

# **VÉRIFICATION PERMANENTE DE L'INTÉGRITÉ ET DE L'AUTHENTICITÉ DE FICHIERS D'UNE INFRASTRUCTURE DÉFINIE PAR DU CODE**

Thèse de Bachelor présentée par

**Sergio Guarino**

pour l'obtention du titre Bachelor of Science HES-SO en

**Informatique et systèmes de communication avec orientation en  
Sécurité informatique**

**Septembre 2022**

Professeur HES responsable

**Mickaël HOERDT**



## TABLE DES MATIÈRES

<b>Remerciements</b>	<b>v</b>
<b>Énoncé du sujet</b>	<b>vii</b>
<b>Résumé</b>	<b>viii</b>
<b>Liste des acronymes</b>	<b>ix</b>
<b>Liste des illustrations</b>	<b>x</b>
<b>Liste des tableaux</b>	<b>xii</b>
<b>Liste des annexes</b>	<b>xii</b>
<b>Introduction</b>	<b>1</b>
<b>1. Chapitre 1 : État de l'art</b>	<b>4</b>
1.1. Intégrité	4
1.2. Outils de vérification d'intégrité de fichiers	6
1.3. Outils avancés de vérification d'intégrité de fichiers	9
a) dm-verity	9
b) IMA	12
c) FS-Verity	13
1.4. Authenticité	19
a) Algorithmes de signature	19
b) Certificats	21
c) Exemples	23
1.5. Intégrité et authenticité des packages	27
a) apt	28
b) dnf	32
c) Sécurité des gestionnaires de paquets	34
1.6. Infrastructure as Code	35
1.7. Déploiement automatisé d'une infrastructure	36
a) Ansible	36
b) Puppet	37
c) Chef	37
d) SaltStack	37
e) Comparaison	38
<b>2. Chapitre 2 : Conception de l'architecture</b>	<b>41</b>
2.1. Théorie de fonctionnement	42
2.2. Modèle C4	44
a) Niveau 1 : Contexte	45
b) Niveau 2 : Conteneurs	46
c) Niveau 3 : Composants	47
2.3. Diagramme de déploiement	50
2.4. Diagramme de séquence	51
<b>3. Chapitre 3 : Implémentation</b>	<b>54</b>

<b>3.1.</b>	<b>Outils fs-verity .....</b>	<b>54</b>
a)	Modification programme de signature.....	56
<b>3.2.</b>	<b>Automatisation des opérations.....</b>	<b>60</b>
a)	Fonctionnement d'Ansible .....	60
b)	Développement d'un module personnalisé.....	63
c)	Idempotence.....	67
<b>3.3.</b>	<b>Vérification permanente de l'intégrité des fichiers .....</b>	<b>69</b>
a)	Script de vérification.....	69
b)	Service de monitoring.....	69
c)	Fichiers à protéger .....	71
<b>3.4.</b>	<b>Cas d'utilisation .....</b>	<b>72</b>
a)	Prérequis.....	72
b)	Déploiement du playbook .....	74
c)	Modifications manuelles .....	75
<b>3.5.</b>	<b>Considérations de sécurité .....</b>	<b>77</b>
	<b>Conclusion .....</b>	<b>80</b>
	<b>Annexes .....</b>	<b>82</b>
	<b>Références documentaires .....</b>	<b>86</b>

## REMERCIEMENTS

On a souvent l'habitude de se référer à nos accomplissements en première personne : « J'ai fait ... », « J'ai réalisé ... », mais en vérité, la plupart de ce que l'on fait ne serait pas possible sans toutes les autres personnes et événements qui ont contribué à la réussite de quelqu'un. Toute réalisation, qu'elle pourrait sembler petite ou grande, est le résultat du travail collectif de notre société. Même un petit objet comme un crayon demande les efforts d'un grand nombre de personnes : qui extrait les matières premières, qui les traite, qui dessine le crayon, qui le fabrique, qui le vend.

De même, bien que ce document porte mon nom, il peut également être considéré comme un travail collectif, puisque sans l'aide ou la présence d'un certain nombre de personnes, il n'aurait certainement pas été possible. Ainsi, j'y tiens à remercier de façon particulière les personnes suivantes.

Mon professeur encadrant, Mickaël Hoerd, qui m'a suivi pendant ces mois et m'a prodigué des précieux conseils tout le long de la formation. Je vous remercie d'abord d'avoir accepté d'être mon encadrant et non seulement pour tout ce que vous m'avez appris, mais également pour votre patience, votre disponibilité et votre gentillesse.

Tout le corps enseignant de l'Hepia, ainsi que les assistants, avec qui j'ai eu le plaisir de rentrer en contact et qui ont fait tout ce qui était en leur pouvoir pour me transmettre leurs connaissances et faire de moi un Ingénieur.

Mon employeur, Xelios Suisse, mes collègues et en particulier Monsieur Raymond Dafflon qui m'a permis d'effectuer cette formation à temps partiel et a toujours fait le possible pour me soutenir et me permettre de suivre mes études.

Mes camarades d'école, qui ont rendu plus amusant et léger le temps passé en classe et dehors. J'espère qu'on restera en contact dans les années qui viendront et je vous souhaite plein de succès dans votre carrière professionnelle.

Ma famille, qui m'a toujours soutenu et était toujours là pour moi quand j'en avais besoin. J'espère vous serez fières de moi.

Tous mes amis les plus chers : Laura, Giacomo, Massimiliano, Olivier, Anne-Gaëlle, Marco, Angelo et Mariella. Vous êtes les meilleurs amis que quelqu'un pourrait espérer d'avoir à son côté. Je suis honoré d'être dans vos vies et je vous remercie infiniment pour tout ce que vous

avez fait et faites tous les jours pour moi. Vous êtes une grande source d'inspiration pour moi et pour cela je vous admire. Ça me rassure et me rend heureux de savoir d'avoir des personnes comme vous sur lesquelles je pourrai toujours compter.

Enfin, j'aimerais remercier toutes les autres personnes qui ont contribué d'une manière ou d'une autre à rendre tout cela possible.

## ÉNONCÉ DU SUJET

**Descriptif :** De plus en plus de services et d'infrastructure sont aujourd'hui définis par du code (IaC). Ceux-ci sont déployés et maintenus dans un processus d'intégration continue similaire à celui qu'on retrouve en développement logiciel, à la grande différence que le résultat final n'est pas un logiciel, mais une infrastructure et ses services, re-déployés en permanence selon les évolutions /innovations des services.

Une fois déployée, il est difficile de garantir qu'une infrastructure reste sécurisée puisqu'elle peut à tout moment faire l'objet d'une attaque informatique sur les services qu'elle expose en particulier si ceux-ci sont publics. Etant donné que l'infrastructure évolue en permanence, un processus de vérification statique de l'intégrité d'une infrastructure n'est pas envisageable mais doit être compatible avec un processus de déploiement continu, qui implique des changements réguliers des fichiers qui composent les services configurés sur les serveurs.

Dans ce travail, nous proposons de concevoir et d'implémenter une architecture de contrôle continu de l'intégrité et de l'authenticité des fichiers déployés sur une infrastructure informatique hébergeant des services sur des serveurs génériques de type x86-64 exécutant le système d'exploitation Debian GNU/Linux. Cette architecture et implémentation devra prendre en compte les contraintes suivantes :

- Plusieurs centaines de serveurs et plusieurs milliers de services.
- Services exposés sur l'Internet public ne nécessitant pas de persistance de données.
- Facilité d'utilisation avec les outils usuels de déploiement d'IaC.

### Travail demandé :

1. Etat de l'art
  - Outils de déploiement d'infrastructure as Code (Ansible/Puppet/...)
  - Vérification d'intégrité et d'authenticité de fichiers et de volumes sous Debian GNU/Linux
  - Chaînes de confiance et distribution des packages sous GNU/Linux
2. Conception de l'architecture
  - Diagrammes de définition de l'architecture (Séquence/Déploiement/Classe/Composants).
  - Choix des composants existants.
  - Analyse des risques de sécurité de la proposition.
3. Implémentation, tests et démonstration
  - Choix du langage libre
  - Réalisation d'une maquette dans le simulateur GNS3

Candidat :

**GUARINO SERGIO**

Filière d'études : ISC

Professeur responsable :

**HOERDT MICKAËL**

En collaboration avec :

Travail de bachelor soumis à une convention de stage en entreprise : **non**

Travail de bachelor soumis à un contrat de confidentialité : **non**

## RÉSUMÉ

La diffusion des technologies de virtualisation a permis le déploiement simple et rapide d'un très grand nombre de machines virtuelles. Il n'est pas rare de trouver des infrastructures informatiques composées des centaines, voire des milliers de serveurs. Leur gestion et leur configuration doit être gérée de façon automatisée : il serait impensable de modifier chaque machine individuellement. C'est l'objectif de l'Infrastructure as Code : définir une infrastructure informatique par un ensemble de fichiers de configuration de façon à pouvoir gérer un nombre indéfini de machines, de manière simple, au travers d'outils d'automatisation. Mais, avec l'augmentation de la taille d'une infrastructure, garantir la sécurité de chaque élément devient également une tâche complexe, surtout quand ces machines sont exposées à internet pour pouvoir offrir des services (par ex. héberger un site web). Le but de ce travail est de proposer une solution de sécurité qui permette de garantir un contrôle continu de l'intégrité et de l'authenticité de fichiers présents sur les machines qui composent l'infrastructure. Cette solution doit également être compatible avec les outils d'automatisation de l'infrastructure. Ainsi, après une phase d'étude sur l'état de l'art des solutions existantes, on a choisi deux produits : un de vérification d'intégrité et authenticité, l'autre d'automatisation d'infrastructure. Cela nous a permis de concevoir une architecture qui combine ces deux produits et à partir de là, de développer une solution de sécurité complète. Le résultat est un ensemble d'outils, simples d'utilisation, qui travaillent en symbiose et qui fournissent une méthode pour la mise en place du contrôle continu de l'intégrité et de l'authenticité des fichiers d'une infrastructure informatique définie par du code.



Candidat :

**GUARINO SERGIO**

Filière d'études : ISC

Professeur responsable :

**HOERDT MICKAËL**

**En collaboration avec :**

Travail de bachelor soumis à une convention de stage en entreprise : NON

Travail soumis à un contrat de confidentialité : NON



## LISTE DES ACRONYMES

API	Application Programming Interface
APT	Advanced Package Tool
CA	Certification Authority
CLI	Command Line Interface
CPU	Central Processing Unit
DNF	Dandified YUM
DSL	Domain Specific Language
GNS3	Graphical Network Simulator 3
IaC	Infrastructure as Code
ICMP	Internet Control Message Protocol
IMA	Integrity Measurement Architecture
IP	Internet Protocol
RPM	RPM Package Manager
SSH	Secure Shell Protocol
TPM	Trusted Platform Module
VM	Virtual Machine

## LISTE DES ILLUSTRATIONS

Illustration 1: résultats test vitesse de hachage .....	6
Illustration 2: Arbre de Merkle dm-verity. ....	10
Illustration 3: Schéma de la chaine de confiance du processus de démarrage Android.....	11
Illustration 4: Arbre de Merkle fs-verity. ....	15
Illustration 5: Schéma de l'algorithme de chiffrement Diffie-Hellman. ....	20
Illustration 6: Représentation de l'utilisation des clés privées et publiques : à gauche pour le chiffrement, à droite pour la signature .....	21
Illustration 7: Certificats du site hepia.ch .....	22
Illustration 8: Génération d'une clé privée avec openssl .....	23
Illustration 9: Affichage d'une clé privée avec cat.....	23
Illustration 10: Affichage d'une clé privée avec openssl .....	24
Illustration 11: Génération d'une clé publique avec openssl .....	24
Illustration 12: Affichage d'une clé publique avec cat.....	25
Illustration 13: Affichage d'une clé publique avec openssl .....	25
Illustration 14: Génération d'un certificat avec openssl .....	26
Illustration 15: Affichage d'un certificat avec openssl .....	26
Illustration 16: Hachage d'un fichier avec openssl .....	27
Illustration 17: Affichage de la signature d'un fichier .....	27
Illustration 18: Vérification de la signature d'un fichier.....	27
Illustration 19: Exemple de fichier sources.list d'apt .....	29
Illustration 20: Exemple d'arborescence d'un dépôt Debian.....	29
Illustration 21: Exemple de fichier InRelease.....	30
Illustration 22: Exemple de commande lsattr quand fs-verity est activé .....	44
Illustration 23: Exemple de commande lsattr quand il manque le certificat dans le keyring .....	44
Illustration 24: Niveau 1 du Modèle C4.....	45
Illustration 25: Niveau 2 du Modèle C4.....	46
Illustration 26: Niveau 3 du Modèle C4 – fs-verity .....	48
Illustration 27: Niveau 3 du Modèle C4 – Ansible .....	48
Illustration 28: Diagramme de déploiement.....	50
Illustration 29: Diagramme de séquence.....	52
Illustration 30: Page d'aide des outils fs-verity .....	54
Illustration 31: Retours de la commande fsverity digest.....	55
Illustration 32: Partie initiale du digest pour la signature .....	55
Illustration 33: Essai d'activation de fs-verity .....	55
Illustration 34: Essai de vérification de fs-verity .....	56
Illustration 35: Code d'origine du programme cmd_script.c .....	57
Illustration 36: Nouvelles variables du programme cmd_script.c modifié.....	57
Illustration 37: Nouvelles fonctions du programme cmd_script.c modifié .....	58
Illustration 38: Définition de la fonction xzalloc du programme utils.c .....	59
Illustration 39: Définition de la structure du digest dans la librairie fsverity.h .....	59
Illustration 40: Exemple de commande ping Ansible .....	60
Illustration 41: Exemple de playbook Ansible .....	61
Illustration 42: Exemple d'exécution d'un playbook Ansible .....	62
Illustration 43: Exemple de message d'erreur en cas d'erreur de syntaxe YAML.....	63

Illustration 44: Exemple minimale de module Ansible .....	64
Illustration 45: Exemple d'arborescence minimale pour fonctionnement Ansible avec un module personnalisé .....	66
Illustration 46: Retour de la commande ansible --version .....	66
Illustration 47: Exemple de playbook Ansible avec module personnalisé .....	66
Illustration 48: Exemple d'exécution de playbook Ansible avec module personnalisé .....	67
Illustration 49: Exemple de manque d'idempotence dans l'exécution d'un playbook Ansible ....	68
Illustration 50: Fichier de configuration d'un service .....	70
Illustration 51: Exemple de fichier de logs généré par le service de vérification de l'intégrité....	70
Illustration 52: Commande d'activation de la signature obligatoire fs-verity.....	73
Illustration 53: Ajout d'un certificat dans keyring fs-verity .....	73
Illustration 54: Affichage du contenu du keyring fs-verity .....	74
Illustration 55: Arborescence du répertoire d'exécution du playbook Ansible .....	74
Illustration 56: Résultat de l'exécution du playbook Ansible .....	75
Illustration 57: Script de désactivation de fs-verity .....	76
Illustration 58: Copie d'un fichier fs-verity .....	76
Illustration 59: État d'un fichier avant exécution script .....	77
Illustration 60: État d'un fichier après exécution script .....	77
Illustration 61: Affichage du contenu du keyring fs-verity .....	78
Illustration 62: Suppression d'un certificat du keyring fs-verity .....	78
Illustration 63: Essai d'accès à un fichier signé avec fs-verity après suppression du certificat du keyring.....	78
Illustration 64: Tests sur la gestion des signatures fs-verity .....	79

## Références des URL

URL01	<a href="https://source.android.com/security/verifiedboot/dm-verity">https://source.android.com/security/verifiedboot/dm-verity</a>
URL02	<a href="https://events19.linuxfoundation.org/wp-content/uploads/2017/11/fs-verify_Mike-Halcrow_Eric-Biggers.pdf">https://events19.linuxfoundation.org/wp-content/uploads/2017/11/fs-verify_Mike-Halcrow_Eric-Biggers.pdf</a>
URL03	<a href="https://en.wikipedia.org/wiki/File:Public_key_encryption.svg">https://en.wikipedia.org/wiki/File:Public_key_encryption.svg</a>
URL04	<a href="https://en.wikipedia.org/wiki/File:Private_key_signing.svg">https://en.wikipedia.org/wiki/File:Private_key_signing.svg</a>
URL05	<a href="https://ftp.debian.org/debian/dists/buster/">https://ftp.debian.org/debian/dists/buster/</a>
URL06	<a href="https://ftp.debian.org/debian/dists/buster/InRelease">https://ftp.debian.org/debian/dists/buster/InRelease</a>

## LISTE DES TABLEAUX

Tableau 1: Tests exécutés sur une VM Debian de 2 CPUs à 2666 MHz (GNS3, template « Debian 11 ISC » ).....	5
Tableau 2: Comparaison des outils d'automatisation .....	38

## LISTE DES ANNEXES

Annexe 1 – Lien du dépôt GIT du projet .....	83
Annexe 2 – Arborescence des outils fs-verity .....	84
Annexe 3 – Script du service de monitoring fs-verity .....	85

## INTRODUCTION

Ce document est le résultat d'un travail de Bachelor effectué dans le cadre d'une formation en Informatique et Systèmes de Communication à la Haute École du Paysage, d'Ingénierie et d'Architecture (HEPIA) de Genève. Ce travail a débuté fin avril 2022 et a une durée de 4 mois. Il est la continuation du travail de semestre, pendant lequel on a effectué une première recherche, prise en main et évaluation des outils et méthodes qui ont été développés et étudiés plus en profondeur dans le travail de Bachelor. En particulier, le travail de semestre visait à étudier une solution de vérification de l'intégrité d'une infrastructure informatique et en vérifier la faisabilité.

En effet, à partir des années 2000, avec la diffusion de la virtualisation, il y a eu une explosion du nombre de serveurs. Leur création a été énormément simplifiée ne nécessitant plus de hardware dédié, aussi grâce aux outils d'automatisation de configuration et déploiement. Cela a porté à une forte croissance d'architectures réseau comprenant des dizaines, centaines, voire milliers de serveurs que les administrateurs systèmes doivent gérer. Un tel nombre de machines implique également des risques liés à la sécurité, surtout si les machines sont exposées sur internet.

Éviter une compromission des données est donc d'importance fondamentale. Ceci est normalement fait en amont, au travers de configurations de sécurité telles que pare-feu, antivirus, chiffrement et tout autre type de mesure préventives. Toutefois, il peut arriver qu'un acteur malveillant réussisse à pénétrer un système d'une manière ou d'une autre, par exemple en exploitant des failles pas encore connues ou divulguées (zero-day attack). Dans ces cas, il est important d'avoir en place un système qui permette d'identifier toute modification non voulue et notifier l'administrateur système afin de prendre des mesures correctives, qui peuvent varier selon la gravité du problème : de la simple modification d'un paramètre à la réinstallation complète de la machine infectée.

En tenant compte du fait que les serveurs sont configurés au travers d'un processus de déploiement continu, soit un déploiement assez fréquent de fonctionnalités fait de manière automatisée, il faut s'appuyer sur une méthode compatible avec des changements réguliers et qui puisse distinguer entre ceux intentionnels et ceux malveillants. Par exemple si les fichiers de configuration d'un service web sont modifiés, on aimerait avoir une méthode pour savoir si ces changements étaient bien légitimes.

Pour réaliser ce travail de Bachelor on a dans un premier temps effectué des recherches sur les outils et méthodes actuels, pour approfondir ce qui avait été fait lors du travail de semestre. La première phase du projet était donc dédiée à l'état de l'art, qui a été suivi par la conception et l'implémentation des outils choisis. La conception de l'architecture a été reprise du travail de semestre et, à part des petites adaptations, elle est restée essentiellement la même. Quant à l'implémentation, il s'agit d'un travail qu'on a effectué à partir de zéro, puisque on n'avait pas produit de logiciels ou script lors du travail de semestre.

Pour effectuer les recherches nécessaires nous nous sommes basés principalement sur des recherches web et la documentation officielle des outils utilisés. En effet, certains des outils analysés sont très rarement ou pas du tout présents dans des papiers scientifiques ou des livres, certaines informations peuvent se trouver uniquement dans les pages des manuels Linux ou dans des RFC. Toutefois, on a également pu s'aider avec des livres spécialisés dans les arguments qu'on a abordé dans ce rapport, notamment le livre « Network Programmability and Automation » de J. Edelman, S. S. Lowe et M. Oswalt et le livre « Computer & Internet Security » de W. Du.

Pour la partie pratique, tous les différents essais ont été effectués avec l'outil GNS3, qui est un simulateur de réseaux graphique et qui permet de reproduire virtuellement une infrastructure informatique comprenant serveurs, postes clients, switch, routeurs, etc. C'est un outil qui permet le déploiement et l'utilisation d'un grand nombre de machines de manière assez rapide et intuitive, réduisant ainsi le temps nécessaire à la préparation des tests. De plus, étant accessible à partir d'une page web, il évite également l'installation de programmes spécifiques. Et si on ne souhaite pas utiliser un navigateur pour la gestion des machines, on peut accéder à ces dernières via un accès SSH. Toutes les captures d'écran qui affichent un terminal ont été prises par nous-mêmes, ainsi on ne citera pas la source à chaque fois.

Ainsi, le rapport qui suit sera divisé en trois chapitres principaux : l'état de l'art, la conception et l'implémentation.

Dans le chapitre de l'état de l'art, on expliquera les concepts d'intégrité et d'authenticité, ainsi que fournir des outils qui permettent de vérifier l'intégrité de fichiers et volumes, d'autres qui en vérifient l'authenticité et d'autres qui combinent les deux. On verra également comment fonctionnent les gestionnaires de paquets et leur liaison à ces deux aspects de la sécurité informatique. Le chapitre se terminera avec une explication de l'infrastructure définie par du code et l'analyse des outils d'automation d'infrastructure informatique, ainsi que leur comparaison pour choisir celui qui nous convient le mieux.

Le deuxième chapitre reprendra l'architecture d'implémentation du code et servira d'aperçu pour le dernier chapitre. L'architecture sera expliquée à l'aide de trois types de diagrammes et modèles, chacun détaillé dans un sous-chapitre. Un premier sous-chapitre dédié au modèle C4, qui a plusieurs niveaux de détails, de la vue globale du système à une vue détaillée des composants individuels. Un deuxième sous-chapitre dédié au diagramme de déploiement, qui détaille les relations entre les différents éléments de l'architecture. Le dernier sous-chapitre concernera le diagramme de séquence, qui affiche l'ensemble des tâches à effectuer selon un schéma temporel.

Le troisième et dernier chapitre de ce document sera dédié à l'implémentation de la solution. Notamment comment on a adapté les outils existants pour les utiliser selon nos besoins, ainsi que la théorie et la pratique du développement d'un module personnalisé qui permet d'exploiter les outils trouvés et les adapter facilement à une infrastructure définie par du code. Le chapitre continuera avec une proposition de comment la gestion des alertes en cas d'attaque peuvent être mises en place et se poursuivra avec un sous-chapitre dédié aux exemples d'utilisation du produit développé. Le chapitre se terminera avec des considérations de sécurité sur notre projet ainsi qu'une vue sur des possibilités d'extension, intégration et améliorations futures du produit.

Le document se terminera avec une conclusion qui rappelle les différentes étapes du projet, qui fournit un retour réflexif sur ce travail et qui donne une aperçue sur des nouvelles perspectives et questionnements.

Enfin, des annexes seront également fourni sous forme de lien vers un répertoire GIT et qui permettront de reproduire le travail effectué dans le cadre de ce projet.

# 1. CHAPITRE 1 : ÉTAT DE L'ART

Toute infrastructure informatique, de la plus simple à la plus complexe, doit pouvoir être considéré sûre et fiable : garantir sa sécurité est donc fondamental pour pouvoir y faire confiance. Mais quand on a un très grand nombre de machines pour lesquelles on doit ne garantir la sécurité, cela peut devenir vite très complexe et chronophage. Dans ce chapitre on va donc analyser différentes solutions pour pouvoir garantir l'intégrité et l'authenticité des fichiers présents sur des serveurs, ainsi qu'étudier des solutions pour pouvoir mettre en place ce contrôle de manière simple et efficace, surtout quand on doit travailler avec des infrastructures comprenant des centaines de serveurs.

## 1.1. INTÉGRITÉ

Pour s'assurer qu'un serveur n'ait pas été compromis, une des façons les plus efficaces est de vérifier l'intégrité des fichiers les plus importants à son fonctionnement. On dit qu'un fichier est intègre quand il n'a pas subi d'altérations par rapport à son état initial. Mais avoir un fichier intègre n'est pas suffisant, puisqu'il pourrait s'agir d'un fichier mis là par un attaquant. Pour cela on veut également s'assurer qu'il soit authentique, soit qu'il ait été introduit dans le système par une personne ou programme de confiance, dont l'identité ait été vérifiée.

Une des façons pour garantir l'intégrité est avec des fonctions à sens unique, dont la plus utilisée en informatique est le hachage (hash en anglais). Un algorithme de hachage est une fonction qui, à partir de données de n'importe quelle taille, fournit une représentation de longueur fixe de ces données. Ainsi, un hash d'un fichier d'1Ko ou d'1Go aura toujours la même taille. Ils existent plusieurs fonctions de hachage, les plus répandues sont MD5, SHA-1, SHA-256, ainsi que les moins connus Blake2 et RIPEMD-160.

Les premières trois se basent sur la construction de Merkle-Damgård qui, comme des algorithmes similaires, permet de trouver assez rapidement le hash des données fournis en entrée, mais rend très difficile le processus inverse, soit déduire les données à partir du hash. En plus, il garantit une faible probabilité de collision, soit la probabilité d'avoir le même hash pour des types de données différents. Enfin, l'algorithme est fait de telle manière que, si même un seul bit des données initiales devait changer, le résultat final serait complètement différent. Grâce à ce



processus, le hash est une manière très efficace de sécuriser l'information, notamment les mots de passes.

La seule manière pour retrouver les données initiales si on a seulement le hash, c'est d'essayer toutes les combinaisons possibles. Cela demande énormément de ressources et/ou de temps, même si avec les composants (cartes graphiques) de dernière génération qui permettent de calculer plusieurs milliards de hash par seconde, certains algorithmes dont MD5 et SHA-1 ne sont plus recommandés dans certaines applications de sécurité (par ex. hachage de mots de passes). Les technologies actuelles permettent de casser des mots de passes hachés en seulement quelques mois, comparé à plusieurs années il y a 5-10 ans.

En travaillant sur des fichiers, notre projet est moins sensible à ce type de problèmes, donc même un algorithme comme SHA-1 peut-être une solution viable. À titre d'exemple, on a calculé le temps qu'il faudrait pour calculer le hash d'un fichier de 1Go avec les différents algorithmes cités (en utilisant la fonction `time` sous Linux). Ci-dessous un tableau avec les résultats.

	<b>MD5</b>	<b>SHA-1</b>	<b>SHA-256</b>	<b>Blake2</b>	<b>RIPEMD-160</b>
<b>Temps [s]</b>	2.814	3.590	6.998	3.025	6.065
<b>Taille [bits]</b>	128	160	256	256	160

*Tableau 1: Tests exécutés sur une VM Debian de 2 CPUs à 2666 MHz (GNS3, template « Debian 11 ISC »)*

Comme on peut le voir, à part SHA-256 et RIPEMD-160, les autres algorithmes prennent aux alentours de 3 secondes par Go. Ces temps doivent être adaptées à la puissance de la machine qui effectue les hachages et également à la taille du fichier, pour laquelle on a une corrélation linéaire, donc un fichier de 2Go prendra le double du temps qu'un fichier d'1Go (capture ci-dessous)

```

iti@AG1:~$ ls -lh
total 12K
-rw-r--r-- 1 iti iti 2.0G May 23 12:20 abc.txt
drwxr-xr-x 4 iti iti 4.0K Mar 30 11:48 ansible
drwxr-xr-x 3 iti iti 4.0K Apr 6 12:49 builds
-rw-r--r-- 1 iti iti 1.0G May 22 13:07 hello.txt
iti@AG1:~$ time openssl sha1 hello.txt
SHA1(hello.txt)= 2bf31513b8b9b67df2d2ad2f06abe903de89c00b

real    0m3.620s
user    0m2.971s
sys      0m0.639s
iti@AG1:~$ time openssl sha1 abc.txt
SHA1(abc.txt)= 91d50642dd930e9542c39d36f0516d45f4e1af0d

real    0m7.123s
user    0m5.760s
sys      0m1.347s
iti@AG1:~$

```

*Illustration 1: résultats test vitesse de hachage*

Le résultat du hachage est une chaîne de caractères, d'habitude en format hexadécimal, de la longueur définie par l'algorithme et est appelé empreinte du fichier.

