Département des Technologie de l'information et de la communication (TIC)
Filière Télécommunications
Orientation Sécurité de l'information

Travail de Bachelor

Externalisation des clefs privées WireGuard sur des périphériques sécurisés

Étudiant	**Alexis Martins**
Enseignant responsable	Prof. Maxime Augier
Année académique	2023-2024

Yverdon-les-Bains, le 25 juillet 2024

Département des Technologie de l'information et de la communication (TIC)
Filière Télécommunications
Orientation Sécurité de l'information
Étudiant : Alexis Martins
Enseignant responsable : Prof. Maxime Augier

Travail de Bachelor 2023-2024

Externalisation des clefs privées WireGuard sur des périphériques sécurisés

Résumé publiable

Actuellement, les VPNs sont largement utilisés pour protéger les communications sensibles contre des menaces telles que l'interception de données et les attaques de type man-in-the-middle. Parmi les nombreuses solutions VPN, WireGuard se distingue par sa modernité, sa performance, sa sécurité robuste, sa rapidité et sa simplicité d'utilisation.

Le problème principal de ce travail réside dans la gestion de la clé privée de WireGuard, actuellement stockée sur le système de l'utilisateur au niveau de la mémoire, la rendant vulnérable en cas de compromission de la machine. Un attaquant pouvant extraire cette clé pourrait écouter les communications actuelles et usurper l'identité de l'utilisateur dans les sessions futures. Ce risque est un obstacle majeur à l'adoption de WireGuard dans des environnements hautement sécurisés.

Pour répondre à cette problématique, l'objectif de ce travail de Bachelor a été d'intégrer WireGuard avec des périphériques externes sécurisés afin d'externaliser la clé privée du VPN. En déplaçant la clé hors du système de l'utilisateur et en la stockant sur un périphérique sécurisé tel qu'une smartcard.

Le projet a d'abord exploré différentes implémentations et périphériques, optant finalement pour BoringTun (implémentation WireGuard en Rust) ainsi qu'une NitroKey en utilisant le protocole OpenPGP en raison de sa compatibilité et sa simplicité d'intégration.

Avant de passer à l'implémentation directement sur BoringTun, il a été primordial de réaliser un programme de test. Celui-ci a démontré que l'utilisation d'une smartcard pour les opérations cryptographiques est interchangeable avec l'utilisation de la librairie cryptographique de BoringTun, validant ainsi la faisabilité de l'externalisation de la clé privée.

Les résultats de ces tests ont ensuite été transférés directement sur l'implémentation de BoringTun afn d'avoir un VPN totalement fonctionnel avec une smartcard.

Étudiant :	Date et lieu :	Signature :
Alexis Martins	.....	.....
Enseignant responsable :	Date et lieu :	Signature :
Prof. Maxime Augier	.....	.....

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

HEIG-VD

Vincent Peiris
Chef de département TIC

Yverdon-les-Bains, le 25 juillet 2024

Authentification

Le soussigné, Alexis Martins, atteste par la présente avoir réalisé ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Yverdon-les-bains, le 25 juillet 2024

Alexis Martins

Cahier des charges

Motivations

Le choix de ce projet de Bachelor découle de l'importance croissante des VPNs dans la sécurité informatique, en particulier dans le contexte généralisé du télétravail. Actuellement, les VPNs sont largement utilisés pour protéger les communications sensibles contre des menaces telles que l'interception de données et les attaques de type man-in-the-middle. Le paysage des VPNs est diversifié, offrant différents protocoles avec des niveaux variés de sécurité, de performance et d'efficacité.

WireGuard se démarque comme une solution prometteuse grâce à sa simplicité, sa légèreté, sa rapidité et son efficacité, offrant ainsi une alternative attrayante dans un environnement où la performance et la sécurité sont cruciales. Sous Linux ou BSD, WireGuard est implémenté sous forme de module kernel, ce qui le rend très efficace en minimisant les changements de contexte. Malheureusement, ceci complique considérablement la compatibilité avec l'utilisation de périphériques cryptographiques externes. L'auteur a d'ailleurs précisé que cette fonctionnalité ne serait jamais supportée.

Les pratiques courantes de l'industrie demandent, pour les environnements hautement sécurisés, l'utilisation d'un tel périphérique. Le stockage de la clé privée et les opérations cryptographiques doivent être externalisées sur le périphérique sécurisé. Ceci est le moyen le plus simple de garantir la confidentialité de la clé en cas de compromission de la machine, et donc de limiter l'accès indu au VPN à la durée maximale d'une session (configurable côté serveur). Ce pré-requis freine actuellement l'adoption de WireGuard dans ce type d'environnement.

Objectifs

1. Comprendre et expliquer le fonctionnement de WireGuard, en particulier lors du processus de handshake.
2. Étudier les différentes possibilités d'implémentations et de matériel répondant aux besoins.
3. Choisir parmi ces possibilités la solution la plus adaptée et réaliser une implémentation de la fonctionnalité.

Objectifs optionnels

1. Étude du profil de performances des VPNs en user space sur Linux
   - Comprendre l'implémentation des sockets et de la pile réseau du noyau Linux
   - Comprendre les limitations imposées par l'interface tuntap, et les solutions choisies par les implémentations très performantes
   - Explorer les diverses possibilités pour améliorer la performance de cette interface : io_uring, netlink connector, virtIO networking, etc.

Livrables

1. L'implémentation de l'externalisation des clés privées
2. Un rapport contenant :
   - L'état de l'art concernant les VPNs existants sur le marché (OpenVPN, IPSec et WireGuard), leur fonctionnement et la manière dont sont gérées les clés
   - Une explication concernant le choix de l'implémentation WireGuard, ainsi que du matériel sélectionné
   - Les informations du module telles que le fonctionnement et les limitations
   - Un mode d'emploi
   - Un journal de travail

Table des matières

Préambule **v**

Authentification **vii**

Cahier des charges **ix**

1 Introduction **1**

1.1 Problématique 1

1.2 But 2

1.3 Déroulement du projet 2

2 État de l'art **3**

2.1 Principales solutions VPN 3

2.1.1 OpenVPN 3

2.1.2 IPSec 3

2.1.3 WireGuard 4

2.2 Fonctionnement de WireGuard 4

2.2.1 Général 4

2.2.2 Cryptographie 5

2.2.2.1 Noise Protocol Framework 6

2.2.2.2 Curve25519 6

2.2.2.3 ChaCha20 6

2.2.2.4 Poly1305 7

2.2.2.5 BLAKE2 . . . 7
2.2.2.6 SipHash24 . . . 7
2.2.2.7 HKDF . . . 7
2.2.3 Handshake et Noise Protocol Framework . . . 7
2.3 Implémentations de WireGuard . . . 8
2.4 Périphériques sécurisés . . . 9
2.5 Solutions existantes . . . 10
2.6 Problèmes potentiels . . . 11
2.6.1 Compatibilité . . . 11
2.6.2 Performance et robustesse . . . 11
2.6.3 Gestion du renouvellement de la session . . . 11

3 Périphérique sécurisé **13**
3.1 Solution choisie . . . 13
3.1.1 Périphérique choisi . . . 13
3.1.2 Protocole choisi . . . 14
3.1.2.1 OpenPGP et son fonctionnement . . . 14
3.1.3 Librairies utilisées . . . 15
3.2 Implémentation . . . 16
3.2.1 Préparation de la clé . . . 16
3.2.2 Réalisation d'un programme de test . . . 20
3.3 Problèmes rencontrés . . . 23
3.3.1 Limitations des périphériques et transition vers OpenPGP . . . 23

4 Implémentation de WireGuard **25**
4.1 Solution choisie . . . 25
4.1.1 Fonctionnement . . . 26
4.1.2 Cryptographie . . . 27
4.1.3 Utilisation . . . 28
4.1.3.1 Mise à jour de wg-quick . . . 28

4.1.3.2 Préparation d'une configuration WireGuard 28
4.1.3.3 Appliquer une capabilty sur l'exécutable 29
4.1.3.4 Lancement et arrêt du programme 29
4.2 Implémentation 30
4.2.1 Utilisation de la clé privée 30
4.2.2 Obtention de la clé publique 33
4.2.3 Récupération du PIN de l'utilisateur 36
4.3 Tests 37
4.3.1 Environnement de test 37
4.3.1.1 Architecture de l'environnement 37
4.3.1.2 Configuration WireGuard des machines 38
4.3.2 Tests effectués 39
4.4 Problèmes rencontrés 40
4.4.1 Modification de wg-quick 40
4.4.2 Modification de x25519-dalek 41

5 Performances **43**
5.1 Fonctionnement du réseau au niveau kernel 43
5.1.1 Stack réseau du kernel 43
5.1.1.1 Ring buffers 44
5.1.1.2 Socket buffers 44
5.1.1.3 net_device 45
5.1.1.4 Interruptions 45
5.1.1.5 NAPI (New API) 45
5.1.1.6 Vue d'ensemble 46
5.1.2 Implémentation des sockets 47
5.1.3 Interfaces TUN/TAP 48
5.1.4 Fonctionnement spécifique de WireGuard 49
5.2 Limitations et solutions alternatives 49

5.2.1 Limitations de l'interface TUN/TAP 49
5.2.2 Solutions existantes 50
5.2.2.1 io_uring 50
5.2.2.2 (recv/send)mmsg 52
5.3 Preuves de concepts et résultats 52
5.3.1 Mise en pratique des solutions 52
5.3.1.1 I/O classiques 53
5.3.1.2 I/O classiques asynchrones 54
5.3.1.3 io_uring 55
5.3.1.4 io_uring asynchrone 56
5.3.2 Résultats 58
5.3.2.1 Test de débit 59
5.3.2.2 Test de temps de réponse 60
5.3.2.3 Test du temps pris par les appels système 61
5.3.2.4 Test du nombre d'appels système 62
5.4 Travaux futurs 62

6 Conclusion **63**

Bibliographie **65**

Table des Figures **69**

Liste des tableaux **71**

A Journal de travail **73**

Chapitre 1

Introduction

Dans le contexte actuel où la cybersécurité est devenue une priorité majeure face à l'accroissement des menaces en ligne, les réseaux privés virtuels (VPN) ont gagné en importance. Ces outils sont essentiels pour sécuriser les communications sur internet, notamment dans des scénarios tels que le télétravail, où les données sensibles transitent fréquemment entre différents réseaux. Parmi les nombreux VPN disponibles, WireGuard [12] se distingue comme une solution moderne et performante. Conçu pour surpasser les anciens protocoles tels qu'IPSec et OpenVPN, WireGuard offre une combinaison optimale de sécurité robuste, de rapidité et de simplicité.

En parallèle, l'émergence de périphériques externes sécurisés a transformé le paysage de la gestion de l'authentification et de la sécurisation des données. Ces dispositifs offrent un éventail de fonctionnalités telles que le stockage sécurisé de clés privées et de certificats, et la gestion de l'authentification multi-facteurs, jouant ainsi un rôle crucial dans la protection des identités numériques et des communications.

1.1 Problématique

WireGuard fonctionne en utilisant des paires de clés cryptographiques pour établir des connexions sécurisées. Parmi ces clés, la clé privée à long terme est d'une importance capitale, car elle est impliquée dans chaque étape de communication. Actuellement, cette clé est stockée directement sur le système de l'utilisateur, la rendant vulnérable en cas de compromission de la machine. Un attaquant parvenant à extraire cette clé pourrait non seulement écouter les communications actuelles, mais également usurper l'identité de l'utilisateur dans toutes les sessions futures.

1.2 But

L'objectif de ce travail de bachelor est d'intégrer WireGuard avec des périphériques externes sécurisés pour externaliser la clé privée du VPN. Cette approche vise à renforcer la sécurité de WireGuard en minimisant le risque associé à la compromission de la clé privée. En déplaçant la clé hors du système principal de l'utilisateur, nous limitons l'impact potentiel d'une attaque réussie à la session VPN en cours, sans compromettre la sécurité des communications futures. Ce projet explorera la faisabilité technique de cette intégration, tout en cherchant à maintenir, voire à améliorer, les performances et la simplicité d'utilisation de WireGuard.

En poursuivant ces objectifs, ce travail aspire à contribuer à l'évolution des VPNs vers des solutions toujours plus sécurisées et résilientes, capables de protéger efficacement les communications dans un environnement numérique en constante évolution.

1.3 Déroulement du projet

Dans un premier temps, il y aura une phase de recherche concernant le fonctionnement de WireGuard. Le protocole est assez simple, mais il est important de comprendre quelles parties de celui-ci concernent ce projet.

La seconde phase consiste à se renseigner sur des implémentations de WireGuard auxquelles on pourrait apporter des modifications afin de réaliser notre externalisation. Dans cette même phase, il faudra trouver quel périphérique externe sera le candidat idéal pour ce projet. Il faut prendre en compte les compatibilités cryptographiques avec le protocole de WireGuard, ainsi que la compatibilité avec le langage de programmation Rust. Durant cette phase, il y aura aussi une partie de recherche concernant les solutions existantes sur le marché pour la problématique de ce travail de Bachelor.

La troisième phase consiste à mettre en place de la fonctionnalité répondant à la problématique. Le but est d'adapter le programme de test réalisé dans la seconde phase à l'implémentation de WireGuard choisie.

Une quatrième et ultime phase regroupant des objectifs optionnels pourra faire partie du projet dans le cas où les précédents objectifs sont remplis. Cette dernière phase explorera des objectifs n'ayant pas forcément de lien avec la problématique principale, mais permettant de tout de même améliorer soit la sécurité, soit la performance de WireGuard.

Chapitre 2

État de l'art

Ce chapitre vise à explorer, comprendre et poser les bases concernant les divers sujets abordés dans ce travail de Bachelor. On y traite des informations très haut niveau comme les diverses solutions VPN existantes sur le marché afin d'ajouter du contexte. Mais aussi des informations plus précises sur le projet comme le fonctionnement de WireGuard avec une vue spécifique au projet.

Dans ce chapitre, on retrouve également les différentes implémentations et divers périphériques qui répondent aux besoins de ce projet. Finalement, une section sera dédiée aux problèmes qui peuvent survenir durant le projet durant l'implémentation ou suite à celle-ci.

2.1 Principales solutions VPN

2.1.1 OpenVPN

OpenVPN [10] est un protocole VPN open source qui a été fondé dans le début des années 2000. La différence principale avec les deux autres VPNs présentés dans cette courte introduction, vient du fait que OpenVPN soit implémenté totalement dans l'espace utilisateur. Cela signifie qu'il est vu comme n'importe quelle autre application TCP/UDP "normale" (couche 7) pour la partie chiffrée des communications. La partie des communications étant en clair passe par le TUN/TAP. Cette différence signifie que OpenVPN sera par définition moins performant que les deux autres solutions proposées ici.

2.1.2 IPSec

IPSec [23] est le plus ancien des trois protocoles présentés ici, il a été introduit pour la première fois vers 1990. Il utilise des algorithmes cryptographiques plus anciens, mais sûrs

tels qu'AES. IPSec est principalement implémenté côté noyau même si la phase de handshake est réalisée du côté espace utilisateur. C'est pour cette raison que IPSec est aussi proche de WireGuard en termes de performances, il possède de très bonnes primitives cryptographiques et une implémentation partielle dans le noyau (le datapath). IPSec est le VPN le plus utilisé dans l'industrie, mais c'est aussi celui que WireGuard vient remplacer petit à petit.

2.1.3 WireGuard

WireGuard [12, 25] est un protocole VPN moderne conçu pour être à la fois simple et très performant. Créé en 2015 par Jason Donenfeld, il a rapidement gagné en popularité grâce à son approche épurée, se basant sur seulement 4000 lignes de code, contrairement aux alternatives plus anciennes et plus lourdes. Ce protocole utilise des standards cryptographiques de pointe, tels que ChaCha20 pour le chiffrement et Curve25519 pour l'échange de clés, visant à offrir une sécurité robuste tout en maintenant une haute efficacité. Son intégration officielle dans le noyau Linux en 2020 a marqué une étape significative, soulignant sa fiabilité et sa maturité.

La performance remarquable de WireGuard repose sur deux piliers principaux : son intégration directe dans le noyau Linux et l'adoption des dernières avancées cryptographiques. Cette intégration noyau permet un traitement des données plus rapide et plus direct que les solutions basées sur l'espace utilisateur, réduisant ainsi la latence et augmentant le débit. Par ailleurs, l'utilisation de technologies cryptographiques modernes et efficaces minimise le temps de traitement par paquet.

2.2 Fonctionnement de WireGuard

2.2.1 Général

WireGuard est un protocole se situant à la couche 3 (Réseau) du modèle OSI, les communications sont faites via des tunnels UDP. WireGuard met à disposition une interface virtuelle (souvent nommée wg0) à laquelle sont envoyées les données à transmettre. Cette interface va ensuite transmettre ces données à WireGuard pour que celui-ci les chiffre avec ChaCha20-Poly1305. Enfin, les données chiffrées sont envoyées au destinataire.

Afin de savoir quel matériel cryptographique utiliser lors du chiffrement, la configuration de WireGuard est faite avec une liaison entre une IP et une clé publique. Dans WireGuard cette liaison entre une/des adresse(s) IP et une clé publique s'appelle un pair. Ce mécanisme est appelé le Cryptokey Routing. Comme son nom l'indique, selon les adresses IP de destination, on va réaliser un "routage" sur le choix de la clé à utiliser. Afin que le programme fonctionne correctement, il faut définir tous les pairs dans sa configuration. Par exemple, dans la configuration ci-dessous, si on souhaite envoyer un message à l'adresse 10.192.122.3/32, alors on

utilisera la clé commençant par xTIBA5...

