ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Implementation of High-Level Cryptographic Protocols using a SoC Platform

Mémoire présenté en vue de l'obtention du diplôme
d'Ingénieur Civil en informatique à finalité spécialisée

Quentin Delhaye

Directeur
Professeur Frédéric Robert

Superviseur
Sébastien Rabou (Barco Silex)

Service
BEAMS

Année académique
2014 - 2015

Aknowledgements

Abstract

by Quentin Delhaye, Master in Computer Science and Engineering, Professional Focus, Université Libre de Bruxelles, 2014–2015.

Implementation of high level cryptographic protocols using a SoC platform

This is an abstract. Yep.

Résumé

par Quentin Delhaye, Master en ingénieur civil en informatique, à finalité spécialisée, Université Libre de Bruxelles, 2014–2015.

Implementation of high level cryptographic protocols using a SoC platform

Abstract, bitch !

Contents

Aknowledgements **i**

Abstract **ii**

Résumé **iii**

Contents **iv**

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**
1.1 Challenge 1
1.2 Network security 1

2 Technical background **2**
2.1 FPGA 2
2.2 Operating system 4
2.2.1 Device Driver 4
2.2.2 Linux Crypto API 5
2.3 Cryptography 6
2.3.1 Message digest 6
2.3.2 Symmetric cryptography 7
2.3.3 Asymmetric cryptography 8
2.4 Network and VPN 12
2.4.1 SSL/TLS 13
2.4.2 IPsec 15

3 Implementation **18**
3.1 Software 18
3.1.1 OpenVPN 18
3.1.2 OpenSSH 21
3.1.3 OpenSSL 22
3.1.4 Strongswan 22
3.1.5 Linux drivers 23

3.2 Offload . . . 24
3.2.1 BA411E Driver . . . 25
3.2.2 BA414E Driver and Silex engine . . . 25

4 Test protocol **26**
4.1 Experimental setup . . . 26
4.1.1 x86 host . . . 26
4.1.2 Altera Socrates SoCFPGA . . . 26
4.1.3 ARM DS-5 Streamline . . . 27
4.2 Test cases . . . 27
4.3 TLS Connections . . . 28
4.4 Response time – latency . . . 29
4.5 File transfer . . . 29

5 Results and Analysis **30**
5.1 TLS connections . . . 30
5.2 Response time – latency . . . 32
5.2.1 OpenVPN . . . 32
5.2.2 IPsec . . . 34
5.2.3 Comparison . . . 35
5.3 File transfer . . . 36
5.3.1 OpenSSH . . . 36
5.3.2 OpenVPN . . . 38
5.3.3 IPsec . . . 39
5.3.4 Comparison . . . 41

6 Discussion and conclusion **44**
6.1 Future work . . . 44

A OpenSSH patch **47**

B Configuration files **49**
B.1 OpenVPN . . . 49
B.2 Strongswan . . . 50

Bibliography **51**

List of Figures

2.1 Barco Silex' AES IP Core 3
2.2 User and kernel space separation 4
2.3 CBC encryption diagram 7
2.4 CTR encryption diagram 8
2.5 GCM encryption diagram 9
2.6 SSL/TLS and IPsec integration into the TCP/IP network stack 13
2.7 TLS handshake 14
2.8 IPsec (a) transport and (b) tunnel overheads 17
2.9 ESP packet structure 17

3.1 OS data paths 19
3.2 OpenVPN packet flow 20
3.3 OpenSSL debugging benchmark 23
3.4 IPsec user/kernel space workflow, using `ping` as a test case. 24

5.1 TLS connections per minute 31
5.2 OpenVN: ping average response time – software 32
5.3 OpenVN: ping average response time – Software/Hardware 33
5.4 IPsec: ping average response time – Software 35
5.5 IPsec: ping average response time – software/Hardware 36
5.6 Ping average comparison 37
5.7 File transfer over an SSH tunnel 38
5.8 FTP file transfer over an OpenVPN tunnel 39
5.9 FTP file transfer over an IPsec tunnel 40
5.10 Comparison of file transfer methods 42
5.11 OpenSSL benchmark for AES-256-CBC 43

List of Tables

2.1 TCP/IP and OSI model comparison 12

3.1 Role of the softwares 18
3.2 Summary of the ciphers supported by two of Barco Silex' IP cores. . . 25

4.1 Software used for the test 28

5.1 OpenVPN connection time 30
5.2 TLS connections per minute 31
5.3 OpenVN: ping average response time 33
5.4 IPsec: ping average response time 34
5.5 File transfer over an SSH tunnel 37
5.6 FTP file transfer over an OpenVPN tunnel 38
5.7 FTP file transfer over an IPsec tunnel 39
5.8 Packet size frequency for an FTP transfer 42

Chapter 1

Introduction

Altera and Xilinx are trying to push those SoC plateforms to the market to popularize the FPGAs.

Power is critical nowadays, esapcially in datacenters. Several solutions exists: abandonning general purpose processors altoghether and turn towards ARM cores, or add an FPGA to the processor for example. Intel proposed the latter solution in 2014 [**?**] and is on its way to expand that market with the merger with Altera [**?**].

There a re diffirent ways to add cryptographic capabilities to an application:

- Having the cipher built in.
- Using a cryptographic library, such as OpenSSL.
- Using the Linux Crypto API.

Present the two IP cores that will be used in this work: BA411E and BA414E.

1.1 Challenge

1.2 Network security

Chapter 2

Technical background

This chapter will address the technical ground necessary to this work. First comes a quick presentation of FPGAs and how they can be driven from the operating system.
Follows an overview of Linux operating systems, the distinction between user and kernel mode, and the design of device drivers.
Next comes an overview of the cryptographic field with the main primitives: message digests, symmetric ciphers and asymmetric ciphers.
Finally, the ground necessary to understand the inner working and challenges of some VPNs implementations using SSL/TLS and IPsec will be covered.

2.1 FPGA

The aim of this work is to study the impact of offloading cryptographic operations in hardware. In order to reach that goal, an FPGA will be used.

A Field-Prgrammable Gate Array (FPGA) is an integrated circuit that can be programmed after the manufacturing, using a semiconductor intellectual property core (IP core). In this work, we will use some of Barco Silex's crypto IP cores.

In order for the operating system to comunicate with the device, they will need to share some memory. That memory can be directly mapped and accessed from the user-space using `/dev/mem`, or can use a direct memory access module (DMA). From the operating system, we build a bunch of scatterlist in the kernel-space memory, then map those pages to memory descriptor that have a physical address on the DMA. They can be mapped three different ways: `DMA_BIDIRECTIONAL`, `DMA_TO_DEVICE` or `DMA_FROM_DEVICE`. When the CPU write something in those descriptors and synchronize them with the DMA, it does not have to care about them anymore, the DMA in now in charge to send them to the device where registers are ready to read the incomming data. The same goes from the device to the CPU: when the device wants to communicate data to the OS, it writes it on the DMA that will transfer them to the CPU, triggering a flag on the way to notify it.

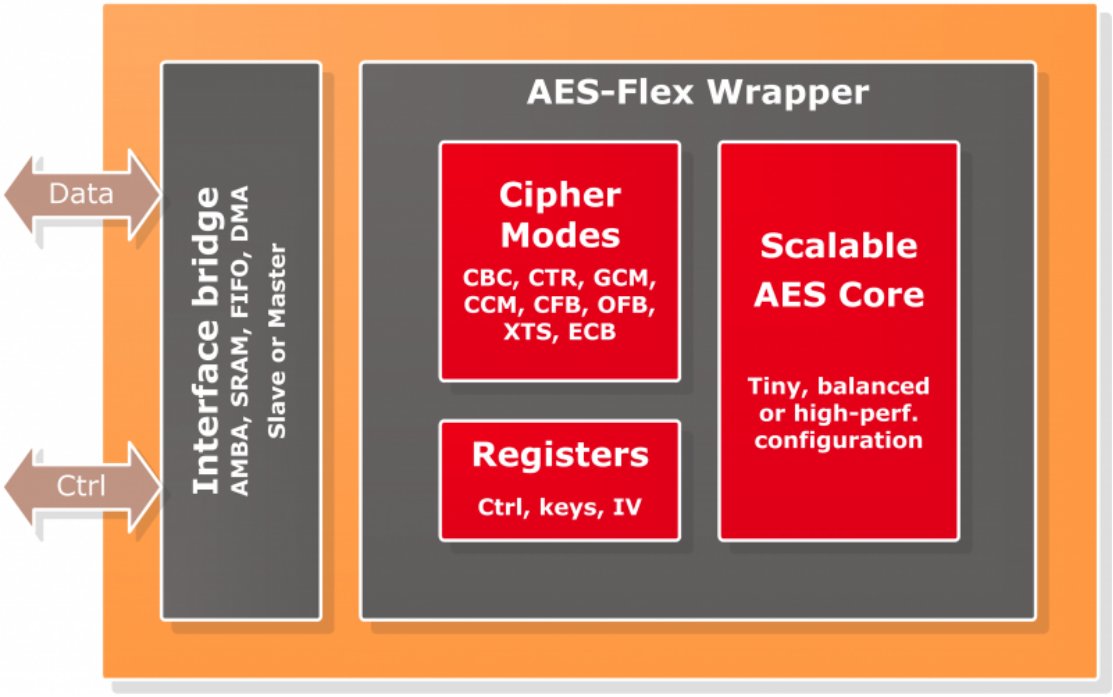


Figure 2.1 – **Barco Silex' AES IP Core:** There are two channels to comunicate with the IP core: the data and control paths [4].

