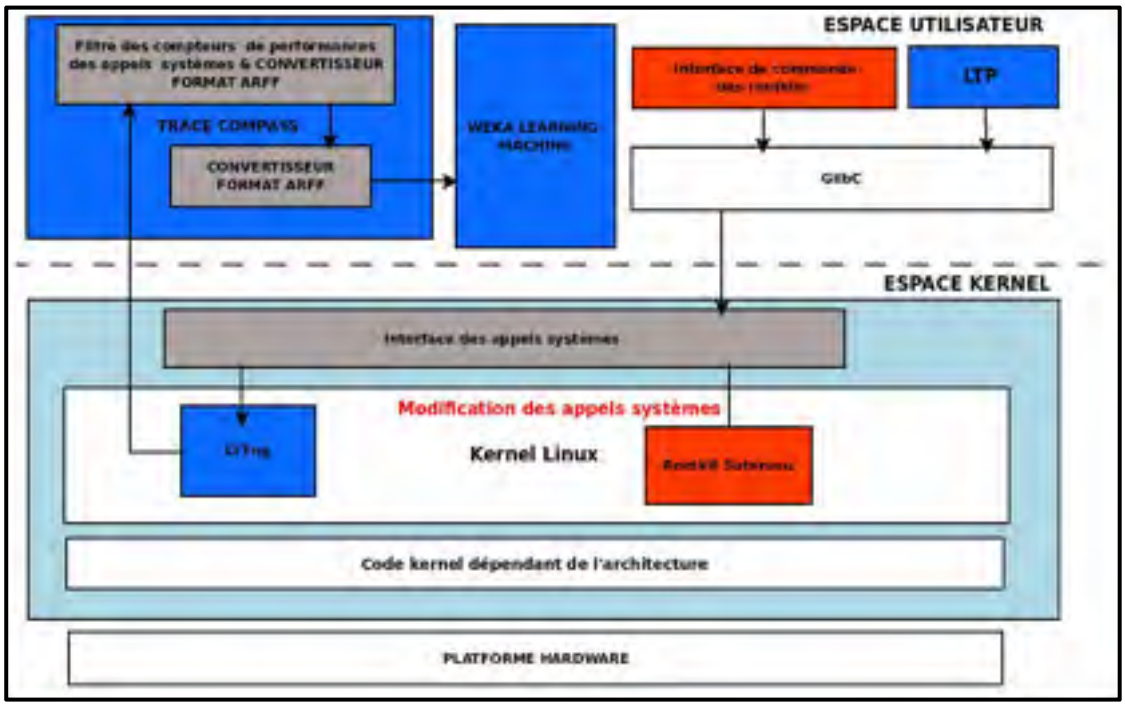

Figure 3.4 Expérimentation basée sur l'attaque de modification des appels systèmes

Cette trace est par la suite passée aux filtres du plugin TMF. Puis, elle est convertie en fichier ARFF pour être classifiée par WEKA.

2) Expérimentation basée sur l'attaque de détournement des appels systèmes

Cette expérimentation consiste à installer le rootkit noyau Kbeast qui effectue le détournement des appels systèmes (*syscall hijacking* en anglais). Une fois qu'on

installe le module noyau du rootkit, on suit les mêmes étapes que l'expérimentation précédente.

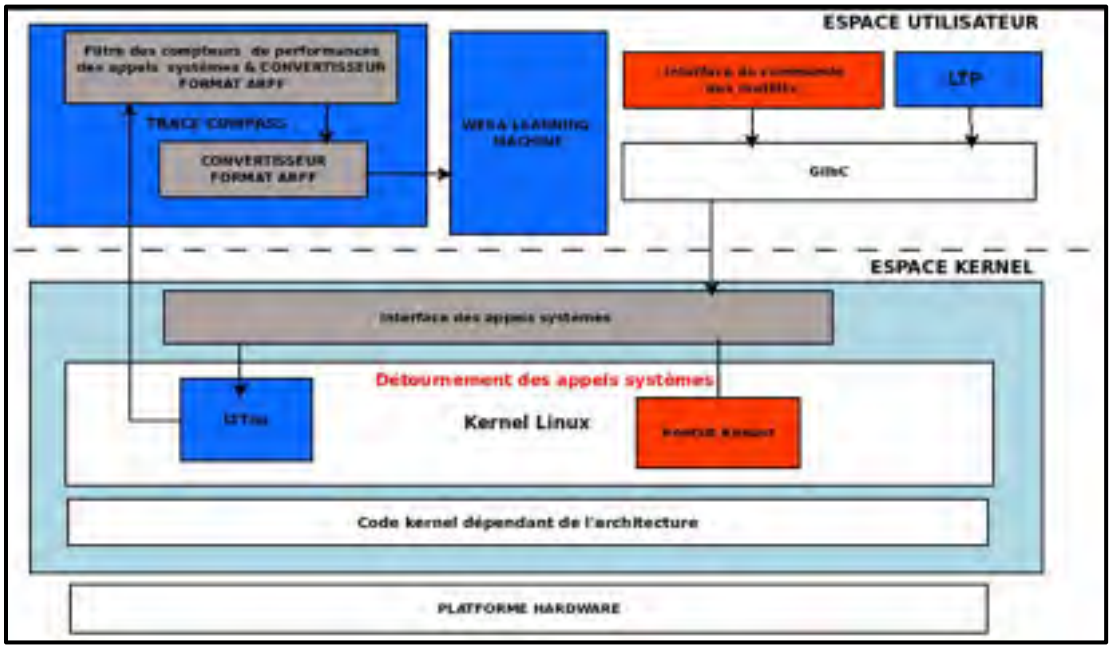

Figure 3.5 Expérimentation basée sur l'attaque par détournement des appels systèmes

3) Expérimentation basée sur l'attaque de dissimulation de processus

Cette expérimentation consiste à installer le rootkit noyau Suterusu qui a été compilé avec l'option de la dissimulation de processus. Dans l'implémentation Linux, les informations des processus est mappé dans un répertoire dans /proc.

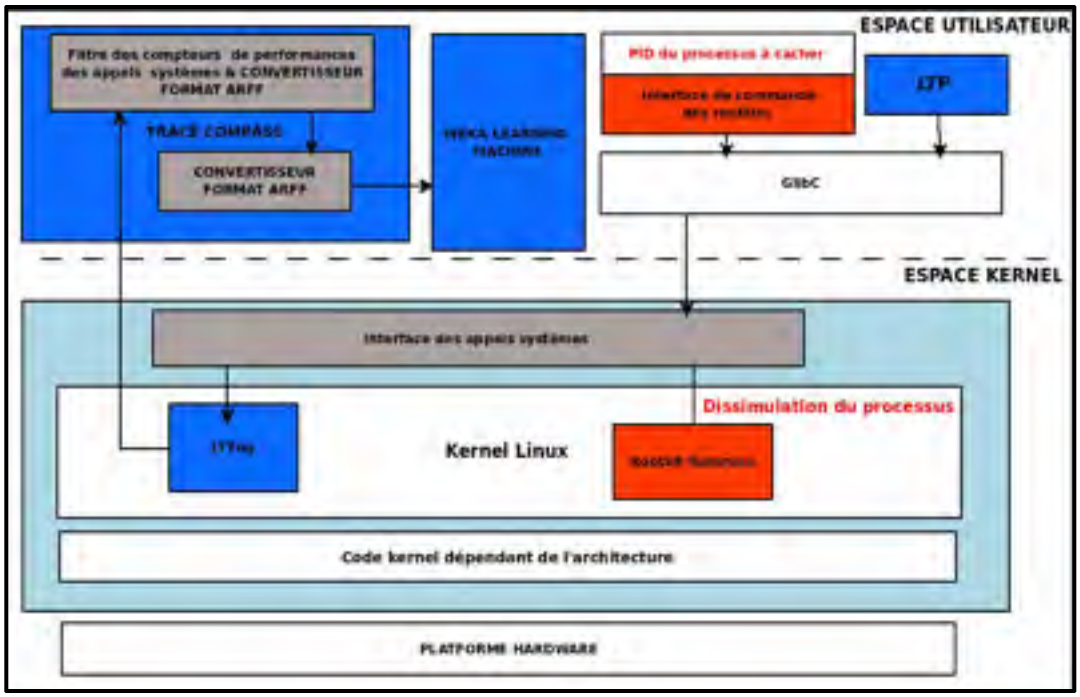

Figure 3.6 Expérimentation basée sur l'attaque par dissimulation des processus

La technique ancienne était de modifié l'appel système sys_getdents et marqué ces processus comme invisibles dans la structure des tâches (Pelaez, 2004). Dans notre cas, Suterusu laisse l'appel système sys_getdents intacte et il change les routines readdir et filldir de /proc. Une fois qu'on installe le module noyau du rootkit, on étudie l'appel système sys_getdents qui est déclenché suite à l'exécution de la commande ps et on déduit à base de son comportement l'existence d'un rootkit. Par la suite, on effectue les mêmes étapes mentionnées dans les deux expérimentations précédentes.

3.3.1.3 Algorithmes d'apprentissages sélectionnés pour les expérimentations

Différents algorithmes sont utilisés dans les différentes études précédemment mentionnées dans la section 2.4.7. Parmi ces algorithmes, on a sélectionné pour l'élaboration des expérimentations ceux les plus utilisés dans la détection des malwares. Les algorithmes sélectionnés sont KNN, adaboostM1, SVM, Multi-layer perceptron, J48, Random forest,

JRIP et Navie Bayes. Comme indiqué dans les descriptifs des expérimentations, ces algorithmes reçoivent comme ensemble de données d'entrée les compteurs de performances matérielles et logiciels des appels systèmes. Ces derniers sont modifiés soit directement par la technique de détournement des appels systèmes et la technique de modification des appels systèmes soit indirectement en changeant la structure des données systèmes utilisés pour lister les processus afin de les cacher. Pour certains algorithmes, des alternatives implémentées par WEKA sont présentes et utilisées dans les expérimentations comme :

- L'algorithme IBK utilisé puisqu'il est l'équivalent de KNN dans WEKA.
- L'algorithme SMO utilisé puisqu'il est une implémentation de SVM dans WEKA.

En vue d'optimiser les caractéristiques du vecteur d'entrée de la machine d'apprentissage, on a eu recours à l'application de la méthode de sélection *Best First* qui est une méthode de sélection enveloppante. Cette méthode de sélection est une stratégie de recherche utilisée dans le domaine d'intelligence artificielle et elle est plus robuste que la méthode hill-climbing. La méthode *Best first* sélectionne la caractéristique la plus prometteuse qui améliore principalement la précision du modèle à partir des caractéristiques générées. Si le chemin étudié est moins prometteur, cette méthode peut revenir en arrière à un sous-ensemble précédent plus prometteur et continuer la recherche à partir de là. Pour éviter de rechercher l'espace de recherche en entier, le critère d'arrêt suivant pour la méthode *Best first* est utilisé : s'il ne peut pas trouver une caractéristique qui améliore la précision de l'estimation du modèle dans les n dernière expansions, il s'arrête et revient à la meilleure solution trouvée (n est réglé à la valeur de 5, car c'est la valeur par défaut dans WEKA) (Chen, 2006). La méthode de test d'hypothèse statistique paired T-test est utilisée pour comparer les résultats des classificateurs utilisés dans cette approche.

3.3.2 Détection par extension du traceur LTTng

Cette section contient des propositions de solutions qui peuvent être intégré au traceur LTTng dans le but de détecter les rootkits noyau. Ces différentes solutions se basent à la fois

sur l'analyse des données statiques et dynamiques. Les attaques visées dans cette partie sont le détournement des appels systèmes, la dissimulation de processus, le mode *promiscuous* de l'interface réseau et l'utilisation des appels systèmes obsolètes. Ci-dessous un diagramme qui représente l'extension proposée du traceur LTTng pour supporter les différentes solutions (voir figure 3.7).

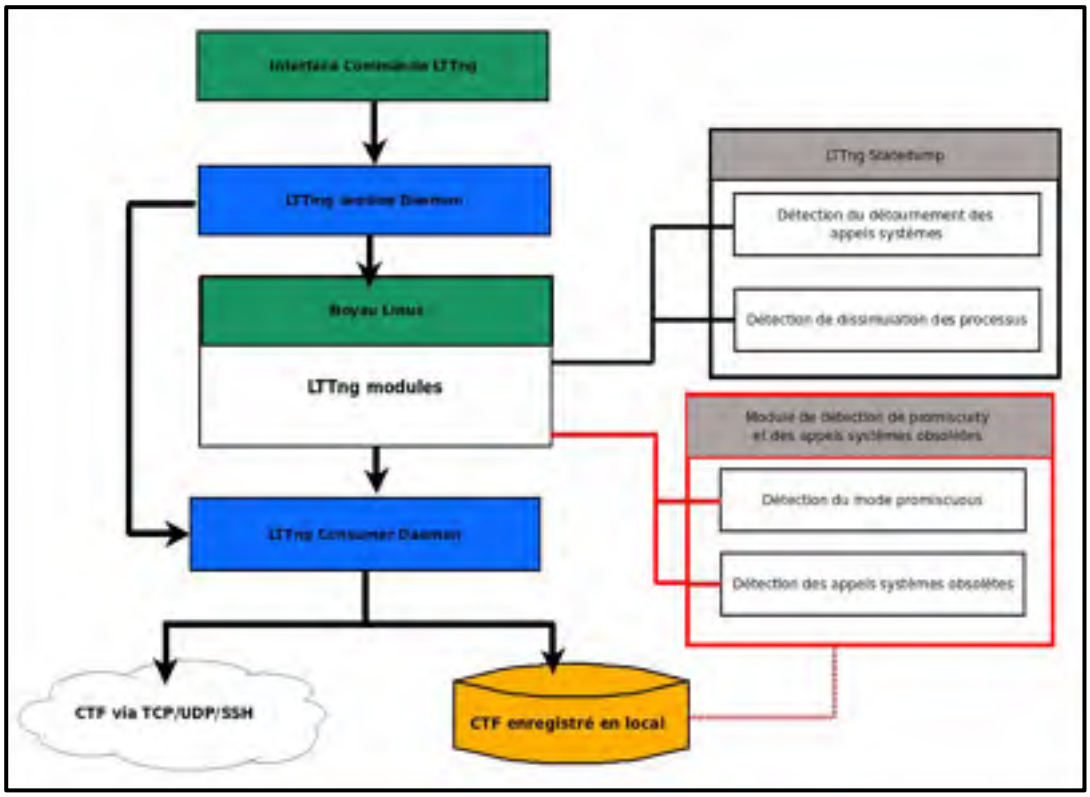

Figure 3.7 Intégration des modules de détection dans *LTTng*
Adaptée de Vocking (2012)

3.3.2.1 Détection du détournement des appels systèmes

Plusieurs outils de détections de rootkits comme *kern_check* se basent sur la comparaison d'adresse des appels systèmes encours d'exécution avec ceux contenus dans la table des symboles (dans le fichier *System.map*). Cette méthode est efficace dans la détection du détournement des tables systèmes ou des entrées ces tables. Bien que cette attaque est classique, elle est encore utilisée par les nouveaux rootkits comme Kbeast (Levine, Grizzard et Owen, 2006; Levine, Grizzard et Owen, 2004).