Code Source : Exemple de configuration WireGuard tirée du site officiel [12]

```
[Interface]
PrivateKey = yAnz5TF+lXXJte14tji3zlMNq+hd2rYUIgJBgB3fBmk=
ListenPort = 51820

[Peer]
PublicKey = xTIBA5rboUvnH4htodjb6e697QjLERt1NAB4mZqp8Dg=
AllowedIPs = 10.192.122.3/32, 10.192.124.1/24

[Peer]
PublicKey = TrMvSoP4jYQlY6RIzBgbssQqY3vxI2Pi+y71lOWWXX0=
AllowedIPs = 10.192.122.4/32, 192.168.0.0/16
```

Afin de commencer à échanger des informations, WireGuard propose de faire un échange de clés avec Diffie-Hellman sur courbes elliptiques Curve25519 (ECDH). Il va utiliser le Noise Protocol Framework (détaillé dans la section 2.2.2.1) dans sa version IK afin d'avoir une paire de clé partagée pour dialoguer (envoyer et recevoir). Ces clés sont dites éphémères, car elles vont être utilisées afin de chiffrer les informations de la session courante. Si on établit une nouvelle session, ces clés seront fondamentalement différentes. Cela garantit d'avoir une sécurité parfaite Dans la documentation officielle, ils appellent cela le 1-RTT étant donné qu'il faut au moins un message avant d'avoir un canal sécurisé sur lequel on peut échanger des informations.

Le fait que l'échange de clés soit fait en un seul message pourrait permettre à un attaquant de réaliser une attaque par rejeu. Cependant, ce cas a été traité lors de la conception de WireGuard en ajoutant un timestamp TAI64N avec l'échange, celui-ci étant chiffré et authentifié. Ainsi le destinataire garde une copie du timestamp envoyé pour chaque pair et à chaque fois qu'un nouveau message est envoyé, le nouveau timestamp doit être supérieur au précédent.

A noter que WireGuard propose aussi d'utiliser une clé pré-partagée afin de renforcer sa sécurité face à des menaces post-quantiques. Les atouts de WireGuard face au post-quantique ont été renforcés suite à son adaptation avec des algorithmes dédiés [19]. Mais cela sort du cadre de ce travail de bachelor.

2.2.2 Cryptographie

Dans cette courte section, le but est de mettre en valeur les algorithmes cryptographiques utilisés par WireGuard afin d'ajouter du contexte à ce travail de Bachelor. De plus, cela permet de démontrer que WireGuard utilise des algorithmes modernes.

Chaque sous-section sera uniquement un bref résumé de chaque primitive et leur but au

sein de WireGuard. Le but n'est pas de rentrer en profondeur dans chaque algorithme. Les parties importantes telles que le Noise Protocole et le handshake seront traitées à part dans une section dédiée.

2.2.2.1 Noise Protocol Framework

Noise [28] est un Framework et non un protocole directement, ce qui signifie qu'il propose des manières de faire afin de construire un protocole sécurisé basé sur Diffie-Hellman. C'est grâce à Noise que dans WireGuard, on arrive à avoir un échange de clés sécurisées. Par défaut, il propose plusieurs patterns que l'on peut choisir en fonction de ce que possède chaque parti lors de l'échange. Les patterns sont définis par deux lettres, chaque lettre pouvant être N, K, X ou I. La première lettre désigne ce que possède l'émetteur (initiateur de la communication) et la seconde ce que possède le répondeur.

- **N :** Aucune clé longue durée connue de l'autre partie.
- **K :** Clé longue durée connue de l'autre partie.
- **X :** Clé longue durée transmise à l'autre partie.
- **I :** Clé longue durée immédiatement transmise de l'émetteur au destinataire (unidirectionnel). Divulgation en cas d'un attaque MITM.

Avec ces 4 lettres, on peut donc construire des patterns comme celui utilisé pour le handshake de WireGuard (NoiseIK). Celui-ci sera détaillé dans la section dédiée au handshake.

2.2.2.2 Curve25519

Curve25519 [3] est la courbe utilisée pour réaliser des opérations avec Diffie-Hellman sur courbes elliptiques. Pour être plus précis, c'est X25519 qui est utilisé (Curve25519 étant l'ancien nom). Si Noise était la description de la manière dont on devait faire l'échange de clés, alors X25519 est l'outil réellement utilisé pour l'échange. Il permet d'obtenir un secret partagé entre deux entités comme ce qui est fait dans le handshake de WireGuard.

2.2.2.3 ChaCha20

ChaCha20 [4] est un algorithme symétrique de chiffrement par flot basé sur Salsa20. C'est cet algorithme qui est utilisé pour le chiffrement de données lorsque le handshake a eu lieu. Par défaut ChaCha20 ne propose aucun mécanisme de vérification d'intégrité, mais il est souvent assemblé avec Poly1305 (ce qui est le cas ici) afin de fournir cette propriété. ChaCha20 possède un état de base dans lequel on met une constante, la clé, la position du bloc courant (pour générer plusieurs blocs pour le flot aléatoire final), ainsi qu'un vecteur d'initialisation. Cet ensemble est passé dans plusieurs rondes afin de rendre les opérations

difficilement réversibles par une personne n'ayant pas la clé. Le résultat final étant donc un flux de bits qui peut être utilisé pour chiffrer l'information.

2.2.2.4 Poly1305

Poly1305 [5] est un MAC (Message Authentication Code) qui permet de garantir l'intégrité et l'authenticité d'un message. On retrouve très souvent ce MAC avec ChaCha20 et WireGuard n'échappe pas à la règle. Ainsi, les données chiffrées échangées suite au handshake sont garanties d'être intègres et authentiques.

2.2.2.5 BLAKE2

BLAKE2 [2] est une fonction de hachage assez similaire à SHA-3 qui permet d'avoir des sorties de tailles variables. Au sein de WireGuard, elle est utile pour effectuer certains hash nécessaires pour faire le MAC et le HMAC qui sont utilisés durant le handshake.

2.2.2.6 SipHash24

SipHash24 [1] est une fonction pseudo-aléatoire qui est utilisée en tant que MAC. Comme on l'a vu, WireGuard stocke des pairs qui sont le lien entre une adresse IP et la clé publique correspondante pour dialoguer. Afin de stocker cela, WireGuard utilise une table de hachage et la fonction utilisée dans cette table est SipHash24.

2.2.2.7 HKDF

HKDF [22] est une fonction de dérivation de clé basée sur HMAC. Elle est notamment utile lorsqu'il faut étendre une valeur (par exemple un hash) et que l'entropie de cette valeur est déjà bonne. Au sein de WireGuard, elle est utilisée pour étendre les sorties de ECDH. Cela est utile, car ça permet de transformer la sortie de ECDH en une clé complètement utilisable pour les chiffrements futurs [24].

2.2.3 Handshake et Noise Protocol Framework

La version IK [6] signifie que l'initiateur possède en amont la clé longue durée du répondeur (K) et que la clé de l'initiateur sera transmise lors du premier échange au répondeur (I). Dans le protocole, ils utilisent aussi dans chaque message une façon particulière d'écrire quelles clés sont envoyées et quels secrets sont créés. Si on prend l'exemple du message A dans le pattern IK, on voit qu'il est écrit "e, es, s, ss". Lorsque c'est une seule lettre comme "e" ou "s", cela signifie que c'est la clé éphémère ou la clé statique (long terme) qui est transmise.

Lorsque c'est un combiné de lettre comme "es" ou "ss", cela indique les composantes d'un secret créé avec Diffie-Hellman. Par exemple quand c'est "es", cela signifie que le secret est composé de la clé éphémère de l'initiateur et de la clé statique du répondeur.

1. **Initiation** (message A) : L'initiateur envoie un message contenant sa clé publique statique, chiffrée avec la clé publique statique du destinataire, ainsi que des parties de clés éphémères. Cela garantit que seul le destinataire prévu peut déchiffrer le message et poursuivre le processus de handshake.
2. **Réponse** (message B) : Le destinataire déchiffre le message avec sa clé privée, valide la clé publique de l'initiateur et complète son côté des échanges Diffie-Hellman en utilisant à la fois les clés éphémères et statiques. Il envoie ensuite une réponse incluant sa clé publique éphémère, chiffrée avec la clé publique de l'initiateur et un hash du processus de handshake jusqu'à ce point, assurant l'intégrité et l'authenticité du processus.
3. **Communication Sécurisée** (messages C et D) : Les messages suivants, comme C et D, sont des exemples de données échangées pendant la session établie. Il peut y avoir un nombre indéterminé de messages échangés dans cette phase.

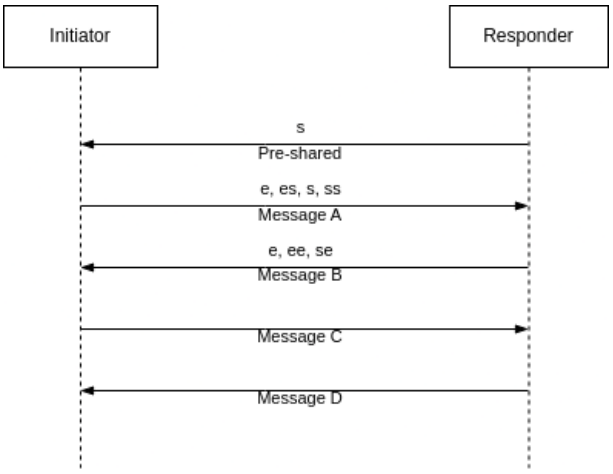


FIGURE 2.1 – Schéma NoiseIK

2.3 Implémentations de WireGuard

Un des objectifs clés de ce projet est que l'implémentation de la fonctionnalité soit faite en Rust. Il y a deux raisons à ce choix, la première étant que Rust est le langage du futur grâce aux différentes sécurités dont il se charge. La seconde vient du fait qu'il est plus judicieux d'utiliser un langage pensé pour la sécurité et la performance dans le cadre d'un projet où ces deux notions sont les plus importantes.

Il faut donc se demander quelle base de code va être utilisée afin d'y ajouter la fonctionnalité. Lors de la réalisation de l'état de l'art, les choix suivants semblent se présenter pour ce projet :

- BoringTun
- Wireguard-rs
- Implémentation personnalisée

La première solution est clairement la plus stable, BoringTun est un programme qui marche et qui est assez connu. Sa force est aussi sa faiblesse, le fait que ça soit un programme complet complexifie grandement les modifications dessus. Le principal inconvénient serait donc de perdre beaucoup de temps à comprendre le code qui y est écrit et de ne pas se concentrer sur l'objectif de ce projet.

La seconde solution est nettement plus simple, le problème par contre est que le projet n'est pas forcément stable. Lorsque l'on lit la brève description du projet sur GitHub, on se rend compte que le créateur déconseille de l'utiliser.

La dernière solution est peut-être celle où il sera le plus simple d'implémenter la fonctionnalité demandée. Par contre le grand désavantage, c'est qu'elle n'existe pas et même si WireGuard est réputé pour être très léger, il n'est sûrement pas possible de l'implémenter et d'y apporter des modifications dans le temps imparti pour ce travail de Bachelor.

2.4 Périphériques sécurisés

Le périphérique utilisé est le second facteur important afin de réaliser ce projet. Plusieurs aspects sont à prendre en compte, le premier étant de savoir si le périphérique supporte la courbe Curve25519. Un second facteur important est de savoir de quelle manière on peut stocker la clé sur ce périphérique. Il existe de multiples solutions telles qu'utiliser OpenPGP ou encore les PIV (Personal Identity Verification) slots. Finalement, il faut aussi savoir s'il existe en Rust déjà une librairie/wrapper qui nous permet de dialoguer avec ce périphérique et d'effectuer des opérations sur la courbe Curve25519.

En réalisant mes recherches, je suis tombé sur les trois clés ci-dessous qui malgré le fait qu'elles servent toutes à la même chose, elles possèdent toutes leurs petits détails qui ont une importance cruciale.

Pour commencer, les clés YubiKey et NitroKey sont assez similaires dans leur manière de fonctionner et dans les options qu'elles proposent. Elles possèdent aussi diverses manières de stocker une clé privée X25519. On retrouve notamment les emplacements PIV qui sont des compartiments dans le périphérique auxquels on se référence avec un numéro d'identification et chacun de ces compartiments possède une utilisation bien particulière telle que le chiffrement, le déchiffrement, la signature, etc. Une seconde opportunité que nous offrent ces clés est la compatibilité qu'elles ont avec OpenPGP. L'avantage de ce moyen est que

OpenPGP est un standard extrêmement connu et avec de nombreuses librairies pour communiquer avec celui-ci. Il reste tout de même une différence entre ces deux clés, étant donné que la NitroKey possède un firmware open-source et que l'on peut mettre à jour. Tandis que la YubiKey n'est pas open-source et on ne peut pas mettre à jour les clés pour des raisons de sécurité.

La seconde option serait la TKey qui est fabriquée par Tillitis (un spin-off de Mullvad). Cette clé est très différente des deux autres, car il est possible de créer ses propres programmes en C ou Rust et de les faire exécuter par la clé. Dans le cadre de ce projet, cela pourrait être pertinent d'implémenter directement sur la clé le code réalisant l'ECDH. C'est un périphérique offrant énormément de flexibilité et de possibilités aux développeurs.

Comme pour l'implémentation sur laquelle ce projet portera, le choix du périphérique sera détaillé dans une section à part.

2.5 Solutions existantes

Lorsque l'on se documente sur des solutions existantes répondant à la problématique, on en trouve très peu. La majorité des solutions sont disponibles sur les autres VPNs présentés en amont dans ce travail.

Il est tout de même possible de trouver une entreprise ayant réalisé un travail similaire. Pro Custodibus est une entreprise américaine qui propose une surcouche graphique réalisée en Go pour WireGuard afin de gérer les connexions VPN. Parmi les multiples fonctionnalités proposées dans leur logiciel, on en retrouve une qui traite le même cas que ce projet [8].

Ils utilisent un agent Python appelé "openpgpcard-x25519-agent" qui va réaliser l'opération ECDH sur la clé, puis le retourner au programme en Go. Afin que le programme sache qu'il doit utiliser l'agent, ils ont décidé d'utiliser une clé privée "spéciale" à mettre dans la configuration afin de bien rediriger le flux du programme de base. Cette solution possède un désavantage majeur. Elle demande d'installer un client externe sur la machine et il faut en plus configurer celui-ci. Cela rend le produit final très fragile étant donné que tout repose sur cet agent et de plus l'expérience utilisateur est nettement moins bonne à cause de la configuration supplémentaire à réaliser.

Dans le cadre de ce projet, il ne sera pas possible d'utiliser une solution similaire. Le but est de garder l'application simple et compacte, afin que celle-ci tienne en un seul exécutable. De plus l'utilisation d'un autre langage que le Rust pour le client ne fait pas de sens, car le choix du Rust vient justement du fait que ce langage est robuste et sécurisé.

2.6 Problèmes potentiels

Je voulais accorder une courte section aux problèmes potentiels auxquels j'ai pensé durant la phase précédant la réalisation. Le but ici n'est pas d'apporter une réponse à ces problèmes, mais simplement de les relever et de les anticiper. Les réponses viendront au fur et à mesure que le projet avance, ainsi que lorsque je me documenterai plus en profondeur sur les technologies à utiliser.

2.6.1 Compatibilité

L'un des points critiques de ce projet est de savoir à quel point il va être possible de rendre compatible le système de smartcard avec l'implémentation WireGuard choisie. Ce problème pourrait survenir par exemple s'il s'avérait être impossible de faire correspondre les résultats de l'échange ECDH de WireGuard et du périphérique. La faisabilité du projet repose grandement sur ce point, il sera alors important de rapidement déterminer à l'aide d'un programme de test que ce point ne freinera pas le projet.

2.6.2 Performance et robustesse

Un second point concerne plutôt la manière dont va réagir le programme après la modification. Au niveau des performances, il ne devrait pas y avoir trop d'impacts, car les smartcards réalisent les opérations de manière presque instantanée. En revanche, la robustesse du programme est un sujet beaucoup plus sensible. Il faudra effectuer des tests afin de vérifier ce qu'il se passe dans des situations pouvant compromettre le fonctionnement du programme. Je pense ici par exemple au cas où le périphérique est retiré durant une session en cours, si l'utilisateur ne touche pas son périphérique pour valider l'opération, etc.

2.6.3 Gestion du renouvellement de la session

Un dernier point qui rejoint légèrement le côté robustesse est de savoir ce qu'il va se passer lors du renouvellement du handshake. Est-ce que l'utilisateur va devoir toutes les cinq minutes devoir toucher son périphérique ? Est-ce qu'une validation initiale permet de renouveler autant de sessions que l'on veut, etc...

Chapitre 3

Périphérique sécurisé

Ce chapitre va permettre d'expliquer la solution qui a été choisie parmi celles trouvées lors de la réalisation de l'état de l'art. On va approfondir le fonctionnement du périphérique lui-même, ainsi que des caractéristiques qui vont permettre de réaliser le projet.

Dans une seconde partie, on présentera les caractéristiques pratiques de cette étape en parlant notamment du programme de test développé et de son fonctionnement.

Finalement, une dernière partie sera là pour aborder les problèmes rencontrés lors de la réalisation de cette partie du projet.

3.1 Solution choisie

3.1.1 Périphérique choisi

Dans l'état de l'art, deux types de périphériques se distinguaient pour ce projet : les périphériques classiques proposés par YubiKey ou NitroKey, et un périphérique plus flexible proposé par Tillitis.