There are two types of interface with such a device: master or slave. The figure 2.1 represent one of Barco Silex' crypto IP cores that proposes two interfaces: one of each type.

- The data path is a master. That is the operating system will not have a direct access to the memory registers of the hardware devide. Instead, they will communicate via a DMA bus: one peer places data on the DMA and notifies the other that he can fetch it.
- The control (ctrl) path is a slave. The operatin system can directly read and write on this part of the device memory.

A master interface has the advantage to offload a part of the memory management to the device, but there is a space overhead to connect the DMA as well as an extra delay to process its data. A slave interface is usually used for sporadic data, avoiding the overhead of a DMA. As an example, the data path of Barco Silex' symmetric cryptography IP core is a master, as there is a lot of data to process, whilst the same data path of their asymmetric cryptography IP core is a slave, for the operands are larger and the computation takes more time.

2.2 Operating system

Modern operating systems run on several levels of privilege. Under Unix, the kernel runs under the highest level and it has no restriction, whereas applications run on the lowest level. We call those two levels kernel space and user space, segragating the privilege and the memory between the two modes (Tanenbaum [43] and Corbet et al. [13, chap. 2]). The figure 2.2 shows this separation. In order for the applications in the user space to reach the hardware, they need to go through the kernel and use the device drivers that are registered there.

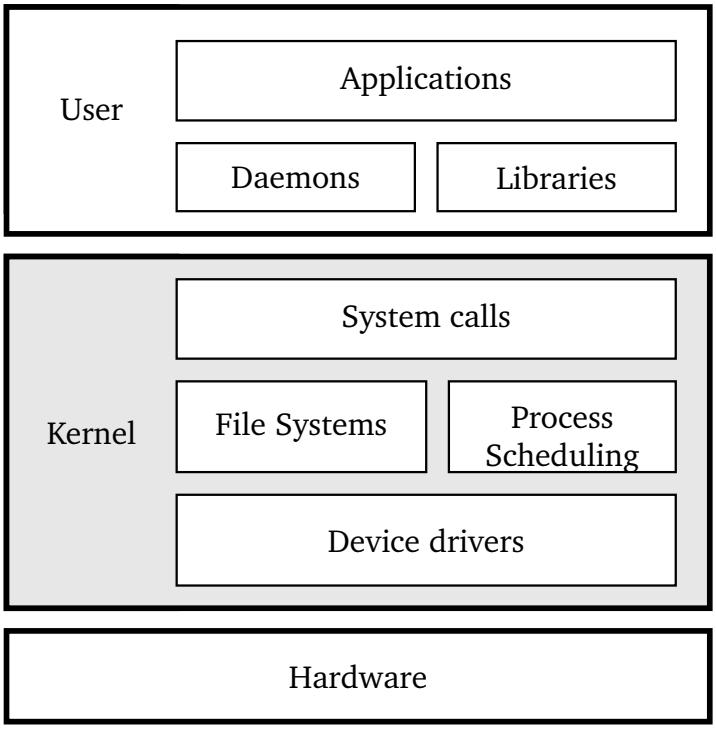


Figure 2.2 – **User and kernel space separation:** The applications and libraries from the the user space communicates with the kernel space via the system calls.

2.2.1 Device Driver

Device drivers allows the operating system to interact with hardware devices. On a Linux operating system, a driver can be loaded into the system while it is running, simply registering itself into the kernel. This dynamic loading and unloading offers a phenomenal modularity to the system, and consequently, a device driver is also called a kernel module.

There are two ways to get the status of a hardware device: either by using interruption, or by actively polling the device, relentlessly asking if it finished its operations. The first case is the cleanest and the most common: when the device has something to send to the driver, or if anything unexpected happened, it

sends an interruption request (IRQ) to the processor, which will in turn execute the interruption service routine (ISR) registered by the driver [13, chap. 10]. The second case is the easiest and is always guaranteed to work, but won't let go of the processor willingly, loading it at 100%, and avoiding any other task to be executed. Fortunately, modern monolithic kernels such as the Linux kernel from 2.6 provide preemptive scheduling [41], that is the scheduler interrupts the running task and assigns the processor ressources it used to an other one. Hence, systems with a lot of processes in need for CPU ressources would not be stalled, but it would not change anything if the only process heavily requesting processor time is the one using the driver.

The vast majority of drivers are meant to be run in kernel space, but some use cases may require the user space to directly access the hardware. In those cases, one can use the Userspace I/O framework.

Koch [25] explains that a device driver has two roles: accessing the device memory and dealing with its interruptions. The device memory can already be accessed from the user space without the need for UIO via `/dev/mem`, but UIO improves this method by preventing the user space from mapping memory that does not belong to the device.
As for the interruptions, the user space alone is not enough. When the device sends an IRQ, the system will react by executing an ISR. The IRQ flag however needs to be deactivated at the end of the routine, and this must be done in kernel mode. Hence, a small kernel module is still needed to handle this ISR.
Of course, this type of driver can also actively poll the status of the device, removing the need for a kernel module as it would not rely on interruptions.

2.2.2 Linux Crypto API

The Linux Crypto API is a cryptographic framework giving access to cryptographic capabilities in the kernel space [32]. Initially, it was developed to support IPsec operations, but it evolved into a general-purpose API.

By providing such an interface, the kernel applications[1] can used cryptographic primitives without having to rely on their concrete implementation. In order to add such an implementation, a kernel has to be developed and register itself with the Linux Crypto API upon its loading. Those kernel modules can either be software implementation of some cryptogrpahic ciphers, or a device driver that will offload those operation to a dedicated hardware device.

The user space can also gain access to the Linux Crypto API, by using an intermediary virtual device `/dev/crypto`, *aka* CryptoDev [11]. That support can be added by registering the CryptoDev kernel module with the kernel, and then access it from the user space, usually using a cryptographic library engine (see section 3.1.3).

[1] Used here as a misnomer to designate the code executed in kernel space.

2.3 Cryptography

Cryptography is the keystone of security. As Menezes et al. [31] defines it: “Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication."

The four main goals are:

Confidentiality keeping information secret from all but those who are authorized to see it.

Integrity ensuring information has not been altered by unauthorized or unknown means.

Source Authentication corroborating the source of information.

Non-repudiation preventing the denial of previous commitments or actions.

In order to achieve those, four cryptographic primitives are needed: message digests, symmetric and asymmetric ciphers, and digital signatures.

2.3.1 Message digest

A message digest is the result of a one-way mathematical function of a fixed size. For a given function, each message has a unique digest, comparable to a fingerprint. Those hash functions are of two types [28]: manipulation detection codes (MDC) to guarantee integrity and message authentication codes (MAC) to guarantee both integrity and source authentication.

An MDC $h(x)$ can follow an iterative construction for a message x including t blocks:

$$\begin{cases} H_0 = \text{initial value} \\ H_i = f(H_{i-1}, x_i), \text{with } i \in [1, t] \\ h(x) = H_t \end{cases}$$

In order to create a MAC, we can add a secret key k to the process, H_i becoming:

$$H_i = f(H_{i-1}, x_i, k), \text{with } i \in [1, t]$$

HMAC is such function $f()$, as defined in RFC 2104 [26]

$$HMAC(k, x) = h((k \oplus opad)|h((k \oplus ipad)|x))$$

with a key k, and two padding block added for security concerns: an outer pad $opad$ and an inner pad $ipad$.

There exist a wide varety of MDCs, ranging from block cipher based such as Miyaguchi-Preneel, customized such as MD5, SHA-1 and SHA-2, or built using modular arithmetic such as MASH-1.

In both schemes, data integrity can be guaranteed because the flip of one bit will irremediably change the digest. However, only a MAC can ensure source authentication since it is the only one based on a shared secret key.

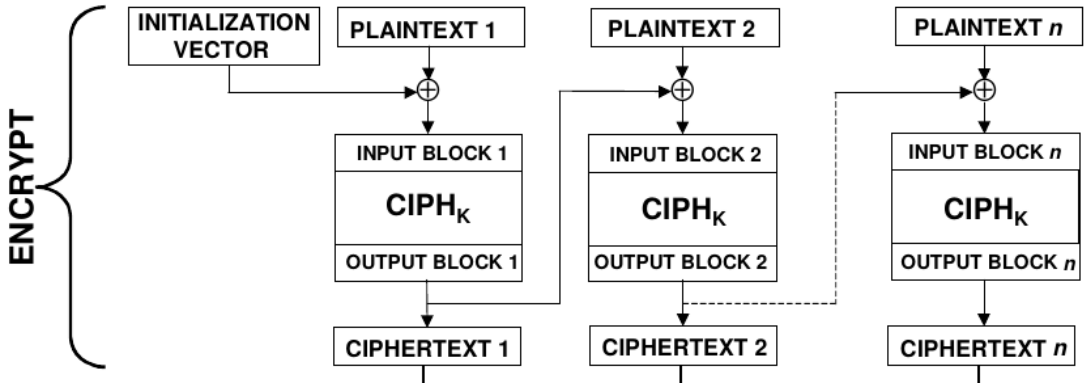


Figure 2.3 – **CBC encryption diagram:** The output of one block is used as the input of the next one. The decryption is very similar. Taken from the NIST recommendation [15].