Grâce à son architecture, l'outil LTTng peut jouer le rôle de détecteur de ce type d'attaque en implémentant la technique de comparaison d'adresses. Le composant LTTng *modules* est notre cible. Le module *statedump*, faisant partie de LTTng *modules*, est le module adéquat pour contenir les nouvelles modifications qui permettent la détection du détournement des appels systèmes et de redirection des tables systèmes. Un nouvel événement devra être ajouté à ce module pour effectuer la détection. La figure 3.8 présente le diagramme de modification permettant la détection de ce type d'attaque.

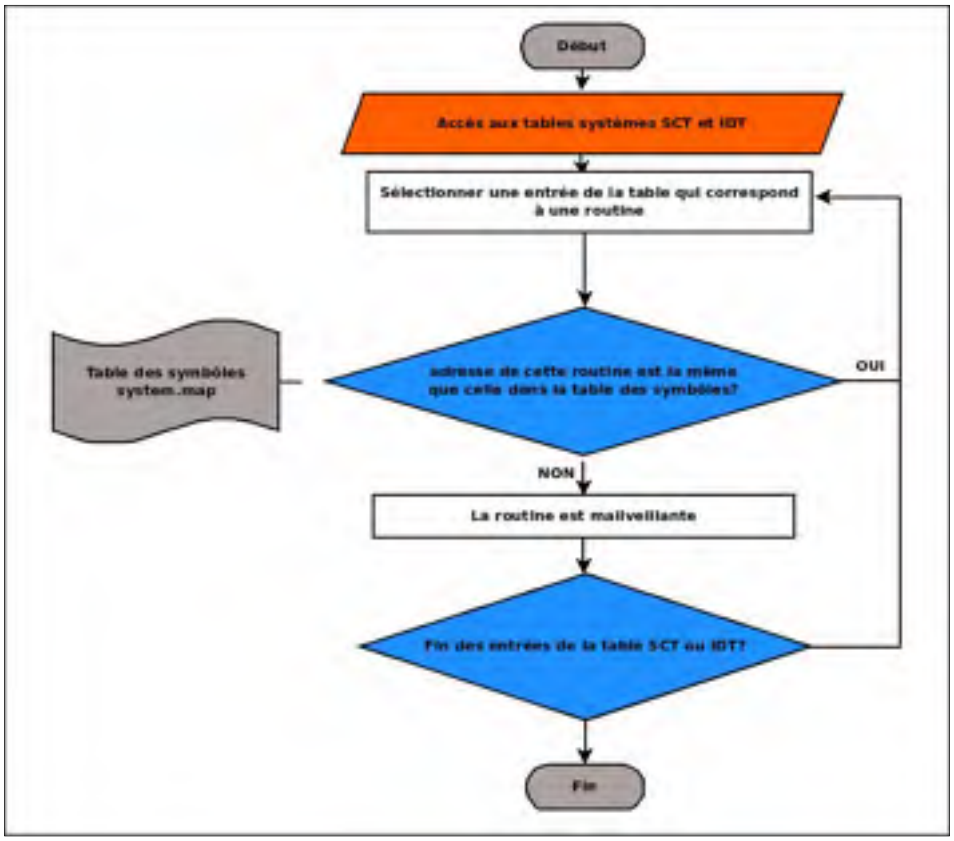

Figure 3.8 Diagramme de détection du détournement
des tables SCT et IDT

3.3.2.2 Détection de la dissimulation des processus

Il n'existe aucune méthode générale pour détecter les attaques des structures de données systèmes puisque ces structures résident dans des sections dynamiques. Dans ce cas, les

contrôleurs d'intégrité ne peuvent rien faire. Les structures de données attaquées doivent être bien étudiées pour définir une façon efficace de détection d'incohérence. Dans cette section, on traite l'attaque de dissimulation des processus soit par le changement de la structure des tâches, soit par le changement des pointeurs de fonctions VFS (voir section 1.3.6). Le traceur LTTng utilise la structure des tâches *task_struct* pour l'affichage des informations liées aux différents processus en cours d'exécution dans une fenêtre de temps bien déterminé. Mais, il n'a pas les instrumentations qui fournissent la liste entière des tâches et des threads dans le noyau.

L'événement *statedump* est l'évènement qui s'exécute au début de traçage, il ne permet pas d'avoir un *snapshot* complet de tâches exécutées. Un nouvel événement devra être ajouté au module *statedump* pour supporter la possibilité d'avoir une image complète de la liste des tâches en exécution dans le système.

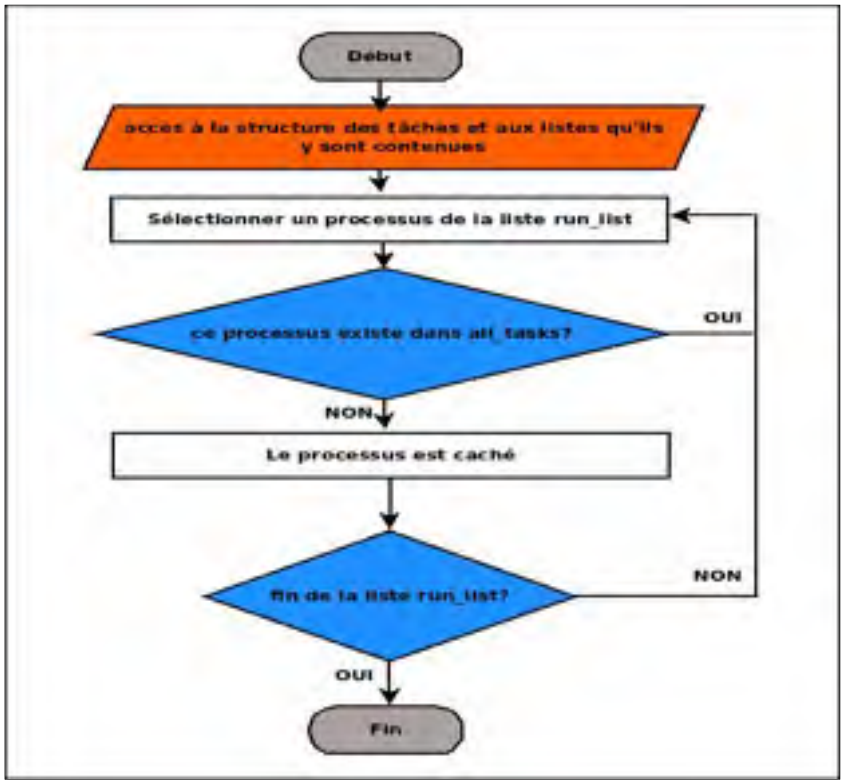

Figure 3.9 Diagramme de détection des processus cachés

Pour effectuer la détection de ce type d'attaque, on peut procéder de trois façons différentes :

- Soit on récupère la liste complète des tâches en cours d'exécutions à partir des deux listes all_tasks et run_list en comparant leurs continus. De cette façon, on peut détecter s'il y a eu une modification de la liste de tâches par un rootkit (voir figure3.9) (Pelaez, 2004; Riley, Jiang et Xu, 2009);
- Soit on utilise la méthode de détection basée sur le crossview. En premier lieu, on récupère la vue de la liste des processus à partir de la structure des tâches au niveau du noyau et l'en compare avec la vue de la liste des processus affichés par la commande ps au niveau utilisateur (de Almeida, 2008; Rutkowska, 2005);
- Soit on combine les deux à la fois. Cette méthode peut nous indiquer que la modification est faite à un autre niveau que la structure des tâches comme la modification des pointeurs de fonctions VFS ou une modification à un niveau supérieur (rootkit niveau utilisateur ou librairie).

3.3.2.3 Détection du mode promiscuité de l'interface réseau

Pour détecter les activités malicieuses sur le réseau, il faut capturer les paquets qui circulent. On a deux types de modes de captures des paquets, le premier est le mode normal où la station connectée au réseau ne reçoit que le trafic qui lui est destiné. Le deuxième est le mode *promiscuous* qui permet de recevoir tout le trafic du réseau. Donc, afin de superviser le réseau, le système doit opérer en mode *promiscuous* (Garfinkel et Rosenblum, 2003; Patil et Meshram, 2012). LTTng permet la détection du mode *promiscuous* des interfaces réseau en se basant sur la l'activité locale des fonctions de la pile réseau du noyau. On peut détecter l'activation du mode *promiscuous* en insérant des *tracepoints* dans la fonction *__dev_set_promiscuity* contenue dans /net/core/dev.c et en vérifiant le paramètre d'entrée *inc* qui correspond à la valeur qui va être ajouté au continu du champ *dev->promiscuity* de la structure *net_device.* Si la valeur du champ est différente de zéro alors le mode *promiscuous* est activé sinon si sa valeur est nulle alors on déduit que le mode activé est le mode normal (Benvenuti, 2006; Reusser et VESELINOVI, 2004). En comparant l'état affiché par la

commande ifconfig avec celui relevé par cette analyse, on déduit alors la présence du rootkit (Manap, 2006; Quynh et Takefuji, 2007).

3.3.2.4 Détection de rootkits qui utilisent les appels systèmes obsolètes

La technique d'utilisation des appels systèmes obsolètes est utilisée par quelques rootkits. Commençant par expliquer la nature de ce rootkit pour arriver au besoin de l'utilisation des appels systèmes obsolètes. Certains rootkits se présentent sous la forme d'un seul exécutable, mais cet exécutable contient, à part son code, un module noyau stocké sous la forme d'un tableau de caractères hexadécimaux. Comme première étape ce module est compilé en échappant la dernière étape d'assemblage. Puis en utilisant l'outil parser on génère du code assembleur contenant dans une seule section les deux sections .text et .bss contenant respectivement le code et les variables globales et statiques. La section .bss est mise dans un nouvel emplacement, plus précisément sous la sous-section .bss_start, les autres sections sont supprimées. On fait à ce stade-là l'assemblage du code afin d'avoir un état final du module. Le résultat de cet assemblage est ensuite passé à l'outil Rip qui supprime l'en-tête ELF afin d'obtenir une image optimale de ce module.

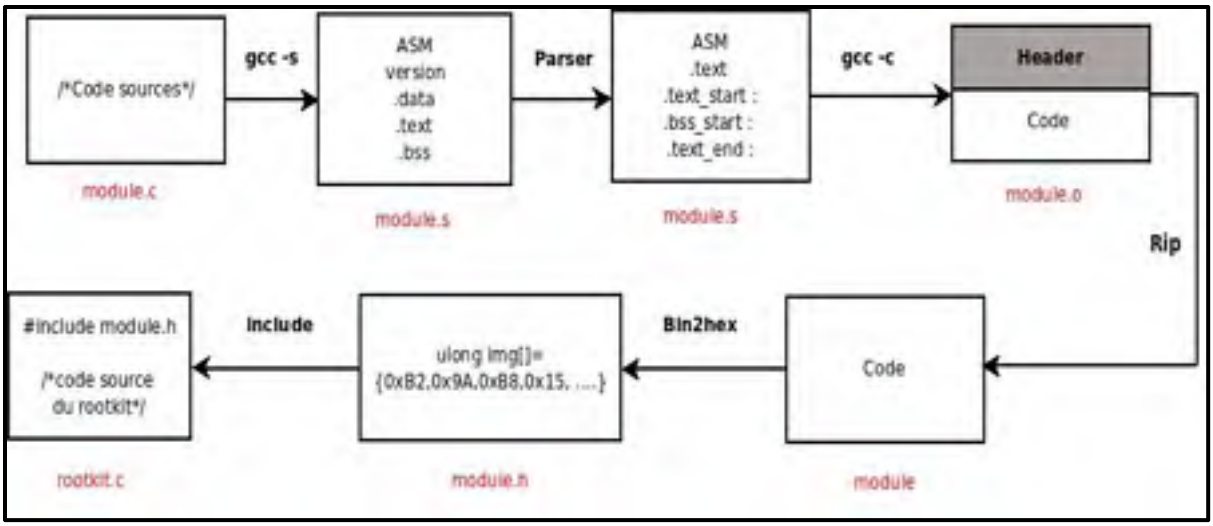

Figure 3.10 Méthode d'intégration du module à injecter dans le noyau
Tirée de Jacob (2007)

Il est enfin réécrit par l'outil bin2hex sous la forme d'un tableau de valeurs hexadécimales. Ce tableau d'hexadécimaux est inclus dans le code du rootkit afin d'être injecté dans le noyau quand ce dernier sera exécuté (Jacob, 2007).

Certains rootkits utilisent la technique LKM pour charger le module associé à leur code source comme expliqué précédemment. Tandis que d'autres rootkits chargent ce type de module injecté dans leur code sans passé par la technique LKM. Plusieurs étapes devraient être suivies afin de pouvoir effectuer l'allocation de mémoire noyau, la résolution des symboles et le chargement du module injecté dans le code du rootkit. En premier lieu, la localisation de kmalloc soit en utilisant get_sym au cas où la technique LKM est activée soit la recherche de cette fonction par son pattern en balayant l'espace mémoire. Une fois trouvée, il faut l'exécuter, mais le problème c'est qu'on ne peut pas l'exécuter à partir de l'espace utilisateur d'où le besoin de l'utilisation d'un appel système. La technique utilisée dans ce cas est la modification d'un appel système obsolète, ce choix est fait, car la manipulation de ce type d'appel système passe inaperçue. En utilisant le détournement des appels systèmes, l'attaquant modifie un des appels système obsolète pour effectuer l'opération d'allocation de l'espace mémoire dans le noyau. Après l'allocation de mémoire, il faut procéder à la relocation des symboles du module injecté dans le rootkit. La dernière étape est maintenant le chargement de ce module par une copie via kmem et le code initial de l'appel système obsolète est remis à l'état initial. Une des façons qu'on peut suivre pour détecter ce type d'attaque est de fixer une liste des appels systèmes obsolètes qu'on active leurs probes dans le noyau à partir de LTTng (la commande est : lttng enable-event syscall_name -k --function syscall_name). Une autre façon de faire est d'activée tous les appels systèmes et vérifier à partir de la trace résultante s'il y on a des appels systèmes obsolètes exécutés (Jacob, 2007). La figure 3.11 présente un diagramme qui décrit la procédure à suivre :

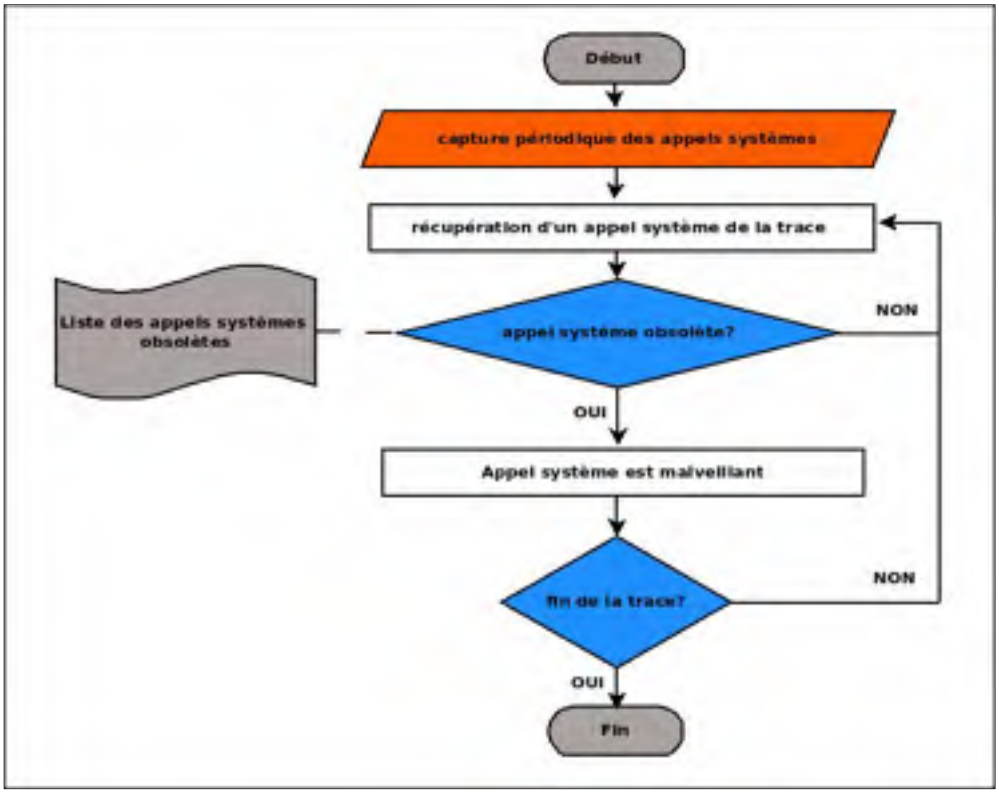

Figure 3.11 Diagramme de détection des rootkits utilisant
les appels systèmes obsolètes

On peut aussi récupérer le nom de processus responsable du déclenchement de cet appel système pour identifier quel logiciel malveillant est responsable de cette attaque.