Le choix final s'est porté sur les périphériques classiques, motivé par deux facteurs. Premièrement, l'utilisation d'un périphérique classique augmente la probabilité de trouver des personnes ayant déjà travaillé avec celui-ci et de disposer de plus de documentation en ligne. Deuxièmement, la solution proposée par Pro Custodibus démontrait une méthode fonctionnelle pour réaliser ce projet en utilisant ces périphériques.

Initialement, le choix se portait sur une YubiKey, car je possédais personnellement deux de leurs produits. Cependant, j'ai fait face à une limitation avec leur produit qui m'a forcé à changer pour une NitroKey, cette limitation est expliquée dans la section relatant les problèmes rencontrés (voir section 3.3.1). Le périphérique pour ce projet sera donc une

NitroKey, précisément une NitroKey A3 Mini.

3.1.2 Protocole choisi

Les smartcards possèdent plusieurs compartiments assignés à des fonctionnalités bien précises, telles que l'authentification FIDO2, la gestion des OTPs, PIV, OpenPGP, etc. Parmi ceux-ci, les deux compartiments qui pouvaient convenir à nos besoins étaient les compartiments OpenPGP et ceux pour les PIVs.

À l'origine, mon choix s'est porté sur les slots PIV en raison de leur simplicité d'utilisation et de l'existence de quelques bibliothèques et outils pour communiquer avec eux. Néanmoins, en raison d'un problème survenu lors de la mise en oeuvre de cette solution, j'ai finalement opté pour les compartiments OpenPGP. Ce choix s'est révélé plus adapté à notre situation pour plusieurs raisons. Tout d'abord, OpenPGP est compatible avec une large gamme de périphériques, y compris ceux de NitroKey. De plus, plusieurs bibliothèques existent pour dialoguer avec des smartcards utilisant OpenPGP, ce qui simplifie grandement l'intégration. Enfin, des implémentations similaires, comme celle proposée par Pro Custodibus, utilisent également OpenPGP, démontrant ainsi sa fiabilité et son efficacité pour ce type de projet.

3.1.2.1 OpenPGP et son fonctionnement

Cette sous-section va permettre d'expliquer le fonctionnement d'OpenPGP, ainsi que de faire le lien avec les smartcards et ECDH.

OpenPGP

OpenPGP [16] est un protocole de cryptographie asymétrique principalement utilisé pour envoyer des emails chiffrés. Cependant, il est également capable de chiffrer d'autres types de données et de réaliser diverses opérations, telles que la signature numérique et l'authentification.

Lors de la création d'une paire de clés avec OpenPGP, il est possible de générer des sous-clés spécifiquement assignées à différentes tâches. Cela permet une gestion granulaire des permissions et renforce la sécurité. Par exemple, une clé principale peut être utilisée uniquement pour la signature et la certification, tandis que des sous-clés peuvent être dédiées au chiffrement des données ou à l'authentification. Voici un exemple d'une clé privée OpenPGP avec ses sous-clés :

Code Source : Exemple de clés privées OpenPGP

```
1 sec>  ed25519/9EEF9F41295F8FC4 2024-05-08 [SC]
2       A36CE5CDA8A536B251D4AE3C9EEF9F41295F8FC4
3       Card serial no. = 000F A470431D
```

```
4 uid                 [ultimate] Alexis Martins (TB Nitro) <alexis@...>
5 ssb>  cv25519/BC263C3E38A4B267 2024-05-08 [E]
```

On retrouve sur cet exemple la clé principale notée à côté du 'sec>', celle-ci peut être utilisée pour la signature (S) et pour la certification (C).

OpenPGP avec les smartcards

La plupart des smartcards du marché, telles que YubiKey ou NitroKey, offrent la possibilité de stocker des clés privées générées avec OpenPGP directement sur la carte. Cela apporte un niveau de sécurité supplémentaire en assurant que les clés privées ne quittent jamais le périphérique sécurisé, même en cas de compromission du système hôte.

Il est possible de dialoguer avec la carte via une interface définie dans le document "Function Specification of the OpenPGP application on ISO Smart Card" [29]. Cette spécification détaille les commandes et les réponses standardisées pour l'interaction avec les clés OpenPGP sur les smartcards. Le but étant de pouvoir envoyer à la carte les opérations à réaliser (chiffrement, signature, etc.) afin que celles-ci soient réalisées sur la carte et que nous ayons uniquement accès au résultat. La plupart des langages de programmation possèdent des librairies qui rendent l'utilisation de cette spécification et des smartcards plus simple pour les développeurs.

ECDH avec OpenPGP

Comme vu précédemment, l'algorithme qui nous importe afin de réaliser ce projet est ECDH. Les courbes elliptiques sont supportées par OpenPGP et les smartcards. Il est en effet possible d'utiliser des algorithmes tels que X25519 pour le chiffrement et Ed25519 pour la signature.

En lisant la RFC 6637 [20] qui traite des courbes elliptiques sur OpenPGP, on peut apprendre que l'opération réalisée par ECDH pour obtenir une clé partagée est un déchiffrement. Ainsi, nous recevons la clé publique V de notre pair qui est égal à $v \cdot G$ avec v sa clé privée et G un point sur la courbe. Pour calculer le secret partagé S, il nous suffit de faire une exponentiation (opération de déchiffrement) par notre clé privée r, ainsi $S = r \cdot V$.

3.1.3 Librairies utilisées

Comme vu dans la section précédente, il existe un standard expliquant la manière dont il faut communiquer avec les smartcards. Cependant, il serait complexe et peu pratique de directement devoir dialoguer en utilisant celui-ci.

En Rust, il existe de multiples librairies permettant de faire cet interfaçage, que ça soit avec OpenPGP ou les smartcards. Toutefois deux d'entre elles ressortent plus que les autres, ce sont openpgp-card et openpgp-card-sequoia.

openpgp-card [31] est une implémentation fidèle de la spécification OpenPGP pour les smartcards. Cette bibliothèque utilise des structures de données simples et fournit des fonctions qui réalisent des actions spécifiques conformément à la spécification. Elle permet une interaction directe avec les smartcards, offrant une couverture complète des fonctionnalités définies par le standard.

openpgp-card-sequoia [32] est une bibliothèque de plus haut niveau qui repose sur openpgp-card. Elle n'implémente pas directement les fonctionnalités d'OpenPGP, mais celles de SequoiaPGP, une version d'OpenPGP développée en Rust, qui met l'accent sur la sécurité et la confidentialité. Cette bibliothèque abstrait davantage les détails de bas niveau, simplifiant ainsi l'utilisation des smartcards tout en intégrant les fonctionnalités avancées de SequoiaPGP.

Le choix final s'est porté sur openpgp-card, car sa simplicité que ça soit dans l'utilisation des types ou dans les fonctions permet d'avoir une meilleure gestion de ce qu'il se passe. Je pense que rester aussi avec une librairie plus bas niveau va permettre de mieux se raccrocher à la librairie utilisée par l'implémentation de WireGuard sans avoir des problèmes de conversion d'un type à un autre, etc.

Il est également important de préciser que openpgp-card ne contient pas directement le code pour communiquer avec les smartcards. Pour établir la communication avec les cartes, openpgp-card s'appuie sur des implémentations de traits comme CardBackend et CardTransaction. Par exemple, la crate card-backend-pcsc [30] fournit des implémentations de ces traits en utilisant le wrapper Rust pour PC/SC, un standard d'interaction avec les smartcards et les lecteurs de cartes.

3.2 Implémentation

Le but de cette partie était de prouver que le projet était réalisable. Il est important avant de se lancer dans la modification de l'implémentation choisie de démontrer que les librairies utilisées par la smartcard sont compatibles avec les librairies cryptographiques utilisées par l'implémentation.

3.2.1 Préparation de la clé

Avant de se lancer dans la partie implémentation, il faut passer par une phase de préparation où l'on va installer la clé privée OpenPGP sur la smartcard. Pour générer la paire de clés, il est possible de le faire directement sur la clé ou en les générant localement dans un premier temps, puis en les poussant sur la smartcard. C'est cette deuxième méthode que j'ai préféré utiliser pour le projet, afin de toujours avoir une copie en local des clés privées pour faciliter les tests.

Pour faire la génération, il est possible d'utiliser l'utilitaire 'gpg' installé par défaut sur les systèmes UNIX. Il suffit de générer un trousseau de clé de type 'ECC' et plus spécifiquement Curve25519 comme sur l'exemple ci-dessous.

Code Source : Génération d'une paire de clé avec OpenPGP

```
$ gpg --expert --full-gen-key
gpg (GnuPG) 2.2.27; Copyright (C) 2021 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
   (7) DSA (set your own capabilities)
   (8) RSA (set your own capabilities)
   (9) ECC and ECC
  (10) ECC (sign only)
  (11) ECC (set your own capabilities)
  (13) Existing key
  (14) Existing key from card
Your selection? 9
Please select which elliptic curve you want:
   (1) Curve 25519
   (3) NIST P-256
   (4) NIST P-384
   (5) NIST P-521
   (6) Brainpool P-256
   (7) Brainpool P-384
   (8) Brainpool P-512
   (9) secp256k1
Your selection? 1
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y

You need a user ID to identify your key; the software constructs the user ID
from the Real Name, Comment and Email Address in this form:
    "Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name: votre_nom
E-mail address: votre_adresse@mail.com
```

```
Comment: un_commentaire
You selected this USER-ID:
    "votre_nom (un_commentaire) <votre_adresse@mail.com>"

Change (N)ame, (C)omment, (E)-mail or (O)kay/(Q)uit? O
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilise the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilise the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: key 55E869A060E07CD8 marked as ultimately trusted
gpg: revocation certificate stored as
  ↪  '/home/alexis/.gnupg/openpgp-revocs.d/561F4B3927CFB79C5071B5C455E869A060E07CD8.rev'
public and secret key created and signed.

pub   ed25519 2024-05-14 [SC]
      561F4B3927CFB79C5071B5C455E869A060E07CD8
uid                      votre_nom (un_commentaire) <votre_adresse@mail.com>
sub   cv25519 2024-05-14 [E]
```

Une fois cela fait, on peut récupérer l'identifiant de la clé en utilisant la commande de 'gpg' pour récupérer les clés sur la machine. Il est important de copier l'identifiant de la clé de chiffrement.

Code Source : Clés OpenPGP disponibles sur le système

```
$ gpg --list-secret-keys --keyid-format=long
/home/alexis/.gnupg/pubring.kbx
------------------------------
sec   ed25519/55E869A060E07CD8 2024-05-14 [SC]
      561F4B3927CFB79C5071B5C455E869A060E07CD8
uid                 [ultimate] votre_nom (un_commentaire) <votre_adresse@mail.com>
ssb>  cv25519/CFE0EE6C9CB05DF3 2024-05-14 [E] --> Cet ID
```

Finalement, en utilisant encore l'utilitaire 'gpg', il est possible de pousser la clé de chiffrement dans son slot respectif sur la smartcard.

Code Source : Externaliser la clé privée sur une smartcard

```
$ gpg --edit-key CFE0EE6C9CB05DF3
  ↪  130
gpg (GnuPG) 2.2.27; Copyright (C) 2021 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

```
5
6 Secret key is available.
7
8 sec  ed25519/55E869A060E07CD8
9      created: 2024-05-14  expires: never       usage: SC
10     trust: ultimate      validity: ultimate
11 ssb  cv25519/CFE0EE6C9CB05DF3
12     created: 2024-05-14  expires: never       usage: E
13 [ultimate] (1). votre_nom (un_commentaire) <votre_adresse@mail.com>
14
15 gpg> key 1
16
17 sec  ed25519/55E869A060E07CD8
18     created: 2024-05-14  expires: never       usage: SC
19     trust: ultimate      validity: ultimate
20 ssb* cv25519/CFE0EE6C9CB05DF3
21     created: 2024-05-14  expires: never       usage: E
22 [ultimate] (1). votre_nom (un_commentaire) <votre_adresse@mail.com>
23
24 gpg> keytocard
25 Please select where to store the key:
26    (2) Encryption key
27 Your selection? 2
28
29 Replace existing key? (y/N) y
30
31 sec  ed25519/55E869A060E07CD8
32     created: 2024-05-14  expires: never       usage: SC
33     trust: ultimate      validity: ultimate
34 ssb* cv25519/CFE0EE6C9CB05DF3
35     created: 2024-05-14  expires: never       usage: E
36 [ultimate] (1). votre_nom (un_commentaire) <votre_adresse@mail.com>
37
38 gpg> save
```

Il reste important de préciser qu'ici la création de la clé a été faite dans un premier temps sur la machine, puis celle-ci a été transférée sur la carte, mais ce n'est pas la méthode recommandée. Dans un environnement réel où la sécurité est un facteur clé, le mieux est de générer directement la clé sur la carte, ainsi personne ne pourrait copier la clé depuis la machine. Dans mon cas, la clé était générée sur la machine surtout pour des raisons de test afin de vérifier que les valeurs obtenues étaient bonnes.

3.2.2 Réalisation d'un programme de test

Afin de valider le fonctionnement de ce programme de test, il va falloir obtenir le même résultat pour les deux cas suivants :

- Récupérer la clé publique liée à la clé privée présente sur la smartcard et faire l'ECDH avec une clé privée générée par la librairie directement dans le code
- Récupérer une clé publique générée avec la librairie cryptographique de l'implémentation WireGuard et réaliser l'ECDH sur la smartcard

Si ces deux cas donnent le même résultat (par exemple le même tableau de bytes), alors on pourra affirmer que l'utilisation d'une smartcard ou l'utilisation de la librairie cryptographique de base est interchangeable. Il sera donc possible dans le code final d'ajouter un cas supplémentaire lors de l'opération ECDH dans le cas où l'utilisateur utilise une smartcard.

Dans le main, je prépare simplement ma clé et celles de mon pair afin de pouvoir les utiliser dans les fonctions d'échange de clés. On remarque que j'appelle une fonction récupérant la clé publique liée à la clé privée présente sur la smartcard, celle-ci ne sera pas détaillée ici. Par contre son analyse sera faite lors de son implémentation dans WireGuard avec les autres fonctions (voir section 4.2.2). On peut voir ensuite, que j'appelle les fonctions qui vont créer le secret partagé, celles-ci sont détaillées plus bas dans ce rapport. Je ne m'attarde pas maintenant sur les points spécifiques à la librairie cryptographique utilisée pour ce petit programme, car lorsque j'aborderai l'implémentation WireGuard choisi, un paragraphe y sera dédié.

Code Source : Main du programme de test

```
1  fn main() -> Result<()> {
2
3      // Création d'une paire de clés qui représente celle d'un pair dans la
   ↪  configuration WireGuard
4      println!("Creating keys");
5      let (peer_b_private, peer_b_public) = generate_key();
6      let peer_b_public_bytes = peer_b_public.as_bytes().to_vec();
7
8      // Récupération de la clé publique liée à la clé privée présente sur la smartcard
9      let my_public_key = get_public_from_smartcard()?;
10
11     // Création du secret partagé avec ECDH
12     let shared_key_a = dalek_case(peer_b_private, my_public_key)?;
13     let shared_key_b = openpgp_case(peer_b_public_bytes)?;
14
15     println!("Shared key A (dalek)     : {:?}", shared_key_a);
16     println!("Shared key B (openpgp)   : {:?}", shared_key_b);
17
18     // Vérification que les deux clés partagées sont identiques
```

```
19      assert_eq!(shared_key_a, shared_key_b);
20
21      Ok(())
22  }
```

La première fonction qui est appelée (Dalek) est celle qui utilise une clé privée de la librairie cryptographique et la clé publique qui est stockée sur la smartcard. Celle-ci est simple, elle utilise simplement la fonction d'échange de clés Diffie-Hellman proposée par la librairie.

Code Source : Échange de clé utilisant la fonction de la librairie Dalek

```
1 fn dalek_case(my_private_key: StaticSecret, peer_public: PublicKey) -> Result<[u8;
  ↪  32]> {
2
3       Ok(*my_private_key.diffie_hellman(&peer_public).as_bytes())
4   }
```

La seconde fonction est un peu plus compliquée, car c'est celle qui utilise la clé privée stockée sur la smartcard. Il faut, dans un premier temps, parcourir les smartcards connectées à la machine. Dans un cas d'utilisation classique, un utilisateur possède une seule carte connectée au même temps.

Une fois qu'une carte a été trouvée, on peut créer les objets nécessaires afin de pouvoir effectuer une opération avec celle-ci. Il faut utiliser un objet de type 'Transaction' pour pouvoir dialoguer avec la carte. C'est grâce à celui-ci qu'on va pouvoir appeler le déchiffrement avec la clé publique de notre pair. A noter qu'il faut fournir un PIN à la carte. Les cartes OpenPGP possèdent deux PINs, le premier est le 'admin PIN' qui nous permet d'effectuer des opérations de gestion de la carte comme ajouter ou supprimer des clés. Le second est le 'user PIN' qui nous permet de déverrouiller la carte afin de réaliser des opérations à l'aide de la clé privée étant stockée dessus, comme calculer un échange de clé Diffie-Hellman ou encore récupérer la clé publique associée.

Code Source : Échange de clé utilisant la smartcard

```
1 fn openpgp_case(peer_public_bytes: Vec<u8>) -> Result<[u8; 32]> {
2
3       // Utilisation de la librairie pour communiquer avec la carte
4       let backends = PcscBackend::cards(None)?;
5
6       // On itère sur les cartes disponibles
7       for b in backends.filter_map(|c| c.ok()) {
8
9           println!("Found card !");
10
```