Now rises the question of what and when authenticating. Bellare and Namprempre [8] prooved that the most secure solution is to encrypt then compute the MAC from the ciphertext. We will see in section 2.4 that if IPsec follows this recommandation, SSL/TLS does not and MAC first the plaintext then encrypt the message.

2.3.2 Symmetric cryptography

Consider a cryptosystem (P, C, K, E, D), with P and C the spaces of plaintexts and ciphertexts, E and D the encryption and decryption algorithms and a keyspace K. Menezes et al. [31] and Markowitch [28] define a symmetric scheme as a cryptosystem with two keys $e \in K$ and $d \in K$ respectively associated with the encryption and decryption algorithms. Hence, for all messages x in the message space M: $D_d(E_e(x)) = x$. When using the key pair (e, d), it should be computationally easy to compute d knowing only e, and e from d. In the rest of this work, we will consider the most common case: $e = d$, having thus a single key shared between the peers.

There exists numerous cipher types, and one of the most widespread is the block cipher. A block cipher splits the message into blocks of fixed size and work on them separately. An example of such a cipher is illustrated on the figure 2.3 with the Cipher Block Chaining (CBC) mode. This particular mode is easy to implement and has been robust for a long time, before the rise of the padding oracle attack [46].

An other interresting block cipher mode is the Counter (CTR). Each block of data is treated independently from the others (see figure 2.4, making the scheme parallelizable and thus more interresting on hardware.

Those modes can not work alone and have to be associated with a cipher (“CIPH" on figures 2.3 and 2.4). In 1997, the NIST hosted a competition to

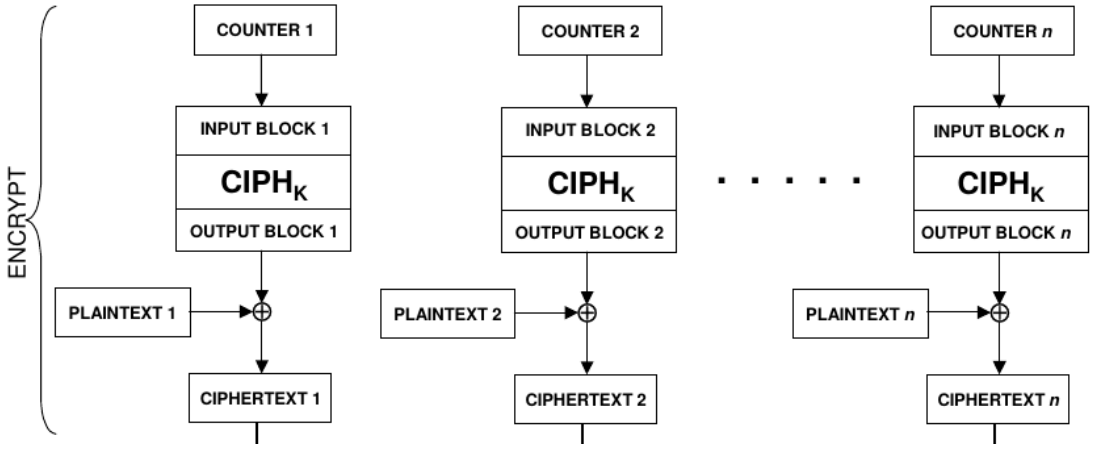


Figure 2.4 – **CTR encryption diagram:** Each block is processed independently from the others. The decryption is very similar. Taken from the NIST recommendation [15].

choose the successor of the Data Encryption Standard (DES). In 2000, Rijndael emerged victorious and was renamed Advanced Encryption Standard (AES). It is still nowadays the most used symmetric block cipher used in a large variety of applications.

Most applications want to use confidential and authenticated channels for their sensible communication. In order to do so, they need to combine a symmetric cipher and a MAC.
New block cipher have emerged to answer the need for such combinations and formed the Authenticated Encryption with Associated Data (AEAD) mode of operation. The most widespread mode in network applications is the Galois Counter Mode (GCM, see figure 2.5), combining the CTR block cipher mode and a GHASH function using multiplications in the Galois field. The advantage of this mode is that the authentication can be computed in parallel with the encryption, itself parallelizable. This mode can thus be highly optimized in hardware, as proved Barco Silex with their BA415 IP core reaching up to 100Gbps [5].

2.3.3 Asymmetric cryptography

Asymmetric cryptography relies on a pair of keys: one private known only to its owner, and one public available to anyone. If we use the same notation for the cryptosystem as in section 2.3.2, the public key would be e, and the private key d. Such cryptography uses two kinds of operations:

- Encryption: the sender uses the public key of the recipient to encrypt the message m into a ciphertext $c = E_e(m)$, and the recipient uses his private key to decrypt it: $m = D_d(m)$.

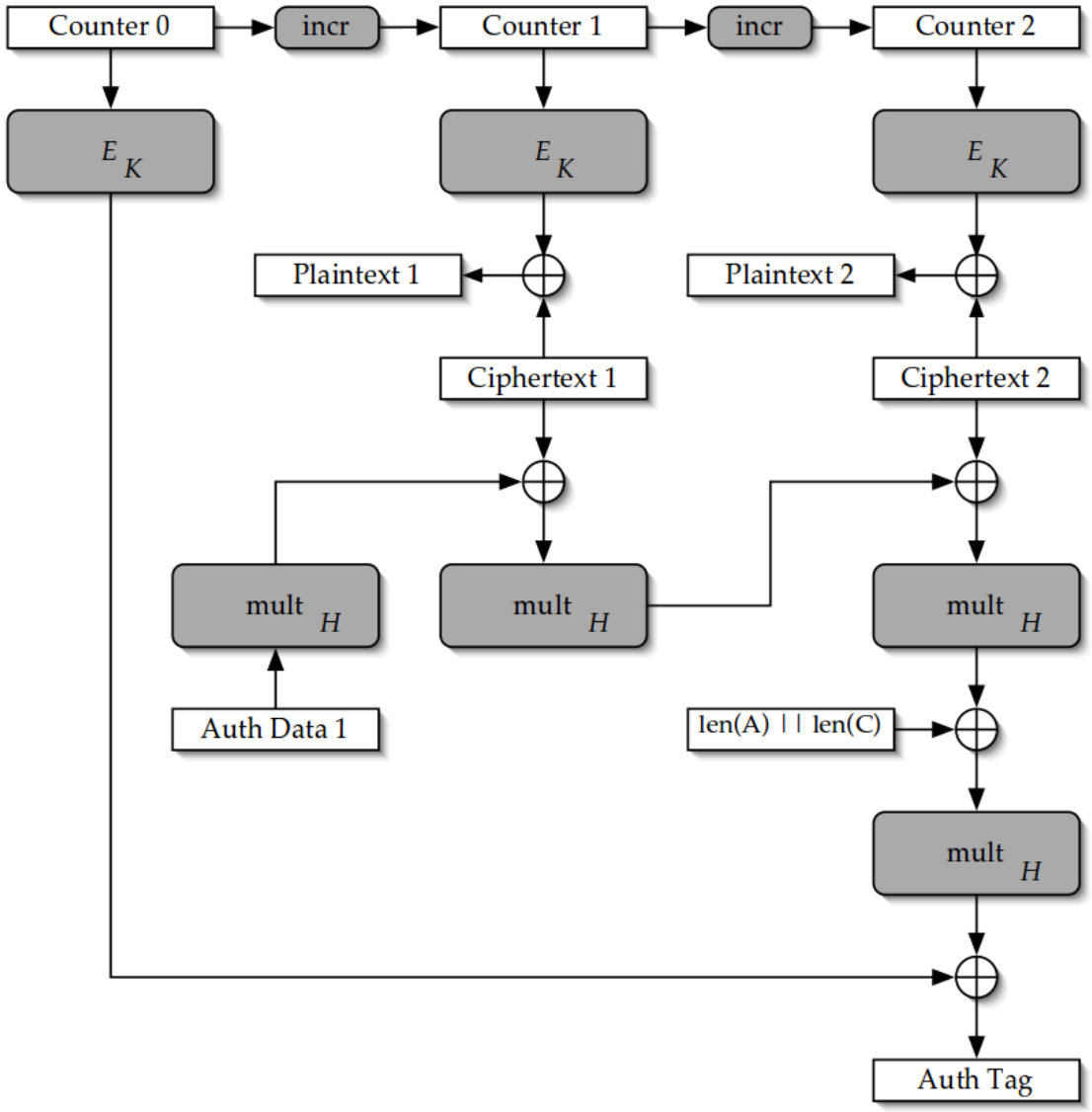


Figure 2.5 – **GCM encryption diagram:** taken from the NIST specification [30]. The ciphertext blocks are formed by *xor*-ing the encrypted counter and the plaintext. The tag is generated by a chain of ciphertext *xor*-ing with Galois field multiplicated data. The decryption works excatly the same way, except the plaintext and ciphertext are swapped.

- Digital signature: the sender generates the signature s of the message with his private key: $s = D_d(m)$, and the recipient uses the public key of the sender to verify the signature: $m = E_e(s)$. Since the sender can not deny having signed the message with his private key, this adds a non-repudiation of origin property to the scheme.

Eventhough the signature algorithms are often more efficient than the usual MACs and the key pairs key stay in use for very long period of time, asymmetric cryptography can not supplant its symmetric counterpart. The throughput rates of modern asymmetric schemes are still well under the symmetric schemes, plus the key sizes are much larger. Whereas the most efficient search for the symmetric key (ignoring the weaknesses specific to certain modes) is an exhaustive search, all public-key systems are vulnerable to more efficient attacks (*e.g.* factorization or discrete logarithm).