3.4 Champs d'application des approches de détection proposées

À l'aide de ces approches de détection proposées, on peut concevoir une solution qui utilise LTTng pour la détection des rootkits soit en intégrant des approches de détection des rootkits à cet outil soit en utilisant ce dernier pour générer les données utiles à la détection. Ces données vont être utilisées par un autre outil de détection conçu à la base du projet trace compass. La figure 3.12 montre l'architecture globale de la solution possible basée sur cette étude qui est à concevoir et à tester. Le détecteur par apprentissage automatique basé sur les données dynamiques du noyau peut être présent soit dans le même poste de travail que le traceur LTTng soit dans un serveur dédié au traitement de la trace LTTng collectée à partir d'un ensemble de machines connectées sur le même réseau.

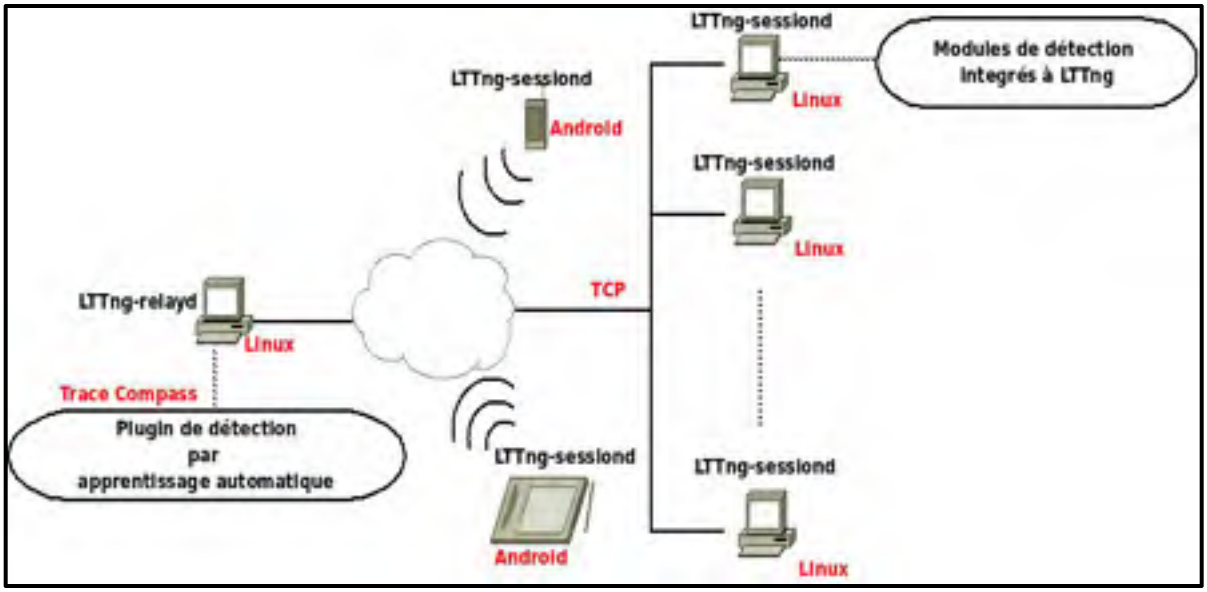

Figure 3.12 Solution possible d'utilisation des approches de détection proposées

Le projet Trace Compass permet le lancement de plusieurs sessions de traçage et leurs configurations sur plusieurs postes de travail ainsi que la récupération de leurs traces.

3.5 Conclusion

Ce chapitre a présenté les différentes approches proposées pour la détection de techniques utilisées par les rootkits du niveau noyau que ce soit par apprentissage automatique ou par intégration des modules de détection dans le traceur *LTTng*. Et il a également répondu à comment ces approches de détection sont-elles appliquées et intégrer dans le traceur *LTTng*.

Le prochain chapitre présente l'étude expérimentale qui a été menée afin de valider l'approche de détection basée sur l'apprentissage automatique à partir des données de performances des appels systèmes qui a été proposée dans ce chapitre.

CHAPITRE 4

EXPÉRIMENTATION ET VALIDATION DE L'APPROCHE BASÉE SUR L'APPRENTISSAGE AUTOMATIQUE

4.1 Introduction

Ce chapitre s'intéresse à l'étude expérimentale qui a été menée afin de valider l'approche de détection par apprentissage automatique (voir section 3.3.1). Ce chapitre est divisé en trois parties :

- La première partie de ce chapitre présente les expérimentations ainsi que les différents données et techniques relatifs à leur mise en œuvre ;
- La deuxième partie décrit les résultats des expérimentations et leurs interprétations;
- La troisième partie décrit la technique de comparaison qui a permis d'identifier un choix final du meilleur algorithme de classification.

4.2 Les choix techniques utilisés dans les expérimentations proposées

Cette section présente la procédure de validation de l'approche de détection par apprentissage automatique. Le séquencement des différentes étapes de cette procédure est décrit dans la figure 4.1 :

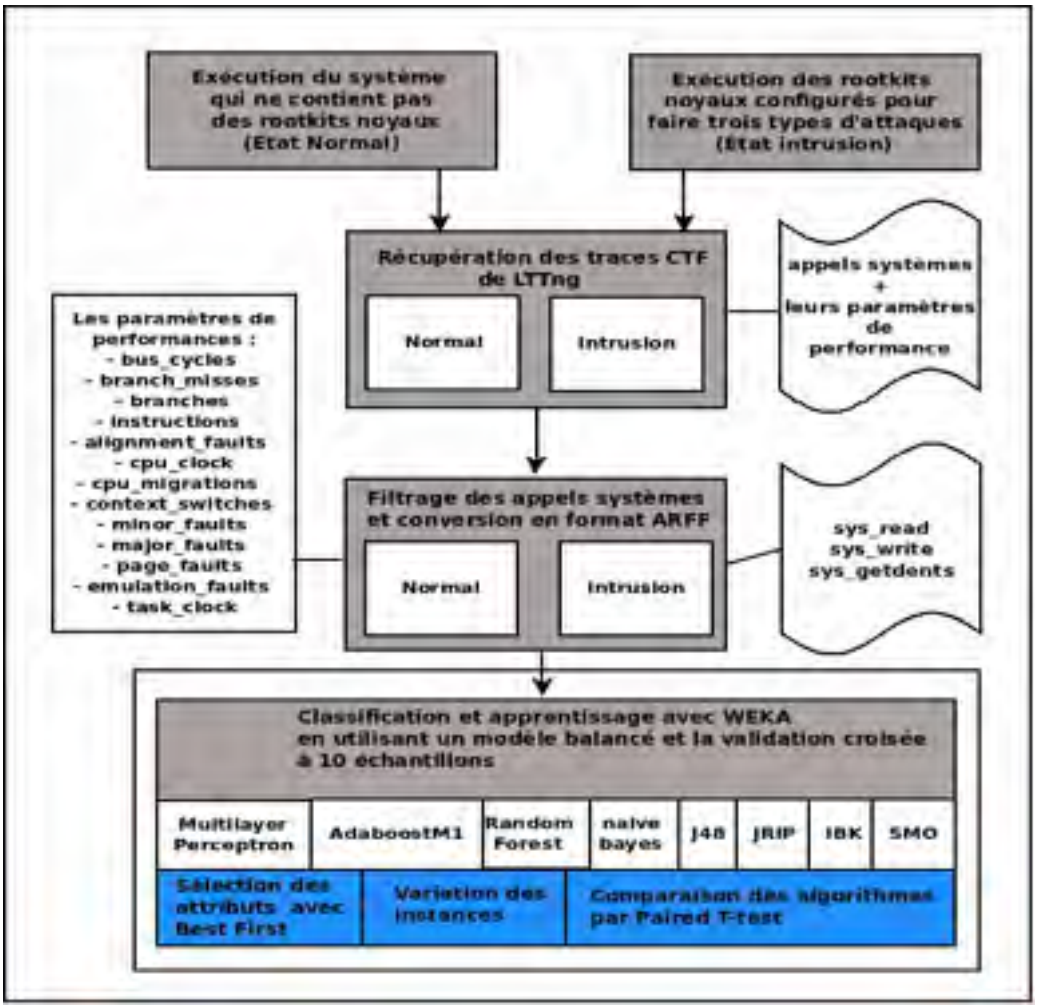

Figure 4.1 Procédure de validation des expérimentations

La validation de la procédure énoncée ci-dessus a été réalisée suivant les axes suivants :

- L'installation de l'environnement et les outils nécessaires pour les expérimentations;
- La récupération et l'analyse des traces de performances des appels systèmes à l'état normal et à l'état infecté par les trois attaques de rootkits;
- L'utilisation d'un modèle balancé pour les deux catégories de données (normal et infecté) pour la classification par WEKA (voir section 2.3.6);
- Une comparaison des résultats des différents classificateurs, qui sont validés par la validation croisée à 10 échantillons, faite à la base de leur stabilité par rapport à la variation du nombre d'instances (Refaeilzadeh, Tang et Liu, 2009);

- Une comparaison des résultats des différents classificateurs avant et après la sélection des attributs par la méthode *Best first* (Chouaib, 2011; Hall et al., 2009);
- La validation du choix du meilleur algorithme de détection par la comparaison des résultats des classificateurs appliqués sur les multiples ensembles de données. Cette comparaison est faite en se basant sur les mesures *F-mesure*, *accuracy*, *precision*, *area under ROC* et le *recall* en appliquant la méthode *paired T-test*. La méthode de validation utilisée dans le paired T-test est la validation croisée à 10 échantillons (Refaeilzadeh, Tang et Liu, 2009).

La méthode de sélection des attributs *Best First* est appliquée sur tous les algorithmes de classification en utilisant la valeur de n =5 qui est la valeur par défaut de la machine d'apprentissage WEKA.

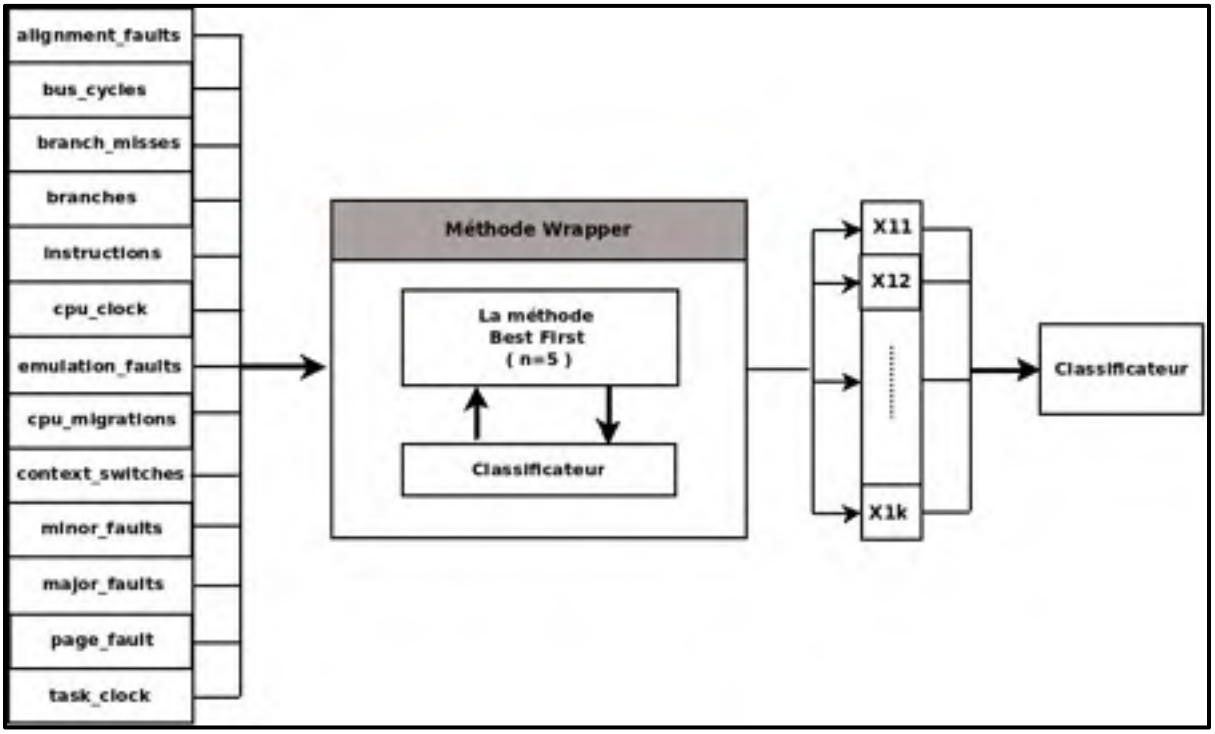

Figure 4.2 Méthode de sélection des attributs *Best first*
Adaptée de Chouaib (2011)

Les compteurs de performance fournie par LTTng qui ont été activés simultanément sont seulement les compteurs qui figurent dans le tableau 4.1 (voir la section 1.2.9 et la section 1.2.10.1). La limitation d'activation des compteurs de performances matérielles est due au fait que le processeur intel core i7 ne peut activer plus que 4 événements matériels à la fois. L'évaluation des algorithmes de classification utilisés est basée sur la comparaison statistique qui utilise la technique de *paired T-test* avec le paramètre alpha=0.05. WEKA utilise le *paired T-test* corrigé comme configuration par défaut (Demšar, 2006; Khorasgan, 2010). Le tableau 4.1 contient les paramètres avec leurs indices qui seront nécessaires pour leur identification après l'application de la méthode de sélection :

Tableau 4.1 Les compteurs de performances utilisés

Les compteurs de performances Matérielles	**Les compteurs de performances logicielles**
10- bus_cycles	1- emulation_faults
11- branch_misses	2- alignment_faults
12- branches	3- cpu_migrations
13- instructions	4- context_switches
	5- minor_faults
	6- major_faults
	7- page_faults
	8- task_clock
	9- cpu_clock

Les résultats des algorithmes appliqués sur les appels systèmes sys_read, sys_write et sys_getdents sont présentés par des tableaux. Les résultats exposés dans le premier tableau présentent les différentes mesures d'évaluation de performance de ces algorithmes en variant le nombre des instances. Le deuxième tableau présente les différentes mesures d'évaluation de performance de ces algorithmes avant et après l'optimisation par l'algorithme *Best first*. La dernière ligne du deuxième tableau identifie les attributs sélectionnés par cette méthode en mentionnant le numéro d'attribut figurant dans le tableau 4.1 et le nombre des attributs

sélectionnés (``attribut : nombre d'attributs``). Le choix de travailler avec 6110 instances, pour faire la sélection, est fait à base des hautes valeurs de performance des algorithmes de classification en utilisant ce volume de données.

