Analyse de la mémoire vive

Importante dans le domaine de l'informatique d'enquête (*cyberforensics*), l'analyse de la mémoire vive (AMV) fait partie de la famille de l'analyse dynamique (AD). Il s'agit d'une approche éprouvée pour effectuer l'analyse des maliciels sur les ordinateurs personnels. Hale Ligh *et al.* [58] ont rédigé un livre complet couvrant l'acquisition et l'analyse de la mémoire vive pour les SE Windows, Linux et Mac OS X principalement. Sur Android, plusieurs travaux (p.ex. [80, 113, 118]) ont démontré qu'il était possible d'extraire les SMS, des informations sur l'état interne du noyau Linux et les classes Java chargées dans l'application par l'AMV. Ce chapitre détaille l'état de la recherche sur l'AMV portant sur Android.

La section 3.1 présente les principales techniques d'acquisition de la mémoire vive présentes sur Android. La section 3.2 fait état des techniques d'analyse applicables à ces captures. L'outil d'AMV appelé *Volatility* et les approches l'utilisant y sont également présentés.

3.1 Acquisition

L'acquisition de la mémoire vive est l'action d'extraire la mémoire volatile d'un appareil en exécution et de la copier dans un fichier appelé capture. Une considération importante de l'acquisition décrite dans les travaux de Chan *et al.* [23] ainsi que Solomon *et al.* [109], est que les données contenues en mémoire vive peuvent être perdues si elles ne sont pas acquises rapidement après la survenue de l'évènement d'intérêt. Le choix de l'algorithme de gestion de la mémoire utilisée dans le SE, l'espace en mémoire disponible, le type d'opérations effectuées sur le système ainsi que la charge de travail du processeur peuvent affecter l'espérance de vie des données qui y sont contenues. Pour ces raisons, il est important qu'un processus d'acquisition soit rapide et affecte lui-même le moins possible ce contenu lors de l'opération de capture de la mémoire vive afin d'éviter la corruption des données d'intérêt.

L'un des différenciateurs des techniques disponibles est l'impact sur le contenu de la mémoire vive. L'étude de Aljaedi *et al.* [6] a démontré que l'acquisition de la mémoire vive par une

application exécutée à même la plateforme analysée peut entraîner une dégradation des données d'intérêt en mémoire vive. Cela se produit lorsqu'une application réserve une plage de mémoire pour y stocker des données sensibles puis libère ces emplacements pour des allocations futures. Dans cette situation, le contenu n'est effacé que lorsque du nouveau contenu y est réécrit. Comme l'utilisation d'une application d'acquisition de la mémoire vive nécessite qu'elle y soit elle-même chargée, il se peut que les espaces en mémoire qu'elle vient occuper aient appartenu à de telles données d'intérêts. Ils sont alors réécrits et irrécupérables à l'AMV. Pour cette raison, il est nécessaire de minimiser la quantité d'espace mémoire qui est affectée par le programme d'acquisition.

Carrier et Grand [19] ont proposé l'utilisation de matériels dédiés de type Peripheral Component Interconnect (PCI). Il a été également démontré possible d'utiliser des périphériques déjà présents sur une plateforme afin d'acquérir la mémoire vive. Notamment, les études de Gladyshev et Almansoori [43] et Zhang *et al.* [137] ont démontré cette capacité par l'utilisation du FireWire alors que Balogh et Mydlo [11] se sont intéressés à utiliser la carte réseau d'un système pour cette tâche. L'utilisation de matériel dédié permet de lire l'entièreté de la mémoire vive, et ce, sans nécessiter l'intervention du SE et du microprocesseur. En effet, l'interaction avec la mémoire vive s'effectue à l'aide d'une composante appelée Direct Memory Access (DMA) permettant à un périphérique de lire ou écrire directement avec la mémoire vive de façon indépendante de l'exécution du microprocesseur. Cette approche permet d'éviter l'envoi d'instructions au microprocesseur nécessitant des lectures et écritures en mémoire vive pour l'acquisition ce qui limite les risques de corruption des données qui s'y trouvent. Cette technique nécessite néanmoins la présence ou l'ajout de composantes électroniques sur le système ciblé et n'a pas été répliquée sur des appareils mobiles Android.

Une autre technique d'acquisition utilisée par Breeuwsma [15] fait appel au bus Joint Test Action Group (JTAG). Le JTAG est, à l'origine, un outil de développement permettant de charger un microprogramme sur un microcontrôleur et d'en faire le déverminage. Il permet d'interrompre l'exécution du microprocesseur pour contrôler son état interne, comme la valeur de ses registres ou des plages mémoires qu'il contient. Le contrôle de l'exécution du microprocesseur est l'une des fonctionnalités du JTAG qu'il est possible d'utiliser pour interrompre l'exécution de celui-ci et de lire l'entièreté de sa mémoire vive. Cette approche peu utilisée dans la littérature demande un accès physique au microcontrôleur de l'appareil visé. Également, elle peut nécessiter des modifications physiques comme des soudures de précision pour relier un connecteur JTAG au microcontrôleur. Ces altérations peuvent entraîner la destruction des composantes en cas d'erreur de manipulations. De plus, certains appareils n'exposent pas de connecteur JTAG ou celui-ci a été désactivé de façon permanente une fois l'installation d'usine terminée.