In practice, public-key cryptography is used to estalish a secure channel between two or more peers and exchange a shared secret key that will be used the encrypt the subsequent communications.

2.3.3.1 RSA

RSA is a public-key scheme proposed in 1978 by three MIT researchers (who gave it their name) [38]. A few years later, they founded RSA Laboratories, which is now in charge of maintaining its standards, alongside many others, as the first Public-Key Cryptography Standards, *aka* PKCS #1. The last version of the standard is the version 2.2 [40] and is defined as a precise key generation protocol allowing encryption and decryption. The keys can be generated by respecting a few steps:

1. randomy choose two large primes p and q;
2. compute the modulus $n = pq$, and consequently we have

$$\phi(n) = (p-1)(q-1)$$

   with $\phi(n)$ as the Euler function;
3. randomly choose the public exponent $e \in]1, \phi(n)[$ $s.t.$ $GCD(e, \phi(n)) = 1$;
4. compute $d \in]1, \phi(n)[$ $s.t.$ $e \cdot d \equiv 1 (mod\ \phi(n))$

With those parameters, we can form a public key with the pair (n, e) and a private key with the pair (n, d).

The encryption and decryption of a given message $m \in \mathbb{Z}_n$ are defined as follows:

Encryption $c = m^e\ mod\ n$
Decryption $m = c^d\ mod\ n$

The scheme is based on modular artihmetic, which is secure because of the factoring problem. For an opponent to decrypt a ciphertext c wihtout knowing

the private key (d, n), he would need to compute the e^{th} roots of c, modulo n. This operation is computationally hard, since an efficient solution is yet to be found for sufficiently large key size.

2.3.3.2 Diffie-Hellman

Diffie-Hellman is a secret key exchange protocol: two parties compute a shared secret ZZ that can be used as a symmetric key during the following exchanges. It uses the same kind of operation as RSA, that is modular exponentiation. The protocol can be one of three types ([36], [16]):

- Static (DH): the actors use their authenticated certificate to compute the shared secret.
- Ephemeral (DHE): the actors create a new pair of public/private keys from which the secret key is derived.
- Anonymous: same as ephemeral, but without signing anything, hence not identifying neither of the actors, hence not providing any authentication. This mode is not advisable since it's vulnerable to main-in-the-middle attack.

A static scheme is easier to implement and requires much less operations, but using ephemeral keys is essential to ensure perfect forward secrecy. Imagine that somehow, an opponent lays his hand on the shared secret. If that secret has already been used, he can decipher all data transfered during past connections. However, if the secret is new for every new connection, the compromission of the shared secret does not jeopardize past communications. This is perfect forward secrecy: using a new key to protect the previous messages.

Hereunder is the generation algorithm of an ephemeral shared secret. For a static secret, Alice and Bob will simply use their static certificate, sparing the modular exponentiation of the ephemeral public key generation.

1. Alice generates once p and g (using precomputed parameters):
   p large prime number
   g a generator of $\mathbb{Z}_p^*$
2. Alice picks a random integer x_a and computes $g^{x_a} mod\ p = y_a$.
3. Alice sends p, g and y_a to Bob, signing everything using her private certificate.
4. Bob checks the signature and picks x_b.
5. Bob computes $y_a^{x_b} mod\ p = g^{x_a x_b} mod\ p = ZZ$, the shared secret to use as a premaster key from which will be derived the symmetric key for further communications.
6. Bob sends $y_b = g^{x_b} mod\ p$, signing everything with his private certificate.
7. Alice checks the signature and computes the same shared secret: $ZZ = y_b^{x_a} mod\ p = g^{x_a x_b} mod\ p$

If the server is Alice, it has to do at least one signature generation, one signature verification and two modular exponentiations. Note that the client, B in

our case, could have one signature less because RFC 5246 [47] leave it as an optional feature, and the server would then have one verification less. However, any sane configuration will have both actors signing their ephemeral public key. If the certificate use RSA, we end up with four modular exponentiations, which can become quite heavy computing wise for certain sizes of key. We will see in chapter 5 that while a 1024-bit key size is easily manageable by full software implementation, hardware offloading become a necessity for 4096-bit key sizes. Moreover, 1024-bit parameter size, both RSA and Diffie-Hellman, are disallowed by the NIST recommandations since 2013 [6].

2.4 Network and VPN

Computer networks are described by two models: the TCP/IP stack and the OSI model, each splitting the network workflow into more or less abstract layers. The RFC 1122 [9] defines the TCP/IP, and the international standard ISO/IEC 7498-1 [19] defines the OSI stack as in the table 2.1. The main difference between the two is the application layer of the TCP/IP stack which corresponds to the three upper layers of the OSI model. Some references, such as Tanenbaum and Wetherall [44], conceptually split the TCP/IP link layer into an additional physical layer.

TCP/IP layering		OSI model	
Layer	Protocols	Layer	Protocols
Application	FTP, SSH	Application	FTP
		Presentation	ASCII, JPEG
		Session	RPC, PAP
Transport	TCP, UDP	Transport	TCP, UDP
Internet	IP, ICMP, IPsec	Network	IP, ICMP, IPsec
Link	PPP, MAC, Ethernet, L2TP	Data link	PPP, MAC, Ethernet, L2TP
		Physical	USB, DSL, IEEE 802.11

Table 2.1 – **TCP/IP and OSI model comparison:** They are globaly the same, except for the application layer of the TCP/IP stack which merge togheter the three upper layers of the OSI model. Between parenthsis are examples of protocols resting on each layer.

Frankel et al. [16] define a VPN as: “virtual network built on top of existing physical networks that can provide a secure communications mechanism for data and control information transmitted between networks."

There exist several major implementations of VPN: SSL, IPsec, L2TP and PPTP. The later was developed by a vendor consortium leaded by Microsoft and proposed in the RFC 2637 and will not be discussed further.

L2TP (Layer 2 Tunneling Protocol) is a protocol that simply offers a tunnel without providing confidentiality [27]. A common use case is to combine IPsec and L2TP, as defined in RFC 3193 [34].

The present work will focus and SSL/TLS and IPsec, the figure **??** showing their respective integration in the network stack.

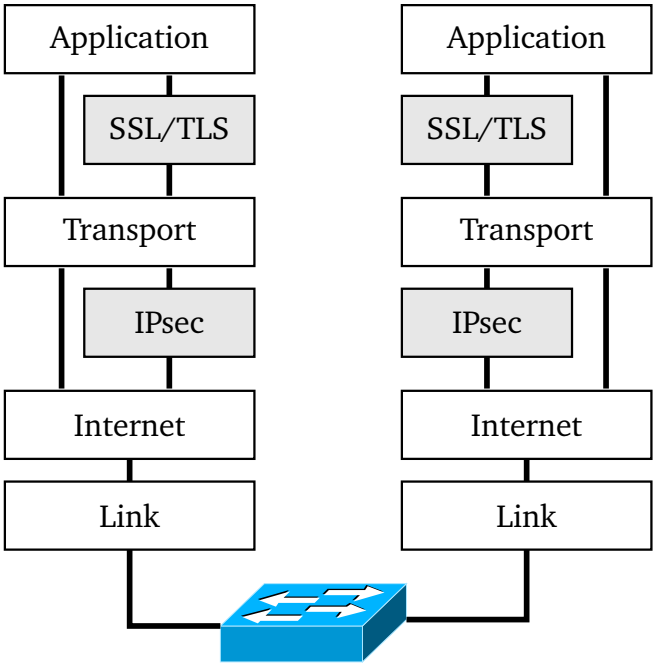


Figure 2.6 – **SSL/TLS and IPsec integration into the TCP/IP network stack:** SSL/TLS works in the application layer in order to secure the transport layer, whilst IPsec secures the Internet layer.

2.4.1 SSL/TLS

Secure Socket Layer (SSL) and Transport Layer Security (TLS) are two protocols ensuring secure communications from the application layer, securing the transport layer. The last version of SSL (version 3.0) dates from 1996 [17] and has been found a critical vulnerability by Google in 2014 [33]. As a consequence, SSL 3.0 support is being dropped, and the IETF is working on a depracation of the standard [7].
TLS is the successor of SSL is at version 1.2 [47], the next version being developed by the IETF [37]. TLS uses X.509 certificates for the asymmetric cryptography used during the key exchange. Those certificates come in pair: one public and a second private, and rely on a chain of trust for the validity of the certificate. Usually, the term certificate is use for the public certificate, and private key for the corresponding private certificate.

The figure 2.7 shows the messages exchanged during the establishement of a connection between two peers.