4.2.1 Expérimentation basée sur l'attaque par la modification des appels systèmes du Suterusu

Cette expérimentation est basée sur la collecte des compteurs de performances des appels systèmes sys_read et sys_write à l'état normal et à l'état infecté du système. L'attaque étudiée dans cette section est la modification des appels systèmes sys_read et sys_write (voir section 3.3.1).

Résultats et interprétations :

Ci-dessous on va présenter les résultats des différents algorithmes de classification appliqués aux compteurs de performances de l'appel système sys_read.

L'algorithme SMO :

Tableau 4.2 Les résultats du classificateur SMO correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,953	0,047	0,953	0,953	0,953	0,953
2000	0,925	0,076	0,925	0,925	0,924	0,925
4000	0,915	0,226	0,914	0,915	0,911	0,844
6110	0.909	0.091	0.914	0.909	0.909	0.909

Le test sur un nombre variable d'instances d'entrée montre une variation de la performance de l'algorithme SMO d'une valeur allant de 1% à 5%, ceci en augmentant progressivement la valeur des instances. Le tableau 4.3 montre une légère amélioration des valeurs des paramètres d'évaluation de l'algorithme SMO sur l'ensemble de ces valeurs. Nous pouvons

dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

Tableau 4.3 Les résultats du classificateur SMO avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0.909	0.091	0.914	0.909	0.909	0.909
Après la sélection	0.912	0.088	0.918	0.912	0.911	0.912
les attributs sélectionnés	3, 5, 7,12 : 4					

L'algorithme Naive bayes :

Tableau 4.4 Les résultats du classificateur naive bayes correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,832	0,168	0,855	0,832	0,829	0,884
2000	0,854	0,147	0,868	0,854	0,852	0,889
4000	0,782	0,068	0,887	0,782	0,798	0,897
6110	0.847	0.153	0.874	0.847	0.844	0.888

Le test sur un nombre variable d'instances d'entrée montre une variation de la performance de l'algorithme naïve bayes d'une valeur allant de 0.7% à 6.5%, ceci en augmentant progressivement la valeur des instances. Cette instabilité des valeurs de performance montre que cet algorithme est sensible aux variations du nombre d'instances. Le tableau ci-dessous présente les valeurs des paramètres d'évaluation de l'algorithme naïve bayes avant et après la sélection des attributs en utilisant la méthode *Best first*. Le tableau 4.5 montre une augmentation des valeurs des paramètres d'évaluation de l'algorithme naive bayes sur

l'ensemble de ces valeurs après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

Tableau 4.5 Les résultats du classificateur naive bayes avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0.847	0.153	0.874	0.847	0.844	0.888
Après la sélection	0.863	0.137	0.884	0.863	0.861	0.934
les attributs sélectionnés	4, 5, 8, 10,12 : 5					

L'algorithme J48 :

Tableau 4.6 Les résultats du classificateur j48 correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,987	0,013	0,987	0,987	0,987	0,993
2000	0,99	0,01	0,99	0,99	0,99	0,99
4000	0,993	0,012	0,993	0,993	0,993	0,996
6110	0,998	0,002	0,998	0,998	0,998	0,998

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme J48 en augmentant progressivement la valeur des instances. Il présente des hautes valeurs de performance comme le taux de TP qui est de l'ordre 99%. Les valeurs des paramètres d'évaluation de l'algorithme J48, avant et après la sélection des attributs en utilisant la méthode *Best first*, sont présentées dans le tableau 4.7. Ce dernier montre une stabilité des valeurs des paramètres d'évaluation de l'algorithme J48. Il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

Tableau 4.7 Les résultats du classificateur j48 avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,998	0,002	0,998	0,998	0,998	0,998
Après la sélection	0,998	0,002	0,998	0,998	0,998	0,999
les attributs sélectionnés	3, 4, 5, 8, 13 : 5					

L'algorithme IBK :

Tableau 4.8 Les résultats du classificateur IBK correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,998	0,002	0,998	0,998	0,998	0,998
2000	0,998	0,002	0,998	0,998	0,998	0,998
4000	0,998	0,004	0,998	0,998	0,998	0,997
6110	0,998	0,002	0,998	0,998	0,998	0,998

Le test sur un nombre variable d'instances d'entrée montre une stabilité de la performance de l'algorithme IBK, ceci en augmentant progressivement la valeur des instances. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme IBK avant et après la sélection des attributs en utilisant la méthode *Best first*. Le tableau 4.9 montre une dégradation importante des valeurs des paramètres d'évaluation de l'algorithme IBK. L'ensemble des valeurs de performances diminue d'une valeur de 15.5% par rapport à la valeur calculée avant la sélection *Best first*.

Tableau 4.9 Les résultats du classificateur IBK avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,998	0,002	0,998	0,998	0,998	0,998
Après la sélection	0,843	0,157	0,844	0,843	0,843	0,843
Les attributs sélectionnés	8:1					

L'algorithme AdaboostM1 :

Tableau 4.10 Les résultats du classificateur AdaboostM1 correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,982	0,018	0,982	0,982	0,982	0,999
2000	0,9	0,101	0,9	0,9	0,899	0,976
4000	0,969	0,035	0,97	0,969	0,969	0,991
6110	0,845	0,155	0,873	0,845	0,842	0,973

Le test sur un nombre variable d'instances d'entrée montre une variation de la performance de l'algorithme adaboostM1 en augmentant progressivement la valeur des instances. Ceci montre la sensibilité de cet algorithme à la variation de valeur d'instances. Le tableau ci-dessous présente les valeurs des paramètres d'évaluation de l'algorithme adaboostM1 avant et après la sélection des attributs en utilisant la méthode *Best first*. Le tableau 4.11 montre une amélioration des valeurs des paramètres d'évaluation de l'algorithme adaboostM1 de l'ordre de 4.5% sur les taux de TP et FP après l'optimisation de ces attributs.

Tableau 4.11 Les résultats du classificateur AdaboostM1avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,845	0,155	0,873	0,845	0,842	0,973
Après la sélection	0.89	0.11	0.9	0.89	0.889	0.978
les attributs sélectionnés	3, 5, 11,12 : 4					

L'algorithme Random Forest :

Tableau 4.12 Les résultats du classificateur random forest correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,993	0,007	0,993	0,993	0,993	1
2000	0,998	0,003	0,998	0,998	0,997	1
4000	0,998	0,004	0,998	0,998	0,998	1
6110	0,999	0,001	0,999	0,999	0,999	1

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme random forest en augmentant progressivement la valeur des instances. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme random forest avant et après la sélection des attributs en utilisant la méthode *Best first*. Le tableau 4.13 montre une dégradation des valeurs des paramètres d'évaluation de l'algorithme random forest. La plupart des valeurs des paramètres d'évaluations de cet algorithme ont dégradé d'une valeur de 1.3% par rapport à la valeur calculée avant la sélection *Best first*.

Tableau 4.13 Les résultats du classificateur random forest avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,999	0,001	0,999	0,999	0,999	1
Après la sélection	0,986	0,014	0,986	0,986	0,986	0,996
les attributs sélectionnés	1, 2, 9,13 : 4					

L'algorithme Mutlilayer Perceptron :

Tableau 4.14 Les résultats du classificateur multilayer perceptron correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,978	0,022	0,978	0,978	0,978	0,966
2000	0,989	0,011	0,989	0,989	0,989	0,989
4000	0,99	0,028	0,99	0,99	0,99	0,992
6110	0,984	0,016	0,984	0,984	0,984	0,997

Le test sur un nombre variable d'instances d'entrée montre une légère variation de la performance de l'algorithme multilayer perceptron en augmentant progressivement la valeur des instances. Il conserve sa haute performance sur l'ensemble de ces paramètres d'évaluation.

Le tableau ci-dessous présente les valeurs des paramètres d'évaluation de l'algorithme multilayer perceptron avant et après la sélection des attributs en utilisant la méthode *Best first*. Le tableau 4.15 montre une légère dégradation des valeurs des paramètres d'évaluation de l'algorithme multilayer perceptron de l'ordre de 0.2% sur l'ensemble de ces valeurs. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

Tableau 4.15 Les résultats du classificateur multilayer perceptron avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,984	0,016	0,984	0,984	0,984	0,997
Après la sélection	0,982	0,018	0,982	0,982	0,982	0,993
les attributs sélectionnés	1,2,3,4,5,6,7,8,9,10,13 : 11					

L'algorithme JRIP :

Tableau 4.16 Les résultats du classificateur Jrip correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,988	0,012	0,989	0,988	0,988	0,997
2000	0,989	0,011	0,989	0,989	0,989	0,993
4000	0,994	0,014	0,993	0,994	0,993	0,991
6110	0,997	0,003	0,997	0,997	0,997	0,998

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme Jrip en augmentant progressivement la valeur des instances.

Cet algorithme présente des hautes valeurs de performance comme le taux de TP qui est de l'ordre 99%. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme random forest avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.17 Les résultats du classificateur Jrip avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,997	0,003	0,997	0,997	0,997	0,998
Après la sélection	0,996	0,004	0,996	0,996	0,996	0,998
les attributs sélectionnés	1, 3, 5, 9,12 : 5					

Ce tableau montre une stabilité des valeurs des paramètres d'évaluation de l'algorithme Jrip sur l'ensemble de ces valeurs (une légère dégradation de l'ordre de 0.1%). Nous pouvons dire que cet algorithme conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*. Ci-dessous on va présenter les résultats des différents algorithmes de classification appliqués aux paramètres de performances de l'appel système sys_write.

L'algorithme SMO :

Tableau 4.18 Les résultats du classificateur SMO correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,903	0,097	0,907	0,903	0,903	0,903
2000	0,968	0,032	0,968	0,968	0,968	0,968
4000	0,956	0,014	0,963	0,956	0,957	0,971
6110	0,97	0,03	0,972	0,97	0,97	0,97

Le test sur un nombre variable d'instances d'entrée montre une variation de la performance de l'algorithme SMO d'une valeur allant de 1% à 7%, ceci en augmentant progressivement la valeur des instances.

Tableau 4.19 Les résultats du classificateur SMO avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,97	0,03	0,972	0,97	0,97	0,97
Après la sélection	0,972	0,028	0,973	0,972	0,972	0,972
les attributs sélectionnés	3, 4, 8, 9, 10, 11,12 : 7					

Ce tableau montre une légère amélioration des valeurs des paramètres d'évaluation de l'algorithme SMO de l'ordre de 0.2% sur l'ensemble de ces valeurs. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme Naive bayes :

Tableau 4.20 Les résultats du classificateur naive bayes correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,925	0,075	0,925	0,925	0,925	0,986
2000	0,978	0,022	0,978	0,978	0,978	0,995
4000	0,989	0,017	0,989	0,989	0,989	0,994
6110	0,986	0,014	0,986	0,986	0,986	0,997

Le test sur un nombre variable d'instances d'entrée montre une amélioration de la performance de l'algorithme naïve bayes, ceci en augmentant progressivement la valeur des instances. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme naïve bayes avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.21 Les résultats du classificateur naive bayes avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,986	0,014	0,986	0,986	0,986	0,997
Après la sélection	0,988	0,012	0,988	0,988	0,988	0,996
les attributs sélectionnés	4, 5,7 : 3					

Ce tableau montre une légère amélioration des valeurs des paramètres d'évaluation de l'algorithme naive bayes de l'ordre de 0.2 % sur la plupart de ces valeurs. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme J48 :

Tableau 4.22 Les résultats du classificateur j48 correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,988	0,012	0,988	0,988	0,988	0,992
2000	0,995	0,005	0,995	0,995	0,995	0,996
4000	0,998	0,005	0,998	0,998	0,998	0,998
6110	0,997	0,003	0,997	0,997	0,997	0,999

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme J48, ceci en augmentant progressivement la valeur des instances.

Cet algorithme présente des hautes valeurs de performance comme le taux de TP qui est de l'ordre de 99%. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme J48 avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.23 Les résultats du classificateur j48 avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,997	0,003	0,997	0,997	0,997	0,999
Après la sélection	0,998	0,002	0,998	0,998	0,998	0,999
les attributs sélectionnés	5, 8,10 : 3					

Ce tableau montre une légère amélioration des valeurs des paramètres d'évaluation de l'algorithme J48 de l'ordre de 0.1% sur l'ensemble de ces valeurs. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme IBK :

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme IBK, ceci en augmentant progressivement la valeur des instances.

Tableau 4.24 Les résultats du classificateur IBK correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,995	0,005	0,995	0,995	0,995	0,995
2000	0,994	0,006	0,994	0,994	0,994	0,994
4000	0,999	0,001	0,999	0,999	0,999	0,999
6110	0,999	0,001	0,999	0,999	0,999	0,999

Cet algorithme présente des hautes valeurs de performance comme le taux de TP qui est de l'ordre de 99%.

Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme J48 avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.25 Les résultats du classificateur IBK avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,999	0,001	0,999	0,999	0,999	0,999
Après la sélection	0,985	0,015	0,985	0,985	0,985	0,985
Les attributs sélectionnés	8:1					

Ce tableau montre une dégradation des valeurs des paramètres d'évaluation de l'algorithme IBK.

L'ensemble des valeurs de performances diminue d'une valeur de 1.4% par rapport à la valeur calculée avant la sélection *Best first*.

L'algorithme AdaboostM1 :

Tableau 4.26 Les résultats du classificateur AdaboostM1 correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,973	0,027	0,974	0,973	0,973	0,998
2000	0,986	0,014	0,986	0,986	0,986	0,999
4000	0,995	0,015	0,995	0,995	0,994	1
6110	0,991	0,009	0,991	0,991	0,991	0,999

Le test sur un nombre variable d'instances d'entrée montre une amélioration de la performance de l'algorithme adaboostM1 sur l'ensemble des expérimentations, ceci en augmentant progressivement la valeur des instances. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme adaboostM1 avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.27 Les résultats du classificateur AdaboostM1 avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,991	0,009	0,991	0,991	0,991	0,999
Après la sélection	0,994	0,006	0,994	0,994	0,994	0,999
les attributs sélectionnés	5,11 : 2					

Ce tableau montre une légère amélioration des valeurs des paramètres d'évaluation de l'algorithme adaboostM1 de l'ordre de 0.3% sur la plupart de ces valeurs. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme Random Forest :

Tableau 4.28 Les résultats du classificateur random forest correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,993	0,007	0,993	0,993	0,993	1
2000	0,999	0,001	0,999	0,999	0,999	1
4000	0,999	0,002	0,999	0,999	0,999	1
6110	0,999	0,001	0,999	0,999	0,999	1

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme random forest d'une valeur de 0,6% sur l'ensemble des expérimentations. Il conserve sa haute performance sur les valeurs de l'ensemble de ces paramètres d'évaluation. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme random forest avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.29 Les résultats du classificateur random forest avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Mesure**	**ROC Area**
Avant la sélection	0,999	0,001	0,999	0,999	0,999	1
Après la sélection	0,999	0,001	0,999	0,999	0,999	1
les attributs sélectionnés	5, 9,11 : 3					

Ce tableau montre que l'algorithme conserve les mêmes valeurs des paramètres d'évaluation après la sélection des attributs en utilisant la méthode *Best first*.