En conclusion, pour vérifier l'intégrité d'un fichier, la méthode privilégiée est celle de calculer son hash une première fois, le stocker et ensuite le comparer avec le hash calculé au moment où on souhaite vérifier l'intégrité du fichier.

On va maintenant voir quels sont les différentes manières d'effectuer le stockage et la comparaison des hash. On présentera en premier des outils qu'on trouve sous forme de programmes à installer, qui sont prêts à l'utilisation et qui demandent peu ou pas de configuration pour être opérationnels après installation. Ensuite, on verra des outils de hachage plus élaborés, implémentés dans le kernel Linux, qui fournissent des méthodes plus sûres d'effectuer la vérification d'intégrité des fichiers d'une machine.

## 1.2. OUTILS DE VÉRIFICATION D'INTÉGRITÉ DE FICHIERS

Après avoir récupéré le hash d'un fichier, il faut trouver une manière efficace de le comparer avec un autre hash de référence. Pour cela, on va voir 6 outils open-source qui sont disponibles sous Debian. Ces outils permettent de faire un screenshot d'un volume entier ou de ses parties pour pouvoir le comparer périodiquement avec un état futur et vérifier l'intégrité de ses fichiers. La liste d'outils a été tirée du manuel Debian (chapitre 4.17.3) et les informations qui les concernent ont été prises principalement des pages de leur manuel correspondant ou leur page web officielle.

## **sXid**

C'est un outil qui vérifie s'il y a eu des changements sur tous les fichiers et dossiers *suid* et *sgid* en se basant sur sa dernière vérification. Les termes *suid* et *sgid* se réfèrent aux droits sur les fichiers ; en particulier l'exécution est toujours faite par le propriétaire du fichier, indépendamment de l'utilisateur qui a lancé la commande (cela pour *suid*, *sgid* est la même chose mais fait référence aux groupes).

Les résultats de l'opération de vérification sont stockés dans un fichier de logs dans `/var/log/sxid.log`. Les changements sont envoyés par mail à l'adresse spécifié dans le fichier de configuration (`/etc/sxid.conf`).

Le format de sortie est assez intuitif avec des + et - pour indiquer les ajouts/suppressions de fichiers et dossiers et des flèches -> pour indiquer les changements de groupes/utilisateurs (ancien -> nouveau). Enfin, la lettre m avant le nom d'un fichier indique que son hash sha256 a changé.

## **AIDE**

Advanced Intrusion Detection Environment (AIDE) est un outil qui de détection d'intrusion qui permet de vérifier l'intégrité de fichiers et dossier sous Linux. De manière similaire à sXid il intègre une base de données dans laquelle sont stockés les hash ainsi que les informations des fichiers (permissions, inode, taille, user, group, xattrs, etc.). Il supporte également les « regular expression », qui permettent d'inclure, exclure et filtrer les fichiers et dossiers facilement. Ce qu'AIDE doit stocker, ainsi que la liste de fichiers et dossier à monitorer, peuvent être configurés dans le fichier de configuration `/etc/aide/aide.conf`.

AIDE tourne en tant que Cron Job, mais la vérification de l'intégrité peut être faite aussi avec des commandes manuelles (commande `aide --check`). Le résultat de la vérification est affiché au terminal/à la console et sauvegardé dans les logs `/var/log/aide`.

Enfin, AIDE peut être sécurisé avec SELinux (Security-Enhanced Linux) avec des règles de contrôle des accès.

## **tripwire**

Tripwire est un outil créé par une entreprise qui porte le même nom, qui offre services liés à la sécurité et à la détection d'intrusions. L'outil est open-source et son fonctionnement est similaire à sXid et AIDE : il est composé d'une database qui contient les hash des fichiers à protéger qui ont été définis dans un fichier de configuration (appelé policy file). À côté de cela,

tripwire inclut des fonctionnalités additionnelles : la plus importante est la génération de clés de chiffrement qui sont utilisées pour la génération de la signature et du certificat qui protégeront la configuration, les politiques de sécurité et la database de tripwire. Deux clés sont générées : une « clé de site » qui va être utilisée pour toute l'infrastructure et une « clé locale » qui sera propre à la machine que tripwire monitore.

Autre fonctionnalité est la génération de rapports, qui se distinguent de simples logs grâce à leur format clair et détaillé. Le rapport peut également être envoyé par mail automatiquement.

Enfin, la possibilité de modifier facilement la base de données contenant les hash, dans le cas où certaines modifications seraient légitimes. L'accès à la base de données est protégé par mot de passe.

Tripwire n'a pas non plus de service associé, il faut donc l'exécuter à intervalles réguliers, par exemple via cron (toutefois cela est fait automatiquement à l'installation).

### **integrit**

Cet outil se base sur le même principe que les autres : il construit une database avec les hashes des fichiers et qui est comparée avec une nouvelle base créée lors du processus de vérification. Après la comparaison, la base devient lecture-seule. Integrit fournit un outil assez basique et simple pour la vérification d'intégrité et manque de beaucoup de fonctionnalités présentes dans les autres outils déjà vus (par ex. ne fonctionne pas avec des fichiers de grande taille, la configuration n'accepte pas les « regular expressions », sécurisation avec des clés de chiffrement, etc.).

### **samhain**

On peut considérer Samhain comme un vrai outil de détection d'intrusion, car il met à disposition un grand nombre de fonctionnalités utiles à cet effet. Par exemple analyse de logs, détection de rootkit (programmes qui permettent à un attaquant d'accéder à la machine infectée), détection de processus cachés, monitoring de ports et vérification de l'intégrité. La première version a été créée en 2001.

Le principe de fonctionnement pour la vérification de l'intégrité reste le même que les autres outils : calcul, stockage et comparaison des hash. Cette fonction de vérification est activée par défaut et tourne en tant que service.

Samhain peut fonctionner en standalone mais fournit aussi un système centralisé de monitoring et maintenance, avec une console web pour sa gestion. Il est aussi modulaire, dans le

sens où toutes les fonctionnalités peuvent être activés ou désactivés selon les besoins de l'administrateur système.

### **Debsums**

Le dernier outil est debsums, qui est légèrement différent des autres : son but est de vérifier l'intégrité des packages Debian. En effet, si on peut s'assurer que les seuls programmes qu'on installe n'ont pas été compromis, on aura garanti leur intégrité.

Avec debsums un hash MD5 est effectué sur les paquets installés et vérifié contre les hash présents sous `/var/lib/dpkg/info/*.md5sums`. Comme les hash vérifiés sont ceux stockés sur la machine, si cette dernière est compromise, pour un attaquant serait facile de remplacer les hash présents dans les fichiers `.md5sums` avec des faux hash. Il faut donc tenir cela en compte dans le cas où cet outil serait utilisé.

## **1.3. OUTILS AVANCES DE VERIFICATION D'INTEGRITE DE FICHIERS**

Sur le même principe de base des outils vus auparavant, soit le hachage de fichiers et la comparaison avec des hash de référence, ils existent des outils qui permettent d'effectuer cette opération de manière beaucoup plus sûre et efficace. Ces sont des outils implémentés dans l'architecture de l'OS, ou quand même qui sont très proche de la couche kernel, et qui fournissent non seulement plus de fonctionnalités, mais aussi une vérification plus avancée de l'intégrité d'un système.

### **a) DM-VERITY**

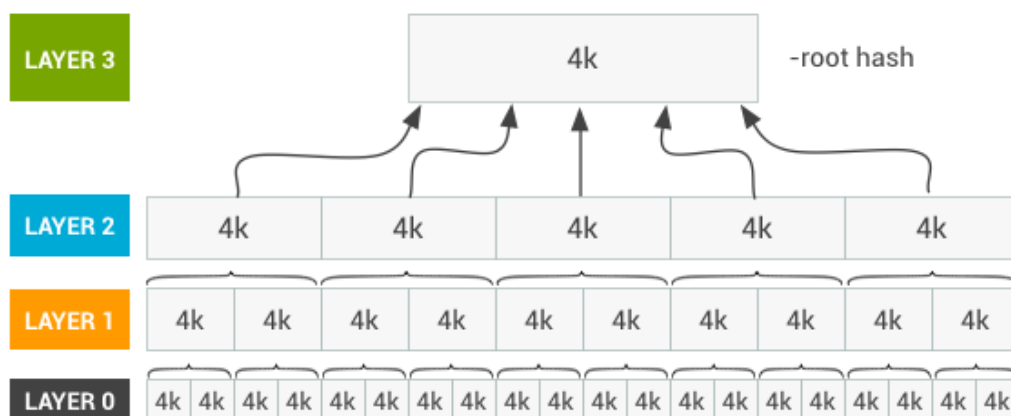
Le premier outil qu'on va présenter s'appelle dm-verity. Il a été introduit en 2011 pour être utilisé avec ChromeOS, mais actuellement sa plus grande implémentation se trouve dans le système opératif Android, néanmoins il peut fonctionner sur tout système Linux.

Le but de dm-verity est de fournir un moyen de vérifier l'intégrité du système au niveau kernel, donc au niveau le plus bas de l'architecture d'un OS. Cela est accompli grâce au hachage de la partition qu'on souhaite protéger, en particulier le hash est effectué pour chacun des blocs de données qui composent la partition. Stocker un hash pour chacun des blocs est une manière assez inefficace de vérifier l'intégrité : si la taille d'un block est de 4096 octets et un hash du block fait 160bits (pour SHA-1), le hash représentera environ le 0.5% de la taille totale du disque. Sur un

disque de 100Go cela correspond seulement à 500Mo, mais ils existent des moyens d'optimiser cela ultérieurement, notamment avec un arbre de hachage.

## Fonctionnement

Un arbre de hachage ou arbre de Merkle, est une construction qui effectue des hash de hash pour pouvoir réduire l'espace nécessaire à garantir l'intégrité d'un élément. Une représentation de son fonctionnement est affichée ci-dessous.



*Illustration 2: Arbre de Merkle dm-verity.*

*Source : tiré de android.com, ref. URL01*

Après avoir calculé les hash de chacun des blocs, les hash sont regroupés jusqu'à correspondre à la taille d'un bloc et ensuite hachés de nouveau. Ce processus est répété jusqu'à qu'il ne reste qu'un seul hash : le hash racine de l'arbre. Selon ce principe, un arbre de Merkle pour une partition de 100Go aurait une taille de seulement 160bits à la place que 500Mo. Et cette taille resterait fixe peu importe la taille de la partition, soit qu'elle fasse 10Go ou 10To. En vérité, même si c'est très efficient en espace, ça devient inefficent pour la vérification de l'arbre, puisqu'il faudra recalculer tous les hash à chaque fois. Pour remédier à cet inconvénient, outre que la racine, aussi les hash des feuilles sont stockés sur le disque. Ainsi, quand on souhaite vérifier un certain bloc, seulement les hash de la feuille qui le concerne doivent être calculé, réduisant le temps de vérification et d'accès. De plus, les calculs sont effectués en parallèle.

Une fois l'arbre obtenu, il est stocké dans une table de mappage, qui sera utilisée pour vérifier l'intégrité des blocs. Outre que l'arbre, la table de mappage contient les informations relatives aux différents blocs pour les identifier (taille, offset, hash, sel d'hachage).

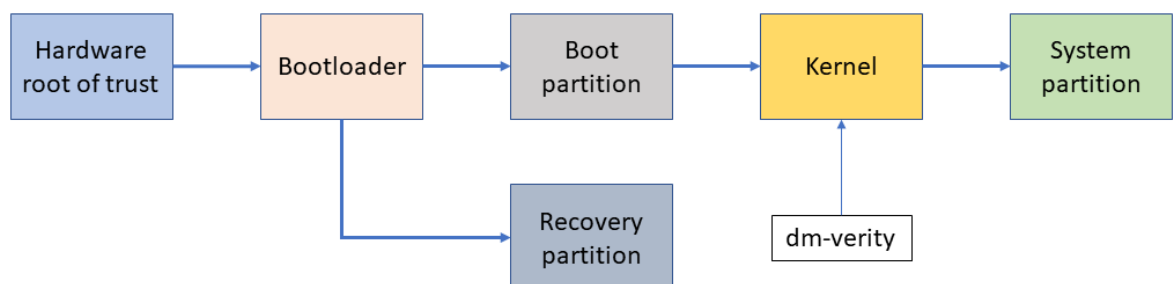
Comme cette table sera la source de confiance de toute la partition, il est fondamental d'assurer son authenticité. Cela s'obtient en signant la table. La clé utilisée pour la signature est encodée en dur dans le matériel sur lequel tourne le système (par exemple dans un TPM si disponible). Comme elle ne peut pas être modifiée, dans le cas où cette clé serait compromise, le changement du matériel serait requis aussi. Accéder à ces clés sans autorisation est une tâche extrêmement complexe, ce qui réduit énormément la probabilité qu'elles puissent être compromises.

Après la signature, la table de mappage est stockée dans la partition de boot. Pour la sécurité, cela implique que cette partition soit considérée absolument sûre et de confiance. Si c'est le cas, la clé de signature peut être stockée dans cette partition, rendant un TPM pas nécessaire.

Enfin, la partition qu'on souhaite protéger est rendue lecture-seule. Le processus de vérification d'intégrité est effectué uniquement à chaque fois qu'on souhaite accéder à un bloc faisant partie de cette partition. De cette manière il n'est pas nécessaire de vérifier l'entièreté de la partition lors du démarrage ou lorsqu'on la monte. Si le processus de vérification devrait échouer, cela veut dire que le bloc a été compromis et le système interdira son accès.

Pour éviter qu'un fichier sain soit considéré comme malveillant, il a été introduit (avec la version 7.0 d'Android) un système de correction d'erreurs (Forward Error Correction). Des codes de correction tel que Reed-Solomon sont utilisés dans le cas d'Android : ils permettent par exemple de récupérer jusqu'à 16-24Mo de blocs à la fois, en occupant seulement le 0.8% de l'espace de la partition.

Dm-verity est partie intégrante de la chaîne de confiance qui a lieu sur un système Android et plus en particulier du verified boot, soit l'ensemble d'étapes mises en œuvre pour assurer l'intégrité du système. Le processus complet (simplifié) de démarrage est résumé dans le schéma ci-dessous.



*Illustration 3: Schéma de la chaîne de confiance du processus de démarrage Android*

*Source : réalisé par l'auteur.*

## **b) IMA**

L'architecture de mesure de l'intégrité (Integrity Measurement Architecture) a été intégrée au kernel Linux à partir de la version 2.6.30. IMA est désactivée par défaut dans les distributions Linux actuelles, pour son activation le kernel doit être recompilé et activer l'option « CONFIG\_IMA », ainsi qu'un certain nombre d'autres options facultatives, dans le fichier de configuration de la partition de boot.

Cette architecture fonctionne de façon similaire à dm-verity. Une première différence est qu'elle fonctionne au niveau des fichiers plutôt que des blocs. En particulier, la première étape à être effectuée est la « mesure » d'un fichier, soit le calcul de son hash. Cet hash est stocké dans une liste qui se trouve soit dans le sous-dossier « <securityfs>/integrity/ima », où <securityfs> est la partition à sécuriser, soit elle est ajoutée dans le module TPM si présent, à la fin du PCR (Platform Configuration Register) qui est un espace de mémoire dans le TPM.

L'étape suivante consiste à signer les valeurs sauvegardées. Cela est fait grâce à une clé privée stockée dans le TPM (si disponible). Comme évoqué pour dm-verity, une signature effectuée de cette manière ne peut pas être falsifiée, puisqu'un attaquant n'aura pratiquement jamais accès à la clé privée. À noter que les clés utilisées pour la signature doivent être créées manuellement.

À partir de Linux 3.7, une nouvelle fonctionnalité a été ajoutée à IMA, appelé IMA-Appraisal (évaluation en anglais). C'est aussi une option à activer dans le kernel lors du démarrage (donc pas besoin de recompiler). IMA-Appraisal ajoute une couche de vérification en assurant que le fichier pour lequel on souhaite garantir l'intégrité soit bien le fichier correct. Cela est fait en ajoutant le hash du fichier à vérifier dans un attribut étendu (security.ima). Si les valeurs ne correspondent pas, l'accès au fichier est interdit. À remarquer que l'attribut security.ima contient un hash qui n'est pas signé, ce qui est pratique quand on doit modifier le fichier, mais n'assure pas une forte protection de l'intégrité et de l'authentification.

Pour remédier à cet inconvénient, IMA est souvent couplé à EVM (Extended Verification Module) : alors que IMA est utilisé pour garantir l'intégrité des fichiers, EVM est l'équivalent mais au niveau des attributs étendus des fichiers. EVM est disponible à partir de Linux 3.3 et doit être activé lors de la compilation du kernel. EVM fournit deux méthodes pour aider avec la détection d'altération de fichiers. Une méthode consiste à stocker en tant qu'attribut étendu security.evm la valeur HMAC des attributs. HMAC (keyed-Hash Message Authentication Code) est un code calculé à partir du hash et d'une clé secrète, permettant ainsi de fournir à la fois



l'intégrité et l'authenticité. La deuxième méthode permet de vérifier l'intégrité des attributs en recalculant le HMAC et en le comparant avec celui contenu dans `security.evm`.

Enfin, on a la dernière étape qui est celle de vérification de l'intégrité des fichiers. Si le hash du fichier auquel on essaye d'accéder ne correspond pas à celui stocké dans le PCR, l'accès au fichier sera interdit. Pour encore plus de sécurité, IMA inclut un module IMA-audit qui permet d'ajouter les hash des fichiers dans les logs d'audit système.

Les étapes de la protection d'intégrité avec IMA se résument de la façon suivante :

- 1) Hachage des fichiers
- 2) Stockage des hash
- 3) Signature de la liste des hash
- 4) Comparaison du hash d'un fichier avec un hash stocké dans les attributs étendus de ce dernier
- 5) Protection des attributs étendus
- 6) Vérification des hash

Depuis mars 2022, IMA intègre un mode supplémentaire de vérification appelé `fs-verity` et qui est discuté dans le chapitre suivant.

### **c) FS-VERITY**

FS-Verity est une couche de support intégrée dans le kernel Linux sur laquelle les systèmes de fichiers peuvent s'appuyer pour garantir l'intégrité et l'authenticité des fichiers en lecture seule. Il est actuellement supporté par les systèmes de fichiers `ext4` et `f2fs` et est disponible à partir de la version 5.4 du kernel Linux (date de sortie : 24 novembre 2019). Le projet a été créé par Eric Biggers et Michael Halcrow, deux employés de Google. Les informations qui suivent ont été extraites à partir de la documentation officielle du kernel Linux et de son code source.

Le principe de fonctionnement est similaire à celui de `dm-verity`, avec la différence qu'il permet de protéger des fichiers individuels plutôt qu'une partition entière. Cela ne veut pas dire que `fs-verity` remplace `dm-verity`. Ce dernier doit être utilisé pour des partitions en lecture seule, alors qu'on emploie `fs-verity` sur des fichiers qui sont sur des partitions en lecture-écriture. Comme pour `dm-verity`, le fonctionnement se base sur la construction d'un arbre de Merkle du fichier au travers d'une `ioctl` (appel système pour la gestion des entrées/sorties). À chaque lecture d'un fichier sur lequel on a activé `fs-verity`, le kernel vérifie l'intégrité du fichier contre l'arbre de Merkle. Si

le fichier est corrompu et que son arbre ne correspond pas à celui stocké en mémoire, l'ouverture échouera. Au travers d'une autre ioctl, on peut récupérer le hash racine de l'arbre et cela est fait en temps constant, indépendamment de la taille du fichier. On peut donc dire que fs-verity est un moyen d'hacher un fichier en temps constant, avec la particularité qu'une lecture qui violerait le hash, échouerait.

## Fonctionnement

Les mêmes principes valables pour l'arbre de Merkle dans dm-verity s'appliquent à fs-verity. Le fichier est décomposé en blocs d'une taille prédéfinie (4096 octets par défaut) et le dernier bloc est rempli (padded en anglais) avec des zéros pour arriver à 4096. Ces blocs sont hachés et les hash vont composer le premier niveau de l'arbre. Ils sont à nouveau regroupés en blocs et hachés jusqu'à arriver au hash racine. Il y a également la possibilité d'ajouter un sel de hachage qui permet de protéger encore mieux les données. Le sel fait en sorte que deux blocs identiques auront un hash différent, rendant ainsi plus difficiles des attaques du type rainbow table, où des attaquants utilisent une liste de hash précalculés pour essayer de retrouver les données à l'origine du hash.

Comme pour dm-verity, pour accélérer l'ouverture d'un fichier, on ne calcule pas tout l'arbre de Merkle à chaque fois, mais uniquement les branches qui concernent les blocs auxquels on souhaite accéder. Pour cette raison, la racine de l'arbre peut être ambiguë. Par exemple, elle ne peut pas faire la différence entre un grand fichier et un petit fichier dont les données sont exactement les mêmes que le hash du niveau de l'arbre le plus élevé.

En particulier, comme spécifié dans les commentaires du code source (fichier verify.c), pour une question d'efficacité, le système de fichiers pourrait mettre en mémoire cache les hash des blocs. Cela fait donc qu'il suffit de vérifier l'arbre uniquement jusqu'à un bloc qui a déjà été vérifié.

Pour résoudre ce problème, fs-verity ne se base pas seulement sur la racine de l'arbre pour effectuer les vérifications, mais aussi sur d'autres données qui, avec le hash de la racine, composent ce qu'on appelle le fs-verity file digest, une structure de données appelée fsverity\_descriptor définie comme suit :

```
struct fsverity_descriptor {
    __u8 version;           /* must be 1 */
    __u8 hash_algorithm;    /* Merkle tree hash algorithm */
    __u8 log_blocksize;     /* log2 of size of data and tree blocks */
    __u8 salt_size;         /* size of salt in bytes; 0 if none */
    __le32 __reserved_0x04; /* must be 0 */
    __le64 data_size;       /* size of file the Merkle tree is built over */
    __u8 root_hash[64];     /* Merkle tree root hash */
};
```

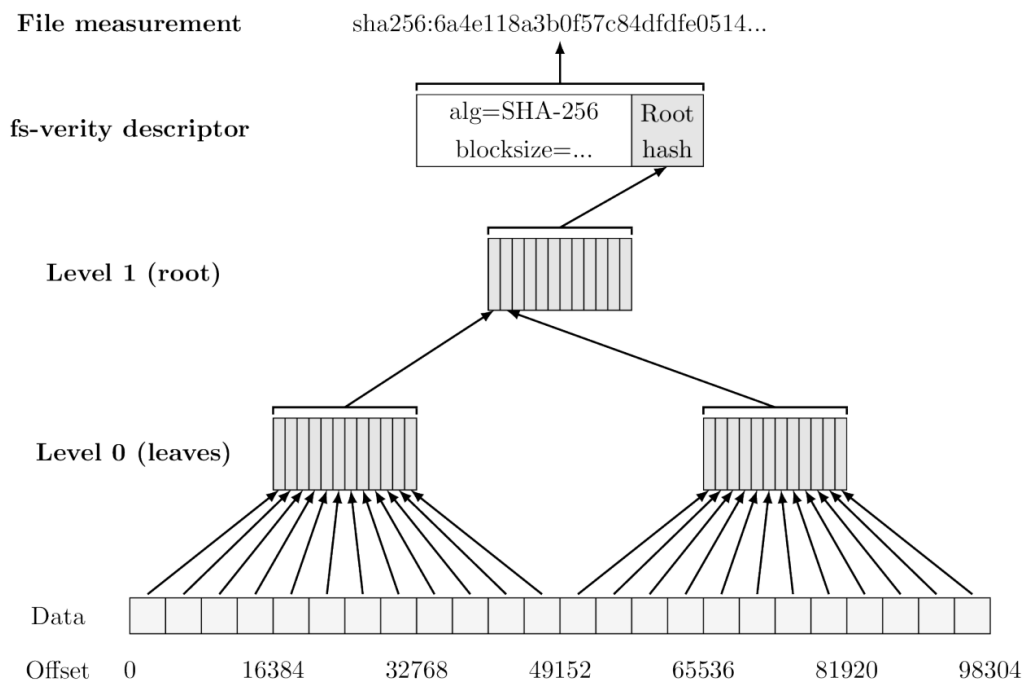
```

    __u8 salt[32];          /* salt prepended to each hashed block */
    __u8 __reserved[144];   /* must be 0's */
};

```

On y retrouve donc non seulement le hash racine, mais d'autres informations importantes telles que l'algorithme de hachage et la taille du fichier.

Ci-dessous une représentation graphique de l'arbre de Merkle et de la structure de données.



*Illustration 4: Arbre de Merkle fs-verity.*

*Source : tiré de la présentation par Mike Halcrow et Eric Biggers du 27 Août 2018, Linux Security Summit, URL02*

## User API

FS-verity met à disposition plusieurs ioctl pour la gestion des fichiers.

FS\_IOC\_ENABLE\_VERITY sert à activer fs-verity sur un fichier et prend comme paramètre un pointeur sur la structure suivante, qui contient les paramètres utiles à la construction de l'arbre de Merkle et éventuellement une signature :

```

struct fsverity_enable_arg {
    __u32 version;
    __u32 hash_algorithm;
    __u32 block_size;
    __u32 salt_size;
    __u64 salt_ptr;
    __u32 sig_size;
    __u32 __reserved1;
    __u64 sig_ptr;
    __u64 __reserved2[11];
};

```

```
};
```

Ces paramètres doivent être initialisés comme suit :

- `version` : doit être 1 ;
- `hash_algorithm` : l'algorithme de hachage des blocs, actuellement uniquement SHA256 et SHA512 sont supportés (comme spécifié dans la librairie `fsverity.h`) ;
- `block_size` : la taille du bloc de l'arbre de Merkle. Par défaut 4096 octets. Autres tailles ne sont pas supportées pour le moment ;
- `salt_size` : la taille du sel d'hachage, en octets. S'il n'y a pas de sel, il faut l'initialiser à zéro. La taille maximale est 32 octets ;
- `salt_ptr` : le pointeur vers le sel ou valeur NULL s'il n'y a pas de sel ;
- `sig_size` : la taille de la signature, en octets. S'il n'y a pas de signature, il faut l'initialiser à zéro. La taille maximale est 16128 octets ;
- tous les champs réservés doivent être initialisés à zéro.

À l'exécution, cette `ioctl` demande au système de fichiers de générer l'arbre de Merkle pour un fichier et le stocke à un endroit propre au système de fichiers. Ensuite, le fichier est marqué comme fichier `fs-verity`. Cette `ioctl` peut prendre du temps si exécuté sur des fichiers de grosse taille et peut-être interrompue par des signaux d'arrêt critiques (`fatal signal` en anglais).

Aussi, pour activer `fs-verity` avec cette `ioctl`, il faut ouvrir le fichier en lecture seule. Il ne peut pas être déjà ouvert en écriture par un autre processus et ne peut pas être ouvert pendant l'exécution de l'`ioctl`, peine de recevoir un message d'erreur. Cela pour s'assurer qu'il ne sera pas possible d'écrire dans le fichier après que `fs-verity` ait été activé et pour s'assurer que son contenu reste le même lors de la génération de l'arbre de Merkle. Si `fs-verity` a été activé correctement, l'appel à l'`ioctl` retournera zéro, un code d'erreur autrement.