```
11          // Création d'une transaction (structure qui permet d'effectuer des
    ↪  opérations sur la carte)
12          let mut card = Card::new(b)?;
13          let mut transaction = card.transaction()?;
14
15          // Permet de déverouiller l'accès à la carte
16          transaction.verify_pw1_user("123456".as_ref())?;
17
18          // On envoie à la carte la clé publique de notre pair afin qu'elle réalise le
    ↪  calcul de la clé partagée
19          match transaction.decipher(Cryptogram::ECDH(&*peer_public_bytes)) {
20              Ok(res) => {
21                  let res_bytes: [u8; 32] = res.try_into()
22                      .map_err(|_| anyhow!("Failed to convert: Vec length is not
    ↪  32"))?;
23                  return Ok(res_bytes);
24              },
25              Err(e) => {
26                  println!("ECDH failed: {}", e);
27                  return Ok([0; 32]);
28              }
29          }
30      }
31
32      Ok([0; 32])
33  }
```

Finalement, si le programme s'est déroulé sans erreurs, on peut retrouver deux tableaux de bytes identiques.

Code Source : Sortie finale du programme de test

```
1 Creating keys
2 Found card !
3 Shared key A (dalek)      : [186, 68, 126, 148, 194, 74, 57, 215, 1, 71, 14, 119, 235,
    ↪  16, 159, 92, 152, 78, 183, 240, 254, 22, 16, 162, 26, 52, 45, 227, 144, 153, 91,
    ↪  55]
4 Shared key B (openpgp)    : [186, 68, 126, 148, 194, 74, 57, 215, 1, 71, 14, 119, 235,
    ↪  16, 159, 92, 152, 78, 183, 240, 254, 22, 16, 162, 26, 52, 45, 227, 144, 153, 91,
    ↪  55]
5
6 Process finished with exit code 0
```

3.3 Problèmes rencontrés

3.3.1 Limitations des périphériques et transition vers OpenPGP

Comme mentionné précédemment, la solution initiale consistait à utiliser les slots PIV des YubiKeys pour stocker les clés nécessaires. Toutefois, j'ai rapidement rencontré un problème majeur : les slots PIV des YubiKeys ne permettaient pas de stocker des clés X25519, une exigence essentielle pour ce projet. Cette incompatibilité a été confirmée par le changelog de l'application graphique de gestion des PIVs, indiquant qu'une version spécifique du firmware était requise. Mes YubiKeys étant d'une version antérieure et Yubico ne permettant pas de mettre à jour le firmware pour des raisons de sécurité, cette option s'est avérée impraticable.

Code Source : Extrait du changelog trouvé sur l'application yubico-piv-tool

```
1 yubico-piv-tool (2.5.0) stable; urgency=medium
2
3   * ykpiv: cmd: ykcs11: Add support for RSA3072 and RSA4096 key types. Available in
    ↪  firmware 5.7.0 and newer
4   * ykpiv: cmd: Add support for ED25519 and X25519 key types. Available in firmware
    ↪  5.7.0 and newer
5   * ykpiv: cmd: Add support for deleting keys. Available in firmware 5.7.0 and newer
6   * ykpiv: cmd: Add support for moving keys between slots. Available in firmware
    ↪  5.7.0 and newer
```

Pour surmonter cette limitation, une commande pour une NitroKey a été passée, car les slots PIV de ce périphérique pouvaient accueillir des clés X25519. Cependant, en attendant la livraison de la NitroKey, j'ai exploré d'autres solutions et me suis tourné vers le protocole OpenPGP. Après quelques recherches et essais, j'ai constaté que le protocole OpenPGP était plus simple à implémenter grâce à l'existence de nombreuses bibliothèques pour interagir avec des smartcards.

En conséquence, la solution finale retenue pour le projet utilise les compartiments OpenPGP, offrant une meilleure compatibilité et une simplicité d'intégration. Cette transition a permis de résoudre les limitations initiales rencontrées avec les YubiKeys et de garantir la faisabilité du projet.

Chapitre 4

Implémentation de WireGuard

Dans cette partie, le but est d'expliquer la solution qui a été choisie pour l'implémentation WireGuard. Dans un premier temps, des détails seront donnés concernant le fonctionnement de l'implémentation dans son ensemble.

Dans une seconde partie, une explication détaillée des modifications apportées à l'implémentation sera donnée. Ces explications comporteront les raisons des modifications, ainsi que le code qui a été ajouté afin de faire fonctionner le programme.

Dans une dernière partie, les problèmes rencontrés durant cette partie seront abordés, ainsi que leurs solutions.

4.1 Solution choisie

Pour rappel, les trois solutions proposées dans l'état de l'art étaient BoringTun (CloudFlare), WireGuard-rs et la réalisation d'une implémentation personnalisée.

La solution retenue pour ce projet est BoringTun. Le point négatif de BoringTun par rapport aux autres implémentations est aussi sa force. Cette implémentation est un vrai programme complet et utilisable. Cela implique que le code est complexe et qu'il y a énormément de matière, mais la structure du code permet aussi de mieux se repérer et au final d'avoir un résultat propre.

Une autre force par rapport aux autres implémentations, c'est que de la documentation sur l'utilisation du logiciel de base existe. Ce qui permet de grandement simplifier les premières phases de mise en place et de prise en main du logiciel de base.

4.1.1 Fonctionnement

Cette sous-section va expliquer la manière dont BoringTun fonctionne à l'interne de façon haut niveau. Le principe est de comprendre l'organisation du code afin de mieux identifier les parties du code importantes pour la réalisation de ce projet.

Le code utilisé pour la réalisation de ce travail est pris directement de la branche master avec le numéro de commit 'f672bb6'. À noter qu'il n'est pas recommandé de se baser sur la version de l'application de la branche master, étant donné que celle-ci est en pleine restructuration. Mais aucun comportement inhabituel n'a été remarqué lors de l'exécution du programme.

Le code est séparé en deux sous-projets Rust, nommés respectivement 'boringtun-cli' et 'boringtun'. Le premier de ces deux projets représente l'application qui sera exécutée sur nos machines. C'est via cet exécutable que l'on va pouvoir lancer BoringTun et celui-ci utilisera toute l'implémentation de WireGuard qui a été réalisée dans le second projet. Il est en réalité uniquement composé d'un main faisant des appels aux fonctions du second projet afin d'initialiser les connexions et faire fonctionner l'application.

Comme dit ci-dessus, le second projet comporte toute la logique de WireGuard. Il est découpé en plusieurs modules et fichiers qui permettent de modéliser les différents objets utilisés dans WireGuard comme le handshake, les tunnels, etc. C'est principalement dans ce sous-projet qu'il va falloir apporter des modifications et c'est aussi celui-ci qui sera détaillé plus bas. Avant de rentrer plus en détails dans le but de chaque partie de ce code, voici une vue d'ensemble de celui-ci.

Code Source : Vue d'ensemble de l'organisation du code de BoringTun

```
.
.. device
... allowed_ips.rs
... api.rs
... dev_lock.rs
... drop_privileges.rs
... epoll.rs
... kqueue.rs
... mod.rs
... peer.rs
... tun_darwin.rs
... tun_linux.rs
.. ffi
... mod.rs
.. jni.rs
.. lib.rs
.. noise
... errors.rs
... handshake.rs
... mod.rs
```

```
21 ... rate_limiter.rs
22 ... session.rs
23 ... timers.rs
24 .. serialization.rs
25 .. sleepyinstant
26 ... mod.rs
27 ... unix.rs
28 ... windows.rs
29 .. wireguard_ffi.h
```

Le premier module fait référence à notre ‘device‘, il permet de modéliser notre configuration WireGuard sur notre machine. Ce module gère les pairs du réseau WireGuard, y compris les connexions et les états, et maintient une structure pour les adresses IP autorisées dans le tunnel. Il inclut aussi des mécanismes de gestion de la concurrence ou encore la gestion des entrées/sorties via des gestionnaires d'événements (epoll/kqueue).

Le module ‘ffi‘ qui signifie ‘Foreign function interface‘ permet, comme son nom l'indique, d'offrir la possibilité à d'autres langages d'intégrer BoringTun via cette interface. Dans le cadre de BoringTun, cette interface expose surtout certaines fonctions à destination de code écrit en C/C++.

Le module ‘noise‘ est responsable de l'implémentation des aspects cryptographiques du protocole WireGuard. Il gère les sessions cryptographiques et les états associés, et met en oeuvre la gestion du handshake essentiel pour établir des connexions sécurisées. C'est ce module qui met en place les fonctions afin de réaliser les échanges vus précédemment dans le Noise Protocole Framework.

Le module ‘sleepyinstant‘ s'assure que les opérations temporelles se passent de la meilleure manière possible en fonction de la plateforme sur laquelle le code est exécuté.

Il reste finalement plusieurs fichiers ne faisant partie d'aucun module, et dont le but est soit de fournir des fonctions outils pour BoringTun ou de fournir une interface pour du code Java.

Le module intéressant pour la réalisation de ce projet va surtout être ‘noise‘, car c'est principalement celui-ci qui s'occupe du fonctionnement du handshake. Il est donc essentiel de remplacer le fonctionnement actuel par celui prenant en compte les smartcards.

4.1.2 Cryptographie

Le but de cette section est d'aborder la librairie cryptographique utilisée pour le handshake au sein de BoringTun. Lors de la partie parlant du périphérique choisi, on a rapidement constaté que la librairie utilisée était ‘dalek‘, plus précisément x25519-dalek [7]. Cependant, le sujet des différentes structures et fonctions utilisées pour le handshake n'a pas été abordé.

La librairie Dalek utilise 5 structures différentes afin de modéliser les différents secrets et clés :

- **StaticSecret :** Secret longue durée, ou en d'autres termes, la clé privée longue durée de la configuration WireGuard.
- **PublicKey :** Clé publique longue durée que l'on transmet aux autres pairs du système.
- **SharedSecret :** Secret partagé suite à un échange de clés avec Diffie-Hellamn avec un nombre d'utilisations illimité.
- **EphemeralSecret :** Secret courte durée ou clé privée courte durée, permettant de créer uniquement un seul secret partagé avec Diffie-Hellman.
- **ReusableSecret :** Secret pouvant être utilisé plusieurs fois, mais n'est pas recommandé sauf pour certains cas spécifiques.

Les structures de type 'StaticSecret' possèdent une fonction qui nous intéresse pour le cadre de ce projet qui est la fonction 'diffie_hellman'. Cette fonction prend en paramètre une clé publique et produit en sortie une nouvelle instance de 'SharedSecret'. À noter que c'est le seul moyen de produire une instance de cette structure, mais on reviendra sur ce détail plus tard dans les problèmes rencontrés.

4.1.3 Utilisation

Afin de pouvoir utiliser BoringTun, il y a des étapes préliminaires à réaliser pour pouvoir lancer le programme correctement. Ces étapes concernent les systèmes d'exploitation Linux, étant donné que le matériel sur lequel le projet a été fait était un Ubuntu 22.04.4 LTS.

Il est aussi important de préciser qu'une marche à suivre est disponible sur leur GitHub afin d'aider les utilisateurs à lancer le programme. Il est possible d'installer l'application directement avec cargo afin de faciliter son utilisation. Dans les étapes ci-dessous, on utilisera directement le code source afin de pouvoir par la suite y apporter des modifications.

4.1.3.1 Mise à jour de wg-quick

Une étape importante qui sera plus longuement détaillée dans la partie discutant des problèmes rencontrés (voir section 4.4.1) est la modification du script wg-quick. Il est crucial d'apporter cette modification à ce script, sinon il n'est pas possible de lancer BoringTun.

4.1.3.2 Préparation d'une configuration WireGuard

Comme lors de l'utilisation de l'implémentation de base de WireGuard, il faut préparer un fichier de configuration que l'on va placer dans '/etc/wireguard/'. Dans le cadre de ce rapport, le fichier s'appellera 'wg0.conf'.

4.1.3.3 Appliquer une capabilty sur l'exécutable

BoringTun a besoin de la capability 'CAP_NET_ADMIN' afin de pouvoir réaliser des opérations liées au réseau et à sa gestion sans avoir besoin de lui donner tous les privilèges root.

Code Source : Ajouter la capability à l'exécutable BoringTun

```
1 $ sudo setcap cap_net_admin+epi /usr/bin/boringtun-cli
```

4.1.3.4 Lancement et arrêt du programme

Lorsque l'on lance BoringTun avec wg-quick, il ne faut pas oublier de préparer les variables d'environnement 'WG_QUICK_USERSPACE_IMPLEMENTATION' et 'WG_SUDO'. La première permet d'indiquer l'implémentation de WireGuard à utiliser, tandis que la deuxième permet à wg-quick d'utiliser les commandes avec des privilèges plus élevés si nécessaire.

Code Source : Code pour démarrer BoringTun sur Linux

```
1 $ sudo WG_QUICK_USERSPACE_IMPLEMENTATION="/usr/bin/boringtun-cli" WG_SUDO=1 wg-quick
  ↪  up wg0
2 [!] Found WG_QUICK_USERSPACE_IMPLEMENTATION. Use userspace implementation instead.
3 [#] /usr/bin/boringtun-cli wg0
4 BoringTun started successfully
5 [#] wg setconf wg0 /dev/fd/63
6 [#] ip -4 address add 172.16.0.2/32 dev wg0
7 [#] ip link set mtu 1420 up dev wg0
8 [#] ip -4 route add 172.16.0.1/32 dev wg0
9 [#] ip -4 route add 10.0.1.0/24 dev wg0
```

Si les étapes précédentes ont bien été réalisées, on peut voir un message annonçant le succès de l'opération. Pour correctement arrêter BoringTun, il suffit d'utiliser la commande standard pour arrêter wg-quick.

Code Source : Code pour arrêter BoringTun sur Linux

```
1 $ sudo wg-quick down wg0
2 [#] ip link delete dev wg0
```

4.2 Implémentation

Cette section va illustrer le code qui a été produit et les modifications qui ont été apportées à BoringTun afin de faire fonctionner la smartcard avec le code initialement présent.

4.2.1 Utilisation de la clé privée

Comme mentionné précédemment dans ce rapport, la partie importante pour ce projet était de remplacer la clé privée long terme, normalement stockée sur la machine, par celle présente sur la smartcard. Il fallait donc, dans un premier temps, chercher tous les endroits où on utilisait cette clé privée. J'ai trouvé trois endroits dans le code où il y avait un appel à la fonction 'diffie_hellman' qui concernaient la clé privée longue durée. Ils se situent tous les trois dans le fichier 'handshake.rs' qui est présent dans le module 'noise'.

La première modification intervient dans la fonction 'parse_handshake_anon', qui est appelée lorsque l'on reçoit un message d'handshake de la part d'un pair avec qui cela n'avait pas encore été fait. Un secret partagé est échangé afin de créer la clé finale pour déchiffrer la clé publique, chiffrée avec ChaCha20 envoyée par l'autre pair. On verra l'utilité de cette clé publique dans la section suivante.

La seconde modification vient lorsque l'on crée des structures 'NoiseParams', qui représentent tous les paramètres dont le Noise Protcol a besoin et qui est ensuite utilisée dans une autre structure nommée 'Handshake'. Celle-ci est importante, car si on crée le mauvais secret alors le handshake n'aura jamais lieu avec l'autre pair.

La troisième modification intervient dans la fonction 'set_static_private', qui permet de définir la clé privée pour les structures 'NoiseParams'.

Le quatrième endroit où cette modification s'applique est dans la fonction 'receive_handshake_initialization', la raison est similaire à la première modification, sauf qu'ici le handshake n'est pas anonyme, mais d'un pair connu.

La dernière modification concerne la fonction 'receive_handshake_response', qui fait exactement pareil que la modification précédente, dans le cas où nous sommes l'initiateur et non le répondeur.

Code Source : Modifications du fichier handshake.rs

```
1 // Ligne 348 :
2
3 // Avant
4 let ephemeral_shared = static_private.diffie_hellman(&peer_ephemeral_public);
5 // Après
6 let ephemeral_shared = extended_diffie_hellman(&static_private,
  ↪  &peer_ephemeral_public);
```

```
7
8  // Ligne 379 :
9
10 // Avant
11 let ephemeral_shared = static_private.diffie_hellman(&peer_ephemeral_public);
12 // Après
13 let ephemeral_shared = extended_diffie_hellman(&static_private,
   ↪  &peer_ephemeral_public);
14
15 // Ligne 406 :
16
17 // Avant
18 self.static_shared = self.static_private.diffie_hellman(&self.peer_static_public);
19 // Après
20 self.static_shared = extended_diffie_hellman(&self.static_private,
   ↪  &self.peer_static_public);
21
22 // Ligne 504 :
23
24 // Avant
25 let ephemeral_shared = self
26 .params
27 .static_private
28 .diffie_hellman(&peer_ephemeral_public);
29 // Après
30 let ephemeral_shared = extended_diffie_hellman(&self.params.static_private,
   ↪  &peer_ephemeral_public);
31
32 // Ligne 595 :
33
34 // Avant
35 let temp = b2s_hmac(
36     &chaining_key,
37     &self
38         .params
39         .static_private
40         .diffie_hellman(&unencrypted_ephemeral)
41         .to_bytes(),
42 );
43 // Après
44 extended_diffie_hellman(&self.params.static_private,
   ↪  &unencrypted_ephemeral).as_bytes(),
```

Maintenant que l'on a vu où s'appliquaient les modifications et pourquoi, on peut se pencher sur le contenu de la fonction qui remplace l'ancien Diffie-Hellman.

Code Source : Fonction réalisant la création d'un secret partagé prenant en compte les smartcards

```rust
pub fn extended_diffie_hellman(static_secret: &StaticSecret, peer_public_key: &PublicKey) -> SharedSecret {

    // Vérifie si la clé privée annoncée dans la configuration correspond à la valeur indiquant qu'une smartcard est utilisée
    // Si ce n'est pas le cas, alors on utilise le secret statique pour effectuer le calcul de la clé partagée
    if *static_secret.as_bytes() != SMARTCARD_INDICATOR {
        return static_secret.diffie_hellman(&peer_public_key);
    } else {
        let backends_result = PcscBackend::cards(None)
            .expect("Failed to get backends");