L'algorithme Multilayer Perceptron :

Tableau 4.30 Les résultats du classificateur multilayer perceptron correspondant à la variation du nombre d'instances

Nombre d'instances	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Mesure**	**ROC Area**
600	0,958	0,042	0,959	0,958	0,958	0,962
2000	0,986	0,014	0,986	0,986	0,986	0,99
4000	0,988	0,019	0,988	0,988	0,988	0,996
6110	0,983	0,017	0,983	0,983	0,983	0,999

Le test sur un nombre variable d'instances d'entrée montre une amélioration de la performance de l'algorithme multilayer perceptron sur l'ensemble des expérimentations. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme multilayer perceptron avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.31 Les résultats du classificateur multilayer perceptron avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,983	0,017	0,983	0,983	0,983	0,999
Après la sélection	0,992	0,008	0,992	0,992	0,992	0,997
les attributs sélectionnés	4, 5, 6, 10, 11, 12 : 6					

Ce tableau montre une légère augmentation des valeurs des paramètres d'évaluation de l'algorithme multilayer perceptron de l'ordre de 0.9% sur la plupart de ces valeurs, mais avec une légère dégradation au niveau de la surface de la courbe ROC de l'ordre de 0.2%. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme JRIP :

Tableau 4.32 Les résultats du classificateur Jrip correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,988	0,012	0,988	0,988	0,988	0,988
2000	0,996	0,004	0,996	0,996	0,996	0,997
4000	0,999	0,003	0,998	0,999	0,998	0,998
6110	0,999	0,001	0,999	0,999	0,999	0,999

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme Jrip d'une valeur de 1% sur l'ensemble des expérimentations. Cet algorithme conserve ses hautes valeurs de performance. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme random forest avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.33 Les résultats du classificateur Jrip avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Mesure**	**ROC Area**
Avant la sélection	0,999	0,001	0,999	0,999	0,999	0,999
Après la sélection	0,997	0,003	0,997	0,997	0,997	0,999
les attributs sélectionnés	3, 5, 7, 9, 10 : 5					

Ce tableau montre une légère dégradation de l'ordre de 0.2% sur la plupart des valeurs des paramètres d'évaluation. Nous pouvons dire qu'il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

Conclusion :

En général, on a un bon taux de détection de l'activité du rootkit sur l'appel système sys_read assuré par les différents algorithmes de l'expérimentation. On trouve que les cinq algorithmes, qui sont J48, JRIP, Random Forest, IBK et MLP ont le plus haut taux de détection en se basant sur les paramètres d'évaluation de ces algorithmes.

Pour l'appel système sys_write, les mêmes algorithmes présentent les meilleurs résultats de détection de l'activité du rootkit déclenché par l'appel système sys_write.

4.2.2 Expérimentation basé sur l'attaque par le détournement des appels systèmes du Kbeast

Cette expérimentation est basé aussi sur la collecte des compteurs de performances des appels systèmes sys_read et sys_write à l'état normal et à l'état infecté du système. L'attaque étudiée dans cette section est le détournement des appels systèmes sys_read et sys_write (voir section 3.3.1).

Résultats et interprétations :

Ci-dessous on va présenter les résultats des différents algorithmes de classification appliqués aux compteurs de performances des appels systèmes sys_read.

L'algorithme SMO :

Tableau 4.34 Les résultats du classificateur SMO correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,967	0,033	0,968	0,967	0,967	0,967
2000	0,948	0,052	0,948	0,948	0,948	0,948
4000	0,953	0,047	0,953	0,953	0,953	0,953
6110	0,964	0,036	0,964	0,964	0,964	0,964

Le test sur un nombre variable d'instances d'entrée montre une variation de la performance de l'algorithme SMO d'une valeur allant de 1% à 2%, ceci en augmentant progressivement la valeur des instances.

Tableau 4.35 Les résultats du classificateur SMO avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,964	0,036	0,964	0,964	0,964	0,964
Après la sélection	0,964	0,036	0,964	0,964	0,964	0,964
les attributs sélectionnés	3, 5, 8, 11, 12, 13 : 6					

Ce tableau montre une stabilité sur l'ensemble des valeurs des paramètres d'évaluation de cet algorithme. Il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme Naive bayes :

Tableau 4.36 Les résultats du classificateur naive bayes correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,947	0,053	0,948	0,947	0,947	0,983
2000	0,762	0,238	0,805	0,762	0,753	0,964
4000	0,91	0,091	0,91	0,91	0,909	0,946
6110	0.91	0.09	0.919	0.91	0.91	0.967

Le test sur un nombre variable d'instances d'entrée montre une dégradation de la performance de l'algorithme naïve bayes d'une valeur allant de 3.7% à 18%, ceci en augmentant progressivement la valeur des instances. Cette instabilité des valeurs de performance montre que cet algorithme est très sensible aux variations du nombre d'instances. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme naïve bayes avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.37 Les résultats du classificateur naive bayes avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0.91	0.09	0.919	0.91	0.91	0.967
Après la sélection	0,921	0,079	0,926	0,921	0,921	0,971
les attributs sélectionnés	3, 5, 8, 10,13 : 5					

Ce tableau montre une amélioration des valeurs des paramètres d'évaluation de l'algorithme naive bayes de l'ordre de 1 % sur la plupart de ces valeurs. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme J48 :

Tableau 4.38 Les résultats du classificateur j48 correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,992	0,008	0,992	0,992	0,992	0,993
2000	0,995	0,005	0,995	0,995	0,995	0,997
4000	0,997	0,003	0,997	0,997	0,997	0,998
6110	0,998	0,002	0,998	0,998	0,998	0,999

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme J48 d'une valeur moyenne de 0,2%, ceci en augmentant progressivement la valeur des instances. Il présente des hautes valeurs de performance comme le taux de TP qui est de l'ordre 99%. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme J48 avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.39 Les résultats du classificateur j48 avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,998	0,002	0,998	0,998	0,998	0,999
Après la sélection	0,997	0,003	0,997	0,997	0,997	0,998
les attributs sélectionnés	3, 7, 10,13 : 4					

Ce tableau montre une légère dégradation des valeurs des paramètres d'évaluation de l'algorithme J48 de l'ordre de 0.1% sur l'ensemble de ces valeurs. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme IBK :

Le test sur un nombre variable d'instances d'entrée montre une stabilité de la performance de l'algorithme IBK, ceci en augmentant progressivement la valeur des instances. Il conserve l'ensemble de ces valeurs de performance.

Tableau 4.40 Les résultats du classificateur IBK correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,995	0,005	0,995	0,995	0,995	0,995
2000	0,997	0,004	0,997	0,997	0,996	0,997
4000	0,997	0,003	0,997	0,997	0,997	0,997
6110	0,997	0,003	0,997	0,997	0,997	0,997

Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme IBK avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.41 Les résultats du classificateur IBK avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,997	0,003	0,997	0,997	0,997	0,997
Après la sélection	0,908	0,092	0,908	0,908	0,908	0,908
les attributs sélectionnés	8:1					

Ce tableau montre une dégradation importante des valeurs des paramètres d'évaluation de l'algorithme IBK. Le taux TP diminue d'une valeur de 9% par rapport à la valeur calculée avant la sélection *Best first*.

L'algorithme AdaboostM1 :

Le test sur un nombre variable d'instances d'entrée montre une dégradation de la performance de l'algorithme adaboostM1 de presque 2%, ceci en augmentant progressivement la valeur des instances.

Tableau 4.42 Les résultats du classificateur AdaboostM1 correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,993	0,007	0,993	0,993	0,993	0,997
2000	0,967	0,033	0,968	0,967	0,967	0,998
4000	0,974	0,026	0,974	0,974	0,974	0,994
6110	0,973	0,027	0,975	0,973	0,973	0,993

Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme adaboostM1 avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.43 Les résultats du classificateur AdaboostM1avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,973	0,027	0,975	0,973	0,973	0,993
Après la sélection	0,974	0,026	0,975	0,974	0,974	0,991
les attributs sélectionnés	4, 5,12 : 3					

Ce tableau montre une légère amélioration des valeurs des paramètres d'évaluation de l'algorithme adaboostM1 de l'ordre de 0.1% sur la plupart de ces valeurs, mais avec une légère dégradation au niveau de la surface de la courbe ROC. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme Random Forest :

Tableau 4.44 Les résultats du classificateur random forest correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,995	0,005	0,995	0,995	0,995	1
2000	0,997	0,003	0,997	0,997	0,997	1
4000	0,999	0,001	0,999	0,999	0,999	1
6110	0.999	0.001	0.999	0.999	0.999	1

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme random forest, ceci en augmentant progressivement la valeur des instances. Il conserve les mêmes valeurs de l'ensemble de ces paramètres d'évaluation.
Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme random forest avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.45 Les résultats du classificateur random forest avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0.999	0.001	0.999	0.999	0.999	1
Après la sélection	0.999	0.001	0.999	0.999	0.999	1
les attributs sélectionnés	1, 3, 5, 8 : 4					

Ce tableau montre une stabilité sur l'ensemble des valeurs des paramètres d'évaluation de l'algorithme random forest.

L'algorithme MultiLayer Perceptron :

Tableau 4.46 Les résultats du classificateur multilayer perceptron correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,993	0,007	0,993	0,993	0,993	1
2000	0,991	0,009	0,991	0,991	0,991	0,999
4000	0,992	0,009	0,992	0,992	0,991	0,996
6110	0.986	0.014	0.986	0.986	0.986	0.99

Le test sur un nombre variable d'instances d'entrée montre une légère dégradation de la performance de l'algorithme multilayer perceptron, ceci en augmentant progressivement la valeur des instances. Il conserve sa haute performance sur l'ensemble de ces paramètres d'évaluation. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme multilayer perceptron avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.47 Les résultats du classificateur multilayer perceptron avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Mesure**	**ROC Area**
Avant la sélection	0.986	0.014	0.986	0.986	0.986	0.99
Après la sélection	0,99	0,01	0,99	0,99	0,99	0,995
les attributs sélectionnés	1, 3, 4, 5,7 ,8,10,11,12,13 : 10					

Ce tableau montre une amélioration des valeurs des paramètres d'évaluation de l'algorithme multilayer perceptron de l'ordre de 1,3% sur l'ensemble de ces valeurs. Il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme JRIP :

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme Jrip d'une valeur moyenne de 0,2%, ceci en augmentant progressivement la valeur des instances.

Tableau 4.48 Les résultats du classificateur Jrip correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,992	0,008	0,992	0,992	0,992	0,993
2000	0,995	0,006	0,995	0,995	0,994	0,997
4000	0,994	0,006	0,994	0,994	0,994	0,996
6110	0,996	0,004	0,996	0,996	0,996	0,998

Cet algorithme présente des hautes valeurs de performance comme le taux de TP qui est de l'ordre 99%. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme Jrip avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.49 Les résultats du classificateur Jrip avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,996	0,004	0,996	0,996	0,996	0,998
Après la sélection	0,997	0,003	0,997	0,997	0,997	0,998
les attributs sélectionnés	3, 5, 6, 7, 9, 10 : 6					

Ce tableau montre une légère augmentation des valeurs des paramètres d'évaluation de l'algorithme Jrip de l'ordre de 0.1% sur l'ensemble de ces valeurs.

Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

Ci-dessous on va présenter les résultats des différents algorithmes de classification appliqués aux paramètres de performances de l'appel système sys_write.

L'algorithme SMO :

Tableau 4.50 Les résultats du classificateur SMO correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,86	0,14	0,877	0,86	0,858	0,86
2000	0,949	0,051	0,95	0,949	0,949	0,949
4000	0,788	0,579	0,763	0,788	0,752	0,604
6110	0,873	0,127	0,899	0,873	0,871	0,873

Le test sur un nombre variable d'instances d'entrée montre une variation de la performance de l'algorithme SMO d'une valeur allant de 8% à 13%, ceci en augmentant progressivement la valeur des instances.

Tableau 4.51 Les résultats du classificateur SMO avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,873	0,127	0,899	0,873	0,871	0,873
Après la sélection	0,885	0,115	0,892	0,885	0,884	0,885
les attributs sélectionnés	10, 11,12 : 3					

Ce tableau montre une légère amélioration des valeurs des paramètres d'évaluation de l'algorithme SMO de l'ordre de 0.8% sur l'ensemble de ces valeurs. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme Naive bayes :

Tableau 4.52 Les résultats du classificateur naive bayes correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,797	0,203	0,849	0,797	0,789	0,913
2000	0,833	0,167	0,875	0,833	0,828	0,956
4000	0,764	0,073	0,882	0,764	0,782	0,898
6110	0.816	0.184	0.865	0.816	0.809	0.914

Le test sur un nombre variable d'instances d'entrée montre une dégradation de la performance de l'algorithme naïve bayes d'une valeur allant jusqu'à 7%. Cette instabilité des valeurs de performance montre que cet algorithme est très sensible aux variations du nombre d'instances. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme naïve bayes avant et après la sélection des attributs en utilisant la méthode *Best first*. Le tableau 4.53 montre une amélioration des valeurs des paramètres d'évaluation de l'algorithme adaboostM1 de l'ordre de 2 % sur la plupart de ces valeurs, mais avec une dégradation importante au niveau de la surface de la courbe ROC d'une valeur de 13%.

Tableau 4.53 Les résultats du classificateur naive bayes avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0.816	0.184	0.865	0.816	0.809	0.914
Après la sélection	0,835	0,165	0,876	0,835	0,831	0,781
les attributs sélectionnés	5,12 : 2					

Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme J48 :

Tableau 4.54 Les résultats du classificateur j48 correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,985	0,015	0,985	0,985	0,985	0,988
2000	0,997	0,004	0,997	0,997	0,996	0,997
4000	0,996	0,003	0,996	0,996	0,996	0,997
6110	0,998	0,002	0,998	0,998	0,998	0,998

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme J48, ceci en augmentant progressivement la valeur des instances. Il présente des hautes valeurs de performance sur l'ensemble de ses valeurs. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme J48 avant et après la sélection des attributs en utilisant la méthode *Best first*. Le tableau 4.55 montre une légère dégradation des valeurs des paramètres d'évaluation de l'algorithme J48 de l'ordre de 0.2% sur la plupart de ses valeurs.