Une autre `ioctl` est `FS_IOC_MEASURE_VERITY`, qui sert récupérer le hash `fs-verity` du fichier, aussi appelé mesure du fichier. L'`ioctl` prend en paramètre un pointeur à la structure suivante :

```
struct fsverity_digest {
    __u16 digest_algorithm;
    __u16 digest_size; /* input/output */
    __u8 digest[];
};
```

Où :

- `digest_algorithm` est l'algorithme utilisé pour le hash et sera donc le même que celui utilisé lors de l'activation de `fs-verity` sur le fichier ;

- digest\_size est la taille du hash (donc 32 octets pour SHA256 par exemple) ;
- digest correspond au hash.

Si le digest a été récupéré correctement, la fonction retournera zéro, un code d'erreur sinon. Cette opération est exécutée en temps constant, peu importe la taille du fichier.

Pour accéder à tous les données fs-verity d'un fichier, on peut utiliser encore une autre ioctl qui est FS\_IOC\_READ\_VERITY\_METADATA et qui prend en paramètre la structure suivante :

```
struct fsverity_read_metadata_arg {
    __u64 metadata_type;
    __u64 offset;
    __u64 length;
    __u64 buf_ptr;
    __u64 __reserved;
};
```

Qui se décompose comme suit :

- metadata\_type, qui spécifie le type de donnée qu'on souhaite récupérer et peut être initialisé avec trois valeurs différentes :

- FS\_VERITY\_METADATA\_TYPE\_MERKLE\_TREE
- FS\_VERITY\_METADATA\_TYPE\_DESCRIPTOR
- FS\_VERITY\_METADATA\_TYPE\_SIGNATURE

Qui permettent de récupérer respectivement l'entiereté de l'arbre de Merkle, le fs-verity descriptor comme vu avant et la signature fs-verity du fichier ;

- offset, qui est le décalage auquel se trouvent les métadonnées ;
- length, la longueur des métadonnées lus. Ça peut être moins que la longueur totale si par exemple l'appel à l'ioctl a été interrompu ;
- buf\_ptr, le pointeur vers le buffer, soit l'endroit où on souhaite stocker les données pour pouvoir les traiter ;
- \_\_reserved, qui doit être initialisé à zéro.

Un point important ici est savoir que les métadonnées fs-verity sont stockées dans un endroit spécifique au système de fichiers, notamment dans les blocs qui suivent la fin du fichier. Cela implique que ces métadonnées ne sont pas directement accessibles à partir de l'espace utilisateur et que si un acteur malveillant voudrait les modifier il faudrait qu'il ait accès aux blocs composants la partition, ce qui veut dire avoir les droits d'administrateur (root) sur le kernel Linux.

Un autre point également important ici est que cette ioctl n'est disponible qu'à partir de la version du kernel Linux 5.12 et que la version du kernel utilisée par Debian 11 (la plus récente

version stable de Debian) est la 5.10. Une mise à jour du kernel est donc nécessaire si on a besoin d'utiliser cette ioctl.

Certaines de ces commandes ioctl peuvent être exploitées aussi grâce à des outils mis à disposition dans le paquet « fsverity » (à installer avec apt par exemple ou on peut télécharger les sources et les compiler pas nous-même), notamment l'activation de fs-verity, le calcul du hash correspondant et la signature d'un fichier.

## Signature

Le processus de signature d'un fichier fs-verity ainsi que de sa vérification se fait en plusieurs étapes et peut se faire de différentes manières.

FS-Verity intègre un mécanisme de vérification de la signature de la part du kernel, qui est défini par l'option `CONFIG_FS_VERITY_BUILTIN_SIGNATURES` (active par défaut). Au moment de l'initialisation de fs-verity par le kernel, un keyring « .fs-verity » est créé et ceci contiendra le certificat contre lequel le kernel vérifiera la signature. Si besoin, on peut utiliser un appel système pour restreindre l'accès au keyring (`sysctl keyctl_restrict_keyring()`) et éviter qu'on puisse ajouter d'autres certificats. En effet, si un attaquant arrivait à ajouter un certificat au keyring, il pourrait signer lui-même des fichiers et le kernel n'aurait pas moyen de savoir qu'il s'agit de fichiers potentiellement dangereux.

Un autre appel système permet de rendre obligatoire la signature d'un fichier fs-verity (`sysctl fs.verity.require_signatures=1`), en assurant donc que tout fichier soit authentique.

La signature s'effectue avec l'ioctl `FS_IOC_ENABLE_VERITY` que, comme vu dans le paragraphe précédent, prend en paramètre une structure qui contient un pointeur vers la signature. En particulier, la signature sera en format PKCS#7 et sera construite à partir d'une structure similaire à celle du digest. La nouvelle structure contient en plus un tableau de caractères contenant le mot « FSVerity » :

```
struct fsverity_formatted_digest {
    char magic[8];                /* must be "FSVerity" */
    __le16 digest_algorithm;
    __le16 digest_size;
    __u8 digest[];
};
```

Comme vu dans le paragraphe précédent, la signature est stockée avec les métadonnées fs-verity d'un fichier et reste accessible uniquement par le kernel (ou à l'espace utilisateur via une ioctl). À chaque ouverture du fichier, le kernel vérifie la signature contre le certificat qui est stocké

dans le keyring « .fs-verity » : si la vérification échoue, le fichier ne s'ouvrira pas et une erreur sera retournée.

Ce mécanisme de vérification se base uniquement sur le kernel, mais on peut imaginer une vérification basée entièrement dans l'espace utilisateur, de la même façon dont on signerait un fichier standard.

Enfin, on peut imaginer une combinaison des deux mécanismes : signer le fichier manuellement et puis laisser au kernel sa vérification. C'est sur ce dernier mécanisme (qu'on appellera hybride) qu'on basera notre implémentation.

## **1.4. AUTHENTICITÉ**

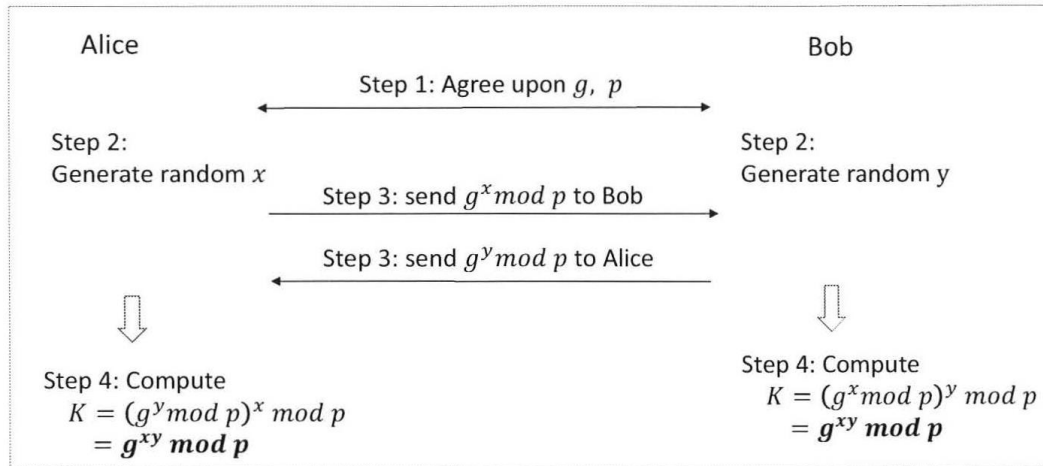
Lorsqu'on souhaite garantir l'intégrité d'un fichier, on peut le faire aussi en s'assurant que l'entité qui le fournit c'est quelqu'un de fiable. Ceci est fait grâce à la signature numérique. La signature garantit non seulement l'intégrité d'un fichier mais aussi son authenticité, soit que l'identité de l'auteur du fichier correspond bien à celle attendue. Une troisième conséquence est la non-répudiation, puisque qui signe un fichier ne peut pas nier l'avoir signé.

La signature en elle-même n'est pas suffisante pour attester de l'identité du signataire. Par exemple, dans le cas d'une usurpation d'identité, on ne pourra pas garantir l'authenticité de la signature. On l'accompagne alors d'un certificat qui est établi par une Autorité de Certification à laquelle on fait confiance. Si toutefois cette Autorité serait compromise, il y a la possibilité de révoquer le certificat et éviter qu'il soit utilisé par un acteur malveillant. Et si la signature a été horodatée, on peut également garantir que tous les fichiers signés avant la compromission soient authentiques. L'horodatage assure donc l'anti-rejeu de la signature, puisque on ne peut pas signer un fichier avec une date antérieure à celle de révocation du certificat.

### **a) ALGORITHMES DE SIGNATURE**

La sécurité d'une signature est due au fait qu'elle est créée avec un algorithme de chiffrement asymétrique. Ce type d'algorithme utilise deux clés : une clé privée qui n'est jamais divulguée intentionnellement et une clé publique qui peut être mise à disposition de tout le monde sans que ça représente des risques pour la sécurité. Le chiffrement asymétrique est né comme alternative au chiffrement symétrique, où une seule clé est utilisée pour le chiffrement et le déchiffrement. La problématique ici est le partage de la clé, soit comment la transmettre sans qu'elle soit compromise. Ce problème a été résolu avec une méthode d'échange appelée Diffie-

Hellman, qui garantit la sécurité de la clé même quand on communique sur un canal non sécurisé. Le fonctionnement de l'algorithme de création de la clé est représenté ci-dessous et on peut voir comme Alice et Bob obtiennent la clé privée de manière sécurisée : même en interceptant les échanges, il ne serait pas possible de déduire facilement  $x$  ou  $y$ .



*Illustration 5: Schéma de l'algorithme de chiffrement Diffie-Hellman.*

*Source : tiré du livre Computer and Internet Security, A Hands-On Approach, 3rd Edition,*

*Wenliang Du*

Dans le chiffrement asymétrique, les clés privée et publique peuvent être à la fois utilisée pour chiffrer un message et pour le déchiffrer. En particulier, dans le cas de la signature, on utilise la clé privée pour signer et la clé publique pour vérifier la signature. En revanche, si on souhaite utiliser ces clés pour transmettre un message chiffré, on utilisera la clé publique de la personne à laquelle on souhaite envoyer le message pour le chiffrement, de façon que le destinataire pourra déchiffrer les données à l'aide de sa clé privée. Une représentation de ces deux cas est montrée ci-dessous.



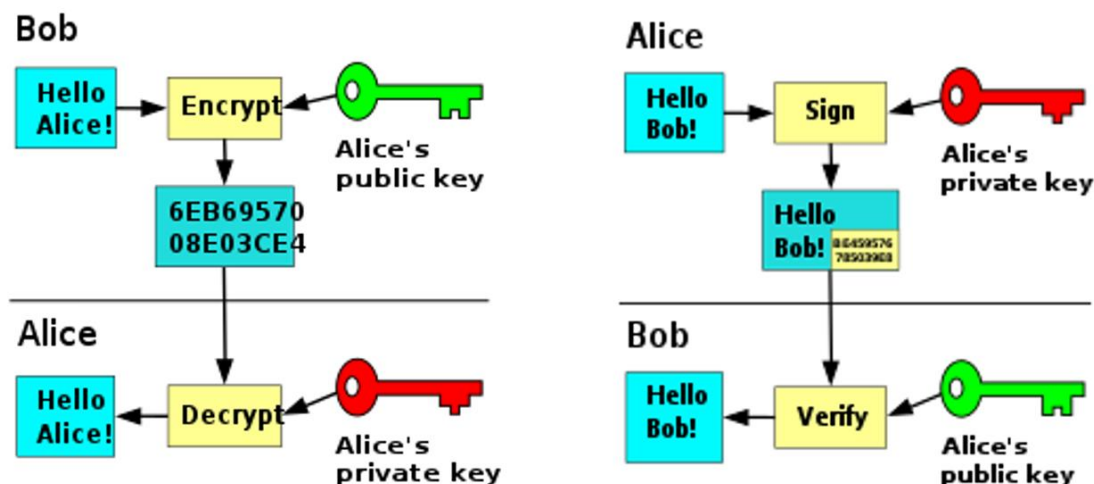


Illustration 6: Représentation de l'utilisation des clés privées et publiques : à gauche pour le chiffrement, à droite pour la signature

Source : tiré de wikipedia.org, URL03, URL04

Même si ces deux clés sont liées mathématiquement, on ne peut pas déduire facilement la clé privée à partir de celle publique. En effet, les algorithmes de génération de ces clés, de la même manière que les fonctions d'hachage, sont tels qu'il est très simple de générer les clés, mais très difficile computationnellement de connaître les données à l'origine de ces clés. Un des algorithmes les plus connus utilisés pour la génération des clés est RSA, où deux nombres premiers aléatoires sont utilisés comme base pour garantir la sécurité des données chiffrées. La puissance de cet algorithme se base sur le fait qu'actuellement il n'existe pas de méthode mathématique de calculer facilement des nombres premiers. RSA est un algorithme qui, même pour le chiffrement, est relativement lent computationnellement. Pour cette raison, il ne devrait pas être utilisé pour chiffrer directement les données à protéger, mais on l'utilise plutôt sur le hash de ces derniers. Autres algorithmes souvent utilisés dans le cadre des signatures sont DSA (Digital Signature Algorithm) et ses dérivés ECDSA et EdDSA.

## b) CERTIFICATS

Lors de la vérification d'une signature, on utilise la clé publique de la personne qui a signé le fichier. Pour prouver la validité de la clé publique, on utilise un certificat. On trouve les certificats sous différents formats, mais le plus diffusé est le X.509, qui a été défini dans la RFC 5280 et par l'UIT. Ce format contient un certain nombre d'informations qui sont utilisées pour attester de la validité de la signature :

- Numéro de version

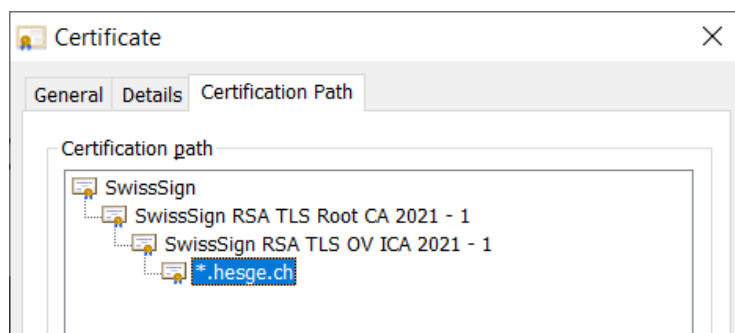
- Numéro de série unique du certificat
- Algorithme utilisé pour la signature
- Nom de l'émetteur
- Période de validité (date d'origine et d'expiration)
- Nom du sujet (CN, Common Name)
- Informations de la clé publique du sujet : l'algorithme utilisé pour sa génération

ainsi que la clé publique

- Des champs optionnels pour ajouter d'autres informations
- Algorithme de signature du certificat
- Signature du certificat

X.509 utilise la syntaxe ASN.1 (Abstract Syntax Notation One) qui le rend compatible avec tout type de plateforme. On le trouve sous différentes extensions et formats d'encodage : pem, crt, p7b et p7s, encodés en ASCII ; der, cer, pfx et p12 qui sont encodés en binaire.

Un exemple très commun d'utilisation des certificats est lors des connexions sécurisées à des sites web, pour garantir que le site qu'on visite est bien légitime. Ci-dessous une capture d'écran qui montre la chaîne de confiance des certificats pour le site [hepia.hesge.ch](https://hepia.hesge.ch) :



*Illustration 7: Certificats du site [hepia.ch](https://hepia.hesge.ch)*

*Source : réalisé par l'auteur*

On voit que le certificat a été délivré par SwissSign, qui est une Autorité de Certification (CA) et à qui on peut faire confiance pour nous assurer que le certificat soit valable. Dans le cadre de ce projet, on générera le certificat par nous-mêmes et on fera confiance à la personne de l'organisation qui a généré le certificat. En effet, il n'y a pas besoin de s'appuyer sur une CA telle que SwissSign, puisque le certificat restera à l'intérieur de notre infrastructure.

## c) EXEMPLES

On va maintenant voir comment générer des clés publiques et privées, comment les utiliser pour signer un fichier et vérifier sa signature et enfin comment générer le certificat correspondant.

Ils existent plusieurs outils et programmes qui permettent de générer des clés de chiffrement. Par exemple l'outil `ssh-keygen`, qui avec la simple commande homonyme permet de générer les clés publiques et privées. Il est utilisé principalement dans le cadre de la communication `ssh`. On pourrait même écrire nous même un script à partir de l'algorithme de chiffrement, mais l'outil qu'on présentera ici est `openssl`.

Les options à disposition sont très nombreuses. Un exemple de création de clé privée est montré ci-dessous, avec la commande `openssl genrsa -aes128 -out private.pem 1024`

```
iti@AG1:~/signature$ openssl genrsa -aes128 -out private.pem 1024
Generating RSA private key, 1024 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
Enter pass phrase for private.pem:
Verifying - Enter pass phrase for private.pem:
```

*Illustration 8: Génération d'une clé privée avec openssl*

L'option `genrsa` dit au programme qu'on souhaite générer une clé privée avec le chiffrement AES128 (option `-aes128`). La clé est sauvegardée avec l'option `-out` suivie du nom de la clé (`private.pem` dans ce cas). Le 1024 à la fin indique la longueur de la clé en octets. Si on affiche la clé que l'on vient de générer avec la commande `cat`, on aura le résultat suivant :

```
iti@AG1:~/signature$ cat private.pem
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4, ENCRYPTED
DEK-Info: AES-128-CBC, 1B1549C735102447AA463E24BD63B83E

pzZ5xVe0aVktNmRIufMj5yK7FieOap38eJW4QMggrkqfiyqTJNndt4utC2rBDCGu
tDXVgwe7sc2sQbPBca9I5VNz6/7m35BFNvp0OWOOp27nE9bflNb01BSP9KK+fLI9
5ztAXmvAwgkWuCVJ4an29gaAiYFC15IiOXwDKmtx+9NvLMdX+6FyAc5U6YCSQVux
au5ec2wr3PkYQkhBc7194KSiR/TnyRbYmDYIUJ0rlTlWjgHc3Hq4G8eTYh1sQWju
lPqZzt8PR5iilPfe8vaBVtTK9x2iwU6fgrtymzaM/jYigycPCTf2iyy70Xf29mBT
K5u/trgChYkrQ0yEt0IW+uTGMmtpHw3RE3N84jQE7Z34Q7IoBtxzGhd3GL8hFFMQ
p50KJTLqWEbYZhnm0d5b2nHJcDFJWldMEacaZ+Z4zx2ntaCWA6kEDkQz2vSjl3cq
lVoM7gkUcTex+XMjNxve3j00X9DjTXfa8fVl2QWvui8B2fDhB+J1HtwQ+e98a/qT
hWwFRxqBEWgWdyRhYO2+izSvv3Lhab/6ED6XokIN+XKq7PK0YG5pbF43YED0jtg
djwgbKdyxjdQuEpET/psV8loIgrv+noKcN6/Pf/wseFwjmsitQcaZIKSiMi57Q+M
OVRSGImPrHAMLdw2zYr4ISJrE+IP7Mucf+LJ+4IK+fvNGoJacuuvV0lgm04O9+1G
75o9QR9ejgRuJeRaysvJ3oe56sdSbcIG5pentMoFSjr0e24U8ExzKozTU2uKsV6x
3aBP7QwdJQyjnjp/fdHvRibpzlOv57dNqPbe0hF6cDvPBXpy4nZWRFZ9HquH21X/
-----END RSA PRIVATE KEY-----
```

*Illustration 9: Affichage d'une clé privée avec cat*

Le contenu de la clé n'est pas lisible, car encodé en Base64. Pour pouvoir lire le contenu de la clé on peut utiliser la commande suivante `openssl rsa -in private.pem -noout -text`. Ci-dessous un affichage partiel du résultat :

```
iti@AG1:~/signature$ openssl rsa -in private.pem -noout -text
Enter pass phrase for private.pem:
RSA Private-Key: (1024 bit, 2 primes)
modulus:
 00:c4:79:85:cc:a1:0f:5c:f9:21:05:8f:9e:81:c9:
 a2:3d:6e:47:06:89:f2:1a:01:8f:4a:0a:15:e8:00:
 8e:75:1d:7b:d9:f0:2e:b5:ef:a8:f8:9a:fa:dd:04:
 b8:35:39:5e:b3:08:b3:9e:2e:e6:35:88:49:47:9b:
 1d:1d:30:86:4f:a5:a9:12:a1:7a:8e:34:b6:31:9d:
 29:d4:75:29:48:c1:bd:d0:8f:fd:34:95:86:ca:1d:
 6e:a2:ca:aa:1c:6a:fc:b5:e5:90:ae:e6:7e:3a:41:
 09:bb:a9:73:10:29:40:e7:0d:80:17:d1:85:9d:1e:
 14:e6:99:cf:bf:43:7a:dc:21
publicExponent: 65537 (0x10001)
privateExponent:
 00:a6:b6:1a:c3:24:52:bd:e8:22:8d:ee:6c:67:e2:
 68:88:c9:9e:f8:bc:6a:32:88:ea:45:bf:c3:10:c6:
 10:1a:ed:f6:9e:59:7f:b9:1e:ef:78:6b:40:bb:f3:
 7a:3e:a7:3c:64:9b:d8:95:3b:64:59:3e:18:37:eb:
 35:4c:04:55:a1:96:56:4d:22:b8:6e:a6:6d:c3:fd:
 93:f3:6c:9e:43:78:d3:d0:95:3d:4d:40:5d:65:c2:
 f1:27:49:be:d4:c8:ca:a5:89:af:b8:a1:9c:7e:b3:
 98:33:71:86:52:4c:c2:94:63:42:8c:ed:7e:ff:f8:
 eb:24:15:50:2a:6f:68:49:31
prime1:
 00:fc:00:09:aa:e0:f1:04:e3:94:f8:24:d3:cc:25:
 1d:8c:53:da:63:12:b4:b0:1a:6c:00:cb:07:84:b3:
 bc:5e:c8:d4:f8:88:ba:66:cd:b9:82:e2:57:e7:02:
 1f:69:3a:b0:90:f6:d8:cf:ad:de:a8:71:e3:48:8b:
 52:7d:35:54:cd
prime2:
 00:c7:97:dd:b9:f7:e9:6f:b7:e6:69:20:99:d9:c3:
 d3:02:26:de:06:5d:c1:c7:88:b0:2e:ca:96:a9:35:
 d2:e2:af:87:6e:52:39:18:56:b5:74:09:6c:c3:f0:
 3c:0e:10:87:5c:dd:24:07:b1:b8:8e:e0:ac:55:e2:
 2e:65:e8:04:a5
exponent1:
 7c:8a:74:3c:ac:40:d1:5e:bc:2f:ca:db:95:a7:be:
 2b:ad:01:60:37:4d:6f:3f:0f:a3:70:b3:bd:84:aa:
 69:48:91:15:34:31:53:5f:56:4a:75:09:65:31:09:
```

*Illustration 10: Affichage d'une clé privée avec openssl*

On verra ainsi les valeurs utilisées par l'algorithme lors de la création de la clé.

À partir de la clé privée on peut générer la clé publique :

```
iti@AG1:~/signature$ openssl rsa -in private.pem -pubout > public.pem
Enter pass phrase for private.pem:
writing RSA key
iti@AG1:~/signature$
```

*Illustration 11: Génération d'une clé publique avec openssl*

En l'affichant en Base64 on a :

```
iti@AG1:~/signature$ cat public.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDEeYXMoQ9c+SEFj56ByaI9bkcG
ifIaAY9KChXoAI51HXvZ8C6176j4mvrdBLg1OV6zCL0eLuY1iElHmx0dMIzPpakS
oXqONLYxnSnUdSlIwb3Qj/001YbKHW6iyqocavy15ZCu5n46QQm7qXMQKUDnDYAX
0YWdHhTmmc+/Q3rcIQIDAQAB
-----END PUBLIC KEY-----
```

*Illustration 12: Affichage d'une clé publique avec cat*

Et son contenu actuel est le suivant :

```
iti@AG1:~/signature$ openssl rsa -in public.pem -pubin -noout -text
RSA Public-Key: (1024 bit)
Modulus:
 00:c4:79:85:cc:a1:0f:5c:f9:21:05:8f:9e:81:c9:
 a2:3d:6e:47:06:89:f2:1a:01:8f:4a:0a:15:e8:00:
 8e:75:1d:7b:d9:f0:2e:b5:ef:a8:f8:9a:fa:dd:04:
 b8:35:39:5e:b3:08:b3:9e:2e:e6:35:88:49:47:9b:
 1d:1d:30:86:4f:a5:a9:12:a1:7a:8e:34:b6:31:9d:
 29:d4:75:29:48:c1:bd:d0:8f:fd:34:95:86:ca:1d:
 6e:a2:ca:aa:1c:6a:fc:b5:e5:90:ae:e6:7e:3a:41:
 09:bb:a9:73:10:29:40:e7:0d:80:17:d1:85:9d:1e:
 14:e6:99:cf:bf:43:7a:dc:21
Exponent: 65537 (0x10001)
```

*Illustration 13: Affichage d'une clé publique avec openssl*

Maintenant qu'on a généré la clé privée, on peut générer aussi le certificat. On se sert toujours d'openssl, avec la commande suivante :

```
openssl req -x509 -key private.pem -out certificate.crt -days 60
```

Où *req* est l'option pour générer un certificat, qu'on souhaite avoir en format X.509. On indique le chemin de notre clé privée ainsi que le chemin de sortie du certificat. On peut également choisir une date de validité dudit certificat avec l'option *-days*. À l'exécution de cette commande on est invités à indiquer les informations qui rempliront les champs du certificats, telles que le pays, la ville, l'adresse mail et ainsi de suite.

```

iti@AG1:~/signature$ openssl req -x509 -key private.pem -out certificate.crt -days 60
Enter pass phrase for private.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CH
State or Province Name (full name) [Some-State]:Genève
Locality Name (eg, city) []:GE
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Hepia
Organizational Unit Name (eg, section) []:ISC
Common Name (e.g. server FQDN or YOUR name) []:Test Certificate
Email Address []:sergio.guarino@etu.hesge.ch

```

*Illustration 14: Génération d'un certificat avec openssl*

Si on ouvre le certificat avec un éditeur de texte, il sera en Base64 comme les clés, mais on peut afficher les données contenues avec openssl :

```

iti@AG1:~/signature$ openssl x509 -in certificate.crt -noout -text
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      22:f8:55:7b:a9:a4:19:84:4b:7f:4b:46:7e:f8:89:6b:34:76:ca:bc
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: C = CH, ST = Genève, L = GE, O = Hepia, OU = ISC, CN = Test Certificate, emailAddress = sergio.guarino@etu.hesge.ch
    Validity
      Not Before: Jul 26 17:22:10 2022 GMT
      Not After : Sep 24 17:22:10 2022 GMT
    Subject: C = CH, ST = Genève, L = GE, O = Hepia, OU = ISC, CN = Test Certificate, emailAddress = sergio.guarino@etu.hesge.ch
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public-Key: (1024 bit)
      Modulus:
        00:c4:79:85:cc:a1:0f:5c:f9:21:05:8f:9e:81:c9:
        a2:3d:6e:47:06:89:f2:1a:01:8f:4a:0a:15:e8:00:
        8e:75:1d:7b:d9:f0:2e:b5:ef:a8:f8:9a:fa:dd:04:
        b8:35:39:5e:b3:08:b3:9e:2e:e6:35:88:49:47:9b:
        1d:1d:30:86:4f:a5:a9:12:a1:7a:8e:34:b6:31:9d:
        29:d4:75:29:48:c1:bd:d0:8f:fd:34:95:86:ca:1d:
        6e:a2:ca:aa:1c:6a:fc:b5:e5:90:ae:e6:7e:3a:41:
        09:bb:a9:73:10:29:40:e7:0d:80:17:d1:85:9d:1e:
        14:e6:99:cf:bf:43:7a:dc:21
      Exponent: 65537 (0x10001)
    X509v3 extensions:
      X509v3 Subject Key Identifier:
        6F:7B:DF:05:BD:1F:5D:45:41:01:5C:7F:69:A7:49:FB:F4:EF:A0:A9
      X509v3 Authority Key Identifier:
        keyid:6F:7B:DF:05:BD:1F:5D:45:41:01:5C:7F:69:A7:49:FB:F4:EF:A0:A9
      X509v3 Basic Constraints: critical
        CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
      82:d2:15:b8:d2:63:0f:ac:84:90:87:50:cc:9d:f9:71:a0:99:
      eb:41:ac:ef:b5:61:91:de:da:ea:63:c1:ab:5e:61:5f:fc:6c:
      ce:73:ac:db:de:d1:41:f7:27:d4:16:a7:5b:c5:63:2d:1d:97:
      c7:7f:29:09:f8:20:94:92:81:c3:73:45:8a:1a:9d:e9:7b:04:
      45:50:9f:94:01:1e:f2:89:7b:20:f2:0b:4e:e2:59:9e:fe:01:
      12:80:db:7d:42:7a:6f:14:7c:0e:3e:92:88:2b:69:67:bf:a7:
      dd:39:62:f2:3b:eb:a5:f0:42:99:8b:66:0b:60:64:36:3f:64:
      27:0b

```

*Illustration 15: Affichage d'un certificat avec openssl*

On retrouve les informations qu'on a rentré lors de la création ainsi que les autres champs listés dans le paragraphe sur les certificats. On remarque que la clé publique est bien la même que celle qu'on a générée à partir de la clé privée.