        // Cherche la carte connectée afin d'effectuer le calcul de la clé partagée
        for b in backends_result.filter_map(|c| c.ok()) {
            let mut card = Card::new(b)
                .expect("Card creation failed");

            let mut transaction = card.transaction()
                .expect("Transaction creation failed");

            // Utilise le code PIN de l'utilisateur pour déverrouiller la carte
            let _ = transaction.verify_pw1_user(get_user_pin().as_ref());

            // Effectue le calcul de la clé partagée
            let shared_secret_bytes = transaction.decipher(Cryptogram::ECDH(&*peer_public_key.as_bytes()))
                .expect("Diffie-Hellman failed");

            // Convertit le résultat en tableau d'octets
            let shared_secret_bytes_array: [u8; 32] = shared_secret_bytes.try_into()
                .expect("Failed to convert result to bytes");

            // Crée un objet SharedSecret à partir du tableau d'octets et le retourne
            return SharedSecret::from_bytes(shared_secret_bytes_array);
        }

        panic!("No valid card found");
    }
}
```

Cette fonction met simplement en pratique le programme de test créé dans le chapitre précédent. Il y a tout de même quelques différences afin qu'elle puisse s'intégrer avec le code de BoringTun.

Premièrement, il faut vérifier si l'utilisateur utilise ou non une smartcard. Pour cela, j'ai repris la méthode vue chez Pro Custodibus où ils utilisaient une valeur de clé privée impos-

sible pour signaler l'utilisation d'une smartcard. Ainsi, l'utilisateur a uniquement besoin de préciser la valeur sa clé privée, comme ci-dessous, pour que le système fonctionne avec sa carte.

Code Source : Fichier de configuration WireGuard indiquant l'utilisation d'une smartcard

```
1 [Interface]
2 ...
3 PrivateKey = AQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
4 ...
```

Si une smartcard est utilisée, le code est similaire au programme de test. On va chercher la carte parmi celles connectées et la déverrouiller avec le PIN de l'utilisateur. La gestion du PIN sera expliquée dans une section suivante (voir section 4.2.3).

Finalement, lorsque la smartcard est déverrouillée, il suffit de réaliser le calcul de la clé partagée. Étant donné que BoringTun utilise les SharedSecret, il faut transformer le tableau de bytes en cette structure. Une section des problèmes rencontrés (voir section 4.4.2) sera dédiée à la fonction 'from_bytes', car cette fonction n'existe pas de base dans la librairie Dalek.

Si une des étapes du processus pour calculer le secret partagé échoue, alors le programme est forcé à paniquer. Ce choix est le plus rationnel, afin d'éviter d'introduire des potentielles failles de sécurité ou simplement d'éviter des comportements inattendus.

4.2.2 Obtention de la clé publique

Un second aspect technique qui n'avait pas été abordé durant le programme de test est la récupération de la clé publique liée à la clé privée dans la smartcard. Cette étape est super importante dans le cas de WireGuard, car lorsque l'initiateur envoie le premier message de handshake au répondeur, il inclut sa propre clé publique chiffrée à une 'chaining_key' dont la clé publique du répondeur fait partie. Ainsi, le répondeur peut déchiffrer cette clé publique et vérifier qu'elle correspond bien à la clé publique de l'un de ses pairs dans le fichier de configuration. À noter que lorsque l'initiateur envoie sa clé publique chiffrée, celui-ci la dérive à partir de sa clé privée qui est présente dans le fichier.

Cette dérivation était donc problématique dans le système qui a été mis en place, car l'initiateur qui possède une smartcard va simplement dériver la valeur d'indication de présence d'une smartcard que l'on met dans le champ 'PrivateKey' du fichier de configuration. J'ai donc réalisé une fonction supplémentaire, qui dans le cas où l'utilisateur utilise une smartcard, récupère cette clé depuis la smartcard plutôt qu'avec la fonction utilisée initialement.

Ce changement a dû être apporté à trois fichiers différents dans lesquels on dérivait la clé publique.

La première modification est très importante, puisque c'est à ce moment dans le code que l'initiateur définit sa clé publique qu'il va ensuite envoyer aux autres pairs. Il va créer une structure 'Handshake' prenant en paramètre cette clé et c'est cette structure qui fait les appels aux fonctions gérant le handshake (initiation et réponse).

La seconde modification impacte la création du 'Device' qui représente la configuration WireGuard donnée à BoringTun. Lors du parsing de la configuration, il assigne la clé privée présente dans la configuration au device, mais il assigne aussi la clé publique en la dérivant.

La dernière modification intervient juste après la précédente. Lorsque le 'Device' est créé, on va aussi créer tous les pairs auxquels il est lié (via la configuration). Pour chacun de ces pairs, une structure 'Tunn' est créée, et lors de sa création, il faut aussi lui passer une clé privée et une clé publique. La structure s'assure qu'on lui passe une clé publique qui correspond bien à la clé privée utilisée pour dialoguer avec chaque pair.

Code Source : Modifications apportées afin de correctement dériver la clé publique avec une smartcard

```
// noise/mod.rs, ligne 202 :

// Avant
let static_public = x25519::PublicKey::from(&static_private);
// Après
let static_public = extended_public_key_derivation(&static_private);

// device/mod.rs, ligne 455 :

// Avant
let public_key = x25519::PublicKey::from(&private_key);
// Après
let public_key = extended_public_key_derivation(&private_key);

// noise/handshake.rs, ligne 400 :

// Avant
let check_key = x25519::PublicKey::from(&static_private);
// Après
let check_key = extended_public_key_derivation(&static_private);
```

On peut maintenant passer aux commentaires concernant la fonction qui était appelée ci-dessus.

Code Source : Fonction réalisant la dérivation de clé publique à partir d'une clé privée stockée sur la smartcard

```
pub fn extended_public_key_derivation(static_secret: &StaticSecret) -> PublicKey {
    if *static_secret.as_bytes() != SMARTCARD_INDICATOR {
```

```
3         return PublicKey::from(static_secret);
4     } else {
5         let backends_result = PcscBackend::cards(None)
6             .expect("Failed to get backends");
7
8         for b in backends_result.filter_map(|c| c.ok()) {
9             let mut card = Card::new(b)
10                .expect("Card creation failed");
11
12            let mut transaction = card.transaction()
13                .expect("Failed to create transaction");
14
15            if let Ok(public_key) = transaction.public_key(KeyType::Decryption) {
16                let public_key_str = public_key.to_string();
17                let public_key_slice = &public_key_str[public_key_str.len() - 64..];
18                let mut public_key_decoded = [0; 32];
19                decode_to_slice(public_key_slice, &mut
  ↪ public_key_decoded).expect("Failed to decode public key");
20                return PublicKey::from(public_key_decoded);
21            } else {
22                panic!("Failed to get public key");
23            }
24        }
25
26        panic!("No valid card found");
27    }
28 }
```

Le début est similaire à la fonction présentée précédemment. On cherche uniquement à accéder à une transaction avec laquelle on pourra appeler des actions à exécuter sur la clé. Il faut donc trouver si une clé est branchée, créer une transaction et nous pouvons faire nos actions.

La fonction ‘public_key‘ est présente dans la librairie OpenPGP qui est utilisée pour ce projet et celle-ci retourne une structure de type ‘PublicKeyMaterial‘. Il n’y a cependant pas de moyen direct de convertir ce type de structure vers une structure ‘PublicKey‘ utilisée par dalek. La solution qui me semblait la plus simple était de récupérer la chaîne de caractères qui est affichée en appelant la fonction ‘to_string‘ et de la parser. On voit ci-dessous la structure de la chaîne qui nous est retournée, les 64 derniers caractères de cette chaîne correspondent à la clé publique en hexadécimal.

Code Source : Sortie de la fonction to_string d’une structure PublicKeyMaterial

```
1 ECC [Cv25519 (ECDH)], data:
  ↪ E402A3D8E099E94478459FCC69996724D91C546CB99D0B8DE3665FAA0A7DAA30
```

Le parsing était assez simple, il suffisait de récupérer les 64 derniers caractères de la chaîne

et d'utiliser la fonction 'decode_to_splice' de la crate 'hex' afin d'obtenir un tableau de bytes. À partir de là, dalek offre une fonction qui permet de créer des 'PublicKey' à partir de tableaux de bytes.

4.2.3 Récupération du PIN de l'utilisateur

Une autre fonction ajoutée au code de BoringTun, est une fonction permettant de récupérer le 'user PIN' de l'utilisateur servant à déverrouiller la clé pour réaliser des opérations avec la clé privée. Cette fonction récupère et stocke le PIN de l'utilisateur dans une variable. Une seconde fonction permet d'accéder à la valeur stockée dans cette variable.

Il est donc impératif de demander à l'utilisateur son PIN et de le stocker s'il souhaite pouvoir utiliser le VPN sans avoir à le taper à chaque rafraîchissement du handshake (2 minutes par défaut sur WireGuard). La fonction permettant de saisir le PIN est appelée une fois dans le main du code de BoringTun-cli. La seconde fonction permettant de retourner ce PIN est appelée dans une des fonctions présentées précédemment.

Code Source : Fonctions de gestion du 'user PIN'

```
lazy_static::lazy_static! {
    static ref USER_PIN: Arc<Mutex<String>> = Arc::new(Mutex::new(String::new()));
}

pub fn set_user_pin() {
    let backends_result = PcscBackend::cards(None)
        .expect("Failed to get backends");

    // Vérifie si une carte existe et demande le code PIN de l'utilisateur
    if backends_result.filter_map(|c| c.ok()).next().is_some() {
        println!("Please enter your smartcard user PIN: ");
        let pin = read_password().expect("Failed to read PIN");
        let mut user_pin = USER_PIN.lock().unwrap();
        *user_pin = pin;
    }
}

fn get_user_pin() -> String {
    // Récupère le code PIN de l'utilisateur et le retourne
    let user_pin = USER_PIN.lock().unwrap();
    user_pin.clone()
}
```

Le PIN devient alors la seule valeur stockée en mémoire sur la machine, ce qui réduit grandement l'impact que peut avoir une attaque.

4.3 Tests

Cette section va aborder la manière dont le résultat final a été testé en détaillant l'environnement de test, ainsi que les tests effectués.

4.3.1 Environnement de test

L'environnement de test est au total composé de trois machines, dont deux qui auront besoin d'installer WireGuard afin de réaliser les tests.

4.3.1.1 Architecture de l'environnement

- **Machine Remote**
  - Description : Elle représente la machine externe qui souhaite se connecter à l'intérieur du réseau. Ma machine physique sera utilisée pour incarner cette machine.
  - IP publique : 172.16.77.1 (vmnet8 par défaut de VMware Workstation Pro)
  - IP privée : -
  - IP VPN : 172.16.0.2
- **Machine Server**
  - Description : Elle représente le serveur de façade du réseau auquel on souhaite se connecter. Elle est représentée par une machine virtuelle Ubuntu 22.04.5 LTS.
  - IP publique : 172.16.77.130 (vmnet8 par défaut de VMware Workstation Pro)
  - IP privée : 10.0.1.1
  - IP VPN : 172.16.0.1
- **Machine Private**
  - Description : Elle représente une machine se trouvant dans le réseau de la machine Server et qui n'est pas accessible par Remote sans le VPN. Elle est aussi représentée par une machine virtuelle Ubuntu 22.04.5 LTS.
  - IP publique : -
  - IP privée : 10.0.1.2
  - IP VPN : -

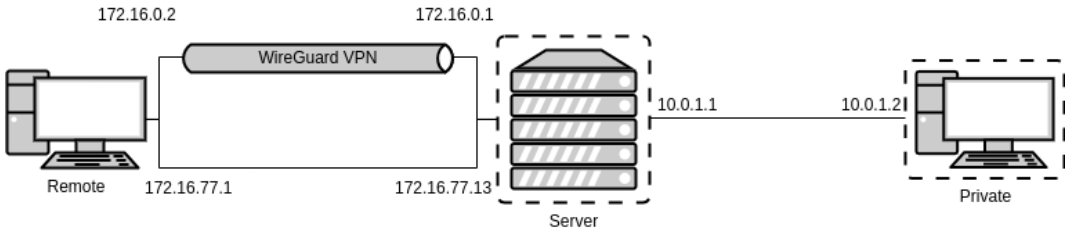


FIGURE 4.1 – Environnement de test

4.3.1.2 Configuration WireGuard des machines

Code Source : Configuration de la machine Remote

```
[Interface]
Address = 172.16.0.2/32
#PrivateKey = gGcbOpTbnZ3evDgwspuz5jqwuyudzD+ezoDfgMQA730= # Pas de smartcard
PrivateKey = AQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA= # Avec smartcard
ListenPort = 51820

[Peer]
PublicKey = pgeRxvy9IgJQ6yzeSOV7UrP6YhdZFvhYjVX4hx++QWk=
AllowedIPs = 172.16.0.1/32, 10.0.1.0/24
Endpoint = 172.16.77.130:51820
```

Code Source : Configuration de la machine Server

```
[Interface]
Address = 172.16.0.1/32
PrivateKey = oDfGXroIJ+YC/yJ6TtrBOPQofMS3f636b+xcZvupUkY=
ListenPort = 51820

[Peer]
#PublicKey = UgnBv+hp3lP7NeJeeaypMs1z6CDAy6cpkDhss4H4xA8= # Normal
PublicKey = 5AKj2OCZ6UR4RZ/MaZlnJNkcVGy5nQuN42ZfqgP9qjA=  # NitroKey
AllowedIPs = 172.16.0.2/32
Endpoint = 172.16.77.1:51820
```

4.3.2 Tests effectués

Table 4.1 : Résultats des tests

Description du test	Résultat attendu	Statut
Remote <-> Server (Via IP publique)	Ping réussi	OK
Server <-> Private (Via IP privée)	Ping réussi	OK
Remote <-> Server (Via IP VPN)	Ping réussi	OK
Remote <-> Server (Ping adr. privée du serveur)	Ping réussi	OK
Remote <-> Private (Ping adr. privée de la machine)	Ping réussi	OK
Application fonctionnelle avec la smartcard	Application fonctionne	OK
Application fonctionnelle sans la smartcard	Application fonctionne	OK
Utilisateur rentre le bon PIN pour la smartcard	Application fonctionne	OK
Utilisateur rentre le mauvais PIN pour la smartcard	Application ne fonctionne pas	OK
Rekeying fonctionnel toutes les deux minutes	Application continue de fonctionner	OK
Retirer la carte durant la session	Application fonctionne tant que le rekeying n'a pas été demandé et ne fonctionne plus après	OK
Smartcard sans bouton physique	Il est possible d'utiliser BoringTun correctement	OK
Smartcard avec bouton physique	Il est possible d'utiliser BoringTun correctement	OK

Deux informations importantes sont à relever dans ce tableau. Le rekeying est une opération qui a lieu toutes les deux minutes afin de rafraîchir la clé utilisée pour la session. La clé utilisée pour le test sur les smartcards avec bouton physique était une YubiKey 5 et elle ne demande en aucun cas à l'utilisateur d'appuyer sur le bouton.

4.4 Problèmes rencontrés

Durant la mise en place de l'environnement de test, ainsi que lors de l'implémentation, des problèmes sont survenus. Cette section permet de détailler la nature du problème, ainsi que la solution ayant permis de le résoudre.

4.4.1 Modification de wg-quick

Lorsque j'ai essayé de lancer pour la première fois BoringTun avec wg-quick au sein de l'environnement de test, celui-ci ne se lançait pas. Pourtant, toutes les instructions présentes sur le GitHub de BoringTun avaient bien été suivies. C'est en cherchant dans les issues du projet que j'ai pu trouver la solution [15] à mon problème.

Afin de pouvoir utiliser BoringTun comme implémentation userspace, il a fallu modifier le programme wg-quick. Le code de base présent dans wg-quick fait en sorte de toujours utiliser l'implémentation kernel de WireGuard si celle-ci est disponible. Seulement dans le cas où celle-ci n'est pas disponible, alors wg-quick utilisera une implémentation en userspace.

Code Source : Fonction de base présente dans le fichier wg-quick

```
1 add_if() {
2         local ret
3         if ! cmd ip link add "$INTERFACE" type wireguard; then
4                 ret=$?
5                 [[ -e /sys/module/wireguard ]] || ! command -v
                  ↪  "${WG_QUICK_USERSPACE_IMPLEMENTATION:-wireguard-go}" >/dev/null
                  ↪  && exit $ret
6                 echo "[!] Missing WireGuard kernel module. Falling back to slow
                  ↪  userspace implementation." >&2
7                 cmd "${WG_QUICK_USERSPACE_IMPLEMENTATION:-wireguard-go}" "$INTERFACE"
8         fi
9 }
```

Il faut alors remanier la vérification afin de vérifier, dans un premier si l'utilisateur a souhaité utiliser une implémentation en userspace. Si ce n'est pas le cas, alors on peut appeler le code de base se chargeant d'utiliser l'implémentation kernel.

Code Source : Fonction modifiée dans le fichier wg-quick

```
1 add_if() {
2         local ret
3         if [ ! -z "${WG_QUICK_USERSPACE_IMPLEMENTATION}" ]; then
4                 echo "[!] Found WG_QUICK_USERSPACE_IMPLEMENTATION. Use userspace
                  ↪  implementation instead." >&2
5                 cmd "${WG_QUICK_USERSPACE_IMPLEMENTATION:-wireguard-go}" "$INTERFACE"
```