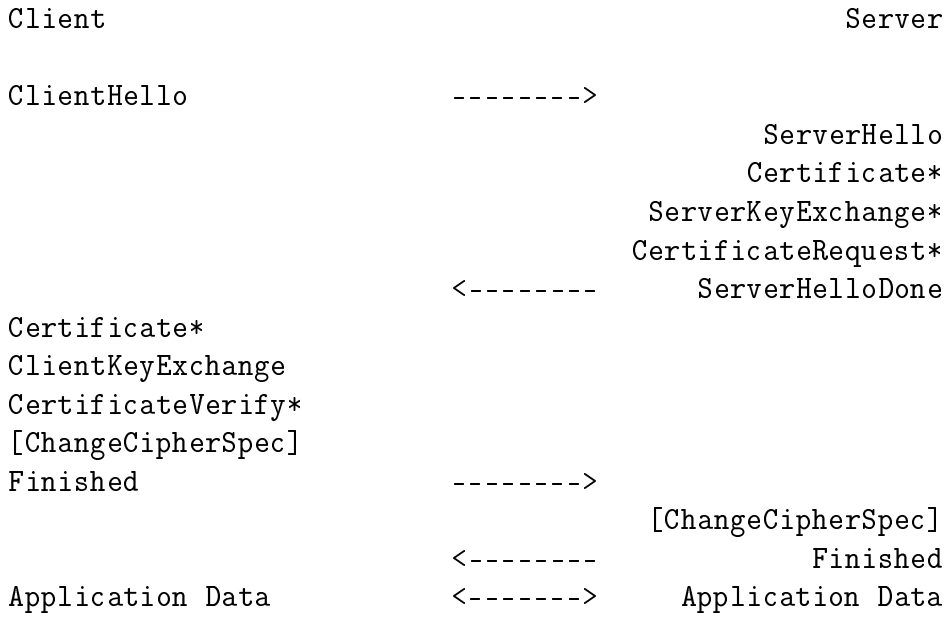


Figure 2.7 – **TLS handshake:** The * denotes messages that are sent only in some situations. Reproduce from [47].

1. Client: ClientHello. The client sends a series of specifications to the server, such as the protocol version and the supported cipher suites.
2. Server: ServerHello and others. The server replies by setting the protocol and the cipher suite to use. It can send its own certificate and request the client's. In some cases, the server can already begin the key echange protocol (*e.g.* Diffie-Hellman).
3. Client: ClientKeyExchange and others. The client veryfies the validity of the server certificate, if any, and sends its own. If necessary, the client may have to change its cipher specification based on the constraints of the server, or request a new specification from the server. The client can also participate in the key exchange protocol.
4. Server: Finished. If the client asked for a new cipher specification, the server can set a new one. Otherwise, the exchange is simply finished and the connection has been successfully established.

Once the connection is established, the peers can exchange application data using the symmetric key negociated during the handshake.

2.4.2 IPsec

IPsec is a protocol whose implementation is a modification of the IP stack in the kernel space. It is a network/internet level security; it examines incomming IP packets and checks if there exists a security association with the destination, and decrypt it on-the-fly if necessary.

IPsec consist of three components [45], each defined in their respective RFC:

Traffic protocols Encapsulating Security Payload (ESP, [23]) and Authentication Header (AH, [22]).
Key management Internet Key Exchange (IKEv2, [21]).
Policy Security Policy Database (SPD, [24]) and Security Association (SA [24]).

As Paterson [35] present them, the SAs are used as repository for cryptographic parameters, and the SPD defines the policies to apply. If the SPD can be populated by hand, typically in a configuration file, the SA management should be left to an automated mechanism: the IKEv2.
The IKEv2 protocol uses four different types of exchanges to fulfill its role:

- `IKE_SA_INIT`: The two peers agree on cryptographic parameters to use. Among these is a Diffie-Hellman shared secret from which are derived symmetric keys used for a special IKE SA.
- `IKE_AUTH`: Once the peers are protected by the IKE SA, they can authenticate themselves to each other and add a first SA in the SA database (SADB). The protocol supports three types of authentication methods:
  - Digital signature using a PKI.
  - MAC using a pre-established secret key.
  - Extensible Authentication Protocol (EAP) defined in Aboba et al. [1].

  At this point, the connection is fully established and ready to use.
- `CREATE_CHILD_SA`: Used to create new SAs between the two peers. It may involve new generations of DH secrets to ensure perfect forward secrecy.
- `INFORMATIONAL`: Exchange of management information.

The figure 2.9 shows the structure of an ESP packet. The ESP header includes two fields: a Security Parameter Index (SPI) and a Sequence Number (SN) The SPI is a value on which both actors agreed on during the key exchange protocol, identifying the SA to use for the communication. The SN is an anti-replay feature, an integer incrementing with each packet processed by the SA. This way, the SA will reject any packet that does not have a SN in the current windows, avoiding an intruder to resend a previously captured packet on the network. As the SN is covered by the integrity, the opponent can not change it without having to change the Integrity Check Value (ICV, *a.k.a.* the message digest). This authentication covers the whole packet, whilst the confidentiality excludes the ESP header.

The ESP trailer includes a padding and two 1-byte fields: the padding length and the next header value, which can specify if the packet is dummy or real. The padding is needed when using block ciphers to obtain a payload which length is an integral multiple of the block length.
The ESP payload includes an optional IV and an optional Traffic Flow Confidentiality (TFC) padding. The TFC padding allows to modify the length of the packet with dummy data, somewhat hiding the true nature of the payload. It differs from the standard padding because its length is not limited to 255 bytes like the latter.
AH is the same as ESP, except is does not offer encryption features, only the authentication. Since ESP can use a null cipher, it arguably offers the same features as AH. The existance of AH is actually historical; when the protocol was developed in the 1990s, there were laws in the United States and other countries preventing the export of product including encryption features. Having an authentication only counterparty allowed such product to be exported without encryption capabilities.

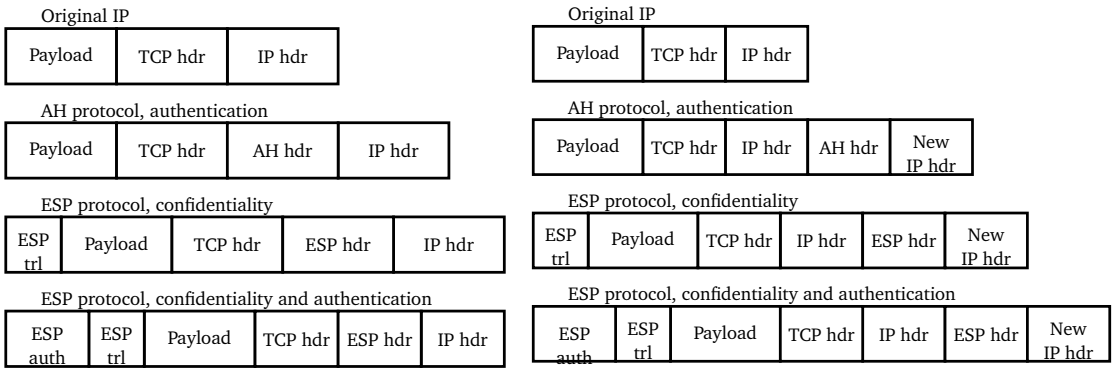
IPsec can be deployed in two modes:

- Transport for end-to-end service and protecting the payload. IPsec examines the incomming payload, checks it has not been tampered with and pass it to the next layer.
- Tunnel protects the entire IP packet by encapsulating it into a new IP packet. As the destination contained in the new IP header can be different from the original, this mode can be used to manage gateways.

The figure 2.8 shows the space overhead imposed by AH and ESP in transport and tunnel modes.

Both protocols add 10 bytes of fixed size fields (SPI, SN, pad length and next header). The other fields are of variable length, or even optional, such as the optional authentication field of minimum 12 bytes, the optional IV of minimum 4 bytes and the padding between 0 and 255 bytes long. Xenakis et al. [48] reached the conclusion in their paper that the use of lightweight – and deprecated – encryption schemes (*e.g.* HMAC-MD5 and DES) hardly had any impact on the throughput of the system and on the delay of the packets. Hence, the space overhead of IPsec is negligible when using more serious schemes (*e.g.* AES-256-CBC with HMAC-SHA-256).

RFC 7321 [29] defines the support for only three AES modes: CBC, CTR and GCM.



(a) Transport (b) Tunnel

Figure 2.8 – **IPsec (a) transport and (b) tunnel overheads:** Tunnel mode adds a custom IP header and moves the AH/ESP header in front of the original IP header. Note: “trl" stands for “trailer", “hdr" for “header", reproduced from [48]

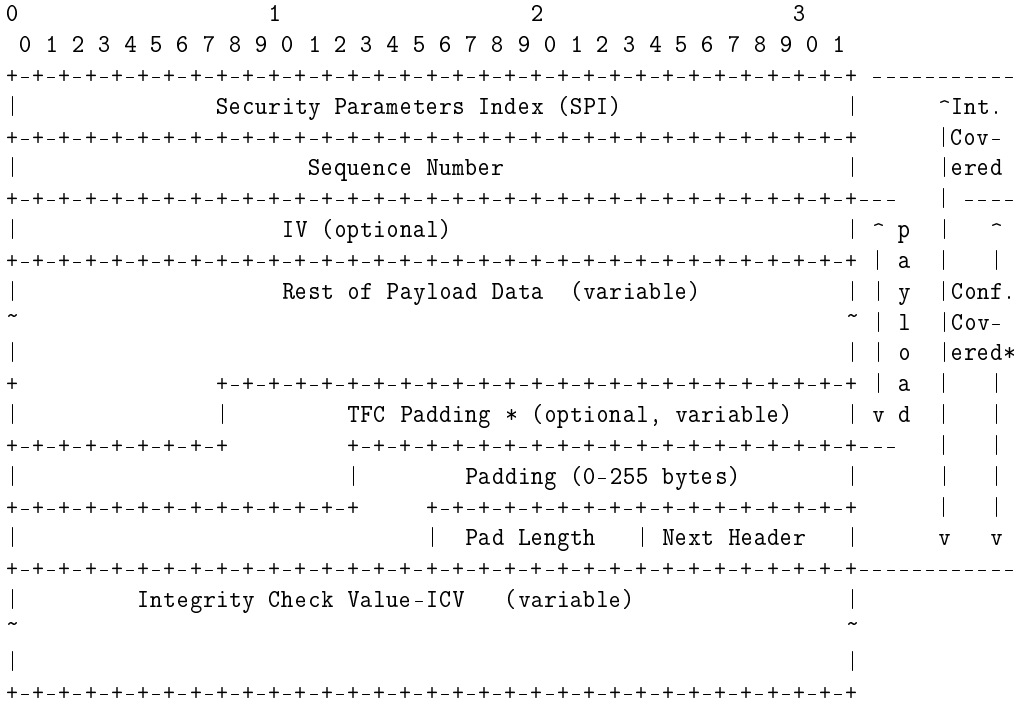


Figure 2.9 – **ESP packet structure:** as defined in RFC 4303 [23].