Sur Android, l'approche matérielle est généralement limitée puisque les plateformes mobiles, principal marché d'utilisateurs du SE, sont peu propices à l'ajout ou l'utilisation de matériel

d'acquisition de la mémoire vive. Pour cette raison, la plupart des approches relevées dans la littérature sont de nature logicielle. *dd* est l'un de ces logiciels utilisés pour faire l'acquisition de la mémoire vive. Il s'agit d'un utilitaire permettant de faire une copie binaire intégrale d'un espace de stockage (volatile ou non) vers un fichier ou un autre périphérique. Il est notamment utilisé pour faire la copie d'image de disque d'un média à l'autre. Il peut être également utilisé pour faire la copie du contenu de la mémoire vive exposée par le noyau Linux à un utilisateur avec les privilèges *root*. La copie s'opère généralement directement sur la mémoire vive, représentée sous la forme d'un fichier virtuel sur Linux (p. ex. */dev/fmem*). Pour ce faire, le noyau du SE doit avoir été compilé avec l'option correspondante pour en activer le support, sans quoi le fichier virtuel représentant la mémoire vive n'existera pas.

Il a été observé par Sylve *et al.* [113] que l'utilisation de *dd* modifie une quantité appréciable de champs en mémoire lorsqu'il est exécuté. Pour un émulateur Android possédant 512 Mo de mémoire vive, l'utilisation de *dd* provoque des écritures sur près de 20% de la totalité du contenu de la mémoire vive, détruisant toute information d'intérêt qui s'y trouve. La propension du SE à réécrire sur des plages mémoires pouvant contenir des données d'intérêt diminue avec l'augmentation de la quantité d'espace mémoire disponible. Plus le système dispose d'espace mémoire de libre, moins il est nécessaire de réutiliser des plages mémoires ayant déjà servi pour faire l'allocation de nouveaux espaces mémoire.

Il est également possible d'acquérir la mémoire vive d'un processus par l'utilisation d'un dévermineur comme *gdb* [44]. Sur Android, il est nécessaire d'avoir les privilèges de l'utilisateur *root* pour connecter *gdb* à un processus en exécution et y lire la mémoire vive si ce processus n'est pas un enfant du processus lançant *gdb*. De plus, cette méthode est limitée à un ou plusieurs processus, mais ne permet pas de cibler le noyau Linux lui-même à moins qu'il n'ait été recompilé pour supporter cette option. Si tel est le cas, il est possible de contrôler l'ensemble de son exécution par un dévermineur distant, mais aux coûts d'importants ralentissements pouvant interférer avec des opérations sensibles aux retards (p. ex. communications réseaux, périphériques, etc.).

Le dévermineur d'applications Android permet également d'analyser les variables en mémoire de celles-ci lors de leur exécution. Une application doit avoir été prévue pour être déverminée au moment de sa compilation, sans quoi le dévermineur ne sera pas autorisé à la contrôler. Cette approche ne permet pas d'observer l'ensemble du contenu de la mémoire vive du SE, des autres processus externes aux applications Android et des autres applications qui ne sont pas configurées pour être déverminables.

L'absence de matériel d'acquisition dédié pour plateforme Android, l'impact négatif de *dd* sur la mémoire vive et les contraintes des dévermineurs ont poussé l'exploration d'autres approches. Deux principales techniques d'acquisition de la mémoire vive se démarquent sur Android : l'acquisition par l'utilisation de l'outil Linux Memory Extraction (LiME) et par

l'utilisation d'une MV ou d'un émulateur. Elles sont présentées à la section suivante.

3.1.1 LiME

Sylve *et al.* [113] ont proposé *dmd*, devenu Linux Memory Extraction (LiME)[3], un LKM capable de faire l'acquisition de la mémoire vive sur les systèmes Linux. Il est possible d'utiliser ce module pour Android puisque le SE possède un noyau Linux. Le module LiME capture la mémoire vive de la façon suivante :

1. Le LKM récupère la première structure du noyau *iomem_resource* contenant la valeur de l'adresse d'un espace d'adressage atteignable. Ce-dernier peut désigner un emplacement en mémoire vive ou correspondre aux registres utilisables d'un périphérique. En effet, les registres internes de certains périphériques peuvent être également atteints par l'utilisation de l'adressage en mémoire.
2. La lecture de la structure du noyau *iomem_resource* permet de déterminer si l'espace mémoire en question correspond bien à un élément de la mémoire vive du système (attribut *iomem_resource->name == "System RAM"*). Dans la négative, celui-ci n'est pas d'intérêt et le module passe au prochain espace mémoire (attribut *iomem_resource->sibling*) et refait la même validation. Dans l'affirmative, il poursuit à l'étape suivante.
3. Les adresses physiques en mémoire de début et de fin du segment désigné par *iomem_resource* à la première étape sont traduites en adresses virtuelles. [1]
4. Le contenu qui se trouve entre l'adresse virtuelle de début et de fin est lu et écrit dans un fichier ou transmit par une communication TCP/IP vers l'acquéreur.

La boucle se termine lorsque la condition *iomem_resource->sibling == NULL* est vraie. Cela signifie que l'ensemble des espaces mémoire accessibles ont été parcourus. LiME permet l'écriture en 3 formats de capture, soit brute, remplissage et LiME. Le premier est une capture brute où toutes les données en mémoire vive sont disposées de façon contigüe dans un seul fichier. Le second format remplit de 0 les espaces en mémoire dont l'adresse ne correspond pas à la mémoire vive du système (par exemple, les plages désignant l'accès aux registres des périphériques). Beaucoup plus volumineux que le format brut, il est possible d'accéder à des données dans ce format par une transposition directe d'une adresse physique vers son emplacement correspondant dans le fichier. Finalement, le troisième format insère au début du fichier un entête présenté à la figure 3.1 et tiré de la documentation de LiME, également décrite par Sylve *et al.* [113].