```
6         elif ! cmd ip link add "$INTERFACE" type wireguard; then
7                 ret=$?
8                 [[ -e /sys/module/wireguard ]] || ! command -v
                  ↪  "${WG_QUICK_USERSPACE_IMPLEMENTATION:-wireguard-go}" >/dev/null
                  ↪  && exit $ret
9                 echo "[!] Missing WireGuard kernel module. Falling back to slow
                  ↪  userspace implementation." >&2
10                cmd "${WG_QUICK_USERSPACE_IMPLEMENTATION:-wireguard-go}" "$INTERFACE"
11        fi
12 }
```

Une fois cette modification apportée, il est possible de suivre les instructions mises à disposition sur le GitHub de BoringTun pour lancer le programme avec wg-quick.

4.4.2 Modification de x25519-dalek

Une seconde difficulté est survenue lorsque j'ai voulu faire l'intégration entre le programme de test réalisé dans le chapitre précédent avec le code de BoringTun. Je pensais qu'en ayant un tableau de bytes en ma possession, il allait être très facile de créer les structures dalek utilisées par BoringTun. Cependant, il se trouve qu'il n'est pas possible de créer une structure 'SharedSecret' à partir d'un tableau de bytes. Je n'ai pas réussi à trouver de raison pour ce choix dans la documentation, mais je suppose que c'est pour des raisons de sécurité, afin de s'assurer que seule la fonction 'diffie_hellman' des librairies dalek puisse en créer.

J'ai donc dû apporter moi-même une modification à la librairie 'x25519-dalek' afin d'ajouter une fonction 'from_bytes' à la structure 'SharedSecret'. Cette fonction permettra de créer un 'SharedSecret' lorsque la smartcard sera utilisée et que celle-ci créera un tableau de bytes lors de l'opération Diffie-Hellman.

Code Source : Fonction ajoutée à la crate x25519-dalek pour les SharedSecret

```
1 impl SharedSecret {
2
3     ...
4     /// Create a SharedSecret from a byte array
5     pub fn from_bytes(bytes: [u8; 32]) -> Self {
6         SharedSecret(MontgomeryPoint(bytes))
7     }
8     ...
9 }
```

Chapitre 5

Performances

Dans ce dernier chapitre, le but est de traiter la partie optionnelle ayant été fixée dans le cahier des charges. Le but est d'explorer et de comprendre le fonctionnement du kernel Linux au niveau réseau, ainsi que de faire un parallèle avec WireGuard et des interfaces TUN/TAP.

Dans un second temps, les limitations de ces interfaces seront abordées et certaines solutions permettant d'outrepasser celles-ci seront proposées.

Finalement, une dernière section illustrera de façon pratique sous forme d'un programme de test la solution la plus prometteuse pour répondre aux problèmes énumérés dans les sections précédentes.

5.1 Fonctionnement du réseau au niveau kernel

Comme expliqué ci-dessus, cette première section vise à apporter toutes les connaissances nécessaires sur l'état actuel du fonctionnement du réseau au sein de Linux. Une vue assez haut niveau sera gardée étant donné que le but de ce chapitre n'est pas de connaître en détail le fonctionnement du kernel, mais uniquement de connaître certains principes essentiels pour la suite. Les informations utilisées pour cette section ont été tirées des articles rédigés par Amr ElHusseiny [13] et Sergey Klyaus [21].

5.1.1 Stack réseau du kernel

Dans cette première sous-section, on va retrouver les quelques structures primordiales au fonctionnement du réseau au sein du kernel Linux, ainsi que le lien entre celles-ci.

5.1.1.1 Ring buffers

Ils sont utilisés par les cartes réseau pour stocker temporairement les paquets reçus avant qu'ils ne soient traités par le processeur. Ils sont au nombre de deux, un pour la réception (Rx buffer) et un second pour la transmission (Tx buffer). Ils sont alloués par le processeur au démarrage du kernel et leur adresse est passée aux cartes réseau afin qu'elles puissent les utiliser. Ces buffers fonctionnent en utilisant la mémoire DMA (Direct Memory Access), permettant aux cartes réseau de déposer les paquets directement en mémoire sans demander d'interruption au processeur. Par exemple dans le cas d'un envoi de paquet, l'application va déposer le paquet en queue du buffer et la carte réseau récupère les paquets en tête pour les envoyer, comme on le voit sur l'image ci-dessous.

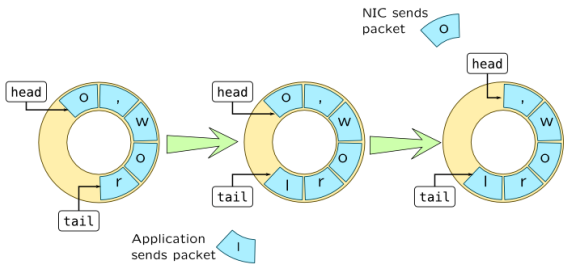


FIGURE 5.1 – Schéma des ring buffers tiré du blog de Sergey Klyaus [21]

5.1.1.2 Socket buffers

Ces structures (aussi appelées 'sk_buff') permettent de représenter un paquet réseau au sein du kernel jusqu'à ce qu'il soit traité par celui-ci. Ces buffers sont séparés en deux parties, les données stockent les headers Ethernet, IP, TCP/UDP, ainsi que les données réelles transportées par le paquet. Ainsi que des métadonnées stockent des pointeurs vers les différentes valeurs stockées par les données ou encore des informations sur l'interface utilisée par le paquet. Ci-dessous, une représentation simplifiée de cette structure avec les champs importants stockés dans la partie des données.

FIGURE 5.2 – Schéma des socket buffers tiré du blog de Sergey Klyaus [21]

Voici une courte explication pour chaque champ :

- **next et prev** : Pointeurs pour créer une liste doublement chaînée de buffers.
- **tstamp** : Horodatage du paquet.
- **sk** : Pointeur vers la structure socket associée. (voir suite du rapport)
- **dev** : Pointeur vers la structure net_device associée. (voir suite du rapport)
- **len** : Longueur des données du paquet.
- **data** : Pointeur vers les données du paquet.

5.1.1.3 net_device

Cette structure permet de représenter une interface réseau dans le noyau Linux. Elle contient des informations essentielles sur l'interface, comme son nom (name), ses identifiants (ifindex et iflink), et ses statistiques (stats). Cette structure est cruciale pour la gestion et la configuration des interfaces réseau.

5.1.1.4 Interruptions

Les interruptions sont utilisées pour arrêter le processeur de ce qu'il est en train de faire et lui faire traiter une tâche urgente. Les interruptions sont regroupées selon deux catégories principales :

- **Interruptions matérielles** : Elles sont générées par le matériel pour signaler des événements, tels que l'arrivée d'un paquet réseau. Elles interrompent immédiatement le processeur pour traiter l'événement, ce qui est coûteux en termes de performances. Pour atténuer cette charge, le gestionnaire d'interruptions masque celles-ci après la première utilisation, et le pilote de la carte réseau commence à utiliser les interruptions logicielles.
- **Interruptions logicielles** : Elles décomposent le traitement des paquets en plusieurs étapes gérables. Après la réception initiale d'un paquet via une interruption matérielle, les interruptions logicielles prennent le relais pour traiter les paquets en file d'attente de manière asynchrone, ce qui minimise la latence et équilibre la charge du processeur.

5.1.1.5 NAPI (New API)

C'est un mécanisme utilisé dans les pilotes de carte réseau pour améliorer les performances en réduisant les interruptions matérielles. Il utilise le polling pour traiter les paquets en attente, permettant ainsi de traiter plusieurs paquets à la fois et de diminuer la surcharge des interruptions, tout en équilibrant la charge du processeur et en minimisant la latence.

Le polling est la vérification régulière par le processeur des cartes réseau pour traiter les paquets en attente afin de diminuer les interruptions.

5.1.1.6 Vue d'ensemble

Voici une explication tirée du blog de Amr ElHusseiny [13] qui reprend toutes les structures vues précédemment et qui explique le lien entre elles.

1. Lors du démarrage du noyau, le processeur alloue les ring buffers et crée des descripteurs de fichiers.

2. Le processeur informe la carte réseau que de nouveaux descripteurs ont été créés pour que celle-ci puisse les utiliser.

3. La mémoire DMA (Direct Memory Access) récupère les descripteurs.

4. Un paquet arrive à la carte réseau.

5. La DMA écrit le paquet dans le buffer Rx.

6. La carte réseau informe le pilote, qui informe à son tour le processeur que du nouveau trafic est prêt à être traité en utilisant une interruption matérielle.

7. Après la première interruption matérielle, le gestionnaire d'interruptions la masque et, à la place, le pilote utilise des interruptions logicielles.

8. L'interruption logicielle réveille le sous-système NAPI, qui appelle la fonction de polling du pilote de la carte réseau.

9. Le processeur traite le paquet et ses données.

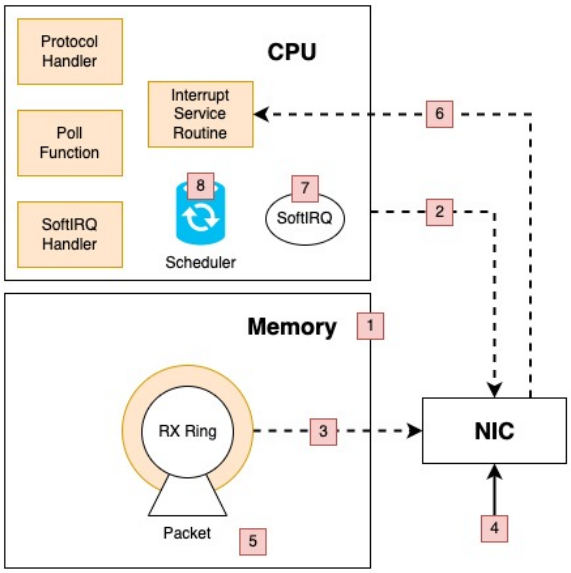


FIGURE 5.3 – Schéma du traitement d'un paquet dans le kernel tiré du blog de Amr ElHusseiny [13]

5.1.2 Implémentation des sockets

Les sockets servent de point de communication entre les applications et le noyau pour la transmission des données réseau. Ils permettent aux applications d'envoyer et de recevoir des paquets de données en utilisant des appels système.

Les sockets dans le noyau Linux sont représentés par deux structures principales : socket et sock. La structure socket contient des informations générales sur le socket, telles que son état (state), son type (type), le pointeur de fichier associé (file), et les opérations possibles (ops). La structure sock, quant à elle, stocke des informations spécifiques au réseau, comme les adresses locales (skc_rcv_saddr) et distantes (skc_daddr), ainsi que les ports (skc_dport). Ces deux structures sont essentielles pour gérer les connexions réseau et le transfert de données.

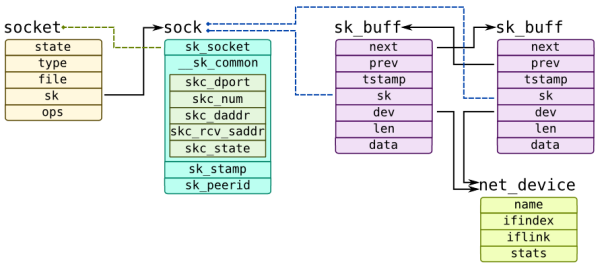


FIGURE 5.4 – Schéma global du lien entre les différentes structures tiré du blog de Sergey Klyaus [21]

La création et la gestion des sockets passent par plusieurs appels système, tels que socket(), bind(), listen() et accept(). Lorsqu'un socket est créé, le noyau alloue des ressources et initialise les structures socket et sock. Par exemple, l'appel socket() initialise un nouveau socket, bind() associe le socket à une adresse et un port locaux, listen() met le socket en mode écoute, et accept() accepte les connexions entrantes. Ces opérations permettent au noyau de gérer efficacement les connexions réseau et les descripteurs de fichiers associés.

Lorsqu'un paquet arrive à une interface réseau (net_device), il est encapsulé dans un sk_buff et associé à un socket (sock). Le sk_buff contient des pointeurs vers les structures sock et net_device, permettant au noyau de déterminer l'application destinataire. Les données sont ensuite ajoutées à la file d'attente de réception du socket et récupérées par l'application via des appels système comme recv().

5.1.3 Interfaces TUN/TAP

Ces interfaces sont des cartes réseau virtuelles [27, 17] c'est-à-dire que la carte n'existe pas en tant que telle et que sa gestion est faite de manière logicielle. Cela est pratique typiquement pour les applications en userspace qui ont besoin de directement interagir avec le trafic réseau. Elle leur suffit d'avoir accès au descripteur de fichier de cette interface pour récupérer tout le trafic qui leur est destiné.

Les interfaces TUN fonctionnent au troisième niveau de la couche OSI (Réseau), elles transportent des paquets IP et permettent donc aux applications de directement recevoir des paquets IP qu'elles devront traiter elles-mêmes. Ces interfaces sont utiles afin de réaliser des tunnels IP typiquement dans le cas d'un VPN où la carte réseau reçoit un paquet destiné au VPN qu'il devra déchiffrer avant de le transmettre à l'interface TUN.

Le second type d'interfaces est les interfaces TAP, celles-ci travaillent à la deuxième couche du modèle OSI (Liaison), et transportent des frames Ethernet. Elles sont utilisées pour faire des bridges Ethernet par exemple dans le cas de machines virtuelles, afin que celles-ci puissent dialoguer entre elles.

Les interactions avec ces interfaces virtuelles se font comme avec une interface physique, cela veut dire qu'au niveau kernel, celles-ci représentent simplement un 'net_device'.

5.1.4 Fonctionnement spécifique de WireGuard

Lorsque l'on lance WireGuard, celui-ci va créer une interface TUN généralement appelée 'wg0'. Les applications en userspace envoient des paquets IP à cette interface, qui sont ensuite encapsulés dans des paquets WireGuard et chiffrés. Ces paquets chiffrés sont transmis via les cartes réseau physiques (comme eth0 ou wlan0). On peut voir ci-dessous un exemple du fonctionnement dans le cadre d'une implémentation WireGuard classique dans le kernel.

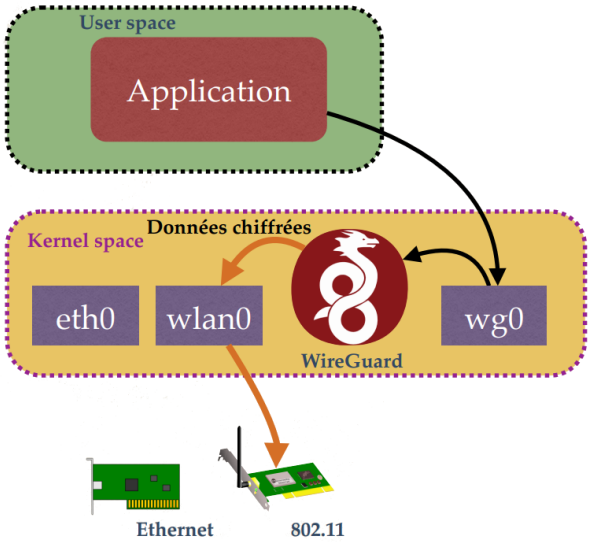


FIGURE 5.5 – Schéma du fonctionnement de WireGuard avec son implémentation kernel tiré du cours de SRX [14]

5.2 Limitations et solutions alternatives

5.2.1 Limitations de l'interface TUN/TAP

Dans le cadre de cette sous-section, ce sont les limitations concernant les interfaces TUN/-TAP qui sont abordées, mais on peut les généraliser à l'utilisation d'autres interfaces ou autres supports. Cela vient du fait que la principale limitation soit l'utilisation d'appels système. Lorsque l'on souhaite lire ou écrire sur l'interface, chacune de ces opérations demande

un appel système par donnée ce qui est très coûteux pour le processeur.

Malheureusement, il y a très peu d'articles traitant directement le cas des interfaces TUN/-TAP, mais il a été possible de trouver un article très complet ayant réalisé une analyse du coût des appels système. D'après cet article [33], les appels système sont coûteux pour plusieurs raisons :

- **Temps de changement de mode** : Le passage du mode utilisateur au mode noyau et vice-versa implique un vidage du pipeline du processeur et des sauvegardes de registres, ce qui augmente la latence.
- **Empreinte des appels système** : Pendant l'exécution en mode noyau, les caches du processeur et les TLB sont remplis avec des données spécifiques au noyau, polluant ainsi les structures du processeur utilisées par le mode utilisateur.
- **Impact sur l'IPC utilisateur** : Les appels système fréquents perturbent les performances des applications en réduisant l'IPC (instructions par cycle), à cause des interruptions directes et de la pollution des caches.
- **Coût du changement de mode sur l'IPC du noyau** : Les changements fréquents de mode nuisent également à l'efficacité du mode noyau, avec un IPC significativement plus bas par rapport au mode utilisateur.

5.2.2 Solutions existantes

Le problème à résoudre est donc de trouver un moyen qui nous permet de traiter plusieurs informations/connexions en parallèle tout en réduisant les appels système. S'il est possible de traiter deux données au lieu d'une seule avec l'équivalent d'un ‘read‘ ou ‘write‘, alors le temps passé dans les changements de contexte et tout ce qui en découle, devrait lui aussi être divisé par deux.

Afin de répondre à ce besoin, il y a quelques solutions qui sont ressorties lors de mes recherches sur internet. Dans les sous-sections qui suivent, j'explique le fonctionnement de chacune des solutions.

À noter qu'une des solutions qui ressortait beaucoup était ‘epoll‘, mais ce mécanisme est déjà utilisé au sein de WireGuard et il n'a donc pas été exploré.

5.2.2.1 io_uring

io_uring [26, 18] est une interface asynchrone et générique sortie en 2018 permettant de réaliser des opérations d'entrée/sortie en minimisant le nombre d'appels système utilisés. Elle fonctionne en utilisant deux files, aussi appelées ring buffers, qui sont des structures directement partagées avec le kernel. On retrouve un buffer (submission queue ou file de soumission) dans lequel le userspace va déposer les tâches que le kernel doit réaliser et un

buffer (completion queue ou file de complétion) dans lequel le kernel va déposer les résultats pour le userspace.

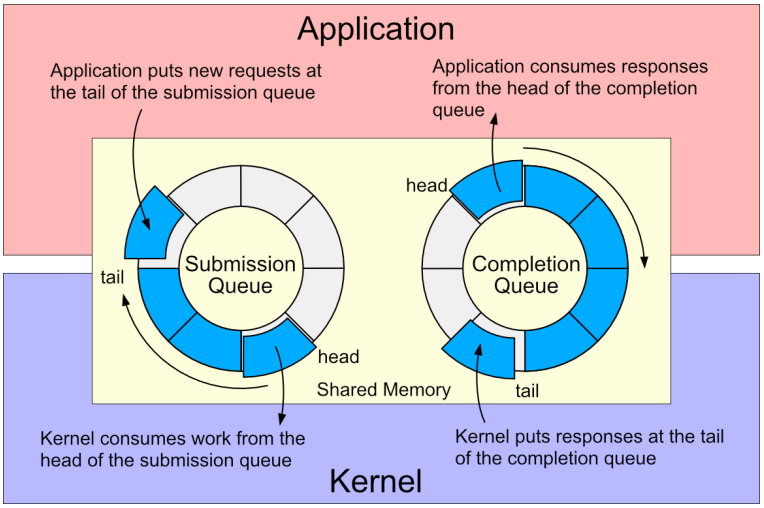


FIGURE 5.6 – Schéma des buffers de io_uring de Donald Hunter [18]

Une fois que l'on a mis toutes les tâches qui doivent être exécutées dans le file de soumission, on peut soumettre au kernel et celui-ci récupère ces tâches pour les exécuter. Ce principe de soumission des opérations est identique pour les opérations de lecture (récupération des résultats). Les appels système classiques pour écrire et lire sont remplacés par trois appels système spécifiques à io_uring :

- **io_uring_setup** : Configurer un contexte pour exécuter des opérations d'entrées/sorties asynchrones. Cette fonction initialise une structure io_uring qui permet la gestion des opérations d'entrées/sorties de manière asynchrone, en configurant les ressources nécessaires pour les files de soumission et de complétion.
- **io_uring_register** : Enregistrer des fichiers ou des buffers utilisateur pour des opérations d'entrées/sorties asynchrones. Cette fonction permet d'associer des descripteurs de fichiers ou des buffers mémoire aux opérations d'entrées/sorties gérées par io_uring, optimisant ainsi la gestion des ressources et réduisant les coûts des appels système.
- **io_uring_enter** : Initier et/ou compléter des opérations d'entrées asynchrones. Cette fonction est utilisée pour soumettre des requêtes d'entrées/sorties à l'anneau de soumission et pour récupérer les résultats des opérations terminées à partir de l'anneau de complétion, permettant ainsi une gestion efficace et non bloquante des entrées/sorties. C'est cet appel système qui va remplacer les appels classiques tels que read/write.

La manière optimale d'utiliser cette solution est donc de pousser sur la file de soumission plusieurs opérations et finalement de demander au kernel de les traiter. Ainsi, le kernel

traite plusieurs tâches en ayant réalisé un seul appel système (io_uring_enter). Il est tout de même important de noter que les tâches que l'on envoie ne sont pas forcément traitées dans l'ordre et qu'elles ne reviennent pas dans l'ordre non plus dans la file de complétion.

Un autre avantage de io_uring est sa généricité vis-à-vis de l'interface proposée, peu importe l'objet avec lequel on souhaite communiquer (socket, fichier ou encore interface réseau). Ce qui n'est pas le cas de 'epoll' par exemple qui différencie les fichiers des sockets.

Il existe aussi beaucoup de librairies en Rust proposant une implémentation de cette interface telles que tokio-uring, io-uring, glommio, et bien d'autres.

Le désavantage majeur de io_uring vient de sa jeunesse, il existe pour l'instant très peu de personnes connaissant très bien cette technologie. La communauté n'a pas non plus eu le temps d'auditer son code et de s'assurer qu'il n'y ait pas de vulnérabilités. On le remarque d'ailleurs avec Google qui révèle que 60% des exploitations du kernel Linux qui ont été remises en 2023 sont dues à io_uring.

5.2.2.2 (recv/send)mmsg

Cette solution propose une variante des appels système classiques 'recv' et 'send' que l'on retrouve lorsque l'on utilise des sockets. Elles permettent de recevoir et d'envoyer plusieurs messages en une seule fois, ce qui peut améliorer les performances en réduisant le nombre de transitions entre le userspace et le kernel space.

Au lieu de passer un seul message, on passe un vecteur de messages à envoyer et le kernel s'occupe de tous les envoyer en ayant réalisé un seul appel système. Cela permettrait d'améliorer grandement les performances si on arrive à faire des grands batchs.

Le seul problème de cette solution est que très peu de librairies en Rust proposent le support direct pour ces appels système.

5.3 Preuves de concepts et résultats

5.3.1 Mise en pratique des solutions

Afin de pouvoir tester les potentielles solutions trouvées, j'ai décidé d'utiliser un programme de test réalisant une opération réseau simple. Ce programme se charge de créer une interface TUN (tun0) sur laquelle il va recevoir des pings faits depuis ma machine utilisateur en mode flood. Lorsqu'il reçoit un ping, il va récupérer le paquet (echo request) et va créer lui-même la réponse (echo reply). Une fois la réponse créée, celle-ci est déposée sur l'interface TUN afin que ma machine ayant réalisé le ping reçoive la réponse. À noter que ce programme de test a été pris depuis un repo GitHub [11] qui proposait déjà tout le mécanisme cité ci-dessus, mon but est d'y apporter des modifications afin de voir les gains de performances.

Au total, quatre variantes de ce programme de test seront produites afin d'essayer à chaque fois d'améliorer ses performances. On va retrouver deux versions principales avec de l'I/O classique et io_uring, puis chaque version aura une déclinaison supplémentaire en asynchrone.

5.3.1.1 I/O classiques

J'ai commencé par prendre le programme de base trouvé sur GitHub et y apporter quelques modifications pour qu'il corresponde totalement à mes besoins, telles que l'ajout de la création du TUN avec la crate 'tun-tap' [35]. Ce programme continue d'utiliser les appels système classiques tels que 'read' afin de lire chaque paquet arrivant sur le TUN, puis 'write' pour déposer la réponse. L'exécution se fait de façon séquentielle, c'est-à-dire que chacun des paquets est traité l'un après l'autre.

Code Source : Extrait du programme de test classique