Chapter 3

Implementation

This chapter will present how we implemented the protocols presented in chapter 2 on board. The figure 3.1 shows the generic data flow in the operating system through the user and kernel spaces. We will first detail the software part, with the OpenVPN and openSSH as the applications, and OpenSSL as a cryptographic library on which both applications rely. Then will come strongswan, a user-space abstraction layer giving access to IPsec and we will see how different it is from an OpenVPN implementation. Lastly, the standard Linux cryptographic kernel modules and network drivers will be listed and briefly discussed.

Before closing the chapter, we will present the two main IP cores to which the cryptographic operations will be offloaded and the associated drivers.

3.1 Software

In order to implement all the required cryptographic protocol, several softwares are needed. Their role is summarized in table 3.1.

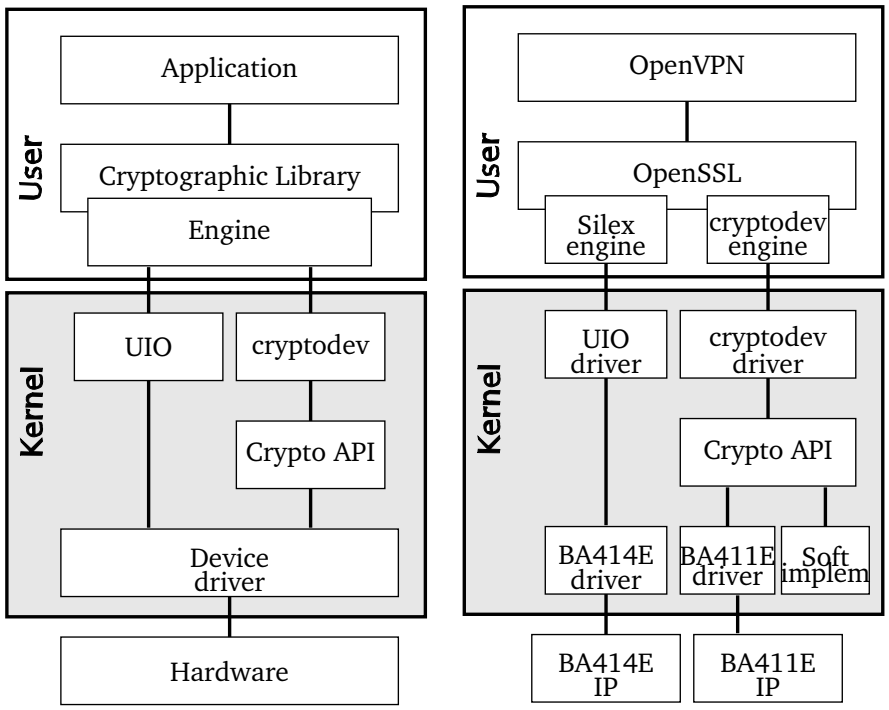
OpenVPN	SSL/TLS
OpenSSH	SSH
OpenSSL	Cryptographic ciphers implementation
Strongswan	IPsec

Table 3.1 – **Role of the softwares:** Each software implements different protocols. OpenSSL is particular since it implements the cryptographic algorithms themselves and is used by the other applications.

3.1.1 OpenVPN

OpenVPN is a very popular VPN solution relying on the application layer of the TCP/IP stack by using the SSL/TLS protocols.

It uses regular TCP/UDP network protocols, which can be an advantage over IPsec if the ISP decides to block the latter.



(a) Generic (b) Specific

Figure 3.1 – **OS data paths:** (a) as a generic abstraction and (b) with some specific blocks replaced with custom implementations.

OpenVPN offers two different TUN/TAP virtual interfaces:

TUN Virtual layer 3 IP tunnel, seen as a point-to-point interface from the kernel point of view.

TAP Virtual layer 2 ethernet tunnel.

Since OpenVPN aims at network security at the application level, it needs virtual network interface to read and write the data. The `tun` adapter will be used when OpenVPN wants to establish a point-to-point connection between two client, while the `tap` device is used when there is one server acting as a bridge, hence managing lots of clients at the same time.

OpenVPN as two choices when it wants to sends its data over the internet: either encapsulating its IP packet into TCP or UDP packets. The problem of encapsulating IP into TCP is that inside IP packets, there already are TCP frames. The reason is that IP has been designed to work on unreliable networks, and TCP includes in its standard protocols to retry and drop packets. Hence, encapsulating an IP packet into a TCP packet produces a redundancy by nesting reliability layers. UDP is the unreliable counterparty of TCP, offering a better alternative for encapsulation. If we refer to the figure 3.2, `iperf` would send its regular `IP(TCP)` packet over the virtual `tun` device, which will then compress, fragment

and encrypt the frame before encaspulating it into a new UDP packet that will finally be sent through the physical `eth0` interface.

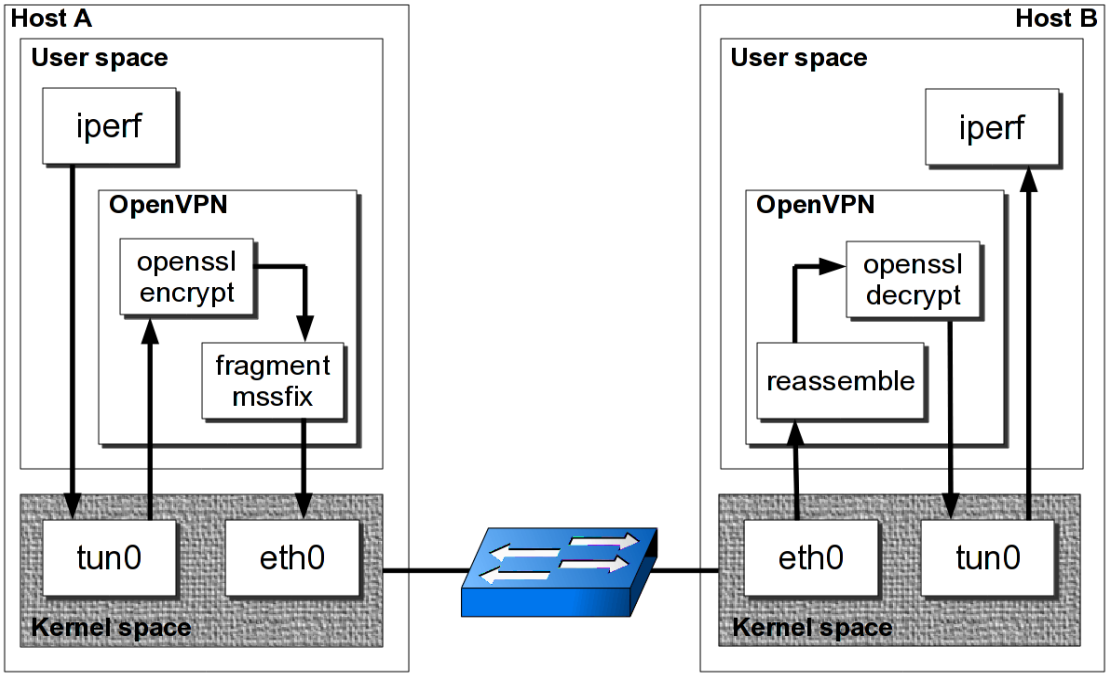


Figure 3.2 – **OpenVPN packet flow:** it shows two hosts using `iperf` inside an OpenVPN tunnel. This figure comes from the OpenVPN documentation [12] and has inverted the encryption and fragmentation.

At this point, it is important to note that if the figure 3.2 shows a workflow in which the encryption taking place before the fragmentation, a quick look at the source code proves the opposite. A sample of the said code has been writen down in the listing 3.1.

An other way to look at this workflow is to completely isolate the OpenVPN block from both hosts, and exporting it to an independent device such as a router. The hosts would be stripped off the virtual interface – `tun0` would become physical, and would communicate normally over the internet, at least from their point of view. All their communications would be routed by the router of their local area network, incomming on a physical `tun0` interface and outputed to the internet on the `eth0` interface. Such implementation are used when an adminstrator wishes to secure a whole local area network without bloating each client with a VPN application. In this case, there exist router firmware packages running OpenVPN, such as DD-WRT and OpenWRT. Whatever the method used, the crucial point is that the VPN tunnel usage is transparent to the application.