1. Les adresses virtuelles constituent une représentation de l'espace d'adressage exposée par le SE à un processus. Les valeurs de ces adresses virtuelles doivent être traduites à la volée par le SE et des composantes physiques dédiées du processeur vers les adresses physiques afin d'obtenir l'emplacement réel des données dans les médias de stockage de la mémoire vive. Ce mécanisme permet notamment d'offrir une plage continue de valeurs d'adressage et est prévisible aux yeux d'un processus. Cette plage et les données qui s'y trouvent peuvent tout de même être réaffectées à une autre adresse physique en mémoire ou même être changées de médium de stockage sans impact pour le programme qui les utilise. Hale Ligh *et al.* [58] détaille la mémoire virtuelle au chapitre 1 de leur ouvrage et le lecteur est invité à s'y référer pour de plus amples détails.

```
typedef struct {
        unsigned int magic;     // Always 0x4C694D45 (LiME)
        unsigned int version;       // Header version number
        unsigned long long s_addr;  // Starting address of physical RAM range
        unsigned long long e_addr;  // Ending address of physical RAM range
        unsigned char reserved[8];  // Currently all zeros
} __attribute__ ((__packed__)) lime_mem_range_header;
```

Listing 3.1 – Entête d'une capture de mémoire vive LiME.

En détail, *magic* est une valeur fixe désignant le début de l'entête, *version* désigne la version utilisée de l'outil, *s_addr* indique l'adresse physique où débute le segment qui suit sur l'appareil, *e_addr* indique sa fin et *reserved* est réservé pour des usages futurs. La taille du segment est obtenue par l'équation suivante.

$$t = e\_addr - s\_addr$$

Une quantité de t octets se trouvent après l'entête et correspondent au contenu du segment en mémoire. Si un autre segment de mémoire vive doit être ajouté dans le fichier et que l'espace d'adressage n'est pas contigu au précédent, un autre entête est ajouté à la fin du segment précédent. Le nouveau segment est à son tour ajouté après ce deuxième entête. Ce processus se poursuit jusqu'à ce qu'il n'y ait plus de segment de mémoire vive à copier.

L'utilisation de LiME nécessite que le noyau Linux permette le chargement d'un LKM. Il s'agit d'une limitation importante puisque cette fonctionnalité est désactivée par défaut sur les versions d'origines d'Android. Toutefois, il est possible de recompiler un noyau Linux d'Android à partir du code source pour l'activer pour ensuite déployer ce noyau sur l'appareil visé. Les étapes pour la configuration de LiME sont présentées par Macht [80] et, plus récemment, sur la page Web du projet *Volatility*[2]. Elles se résument de la façon suivante :

1. Télécharger le code source de LiME[3].
2. Télécharger le code source du noyau Linux utilisé sur Android pour l'appareil ciblé. Le processus pour les appareils Nexus de Google est présenté dans la documentation officielle d'Android[4]
3. Compiler le noyau. L'annexe D présente le script de compilation utilisé dans le cadre de la maîtrise. Il est à noter que le script ajoute les instructions *CONFIG_MODULES=y* et *CONFIG_MODULE_UNLOAD=y* au fichier de configuration utilisé pour la compilation. Ces options sont responsables d'ajouter le support des LKM par le noyau.
4. Compiler LiME à l'aide du noyau précédemment compilé. L'annexe E détaille cette étape.
5. Déployer le noyau modifié sur la plateforme visée. La réalisation de cette étape varie grandement en fonction de l'appareil visé.

2. Voir `https://github.com/volatilityfoundation/volatility/wiki/Android`
3. Voir `https://github.com/504ensicsLabs/LiME` .
4. Voir `https://source.android.com/source/building-kernels.html` .

6. Débloquer l'utilisateur *root* de l'appareil. Ce processus varie également d'une plateforme à l'autre.
7. Téléverser la version compilée de LiME sur l'appareil.
8. Lancer l'acquisition au moment désiré à l'aide d'une commande passée à l'appareil via l'invite de commandes accessible par l'utilisation de l'Android Debug Bridge (ADB).

L'un des avantages de LiME rapportés par Sylve *et al.* [113] est qu'il a un impact significativement moindre que l'utilisation de *dd* sur Android sur le contenu de la mémoire vive. L'étude rapporte que LiME conserve près de 99% du contenu de la mémoire intacte pendant l'acquisition alors que l'utilisation de *dd* est de l'ordre de 80%. Les chances de corrompre une donnée sensible en mémoire vive par le processus d'acquisition lui-même sont donc plus faibles. Il permet également le transfert de la capture directement par une connexion TCP plutôt que d'en faire une sauvegarde locale sur l'appareil, évitant de modifier l'état de l'appareil examiné. LiME capture l'entièreté de la mémoire visible par le SE et peut être utilisé sur des appareils physiques comme des téléphones intelligents ou des tablettes, contrairement à l'approche par MV ou émulateur décrite à la section 3.1.2. Ce dernier point constitue un avantage important dans un cas de maliciel muni d'une capacité de détection d'environnement d'analyse comme celles décrites à la section 2.1.2.