```
...

impl<'a> Connection<'a> {

    ...

    pub fn respond(&mut self, nic: &mut Iface) -> std::io::Result<usize> {

        ...

        let nbytes = nic.send(&buf[..buf.len() - unwritten])?; // Écriture sur le TUN
        Ok(nbytes)
    }
}

fn main() {
    let mut nic = Iface::without_packet_info("tun0", Mode::Tun).unwrap();
    cmd("ip", &["addr", "add", "dev", nic.name(), "192.168.0.1/24"]);
    cmd("ip", &["link", "set", "up", "dev", nic.name()]);

    let mut buf = [0u8; 1500];

    loop {
        let nbytes = nic.recv(&mut buf[..]).unwrap(); // Lecture du TUN

        match etherparse::Ipv4HeaderSlice::from_slice(&buf[..nbytes]) {
            Ok(iph) => {

                ...

                if let Some(mut c) = Connection::start(iph, data_buf).unwrap() {
```

```
32                     c.respond(&mut nic).unwrap(); // Traitement de la réponse
33                 }
34             }
35             ...
36         }
37     }
38 }
```

5.3.1.2 I/O classiques asynchrones

Cette variante reprend exactement le programme précédent sauf que les opérations ont été parallélisées avec la crate 'tokio' [34] afin de pouvoir traiter plusieurs paquets. Dans cette version, plusieurs tâches sont lancées lorsque le programme démarre. Chaque tâche se charge de lire les paquets arrivant sur le TUN et d'y déposer la réponse.

Une autre version avait été testée dans laquelle des tâches étaient créées pour la lecture au lancement du programme et à chaque fois une nouvelle tâche était créée pour l'écriture de la réponse, mais les performances étaient moins bonnes que la version actuelle.

Code Source : Extrait du programme de test classique asynchrone avec la création de 10 tâches

```
1
2 ...
3 // Similaire à la version précédente
4 ...
5
6 #[tokio::main]
7 async fn main() {
8
9     let nic = Arc::new(Iface::without_packet_info("tun0", Mode::Tun).unwrap());
10    cmd("ip", &["addr", "add", "dev", nic.name(), "192.168.0.1/24"]);
11    cmd("ip", &["link", "set", "up", "dev", nic.name()]);
12
13    let mut handles = vec![];
14
15    for _ in 0..10 {
16
17        let nic_clone = Arc::clone(&nic);
18        // Permet de créer 10 tâches de lecture
19        let handle = tokio::spawn(async move {
20            let mut buf = [0u8; 1500];
21            loop {
22                let nbytes = nic_clone.recv(&mut buf[..]).unwrap();
23                ...
24                // Similaire à la version précédente
```

```
25                 ...
26             }
27         });
28         handles.push(handle);
29     }
30
31     for handle in handles {
32         handle.await.unwrap();
33     }
34 }
```

5.3.1.3 io_uring

Cette version reprend le programme de base sauf que toutes les opérations de lecture et d'écriture ont été remplacées par celles d'io_uring. La crate qui a été retenue pour réaliser ce programme est 'glommio' [9], car parmi toutes celles testées c'est elle qui permet de réaliser les opérations io_uring avec le plus de facilité et sans avoir à se soucier de l'optimisation des soumissions. Ce qui n'est pas le cas, par exemple, des crates 'io-uring' ou encore 'tokio-uring', qui sont bien plus complexes à maîtriser et qui demandent un effort supplémentaire au développeur pour avoir une gestion optimisée des opérations.

On peut voir dans le code qu'il est tout de même nécessaire de récupérer le descripteur de fichier afin de créer un objet 'glommio'. C'est avec cet objet que l'on va pouvoir faire toutes nos opérations io_uring. On remarque que l'utilisation est vraiment très simple et ne diffère pas de la méthode classique.

Code Source : Extrait du programme de test avec io_uring

```
1 ...
2
3 impl<'a> Connection <'a> {
4
5     ...
6
7     pub async fn respond(&mut self, file: &BufferedFile) -> std::io::Result<usize> {
8
9         ...
10        // Utilisation d'io_uring pour écrire sur le TUN
11        let nbytes = file.write_at(buf[..buf.len() - unwritten].to_vec(),
  ↪  0).await.unwrap();
12        Ok(nbytes)
13    }
14 }
15
16 #[tokio::main]
```

```rust
async fn main() {

    let ex = LocalExecutor::default();
    ex.run(async {
        let nic = Iface::without_packet_info("tun0", Mode::Tun).unwrap();
        cmd("ip", &["addr", "add", "dev", nic.name(), "192.168.0.1/24"]);
        cmd("ip", &["link", "set", "up", "dev", nic.name()]);

        let fd = nic.as_raw_fd();
        let file = unsafe { BufferedFile::from_raw_fd(fd) };

        loop {
            // Utilisation d'io_uring pour lire sur le TUN
            let buf  = file.read_at(0, 88).await.unwrap();
            let nbytes = buf.len();

            match etherparse::Ipv4HeaderSlice::from_slice(&buf[..nbytes]) {
                Ok(iph) => {

                    ...

                    if let Some(mut c) = Connection::start(iph, data_buf,).unwrap() {
                        let _ = c.respond(&file).await.unwrap();
                    }
                }
                ...
            }
        }
    });

}
```

5.3.1.4 io_uring asynchrone

Cette ultime variante reprend la précédente et ajoute la gestion asynchrone grâce à ‘glommio‘ qui propose aussi ces fonctionnalités de façon similaire à ‘tokio‘ à l’aide de tâches. Contrairement au programme asynchrone fait avec ‘tokio‘, celui-ci fait apparaître plusieurs tâches pour la lecture et une nouvelle tâche à chaque fois pour écrire la réponse. Aucune baisse de performances n’a été observée avec ‘glommio‘ dans cette version, au contraire les performances ont été augmentées.

Code Source : Extrait du programme de test avec io_uring avec la création de 10 tâches

```
1
2 ...
3 // Similaire à la version précédente
4 ...
5
6 #[tokio::main]
7 async fn main() {
8
9     ...
10    // Similaire à la version précédente
11    ...
12
13    local_ex.run(async {
14
15        let mut handles : Vec<Task<()>> = Vec::new();
16
17        for _ in 0..10 {
18            // Création de 10 tâches pour gérer la lecture
19            let handle = glommio::spawn_local(async move {
20                let file_read = Arc::new(Mutex::new(unsafe {
  ↪ BufferedFile::from_raw_fd(fd) }));
21                let file_write = Arc::new(Mutex::new(unsafe {
  ↪ BufferedFile::from_raw_fd(fd) }));
22                loop {
23                    let buf = file_read.lock().unwrap().read_at(0,
  ↪ ICMP_PACKET_SIZE).await.unwrap();
24                    let nbytes = buf.len();
25
26                    match etherparse::Ipv4HeaderSlice::from_slice(&buf[..nbytes]) {
27                        Ok(iph) => {
28                            ...
29
30                            let file_write = Arc::clone(&file_write);
31
32                            if let Some(mut c) = Connection::start(iph,
  ↪ data_buf).unwrap() {
33                                // Création d'une tâche pour traiter la réponse
34                                let handle = glommio::spawn_local(async move {
35                                    c.respond(&file_write).await.unwrap();
36                                });
37                                handle.await;
38                            }
39                        }
40                        ...
41                    }
42                }
43            });
```