OpenVPN relies on a cryptographic library, OpenSSL in our case, for all the security operations: encryption, signature, certificate management, etc.

```
1  void encrypt_sign (struct context *c, bool comp_frag)
2  {
3    struct context_buffers *b = c->c2.buffers;
4    const uint8_t *orig_buf = c->c2.buf.data;
5
6    if (comp_frag){
7    /* Compress the packet. */
8      if (lzo_defined (&c->c2.lzo_compwork))
9        lzo_compress (&c->c2.buf, b->lzo_compress_buf, &c->c2.lzo_compwork, &c->c2
             .frame);
10   /* Fragment the packet. */
11     if (c->c2.fragment)
12       fragment_outgoing (c->c2.fragment, &c->c2.buf, &c->c2.frame_fragment);
13   }
14
15   /* Encrypt the packet and write an optional HMAC signature. */
16   openvpn_encrypt (&c->c2.buf, b->encrypt_buf, &c->c2.crypto_options, &c->c2.
         frame);
17 }
```

Listing 3.1 – openvpn compress then encrypt – sample from `forward.c`. It clearly shows that the order of operations in the packet workflow is compression, then fragmentation and finally encryption.

Note that if one wants to use the `none` cipher with a version of OpenVPN ealier than 2.3.7, he would have to update it using a community patch [20].

3.1.2 OpenSSH

OpenSSH relies on an external cryptographic library for all its security operations. Up until the version 6.7 published in october 2014, it had to be compiled against OpenSSL. However, after the infamous security vulnerability heartbleed in april 2014, the developpers took a step to move towards LibreSSL, a fork of OpenSSL managed by OpenBSD developers. Still, there is no official support for any other cryptographic library.

If OpenSSH does support most of OpenSSL ciphers by default, it takes some liberties such as disabling the CBC encryption mode and removing the support of no MAC during a transmission.
The first customization is a response to an old vulnarability[1]. Even if it is not recommended, the user can still enable this mode by explicitly configuring it in the `sshd_config` options.
The second has been taken to prevent the user to strip himself from data authenticity and integrity. However, in the case of testing and benchmarking, skipping

[1] A vulnerability note as been issued by Carnegie Mellon University Computer Emergency Response Team in early 2009 (last revision) [10], in response to a research of University of London [2] presenting a plaintext-recovering attack against SSH when CBC mode is used, but the OpenSSH update took only place in october 2014.

the MAC can be interresting, especially if it is not offloaded in hardware such as the encryption in our case. In order to overcome this limitation, we wrote a patch to apply on OpenSSH 6.7, available in appendix A.

3.1.3 OpenSSL

OpenSSL is an implementation of the SSL/TLS protocols. Unlike OpenVPN, it does implement all the cryptographic suites supported by the protocols in C and assembly. OpenSSL his mainly used as a cryptographic library and offers an high-level interface called EVP to be used by other applications.

OpenSSL can extend its features by supporting cyrptographic engines. Those engines offer custom implementation for specific ciphers. In a sense, a crypto engine is to OpenSSL what the crypto kernel modules are to the Linux Crypto API.
A widely used engine is one of Intel's, proposing acceleration for the AES operations on its processor using the AES-NI instruction set.

Two engines will be used in this work:

- cryptodev, to gain access to the BA411E from the user space.
- silex engine, a custom-made engine to access the BA414E from user space using UIO.

It is to be noted that the present work uses an implementation with all the debug flags activated. Figure 3.3 shows that if it does have an impact on the performance, it is negligible for our tests: in the worst case of the benchmark, the throughput drops only by 2.4%. Moreover, the benchmark maximizes this difference by doing only OpenSSL operations. When OpenSSL will have to share the CPU with other applications, the loss will be even less noticeable.

3.1.4 Strongswan

Strongswan is a full implementation of IPsec relying on the kernel drivers for the networking part, on the Linux Crypto API for the cryptographic part, and on user space crypto libraries for the connection negociation. An other popular implementation is ipsec-tools, but its development lags behind modern Linux and is not up-to-date with the 3.14 Linux kernel headers, making its cross-compilation difficult. Strongswan as two advantages: it has a tremendous and exhaustive documentation, and its user interface is straightforward. Once configured, a simple `ipsec start && ipsec up <connection>` on both sides is enough to create a ready-to-use VPN.

The figure 3.4 illustrates the workflow of Alice communicating with Bob via an IPsec ESP tunnel. The XFRM (read "transform") framework is implementing IPsec and handles the incomming and outgoing packets for established VPN connections [39]. Its name comes from the fact that the kernel transforms packet frames to incorporate IPsec security. Depending on the configuration, XFRM uses

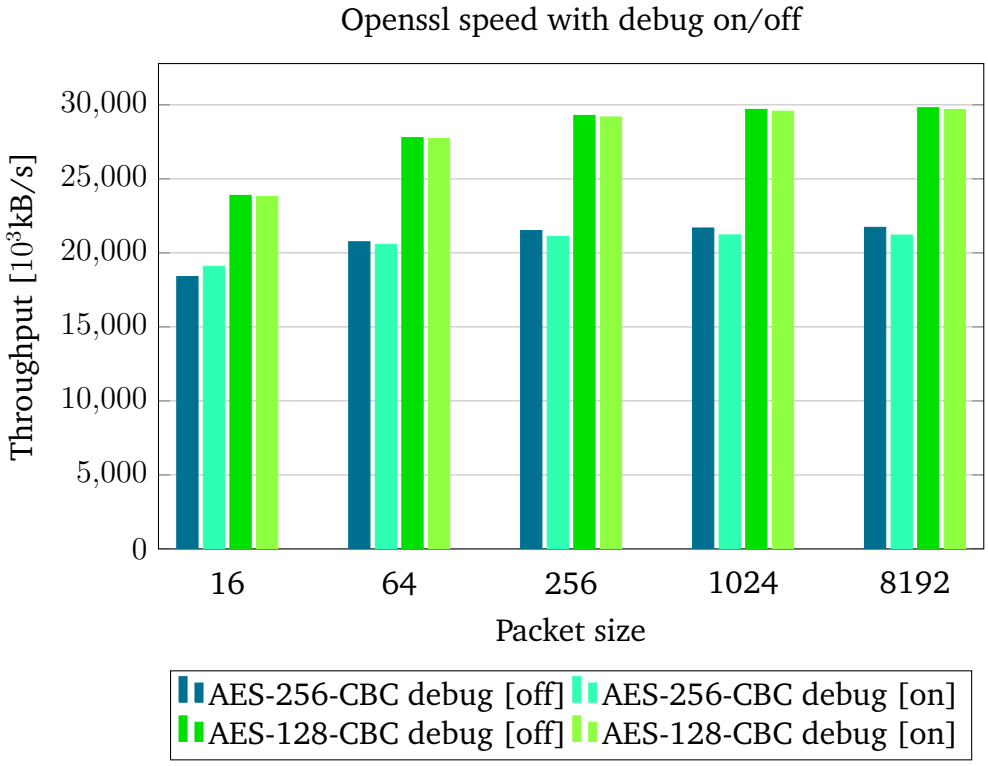


Figure 3.3 – **OpenSSL debugging benchmark:** Software benchmark of Openssl speed for AES mode CBC, with 128- and 256-bit keys, debugging flags (de)activated at compilation (`-fno-inline -g -marm`). The throughput difference ranges from 0.2% and 2.4% , and is more marked for larger keysize, as more debugging data needs to be generated.

the AH or ESP kernel module, which in turn calls the crypto API to encrypt and/or sign the IP packet.

We can also clearly see one of the main advantages of IPsec: it works in the kernel space. Since it does not require a virtual network interface like OpenVPN, the only transfer between the user/kernel space happens when the former wishes to send a packet on the network, passing it to the later – or *vice versa* for incomming packets.

3.1.5 Linux drivers

Several kernel modules are needed to implement the various cryptographic algorithm in software. The GCM alone needs five different modules, and IPsec three others. The following description addresses all the kernel modules required to run all the use cases presented in this work.

`aes_arm` Assembly implementation of AES. This version is optimized to use the ARMv7 instruction set.

`sha256-generic` C implementation of SHA-256.

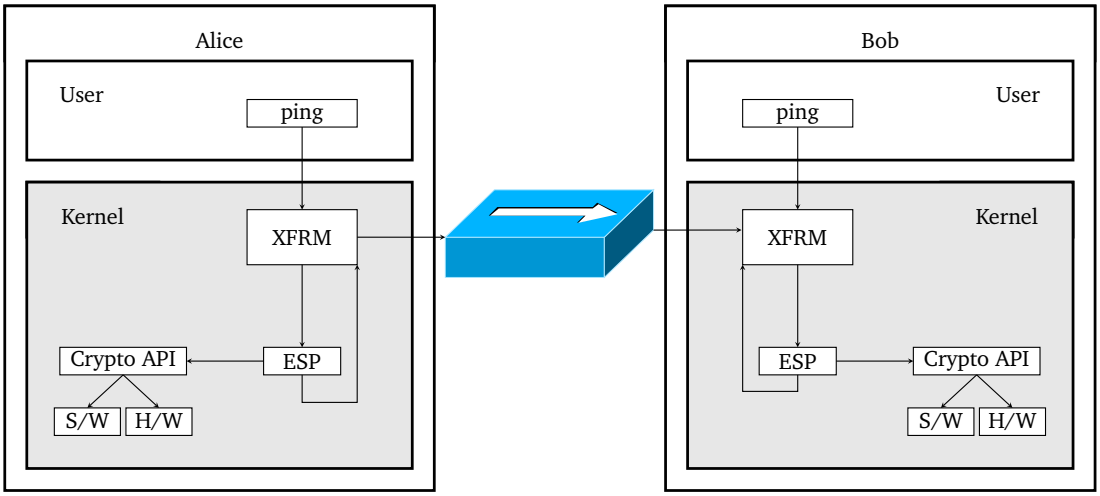


Figure 3.4 – **IPsec user/kernel space workflow, using `ping` as a test case.:**

`gfmult128` Multiplication in $GF(2^128)$, needed by the GCM mode.
`ghash` GHASH function needed by the authentication of the GCM mode.
`seqiv` Sequence IV number generator, needed by the CTR and GCM modes.
`cbc` C based CBC mode.
`gcm` C base GCM mode.
`crypto_null` Null cipher. This basically does nothing on the plaintext, but it is still needed to propose the interface in the kernel.
`tun` TUN/TAP network device, needed by OpenVPN.
`xfrm_user` XFRM operations.
`esp4` IPv4 ESP implementation. The AH counterparty is also available in the `ah4` module.
`ipcomp` IP compression module. Needed by IPsec if the option is activated in the configuration.
`cryptodev` Creates `/dev/crypto`, giving access to the crypto API in the user space.
`uio` User-space I/O driver, allowing to access the hardware memory from the user space, needed by our implementation of the BA414E driver.