Limitations LiME copie l'ensemble de la mémoire vive pendant l'exécution du système et requiert un délai pour compléter l'opération variant selon la capacité de stockage de la mémoire vive. Ce contexte de capture peut entraîner des incohérences dues à des accès qui peuvent survenir pendant l'acquisition. Par exemple, il est possible que les données dont l'adresse est référencée dans un segment de la mémoire n'existent plus, aient été déplacées ou misent en cache sur le disque au moment où cette adresse est lue par LiME. Également, une preuve de concept sur Windows intitulée *Shadow Walker*[110] a permis de démontrer qu'un maliciel peut parvenir à dissimuler sa présence à un outil d'acquisition de la mémoire vive. Pour ce faire, un maliciel doit parvenir à détourner les méthodes système effectuant la lecture de la mémoire vive afin que celles-ci retournent du faux contenu lorsque la plage mémoire contenant le maliciel est lue. Heureusement, ce scénario n'a pas été répliqué sur Android et est très complexe à mettre en exécution.

Un autre aspect négatif de cet outil est que le processus permettant de déployer LiME n'est pas trivial. Pour parvenir à une version fonctionnelle de l'outil, il est nécessaire de bien connaître les principes de compilation d'un noyau Linux. Ce couplage fort entre LiME et la version du noyau utilisée nécessite que les processus de compilation et de déploiement de ces deux composantes soient répétés pour chaque nouvelle version du SE. Stüttgen et Cohen [111] ont tenté de contourner cette contrainte. La technique proposée par les auteurs nécessite d'abord d'identifier un LKM qui utilise les mêmes fonctions et les structures internes du SE que celles requises pour faire l'acquisition de la mémoire vive. Une fois ce LKM identifié, les auteurs réordonnent son flot d'exécution et la valeur de ses structures internes afin que ceux-ci reflètent

le nouveau comportement visé. Conséquemment, le contenu exécutable de ce LKM hôte est remplacé dynamiquement par la routine d'extraction de la mémoire vive puis est exécuté par le noyau. Bien que cette technique ait été démontrée possible sur les plateformes Linux Ubuntu et Fedora, les auteurs soulignent que l'application de cette procédure n'a pas été testée sur Android et que plus de recherche est nécessaire afin de déterminer si elle peut y être adaptée. Également, comme il est expliqué à la section 2.2.2, les LKM ne sont plus supportés dans la version officielle de Google à moins de recompiler le noyau, ce qui complique les approches décrites précédemment.

En résumé, l'outil LiME est efficace pour l'acquisition de la mémoire vive et son utilisation a un faible impact sur celle-ci. Il est également possible de l'utiliser sur des systèmes physiques, pour autant que le code source du SE de ceux-ci soit accessible. Son déploiement demeure toutefois complexe par la nécessité de recompiler et déployer un noyau Linux de même que l'outil lui-même sur l'appareil visé. Ceci étant dit, son utilité demeure justifiée dans le cadre d'un environnement d'analyse de maliciels où il est possible de préconfigurer l'appareil d'analyse avant d'y déployer un maliciels.

3.1.2 Acquisition par machine virtuelle

Présentée à la section 2.2.2, l'utilisation de MV peut permettre d'obtenir la mémoire vive du système invité. La mémoire vive est contenue dans les instantanés générés par l'engin de virtualisation ou d'émulation. Il est possible d'extraire la mémoire vive d'une MV dans plusieurs des engins de virtualisation ou d'émulation disponibles sur le marché comme QEmu [113], VirtualBox [92] et VMware [123]. L'acquisition peut se faire directement par une commande passée à la plateforme de virtualisation ou utiliser l'instantané fait d'une MV pour y extraire la mémoire vive. Cette technique ne provoque pas de corruption de mémoire due à l'acquisition elle-même puisque le système invité est complètement figé à son insu pendant la capture.

Cette approche repose sur l'acquisition de la mémoire vive par des opérations qui sont externes au SE invité. De ce fait, il est improbable qu'un maliciel puisse interférer avec le processus de capture afin de dissimuler l'activité malicieuse. Finalement, l'utilisation d'instantané préserve l'état complet du système invité (p. ex. le stockage, l'état du processeur, etc.) donnant la possibilité de combiner les résultats de l'AMV à l'analyse d'autres composantes du système virtualisé au besoin.

Limitations Pour appliquer cette approche à un appareil physique, il serait nécessaire de pouvoir capturer l'ensemble de l'état de ce dernier afin d'être capable de le répliquer dans un environnement virtuel, et ce, pendant son exécution. À ce jour, le clonage d'un appareil Android avec la version d'origine du SE déployée par le fabricant n'a pas été démontré. Cette approche ne peut donc être utilisée dans le contexte d'une gestion d'incident réel où un analyste désire inspecter un appareil suspect.

Également, la capture de la mémoire vive par l'outil de virtualisation n'est plus supportée par l'émulateur Android officiel de Google. Le retrait de cette fonctionnalité ne permet plus d'utiliser la technique d'extraction décrite par Sylve *et al.* [113] pour Android. En effet, lorsque la commande d'acquisition est lancée sur l'émulateur, un message d'erreur indique que cette fonctionnalité est absente tel que présenté au listage 3.2. À la ligne 1, la connexion pour le contrôle de l'émulateur est établie via l'utilitaire *netcat* depuis le poste hôte. La commande *qemu monitor stdio* est envoyée à l'émulateur à la ligne 4. L'émulateur retourne un message d'erreur à la ligne 5 signifiant que le support de cette fonctionnalité a été retiré.

```
1 pc@current:~\$ nc 127.0.0.1 5554
2 Android Console: type 'help' for a list of commands
3 OK
4 qemu monitor stdio
5 KO: QEMU support no longer available
```

Listing 3.2 – Support de la capture de la mémoire vive retirée sur l'émulateur Android.