On va maintenant voir comment effectuer l'opération de signature d'un fichier, ou, pour être plus précis, de son hash. Pour cet exemple on a créé un fichier *hello.txt* pour lequel on extrait son hash en format hexadécimal (ou en binaire si on le souhaite) avec l'option *sha256* d'openssl. On sauvegarde le résultat dans un fichier qu'on a appelé *hello.hash* :

```
iti@AG1:~/signature$ cat hello.txt
Signature test file
iti@AG1:~/signature$ openssl sha256 hello.txt > hello.hash
iti@AG1:~/signature$ cat hello.hash
SHA256(hello.txt)= 380be5471cfce883e7b44d000cf7cb642fc2ed03369aaa4c54bfd67f23afe68
```

*Illustration 16: Hachage d'un fichier avec openssl*

Pour générer une signature on utilisera l'option *rsautl* d'openssl, qui prend en paramètre la clé privée et le hash du fichier en donnant comme sortie la signature de ce dernier.

```
openssl rsautl -sign -inkey private.pem -in hello.hash -out hello.sig
```

On peut visualiser la composition en hexadécimal (hexdump) de la signature avec le programme *xxd* :

```
iti@AG1:~/signature$ xxd hello.sig
00000000: 034e 11b2 aba3 a91c e73b 8c05 40de f045  .N.....;..@..E
00000010: b8eb aaa2 537e 3333 0e9a ccf0 5a44 d675  ....S~33....ZD.u
00000020: 015f a87e e752 77f8 099e 9378 af45 9792  ._~.Rw....x.E..
00000030: b4b3 db5c 139f 0e61 442c 102f 6174 1fb6  ...\.aD,./at..
00000040: 82d6 8b1f ce41 36d5 1746 d2dc cffc df3f  ....A6...F....?
00000050: e9db f080 d628 17e6 fb83 e673 e283 5bb2  ....(....s..[.
00000060: d899 308a fd3b d0a8 4a05 f2a3 ec5c 11b9  ..0...;..J....\..
00000070: a784 6be4 0bcc edb8 cc3c dd91 bf0b 1d4a  ..k.....<.....J
```

*Illustration 17: Affichage de la signature d'un fichier*

La vérification de la signature consiste à extraire le hash d'origine à partir de la signature et de la clé publique précédemment générée. L'option *-verify* nous permet de faire cela :

```
iti@AG1:~/signature$ openssl rsautl -verify -inkey public.pem -in hello.sig -pubin
SHA256(hello.txt)= 380be5471cfce883e7b44d000cf7cb642fc2ed03369aaa4c54bfd67f23afe68
iti@AG1:~/signature$ cat hello.hash
SHA256(hello.txt)= 380be5471cfce883e7b44d000cf7cb642fc2ed03369aaa4c54bfd67f23afe68
```

*Illustration 18: Vérification de la signature d'un fichier*

On peut voir qu'on obtient bien le même résultat que le hash d'origine. La signature est donc valable et le fichier est intègre. Si on essaie de modifier le fichier, son hash changera et par conséquent la signature ne sera plus valable.

## 1.5. INTÉGRITÉ ET AUTHENTICITÉ DES PACKAGES

Quand on souhaite installer un nouveau programme sous Linux, on a besoin des fichiers sources contenant la configuration, les exécutables, les libraries, et les informations nécessaires à leur installation et intégration dans l'OS. Au moins de compiler un programme à partir de son code source (par ex. avec un *makefile*), on le retrouve sous forme d'archive. L'extension change selon

l'OS sur lequel il doit être installé, par exemple on a .deb pour Debian, .rpm pour Fedora, .apk pour Android, .msi pour Windows, etc.

Comme il y a un très grand nombre de programmes disponibles, leur gestion peut devenir vite assez complexe, surtout si on considère le fait que beaucoup de programmes ont également besoin de programmes supplémentaires pour fonctionner (appelés dépendances en anglais). C'est là qu'un gestionnaire de paquets se relève utile : il s'occupe de l'installation complète d'un paquet et de ses dépendances et également des mises à jour ou de sa suppression, simplifiant et accélérant les tâches à effectuer.

Dans les sections suivantes on va présenter deux outils gestionnaires de paquets, un pour Debian, l'autre pour Fedora et on verra leur principe de fonctionnement, ainsi que les aspects de sécurité correspondants.

#### **a)       APT**

Advanced Packaging Tool (APT) est un gestionnaire de paquets pour Debian. Il est disponible uniquement par ligne de commande et il fournit un ensemble de méthodes pour la gestion des paquets en se basant sur un gestionnaire plus bas niveau, dpkg.

Comme apt se base sur dpkg pour son fonctionnement, on va étudier principalement apt, mais il est important de savoir que beaucoup de commandes font appel à dpkg et que ce que apt fait est de les regrouper et les présenter sous une forme plus simple, rapide et intuitive à utiliser.

Pour installer des paquets, apt effectue plusieurs opérations. Tout d'abord, via la commande « apt update », la liste des paquets disponibles est mise à jour en téléchargeant le fichier InRelease (ou Release) à partir des liens vers les dépôts de paquets (fichier /etc/apt/sources.list). Ce fichier contient la liste des indexes des fichiers des paquets, leur taille, leur hash MD5 et SHA-256 et enfin une signature du fichier InRelease. Le préfixe « In » dans « InRelease » indique que la signature se trouve à l'intérieur du fichier. Il existe aussi un fichier Release, qui est le même à exception du fait que sa signature est dans un fichier à part. Pour des raisons de sécurité, apt ne mettra pas à jour la liste des indexes s'il ne trouve pas de signature valable. Cette option peut toutefois être contournée si besoin. Le certificat de vérification de la signature des dépôts officiels Debian est inclus dans les fichiers d'installation de Debian.



```
iti@AG1:~$ cat /etc/apt/sources.list
# deb http://deb.debian.org/debian bullseye main









deb http://deb.debian.org/debian bullseye main
deb-src http://deb.debian.org/debian bullseye main

deb http://security.debian.org/debian-security bullseye-security main
deb-src http://security.debian.org/debian-security bullseye-security main

# bullseye-updates, to get updates before a point release is made;
# see https://www.debian.org/doc/manuals/debian-reference/ch02.en.html#_updates_and_backports
deb http://deb.debian.org/debian bullseye-updates main
deb-src http://deb.debian.org/debian bullseye-updates main
```

*Illustration 19: Exemple de fichier sources.list d'apt*

## Index of /debian/dists/buster

<a href="#">Name</a>	<a href="#">Last modified</a>	<a href="#">Size</a>
 <a href="#">Parent Directory</a>		-
 <a href="#">ChangeLog</a>	2022-03-26 11:04	5.1M
 <a href="#">InRelease</a>	2022-03-26 11:43	119K
 <a href="#">Release</a>	2022-03-26 11:37	116K
 <a href="#">Release.gpg</a>	2022-03-26 11:43	2.4K
 <a href="#">contrib/</a>	2022-03-26 11:37	-
 <a href="#">main/</a>	2022-03-26 11:37	-
 <a href="#">non-free/</a>	2022-03-26 11:37	-

Apache Server at ftp.debian.org Port 80

*Illustration 20: Exemple d'arborescence d'un dépôt Debian.*

*Source : tiré de [debian.org](https://www.debian.org), URL05*

```

-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

Origin: Debian
Label: Debian
Suite: oldstable
Version: 10.12
Codename: buster
Changelogs: http://metadata.ftp-master.debian.org/changelogs/@CHANGEPATH@_changelog
Date: Sat, 26 Mar 2022 11:35:38 UTC
Acquire-By-Hash: yes
Architectures: amd64 arm64 armel armhf i386 mips mips64el mipsel ppc64el s390x
Components: main contrib non-free
Description: Debian 10.12 Released 26 March 2022
MD5Sum:
768b559d904c71effb13f1c617f44ad9 1356807 contrib/Contents-amd64
66985257414014fa05a35df566904129 102603 contrib/Contents-amd64.gz
f15ab65dc63759004ec988abecd65e9e 1075476 contrib/Contents-arm64
ecc9200020eb4b13aa4813171eb9118d 84016 contrib/Contents-arm64.gz
676e2e1aba5bac2737acba11457c1b80 1072443 contrib/Contents-armel
a797ac094917d1cf47a9225a7a9c3216 83766 contrib/Contents-armel.gz
47bc4a4cd68ebc9481fab48e1f0b9003 1080904 contrib/Contents-armhf
82aa19149bdf972afba35c8e9e77bb17 84798 contrib/Contents-armhf.gz
0961029e41979df9f5e9307ec1263dc8 1143854 contrib/Contents-i386
53d5a05a5949ece1122f4a00063199ad 90446 contrib/Contents-i386.gz
3c268400200bc805cafcc108c5188e44 1075642 contrib/Contents-mips
527e948e4357fbb7bd57ff03a426dc5e 84001 contrib/Contents-mips.gz
f3fbd545d9c657455e13228df11d6b50 1074552 contrib/Contents-mips64el
6cc26c41eb6f1c0984c8124c354e3937 83874 contrib/Contents-mips64el.gz
f0b81c783d3d131e7bc6e8650028cae6 1075659 contrib/Contents-mipsel
99663ce5790eda2371ff7f7e0b24b588 84014 contrib/Contents-mipsel.gz
6c85135a1a9a4f1eaa04108339e58045 1076324 contrib/Contents-ppc64el
3347cef1062fe2f4c0b83a0e3e0197ae 84035 contrib/Contents-ppc64el.gz
1535a8917c15a80e810021ff1e2607c4 1072966 contrib/Contents-s390x
16e48dca693f5f0a7dd57cf5c4eb7251 83555 contrib/Contents-s390x.gz
dc4c2199653a31f5dfa98270501976e9 4561198 contrib/Contents-source
b28e95e92838e437cc69006cbf7910fc 373687 contrib/Contents-source.gz
d41d8cd98f00b204e9800998ecf8427e 0 contrib/Contents-udeb-amd64
4a4dd3598707603b3f76a2378a4504aa 20 contrib/Contents-udeb-amd64.gz

```

*Illustration 21: Exemple de fichier InRelease.*

*Source : tiré de [debian.org](https://www.debian.org), URL06*

Les liens par défaut qui se trouvent dans le fichier « sources.list » sont ceux officiels de la distribution Linux en cours d'utilisation. Comme expliqué dans le manuel d'apt, ces dépôts sont maintenus et vérifiés par des personnes autorisées et uniquement un développeur officiel peut charger des mises à jour ou des nouveaux paquets. Toutefois, on est libres d'ajouter des liens à d'autres dépôts de paquets. Ce fichier pourrait donc constituer une possible cible d'une attaque puisque on pourrait y ajouter des liens vers des dépôts compromis.

Pour clarifier, la vérification de la signature du fichier Release veut dire que, du point de vue de la sécurité, on fait confiance à la personne qui a signé le fichier (donc le développeur officiel), mais n'implique pas qu'on puisse faire confiance aux paquets qu'on installe. En effet, apt ne vérifie pas la signature des paquets individuellement. Cela peut être réalisé avec d'autres commandes, telles que `debsig-verify` et `debsign`.

Dans le cas de Debian, la chaîne de confiance commence lorsqu'un développeur (maintainer) ajoute un nouveau paquet ou une mise à jour. Cet envoi sera signé avec une clé contenue dans le keyring de confiance de Debian. La clé est vérifiée par d'autres maintainers Debian pour garantir l'identité du développeur. Après l'ajout du paquet dans l'archive Debian, la signature est retirée et le hash du paquet est ajouté à un fichier Packages, qui à son tour est haché

et ajouté dans le fichier Release. Ce dernier est enfin signé et ajouté avec le fichier Packages et le paquet sur les serveurs miroirs officiels de Debian.

Grâce à cette procédure, deux types d'attaques peuvent être prévenus :

- Les attaques « man in the middle » où, s'il n'y avait pas de signature, un attaquant pourrait intercepter le téléchargement du paquet, en contrôlant certains éléments réseau comme un router ou un switch ou même en redirigeant le trafic (par ex. en compromettant le DNS).
- Les compromissions des serveurs miroirs : même si un attaquant arrivait à modifier le fichier Release, il ne pourrait pas falsifier la signature. De la même façon, il ne pourrait pas ajouter des paquets sans les signer avec une clé autorisée.

Si toutefois serait le serveur principal Debian à être compromis (donc celui qui contient les clés), la signature du fichier Release ne serait pas suffisante à se protéger.

Après avoir mis à jour la liste des paquets disponibles, il est possible de procéder à l'installation de ces derniers avec la commande « apt install nom\_du\_paquet ». Dans cette étape, apt s'occupe de télécharger et installer le paquet demandé. C'est à cette étape que apt installe également les dépendances de ce paquet, soit tous les autres programmes nécessaires à son bon fonctionnement. Le chemin d'installation sur l'hôte est propre à chaque paquet et il est spécifié dans les sources du paquet-même, les dossiers les plus importants et communs sont :

- /var/cache/apt/archives ici sont téléchargé les archives des paquets à installer.
- /etc, /usr/lib, /usr/bin, /bin, selon le programme installé, ses fichiers de configuration et exécutables peuvent se trouver dans un de ces répertoires ou sous-répertoires.
- /usr/lib/systemd/system, /etc/systemd/system et parfois /run/systemd/system sont les répertoires où on trouve les services (mais pas tous les programmes tournent aussi en tant que services).

Si on souhaite mettre à jour un paquet existant, il est possible de le faire au travers de deux commandes différentes. D'un côté on a « apt upgrade », qui effectue une mise à jour (si disponible) pour tous les paquets installés actuellement sur le système. D'un autre côté on a « apt full-upgrade », qui gère aussi les dépendances des paquets. En effet, si une dépendance n'est plus nécessaire, full-upgrade s'occupe de l'effacer, alors qu'un simple « upgrade » ne le fait pas.

Enfin, quand on souhaite effacer un programme, on a aussi plusieurs options :

- Apt remove, qui efface tout simplement les exécutables du programme (sans effacer les fichiers de configuration)

- Apt purge, qui efface les exécutables et les fichiers de configuration
- Apt clean, qui efface le cache des archives téléchargés
- Apt autoremove, qui s'occupe d'effacer les dépendances des paquets qui ne sont plus installés

## **b) DNF**

De manière similaire à Debian, Fedora possède un gestionnaire de paquets bas-niveau, appelé rpm, qui est l'équivalent de dpkg. C'est dnf (appelé aussi Dandified YUM), qui est une évolution d'un autre gestionnaire de paquets très populaire, yum. Ici on parle de Fedora, mais cela est valable pour toutes les distributions de Red Hat Linux, ainsi que toutes les distributions basées sur rpm.

Une première différence avec apt, est qu'il n'y a pas besoin de mettre à jour la liste des paquets disponibles, car dnf le fait automatiquement. On peut donc directement installer un nouveau paquet avec la commande « dnf install ». Les dépendances de ce paquet seront également installées. Il y a aussi la possibilité d'installer un ensemble de paquets pour un usage spécifique (commande « dnf group list » pour afficher la liste disponible), par exemple le groupe de paquets « Development Tools » inclut tous les programmes les plus souvent utilisés par des développeurs.

Les commandes « dnf remove » et « dnf autoremove » permettent respectivement d'effacer un programme ou d'effacer le programme avec ses dépendances.

Pour la mise à jour de tous les paquets, on utilise la commande « dnf upgrade », mais on peut également passer un paquet en paramètre et uniquement celui indiqué sera mis à jour.

Il existe également une option pour effectuer des mises à jour automatiquement, assurant ainsi d'avoir toujours la dernière version d'un programme installée. Cela peut être fait avec la commande dnf-automatic. Un fichier de configuration (etc/dnf/automatic.conf) permet de configurer certaines options comme l'intervalle auquel effectuer les mises à jour, l'ajout d'une adresse mail auquel envoyer des notifications suite à une mise à jour ou simplement pour être notifié de quand une mise à jour est disponible. Cette option de mise à jour automatique doit toutefois être utilisée avec prudence : lors d'une mise à jour, le programme n'est pas disponible et pourrait causer des interruptions de service. Aussi, une nouvelle version pourrait présenter des problèmes d'incompatibilité avec les autres programmes déjà installés. On en déconseille donc

l'utilisation pour effectuer des mises à jour automatiques, mais elle se révèle utile si on souhaite l'utiliser uniquement comme moyen de notification.

Enfin, une fonctionnalité utile est la commande « `dnf history` », qui permet d'afficher l'historique de tous les paquets installés, effacés, mis à jour. Pour cela il faut activer l'option « `history_record` » dans le fichier de configuration de `dnf`. Il est même possible d'effectuer un « `rollback` » à un certain point de l'historique et annuler toutes les opérations effectuées après un certain moment.

Les paquets en format `rpm` ne sont pas signés par défaut, toutefois on a disposition un certain nombre d'options pour pouvoir le faire. La signature est ajoutée après que les binaires et fichiers de configuration ont été rassemblés en un paquet. Pour pouvoir ajouter une signature `pgp`, on utilise la commande « `rpm --addsign` » ou « `rpmsign --signfiles` ». Un paquet peut être signé plusieurs fois et par plusieurs personnes. La clé de signature se trouve dans le keyring `rpm` et est installé avec l'OS, mais on peut aussi signer le paquet avec une clé personnelle.

Un paquet signé peut être intégré à IMA via le plugin « `rpm-plugin-ima` », cela permet d'ajouter la signature dans les attributs étendus du paquet (`security.ima`).

Une autre fonctionnalité implémentée dans `rpm` que nous ne trouvons pas dans `apt` ou `dpkg` est la possibilité de signer un paquet avec une signature `fs-verity`. En particulier, l'utilisation de `fs-verity` est divisée en deux étapes.

En premier lieu, lors de la création d'un paquet, l'arbre de Merkle utilisé par `fs-verity` est calculé pour chacun des fichiers composant le paquet. Pour chaque fichier, la racine de l'arbre est signée et la signature est ajoutée en tant que métadonnée du paquet `rpm` (plus précisément, l'ensemble des signatures est regroupé dans un tableau et c'est ce tableau à être stocké). La vérification de la signature est faite lors de l'installation du paquet pour chacun des fichiers. Pour cela `rpm` utilise une clé publique chargée dans le keyring du kernel. À noter que pour la signature `fs-verity` a besoin d'une clé et d'un certificat, les deux doivent se trouver dans le keyring.

La deuxième étape est lors de l'installation du paquet (pour laquelle il faut avoir installé le plugin `fs-verity` « `rpm-plugin-fsverity` ») : `rpm` va activer `fs-verity` sur tous les fichiers du paquet, une fois ceux-ci installés et après que l'arbre de Merkle a été calculé. À cette étape, l'arbre de Merkle n'est pas encore ajouté dans les métadonnées pour une question d'optimisation d'espace (l'arbre prenant un 127ème de la taille totale), vu que pas tous les utilisateurs en auront besoin.

Cette fonctionnalité sera disponible uniquement à partir de la version 37 de Fedora (aujourd'hui la dernière version est la 36).

### **c) SÉCURITÉ DES GESTIONNAIRES DE PAQUETS**

Lors de la protection contre des attaques informatiques, il se peut qu'on fasse recours à un pare-feu logiciel, tel que iptables. Mais iptables, et tout autre programme qu'on installe sur un serveur, est presque toujours installé en passant par un gestionnaire de paquets. Comment peut-on s'assurer que l'iptables qu'on installe, est bien celui légitime et n'inclut pas du code malveillant, comme des portes dérobées ? On repose toute notre confiance dans le fait qu'on installe un programme via apt ou dnf. Les gestionnaires de paquets représentent donc une partie fondamentale dans la chaîne de confiance d'une infrastructure informatique. La compromission d'un tel outil pourrait rendre obsolète l'entièreté de l'infrastructure, car on ne pourrait plus faire confiance à tout ce qui a été installé par le biais de ces gestionnaires.

On parle donc d'attaque à la supply-chain, que dans notre cas se réfère aux gestionnaires de paquets, mais dont sa définition s'élargit également aux fournisseurs de logiciels. Par exemple quelqu'un pourrait compromettre le code source d'iptables (ou tout autre logiciel) : dans ce cas, même si apt n'est pas compromis, on se retrouverait avec un programme potentiellement dangereux. On va toutefois se concentrer uniquement sur les gestionnaires de paquets, puisqu'effectuer une évaluation des vulnérabilités de chaque programme qu'on souhaite installer et apporter des modifications par conséquent serait un processus trop complexe et chronophage : une société se retrouverait à passer presque tout son temps à réviser le code et effectuer des tests.

Ainsi, la protection des gestionnaires de paquets peut se faire de différentes manières. Une première étape est de s'assurer que les sources d'installation soient fiables : dans le cas d'apt, les liens vers les dépôts officiels peuvent être considéré comme sûrs, à condition que ces liens pointent vers les bons dépôts. En effet, les liens sont sous forme d'adresse web (par ex. <http://deb.debian.org>), ce qui signifie que, lorsqu'apt contacte le serveur Debian, une requête DNS est effectuée et le DNS représente un autre point d'entrée pour une attaque : si le serveur DNS est compromis, le trafic des requêtes faites par apt pourrait être redirigé vers un dépôt contenant des logiciels malveillants. Pour se protéger dans ce cas, il faut avoir un niveau de protection élevé du serveur DNS. Mais comment notre serveur DNS devra dans tous les cas faire appel aux serveurs root DNS, avoir la garantie qu'on puisse faire totalement confiance à toute la chaîne, est impossible. Il va falloir faire des compromis et faire confiance aux éléments externes à notre infrastructure, mais on doit mettre en œuvre les mesures nécessaires pour faire confiance à tout ce qui est géré en interne, dans la mesure du possible.

Dans ce but, on peut renforcer la sécurité de notre gestionnaire de paquets pour garantir tout programme installé n'a pas été compromis. Même si ce projet est orienté vers des environnement Debian, les informations valables pour rpm pourraient servir de source d'inspiration pour une intégration dans Debian. Par exemple, on pourrait imaginer d'intégrer le mécanisme fs-verity directement dans apt, de cette manière tout paquet installé via apt aurait une vérification d'intégrité intégrée. Dans la pratique, implémenter cette couche dans la version officielle d'apt serait un processus assez complexe : en premier lieu parce qu'il faudrait avoir la validation des développeurs Debian à intégrer nos modifications dans le projet. Toutefois on pourrait effectuer un fork du projet GitHub apt et développer notre propre version qui inclut fs-verity. Aussi cette étape se révèle hors de portée : il faudrait pour cela devoir maintenir notre version d'apt et la mettre à jour à chaque fois que le projet original est modifié (par exemple lors de la correction de bugs ou si des failles sont découvertes). On peut alors essayer d'intégrer fs-verity dans des outils existants qui ne demandent pas leur modification totale, mais des simples adaptations ou des développements spécifiques, qui seraient moins complexes à mettre en œuvre par rapport à une intégration dans apt. Dans le chapitre suivant on verra certains de ces outils qui sont utilisés actuellement dans la configuration des infrastructures informatiques.

## 1.6. INFRASTRUCTURE AS CODE

Toute infrastructure informatique a besoin d'être maintenue : mises à jour, installation de nouveaux programmes et changements de configuration en sont des exemples. Lorsque l'infrastructure est constituée de seulement quelques machines les modifications peuvent se faire manuellement, mais, lorsque la taille augmente, le maintien de différents hôtes et serveurs devient vite un processus très chronophage. Et il n'y a pas besoin d'avoir des gros parcs informatiques, même une infrastructure avec une dizaine de machines peut se révéler trop grande pour la gestion manuelle. Dès là la nécessité de trouver un moyen de simplifier et rendre le plus efficace et efficient possible la gestion d'une infrastructure.

La solution on la trouve dans le concept connu comme Infrastructure as Code ou IaC (infrastructure définie par du code en français). Le principe est de s'approcher d'une infrastructure informatique d'une manière similaire au développement logiciel et la gérer au travers de code et d'un ensemble de processus qui automatisent et accélèrent le provisionnement de l'infrastructure. Ainsi, la modification de quelques lignes de code permet d'allouer différentes ressources computationnelles ou de mémoire à une certaine machine ou de la configurer d'une certaine

manière, par exemple sa configuration réseau ou l'installation et configuration de programmes tels qu'un serveur web, un pare-feu, un proxy, etc.

## 1.7. DÉPLOIEMENT AUTOMATISÉ D'UNE INFRASTRUCTURE

La gestion d'une infrastructure informatique ne doit donc pas devenir une tâche complexe et réservée à des personnes fortement spécialisée dans l'administration réseau. Par exemple ça pourrait consister en l'écriture d'un simple script à exécuter sur la machine cible et qui effectue une liste d'opérations prédéfinies. Il s'avère qu'ils existent déjà des nombreux outils qui ont été développés à ces fins, et sont connus en informatique comme outils d'automatisation.

Les outils les plus répandus et utilisés choisis pour cette recherche sont Ansible, Puppet, Chef et SaltStack. Ils ont été tirés de recherches web et du livre *Network Programmability & Automation*. On donnera une vision globale du produit et on se concentrera uniquement sur les fonctionnalités utiles à nos fins. La description de chaque outil a été reprise du livre susmentionné ou du site officiel de l'outil en question. Dans le chapitre implémentation on donnera une vue plus approfondie sur le fonctionnement du produit choisi pour ce projet.

### a) **ANSIBLE**

Ansible est un produit open-source créé par la société Red Hat en 2012. C'est un programme écrit en Python et pour son fonctionnement il se base sur une architecture dite *agentless*, soit qui ne nécessite pas de l'installation d'un agent (soit un service dédié) sur la machine cible de l'automatisation, mais une simple connexion SSH suffit pour exécuter des tâches. Ce mode de fonctionnement est défini *push*. Bien sûr, cela demande dans tous les cas une phase de préparation de la cible, mais configurer un accès SSH est plus simple et rapide qu'installer et configurer un agent.

Même si Ansible tourne sous Python, la liste des opérations à effectuer est écrite en format *YAML*, qui rend la compréhension des instructions assez intuitive. Le fonctionnement se base sur un ensemble de modules qui incluent des opérations prédéfinies qui permettent par exemple de redémarrer un service, installer une application, lancer des commandes spécifiques, etc. Mais on a la possibilité de développer nos propres modules et ainsi intégrer les commandes d'une application propriétaire par exemple.



## **b) PUPPET**

Puppet est un produit créé par la société homonyme en 2015. Le programme est aussi open-source et écrit en Ruby. Il se compose de 2 parties : un Puppet master qui a un rôle de gestion et le Puppet client qui s'occupe de l'hôte sur lequel il est installé. Un processus de signature de certificats permet à ces deux éléments de communiquer de manière sûre : le client envoie un certificat avec son ID au master qui le signe et le renvoie au client.