```
44              handles.push(handle);
45          }
46
47          for handle in handles {
48              handle.await;
49          }
50
51      });
52  }
```

5.3.2 Résultats

Afin de pouvoir correctement comparer les versions présentées précédemment, deux tests seront effectués sur chaque version. Le premier test consiste à faire 20000 pings en mode flood sur l'interface TUN, le but étant de vérifier la réaction face à une charge raisonnable à traiter. Le second test envoie quant à lui 100000 pings en mode flood, mais en séparant ceux-ci en 5 envois séparés faisant chacun 20000 pings. Ainsi, on peut tester chaque solution face à une charge considérable. D'ailleurs chaque paquet envoyé avait une taille de 1500 bytes. À noter que pour les programmes asynchrones, les tests ont été effectués sur des programmes utilisant 10 tâches en parallèle car c'était le nombre de tâches offrant le meilleur résultat.

Lors de chacun de ces tests, quatre métriques ont été relevées afin de déterminer l'efficacité des solutions.

- **Débit** : Récoltée en prenant la taille totale des paquets envoyés et en la divisant par le temps que cela a pris.
- **Temps de réponse moyen** : Récupérée directement depuis la sortie de la commande 'ping'.
- **Temps total pris par les appels système** : Récupérée depuis la commande 'strace' qui est lancée avec le programme.
- **Nombre d'appels système** : Récupérée depuis la commande 'strace' qui est lancée avec le programme.

5.3.2.1 Test de débit

Pour les débits, on remarque que les I/O classiques ont d'extrêmement meilleurs résultats que leur version avec io_uring. Il est cependant intéressant de relever que les performances d'io_uring augmentent avec la charge étant donné que 'glommio' cherche à regrouper un maximum d'opérations dans une seule soumission.

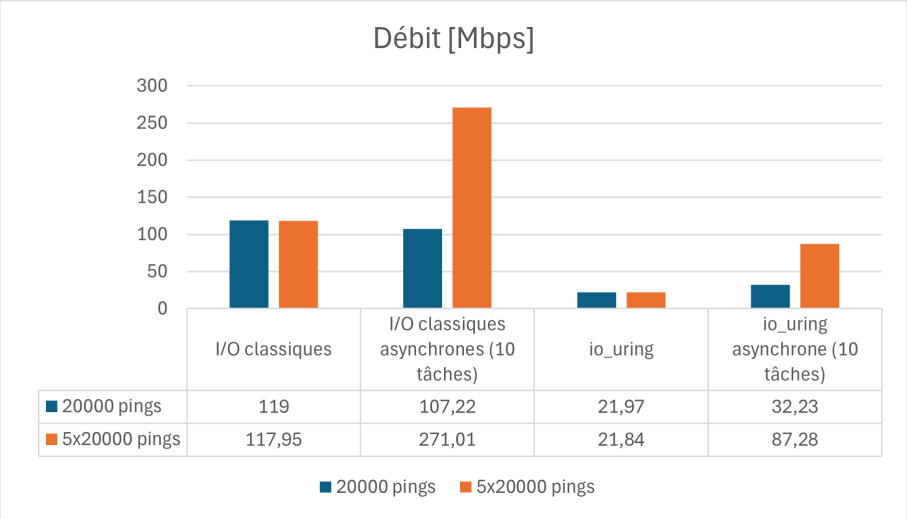


FIGURE 5.7 – Résultats du test de débit

D'autres tests ont été faits en augmentant le nombre d'instances ou en faisant varier la taille des paquets envoyés. L'augmentation de ces deux valeurs fait augmenter le débit jusqu'à un certain niveau. Plus on s'approche d'un très grand nombre d'instances, moins le débit augmente. Par exemple la différence entre 10 et 100 instances n'est que de quelques dizaines de 'Mbps' pour la version classique asynchrone, alors qu'on est proche de la centaine de 'Mbps' pour la différence entre 1 et 5 instances. On remarque le même impact sur io_uring avec des valeurs moins prononcées comme le montre le graphique.

5.3.2.2 Test de temps de réponse

Pour ce test, les résultats ne sont pas très étonnants, ni très intéressants. On remarque des résultats classiques pour du temps de réponse à des pings surtout sur une machine locale.

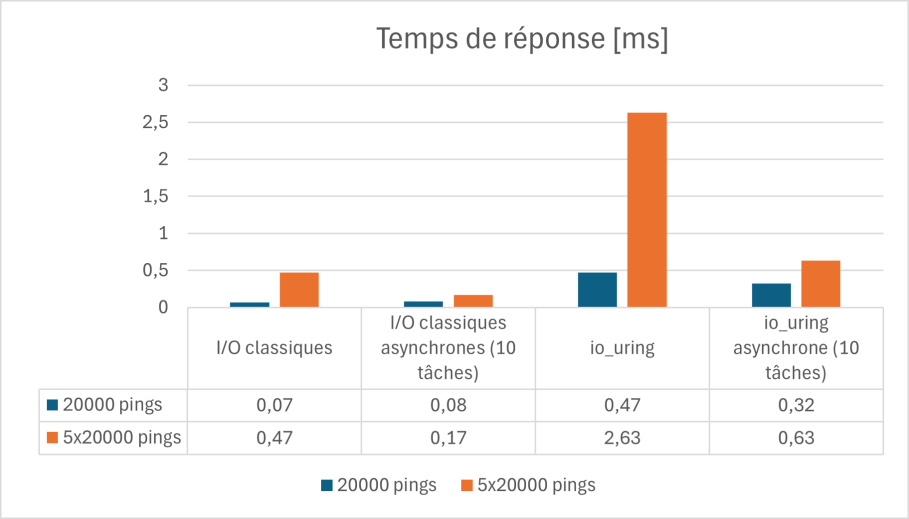
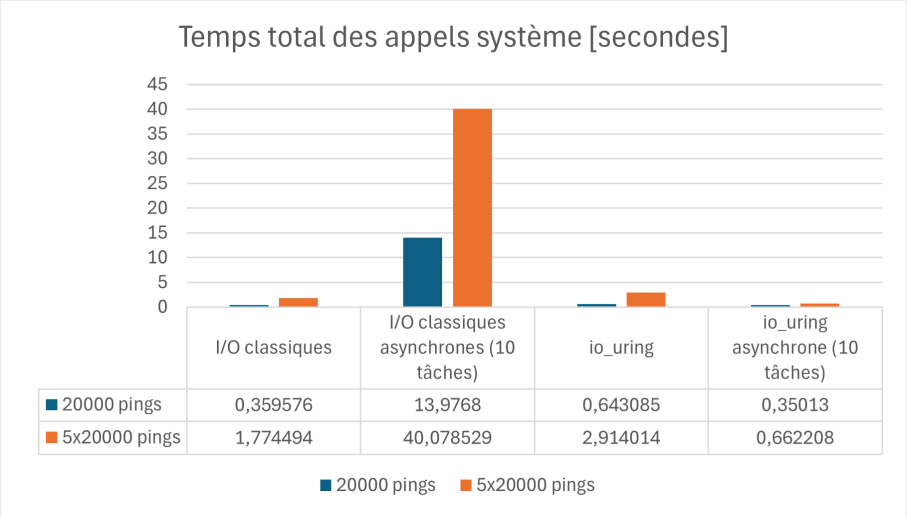


FIGURE 5.8 – Résultats du test de temps de réponse

De façon similaire au test précédent, l'augmentation de la charge ne fait qu'augmenter le temps de réponse requis pour chaque paquet.

5.3.2.3 Test du temps pris par les appels système

C'est à partir d'ici qu'io_uring commence à devenir une technologie avantageuse. Pour rappel dans ce chapitre, nous savons que le plus d'io_uring par rapport aux méthodes classiques se joue au niveau des appels système. Il optimise grandement ce qui se passe avec les appels système que ça soit au niveau de leur nombre ou de leur temps de traitement. On retrouve donc cela dans ce graphique où malgré la charge de travail, io_uring possède des performances excellentes. À noter que pour la version classique asynchrone, la majeure partie du temps des appels système est due à la gestion de la concurrence.



	I/O classiques	I/O classiques asynchrones (10 tâches)	io_uring	io_uring asynchrone (10 tâches)
20000 pings	0,359576	13,9768	0,643085	0,35013
5x20000 pings	1,774494	40,078529	2,914014	0,662208

FIGURE 5.9 – Résultats du test du temps pris par les appels système

5.3.2.4 Test du nombre d'appels système

Enfin, le test le plus intéressant pour la fin. Comme attendu, io_uring excelle dans cette partie surtout dans sa version asynchrone. Cela s'explique car il y a plusieurs tâches qui traitent les lectures et les écritures au même temps, donc la file de soumission est remplie avec plus qu'une tâche, ce qui permet en un seul appel système io_uring_enter de traiter plusieurs opérations. Ce qui nous permet donc d'arriver à 80000 appels système au lieu des 200000 réalisés par la version classique. Il y a d'ailleurs de fortes chances que cette valeur de 80000 soit la limite maximale pour notre programme, car même en augmentant le nombre de tâches s'exécutant en parallèle, on ne parvient pas à réduire le nombre d'appels système.

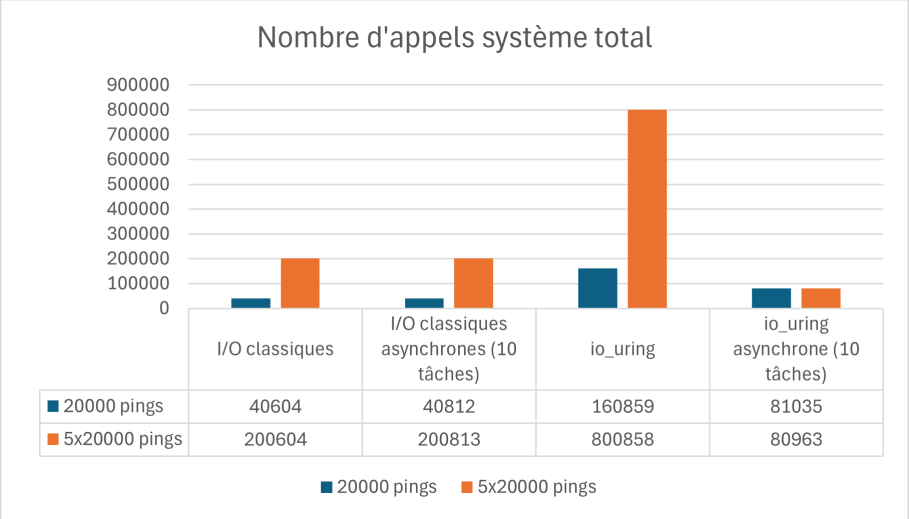


FIGURE 5.10 – Résultats du test du nombre d'appels système

Le grand challenge en réalisant ce programme de test était de vérifier s'il était possible d'atteindre d'une certaines manière un état de saturation. C'est-à-dire un état où io_uring accepte toutes les nouvelles requêtes sans créer de nouvel appel système. Ces deux derniers graphiques permettent de prouver qu'il est possible d'atteindre cet état.

5.4 Travaux futurs

Maintenant que nous savons que io_uring pourrait être une solution viable pour optimiser les opérations réseaux, une des perspectives futures serait de potentiellement réaliser une librairie combinant la gestion du TUN avec io_uring, afin de pouvoir par la suite l'intégrer dans BoringTun.

Chapitre 6

Conclusion

Lors de la réalisation de ce travail de Bachelor, il a été possible de démontrer la faisabilité et l'efficacité de l'intégration de BoringTun avec des périphériques sécurisés pour la gestion de la clé privée longue durée. L'utilisation de ces périphériques tels que des NitroKey ou Yubikey, ainsi que OpenPGP permet de grandement renforcer la sécurité générale de l'application en cas de compromission du système de l'utilisateur. Un attaquant prenant le contrôle de la machine ne peut plus simplement récupérer la clé privée, ce qui pourrait entraîner la compromission des sessions passées et l'usurpation d'identité pour les sessions futures. Il peut dans le pire des cas récupérer le secret partagé créé avec Diffie-Hellman pour la session courante ou le PIN de la clé de l'utilisateur, ce qui n'a aucun impact sur la sécurité de l'application elle-même.

Quant à la seconde partie de ce travail, le but était d'explorer le fonctionnement d'io_uring et de vérifier si cela permettait d'avoir de meilleures performances dans le cas des opérations d'entrée/sortie réseau. Les programmes de test ont montré avec succès, que io_uring pouvait, dans certains cas, améliorer les performances en réduisant notamment le nombre d'appels système utilisés. Il faudrait maintenant vérifier dans le cas d'une application réelle telle que BoringTun, s'il est possible de retrouver ces améliorations.

Bibliographie

[1] Jean-Philippe Aumasson and Daniel J. Bernstein. Siphash : a fast short-input PRF. *IACR Cryptol. ePrint Arch.*, page 351, 2012.

[2] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2 : simpler, smaller, fast as MD5. In *ACNS*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2013.

[3] Daniel Bernstein. Curve25519 : new diffie-hellman speed records. `https://cr.yp.to/ecdh/curve25519-20060209.pdf`, 2006. 2024-03-10.

[4] Daniel Bernstein. Chacha, a variant of salsa20. `https://cr.yp.to/chacha/chacha-20080128.pdf`, 2008. 2024-03-10.

[5] Daniel Bernstein. A state-of-the-art message-authentication code. `https://cr.yp.to/mac.html`, 2008. 2024-03-10.

[6] Karthikeyan Bhargavan. Noise explorer, 2019. 2024-03-19.

[7] Dalek Cryptography. Crate rust x25519-dalek. `https://crates.io/crates/x25519-dalek`, 2024. 2024-06-11.

[8] Pro Custodibus. Solution de pro custodibus. `https://www.procustodibus.com/blog/2023/03/openpgpcard-wireguard-guide/`, 2024. 2024-05-10.

[9] DataDog. Crate rust glommio. `https://crates.io/crates/glommio`, 2024. 2024-07-16.

[10] Francis Dinha. Openvpn website. `https://openvpn.net/faq/what-is-openvpn/`, 2024. 2024-03-19.

[11] Dave Dixx. Programme de test implémentant icmp en rust. `https://github.com/xphoniex/icmp-rust`, 2024. 2024-07-02.

[12] Jason A. Donenfeld. Wireguard : Next generation kernel network tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[13] Amr ElHusseiny. Blog de amr elhusseiny. `https://amrelhusseiny.github.io/blog/004_linux_0001_understanding_linux_networking/004_linux_0001_understanding_linux_networking_part_1/`, 2024. 2024-06-30.

[14] Linus Gasser. Cours de srx sur les vpns. page 21, 2023.

[15] BoringTun GitHub. Fix wg-quick. https://github.com/cloudflare/boringtun/issues/238#issuecomment-1276572644, 2024. 2024-06-18.

[16] OpenPGP Working Group. Site web de openpgp. https://www.openpgp.org/, 2024. 2024-05-14.

[17] Hechao. Blog de hechao. https://hechao.li/2018/05/21/Tun-Tap-Interface/, 2024. 2024-07-01.

[18] Donald Hunter. Why you should use io_uring for network i/o. https://developers.redhat.com/articles/2023/04/12/why-you-should-use-iouring-network-io, 2023. 2024-07-01.

[19] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Fiona Johanna Weber, and Philip R. Zimmermann. Post-quantum wireguard. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 304–321, 2021.

[20] Andrey Jivsov. Elliptic curve cryptography (ECC) in openpgp. *RFC*, 6637 :1–15, 2012.

[21] Sergey Klyaus. Blog de sergey klyaus. https://myaut.github.io/dtrace-stap-book/kernel/net.html, 2024. 2024-06-30.

[22] Hugo Krawczyk. Cryptographic extraction and key derivation : The HKDF scheme. In *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648. Springer, 2010.

[23] Auteurs multiples. Ipsec wikipedia. https://en.wikipedia.org/wiki/IPsec, 2024. 2024-03-19.

[24] Auteurs multiples. Wikipedia de hkdf. https://en.wikipedia.org/wiki/HKDF, 2024. 2024-03-10.

[25] Auteurs multiples. Wikipedia de wireguard. https://en.wikipedia.org/wiki/WireGuard, 2024. 2024-03-05.

[26] Auteurs multiples. Wikipédia io_uring. https://en.wikipedia.org/wiki/Io_uring, 2024. 2024-07-01.

[27] Auteurs multiples. Wikipédia tun/tap. https://en.wikipedia.org/wiki/TUN/TAP, 2024. 2024-07-01.

[28] Trevor Perrin. The noise protocol framework. http://www.noiseprotocol.org/noise.pdf, 2018. 2024-03-10.

[29] Achim Pietig. Functional specification of the openpgp application on iso smart card operating systems. https://gnupg.org/ftp/specs/OpenPGP-smart-card-application-3.4.1.pdf, 2024. 2024-05-14.

[30] Heiko Schaefer. Crate rust card-backend-pcsc. https://crates.io/crates/card-backend-pcsc, 2024. 2024-05-14.

[31] Heiko Schaefer. Crate rust openpgp-card. https://crates.io/crates/openpgp-card, 2024. 2024-05-14.

[32] Heiko Schaefer. Crate rust openpgp-card-sequoia. `https://crates.io/crates/openpgp-card-sequoia`, 2024. 2024-05-14.

[33] Livio Soares and Michael Stumm. Flexsc : Flexible system call scheduling with exception-less system calls. In *OSDI*, pages 33–46. USENIX Association, 2010.

[34] tokio.rs. Crate rust tokio. `https://crates.io/crates/tokio`, 2024. 2024-07-16.

[35] Michal Vaner. Crate rust tun-tap. `https://crates.io/crates/tun-tap`, 2024. 2024-07-16.

Table des figures

2.1 Schéma NoiseIK 8

4.1 Environnement de test 38

5.1 Schéma des ring buffers tiré du blog de Sergey Klyaus [21] 44
5.2 Schéma des socket buffers tiré du blog de Sergey Klyaus [21] 44
5.3 Schéma du traitement d'un paquet dans le kernel tiré du blog de Amr ElHusseiny [13] 47
5.4 Schéma global du lien entre les différentes structures tiré du blog de Sergey Klyaus [21] 48
5.5 Schéma du fonctionnement de WireGuard avec son implémentation kernel tiré du cours de SRX [14] 49
5.6 Schéma des buffers de io_uring de Donald Hunter [18] 51
5.7 Résultats du test de débit 59
5.8 Résultats du test de temps de réponse 60
5.9 Résultats du test du temps pris par les appels système 61
5.10 Résultats du test du nombre d'appels système 62

Liste des tableaux

4.1 Résultats des tests 39

Annexe A

Journal de travail

Jours	Tâches
20.02.24 - 25.04.24	Réalisation de l'état de l'art • Comprendre le fonctionnement de WireGuard (white-paper), les primitives cryptographiques utilisées, etc... • Recherche des implémentations existantes • Recherche des périphériques existants • Recherche d'autres solutions existantes
07.04.24 - 08.04.24	Rédaction du cahier des charges final
11.04.24	Rendu du cahier des charges
25.04.24 - 09.05.24	Création du POC permettant de prouver la compatibilité entre la cryptographie du périphérique/OpenPGP avec BoringTun
09.05.24 - 25.05.24	Rédaction de la partie sur le choix du périphérique et du POC réalisé • Explications sur le choix de YubiKey/Nitrokey + OpenPGP pour ce projet • Détailler les librairies utilisées pour la réalisation du POC • Expliquer comment fonctionne le POC • Expliquer les problèmes rencontrés durant la réalisation
25.05.24	Rendu du rapport intermédiaire

Jours	Tâches
25.05.24 - 07.06.24	Ajout du support des périphériques sécurisés à BoringTun
10.06.24 - 19.06.24	Documentation de la partie concernant BoringTun et l'ajout du support des périphériques sécurisés • Explications concernant le choix de BoringTun par rapport aux autres implémentations • Détailler son organisation au niveau du code et son fonctionnement • Mise en lien avec la cryptographie vue précédemment avec les périphériques sécurisés • Expliquer comment fonctionne le code créé et les modifications apportées au code de base • Explication du laboratoire de test mis en place afin de tester mon code • Détailler les tests réalisés afin de valider le fonctionnement du programme et comprendre ses limites • Explication des problèmes rencontrés
19.06.24 - 11.07.24	Objectifs optionnels sur l'amélioration des performances • Se renseigner sur le fonctionnement interne du kernel Linux au niveau du réseau (ring buffers, socket buffers, etc...) • Se renseigner sur les solutions performantes existantes afin de réaliser un POC (choix de io_uring) • Tester les différentes librairies permettant de gérer des TUN devices (tun-tap, tokio-tun), ainsi que pour faire de l'io_uring (tokio-uring, io-uring, glommio). Finalement le meilleur moyen de faire fonctionner le POC a été de prendre la librairie tun-tap avec glommio afin d'avoir un POC simple et fonctionnel. • Mesurer les performances proposées par les POCs standards comparés aux POCs io_uring.
01.07.24 - 18.07.24	Rédaction du rapport concernant la partie sur les performances
12.07.24	Rédaction du résumé publiable
12.07.24	Rédaction de l'affiche

Jours	Tâches
18.07.24 - 25.07.24	Relecture des documents, ajout de détails, préparation des rendus finaux, etc...
25.07.24	Rendu du travail de Bachelor