Pour cette raison, il est nécessaire d'utiliser des outils de virtualisation comme VirtualBox ou VMware pour obtenir une capture de cette façon. Or, ces systèmes permettent uniquement de virtualiser l'architecture de microprocesseur Intel x86/x64. Pour cette raison, ces outils ne sont compatibles qu'avec une mouture d'Android non officielle appropriée à ce type d'architecture appelée Android-x86 [64]. Or, cette version comporte des différences avec la version d'origine d'Android, ne virtualise pas l'activité de capteurs comme l'accéléromètre et n'est pas mise à jour au même rythme que la version officielle d'Android puisque ces mises à jour doivent être adaptées avant d'être applicables.

Finalement, tel qu'il est présenté à la section 2.1, il est possible pour un maliciel de détecter s'il est exécuté dans un environnement de virtualisation et d'inhiber un comportement malicieux si tel est le cas. Ainsi, les données recueillies par un environnement de virtualisation peuvent différer de ce qui serait collecté sur un appareil physique si un maliciel inhibe ses actions pour éviter la détection. Pour cette raison, la validité des résultats peut être en doute s'ils reposent sur des données recueillies par une MV.

3.2 Analyse

Les données acquises prennent la forme de valeurs binaires brutes nécessitant un traitement pour être utilisables à l'analyse. Comme décrit dans la méta-analyse de Garcia [41], il est possible de faire une recherche de chaînes de caractères identifiables contenues dans un fichier de capture afin d'extraire des valeurs indicatrices d'une activité malicieuse. Cette méthode a l'avantage d'être très rapide à exécuter puisque des outils optimisés pour ce type de recherche existent et sont communs. Par exemple, *grep* est un outil inclus dans le noyau Linux permettant d'identifier les chaînes de caractères contenues dans un fichier qui coïncident à une expression régulière soumise par l'utilisateur. Cette approche est limitée si la chaîne recherchée change

(p. ex. changement de l'adresse d'un serveur de commande et contrôle du maliciel), si elle est segmentée (p. ex. plusieurs segments non contigus plutôt qu'un tableau continu en mémoire) ou si elle est inconnue (p. ex. une clef de chiffrement binaire).

Une autre technique procède plutôt en parcourant l'ensemble du fichier de capture et en recensant les chaînes d'octets répondant à un ensemble de critères indiquant la présence d'une chaîne de caractères. Par exemple, l'outil *strings* identifie l'emplacement dans le fichier où se trouvent toutes séquences d'au moins 4 octets ou plus pouvant représenter un caractère et se terminant par un octet non imprimable (p. ex. l'octet nul '0x00'). Cette approche relève beaucoup de faux positifs devant être triés. Ce travail est généralement réalisé par un expert pouvant être assisté par un ou des algorithmes permettant de trier la pertinence des résultats (p. ex. Beebe et Clark [13]).

Les techniques reposant sur les chaînes de caractères ne sont pas en mesure de relever des variables d'un type autre que des chaînes de caractères comme la valeur d'un nombre à virgule flottante de type *float* ou une structure interne du noyau. Également, le contenu récupéré est décontextualisé, c'est-à-dire que les informations liées à son contexte d'exécution où est utilisé le texte ne sont pas récupérées par ces techniques. Cette contrainte peut entraîner des faux positifs si la chaîne de caractères visée apparaît dans un contexte légitime et sans présence malicieuse dans le système. Par exemple, un navigateur Web pourrait avoir dans son espace mémoire toutes les chaînes de caractères indiquant la présence d'un comportement malicieux si le contenu de la page Web affichée à l'écran décrit les chaînes de caractères du maliciel en question. Cet exemple représente le cas d'une application légitime qui serait faussement caractérisée comme un maliciel. À l'inverse, si une variante du maliciel utilise une adresse de serveur différente, celle-ci pourrait passer inaperçue.

Puisque le contexte d'exécution importe pour effectuer une AMV plus complète, une représentation de la topologie de la mémoire vive est préférable pour refléter l'état internes du SE au moment de la capture. Cela permet de récupérer des structures interne du SE comme l'inventaire des processus en exécution, la liste des fichiers utilisés de même que leur contenu tel que présenté par Case *et al.* [20]. Cette reconstruction de la topologie de la mémoire vive est une stratégie répandue dans les travaux portant sur l'AMV, tous SE confondus (p. ex. [29, 41, 58, 94, 121, 125]).

3.2.1 Volatility

Sur Android, l'analyse de la mémoire vive a été principalement étudiée à l'aide d'un logiciel libre appelé *Volatility*[5] programmé en *Python* [58, 80, 113]. Cette section vise à présenter le fonctionnement global de *Volatility* et son utilisation pour des systèmes Linux et, par extension, Android. Le lecteur est référé à l'ouvrage de Hale Ligh *et al.* [58] pour une compréhension

5. `https://github.com/volatilityfoundation/volatility`

approfondie de l'outil et de ses capacités, notamment en ce qui a trait à Mac OS X et Windows.

Description *Volatility* permet de reconstruire la représentation des structures spécifiques au SE présent sur le système d'où provient la capture depuis la mémoire vive. Pour se faire, *Volatility* charge un profil associé à une capture de mémoire vive lui permettant d'interpréter ce fichier en tenant compte du contexte de l'acquisition. Un profil comprend les éléments présentés au tableau 3.1 tiré de Hale Ligh *et al.* [58].