Le service qui tourne sur l'hôte vérifie à des intervalles réguliers (configurables) l'état de la machine sur lequel il est installé et l'envoie au serveur de gestion. Si le serveur remarque une différence par rapport à l'état désiré de la machine, il intervient en envoyant les instructions à exécuter au client. La liste d'instructions est appelée catalogue, qui s'obtient de la compilation de « manifestes » écrits en « Puppet code », un langage suffisamment simple et intuitif. Le manifeste peut utiliser des gabarits (templates) et des fichiers pour compléter les opérations de configuration. Ces trois composants forment les modules Puppet. Comme pour Ansible, ils existent des modules prédéfinis, mais il est possible d'en développer selon ses propres besoins.

## **c) CHEF**

Chef est une société fondée en 2008, son produit phare, Chef Infra, est open-source et écrit en Ruby. De manière similaire à Puppet, il se compose de plusieurs parties : une station de travail depuis laquelle on écrit en Ruby les opérations à effectuer (recettes), qui sont regroupés dans un classeur (cookbook) qui est envoyé vers le deuxième composant, le serveur. Ceci agit en tant que hub de communication et gestionnaire des opérations : il déploie la configuration sur les machines cibles, sur lesquelles tourne un client. Chaque client envoie à son tour l'état des machines vers le serveur et s'occupe également d'exécuter les tâches qui lui sont assignées.

La communication entre ces composants se fait au travers d'un API REST hébergé sur le serveur et se fait de manière sécurisée : une clé publique est stockée sur le serveur, tandis que la station de travail et chaque client ont une clé privée.

## **d) SALTSTACK**

Salt ou SaltStack est un projet open-source créé par Thomas S. Hatch en 2011. C'est une solution écrite en Python tandis que les instructions de configuration sont écrites en un format propre à Salt, le SLS (SaLt State), un mélange de YAML et Jinja. D'autres formats sont également

acceptés par défaut, tel que Mako et HJSON, mais il est possible d'ajouter de formats personnalisés, ce qui permet d'étendre SaltStack au-delà de ses capacités de base.

Salt fournit au même temps une architecture avec et sans agents. Dans celle avec agents, un serveur (appelé Master) communique avec les clients (appelés Minions) en utilisant le protocole ZeroMQ (très léger et rapide, utilisé dans les systèmes distribués), qui permet à un Master de gérer plus de 30'000 Minions (livre Netw Prog. & Autom.). Comme les agents peuvent être gérés par plusieurs serveurs (à leur tour gérés par un serveur principal), cela permet d'augmenter considérablement les hôtes gérables.

Dans l'architecture sans agents, une communication SSH est utilisée.

### e) COMPARAISON

Les outils vus sont tous des produits de haut niveau et extrêmement efficaces. Dans ce paragraphe on va en faire une comparaison pour pouvoir choisir celui qui se rapproche le plus à notre cas d'utilisation.

Ci-dessous un tableau qui résume les principales caractéristiques des outils vus. On y a inclus également certaines informations pas encore discutées dans les chapitres précédents.

	<b>Ansible</b>	<b>Puppet</b>	<b>Chef</b>	<b>SaltStack</b>
<b>Architecture</b>	sans serveur (agentless)	client/serveur	client/serveur	client/serveur et agentless
<b>Communication</b>	SSH	SSL	SSH, SSL	ZeroMQ
<b>Langage de programmation</b>	Python	Ruby	Ruby	Python
<b>Langage de configuration</b>	YAML	Puppet DSL	Ruby	YAML
<b>Open-source</b>	Oui	Oui	Oui	Oui
<b>Scalability</b>	Haute	Haute	Haute	Haute
<b>Compatibilité gestion OS</b>	UNIX	UNIX, Windows	UNIX, Windows	Unix, Windows
<b>Disponibilité Cloud</b>	AWS, Azure	AWS, Azure	AWS, Azure	AWS, Azure
<b>Prix du support pour 100 machines par année (en \$)<sup>1</sup></b>	10'000	11'200 – 19'900	13'700	15'000

Tableau 2: Comparaison des outils d'automation

<sup>1</sup> Source: <https://www.edureka.co/blog/chef-vs-puppet-vs-ansible-vs-saltstack/>

Maintenant qu'on a une vue globale des différents outils, on peut évaluer les avantages et inconvénients de chaque point.

En premier, une conséquence d'une architecture sans agents est que ça rend le processus de mise en œuvre beaucoup plus rapide et léger. Au niveau sécurité, un manque d'agents réduit la surface d'attaque et de vulnérabilités d'un système. Dans les deux cas, si c'est le serveur qui héberge la solution d'automation à être compromis, d'énormes dégâts peuvent en correspondre.

Les protocoles de communication utilisés sont tous sécurisés et rapides, donc adéquats pour notre cas d'application. Un petit inconvénient du SSH est qu'il faut copier les clés SSH sur l'hôte auquel on souhaite se connecter, mais cela peut être facilement résolu en copiant les clés directement lors de l'approvisionnement de l'infrastructure. C'est dans tous les cas un obstacle mineur par rapport à l'installation et configuration d'un agent.

Quant aux langages de programmation, Python et Ruby sont tous les deux des langages haut niveau et très puissants, permettant grande flexibilité du système. Ces langages sont utilisés uniquement lorsqu'on souhaite développer des fonctionnalités supplémentaires pour le système d'automation, comme par ex. des modules personnalisés. Python est beaucoup plus diffusé que Ruby, ce qui implique qu'il sera plus probable de trouver des administrateurs systèmes avec une connaissance préalable de ce langage. Aussi si on tient en compte le fait que la courbe d'apprentissage de Ruby est plus lente.

Le langage utilisé lors de la configuration est également très important. YAML est un langage assez simple à apprendre et utiliser, même s'il peut être parfois difficile à déboguer en cas de problèmes. On a vu que Ruby demande plus d'effort, toutefois il permet souvent d'aller plus en profondeur que YAML et d'écrire des commandes plus explicites et détaillées.

En outre, le fait que tous les langages soient open-source et permettent une haute extensibilité de l'infrastructure les rend idéal pour notre projet. Un code ouvert a énormément d'avantages : par exemple transparence, plus de sécurité, flexibilité et liberté de développement entre autres.

Il y a également des différences au niveau des OS cibles supportés, par ex. Ansible ne supporte pas l'automation de serveurs Windows, mais pour notre cas d'utilisation cela n'a pas d'importance puisque notre infrastructure est composée de machines Debian. Néanmoins c'est une chose à prendre en compte quand on souhaite déployer cette solution ailleurs.

La disponibilité sur le Cloud peut être une option bonne à avoir, puisque si le système est déjà intégré par les fournisseurs de services cloud tel que Amazon et Microsoft, ça accélère la phase de démarrage et permet d'effectuer des tests rapidement.

Enfin, les coûts du support sont aussi à prendre en considération, même s'ils seront souvent négligeables par rapport aux coûts du personnel, coûts qui restent dépendent de la localisation géographique des salariés.

À la lumière de ces faits, le choix de la solution de prédilection tombe sur Ansible. Ceci pour plusieurs raisons, en effet on trouve que :

- un système sans agents est plus sûr et simple
- la simplicité de configuration en YAML est fort appréciée
- l'utilisation de Python permet de simplifier la programmation de modules personnalisés
- la compatibilité uniquement avec les systèmes UNIX ne représente pas un obstacle
- les coûts de support sont inférieurs

La familiarité avec l'outil est également importante et en ayant déjà utilisé Ansible dans le passé, on trouve qu'il sera plus efficace de se baser sur un outil que l'on connaît déjà et qui dans tous les cas s'adapte parfaitement à notre cas d'utilisation.

Même si Ansible est la solution choisie, cela n'enlève rien aux autres outils, qui restent des produits puissants et très performants, mais qui s'adaptent mieux à d'autres cas d'application que le nôtre.

## 2. CHAPITRE 2 : CONCEPTION DE L'ARCHITECTURE

Dans la première partie de ce document on a eu un aperçu des différents outils et méthodes qui permettent de garantir l'intégrité et l'authenticité des fichiers gérés par les serveurs d'une infrastructure informatique. Dans cette deuxième partie on va sélectionner les outils plus adéquats et on va les rassembler pour réaliser un produit de vérification de l'intégrité d'une infrastructure.

Certains éléments d'une machine sont plus sensibles que d'autres, par exemple si notre éditeur de texte était compromis, cela serait moins grave qu'une faille dans un gestionnaire de mots de passes. Ainsi, on n'a pas besoin de protéger l'entièreté d'un système de fichiers, mais il suffit d'assurer l'intégrité des parties fondamentales à son fonctionnement et de tout autre élément qu'on souhaite protéger. Ce prérequis nous porte à préférer FS-Verity à dm-verity comme outil de vérification d'intégrité, puisqu'il permet d'agir au niveau des fichiers individuels. Il est également le choix de prédilection par rapport à IMA, malgré ce dernier aurait également pu être utilisé pour atteindre le même objectif. En premier, c'est un outil moins complexe et élaboré, qui offre toutes les fonctionnalités dont on a besoin et des méthodes simples pour les implémenter. D'un autre côté, certains choix de développement font qu'il peut être considéré plus sûr par rapport à IMA, par exemple le fait que les métadonnées fs-verity ne sont pas stockées en tant qu'attributs étendus du fichier, comme c'est le cas pour IMA, mais sont écrits directement dans les blocs du système de fichiers, rendant plus difficile leur compromission.

FS-Verity fournira donc la protection de l'intégrité et l'authenticité des fichiers. Comme on souhaite le déployer dans une infrastructure définie par du code, il doit être intégré aux outils d'automation existants ou en tous cas doit pouvoir être exploité par ces derniers. Le choix du moteur d'automation est tombé sur Ansible, selon les critères listés dans le chapitre précédent. Or, Ansible (ainsi que tous les autres outils) n'inclut pas des fonctionnalités spécifiques qui permettent d'utiliser fs-verity dans la manière dont il a été conçu. Cela nous porte au besoin de développer un module Ansible personnalisé, qui pourra être utilisé lors de la gestion d'un parc de machines. Le développement de ce module est expliqué dans le chapitre correspondant dans la section implémentation. Tandis que dans le chapitre suivant on détaillera certains choix conceptuels du projet qui nous seront utiles à mieux comprendre l'architecture choisie.

## 2.1. THÉORIE DE FONCTIONNEMENT

Après avoir défini les principaux outils que l'on va utiliser pour notre projet, il est important de spécifier la manière dont ces outils seront utilisés. En particulier, clarifier leur rôle, leurs interactions et les critères sur lesquels nos choix se basent.

Pour pouvoir garantir l'intégrité d'un ou plusieurs fichiers on va se baser sur les mécanismes de vérification de fs-verity. L'outil peut être activé sur n'importe quel fichier, de n'importe quelle extension et dans n'importe quel format d'encodage. Par conception, il ne peut pas être activé sur des dossiers. Aussi, il ne devrait pas être activé sur des fichiers qui sont censés être modifiés régulièrement (par ex. des fichiers de logs), car fs-verity rend le fichier en lecture seule. Le seul moyen de désactiver fs-verity sur un fichier est de remplacer ce dernier.

Une fonctionnalité intéressante mise à disposition par fs-verity est la possibilité d'inclure une signature lors de l'activation du fichier fs-verity et de la rendre obligatoire. Elle se fait avec une clé privée et un certificat. La clé privée doit absolument être protégée : sa divulgation rendrait obsolète la protection offerte par fs-verity, puisque tout le monde pourra activer fs-verity sur des fichiers. Le certificat doit être ajouté au keyring fs-verity de la machine sur laquelle le fichier se trouve, cela pour que le kernel puisse authentifier la signature contre le certificat stocké dans le keyring. Un certificat non valable ou révoqué empêcherait l'ouverture d'un fichier sur lequel on a activé fs-verity au préalable.

Un point important qui concerne la signature est qu'elle s'effectue sur le digest fs-verity d'un fichier. Le digest correspond au hash du fs-verity descriptor, soit la structure de données qui contient les métadonnées fs-verity. Donc, pour activer fs-verity sur un fichier on a besoin uniquement d'un hash, pas de tout le contenu du fichier.

Ces trois derniers paragraphes nous portent à la réflexion suivante :

- On peut renforcer la sécurité en rendant la signature obligatoire
- La signature se fait sur le digest fs-verity d'un fichier
- La signature a besoin d'une clé privée qui ne doit pas être divulguée

Par conséquent, pour éviter de divulguer la clé privée, il va falloir générer la signature sur un serveur retenu sûr et ensuite la transférer sur la machine sur laquelle se trouve le fichier à protéger.

Ce procédé est techniquement très sûr et efficace pour les raisons suivantes :

- La clé privée ne sort jamais du serveur sur lequel a été générée et sur lequel on signe.

- Le fait qu'on ait besoin du digest du fichier et non pas du fichier entier, permet d'effectuer cette opération sur un très grand nombre de fichiers de n'importe quelle taille, puisque le digest ne fait que quelques octets de taille (32 octets dans le cas de SHA256), rendant très rapide et efficace le transfert du digest au travers du réseau.
- La signature (qui est un fichier contenant le digest chiffré avec la clé privée) générée de cette manière peut être transférée sans problèmes, puisque si quelqu'un la modifiait sans autorisation, on ne pourrait pas l'utiliser pour activer fs-verity sur le fichier à protéger.

Ce mode de fonctionnement, qu'on appelle hybride, s'appuie sur les fonctionnalités du kernel, mais a besoin d'une intervention faite à partir de l'espace utilisateur pour la génération de la signature. Le kernel effectue ensuite le processus de vérification et prends les mesures nécessaires en cas d'incohérence (le fichier ne s'ouvre pas s'il est corrompu ou sans signature). Par contraste, on peut imaginer également une solution entièrement basée sur l'espace utilisateur, où on ne stocke pas la signature le long des métadonnées fs-verity, mais dans un endroit de notre choix (ça peut être un fichier à part ou à la fin du fichier à signer, comme pour c'est le cas pour les fichiers Release et InRelease de apt). Ça implique aussi que la vérification de la validité de la signature est laissée à l'espace utilisateur.

Les étapes de récupération du digest, de génération de la signature et d'activation de fs-verity peuvent et doivent être gérée par Ansible. Plus précisément, un module spécialement développé à ce but pourra inclure toutes ces fonctionnalités et automatiser ces procédés en permettant de garder la sécurité de chaque élément. Le module pourra également inclure des étapes intermédiaires et d'autres fonctionnalités utiles nécessaires à la mise en place de notre solution de sécurité.

Pour exploiter les différentes fonctions mises à disposition par fs-verity, le module Ansible pourra s'appuyer sur les outils fs-verity fourni par Eric Biggers. FS-Verity a été implémenté dans le kernel et on a accès à ses fonctionnalités par le biais d'appels ioctl. Ces outils fournissent une interface utilisateur pour une gestion simplifiée de ces appels et incluent notamment le calcul du digest du fichier, la génération de la signature et l'activation de fs-verity sur un fichier. Toutefois ils ne sont pas complètement adaptés à notre cas d'utilisation. En effet, la commande qui permet de générer la signature prend en paramètre un fichier, pas le digest. Pour notre projet, il va falloir modifier le code source des outils fs-verity pour pouvoir générer la signature à partir d'un digest.

Enfin, on souhaite avoir une méthode qui permette de vérifier si fs-verity est bien activé sur un fichier. Pour cela, un script devra être réalisé et qui pourra tourner en tant que service sur la machine sur laquelle se trouvent les fichiers à protéger ou alors à partir d'un serveur qui se

connecte aux différents postes à vérifier. Le processus de vérification est en soit très simple : il suffit de s'assurer que fs-verity est activé sur le fichier et que ce dernier peut être ouvert. Un exemple d'une solution simple et élégante qui permet d'effectuer ces deux étapes est la commande « *lsattr* », qui permet de vérifier si fs-verity est activé sur le fichier et fonctionne seulement si le fichier peut être ouvert. FS-Verity empêchera toute ouverture d'un fichier qui ne réponds pas aux critères de sécurité. Une autre solution est d'utiliser les outils fs-verity, qui avec la commande « *measure* » permet d'effectuer l'opération de vérification.

```
iti@ansible-test:~$ lsattr example.txt
-----e-----V- example.txt
```

*Illustration 22: Exemple de commande lsattr quand fs-verity est activé*

```
iti@ansible-test:~$ lsattr example.txt
[ 32.004686] fs-verity (vda1, inode 133960): File's signing cert isn't in the fs-verity keyring
lsattr: Required key not available While reading flags on example.txt
```

*Illustration 23: Exemple de commande lsattr quand il manque le certificat dans le keyring*

Dans les prochains chapitres on va voir comment les différents composants de notre solution interagissent ensemble et avec l'environnement extérieur. On s'aidera avec différents schémas et diagrammes d'architecture et on réutilisera partiellement les éléments produits lors du travail de semestre, adaptés aux nouvelles décisions prises pour ce projet.

## 2.2. MODÈLE C4

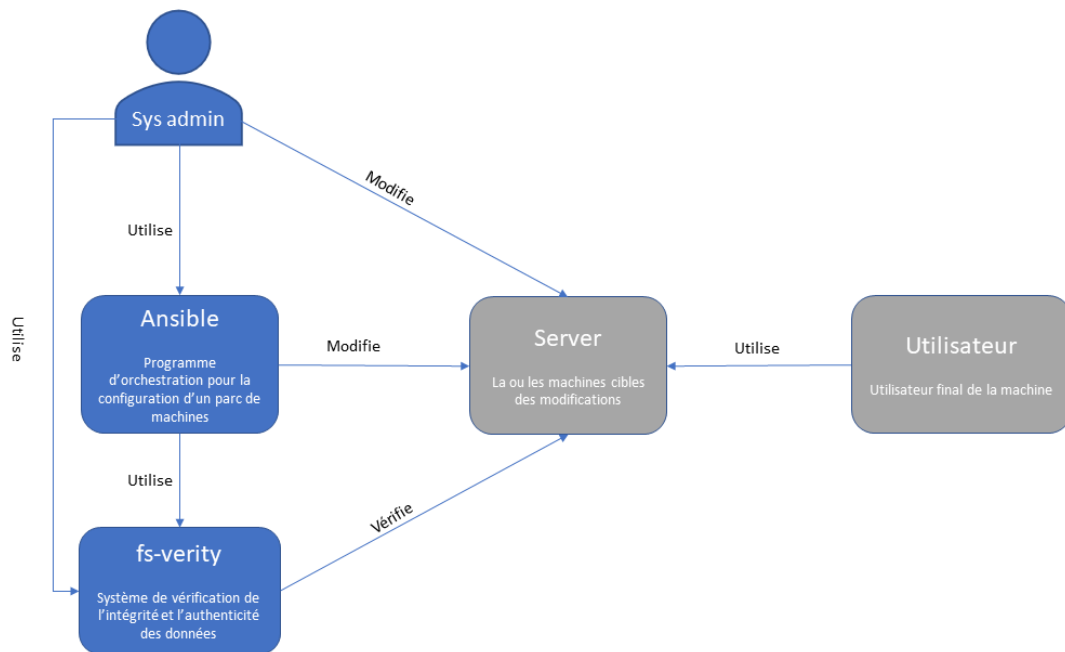
Le premier modèle de représentation d'architecture qu'on va discuter est le modèle C4, appelé de cette manière parce qu'il se compose de 4 niveaux de détails (qui commencent avec la lettre C) : Contexte, Conteneurs, Composants, Code.

Dans le cadre de ce projet, on s'étendra jusqu'au 3ème niveau, les composants. En effet, la schématisation du code est plutôt utile lorsqu'on développe des logiciels complexes. Notre projet ne contient pas suffisamment de code pour justifier une représentation du 4ème niveau. Toutefois, le code sera largement discuté, commenté et expliqué en partie dans la section implémentation de ce document et en partie dans la documentation présente sur le git du projet.



### a) NIVEAU 1 : CONTEXTE

Dans le premier niveau du modèle C4 on représente les parties fondamentales qui composent le projet ainsi que les liaisons entre eux. L'illustration ci-dessous montre les éléments du système, qui sont représentés par des rectangles, qui seront bleus pour indiquer les parties du système qu'on a produit et gris pour les parties existantes. Les relations entre les éléments sont représentées par des flèches. Le schéma qui suit a été repris du travail de semestre et adapté à notre projet.



*Illustration 24: Niveau 1 du Modèle C4*

Dans cette première représentation, on trouve deux parties principales qui constituent les éléments de notre conception : la partie Ansible et celle fs-verity, ainsi que l'administrateur système qui les utilise. En particulier, ce dernier va se servir d'Ansible pour gérer l'infrastructure informatique et Ansible s'appuiera sur les outils fs-verity pour habilitier ce dernier sur les cibles des modifications. En plus, l'administrateur système pourra se servir des outils fs-verity pour effectuer des opérations manuellement, le rendant ainsi indépendant d'Ansible pour les modifications. Enfin, un utilisateur du ou des serveurs en question (l'utilisateur peut être aussi bien une personne que d'autres services, sites web, bases de données, etc.) pourra se servir des applications présentes sur le serveur de manière totalement transparente et sûre, sans besoin qu'il y ait des interactions supplémentaires. En effet, fs-verity est complètement transparent à l'espace utilisateur : les fichiers sur lesquels on l'active peuvent être ouverts et lu normalement (mais pas

modifiés), puisque la vérification d'intégrité est faite par le module fs-verity dans le kernel du serveur.

## b) NIVEAU 2 : CONTENEURS

Le niveau 2 du modèle reprend le niveau 1 et montre plus en détail le contexte de notre projet. Les blocs Ansible et fs-verity sont décomposés en montrant les différents modules dont ils sont constitués et leurs interactions. La base de cette architecture a aussi été reprise du travail de semestre et modifiée par conséquent.

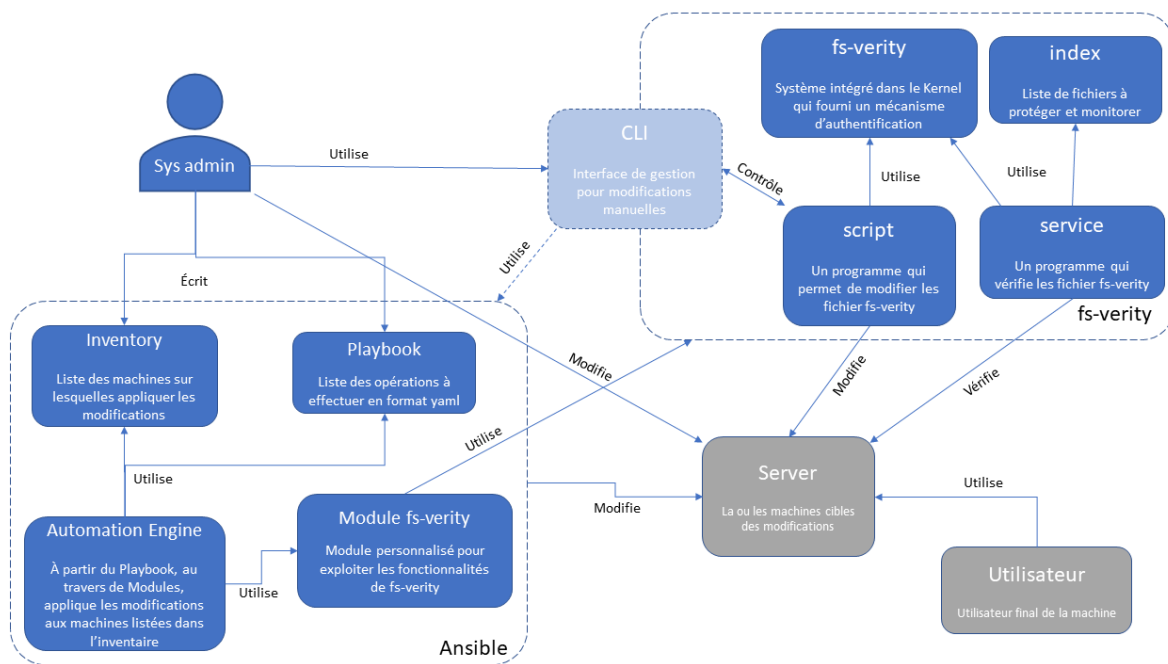


Illustration 25: Niveau 2 du Modèle C4

Dans ce deuxième modèle, on voit plus clairement les éléments qui composent les deux éléments à la base de notre projet. En particulier, Ansible est constitué des éléments qui suivent :

- Un inventaire, qui contient la liste des machines sur lesquelles on souhaite mettre en place fs-verity.
- Un playbook, qui est la liste des opérations à effectuer sur les machines indiquées dans l'inventaire.
- Un module fs-verity, qui a été écrit par nous-même et qui utilise les fonctionnalités de fs-verity et les rend disponibles au playbook
- Le moteur d'automation d'Ansible, qui interprète les données contenues dans l'inventaire et le playbook et se sert du module fs-verity pour appliquer les changements souhaités sur les cibles spécifiées.

L'inventaire est écrit par l'administrateur système, tandis que le module fs-verity est fourni par nous. On fournira également une proposition de playbook qui utilise notre module, mais il peut être adapté selon les besoins de l'administrateur, alors que le module fourni déjà tout le nécessaire et n'a en principe pas besoin d'être modifié. Par exemple, dans notre playbook on efface le fichier de signature après avoir activé fs-verity, mais l'ingénieur système pourrait vouloir le stocker ailleurs à la place de l'effacer. On lui laisse donc libre choix quant à l'utilisation du playbook.

De l'autre côté on a fs-verity, qui se compose également de plusieurs parties :

- Le module implémenté dans le kernel, qui effectue les opérations de vérification des fichiers avant leur ouverture et sur lequel on s'appuie pour l'activation de fs-verity.
- Un service qui sert à effectuer la vérification constante de la bonne activation de fs-verity sur les machines à protéger. Pour son fonctionnement, on lui indique une liste de fichiers à protéger et le processus de vérification se base sur la couche fs-verity du kernel.
- Un index, contenant une liste avec les chemins des différents fichiers à protéger.
- Une méthode sous forme de script qui permet la désactivation de fs-verity sur un fichier pour pouvoir le modifier. Comme fs-verity n'inclut pas d'option pour le désactiver, le seul moyen de le faire est de remplacer le fichier qu'on souhaite modifier (l'effacement d'un fichier étant consenti). La désactivation comporte donc une simple opération de remplacement, qui peut être fait en plusieurs manières différentes. Ici on fournira une méthode, mais l'administrateur système pourra sans autre se servir de n'importe quel autre moyen il retient le plus adéquat.
- Une CLI, qui fournit une interface utilisateur pour le script de désactivation de fs-verity. Cette interface peut prendre la forme d'une simple commande, ne doit pas être considérée comme une vraie et propre interface graphique comme ça pourrait l'être une page web de gestion d'un produit par exemple.

### **c) NIVEAU 3 : COMPOSANTS**

Dans le troisième et dernier niveau de notre architecture on va rentrer dans les détails des conteneurs Ansible et fs-verity, à l'aide de deux schémas séparés.