Tableau 4.55 Les résultats du classificateur j48 avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,998	0,002	0,998	0,998	0,998	0,998
Après la sélection	0,996	0,004	0,996	0,996	0,996	0,997
les attributs sélectionnés	7, 8,13 : 3					

Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme IBK :

Tableau 4.56 Les résultats du classificateur IBK correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,982	0,018	0,982	0,982	0,982	0,982
2000	0,993	0,008	0,993	0,993	0,992	0,993
4000	0,999	0,001	0,999	0,999	0,999	0,999
6110	0,999	0,001	0,999	0,999	0,999	0,999

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme IBK, ceci en augmentant progressivement la valeur des instances. Il présente des hautes valeurs de performance comme le taux de TP qui est de l'ordre 99%. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme J48 avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.57 Les résultats du classificateur IBK avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,999	0,001	0,999	0,999	0,999	0,999
Après la sélection	0,778	0,222	0,778	0,778	0,778	0,778
les attributs sélectionnés	8 : 1					

Ce tableau montre une dégradation importante des valeurs des paramètres d'évaluation de l'algorithme IBK. Le taux TP diminue d'une valeur de 22,1% par rapport à la valeur calculée avant la sélection *Best first*.

L'algorithme AdaboostM1 :

Tableau 4.58 Les résultats du classificateur AdaboostM1 correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,903	0,097	0,907	0,903	0,903	0,97
2000	0,954	0,046	0,955	0,954	0,954	0,987
4000	0,902	0,034	0,93	0,902	0,907	0,971
6110	0,932	0,068	0,936	0,932	0,932	0,979

Le test sur un nombre variable d'instances d'entrée montre une variation de la performance de l'algorithme adaboostM1 de presque 3%.

Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme adaboostM1 avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.59 Les résultats du classificateur AdaboostM1 avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,932	0,068	0,936	0,932	0,932	0,979
Après la sélection	0,932	0,068	0,936	0,932	0,932	0,979
les attributs sélectionnés	5, 8,12 : 3					

Ce tableau montre une stabilité des valeurs des paramètres d'évaluation de l'algorithme adaboostM1. Il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme Random Forest :

Tableau 4.60 Les résultats du classificateur random forest correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,995	0,005	0,995	0,995	0,995	1
2000	0,997	0,003	0,997	0,997	0,997	1
4000	0,999	0,002	0,999	0,999	0,999	1
6110	0,999	0,001	0,999	0,999	0,999	1

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme random forest d'une valeur totale de 0,4%, ceci en augmentant progressivement la valeur des instances. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme random forest avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.61 Les résultats du classificateur random forest avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,999	0,001	0,999	0,999	0,999	1
Après la sélection	0.997	0.003	0.997	0.997	0.997	0.999
les attributs sélectionnés	3, 7, 8 : 3					

Ce tableau montre une légère dégradation des valeurs des paramètres d'évaluation de l'algorithme random forest. La plupart des valeurs des paramètres d'évaluations de cet algorithme ont dégradé d'une valeur de 0.2% par rapport à la valeur calculée avant la sélection *Best first*.

L'algorithme MultiLayer Perceptron :

Tableau 4.62 Les résultats du classificateur multilayer perceptron correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,983	0,017	0,983	0,983	0,983	0,982
2000	0,986	0,015	0,986	0,986	0,985	0,991
4000	0,982	0,015	0,982	0,982	0,982	0,998
6110	0,986	0,014	0,986	0,986	0,986	0,999

Le test sur un nombre variable d'instances d'entrée montre une légère variation de la performance de l'algorithme multilayer perceptron, ceci en augmentant progressivement la valeur des instances.

En général, l'algorithme conserve sa haute performance sur l'ensemble de ces paramètres d'évaluation. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme multilayer perceptron avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.63 Les résultats du classificateur multilayer perceptron avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,986	0,014	0,986	0,986	0,986	0,999
Après la sélection	0,982	0,018	0,982	0,982	0,982	0,999
les attributs sélectionnés	4, 5, 7, 9, 11, 12, 13 : 7					

Ce tableau montre une légère dégradation des valeurs des paramètres d'évaluation de l'algorithme multilayer perceptron sur l'ensemble de ces valeurs. Nous pouvons qu'on

général il conserve sa haute performance après l'optimisation de ses attributs en utilisant la méthode de sélection *Best first*.

L'algorithme JRIP :

Tableau 4.64 Les résultats du classificateur Jrip correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,975	0,025	0,975	0,975	0,975	0,976
2000	0,988	0,013	0,988	0,988	0,987	0,995
4000	0,992	0,012	0,992	0,992	0,992	0,993
6110	0,995	0,005	0,995	0,995	0,995	0,998

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme Jrip, ceci en augmentant progressivement la valeur des instances. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme random forest avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.65 Les résultats du classificateur Jrip avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,995	0,005	0,995	0,995	0,995	0,998
Après la sélection	0,997	0,003	0,997	0,997	0,997	0,998
les attributs sélectionnés	3, 7, 9, 10, 11, 13 : 6					

Conclusion :

Les meilleurs algorithmes pour la détection de du syscall Hijacking de l'appel système sys_read et sys_write sont J48, IBK, random forest, JRIP et Multilayer perceptron.

4.2.3 Expérimentation basée sur l'attaque par la dissimulation de processus du Suterusu

Cette expérimentation est faite sur l'appel système sys_getdents en appliquant l'attaque par dissimulation des processus du rootkit Suterusu. L'appel système sys_getdents étudié est celui résultant de l'exécution de la commande ps. Dans les deux cas de test, normal et infecté, on a conservé les mêmes conditions de l'environnement. Tout d'abord, on a exécuté la commande ps afin de collecter les appels systèmes sys_getdents qui lui sont associé (c'est l'état normal). Puis, on a lancé le rootkit pour cacher firefox. Ensuite, on a refait la collection des appels systèmes sys_getdents de la commande ps (c'est l'état infecté).

Résultats et interprétations :

Ci-dessous les résultats d'analyses des différents algorithmes de classification appliquées aux compteurs de performances de l'appel système sys_getdents.

L'algorithme SMO :

Tableau 4.66 Les résultats du classificateur SMO correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,705	0,295	0,804	0,705	0,679	0,705
2000	0,792	0,208	0,799	0,792	0,791	0,792
4000	0,802	0,199	0,856	0,802	0,794	0,802
6110	0,846	0,153	0,877	0,846	0,843	0,846

Le test sur un nombre variable d'instances d'entrée montre une augmentation de la performance de l'algorithme SMO d'une valeur totale de 14 % sur l'ensemble des expérimentations. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme SMO avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.67 Les résultats du classificateur SMO avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,846	0,153	0,877	0,846	0,843	0,846
Après la sélection	0,85	0,149	0,885	0,85	0,846	0,85
les attributs sélectionnés	3, 4, 5, 7, 8, 9, 10, 11, 13 : 9					

Ce tableau montre une légère amélioration des valeurs des paramètres d'évaluation de l'algorithme SMO de l'ordre de 0.4% sur l'ensemble de ses valeurs de performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme Naive bayes :

Tableau 4.68 Les résultats du classificateur naive bayes correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,72	0,28	0,73	0,72	0,717	0,804
2000	0,748	0,252	0,756	0,748	0,746	0,831
4000	0,733	0,267	0,744	0,733	0,73	0,82
6110	0.727	0.272	0.741	0.727	0.723	0.817

Le test sur un nombre variable d'instances d'entrée montre une légère variation de la performance de l'algorithme naïve bayes, ceci en augmentant progressivement la valeur des

instances. Mais on peut considérer qu'on général cet algorithme conserve ses valeurs de performance. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme naïve bayes avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.69 Les résultats du classificateur naive bayes avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0.727	0.272	0.741	0.727	0.723	0.817
Après la sélection	0,782	0,217	0,821	0,782	0,775	0,803
les attributs sélectionnés	3, 4, 8 : 3					

Ce tableau montre une amélioration, après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*, des valeurs des paramètres d'évaluation de cet algorithme sur la plupart de ses valeurs, mais avec une dégradation au niveau de la surface de la courbe ROC.

L'algorithme J48 :

Tableau 4.70 Les résultats du classificateur j48 correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,968	0,032	0,968	0,968	0,968	0,973
2000	0,979	0,021	0,979	0,979	0,979	0,982
4000	0,982	0,018	0,982	0,982	0,982	0,989
6110	0,985	0,015	0,985	0,985	0,985	0,991

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme J48, ceci en augmentant progressivement la valeur des instances. Il présente des hautes valeurs de performance comme le taux de TP qui est de l'ordre 98%. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme J48 avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.71 Les résultats du classificateur j48 avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,985	0,015	0,985	0,985	0,985	0,991
Après la sélection	0,983	0,017	0,983	0,983	0,983	0,989
les attributs sélectionnés	3, 4,11 : 3					

Ce tableau montre une légère dégradation des valeurs des paramètres d'évaluation de l'algorithme J48 de l'ordre de 0.3% sur la plupart de ses valeurs. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme IBK :

Tableau 4.72 Les résultats du classificateur IBK correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,983	0,017	0,983	0,983	0,983	0,983
2000	0,995	0,006	0,995	0,995	0,994	0,995
4000	0,996	0,004	0,996	0,996	0,996	0,996
6110	0,998	0,002	0,998	0,998	0,998	0,998

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme IBK, ceci en augmentant progressivement la valeur des instances. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme J48 avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.73 Les résultats du classificateur IBK avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,998	0,002	0,998	0,998	0,998	0,998
Après la sélection	0,682	0,318	0,682	0,682	0,681	0,68
les attributs sélectionnés	8 : 1					

Ce tableau montre une dégradation importante des valeurs des paramètres d'évaluation de l'algorithme IBK. L'ensemble des valeurs diminue d'une valeur de l'ordre de 31.6% par rapport à la valeur calculée avant la sélection *Best first*.

L'algorithme AdaboostM1 :

Tableau 4.74 Les résultats du classificateur AdaboostM1 correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,783	0,217	0,829	0,783	0,776	0,818
2000	0,803	0,198	0,849	0,803	0,796	0,841
4000	0,794	0,206	0,842	0,794	0,787	0,834
6110	0,805	0,193	0,849	0,805	0,799	0,846

Le test sur un nombre variable d'instances d'entrée montre une petite variation de la performance de l'algorithme adaboostM1 de presque 2%, ceci en augmentant

progressivement la valeur des instances. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme adaboostM1 avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.75 Les résultats du classificateur AdaboostM1avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,805	0,193	0,849	0,805	0,799	0,846
Après la sélection	0,804	0,195	0,844	0,804	0,798	0,806
les attributs sélectionnés	3, 8 : 2					

Ce tableau montre une légère dégradation des valeurs des paramètres d'évaluation de l'algorithme adaboostM1 de l'ordre de 0.1% sur la plupart de ces valeurs et d'une dégradation au niveau de la surface de la courbe ROC de 4%.

Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme Random Forest :

Tableau 4.76 Les résultats du classificateur random forest correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,97	0,03	0,97	0,97	0,97	0,993
2000	0,987	0,014	0,987	0,987	0,986	0,997
4000	0,994	0,006	0,994	0,994	0,994	0,999
6110	0,994	0,006	0,994	0,994	0,994	0,999

Le test sur un nombre variable d'instances d'entrée montre une légère augmentation de la performance de l'algorithme random forest d'une valeur totale de 2.4%, ceci en augmentant progressivement la valeur des instances.

Cet algorithme conserve sa haute performance sur l'ensemble de ces paramètres d'évaluation. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme random forest avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.77 Les résultats du classificateur random forest avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,994	0,006	0,994	0,994	0,994	0,999
Après la sélection	0,982	0,018	0,982	0,982	0,982	0,997
les attributs sélectionnés	4, 8, 10,11 : 4					

Ce tableau montre une légère dégradation des valeurs des paramètres d'évaluation de l'algorithme random forest.

L'algorithme MultiLayer Perceptron :

Tableau 4.78 Les résultats du classificateur multilayer perceptron correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,992	0,008	0,992	0,992	0,992	0,997
2000	0,948	0,052	0,951	0,948	0,948	0,99
4000	0,947	0,054	0,947	0,947	0,946	0,987
6110	0,971	0,028	0,972	0,971	0,971	0,997

Le test sur un nombre variable d'instances d'entrée montre une légère variation de la performance de l'algorithme multilayer perceptron, ceci en augmentant progressivement la valeur des instances. Il conserve sa haute performance sur l'ensemble de ces paramètres d'évaluation. Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme multilayer perceptron avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.79 Les résultats du classificateur multilayer perceptron avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**F-Mesure**	**ROC Area**
Avant la sélection	0,971	0,028	0,972	0,971	0,971	0,997
Après la sélection	0,963	0,037	0,964	0,963	0,963	0,985
les attributs sélectionnés	3, 4, 8, 11 : 4					

Ce tableau montre une légère dégradation des valeurs des paramètres d'évaluation de l'algorithme multilayer perceptron de l'ordre de 2,8% sur la plupart de ses valeurs.

Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

L'algorithme JRIP :

Le test sur un nombre variable d'instances d'entrée montre une augmentation de la performance de l'algorithme Jrip augmentant progressivement la valeur des instances.

Tableau 4.80 Les résultats du classificateur Jrip correspondant à la variation du nombre d'instances

Nombre d'instances	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
600	0,938	0,062	0,938	0,938	0,938	0,954
2000	0,976	0,025	0,976	0,976	0,975	0,988
4000	0,98	0,02	0,98	0,98	0,98	0,986
6110	0,983	0,017	0,983	0,983	0,983	0,989

Le tableau qui suit présente les valeurs des paramètres d'évaluation de l'algorithme random forest avant et après la sélection des attributs en utilisant la méthode *Best first*.

Tableau 4.81 Les résultats du classificateur Jrip avant et après l'application de la méthode *Best first*

Nombre d'instances 6110	TP Rate	FP Rate	Precision	Recall	F-Mesure	ROC Area
Avant la sélection	0,983	0,017	0,983	0,983	0,983	0,989
Après la sélection	0,983	0,017	0,983	0,983	0,983	0,988
les attributs sélectionnés	3, 5, 8, 11,12 : 5					

Ce tableau montre une stabilité des valeurs des paramètres d'évaluation de cet algorithme. Nous pouvons dire qu'on général il conserve sa haute performance après l'optimisation de ces attributs en utilisant la méthode de sélection *Best first*.

Conclusion :

Les résultats montrent que les algorithmes les mieux adaptés pour la détection du rootkit sont J48, JRIP, Random Forest, IBK et MLP.

4.2.4 La validation du choix de l'algorithme le plus adéquat sur les multiples ensembles de données

Pour choisir le meilleur algorithme, on a opté à l'application de la méthode statistique paired T-test sur l'ensemble des *datasets* des expérimentations. Ces *datasets* correspondent aux lignes des tableaux contenant les valeurs des mesures *F-mesure*, *accuracy*, *precision*, *area under ROC* et le *recall*. Ces lignes sont présentées comme suit :

- 1er ligne : Le *dataset* correspondant à l'appel systèmes sys_read du détournement des appels systèmes par kbeast,
- 2 eme ligne : Le *dataset* correspondant à l'appel systèmes sys_write du détournement des appels systèmes par kbeast,
- 3 eme ligne : Le *dataset* correspondant à l'appel systèmes sys_getdents de la dissimulation de processus par suterusu,
- 4 eme ligne : Le *dataset* correspondant à l'appel systèmes sys_read de la modification des appels systèmes par suterusu,
- 5 eme ligne : Le *dataset* correspondant à l'appel systèmes sys_write de la modification des appels systèmes par suterusu.

Le choix de l'algorithme de base pour cette méthode de test est fait à base d'une sélection de l'un des algorithmes présentant les meilleurs résultats sur l'ensemble des *datasets*. Les meilleurs algorithmes sont J48, Jrip, random forest et IBK. Comme algorithme de base en a choisi arbitrairement le random forest. Les critères utilisés pour effectuer la comparaison par paired T-test est basé sur les paramètres F-mesure, area under ROC, precision, recall et accuracy. La somme des résultats des comparaisons effectuées par paired T-test est mise à la dernière ligne pour chaque algorithme. Ces valeurs sont les résultats de comparaison des 7 algorithmes par rapport à l'algorithme de base.