Composantes	Description
Métadonnées	Nom du SE, version, numéro du *build*.
Informations sur les appels systèmes	Indexes et noms.
Constantes du SE	Les variables à position fixe en mémoire du SE.
Type des primitives	Types des variables primitives (habituellement en *C*), incluant le nombre de bits des *integer*, *float*, etc.
System map	Structure contenant l'ensemble des adresses pour les variables et méthodes globales qui doivent être à un emplacement fixe en mémoire pour que le SE fonctionne correctement. Ce fichier est nécessaire pour les systèmes Macs et Linux.

TABLEAU 3.1 – Éléments d'un profil.

Un profil est constitué de 3 composantes requises pour reconstruire les structures en mémoire vive, soit les *VTypes*, les *Overlays* et les objets de *Volatility*.

Les *VTypes* sont des classes d'objets spécifiques à, et manipulables par *Volatility* qui permettent de transposer des valeurs binaires en mémoire vive vers leur structure C correspondante dans un format compatible avec *Python*. Une comparaison à la figure 3.1 présente une structure *C* et le *VType* équivalent en *Python*.

```
1 struct process {
2     int pid;
3     int parent_pid;
4     char name[10];
5     char * cmd;
6     void * ptv;
7 };
```

```
1 'process':[26, {
2     'pid':[0,['int']],
3     'parent_pid':[4,['int']],
4     'name':[8,['array',10,['char']]],
5     'cmd':[18,['pointer',['char']]],
6     'ptv':[22,['pointer',['void']]]
7 }]
```

FIGURE 3.1 – Structure *C* (gauche) et équivalent en *VType* (droite).

Le nom de la structure est indiqué à la ligne 1 du *VType* suivi du nombre d'octets qu'elle nécessite en mémoire. Chaque ligne subséquente présente le nom de la variable, l'*offset* en octets où elle se trouve à partir du début de la structure et finalement son type. Si le type est un tableau comme à la ligne 4, l'identifiant *array*, la taille du tableau et le type des variables qu'il contient sont également indiqués. Ces *VTypes* sont essentiels à la reconstruction des structures contenues en mémoire vive et chaque version de SE nécessite un ensemble de VTypes spécifique. Plus particulièrement sur Linux et Android, ils doivent être générés lors du processus de compilation du noyau à l'aide de *dwarfdump*, un outil de développement permettant d'extraire

automatiquement ces informations pour l'ensemble des structures présentes dans le noyau à partir du code source de celui-ci.

Il est possible d'ajouter ou de modifier les définitions des *VTypes* en appliquant des *Overlays* sur celles-ci. Les *Overlays* sont des définitions suivant le format d'un *VType* permettant de redéfinir certaines structures afin qu'elles correspondent au contexte d'analyse. Par exemple, un pointeur de type *void* pourrait correspondre à une structure connue de l'analyste et applicable pour une version modifiée de Linux. Ainsi, lorsque ce contexte d'analyse sera rencontré, l'*Overlay* correspondant peut être appliqué pour que *Volatility* reconstruise cette structure référencée et extraie son contenu de la capture de la mémoire vive.

En combinant l'utilisation des *VTypes* et des *Overlays* correspondants, il est possible de recréer des objets *Volatility*. Ces objets sont des structures reconstruites à partir de valeurs binaires contenues dans le fichier de capture de mémoire. Cette reconstruction s'effectue en récupérant le contenu situé à une adresse mémoire spécifique (p. ex. l'adresse d'une constante) ou par la reconnaissance d'un patron de valeurs appartenant à une structure du SE connue au moment de l'analyse (p. ex. l'entête d'une structure de données). Concrètement, les objets *Volatility* sont des classes *Python* contenant les variables définies par les *VTypes* et leur valeur. Ces classes peuvent être étendues pour offrir des fonctions permettant de manipuler leur contenu afin de raffiner l'extraction de marqueurs pertinents à l'analyse.

Dans le cas d'un système Linux, un profil est une archive Zip contenant l'information sur les structures de données du noyau Linux et ses *debug symbols* [124]. Contrairement à Windows pour lequel l'outil possède un répertoire de profils préconstruits, un profil Linux doit être construit par l'analyste spécifiquement pour chaque version du SE analysée.

La capture de mémoire vive peut être chargée dans l'outil une fois le bon profil *Volatility* choisi. Au moment du chargement, une succession de traductions d'espaces d'adressage est appliquée. Leur but est d'abstraire les accès au fichier de capture pour permettre à l'analyste de naviguer dans la capture de la mémoire vive comme s'il était sur le système acquis lui-même. Notamment, il permet d'explorer directement des emplacements en mémoire en utilisant des valeurs d'adresses telles qu'originalement vues par le SE lors de son exécution et non en fonction d'*offset* dans le fichier de capture. Par exemple, un premier filtre effectue la traduction d'adressage de la mémoire virtuelle vers l'espace d'adressage en mémoire physique en fonction de l'architecture du processeur, un second convertit cette valeur en un emplacement dans le fichier en fonction du type de fichier de la capture (p. ex. pour un format LiME, il faut tenir compte des entêtes du fichier) et finalement un dernier filtre effectue la récupération dans le fichier en fonction de l'*offset* calculé par les étapes précédentes. Il est possible de récupérer une information en n'utilisant qu'une partie de la séquence d'adressage, par exemple si l'analyste connaît directement l'adresse physique en mémoire qui l'intéresse.