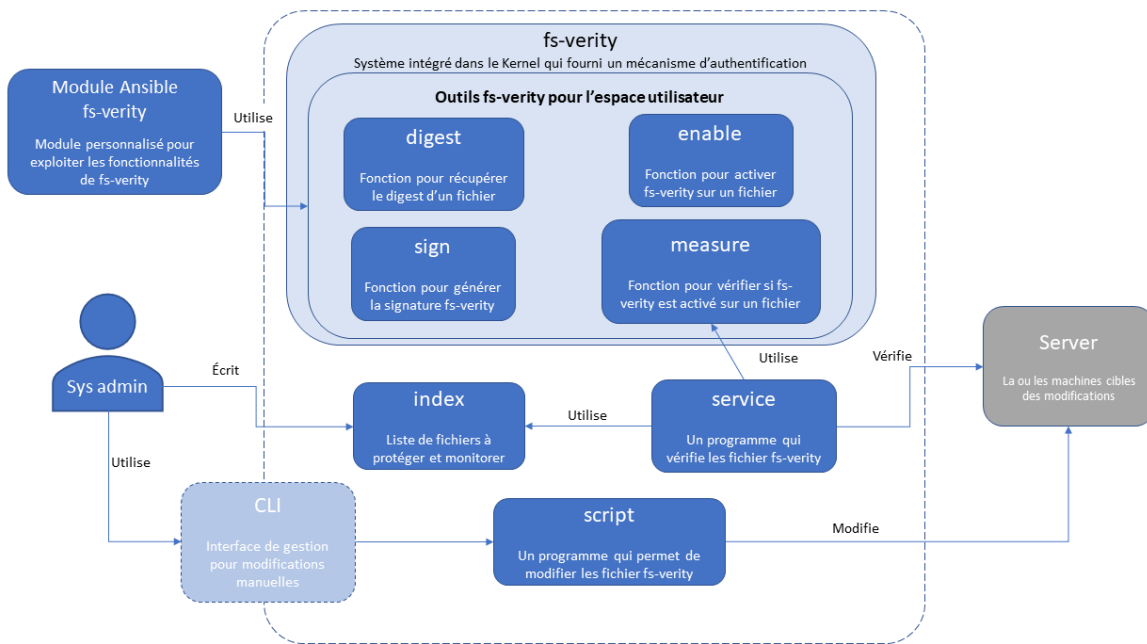


Illustration 26: Niveau 3 du Modèle C4 – fs-verity

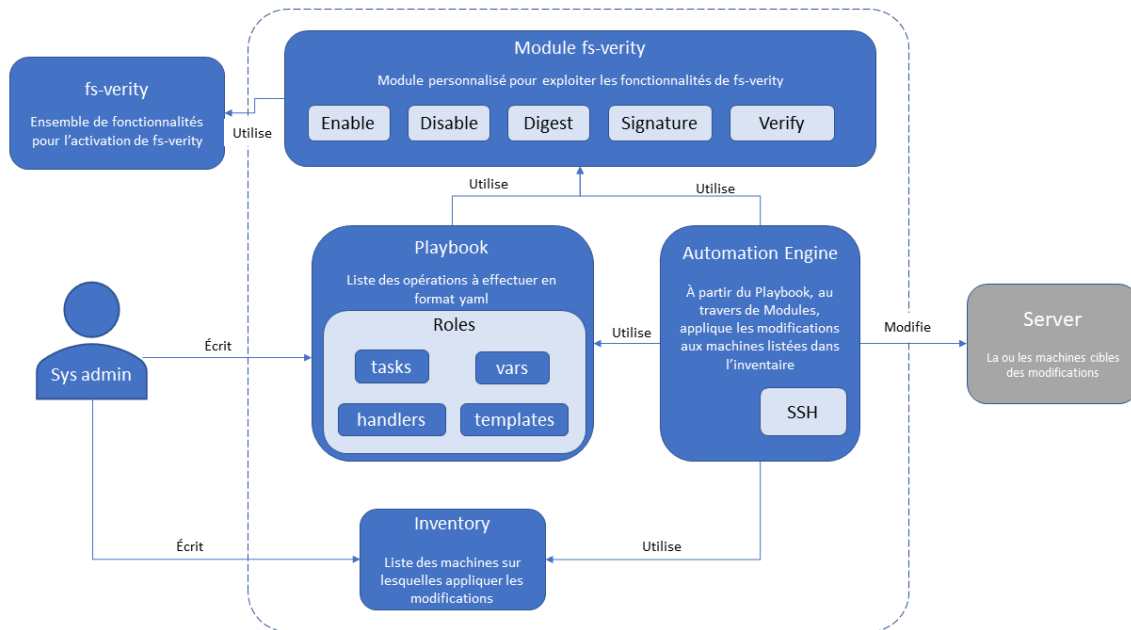


Illustration 27: Niveau 3 du Modèle C4 – Ansible

Le premier de ces deux schémas nous montre la structure du conteneur fs-verity. En particulier, on voit le détail de différents composants qui sont utilisés par fs-verity pour les opérations de protection d'intégrité. Les outils disponibles dans l'espace utilisateurs font appel directement aux fonctionnalités du kernel et des fonctions qui nous fournissent on va en utiliser quatre :

- Digest, qui permet de récupérer le digest d'un fichier, généré à partir selon un format spécifié par fs-verity ;
- Sign, qui va générer le fichier de signature et que l'on va modifier pour l'adapter à nos besoins ;
- Enable, pour activer fs-verity sur un fichier ;
- Measure, pour vérifier si fs-verity est activé.

Le module Ansible s'appuiera aussi sur ces fonctions pour interagir avec les fichiers à traiter. Par la suite, nous avons les mêmes blocs que le schéma des conteneurs :

- Un index qui contient la liste des chemins des fichiers à vérifier ;
- Le service de vérification, qui monitore la liste de l'index ;
- Le script de désactivation de fs-verity ;
- La GUI.

Ensuite, on a le détail du conteneur Ansible. Ici, on a détaillé le contenu du module qu'on a développé et qui contient notamment les fonctions suivantes :

- Enable : pour activer fs-verity sur un fichier ;
- Disable : pour désactiver fs-verity sur un fichier ;
- Digest : pour récupérer le digest d'un fichier ;
- Signature : pour générer le fichier de signature ;
- Verify : pour vérifier si fs-verity est bien activé sur le fichier d'intérêt.

Ces fonctions font appel aux outils fs-verity disponibles dans l'espace utilisateurs.

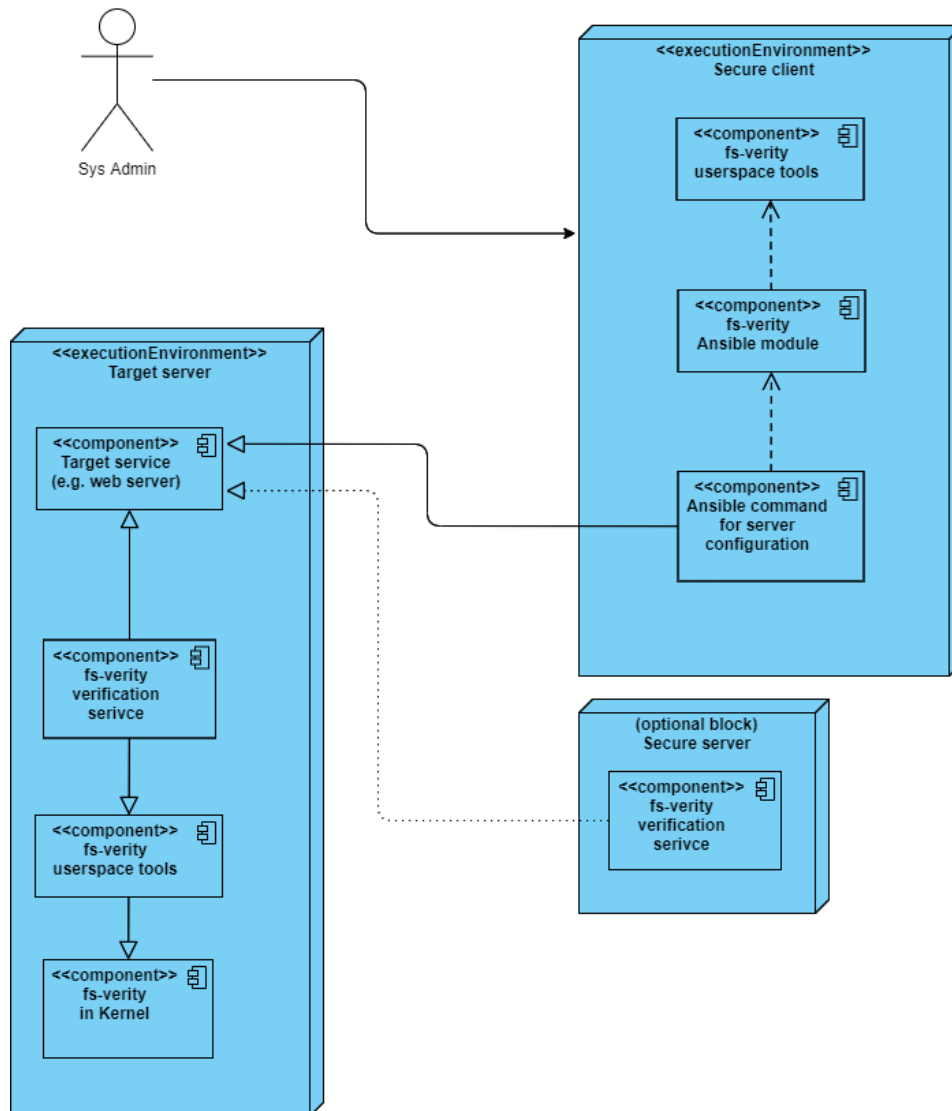
Aussi, dans notre schématisation on retrouve le playbook qui fait appel à des rôles qui utilisent des tâches (tasks), variables (vars), templates (des modèles de fichiers) et handlers (des tâches qui sont exécutées à la suite d'événements déterminés).

On retrouve également l'inventaire qui, étant un fichier, n'est pas composé de plusieurs éléments.

Enfin, le moteur d'automation Ansible, prend en compte tous les autres blocs pour apporter les modifications demandées sur le serveur au travers d'une connexion SSH qui aura été configurée au préalable.

## 2.3. DIAGRAMME DE DÉPLOIEMENT

Un diagramme de déploiement sert à représenter comment sont réparties physiquement les composants du système qu'on souhaite décrire et les relations entre eux. Ce diagramme aussi a été partiellement repris de celui présenté lors du travail de semestre.



*Illustration 28: Diagramme de déploiement*

Le schéma est divisé en deux blocs principaux : la machine hôte (secure client) à partir de laquelle on effectuera les opérations sensibles et le client (Target server) qu'on souhaite protéger. Dans l'architecture proposée, l'administrateur système effectue les opérations à partir d'un poste client sur lequel tourne Ansible et où sont installés les outils fs-verity. Un module Ansible spécialement développé utilisera ces outils pour la génération de la signature et l'activation de fs-verity sur les fichiers du serveur cible. Ansible appliquera les modifications demandées, dont

l'activation de fs-verity sur certains fichiers, sur la machine à protéger. Ici, un service de vérification d'intégrité basé sur fs-verity tournera en continu et il sera basé sur les outils fs-verity, qui à leur tour feront appel aux ioctl pour interagir avec les fonctionnalités fs-verity du kernel.

On remarque ici un point intéressant, qui n'était pas évoqué jusqu'à présent : l'activation de fs-verity dans le kernel a besoin d'être faite uniquement sur le serveur à protéger. En effet, les outils fs-verity qu'Ansible utilisera fournissent des fonctions qui n'interagissent pas avec le kernel.

Enfin, on trouve dans le schéma un bloc supplémentaire : un serveur à part sur lequel tourne le service de vérification qui va monitorer l'intégrité d'une machine autre que celle où tourne le service. Ce bloc est optionnel et il est une proposition d'amélioration de la sécurité du système. En effet, si un attaquant compromettrait le service de vérification, on n'aurait pas d'autre moyen de nous assurer que fs-verity est bien activé sur un fichier outre que d'effectuer la vérification manuellement.

## **2.4. DIAGRAMME DE SÉQUENCE**

Le dernier diagramme que l'on présente est celui de séquence, qui montre l'ordre temporel des interactions entre les différentes parties du système.

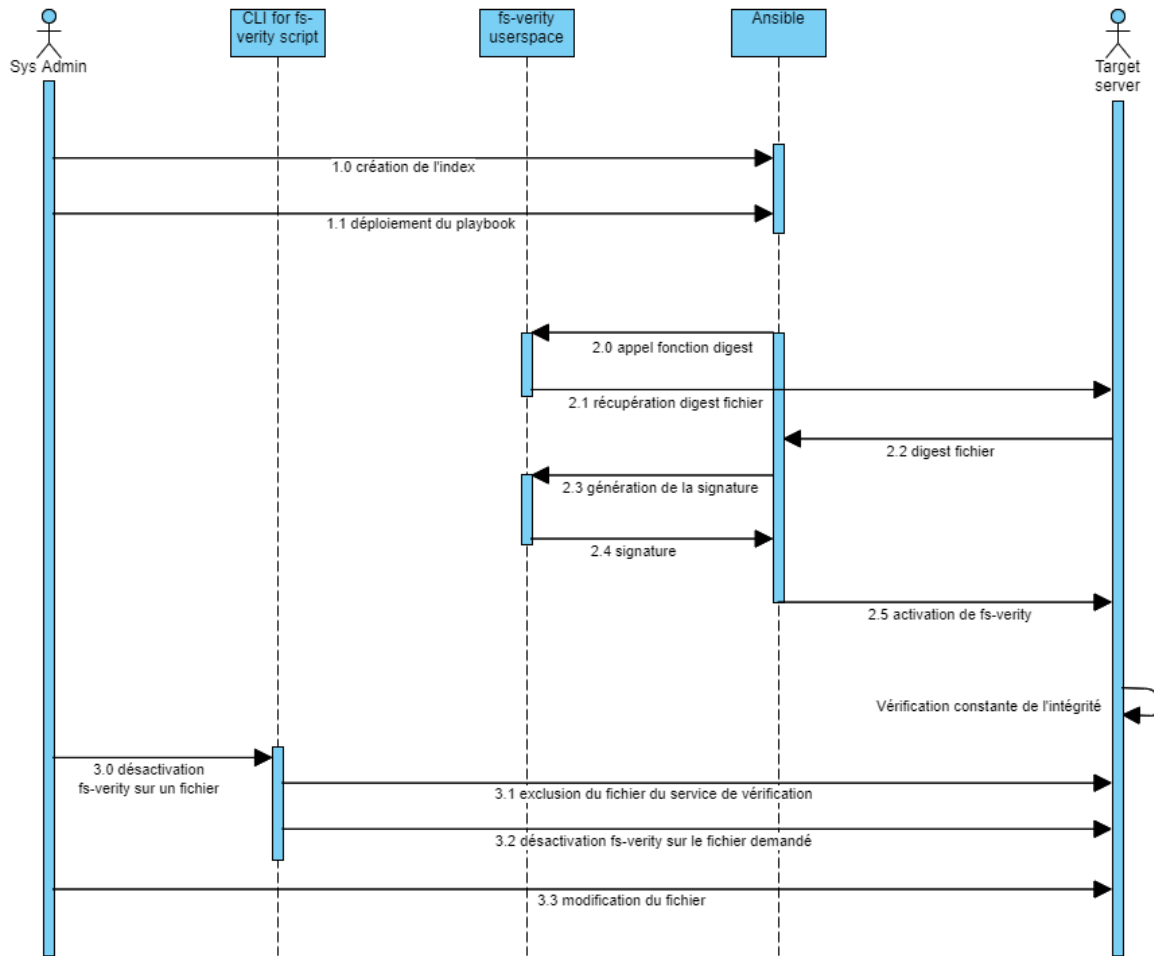


Illustration 29: Diagramme de séquence

Le diagramme est divisé en cinq colonnes : la première et dernière représentent les acteurs qui interagissent avec le système, les trois autres colonnes représentent les éléments du système et les lignes pointillées verticales le passage du temps. La longueur des barres bleues verticales n'est pas une indication de la quantité de temps qui s'est écoulé, mais elles servent plutôt à donner un ordre aux différentes opérations.

Dans un premier temps on a la création de la liste de fichiers à protéger et qui est fournie en paramètre à Ansible (étape 1.0), ensuite l'administrateur système peut déployer le playbook Ansible (1.1). La première chose qu'Ansible (ou mieux, son module) fait lors de ce déploiement est d'appeler la fonction de récupération du digest d'un fichier (2.0). Il est récupéré du serveur (2.1) et envoyé vers le moteur d'Ansible (2.2). On peut ainsi générer la signature (2.3 et 2.4) et l'envoyer vers le serveur pour activer fs-verity en signant le fichier (2.5). C'est uniquement à partir de ce moment que le service de vérification d'intégrité qui tourne sur le serveur peut effectuer le monitoring constant de l'intégrité. Dans le schéma cela est représenté par une boucle sur la barre



temporelle du serveur. Enfin, quand l'administrateur système souhaite effectuer des modifications manuelles, il devra d'abord désactiver fs-verity (3.0), mais pour cela notre CLI (ou mieux, le script qui est derrière) devra d'abord exclure le fichier à modifier de la liste de fichiers à monitorer par le service (3.1), peine de générer une fausse alerte. Seulement après la CLI pourra désactiver fs-verity sur le fichier (3.2). L'administrateur sera alors libre de modifier le fichier (3.3). Pour la réactivation de fs-verity sur le fichier, on peut repasser par les étapes 1.0 et 1.1.

### 3. CHAPITRE 3 : IMPLÉMENTATION

Le dernier chapitre de ce document illustre les étapes prises pour la mise en pratique de la solution discutée dans la section conception. Dans un premier temps on va discuter de ce qui a été produit, notamment le code source des différentes parties. Ensuite, on montrera comment utiliser tous les outils ensemble pour pouvoir garantir l'intégrité et l'authenticité d'une infrastructure informatique.

#### 3.1. OUTILS FS-VERITY

FS-Verity est une fonctionnalité intégrée dans le kernel Linux dont l'espace utilisateur peut y avoir accès au travers d'appels ioctl. Le développeur de fs-verity, Eric Biggers, nous mets à disposition des outils que l'on peut utiliser pour interagir avec le kernel sans besoin d'écrire nos propres fonctions à partir de zéro. Ces outils peuvent être installés via apt ou compilés manuellement. En effet, lors de l'installation via apt, il y a une fonction qui n'est pas incluse dans celles disponibles : `dump_metadata`, pour récupérer les métadonnées fs-verity. Pour rappel, cette fonction ne peut être utilisée qu'à partir de la version 5.12 du kernel Linux et la version par défaut de Debian 11 est 5.10. Néanmoins, cela ne représente pas un problème pour notre projet, puisqu'on ne se servira pas de cette fonction.

Les autres fonctions qu'on à disposition soit qu'on passe par apt soit qu'on compile manuellement, sont résumées ci-dessous :

```
iti@ansible-test:~$ fsverity --help
Usage:
  Compute the fs-verity digest of the given file(s), for offline signing:
    fsverity digest FILE...
        [--hash-alg=HASH_ALG] [--block-size=BLOCK_SIZE] [--salt=SALT]
        [--compact] [--for-builtin-sig]

  Enable fs-verity on a file:
    fsverity enable FILE
        [--hash-alg=HASH_ALG] [--block-size=BLOCK_SIZE] [--salt=SALT]
        [--signature=SIGFILE]

  Display the fs-verity digest of the given verity file(s):
    fsverity measure FILE...

  Sign a file for fs-verity:
    fsverity sign FILE OUT_SIGFILE --key=KEYFILE
        [--hash-alg=HASH_ALG] [--block-size=BLOCK_SIZE] [--salt=SALT]
        [--cert=CERTFILE]

  Standard options:
    fsverity --help
    fsverity --version

Available hash algorithms: sha256 sha512
```

*Illustration 30: Page d'aide des outils fs-verity*

On retrouve quatre fonctions. La première est `fsverity digest FILE`, qui retourne le digest du fichier qu'on lui donne en paramètre. Plusieurs options permettent de définir le format de sortie ainsi que l'algorithme d'hachage et la taille du block. Pour notre cas d'utilisation, on va utiliser l'option `--compact` qui retourne uniquement le hash en format hexadécimal. Ci-dessous des exemples des différents retours de la commande :

```
iti@ansible-test:~$
iti@ansible-test:~$ fsverity digest hello.txt
sha256:3d248ca542a24fc62d1c43b916eae5016878e2533c88238480b26128a1f1af95 hello.txt
iti@ansible-test:~$
iti@ansible-test:~$ fsverity digest hello.txt --compact
3d248ca542a24fc62d1c43b916eae5016878e2533c88238480b26128a1f1af95
iti@ansible-test:~$
iti@ansible-test:~$ fsverity digest hello.txt --compact --for-builtin-sig
4653566572697479010020003d248ca542a24fc62d1c43b916eae5016878e2533c88238480b26128a1f1af95
iti@ansible-test:~$
```

*Illustration 31: Retours de la commande fsverity digest*

L'option `--for-builtin-sig` retourne le `fs-verity_formatted_digest` en format hexadécimal pour pouvoir l'utiliser pour générer une signature avec une méthode outre que celle fournie avec les outils `fs-verity`. En effet, si on interprète les premiers 8 octets du résultat, on peut voir qu'ils correspondent bien au mot « FSVerity » qui se trouve dans la structure citée.

```
iti@ansible-test:~$ echo 4653566572697479 | xxd -r -p && echo ""
FSVerity
iti@ansible-test:~$
```

*Illustration 32: Partie initiale du digest pour la signature*

La deuxième fonction fournie par les outils est `fsverity enable FILE`, pour activer `fs-verity` sur un fichier. Outre que la possibilité de modifier la fonction d'hachage, la taille du bloc et ajouter un sel, cette fonction nous permet d'ajouter un fichier contenant la signature `fs-verity`, qui sera ajoutée aux métadonnées. Si on rend la signature obligatoire, tout tentative d'activer `fs-verity` sans un fichier de signature résultera en une erreur :

```
iti@ansible-test:~$ fsverity enable file.txt
[93414.400327] fs-verity (vda1, inode 139291): require_signatures=1, rejecting unsigned file!
ERROR: FS_IOC_ENABLE_VERITY failed on 'file.txt': Operation not permitted
iti@ansible-test:~$
```

*Illustration 33: Essai d'activation de fs-verity*

La troisième fonction est `fsverity measure FILE`, qui elle aussi retourne le digest du fichier, mais à différence de la commande `fsverity digest` qui calcule le digest à chaque fois, `fsverity measure` récupère le digest qui est stocké dans les métadonnées `fs-verity`. Cela ne fonctionne donc que si `fs-verity` est activé sur le fichier et résultera en une erreur autrement :

```
iti@ansible-test:~$ fsverity measure file.txt
ERROR: FS_IOC_MEASURE_VERITY failed on 'file.txt': No data available
iti@ansible-test:~$
```

*Illustration 34: Essai de vérification de fs-verity*

Enfin, la dernière fonction est celle qui nous permet de générer le fichier de signature : `fsverity sign FILE OUT_SIGFILE --key=KEYFILE --cert=CERTFILE`. Pour la génération du fichier de signature, la fonction nécessite d'une clé privée et d'un certificat. Vu que la machine sur laquelle on souhaite activer fs-verity est potentiellement vulnérable, il est indispensable d'éviter d'y stocker la clé privée. Cela nous porte au besoin de générer le fichier de signature sur une machine à laquelle on fait confiance. Pour une question d'efficacité, on aimerait éviter de copier le fichier entre machines pour y effectuer la commande `fsverity sign` de génération de la signature (qui demande un fichier en paramètre) : si on a un très grand nombre de machines et de fichiers, cela aurait un poids non négligeable sur l'utilisation de la bande passante du réseau, surtout dans le cas où les fichiers seraient de grande taille. Aussi, ils occuperaient beaucoup d'espace disque sur la machine qui effectue la génération de la signature, rendant le tout beaucoup plus lourd et inefficace. On a ainsi décidé de modifier le script de génération de la signature pour qu'il prenne en paramètre directement le digest du fichier et non pas le fichier en entier. Dans le chapitre qui suit on va montrer comment cela a été réalisé.

#### **a) MODIFICATION PROGRAMME DE SIGNATURE**

En analysant le code source des outils fs-verity, on a retrouvé les parties du code qui effectuent l'opération de génération du fichier de signature. Ces fonctions sont rassemblées dans un fichier nommé `cmd_sign.c` qui est dans le dossier `programs` de l'arborescence des outils (c.f. Annexes). Dans ce programme, ce sont cinq les fonctions utilisées pour effectuer la signature :

```

115
116     if (!open_file(&file, argv[0], O_RDONLY, 0))
117         goto out_err;
118
119     if (!get_file_size(&file, &tree_params.file_size))
120         goto out_err;
121
122     if (libfsverity_compute_digest(&file, read_callback,
123                                   &tree_params, &digest) != 0) {
124         error_msg("failed to compute digest");
125         goto out_err;
126     }
127
128     if (libfsverity_sign_digest(digest, &sig_params,
129                                &sig, &sig_size) != 0) {
130         error_msg("failed to sign digest");
131         goto out_err;
132     }
133
134     if (!write_signature(argv[1], sig, sig_size))
135         goto out_err;
136

```

*Illustration 35: Code d'origine du programme cmd\_script.c*

- Open\_file, pour ouvrir le fichier ;
- Get\_file\_size, pour récupérer la taille du fichier (c'est un paramètre qui fait parties des métadonnées fs-verity) ;
- Libfsverity\_compute\_digest, pour calculer le digest du fichier passé en paramètre ;
- Libfsverity\_sign\_digest, qui génère la signature à partir du digest ;
- Write\_signature, qui écrit sur le disque le fichier de signature.

Le résultat des modifications est affiché ci-dessous (les parties non fondamentales du programme ont été omises dans cette capture d'écran) :

```

44
45  /* Sign a file for fs-verity by computing its digest, then signing it. */
46  int fsverity_cmd_sign(const struct fsverity_command *cmd,
47                        int argc, char *argv[])
48  {
49      //struct filedes file = { .fd = -1 };
50      //struct libfsverity_merkle_tree_params tree_params = { .version = 1 };
51      struct libfsverity_signature_params sig_params = {};
52      struct libfsverity_digest *digest = NULL;
53      char digest_hex[FS_VERITY_MAX_DIGEST_SIZE * 2 + 1];
54      unsigned char *digest_bin;
55      char *digest_hexa;
56      u8 *sig = NULL;
57      size_t sig_size;
58      int status;
59      int c;
60

```

*Illustration 36: Nouvelles variables du programme cmd\_script.c modifié*

```

120
121 > // if (!open_file(&file, argv[0], O_RDONLY, 0))...
123
124 > // if (!get_file_size(&file, &tree_params.file_size))...
126
127 > // if (libfsverity_compute_digest(&file, read_callback, ...
132
133 // Allocating memory
134 digest_bin = xzalloc(sizeof(*digest_bin) * 32);
135 digest = xzalloc(sizeof(*digest) * 64);
136 digest_hexa = xzalloc(sizeof(*digest_hexa) * 64);
137
138 // Converting to binary
139 strncpy(digest_hexa, argv[0], SHA256_DIGEST_SIZE*2);
140 hex2bin((const char *)digest_hexa, digest_bin, strlen(digest_hexa) / 2);
141
142 // Initialising Digest
143 digest->digest_algorithm = FS_VERITY_HASH_ALG_SHA256;
144 digest->digest_size = SHA256_DIGEST_SIZE;
145 memcpy(digest->digest, digest_bin, digest->digest_size);
146
147 // Freeing memory
148 free(digest_bin);
149 free(digest_hexa);
150
151 > // Test output ...
155
156
157 if (libfsverity_sign_digest(digest, &sig_params,
158 | | | &sig, &sig_size) != 0) {
159 | | | error_msg("failed to sign digest");
160 | | | goto out_err;
161 | | }
162
163 if (!write_signature(argv[1], sig, sig_size))
164 | | goto out_err;
165

```

*Illustration 37: Nouvelles fonctions du programme cmd\_script.c modifié*

Pour modifier le programme, on a procédé de la façon suivante. Tout d’abord, on a mis en commentaires toutes les parties qui concernaient la gestion du fichier, notamment les variables (lignes 49 et 50) et les trois fonctions (lignes 121 à 131).

Ensuite, on a initialisé deux variables pour le stockage du digest : une pour le format en hexadécimal, l’autre en binaire (lignes 54 et 55).

La première opération à effectuer sur les variables qui concernent le digest, est l’allouement de la mémoire (lignes 134 à 136). Cela est fait avec la fonction `xzalloc`, définie dans le fichier `utils.c` :

```

36 void *xzalloc(size_t size)
37 {
38     return memset(xmalloc(size), 0, size);
39 }

```

*Illustration 38: Définition de la fonction xzalloc du programme utils.c*

L'étape suivante est de convertir le format du digest en binaire. Comme montré en Figure xxx, la commande fsverity digest retourne une chaîne de caractères en hexadécimal, mais la fonction de signature du digest a besoin de ce dernier en format binaire. Cela est ressorti de l'analyse du code ainsi que de tests effectués pour comprendre le fonctionnement du programme. La conversion en binaire se fait en deux étapes : d'abord on copie le digest qu'on a reçu en paramètre lors de l'appel de la fonction fsverity digest (ligne 139) et puis on le converti en format binaire (ligne 140). Une remarque concernant la ligne 139 : on voit dans l'appel de la fonction strncpy qu'on lui passe comme paramètre argv[0]. Or, normalement cela devrait correspondre au mot « fsverity », mais la façon donc cmd\_sign.c est appelé par l'exécutable fsverity fait que argv[0] correspond au premier paramètre qu'on passe au programme après le mot digest. Donc, dans le cas de fsverity digest FILE (...), argv[0] contiendra FILE.