La valeur de chaque comparaison peut être l'une des trois valeurs suivantes :

- v : pour *victory*, ceci indique que l'algorithme est meilleur que l'algorithme de base.

- : le vide indique que le paired T-test ne peut pas décider lequel de ces deux algorithmes est meilleur.
- * : l'astérisque indique que cet algorithme est moins bon que l'algorithme de base.

Ci-dessous une présentation des résultats de comparaison de paired T-test sur l'ensemble des *datasets* correspondants aux expérimentations.

F-mesure :

Tableau 4.82 Les résultats de comparaison par rapport à F-mesure des classificateurs avec multiples datasets en utilisant paired T-test

Random Forest	J48	Naive Bayes	Multilayer Perceptron	SMO	IBk	AdaBoostM1	JRip
1.00	1.00	0.92 *	0.98 *	0.96 *	1.00	0.97 *	1.00 *
1.00	1.00 *	0.77 *	0.99 *	0.85 *	1.00	0.93 *	1.00 *
0,99	0.98 *	0.69 *	0.96 *	0.82 *	1.00 v	0.76 *	0.98 *
1.00	1.00	0.87 *	0.98 *	0.91 *	1.00	0.86 *	1.00 *
1.00	1.00 *	0.99 *	0.99 *	0.97 *	1.00	0.99 *	1.00
Algorithme de base (v/ /*)	(0/2/3)	(0/0/5)	(0/0/5)	(0/0/5)	(1/4/0)	(0/0/5)	(0/1/4)

En se basant sur F-mesure, on constate que les deux meilleurs algorithmes sont IBK et random forest.

Selon le paired T-test, IBK est meilleur que random forest dans la détection dissimulation des processus (*victory* au niveau de la 3 ème ligne).

Area under ROC :

En se basant sur area under ROC, on constate que random forest est le meilleur algorithme, car tous les autres algorithmes sont au moins trois fois moins performants que random forest sur l'ensemble des 5 *datasets*.

Tableau 4.83 Les résultats de comparaison par rapport à area under ROC des classificateurs avec multiples datasets en utilisant paired T-test

Random Forest	J48	Naive Bayes	Multilayer Perceptron	SMO	IBk	AdaBoostM1	JRip
1.00	1.00 *	0.97 *	0.99 *	0.96 *	1.00 *	0.99 *	1.00 *
1.00	1.00 *	0.91 *	1.00 *	0.87 *	1.00	0.98 *	1.00 *
1.00	0.99 *	0.82 *	0.99	0.85 *	1.00	0.85 *	0.99 *
1.00	1.00 *	0.89 *	1.00 *	0.91 *	1.00 *	0.97 *	1.00 *
1.00	1.00 *	0.99 *	1.00 *	0.97 *	1.00 *	1.00 *	1.00 *
Algorithme de base (v/ /*)	(0/0/5)	(0/0/5)	(0/1/4)	(0/0/5)	(0/2/3)	(0/0/5)	(0/0/5)

Precision :

Tableau 4.84 Les résultats de comparaison par rapport à precision des classificateurs avec multiples datasets en utilisant paired T-test

Random Forest	J48	Naive Bayes	Multilayer Perceptron	SMO	IBk	AdaBoostM1	JRip
1.00	1.00	0.86 *	0.98 *	0.96 *	1.00	1.00	1.00
1.00	1.00	1.00	0.99	1.00	1.00	0.97 *	1.00
1.00	0.99 *	0.80 *	0.98	0.98 *	1.00	0.97 *	0.98 *
1.00	1.00	0.77 *	0.99 *	0.87 *	1.00	0.77 *	1.00
1.00	1.00	0.99 *	0.99 *	1.00	1.00	0.99 *	1.00
Algorithme de base (v/ /*)	(0/4/1)	(0/1/4)	(0/2/3)	(0/2/3)	(0/5/0)	(0/1/4)	(0/4/1)

En se basant sur *precision*, on constate qu'encore une fois les deux meilleurs algorithmes sont IBK et random forest.

Le paired T-test ne peut pas décider lequel est meilleur d'entre eux.

Recall :

Tableau 4.85 Les résultats de comparaison par rapport à recall des classificateurs avec multiples datasets en utilisant paired T-test

Random Forest	J48	Naive Bayes	Multilayer Perceptron	SMO	IBk	AdaBoostM1	JRip
1	1.00 *	0.98 *	0.98 *	0.97 *	1.00	0.95 *	1.00
1	1.00	0.63 *	0.98 *	0.75 *	1.00	0.89 *	1.00 *
0,99	0.98 *	0.61 *	0.95 *	0.70 *	1.00 v	0.63 *	0.98 *
1	1.00 *	0.98 *	0.97 *	0.96 *	1.00	0.98 *	1.00
1	1.00 *	0.98 *	0.99 *	0.94 *	1.00	0.99 *	1.00
Algorithme de base (v/ /*)	(0/1/4)	(0/0/5)	(0/0/5)	(0/0/5)	(1/4/0)	(0/0/5)	(0/3/2)

En se basant sur *recall*, on constate que l'IBK est le meilleur algorithme, car tous les autres algorithmes sont au moins deux fois moins performants que l'algorithme de base random forest.

Et aussi, car ce dernier est moins performant que IBK dans la détection de dissimulation des processus (*victory* au niveau de la 3 ème ligne) selon les résultats du paired T-test.

Accuracy :

En se basant sur *accuracy*, on constate que les deux meilleurs algorithmes sont IBK et random forest. Mais ce dernier est encore une fois moins performant que IBK dans la détection de la dissimulation des processus.

Tableau 4.86 Les résultats de comparaison par rapport à accuracy des classificateurs avec multiples datasets en utilisant paired T-test

Random Forest	J48	Naive Bayes	Multilayer Perceptron	SMO	IBk	AdaBoostM1	JRip
99,90	99.78	91.03 *	98.29 *	96.32 *	99.74	97.35 *	99.63 *
99,90	99.74 *	81.56 *	98.64 *	87.31 *	99.90	93.44 *	99.63 *
99,43	98.47 *	72.72 *	96.29 *	84.60 *	99.75 v	80.54 *	98.12 *
99,87	99.73	84.71 *	98.19 *	90.92 *	99.84	84.55 *	99.62 *
99,92	99.72 *	98.64 *	98.88 *	96.97 *	99.92	99.02 *	99.83
Algorithme de base (v/ /*)	(0/2/3)	(0/0/5)	(0/0/5)	(0/0/5)	(1/4/0)	(0/0/5)	(0/1/4)

Comme résultat final, IBK est meilleur que random forest au niveau des paramètres *accuracy*, *recall* et *F-mesure* mais moins performant au niveau de area under ROC. Pour le paramètre *precision*, le paired T-test ne peut pas juger lequel est meilleur. Donc selon les résultats de paired T-test, IBK est le meilleur algorithme pour la détection des trois types d'attaques présentés dans les expérimentations.

4.3 Conclusion et perspectives

Ce chapitre a décrit comment, dans un premier temps, il est possible de valider l'approche de détection des attaques des rootkits par modifications directe et indirecte du comportement des appels systèmes par les rootkits Kbeast et Suterusu. Et comment, dans un deuxième temps, procéder au choix de l'algorithme de classification qui convient le mieux à la détection de ces trois techniques d'attaques. La validation de l'approche de détection en se basant sur les compteurs de performances matérielles et logicielles a été réalisée en suivant deux volets :

- Un premier dans lequel il y a eu la validation de la performance des algorithmes en variant le nombre d'instances d'entrées;

- Un deuxième où il y a eu la validation de performance des algorithmes en appliquant la méthode d'optimisation par sélection d'attributs *Best First* qui a montré l'importance de tous les compteurs de performances utilisés dans notre étude;

Quant au choix du meilleur algorithme de classification, ce dernier a été basé sur :

- Premièrement, l'utilisation de tous les ensembles de données résultant de l'ensemble des expérimentations comme entrée à la méthode de comparaison statistique utilisée dans cette étude qui est le paired T-test;
- Deuxièmement, une sélection de cinq critères d'évaluation des algorithmes de classification qu'on s'est basé pour la validation de notre choix.

4.4 Limite de cette recherche

À l'issue de ce mémoire, nous considérons que les approches proposées et les contributions réalisées constituent des résultats probants permettant l'amélioration du taux de détection des rootkits au niveau du noyau Linux. Malgré ces bons résultats, les approches présentent quelques limitations qui dépendent principalement de l'environnement d'expérimentation et de la nature des attaques étudiées. Pour l'approche de détection basée sur l'apprentissage automatique, les principales limitations sont :

- La limitation du nombre des compteurs de performances matériels qui est due à la nature du processuer intel i7 qui ne peut activer que 4 compteurs à la fois.
- L'approche est dédiée principalement aux systèmes d'exploitation basés sur un noyau linux.
- Les attaques traités sont principalement ceux qui touchent les appels systèmes et les tables systèmes.

La principale limitation de l'approche basée sur l'extension de l'outil LTTng est que cette approche n'effectue que la détection des attaques qui touchent les données statiques du

noyau. Elle ne permet pas la détection des attaques qui dupliquent les tables systèmes du noyau linux.

Nous proposons des solutions pour les résoudre sous forme de perspectives et de recommandations de recherche dans la conclusion générale.

CONCLUSION GÉNÉRALE

Les rootkits mode noyau représentent une menace considérable pour tout système informatique, car ils fournissent à un intrus la possibilité de cacher la présence de son activité malveillante. Considérant la menace des rootkit mode noyau, la collecte de données fiables d'un système compromis devient un problème central dans le domaine de la sécurité informatique.

Afin de remédier à la présente situation, cette recherche propose des approches d'analyse et de détection des rootkits niveau noyau. Les questions suivantes ont été considérées et investiguées :

1. Quelles sont les types de rootkits noyau, les techniques qu'ils utilisent et les mesures du noyau qui peuvent être utilisé pour leur détection?

   Dans notre travail, nous avons focalisé sur la classe des rootkits du niveau noyau et sur les techniques du détournement, de modification des tables et des appels systèmes et de la dissimulation des processus. Les structures de données, les fonctions et les appels systèmes du noyau sont les principales mesures étudiées et qui peuvent être considérés comme des indicateurs de présence des rootkits. Les paramètres de performances des appels systèmes ainsi que leur temps d'exécution permettent de donner un aperçu sur l'état des appels systèmes. Ces différentes mesures peuvent être récupérées à partir du traceur *LTTng*.

2. Quelles sont les techniques d'analyse adéquate à la détection des rootkits noyau?

   L'analyse dynamique est le seul type d'analyse adapté à la nature du traceur *LTTng*. En utilisant cette méthode, nous focalisons sur les données statiques et dynamiques du noyau qui sont liées aux mesures déjà mentionnées.

3. Peut-on intégrer des mécanismes de détection des techniques d'attaques utilisées par les rootkits dans le traceur LTTng à partir des données statiques et des données dynamiques du noyau Linux?

   L'architecture de l'outil LTTng permet d'intégrer plusieurs techniques de détection des attaques des rootkits à savoir :

   - La détection du détournement des tables systèmes comme la SCT et l'IDT.
   - La détection du détournement des appels systèmes.
   - La détection de l'utilisation des appels systèmes obsolètes.
   - La détection des processus cachés.
   - La détection de l'activation du mode *promiscuous* de la carte réseau.

4. Peut-on détecter les techniques d'attaques utilisées par les rootkits par apprentissage automatique basées sur les données dynamiques du noyau Linux? Si oui, comment alors établir cette approche et comment choisir le meilleur algorithme de classification ?

   L'approche de détection par apprentissage automatique basé sur les compteurs de performance matérielle et logicielle est plus prometteuse que celle basée sur le temps d'exécution des appels systèmes. Les algorithmes les plus connus utilisés dans la détection des malwares en général ont été sélectionnés pour valider cette approche. Puis à la base de leurs performances, une comparaison a été faite pour valider le meilleur algorithme.

Les principales contributions de notre travail sont :

1) La proposition des modules de détection des rootkits comportant plusieurs approches qui peuvent être intégrées dans l'architecture du traceur LTTng et qui utilisent ses fonctionnalités.

2) La mise en place d'un environnement d'expérimentation des rootkits niveau noyau.
3) La proposition d'une approche de détection des rootkits basés sur l'apprentissage automatique à base des compteurs de performances matérielles et logicielles recueillis à partir de *LTTng*. Aussi la validation de cette approche en utilisant huit algorithmes parmi les algorithmes les plus utilisés dans ce domaine, leur optimisation et le choix du meilleur parmi eux. La validation est faite contre les techniques d'attaque des appels systèmes utilisés par les rootkits Suterusu et Kbeast, ces techniques sont :

    - Le détournement des appels systèmes
    - La modification des appels systèmes
    - La dissimulation de processus

Une autre contribution découle de cette contribution majeure qui est le développement d'un outil de filtrage des traces LTTng en format CTF et de conversion des données recueillies en format ARFF qui est le format d'entrée de la machine d'apprentissage WEKA.

Le développement d'un outil de détection des rootkits niveau noyau peut avoir un grand impact sur l'industrie. L'avantage d'intégrer les modules de détection contenant les approches mentionnées dans la section 3.4.2 dans l'outil LTTng peut-être très bénéfique dû au fait que ce traceur fonctionne sur multiple distribution Linux et sur multiple architecture matérielle. Ce qui élargit le champ d'application de ce genre de détecteur. Cet outil peut être installé soit sur des postes de travail soit sur des équipements embarqués fonctionnant sur Linux. Les téléphones portables androïde peuvent aussi être cible parfaite pour ce genre d'outil dû la portabilité de LTTng sur ce système.

Un deuxième aspect peut être pris en compte en appliquant la deuxième approche de détection décrite dans la section 3.4.1. À base de cette approche, nous pouvons développer un autre outil dans le projet trace Compass. Cet outil va contenir à la fois l'outil de filtrage et de conversion décrit dans la section 3.4.1.2 et le classificateur sélectionner afin de détecter les attaques décrites dans la section 3.4.1. Il peut être mis dans un poste serveur et centraliser par

la suite le traitement nécessaire pour la détection des rootkits. La trace collectée des différents équipements sera envoyée au serveur contenant le détecteur via la fonctionnalité d'envoi de la trace sur le réseau de LTTng. Ceci va être bénéfique surtout pour les équipements embarqués ayant des ressources matérielles limitées.

À la lumière de ce qui a été traité dans le présent mémoire, plusieurs autres travaux pourraient être faits à base du traceur LTTng comme l'investigation et la validation des approches suivantes :

- L'approche de détection de la prise de commande des rootkits via le réseau par apprentissage automatique à base des mesures réseaux;
- L'approche de détection des malwares à base des mesures systèmes autres que les mesures étudiées soit par processus ou autres et leur optimisation en utilisant diverses méthodes afin de définir soit une séquence d'attributs minimale commune ou d'identifier des patterns d'attributs pour chaque classe de malware.
- Le prototype de détecteur de rootkits à base de trace Compass fonctionnant à base d'apprentissage automatique qui centralise le traitement de détection pour un ensemble d'équipement embarqué soit des téléphones portables ou autres;

Les approches de détection des rootkits proposées par ce mémoire peuvent être étendues à en incluant d'autres approches qui détecte non seulement la présence des rootkits, mais aussi les tentatives d'installation des rootkits, d'escalade de privilège et bien d'autres techniques d'infiltrations, ceci en exploitant et en élargissant les fonctionnalités du traceur LTTng.