Une fois le profil et la capture chargés, *Volatility* est en mesure d'appliquer des analyses sous la

Greffons	*Description*
linux_apihooks	Vérifie s'il y a détournements d'appels à l'API du noyau.
linux_arp	Affiche la table de routage Address Routing Protocol (ARP).
linux_banner	Affiche des informations sur le noyau du système.
linux_bash	Récupère l'historique des commandes *bash*.
linux_bash_env	Récupère les variables d'environnement dynamiques d'un processus *bash*.
linux_bash_hash	Récupère la table de hachage *bash* de la mémoire de ce processus.
linux_check_afinfo	Vérifie l'adresse des pointeurs des fonctions d'opération des protocoles réseau.
linux_check_creds	Valide si des processus partagent la même structure de privilèges.
linux_check_evt_arm	Vérifie la table d'exception afin de valider si un pointeur d'appel système a été modifié.
linux_check_fop	Valide si les structures d'opération ont été modifiées par un exploit de type *rootkit*.
linux_check_idt	Valide si l'IDT Interrupt Descriptor Table (IDT) a été modifiée
linux_check_inline_kernel	Recherche des altérations aux appels aux méthodes du noyau.
linux_check_modules	Compare la liste des modules avec les entrées dans *sysfs info*, si disponible.
linux_check_syscall	Valide si la table d'appels système a été modifiée.
linux_check_syscall_arm	Valide si la table d'appels système a été modifiée (pour processeur ARM).
linux_check_tty	Vérifie si des périphériques de type tty ont été détournés.
linux_dentry_cache	Extrait les fichiers présents dans la cache *dentry*.
linux_dmesg	Récupère le tampon de *dmesg*.
linux_dump_map	Écrit les champs de mémoire spécifiés sur stockage physique.
linux_dynamic_env	Récupère les variables d'environnement dynamiques d'un processus.
linux_elfs	Identifie et récupère les fichiers binaires ELF chargés dans l'espace mémoire des processus.
linux_enumerate_files	Répertorie l'ensemble des fichiers référencés par la cache du système de fichiers.
linux_find_file	Liste et récupère les fichiers en mémoire vive.

TABLEAU 3.2 – Partie 1 - Documentation des greffons pour Linux de *Volatility*.

forme de modules *Python* appelés greffons (ou *plug-ins*) permettant d'extraire les informations désirées sur le SE et ses différentes composantes au moment de la capture.

Android L'AMV sur Android à l'aide de *Volatility* a été principalement étudiée à l'aide de deux approches. La première consiste à utiliser les greffons visant Linux pour extraire l'information qui est commune à tous les SE ayant un noyau Linux. L'ensemble de ces greffons permet de sonder le noyau pour y extraire des traces d'activités anormales. Par exemple, il est possible de récupérer des fichiers entiers ou partiels manipulés en mémoire vive, d'obtenir la liste des processus en exécution au moment de la capture, d'obtenir l'état des connexions réseaux ainsi que les paquets en attentes de traitement par processus ou de vérifier s'il y a eu un détournement d'appels systèmes (ou *system call hooking*). L'ensemble de ces fonctionnalités est détaillé dans la documentation officielle de *Volatility* [124] de même que dans l'ouvrage de Hale Ligh *et al.* [58]. Les tableaux 3.2, 3.3 et 3.4 présentent la liste de greffons Linux ainsi que leur documentation officielle rattachée (traduction libre).

Greffons	*Description*
linux_getcwd	Liste le chemin d'accès courant pour chacun des processus.
linux_hidden_modules	Inspecte la mémoire vive pour y identifier des modules cachés du noyau.
linux_ifconfig	Répertorie les interfaces réseaux actives.
linux_info_regs	Méthode analogue à *info registers* dans *gdb*.
linux_iomem	Produit une sortie similaire à */proc/iomem* sur Linux.
linux_kernel_opened_files	Liste les fichiers chargés dans l'espace mémoire du noyau.
linux_keyboard_notifiers	Reconstruit la chaîne d'appels des notifications du clavier.
linux_ldrmodules	Compare la sortie de *proc maps* avec la liste des bibliothèques logicielles de l'entité *libdl*.
linux_library_list	Répertorie l'ensemble des bibliothèques logicielles chargées en mémoire d'un processus.
linux_librarydump	Extrait les bibliothèques logicielles partagées dans la mémoire d'un processus.
linux_list_raw	Répertorie les applications ayant des *sockets* en mode *promiscuité*.
linux_lsmod	Inventorie les modules du noyau chargés.
linux_lsof	Liste les descripteurs de fichier et leur chemin d'accès.
linux_malfind	Tente d'identifier la présence d'activités malicieuses par l'analyse de la topologie de la mémoire des processus.
linux_memmap	Extrait la topologie de la mémoire vive pour les tâches Linux.
linux_moddump	Extrait les modules du noyau chargés au moment de la capture.
linux_mount	Liste les *fs/devices* montés
linux_mount_cache	Liste les *fs/devices* montés depuis la *kmem_cache*.
linux_netfilter	Liste les détournements présents dans *Netfilter*.
linux_netscan	Identifie les structures de connexions réseaux présentes dans la mémoire vive.
linux_netstat	Identifie les ports ouverts et connexions établies.
linux_pidhashtable	Énumère les processus via la table de hachage des identifiant de processus (Process Id, PID).
linux_pkt_queues	Écrit l'ensemble des queues par processus de paquets présents en mémoire vive sur le disque.

TABLEAU 3.3 – Partie 2 - Documentation des greffons pour Linux de *Volatility*.