Après avoir converti le digest en binaire, on peut finalement initialiser la structure qui le contient, qui est la suivante (définie dans la librairie fsverity.h) :

```

30
31 struct fsverity_digest {
32     __u16 digest_algorithm;
33     __u16 digest_size;
34     __u8 digest[];
35 };
36

```

*Illustration 39: Définition de la structure du digest dans la librairie fsverity.h*

Les lignes 142 à 144 effectuent cette opération. La structure est maintenant prête pour être utilisée par la fonction de génération de signature, qui s'occupera d'ajouter le mot « FSVerity » dans une structure similaire à celle du digest (ligne 454 de sign\_digest.c).

Enfin, il ne faut pas oublier de libérer la mémoire qu'on a alloué pour nos variables. Cela est fait aux lignes 147, 148 et 175.

Les modifications au programme sont maintenant terminées. La seule opération qui reste est compiler manuellement les outils fs-verity pour faire en sorte que ces changements puissent être utilisés. Grâce au makefile présent dans les sources des outils, la commande make suffit à tout recompiler. Comme après la compilation l'exécutable se trouve dans le dossier où on a exécuté la

commande `make`, on peut le copier dans le repertoire d'installation des outils `fs-verity` (soit dans `/usr/bin/`) et ainsi l'utiliser sans spécifier son chemin.

Maintenant il est possible pour nous de faire appeler ce programme modifié par Ansible, qui se chargera d'effectuer les opérations `fs-verity` sur les fichiers.

### 3.2. AUTOMATISATION DES OPÉRATIONS

Dans le cahier des charges de notre projet on nous demande de faire en sorte que notre architecture soit compatible et puisse être déployée sur plusieurs centaines de serveurs et plusieurs milliers de services. Naturellement, il serait impensable d'effectuer les opérations d'activation de `fs-verity` manuellement. Par conséquent, il faut utiliser des outils qui nous permettent d'automatiser ces opérations. Comme vu dans les chapitres dédiés à l'automation d'une infrastructure définie par du code, l'outil de prédilection pour notre projet est Ansible. On va d'abord voir le fonctionnement d'Ansible de manière générale et ensuite on expliquera comment il sera utilisé dans le cadre de ce projet.

#### a) FONCTIONNEMENT D'ANSIBLE

Ansible est un moteur d'automatisation qui permet de provisionner, configurer, déployer une infrastructure informatique et de manière plus générale, d'automatiser certains processus qui seraient autrement exécutés manuellement.

Pour exécuter ces opérations, Ansible se base sur un système dit *agentless*, qui permet la gestion d'un client sans besoin d'avoir un service (agent) installé sur le client à gérer, ce qui rend plus légère et rapide la mise en place de la communication. La gestion s'effectue au travers d'une connexion `ssh`. Cela demande aussi une configuration préalable dans la majorité des cas, mais ça reste une manière simple et efficace de communiquer avec les postes clients. Le bon fonctionnement de la communication peut être testé avec la commande `ping` (qui n'est pas un `ping ICMP`, mais vérifie la connexion `ssh` vers le client). Ci-dessous un exemple :

```
iti@AG1:~/test-module$ ansible -i inventory all -m ping
ansible-test | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python3"
  },
  "changed": false,
  "ping": "pong"
}
```

*Illustration 40: Exemple de commande ping Ansible*



Une fois que la communication est en place, il est possible d'effectuer des opérations sur les clients. Une façon de faire est comme l'on vient de voir, soit via la ligne de commande. Mais cela devrait être utilisé uniquement pour effectuer des tests et non pour configurer une infrastructure entière. La façon correcte d'instruire Ansible sur les opérations à exécuter est de les décrire dans un playbook : un fichier écrit en langage YAML, qui contient la liste d'instructions à suivre.

Dans la configuration minimale, tout ce dont Ansible a besoin pour fonctionner sont un playbook et un inventaire. Ce dernier contient la liste des machines sur lesquelles on souhaite effectuer les modifications. On le retrouve dans le format suivant (ceci est un exemple) :

```
[web]
Apache-1
Apache-2
Apache-3
[log]
Syslog-GE
Syslog-ZH
```

Les mots entre crochets permettent d'effectuer des regroupements de machines. Ainsi, si on souhaite installer le service apache sur tous nos serveurs web, il suffit d'indiquer dans le playbook que les opérations doivent être effectuées sur le groupe web. Si en revanche on souhaite appliquer des modifications à toutes les machines de la liste, le mot clé `all` nous permet de faire cela (pas besoin de l'explicitier dans l'inventaire). La liste des machines peut être composé de hostname, adresses IP ou les deux. Le chemin par défaut de l'inventaire est `/etc/ansible/hosts`, où `hosts` est le nom de l'inventaire. En alternative, on peut indiquer manuellement où se trouve l'inventaire avec l'option `-i` quand on exécute une commande Ansible.

Pour écrire un playbook, on doit suivre un format de fichier spécifique. Ci-dessous un exemple d'un playbook pour copier un fichier sur tous les clients de l'inventaire :

```
1  ---
2  - hosts: all
3    vars:
4      filename: "/home/iti/example.txt"
5
6    tasks:
7      - name: copy file to all clients
8        copy:
9          src: "{{ filename }}"
10         dest: "{{ filename }}"
11
```

*Illustration 41: Exemple de playbook Ansible*

Le playbook doit commencer avec trois tirets. Ensuite, on liste certaines options, dans ce cas les clients auxquels le playbook s'applique (hosts : all) et on a définit également une variable filename, qui contient le chemin du fichier à copier. Après cette première partie, on a une deuxième section dédiée aux tâches à effectuer (tasks). On donne un nom à la tâche pour pouvoir la distinguer facilement lors du déploiement du playbook et ensuite on écrit les opérations à effectuer sur les clients. Pour cet exemple, on a utilisé la commande copy, qui demande au moins deux paramètres : le chemin source (src), qui sera le chemin sur la machine qui exécute Ansible et le chemin de destination (dest), qui est le chemin du fichier sur la machine à laquelle Ansible se connectera.

Le playbook est complet. On peut maintenant le tester avec la commande `ansible-playbook test.yml -i inventory`, où `test.yml` est notre playbook. L'option `-i inventory` peut être placée soit avant soit après `test.yml`. Voici le résultat de la commande :

```
iti@AG1:~$ ansible-playbook test.yml -i inventory
PLAY [all] *****

TASK [Gathering Facts] *****
ok: [ansible-test]

TASK [copy file to all clients] *****
changed: [ansible-test]

PLAY RECAP *****
ansible-test      : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

#### *Illustration 42: Exemple d'exécution d'un playbook Ansible*

On y retrouve deux tâches qui ont été exécutées. Une appelée « Gathering Facts », c'est une tâche obligatoire qu'Ansible effectue pour s'assurer qu'il peut bien se connecter aux clients. On pourrait l'exclure avec l'option « `gathering_facts : no` » dans le playbook. L'autre est notre tâche, qui porte le nom donné dans le playbook. À la fin de l'exécution de la commande, on a une récapitulation des opérations :

- Ok, si une tâche s'est bien terminée
- Changed, si des changements ont été effectués sur les clients
- Unreachable, si Ansible n'a pas pu se connecter au client
- Failed, s'il y a eu une erreur lors de l'exécution
- Skipped, si la tâche a été sauté parce que redondante
- Rescued, quand une tâche ne peut pas être exécutée et que dans le playbook on a défini un scénario de secours
- Ignored, quand une tâche est ignorée. Par exemple on peut exécuter Ansible en modalité de vérification : cela n'apportera pas de modifications aux clients, mais vérifiera si le playbook peut bien être exécuté.

Dans l'écriture du playbook il faut faire très attention à respecter la syntaxe et indentation YAML, sinon l'exécution du playbook échouera :

```
ERROR! We were unable to read either as JSON nor YAML, these are the errors we got from each:
JSON: Expecting value: line 1 column 1 (char 0)

Syntax Error while loading YAML.
  mapping values are not allowed in this context

The error appears to be in '/home/iti/test.yml': line 8, column 9, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

- name: copy file to all clients
  copy:
    ^ here
```

*Illustration 43: Exemple de message d'erreur en cas d'erreur de syntaxe YAML*

Dans l'exemple précédent on a utilisé la commande copy pour montrer le fonctionnement d'Ansible. Chaque commande est définie dans un module, qui est un fichier écrit en langage python dans lequel sont programmées les opérations que la commande doit effectuer et les options qu'elle peut prendre en compte. Pour connaître toutes les fonctionnalités d'un module, on peut se référer à la documentation correspondante, qui peut être affichée avec la commande `ansible-doc` suivie du nom du module. Tandis que pour savoir quels sont les modules qu'on a à disposition, on peut utiliser la documentation officielle présente sur le site d'Ansible ou alors dans son dépôt git. Si on a besoin de certaines fonctionnalités qu'on ne trouve dans aucun module, il est possible d'en développer un personnalisé. C'est exactement notre cas, puisqu'il n'existe pas de module qui permette la gestion de fs-verity. On va donc voir dans le prochain chapitre comment cela a été implémenté.

## **b) DÉVELOPPEMENT D'UN MODULE PERSONNALISÉ**

Pour la création du module qui fournit à l'utilisateur d'Ansible un moyen d'exploiter les fonctions de fs-verity, on s'est basés sur plusieurs sources, notamment la documentation officielle d'Ansible et certains tutoriels trouvés en ligne.

La première étape du développement est d'écrire la structure du module. Ci-dessous un exemple du format et du minimum nécessaire :

```

1  #!/usr/bin/python
2
3  from ansible.module_utils.basic import *
4
5  def enable(module):
6      filename = module.params["file"]
7      rc, out, err = module.run_command(["fsverity", "enable", filename])
8      response = {"rc": rc, "out": out, "err": err}
9      return response
10
11 def main():
12
13     fields = {
14         "enable": {"default": True, "required": False, "type": "bool"},
15         "file": {"required": True, "type": "str"},
16     }
17
18     module = AnsibleModule(argument_spec=fields)
19
20     if module.params["enable"] == True:
21         response = enable(module)
22
23     result = dict(changed=True, message=response)
24     module.exit_json(**result)
25
26
27 if __name__ == '__main__':
28     main()
29

```

*Illustration 44: Exemple minimale de module Ansible*

Au tout début on a `#!/usr/bin/python`, qui sert à spécifier l'interpréteur à utiliser. En effet on n'est pas obligés d'utiliser python, mais ça simplifie énormément le travail.

Ensuite on importe les librairies nécessaires, dans ce cas toutes celles de `ansible.module_utils.basic`, qui fournissent les fonctions qui permettent la création du module. Pour cet exemple on va en utiliser une seule, `AnsibleModule`, donc on aurait aussi pu importer uniquement celle-ci.

On passe ensuite aux deux dernières lignes du code écrit : c'est de la syntaxe python et on dit au programme d'appeler la fonction `main()` quand il est exécuté. Dans la fonction `main` on a une première variable qu'on a appelé `fields` et est un dictionnaire contenant tous les champs qu'on peut donner en paramètre lors de l'utilisation du module. Pour comparaison, dans le cas du module `copy`, ici on aurait retrouvé les champs `src` et `dest`. Chacun des champs peut avoir aussi des paramètres, notamment la valeur par défaut, s'il est obligatoire ou pas et le type de données attendus (booléen, chaîne de caractères, nombre, etc.).

La variable `fields` est ensuite utilisée comme paramètre lors de l'initialisation de la variable qu'on a appelé `module` (ligne 18). Cette variable est initialisée avec la classe `AnsibleModule`, qui

est celle à la base de toute la gestion du module Ansible. La classe peut accepter un grand nombre de paramètres, mais pour rester concis on renvoie à la documentation officielle et ici on se concentrera uniquement sur ceux utiles à notre cas.

Aux deux lignes suivantes (20-21) on spécifie les actions à entreprendre lorsque dans le playbook on fixe le paramètre `enable` à `True`. Dans cet exemple on appelle la fonction `enable`, qui fonctionne de la manière suivante : en prenant en paramètre le module, on peut récupérer également les autres paramètres utilisés dans le playbook. Ici, on récupère la valeur de `file` et on la stocke dans la variable `filename`. La ligne suivante (7) est le cœur de tout le module : c'est la ligne qui effectue les changements sur le client sélectionné. La fonction `run_command` est l'équivalent d'exécuter une commande dans un terminal. Ainsi, dans notre cas ceci est l'équivalent d'exécuter « `fsverity enable filename` » où `filename` est le chemin du fichier sur lequel on souhaite activer `fs-verity` et qu'on est censés donner en paramètre au playbook. Ce qui nous retourne cette commande est stocké dans trois variables, `rc`, `out` et `err`, qui correspondent respectivement au code de retour, à la sortie standard du terminal (`stdout`) et à celle d'erreur (`stderr`). Ces variables on peut les utiliser comme retour de notre fonction `enable`.

Notre module se termine aux lignes 23 et 24 : on ajoute au résultat le paramètre `changed`, qui est un de ceux vu dans le chapitre précédent et qui sert de récapitulation des tâches effectuées par le playbook. Dans d'autres cas on aurait pu mettre d'autres états, par exemple si on voulait récupérer le digest `fs-verity` du fichier on aurait fixé `changed` à `False` car on n'effectue pas de modifications sur le client. Enfin, l'appel à la fonction `exit_json` termine l'exécution du module et retourne le résultat en passé en paramètre en format JSON. Dans le cas où on doit terminer le module à cause d'une erreur, on peut utiliser la fonction `fail_json`.

Ceci était un exemple pour montrer le développement de base d'un module Ansible. La fonction `enable` utilisée ici fonctionnerait uniquement sur un client sur lequel on n'a pas rendu la signature obligatoire. Dans ce cas on aurait dû inclure également un paramètre supplémentaire pour l'ajout du fichier de signature.

Pour rendre fonctionnel notre module et le tester il y a encore une étape : indiquer à Ansible d'inclure ce module dans ceux disponibles. Cela se fait en plaçant le code écrit dans un dossier `library` :

```
iti@AG1:~/test$ tree
.
├── inventory
├── library
│   └── fsverity.py
└── test.yml
1 directory, 3 files
```

*Illustration 45: Exemple d'arborescence minimale pour fonctionnement Ansible avec un module personnalisé*

Le nom que l'on donne au fichier sera le nom dont le module devra être appelé (sans l'extension .py). En alternative, on peut le mettre dans le répertoire par défaut des modules, que l'on peut afficher avec la commande `ansible --version` :

```
iti@AG1:~$ ansible --version
ansible 2.10.8
  config file = None
  configured module search path = ['/home/iti/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3/dist-packages/ansible
  executable location = /usr/bin/ansible
  python version = 3.9.2 (default, Feb 28 2021, 17:03:44) [GCC 10.2.1 20210110]
```

*Illustration 46: Retour de la commande `ansible --version`*

Dans notre playbook la structure suivante :

```
1  ---
2  - hosts: all
3    vars:
4      filename: "/home/iti/example.txt"
5
6    tasks:
7      - name: enable fs-verity
8        fsverity:
9          enable: True
10         file: "{{ filename }}"
11         register: result
12
13      - debug: var=result
14
```

*Illustration 47: Exemple de playbook Ansible avec module personnalisé*

On retrouve l'appel à notre module et l'utilisation des deux paramètres `enable` et `file`. Ici on a ajouté le module `register`, qui permet d'enregistrer le résultat (celui passé en paramètre à `exit_json`) dans une variable et l'afficher à l'aide de la tâche `debug`.

Lors de l'exécution du playbook, on retrouve nos tâches et la valeur de retour de l'exécution de la commande. Dans notre cas `fsverity enable` ne retourne rien, mais si on exécute `fsverity`

mesure sur le fichier, on peut voir que notre module a bien fonctionné (affiché au fond de la capture ci-dessous).

```
iti@AG1:~/test$ ansible-playbook test.yml -i inventory

PLAY [all] *****

TASK [Gathering Facts] *****
ok: [ansible-test]

TASK [enable fs-verity] *****
changed: [ansible-test]

TASK [debug] *****
ok: [ansible-test] => {
  "result": {
    "changed": true,
    "failed": false,
    "message": {
      "err": "",
      "out": "",
      "rc": 0
    }
  }
}

PLAY RECAP *****
ansible-test      : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

iti@AG1:~/test$
iti@AG1:~/test$ ssh iti@ansible-test fsverity measure /home/iti/example.txt
sha256:d54c7e959ecd3caa8f181f4e0c2ddf7a350f62a64102e8d70efce49128ac9c48 /home/iti/example.txt
iti@AG1:~/test$
```

*Illustration 48: Exemple d'exécution de playbook Ansible avec module personnalisé*

Les explications fournies ici servent d'exemple pour le développement d'un module. Les différentes fonctions développées dans le cadre de ce projet peuvent être consultées dans le git du projet.

### c) IDEMPOTENCE

Une des caractéristiques d'Ansible est la possibilité d'effectuer plusieurs fois la même opération sans changer le résultat. On appelle idempotence la propriété d'un système de garantir que lorsqu'il effectue une opération, ça donnera toujours le même résultat, peu importe l'état initial et peu importe combien de fois on effectue l'opération. Si on prend comme exemple le module copy vu avant, il est considéré idempotent parce que l'état final après l'exécution du playbook est toujours la présence du fichier source dans le chemin de destination. Cela est vrai soit que sur le client cible le fichier n'existe pas, soit qu'il existe est et le même, soit qu'il existe mais avec un contenu différent. Après exécution du playbook on aura toujours la garantie que le fichier source a été copié à l'endroit de destination. Un autre exemple est lors de l'installation de paquets : apt ne réinstalle pas les paquets à chaque fois qu'on lance la commande `apt install`, mais vérifie d'abord si le paquet est déjà installé, en rendant le tout plus optimisé et surtout idempotent.

Ansible aussi effectue les opérations de manière idempotente et c'est un aspect fondamental de son fonctionnement : Ansible vérifie si le résultat souhaité a déjà été atteint et dans ce cas n'effectue pas de modifications. Cela le rend extrêmement rapide lorsqu'on doit répéter des opérations ou lancer plusieurs fois un playbook pour effectuer des tests. En effet, souvent les playbook peuvent être constitués d'un grand nombre de tâches et dans le cas d'une infrastructure avec plusieurs centaines de serveurs sur lesquels appliquer des modifications, la possibilité de ne pas effectuer certaines opérations quand il n'est pas nécessaire, comporte une énorme gagne de temps.

L'idempotence peut se faire soit au niveau du module (comme pour copy), soit au niveau du playbook. Tel qu'on l'a montré, le module vu dans le chapitre précédent n'est pas idempotent. En effet, si on essaie de l'exécuter une deuxième fois on a une erreur :

```
iti@AG1:~/test$ ansible-playbook test.yml -i inventory

PLAY [all] *****

TASK [Gathering Facts] *****
ok: [ansible-test]

TASK [enable fs-verity] *****
changed: [ansible-test]

TASK [debug] *****
ok: [ansible-test] => {
  "result": {
    "changed": true,
    "failed": false,
    "message": {
      "err": "ERROR: FS_IOC_ENABLE_VERITY failed on '/home/iti/example.txt': File exists\n",
      "out": "",
      "rc": 1
    }
  }
}

PLAY RECAP *****
ansible-test      : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

*Illustration 49: Exemple de manque d'idempotence dans l'exécution d'un playbook Ansible*

Pour faire en sorte que le module soit idempotent, il faudrait dans notre cas ajouter une vérification du fichier avant d'essayer d'activer fs-verity : si le fichier est déjà protégé par ce système, on ignore les changements et on continue l'exécution des instructions.

De manière similaire, on peut écrire un playbook pour qu'il soit idempotent. Par exemple, dans le cas où on l'utilise pour effectuer des modifications sur un fichier de configuration d'un serveur web apache, on pourrait ajouter dans le playbook une option pour redémarrer le service apache après avoir modifié le fichier : cela fera en sorte qu'à la fin de l'exécution, le serveur web sera fonctionnel avec la nouvelle configuration.



### 3.3. VÉRIFICATION PERMANENTE DE L'INTÉGRITÉ DES FICHIERS

L'activation de fs-verity sur des fichiers au travers d'Ansible ou manuellement garanti qu'il soit protégé contre la corruption de leur contenu. Toutefois, un programme qui utilise ces fichiers n'est pas en mesure de vérifier si fs-verity est activé ou non sur les fichiers qu'il utilise. Dès là le besoin de fournir à l'administrateur système un moyen de vérifier que fs-verity soit bien actif sur les fichiers d'intérêt.

On a alors écrit un script de vérification qui sera exécuté en tant que service sur la machine à protéger et qui garantira ainsi un contrôle permanent de l'intégrité et authenticité des fichiers. La mise en place se fait en deux étapes : le développement du script et son installation en tant que service.

#### a) SCRIPT DE VÉRIFICATION

Le code source du script peut être consulté sur le git du projet ou en annexe, mais le fonctionnement est le suivant. Le script prend en paramètre un fichier d'index, où sont listés tous les fichiers qui sont à protéger. On fait en sorte qu'uniquement des chemins valables sont pris en considération, un chemin faux ou le chemin d'un répertoire sont ignorés (pour rappel, on ne peut pas activer fs-verity sur un répertoire). Pour chaque chemin du fichier d'index, on récupère ses attributs avec la commande lsattr. Le 20ème caractère du résultat doit être une V, qui sert à indiquer que fs-verity est bien activé. Si on a rendu la signature obligatoire lors de l'initialisation de fs-verity, cette commande vérifiera au même temps l'intégrité et l'authenticité du fichier, au travers du mécanisme de vérification intégré dans le kernel. Si fs-verity n'est pas activé sur un fichier, ceci sera ajouté à un fichier de logs (chemin par défaut : /var/log/fsverity.log) avec la date et l'heure de la vérification. Enfin, on a ajouté une ligne qui met en pause le programme pendant 10 secondes, ceci pour éviter de remplir le fichier de log trop rapidement dans le cas où les fichiers resteraient sans fs-verity pendant longtemps.

Pour être efficace ce script doit tourner en tant que service, permettant ainsi un contrôle continu de notre machine.

#### b) SERVICE DE MONITORING

La création d'un service sous Linux se fait en ajoutant un fichier de configuration dans le répertoire /etc/systemd/system. Le fichier qu'on a préparé pour notre projet est le suivant :

```

1  [Unit]
2  Description=FS-Verity monitoring service
3
4  [Service]
5  Type=simple
6  Restart=always
7  RestartSec=1
8  User=root
9  ExecStart=/usr/bin/python3 /home/iti/service/fsverity_service.py
10
11 [Install]
12 WantedBy=multi-user.target
13

```

*Illustration 50: Fichier de configuration d'un service*

Dans la première section du fichier on a inséré la description du service. Dans la deuxième section on a ajouté deux lignes (6 et 7) pour faire en sorte que le service essaie de redémarrer s'il devrait s'arrêter soudainement. Les essais de redémarrage sont effectués trois fois (option par défaut de systemd) à intervalle d'une seconde (ligne 7). On a également spécifié qu'on souhaite exécuter notre script en tant qu'utilisateur root et avec l'option ExecStart on spécifie la commande que le service doit exécuter. Dans notre cas on va exécuter le script qu'on a créé. Dans la dernière section du fichier on a une option qui sert à indiquer au service de démarrer quand l'environnement Linux est « prêt », soit quand tous les modules de base sont déjà démarrés (par ex. interface réseau, etc.).

Après avoir copié ce fichier dans le répertoire de systemd, on peut démarrer le service avec la commande `systemctl start fsverity`. Et avec la commande `systemctl enable fsverity` on fait en sorte qu'il soit activé à chaque (re)démarrage de la machine.

Lorsque le service tourne et qu'il trouve des fichiers listés dans l'index qui n'ont pas de fs-verity activé, il les ajoute au fichier de logs :

```

iti@AG1:/etc/systemd/system$ cat /var/log/fsverity.log
2022-08-17 02:47:13 /home/iti/example.txt
2022-08-17 02:47:13 /home/iti/index
2022-08-17 02:47:13 /home/iti/test.yml
2022-08-17 02:47:24 /home/iti/example.txt
2022-08-17 02:47:24 /home/iti/index
2022-08-17 02:47:24 /home/iti/test.yml
2022-08-17 02:47:35 /home/iti/example.txt
2022-08-17 02:47:35 /home/iti/index
2022-08-17 02:47:35 /home/iti/test.yml

```

*Illustration 51: Exemple de fichier de logs généré par le service de vérification de l'intégrité*

On peut voir ici l'intervalle de 10 secondes entre une vérification et une autre. Autre possibilité est d'indiquer au script que l'on ne souhaite pas ajouter un fichier dans les logs s'il a

déjà été inclus dans les derniers 60 minutes par exemple, pour donner le temps à l'administrateur système de prendre les mesures nécessaires sans polluer le fichier de logs.

Dans cette implémentation, notre service tourne sur la machine à protéger, on y trouve également le fichier d'index et les logs. Pour améliorer la sécurité de cette solution, on peut imaginer de déporter tous ces fichiers sur un serveur tiers qui s'occuperait d'effectuer le monitoring. En effet, comme ça nous le rappelle la locution latine « *Quis custodiet ipsos custodes ?* » (qui surveille les surveillants ?), à l'état actuel si notre client est compromis par une attaque, on ne peut pas garantir que l'attaquant n'ait pas modifié le fichier d'index ou les logs ou arrêté le service. Vu qu'il nous faudrait un service qui monitore le service et ses fichiers, la solution serait d'effectuer la vérification depuis un autre serveur (auquel on pourra faire confiance et que l'on n'aura pas besoin de monitorer à son tour).

On peut également imaginer d'inclure notre service et les fichiers à lui liés dans l'index, ceci donnerait une sécurité en plus dans le cas où le service tournerait sur la machine à protéger. Mais non seulement, on peut charger une application externe de monitorer notre service (par ex. Icinga2) ou alors configurer le service pour qu'il envoie des messages réguliers (par ex. un ping) vers un système tiers pour notifier son bon fonctionnement : si le système de réception ne reçoit pas un message pendant un certain temps, il pourra lancer une alerte (par ex. par mail).

### **c) FICHIERS À PROTÉGER**

Lors de la mise en place de notre système de monitoring ainsi que lors de l'activation de fs-verity sur les fichiers avec Ansible, un administrateur système doit se poser la question pour quels fichiers il souhaite activer cette protection. Le choix reste bien évidemment à l'exploitant de notre solution, mais ici on aimerait donner quelques conseils sur quels fichiers on retient soient importants.

Un des critères principaux pour le choix des fichiers est qu'ils soient persistants : il n'est ni conseillé ni possible d'activer le mécanisme des protections sur des fichiers en changement constant, comme les fichiers de logs. Comme fs-verity rend les fichiers en lecture seule, l'idéal est de choisir des fichiers qui ne sont pas censés être modifiés régulièrement. Par exemple, des fichiers de configuration d'un serveur web : une fois que le serveur est en place, on ne modifie pas souvent sa configuration.

Ainsi, si on suit la hiérarchie du système de fichiers Linux, on devrait certainement protéger les exécutables qui se trouvent dans le répertoire /bin (c'est où sont stockés par ex. les binaires des

commandes `cat`, `ls`, `cp`, etc.). Un autre répertoire important est `/boot`, où on trouve tous les éléments utilisés lors du démarrage du système (notamment le kernel). On a ensuite `/etc`, qui contient un grand nombre de fichiers très importants, notamment les fichiers de configuration de presque tous les services du système. Une liste non exhaustive des fichiers et répertoires importants est la suivante : `xdg`, `logs`, `security`, `sudoers`, `rc2`, `rc5`, `ssh`, `dpkg`, `systemd`, `passwd`, `shadow`, `ssl/certs`, `sysctl`, `dhcp`, `wireguard`, `default`, `pam.d`, `iproute2`, `apt`, `grub`, `network/interfaces`, `apparmor`. Enfin, on a le répertoire `/usr` qui est moins critique par rapport aux autres, mais on y retrouve la majorité des utilitaires et applications pour les utilisateurs (et qui ne sont donc pas utilisés par le système opératif pour son fonctionnement de base). On y retrouve par exemple notre version des outils `fsverity`.