BIBLIOGRAPHIE

Alzarooni, KMA. 2012. « Malware variant detection ». UCL (University College London), 212 p.

Anderson, Blake, Daniel Quist, Joshua Neil, Curtis Storlie et Terran Lane. 2011. « Graph-based malware detection using dynamic analysis ». *Journal in Computer Virology,* vol. 7, n° 4, p. 247-258.

Arnold, Thomas Martin. 2011. « A comparative analysis of rootkit detection techniques ». University of Houston-Clear Lake, 113 p.

Barros, Michael Taynnan, Reinaldo Cezar Gomes, de Alencar, Marcelo Sampaio et Anderson Fabiano Costa. 2013. « FEATURE FILTERING TECHNIQUES APPLIED IN IP TRAFFIC CLASSIFICATION ».

Basili, Victor R, et Richard W Selby. 1991. « Paradigms for experimentation and empirical studies in software engineering ». *Reliability Engineering & System Safety,* vol. 32, n° 1, p. 171-191.

Basili, Victor R, Richard W Selby et David H Hutchens. 1986. « Experimentation in software engineering ». *Software Engineering, IEEE Transactions on*, n° 7, p. 733-743.

Benvenuti, Christian. 2006. *Understanding Linux network internals*. " O'Reilly Media, Inc.".

Bornhofen, Fabian. 2013. « Linux Tracing: LTTng vs. SystemTap ». p. 25.

Bouckaert, Remco R, Eibe Frank, Mark A Hall, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann et Ian H Witten. 2010. « WEKA---Experiences with a Java Open-Source Project ». *The Journal of Machine Learning Research,* vol. 11, p. 2533-2541.

Bourque, Pierre. 2000. « Le cadre de Basili : Concepts, extensions et un exemple de son utilisation ».

Brodbeck, Robert C. 2012. *Covert android rootkit detection: Evaluating linux kernel level rootkits on the android operating system*. DTIC Document, 98 p.

Chen, Zhihao. 2006. *Reduced-parameter modeling for cost estimation models*. ProQuest.

Chouaib, Hassan. 2011. « Sélection de caractéristiques: méthodes et applications ». 153 p.

Ciliendo, Eduardo, et Takechika Kunimasa (168). 2007. *Linux performance and tuning guidelines*. IBM, International Technical Support Organization.

Cohen, William W. 1995. « Fast effective rule induction ». In *Proceedings of the twelfth international conference on machine learning*. p. 115-123.

David, Francis M, Ellick M Chan, Jeffrey C Carlyle et Roy H Campbell. 2008. « Cloaker: Hardware supported rootkit concealment ». In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. p. 296-310. IEEE.

de Almeida, André Jorge Marques. 2008. « Rootkits-Detection and prevention ». p. 106.

Demme, John, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan et Salvatore Stolfo. 2013. « On the feasibility of online malware detection with performance counters ». *ACM SIGARCH Computer Architecture News,* vol. 41, n° 3, p. 559-570.

Demšar, Janez. 2006. « Statistical comparisons of classifiers over multiple data sets ». *The Journal of Machine Learning Research,* vol. 7, p. 1-30.

Desfossez, Julien, Justine Dieppedale et Gabriel Girard. 2011. « Stealth malware analysis from kernel space with Kolumbo ». *Journal in computer virology,* vol. 7, n° 1, p. 83-93.

Desnoyers, M. 2011. « Common trace format (ctf) specification ». < http://www.efficios.com/ctf >.

Desnoyers, Mathieu. 2009. « Low-impact operating system tracing ». École Polytechnique de Montréal.

Dinaburg, Artem, Paul Royal, Monirul Sharif et Wenke Lee. 2008. « Ether: malware analysis via hardware virtualization extensions ». In *Proceedings of the 15th ACM conference on Computer and communications security*. p. 51-62. ACM.

Elhadi, Ammar Ahmed E, Mohd Aizaini Maarof et BI Barry. 2013. « Improving the detection of malware behaviour using simplified data dependent api call graph ». *Int J Secur Appl,* vol. 7, n° 5, p. 29-42.

Fahem, Rafik. 2012. « Points de trace statiques et dynamiques en mode noyau ». École Polytechnique de Montréal, 79 p.

Firdausi, Ivan, Charles Lim, Alva Erwin et Anto Satriyo Nugroho. 2010. « Analysis of machine learning techniques used in behavior-based malware detection ». In *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*. p. 201-203. IEEE.

Fournier, Pierre-Marc. 2009. « Analyse automatisée des causes de blocage de processus à partir d'une trace d'exécution ». École Polytechnique de Montréal, 103 p.

Gandotra, Ekta, Divya Bansal et Sanjeev Sofat. 2014. « Malware analysis and classification: A survey ». *Journal of Information Security,* vol. 2014, p. 9.

Garfinkel, Tal, et Mendel Rosenblum. 2003. « A Virtual Machine Introspection Based Architecture for Intrusion Detection ». In *NDSS*. Vol. 3, p. 191-206.

Gebai, Mohamad, Francis Giraldeau et Michel R Dagenais. 2014. « Fine-grained preemption analysis for latency investigation across virtual machines ». *Journal of Cloud Computing: Advances, Systems and Applications,* vol. 3, n° 1, p. 41.

Gregg, Brendan D. 2015. « perf Examples ». < http://www.brendangregg.com/perf.html >.

Hall, Mark, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann et Ian H Witten. 2009. « The WEKA data mining software: an update ». *ACM SIGKDD explorations newsletter,* vol. 11, n° 1, p. 10-18.

Idika, Nwokedi, et Aditya P Mathur. 2007. « A survey of malware detection techniques ». *Purdue University,* vol. 48.

Jacob, G. 2007. « Technologie Rootkit sous Linux/Unix ». *Linux Magazine, Virus UNIX, GNU/Linux & Mac OS X*, p. 47.

Joy, Jestin, et Anita John. 2011. *A host based kernel level rootkit detection mechanism using clustering technique*. Coll. « Trends in Computer Science, Engineering and Information Technology ». Springer, 564-570 p.

Kendall, Kris, et Chad McMillan. 2007. « Practical malware analysis ». In *Black Hat Conference, USA*. p. 10.

Khorasgan, Reihaneh Rabbany. 2010. « Comparison of Different Classification Methods ». In., p. 2.

Kolter, Jeremy Z, et Marcus A Maloof. 2004. « Learning to detect malicious executables in the wild ». In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. p. 470-478. ACM.

Lacombe, Eric. 2009. « Sécurité des noyaux de systèmes d'exploitation ». INSA de Toulouse, 177 p.

Lacombe, Éric, Frédéric Raynal et Vincent Nicomette. 2007. « De l'invisibilité des rootkits: application sous Linux ». In *Eric Filiol, éditeur: Actes du 5ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'07), Rennes, France*. p. 29.

Laperle, Marc-André. 2015. « Analyzing Eclipse Applications with Trace Compass ». In *EclipseCon NA 2015*. (San Francisco), p. 50. < https://www.eclipsecon.org/na2015/sites/default/files/slides/EclipseConAmerica2015.pdf >.

Levine, JF, Julian B Grizzard et Henry L Owen. 2006. « Detecting and categorizing kernel-level rootkits to aid future detection ». *Security & Privacy, IEEE,* vol. 4, n° 1, p. 24-32.

Levine, John, Julian Grizzard et Henry Owen. 2004. « A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table ». In *Information Assurance Workshop, 2004. Proceedings. Second IEEE International*. p. 107-125. IEEE.

Lineberry, Anthony. 2009. « Malicious Code Injection via/dev/mem ». *Black Hat Europe*, p. 11.

LTP-developers. 2012. « linux test project ». < http://linux-test-project.github.io/ >. Consulté le 12-01-2013.

Malone, Corey, Mohamed Zahran et Ramesh Karri. 2011. « Are hardware performance counters a cost effective way for integrity checking of programs ». In *Proceedings of the sixth ACM workshop on Scalable trusted computing*. p. 71-76. ACM.

Manap, Saliman. 2006. « Rootkit: Attacker undercover tools ». < https://cyber-defense.sans.org/resources/papers/gsec/rootkit-attacker-undercover-tools-102458 >.

Marangozova-Martin, Vania, et Generoso Pagano. 2013. « Gestion de traces d'exécution pour le systèmes embarqués: contenu et stockage ».

MARTIN, Benjamin, Romain PAULIAT et Alexandre PELLETIER. 2005. « ROOTKIT ET NOYAU LINUX 2.6. 28 Etat de l'art et détournement d'appels système ». p. 26.

Nguyen, Thuy TT, et Grenville Armitage. 2008. « A survey of techniques for internet traffic classification using machine learning ». *Communications Surveys & Tutorials, IEEE,* vol. 10, n° 4, p. 56-76.

« Paired t-tests ». 2004. < http://www.statstutor.ac.uk/types/quick-reference-leaflet/paired-t-tests/ >. Consulté le 15-12-2014.

Park, Younghee, Douglas Reeves, Vikram Mulukutla et Balaji Sundaravel. 2010. « Fast malware classification by automated behavioral graph matching ». In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*. p. 45. ACM.

Patil, Suchita, et BB Meshram. 2012. « Network Intrusion Detection and Prevention techniques for DoS attacks ». *International Journal of Scientific and Research Publication,* vol. 2, n° 7.

Pauna, Adrian. 2012. « Improved self adaptive honeypots capable of detecting rootkit malware ». In *Communications (COMM), 2012 9th International Conference on*. p. 281-284. IEEE.

Pelaez, Raul Siles. 2004. « Linux kernel rootkits: protecting the system's" ring-zero" ». *GIAC Unix Security Administrator (GCUX)*, p. 169.

Preda, Mila Dalla, Mihai Christodorescu, Somesh Jha et Saumya Debray. 2007. « A semantics-based approach to malware detection ». *ACM SIGPLAN Notices,* vol. 42, n° 1, p. 377-388.

Quynh, Nguyen Anh, et Yoshiyasu Takefuji. 2007. « Towards a tamper-resistant kernel rootkit detector ». In *Proceedings of the 2007 ACM symposium on Applied computing*. p. 276-283. ACM.

Rabek, Jesse C, Roger I Khazan, Scott M Lewandowski et Robert K Cunningham. 2003. « Detection of injected, dynamically generated, and obfuscated malicious code ». In *Proceedings of the 2003 ACM workshop on Rapid malcode*. p. 76-82. ACM.

Ramaswamy, Ashwin. 2008. « Detecting kernel rootkits ». p. 30.

Refaeilzadeh, Payam, Lei Tang et Huan Liu. 2009. « Cross-validation ». In *Encyclopedia of database systems*. p. 532-538. Springer.

Reusser, Halm, et RANKO VESELINOVI. 2004. « Promiscuous Mode Detector ».

Revathi, S, et A Malathi. 2014. « Detecting User-To-Root (U2R) Attacks Based on Various Machine Learning Techniques ». p. 3.

Rhee, Junghwan, Ryan Riley, Dongyan Xu et Xuxian Jiang. 2009. « Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring ». In *Availability, Reliability and Security, 2009. ARES'09. International Conference on*. p. 74-81. IEEE.

Rieck, Konrad, Philipp Trinius, Carsten Willems et Thorsten Holz. 2011. « Automatic analysis of malware behavior using machine learning ». *Journal of Computer Security,* vol. 19, n° 4, p. 639-668.

Riley, Ryan, Xuxian Jiang et Dongyan Xu. 2009. « Multi-aspect profiling of kernel rootkit behavior ». In *Proceedings of the 4th ACM European conference on Computer systems*. p. 47-60. ACM.

Rioult, François. 2011. « Interprétation graphique de la courbe ROC ». In *Extraction et Gestion des Connaissances (EGCâ€™ 11)*. p. 6 p.

Rutkowska, Joanna. 2005. « Thoughts about cross-view based rootkit detection ». *Unveröffentlichtes Memorandum*, p. 3.

Rutkowski, Jan K. 2002. « Execution path analysis: finding kernel based rootkits ». < http://phrack.org/issues/59/10.html >.

Salzman, Peter Jay, Michael Burian et Ori Pomerantz. 2001. « The linux kernel module programming guide ». *TLDP: http://tldp. org/LDP/lkmpg/2.4/html*, p. 82.

Santos, Igor, Jaime Devesa, Félix Brezo, Javier Nieves et Pablo Garcia Bringas. 2013. « Opem: A static-dynamic approach for machine-learning-based malware detection ». In *International Joint Conference CISIS'12-ICEUTE´ 12-SOCO´ 12 Special Sessions*. p. 271-280. Springer.

Santos, Igor, Javier Nieves et Pablo G Bringas. 2011. « Semi-supervised learning for unknown malware detection ». In *International Symposium on Distributed Computing and Artificial Intelligence*. p. 415-422. Springer.

Schultz, Matthew G, Eleazar Eskin, Erez Zadok et Salvatore J Stolfo. 2001. « Data mining methods for detection of new malicious executables ». In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. p. 38-49. IEEE.

Shah, Alkesh, et J Giffin. 2008. *Analysis of rootkits: Attack approaches and detection mechanisms*. Technical report, Georgia Institute of Technology, 7 p.

Shields, Tyler. 2008. « Survey of Rootkit Technologies and Their Impact on Digital Forensics ». *Personal Communication*, p. 11.

Sigillito, VG, SP Wing, LV Hutton et KB Baker. 1989. « 1 Objectif ». *networks,* vol. 10, p. 262-266.

Sun, He, Kun Sun, Yuewu Wang, Jiwu Jing et Sushil Jajodia. 2014. « TrustDump: Reliable Memory Acquisition on Smartphones ». In *Computer Security-ESORICS 2014*. p. 202-218. Springer.

Tian, Ronghua, Lynn Margaret Batten et SC Versteeg. 2008. « Function length as a tool for malware classification ». In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*. p. 69-76. IEEE.

Tian, Ronghua, MR Islam, Lynn Batten et Steven Versteeg. 2010. « Differentiating malware from cleanware using behavioural analysis ». In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*. p. 23-30. IEEE.

Vandeven, Sally. 2014. *Linux Rootkit Detection With OSSEC* 41 p.

Vöcking, Heye. 2012. « Performance analysis using Great Performance Tools and Linux Trace Toolkit next generation ». p. 17.

Wang, Xueyang, et Ramesh Karri. 2013. « Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters ». In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*. p. 1-7. IEEE.

Wang, Xueyang, et Ramesh Karri. 2014. « Detecting Kernel Control-Flow Modifying Rootkits ». In *Network Science and Cybersecurity*. p. 177-187. Springer.

Weaver, Vincent M. 2013. « Linux perf_event features and overhead ». In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*. p. 80.

Xia, Yubin, Yutao Liu, Haibo Chen et Binyu Zang. 2012. « CFIMon: Detecting violation of control flow integrity using performance counters ». In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. p. 1-12. IEEE.

Yao, Wang. 2009. « Rootkit on Linux x86 v2.6 ». < http://fr.slideshare.net/fisher.w.y/rootkit-on-linux-x86-v26 >.