Afin d'améliorer l'analyse sur les plateformes Android, la seconde méthode proposée par Macht [80] et 504ensics Labs [1] procède en extrayant la représentation interne des variables et des structures des applications Android par la reconstitution de la MVD depuis une capture de la mémoire vive. En identifiant l'emplacement en mémoire de la MVD relativement au début du processus système *zygote*, il est possible de retrouver l'instance de la MVD dans l'ensemble des autres applications Android en appliquant ce même *offset* à ces applications. Cela est dû au fait qu'une application Android est d'abord initialisée par un appel système *fork*[6] sur *zygote* qui contient toutes les composantes de bases requises pour son lancement, dont la MVD. L'application charge ensuite les composantes qui lui sont uniques (p. ex. classes spécifiques, constantes, bibliothèques logicielles, etc.). En récupérant le contenu de la MVD, il est possible de récupérer l'ensemble des structures Java contenues dans l'application et d'avoir une pleine visibilité sur l'ensemble des structures internes et leur valeur pour une application Android.

6. *fork* permet de générer un nouveau processus identique à un autre et qui partage les mêmes références en mémoire vive que celui d'origine. Les références en mémoire sont modifiées vers un espace mémoire différent dès que l'un ou l'autre des processus effectue une écriture, évitant la corruption de la donnée pour l'autre processus.

Greffons	*Description*
linux_plthook	Recherche dans la table des liens des procédures (Procedure Linkage Table, PLT) pour des appels suspects.
linux_proc_maps	Récupère la topologie de la mémoire des processus.
linux_proc_maps_rb	Reconstitue la topologie de la mémoire des processus pour Linux via l'utilisation d'arborescence rouge-noir.
linux_procdump	Extrait l'exécutable d'un processus et le sauvegarde sur le disque.
linux_process_hollow	Recherche des indicateurs de processus évidés.
linux_psaux	Reproduit la sortie de la commande *ps aux* sur Linux qui permet d'obtenir l'ensemble des processus en exécution, leur chemin d'accès et l'heure de leur démarrage.
linux_psenv	Liste les processus avec les variables d'environnement statiques qui y sont rattachées.
linux_pslist	Reconstruit la liste des tâches Linux actives en parcourant la liste chaînée *task_struct->task list*.
linux_pslist_cache	Reconstruit les tâches Linux présentes dans la structure interne *kmem_cache*.
linux_pstree	Identifie les relations parents/enfants entre les processus en exécution.
linux_psxview	Tente de trouver des processus cachés en mémoire vive en croisant les résultats obtenus par les différentes façons de lister les processus présents en mémoire.
linux_recover_filesystem	Récupère l'ensemble du système de fichiers en cache depuis la mémoire vive, s'il y en a un.
linux_route_cache	Récupère la cache de la table de routage depuis la mémoire vive.
linux_sk_buff_cache	Récupère les paquets réseaux depuis la structure interne *sk_buff kmem_cache*.
linux_slabinfo	Duplique la structure */proc/slabinfo* sur un système en exécution.
linux_strings	Récupère les chaînes de caractères à partir d'une valeur d'adresse physique convertie en adresse virtuelle.
linux_threads	Liste les fils d'exécutions des processus.
linux_tmpfs	Récupère les systèmes de fichiers *tmpfs* en mémoire vive.
linux_truecrypt_passphrase	Tente d'extraire la phrase secrète en cache de l'utilitaire de chiffrement Truecrypt.
linux_vma_cache	Récupère les allocations de mémoire virtuelle à l'aide de la cache de *vm_area_struct*.

TABLEAU 3.4 – Partie 3 - Documentation des greffons pour Linux de *Volatility*.

Les variables, les objets et leur classe d'appartenance sont interprétables de cette façon. Nasim *et al.* [88] ont également utilisé cette représentation de la MVD en mémoire vive. Leur approche a permis d'identifier les classes Java qui s'y trouvent et de les comparer avec celles déclarées dans le code décompilé de l'application. Les classes présentent en mémoire et qui n'ont pas d'équivalence dans l'APK sont identifiées comme ayant été chargées dynamiquement et sont une source potentielle d'injection de comportements échappant à l'analyse statique. L'abolition de la MVD dans les versions actuelles d'Android rend ces techniques obsolètes. À ce jour, peu de travaux se sont intéressés à l'étude de l'AMV sur les versions d'Android reposant sur ART.

Résumé

Ce chapitre résume les principaux travaux réalisés à ce jour sur Android dans le domaine de l'AMV. Bien que la pertinence de cette approche ait été démontrée pour d'autres SE

appartenant à l'univers PC et Mac, les études menées sur Android demeurent limitées. Il est possible de dégager deux courants principaux de l'état de l'art qui se positionnent le long d'un continuum. À un pôle se trouvent les méthodes génériques qui sont résistantes aux mises à jour majeures comme c'est le cas pour l'approche par recherche de chaîne de caractères. À l'opposé se trouvent les approches spécifiques à la plateforme qui permettent d'extraire des résultats très précis et spécifiques à la plateforme, mais dont le bon fonctionnement est fortement couplé à la structure interne du SE et qui doivent refléter les changements à celle-ci. Les techniques reposant sur la MVD comme celles impliquées dans DalvikInspector de 504ensics Labs [1] et dans les modules *Volatility* de Macht [80] représentent bien cette réalité. Afin de combler cet écart, il importe de proposer une technique capable d'extraire des éléments spécifiques aux structures internes du SE à partir d'une capture de mémoire vive. Il incombe que la technique demeure suffisamment générique pour être valide avec peu ou pas d'adaptations entre les versions du SE dans un contexte d'analyse de maliciels. Le chapitre suivant propose une méthode répondant à ces critères et l'applique à un maliciel expérimental afin d'en démontrer l'efficacité.