Ceux-ci et tout autre fichier retenu important devraient être pris en considération pour la protection offerte par notre projet.

### 3.4. CAS D'UTILISATION

Après avoir développé les outils vus dans les chapitres précédents, on va maintenant voir comment mettre tout ensemble pour obtenir une solution complète de vérification automatisé de l'intégrité et de l'authenticité de fichiers sur un serveur.

#### a) PRÉREQUIS

Avant de pouvoir appliquer notre système de sécurité au sein d'une infrastructure informatique il faut que les machines qui la compose soient préparées à l'avance avec les programmes et la configuration nécessaires. Les instructions qui suivent sont valables pour une infrastructure avec des machines tournant sous Debian 11. Le but de ce chapitre n'étant pas d'expliquer comment déployer une infrastructure, on va considérer que les machines aient déjà été déployée et prêtes à l'utilisation. Cela veut dire par exemple qu'elles aient une interface réseau active et que la communication réseau fonctionne, notamment on doit pouvoir effectuer un accès `ssh`.

La première étape est celle d'installer les paquets nécessaires pour utiliser Ansible. Comme Ansible est un produit RedHat, il n'est pas disponible par défaut dans le dépôt des paquets Debian. Il faut donc l'ajouter manuellement à la liste des dépôts.

```
apt install software-properties-common
apt-add-repository ppa:ansible/ansible
```

```
apt update
apt install ansible
```

Ensuite, sur cette même machine, on va installer les outils fs-verity, téléchargé depuis le git de notre projet. Il y a trois paquets qui sont requis pour les outils fs-verity et on peut les installer avec apt :

```
apt install gcc-multilib libssl-dev openssl
```

Les outils doivent être compilés manuellement avec la commande make, une fois dans le répertoire du projet qu'on a téléchargé. On copie ensuite l'exécutable fsverity dans le répertoire /usr/bin.

Les étapes suivantes consistent à activer fs-verity sur les clients avec le programme tune2fs :

```
sudo tune2fs -O verity /dev/sda1
```

Il faut également rendre obligatoire la signature des fichiers fs-verity :

```
iti@ansible-test:~$ sudo sysctl fs.verity.require_signatures=1
fs.verity.require_signatures = 1
iti@ansible-test:~$
```

*Illustration 52: Commande d'activation de la signature obligatoire fs-verity*

Après avoir créé une clé privée et un certificat comme vu dans le chapitre sur l'authenticité, soit :

```
openssl genrsa -aes128 -out private.pem 1024
openssl req -x509 -key private.pem -out certificate.crt -days 60
```

Et après avoir converti le certificat en format der :

```
openssl x509 -in certificate.crt -out cert.der -outform der
```

On doit l'ajouter dans le keyring fs-verity :

```
iti@ansible-test:~$ sudo keyctl padd asymmetric ' ' %keyring:.fs-verity < cert.der
932595903
```

*Illustration 53: Ajout d'un certificat dans keyring fs-verity*

Et on peut vérifier qu'il a bien été ajouté :

```
iti@ansible-test:~$ sudo keyctl show %:.fs-verity
Keyring
404424411 --a-swrv      0      0 keyring: .fs-verity
932595903 --als--v      0      0 \_ asymmetric: Hepia: Test Certificate: 6f7bdf05bd1f5d4541015c7f69a749fbf4efa0a9
iti@ansible-test:~$
```

*Illustration 54: Affichage du contenu du keyring fs-verity*

Sur le client aussi on doit installer les outils fs-verity, mais ici on n'a pas besoin d'utiliser les nôtres, puisque on n'a pas besoin du script de génération de la signature. Ainsi, la commande à utiliser est :

```
apt install fsverity
```

Le client est maintenant prêt. Bien évidemment, toutes ces opérations peuvent être effectuées par Ansible, puisqu'il ne serait pas envisageable de configurer des dizaines ou centaines de machines manuellement. Ici on a simplement montré les différentes étapes. Un playbook Ansible qui automatise ce processus est disponible dans le git du projet.

## b) DÉPLOIEMENT DU PLAYBOOK

Après la préparation des clients, on peut déployer notre playbook qui va activer fs-verity sur toutes les machines désignées. Elles devront être incluses dans l'inventaire. Autre étape est la création de la liste de fichiers à protéger sur chaque machine. Cette liste peut être différente pour chaque machine, ça sera l'administrateur système qui pourra décider et modifier la liste à son gré. Ci-dessous un exemple de la structure des différents fichiers qu'Ansible va utiliser pour le déploiement. Ces fichiers peuvent être téléchargés depuis le git du projet. Cette structure peut subir des modifications à la suite des adaptations du playbook.

```
iti@AG1:~/custom$ tree
.
├── inventory.ini
├── library
│   └── fs-verity.py
├── play.yml
├── roles
│   └── demo
│       ├── tasks
│       │   ├── fsverity.yml
│       │   └── main.yml
│       ├── templates
│       └── vars
│           └── main.yml
6 directories, 6 files
```

*Illustration 55: Arborescence du répertoire d'exécution du playbook Ansible*

Le playbook est déployé avec la commande suivante :

```
ansible-playbook play.yml -i inventori.ini
```

Et le résultat sera similaire au suivant :

```
iti@AG1:~/custom$ ansible-playbook play.yml -i inventory.ini

PLAY [FS-Verity] *****

TASK [Gathering Facts] *****
ok: [ansible-test]

TASK [demo : include_tasks] *****
included: /home/iti/custom/roles/demo/tasks/fsverity.yml for ansible-test => (item=/home/iti/abc.txt)
included: /home/iti/custom/roles/demo/tasks/fsverity.yml for ansible-test => (item=/home/iti/def.txt)

TASK [demo : get file digest] *****
changed: [ansible-test]

TASK [demo : generate verity signature for file] *****
Enter PEM pass phrase:
changed: [ansible-test]

TASK [demo : copy signature to host] *****
changed: [ansible-test]

TASK [demo : enable fs-verity on file] *****
changed: [ansible-test]

TASK [demo : delete signature on host] *****
changed: [ansible-test]

TASK [demo : delete signature on server] *****
changed: [ansible-test]

TASK [demo : check if fsverity is enabled] *****
changed: [ansible-test]
```

*Illustration 56: Résultat de l'exécution du playbook Ansible*

Les tâches sont effectuées en boucle : le playbook est exécuté pour chaque fichier de la liste (dans ce cas il y avait deux fichiers, abc.txt et def.txt).

Le module Ansible qu'on a développé inclut également une fonction de désactivation de fs-verity sur un ou plusieurs fichiers. Celle-ci peut être utilisée dans une tâche du playbook pour désactiver fs-verity et c'est la méthode à préférer lorsque on a besoin de le désactiver sur un grand nombre de fichiers.

Après que le playbook Ansible a été déployé, on peut démarrer le service de vérification de fs-verity, comme montré dans le chapitre correspondant. Cette tâche peut aussi être effectuée par Ansible.

### **c) MODIFICATIONS MANUELLES**

Quand on souhaite désactiver fs-verity sur un seul fichier et qu'on ne souhaite pas utiliser Ansible, on peut faire cela manuellement. À cet effet on a écrit un petit script qui nous permet la désactivation tout en gardant le service de vérification actif. On le trouve ci-dessous :

```

1  #!/bin/bash
2
3  index="/home/iti/index"
4
5  # Remove file from index
6  sed -i "s|$1||" $index
7
8  # Disable fs-verity
9  cp $1 $1.temp
10 mv $1.temp $1
11

```

*Illustration 57: Script de désactivation de fs-verity*

C'est un script écrit en bash et il est composé comme suit. En premier on trouve `#!/bin/bash` qui dit à qui exécute le programme d'utiliser le programme bash pour l'exécution. Ensuite on a ajouté une variable `index` dans laquelle on rentrera le chemin du fichier d'index que notre service `fsverity` utilise pour effectuer la vérification. On rentre après dans le vif du programme : la commande `sed` supprime le fichier de la liste d'index, cela évitera que notre service génère une alerte. Les deux dernières lignes sont celles qui effectuent l'opération de désactivation : on copie le fichier en le renommant et puis on remplace le remplace avec sa copie. En effet, la copie d'un fichier ne copie pas les métadonnées `fs-verity`, même si c'est effectué sur une machine avec `fs-verity` activé. Les fichiers `fs-verity` ne peuvent donc pas être transférés d'une machine à l'autre : il faut l'activer pour chaque fichier individuellement. Ci-dessous un exemple de copie :

```

iti@ansible-test:~$ cp example.txt example.txt2
iti@ansible-test:~$ lsattr
-----e----V- ./example.txt
-----e----- ./cert.der
-----e----- ./example.txt2
-----e----V- ./hello.txt
-----e----- ./file.txt
-----e----- ./abc
iti@ansible-test:~$ 

```

*Illustration 58: Copie d'un fichier fs-verity*

Ci-dessous un exemple de désactivation de `fsverity` avec notre script (qui pourra être copié dans `/usr/bin` et ainsi être appelé directement) avec deux captures d'écran : la première avant désactivation et la deuxième après désactivation.



```

iti@ansible-test:~$ lsattr abc.txt
-----e-----V- abc.txt
iti@ansible-test:~$
iti@ansible-test:~$ cat index
/home/iti/example.txt
/home/iti/index
hello
/home/iti/*
/home/iti/test.yml
/home/iti/abc.txt
/home/iti/service/*

```

*Illustration 59: État d'un fichier avant exécution script*

```

iti@ansible-test:~$ sudo disable /home/iti/abc.txt
iti@ansible-test:~$
iti@ansible-test:~$ lsattr abc.txt
-----e----- abc.txt
iti@ansible-test:~$
iti@ansible-test:~$ cat index
/home/iti/example.txt
/home/iti/index
hello
/home/iti/*
/home/iti/test.yml
/home/iti/service/*

```

*Illustration 60: État d'un fichier après exécution script*

On peut voir que fs-verity a bien été désactivé et qu'on a enlevé le fichier de l'index.

### 3.5. CONSIDERATIONS DE SECURITE

Après avoir déployé notre solution, il est important d'explicitier son périmètre de sécurité et les différentes possibilités de protection supplémentaire qui se présentent à nous ainsi que des éventuelles problématiques liées à la sécurité. Certains aspects ont déjà été évoqué tout le long du document, ici on va discuter des certains cas pas encore abordés.

Une des premières questions à laquelle on peut essayer de répondre est de savoir quoi faire dans le cas où notre clé privée soit compromise. Cette clé est à la base de toute la sécurité de la solution : si on ne peut pas garantir l'identité de qui signe les fichiers fs-verity (donc le détenteur de la clé privée), on perd la confiance de notre système. Ainsi on va voir la procédure à effectuer pour révoquer le certificat stocké dans le keyring fs-verity contre lequel la signature est vérifiée. Cette procédure est également utile lorsqu'on souhaite remplacer un certificat expiré (si on a donné une date d'expiration par exemple).

Avant tout, il faut générer la nouvelle clé et le nouveau certificat (par exemple avec `openssl`, comme vu dans les chapitres précédents). Ensuite, avec la commande `keyctl show %:.fs-verity` on peut afficher la liste des certificats qu'on a ajouté dans le keyring `fs-verity` :

```
iti@ansible-test:~$ sudo keyctl show %:.fs-verity
Keyring
404424411 --a-swr    0    0 keyring: .fs-verity
932595903 --als--v   0    0 \_ asymmetric: Hepia: Test Certificate: 6f7bdf05bd1f5d4541015c7f69a749fbf4efa0a9
iti@ansible-test:~$
```

*Illustration 61: Affichage du contenu du keyring `fs-verity`*

Et ils peuvent être révoqué avec l'option `unlink` de `keyctl`, suivie de l'identifiant du certificat :

```
iti@ansible-test:~$ sudo keyctl unlink 932595903 %:.fs-verity
iti@ansible-test:~$
iti@ansible-test:~$ sudo keyctl show %:.fs-verity
Keyring
404424411 --a-swr    0    0 keyring: .fs-verity
iti@ansible-test:~$ sudo keyctl list %:.fs-verity
keyring is empty
iti@ansible-test:~$
```

*Illustration 62: Suppression d'un certificat du keyring `fs-verity`*

Après vérification, on voit bien que le certificat a été révoqué. Maintenant, si on essaie d'ouvrir un fichier `fs-verity` signé et que le certificat n'est pas dans le keyring, on a une erreur :

```
iti@ansible-test:~$ cat example.txt
[38488.063041] fs-verity (vda1, inode 133960): File's signing cert isn't in the fs-verity keyring
cat: example.txt: Required key not available
```

*Illustration 63: Essai d'accès à un fichier signé avec `fs-verity` après suppression du certificat du keyring*

Pour plus de sécurité, on peut éventuellement restreindre l'accès au keyring et empêcher que tout nouveau certificat soit ajouté. Ceci avec la commande : `keyctl restrict_keyring %keyring:fs-verity` (à exécuter après avoir ajouté notre certificat).

Une autre option qui permet de garantir la sécurité de notre système est l'obligation de signature des fichiers. Cette opération doit absolument être faite en amont, avant que tout fichier ait `fs-verity` activé. En effet, si on rend la signature obligatoire après avoir activé `fs-verity` sur un fichier, le fichier est toujours accessible, même sans signature :

```
iti@ansible-test:~$  
iti@ansible-test:~$ cat file.txt  
Hepia 123  
iti@ansible-test:~$  
iti@ansible-test:~$ sudo sysctl fs.verity.require_signatures=0  
fs.verity.require_signatures = 0  
iti@ansible-test:~$  
iti@ansible-test:~$ fsverity enable file.txt  
iti@ansible-test:~$  
iti@ansible-test:~$ fsverity measure file.txt  
sha256:d4920dcd9f4c7db6d6bd3f5b4116f500b706947c40cafae3c042b7dc3d967179 file.txt  
iti@ansible-test:~$  
iti@ansible-test:~$ sudo sysctl fs.verity.require_signatures=1  
fs.verity.require_signatures = 1  
iti@ansible-test:~$  
iti@ansible-test:~$ fsverity measure file.txt  
sha256:d4920dcd9f4c7db6d6bd3f5b4116f500b706947c40cafae3c042b7dc3d967179 file.txt  
iti@ansible-test:~$  
iti@ansible-test:~$ cat file.txt  
Hepia 123  
iti@ansible-test:~$  
iti@ansible-test:~$ █
```

*Illustration 64: Tests sur la gestion des signatures fs-verity*

## CONCLUSION

Tout le long de ce document on a discuté de la protection d'une infrastructure informatique du point de vue de l'intégrité et de l'authenticité des fichiers qui sont hébergés sur les machines qui composent l'infrastructure. En particulier le but de ce travail était de fournir une solution de sécurité qui vérifie l'intégrité et l'authenticité de fichiers dans le contexte d'une infrastructure définie par du code, qui peut être constitué de centaines de serveurs. La solution devait en plus être compatible avec les produits d'automatisation d'une infrastructure, afin de simplifier le processus de déploiement et vérification.

Ainsi, dans la première partie de ce rapport, on a discuté du panorama des solutions existantes de vérification d'intégrité et authenticité : on a analysé plusieurs façons d'aborder le problème, notamment le stockage et vérification des hash, la vérification de l'intégrité intégrée dans les gestionnaires de paquets, on a vu les méthodes d'authentification ainsi que les outils présents dans le kernel Linux pour garantir la protection nécessaire. On a également analysé les solutions d'automatisation d'une infrastructure et on les a comparés entre elles. Nous avons finalement choisi deux outils : fs-verity et Ansible. Ces deux outils ont été utilisés comme base à partir de laquelle nous avons conçu notre architecture et que nous avons décrit dans le chapitre 2. Nous nous sommes aidés avec différents schémas et diagrammes, notamment le modèle C4 dont on a vu trois niveaux, le diagramme de séquence et le diagramme de déploiement. À partir du modèle conceptuel, on a pu développer la solution proposée. Notamment, nous avons adapté les outils fs-verity afin d'offrir plus de sécurité et d'efficacité dans notre solution et nous avons également développé un module Ansible pour le déploiement de fs-verity sur toute l'infrastructure. Aussi, un outil de vérification de l'intégrité et authenticité des fichiers a été développé et assure un monitoring continu en tournant comme service. Enfin, on a montré comment utiliser tous ces outils au travers d'exemples d'utilisation.

Ce travail a duré environ quatre mois et pendant ce temps il a fallu trouver le bon équilibre entre le temps à consacrer au projet et le temps passé en emploi pendant la journée. Ainsi, une bonne dose d'organisation était nécessaire. Le travail de recherche initial ça nous a permis d'approfondir certains arguments et d'en apprendre des nouveaux, notamment lorsqu'il s'agissait d'explorer les différents outils vus dans ce projet. Il a fallu par exemple analyser le code source d'outils développés par autrui, ce qui permet non seulement d'apprendre des nouvelles méthodes et fonctions, mais aussi comment d'autres ingénieurs abordent le même problème, fournissant ainsi des précieux points de vue différents. Lors de la phase de conception, il a fallu réfléchir à

comment ressembler tous les outils en une seule et unique solution de sécurité. Ici on avait comme base ce qui avait été fait lors du travail de semestre et il a fallu réimaginer le tout pour concevoir une nouvelle approche. Enfin, lors de la partie pratique d'implémentation de la solution, il a été intéressant et stimulant de développer et tester les différents outils et découvrir leur comportement et leurs interactions avec les systèmes existants. Ce projet a été une opportunité de mettre en place tout ce qui a été appris lors du cursus de Bachelor et les fréquents échanges avec le professeur encadrant ont porté à une connaissance plus approfondie de tous ces sujets et outils.

La fin de ce document ne doit pas forcément impliquer la fin du projet. Au contraire, avec ce travail on a jeté les bases pour des possibles améliorations et développements futurs. Notamment, il serait intéressant de développer ultérieurement les outils créés dans le cadre de ce travail et les tester dans un environnement plus étendu de ce qui a été possible lors de nos tests. Les cas d'utilisations possibles étant pratiquement infinis, il serait très intéressant de voir comment nos outils se comporteraient dans des scénarios que l'on n'a pas abordé dans ce projet, pour les adapter et les compléter avec ce qu'on découvrira lors de ces tests.

La perfection n'existant pas, il y a toujours de l'espace pour améliorer un projet et pour améliorer nous-mêmes, en apprenant toujours plus et en mettant en pratique ce qu'on a appris.

## **ANNEXES**

## **ANNEXE 1 – LIEN DU DÉPÔT GIT DU PROJET**

<https://gitedu.hesge.ch/sergio.guarino/travail-de-bachelor-guarino-sergio>

## ANNEXE 2 – ARBORESCENCE DES OUTILS FS-VERITY

```
iti@AG1:~/fsverity-utils$ tree
.
├── common
│   ├── common_defs.h
│   ├── fsverity_uapi.h
│   └── win32_defs.h
├── fsverity
├── include
│   └── libfsverity.h
├── lib
│   ├── compute_digest.c
│   ├── enable.c
│   ├── hash_algs.c
│   ├── libfsverity.pc.in
│   ├── lib_private.h
│   ├── sign_digest.c
│   └── utils.c
├── LICENSE
├── Makefile
├── man
│   └── fsverity.1.md
├── NEWS.md
├── programs
│   ├── cmd_digest.c
│   ├── cmd_dump_metadata.c
│   ├── cmd_enable.c
│   ├── cmd_measure.c
│   ├── cmd_sign.c
│   ├── cmd_sign.c.backup
│   ├── fsverity.c
│   ├── fsverity.h
│   ├── test_compute_digest.c
│   ├── test_hash_algs.c
│   ├── test_sign_digest.c
│   ├── utils.c
│   └── utils.h
├── README.md
├── scripts
│   ├── do-release.sh
│   ├── run-sparse.sh
│   └── run-tests.sh
├── testdata
│   ├── cert.pem
│   ├── file.sig
│   └── key.pem
└── 7 directories, 36 files
```



## ANNEXE 3 – SCRIPT DU SERVICE DE MONITORING FS-VERITY

```

1  #!/usr/bin/python3
2
3  import subprocess, os, time
4  from datetime import datetime
5
6  INDEX_PATH = "/home/iti/index" # List of files to verify
7  LOG_FILE = "/var/log/fsverity.log" # Output log file
8
9  # Function to verify if FS-Verity is enabled on file
10 # Input: output of command lsattr
11 # Output: boolean, True if FS-Verity is enabled, False otherwise
12 def check_verity(attr: str) -> bool:
13     if attr[20] == "v":
14         return True
15     else:
16         return False
17
18
19 # Function to get attributes of file passed as argument
20 # Input: file path
21 # Output: attributes of file
22 def get_file_attr(file: str) -> str:
23     cmd = "lsattr " + file
24     r = subprocess.run(["lsattr", file], capture_output=True)
25     r = str(r.stdout, "UTF-8") # return is binary. Converting to string.
26     return r
27
28
29 # Function to log all non FS-Verity files
30 # Input: timestamp, string formatted; log entry, string formatted
31 # Output: None
32 def log_entry(timestamp: str, entry: str) -> None:
33     with open(LOG_FILE, "a") as log:
34         log.writelines(timestamp + " " + entry) # adding log entry
35
36
37 if __name__ == "__main__":
38     with open(INDEX_PATH, "r") as index:
39         lines = index.readlines()
40         # print(lines)
41         for file in lines:
42             file = file[:-1] # removing \n
43
44             if not os.path.exists(file): # skipping inexistent paths
45                 continue
46             if not os.path.isfile(file): # skipping directories
47                 continue
48
49             file_attr = get_file_attr(file)
50
51             if not check_verity(file_attr):
52                 timestamp = datetime.now().replace(microsecond=0) # getting current time.
53                 log_entry(str(timestamp), file_attr[23:])
54
55             time.sleep(10) # wait 10 seconds before checking again

```

## RÉFÉRENCES DOCUMENTAIRES

APT (software), 2022. *Wikipedia*. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: [https://en.wikipedia.org/w/index.php?title=APT\\_\(software\)&oldid=1087313864](https://en.wikipedia.org/w/index.php?title=APT_(software)&oldid=1087313864)Page Version ID: 1087313864

BIGGERS, Eric, [sans date]. *kernel/git/ebiggers/fsverity-utils.git - Userspace utilities for fs-verity*. [en ligne]. [Consulté le 18 août 2022 a]. Disponible à l'adresse: <https://git.kernel.org/pub/scm/linux/kernel/git/ebiggers/fsverity-utils.git/tree/>

BIGGERS, Eric, [sans date]. *fs-verity: read-only file-based authenticity protection — The Linux Kernel documentation*. [en ligne]. [Consulté le 18 août 2022 b]. Disponible à l'adresse: <https://www.kernel.org/doc/html/latest/filesystems/fsverity.html>

BOEYEN, Sharon, SANTESSON, Stefan, POLK, Tim, HOUSLEY, Russ, FARRELL, Stephen et COOPER, David, 2008. RFC 5280 : *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile* [en ligne]. Request for Comments. Internet Engineering Task Force. [Consulté le 18 août 2022].

BROZ, Milan, [sans date]. *DMVerity · Wiki. GitLab*. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMVerity>

Changes/FsVerityRPM - Fedora Project Wiki, [sans date]. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://fedoraproject.org/wiki/Changes/FsVerityRPM>

CHINTHAGUNTLA, Keerthi, 2020. Linux package management with YUM and RPM. *Enable Sysadmin*. [en ligne]. 22 avril 2020. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://www.redhat.com/sysadmin/how-manage-packages>

CORBET, Jonathan, 2005. The Integrity Measurement Architecture [LWN.net]. [en ligne]. 24 mai 2005. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://lwn.net/Articles/137306/>

CORBET, Jonathan, 2011. *dm-verity* [LWN.net]. [en ligne]. 19 septembre 2011. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://lwn.net/Articles/459420/>

Digital signature, 2022. *Wikipedia*. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: [https://en.wikipedia.org/w/index.php?title=Digital\\_signature&oldid=1104707362](https://en.wikipedia.org/w/index.php?title=Digital_signature&oldid=1104707362)Page Version ID: 1104707362

DNF, *dnf latest documentation*, 2022. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://dnf.readthedocs.io/en/latest/index.html>

EDELMAN, Jason, LOWE, Scott S. et OSWALT, Matt, 2018. *Network Programmability & Automation*. . 1st Edition.

FERNÁNDEZ-SANGUINO PEÑA, Javier, [sans date]. *Securing Debian Manual*. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://www.debian.org/doc/manuals/securing-debian-manual/index.en.html>

Filesystem Hierarchy Standard, 2022. *Wikipedia*. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse:

[https://en.wikipedia.org/w/index.php?title=Filesystem\\_Hierarchy\\_Standard&oldid=1101996930](https://en.wikipedia.org/w/index.php?title=Filesystem_Hierarchy_Standard&oldid=1101996930)  
Page Version ID: 1101996930

GARNETT, Alex et ELLINGWOOD, Justin, 2022. Ubuntu and Debian Package Management Essentials | DigitalOcean. [en ligne]. 17 mars 2022. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://www.digitalocean.com/community/tutorials/ubuntu-and-debian-package-management-essentials>

Implementing dm-verity, [sans date]. *Android Open Source Project*. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://source.android.com/docs/security/verifiedboot/dm-verity>

KAPOOR, Nikita, 2021. What is Digital Signature? - GeeksforGeeks. [en ligne]. 31 août 2021. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://www.geeksforgeeks.org/what-is-digital-signature/>

Linux kernel, 2022. *Wikipedia*. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: [https://en.wikipedia.org/w/index.php?title=Linux\\_kernel&oldid=1102180326](https://en.wikipedia.org/w/index.php?title=Linux_kernel&oldid=1102180326) Page Version ID: 1102180326

Linux man pages online, [sans date]. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://www.man7.org/linux/man-pages/>

MILLER, Adam, SVISTUNOV, Maxim, DOLEŽELOVÁ, Marie et ET AL., 2020. RPM Packaging Guide. [en ligne]. 17 avril 2020. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://rpm-packaging-guide.github.io/>

Package management - openSUSE Wiki, 2019. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: [https://en.opensuse.org/Package\\_management](https://en.opensuse.org/Package_management)

Package Management, [sans date]. *Ubuntu*. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://ubuntu.com/server/docs/package-management> Ubuntu is an open source software operating system that runs from the desktop, to the cloud, to all your internet connected things.

PRAKASH, Abhishek, 2020. What is a Package Manager in Linux? [en ligne]. 4 octobre 2020. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://itsfoss.com/package-manager/>

Public key certificate, 2022. *Wikipedia*. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: [https://en.wikipedia.org/w/index.php?title=Public\\_key\\_certificate&oldid=1104231087](https://en.wikipedia.org/w/index.php?title=Public_key_certificate&oldid=1104231087) Page Version ID: 1104231087

Public-key cryptography, 2022. *Wikipedia*. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: [https://en.wikipedia.org/w/index.php?title=Public-key\\_cryptography&oldid=1102231347](https://en.wikipedia.org/w/index.php?title=Public-key_cryptography&oldid=1102231347) Page Version ID: 1102231347

SIDHPURWALA, Huzaifa, 2020. How to use the Linux kernel's Integrity Measurement Architecture. [en ligne]. 22 octobre 2020. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://www.redhat.com/en/blog/how-use-linux-kernels-integrity-measurement-architecture>

WENLIANG, Du, 2022. Computer and Internet Security, A Hands-On Approach. . 3rd Edition.

X.509, 2022. Wikipedia. [en ligne]. [Consulté le 18 août 2022]. Disponible à l'adresse: <https://en.wikipedia.org/w/index.php?title=X.509&oldid=1104964862>Page Version ID: 1104964862