3.2 Offload

The table 3.2 summarizes the ciphers supported by the two Barco Silex' IP cores used in this work.

For this tests, only one instance of each IP core will be programmed on the FPGA. As all the tests will be single-threaded, having two or more instances of the same IP core would not improve the performance, as they could not be used in parallel anyway.

IP core	Ciphers
BA414E	RSA, DH, DSA, ECDSA, ECDH
BA411E	AES modes CBC, CTR, GCM, CCM, CFB, OFB, XTS, ECB

Table 3.2 – **Summary of the ciphers supported by two of Barco Silex' IP cores.:**

3.2.1 BA411E Driver

This driver takes the rightern path of figure 3.1b, which is the best because the most versatile. By pluging the driver into the Linux Crypto API, we offer a standard kernel interface that can be used by any other kernel driver, such as the ESP driver, or user-space applications via the cryptodev driver and OpenSSL engine.

However, at the current state of development, the user-space and the kernel applications do not share the same driver. The former is using an IRQ-based driver, whilst the latter is actively polling the hardware. The reason why the IRQ-based driver can not be used by kernel-space application is because the current implementation uses sleep methods in order to free ressources while the hardware is busy. The problem is that the functions that called the device driver do not support to sleep during their execution. Hence, when the device driver tells the scheduler to put him into sleep mode, the parent function panics and crashes the kernel, forcing the system to reboot. An alternative is under development and is further discussed in the last part of this work, in section 6.1.

Whichever method is used to fetch the state of the hardware, the rest of the driver follow the exact same canvas as the default software implementation of the different AES modes.

3.2.2 BA414E Driver and Silex engine

At the moment of development, there is no asymmetric cryptography interface in the Linux Crypto API[2].

Pressed by exhibitions and the complexity of interfacing the functions with the crypto API, Barco Silex' developmement team took the choice to access the hardware from user space.
The BA414E can thus not be accessed using the same path as the BA411E and a user-space driver will be needed, that is the leftern path of figure 3.1b. The complete work flow thus needs three elements:

- A device driver handling the IRQ, initializing the device and creating a simple file system that will be used from the user space.
- The UIO kernel module supervising the memory access of the user space.
- The custom OpenSSL silex engine that translates the requests from OpenSSL into operations processable by the IP core.

[2]A request for comment patch as been submitted to the Linux kernel mailing list [42] in late April 2015, proposing a standard interface for public key encryption in the crypto API.

Chapter 4

Test protocol

This chapter presents the platform on which the tests will be undertaken, as well as the tests themselves.

4.1 Experimental setup

The experimental environment is built around a standard x86 host and an ARM Cortex-A9 alongside an Altera Cyclone V FPGA as the target. Both are linked toghether through a network capped by 100Mbps switches. Both stations have gigabits ethernet interface and could hence be directly connected to each other, but in that case the communication would be limited by the I/O transfers of the storage units – a hard drive disk in one case, an micro-SD card in the second – on which we can not depend to set a constant throughput limitation, as it is highly influenced by the data block size and general health of the support.

4.1.1 x86 host

The desktop host runs on Windows 7 Professional 64-bit, but a virtual machine using a Linux distribution is used for the developement and testings.

OS Ubuntu 12.04 LTS, kernel 3.16
CPU Intel Core-i5 M560, 2.67GHz (two logical core out of four)
RAM 1GB DDR3

4.1.2 Altera Socrates SoCFPGA

OS Yocto project, kernel 3.14
CPU Dual core ARM Cortex-A9, 800MHz
RAM 1GB DDR3
FPGA Altera Cyclone V

4.1.3 ARM DS-5 Streamline

Streamline is a toolsuite included in ARM Development Studio. By installing a kernel module and running a daemon on the board, it can fetch a huge amount of data without over-loading the system. This tool is able to gather information on the CPU utilisation, the amount of context switches, of interrupts, of memory used, and much more. If the applications, libraries and kernel modules have been compiled with the required debugging options, it can also build the call path of a run, giving extensive statistics on the most ressource-hungry functions.

If it has close to no impact on the performance of the board, the same does not go for the hosting machine. A serious drawback is his memory consumption that, when combined with the ressources used by the virtual machine, bring the hosting machine to its knees.
The user should also be aware that if Streamline if a very powerfull and versatile tool, it is not reliable at very high resolution. The data should not be analyzed in windows smaller than a millisecond.

4.2 Test cases

The aim of this work is to test the implementations under realistic situations. To reach this goal, three test cases will be used:

- Lots of connections without fetching any data.
- Latency of the connection. This test can also illustrate the transfer of small amount of data.
- Large file transfer on an established connection.

Each of those tests will go from a raw run, without any encryption nor authentication serving as a comparison ground, gradually increasing security up to a complete encryption/authentication scheme.

The table 4.1 summarizes the softwares used and tjeir respective version.
As AES mode CBC and SHA-2 are widely used, they will be used with a serious security paramter using keys of 256-bit. Some test case will also experiment some implementation of AES-GCM to compare its performance and justify the need for its offloading in hardware.

For the IPsec and OpenVPN implementation, the cipher/authentication pair null/null will be used to quantify the overhead of the encapsulation. It should

OpenVPN	2.3.6
OpenSSL	1.0.2a
Strongswan	5.3.0
OpenSSH	6.7p1

Table 4.1 – **Software used for the test:** note that OpenSSH was updated with the patch in appendix A.

however not be forgotten that it is not to be used as a production configuration. As the RFC 7321 [29, pg. 7] sates for IPsec: “Note that while authentication and encryption can each be ‘NULL’, they MUST NOT both be ‘NULL”".

We will use two types of drivers for the BA411E: an interuption-based for OpenVPN and OpenSSH, and a polling-based for IPsec.

All the tests will be conducted with the ESP protocol in tunnel mode, so that we have the worst case scenario; as AH imposes a smaller overhead, the perfomance could only be better.

4.3 TLS Connections

This benchmark is done only with OpenVPN 2.3.6. Since there is no standard support for those operation in the kernel yet, it would not have made sense to use IPsec, since it would have had to fallback to OpenSSL, then following the same path as OpenVPN do. As soon as the public key operations can be plugged into the Linux Crypto API, this use case should however be tested.

For this use case, the board is configured in server mode, so that it can accept connections from any client. The server does not connect with a specific client and waits for incomming connection requests (see listing B.2). The virtual machine will then execute then clients in parallel using the script 4.1. The only option differentiating the clients is their IP address and port number. Otherwise, all the clients share the same basic configuration file (see listing B.1), which tell them to renogociate a new connection every second. Hence, if a connection could be made with no delay and if the processes scheduling were ideal, the server would have to address 600 connections per minute.

```
1 #!/bin/bash
2
3 timeout=120
4 rsa=1024
5 remote_ip=150.158.232.208
6
7 for i in ‘seq 1 10‘; do
8   timeout ${timeout} openvpn --config client_${i}_${rsa}.ovpn --verb 2 --remote
        ${remote_ip} &
9 done
```

Listing 4.1 – Script starting ten clients in parallel who will stress the server.

As this experiment can be very unstable and vastly depends on the operating system scheduling, each test case has been repeated five times to ensure stable results.

The security parameter tested is a standard TLS-DHE-RSA, hence forcing the peers the renogociate a new shared secret and ephemeral Diffie-Hellman parameters at each connection attempt.

4.4 Response time – latency

This use case exchanges ICMP request of various sizes via the `ping` command. The initiating peer sends an ICMP echo request to the remote peer, which then answers with an ICMP echo reply.
For each packet size, 1000 requests were flooded to the board, that is "*outputs packets as fast as they come back or one hundred times per second, whichever is more*", according to the `ping` command manual.

The following loop shows the options used for the test as well as the payload sizes:

```
1 for i in 56 1000 8000 16000; do
2   sudo ping -f -s ${i} -c 1000 150.158.232.241
3 done
```

4.5 File transfer

The file transfered is an uncompressed block of 128MB of random data generated using the following command:

```
1   $ head -c $((1024*1024*128)) /dev/urandom > heavy.file
```

For OpenSSH, the commmand `scp` will be used to transfer the data securely over an SSH tunnel. As for OpenVPN and IPsec, a tunnel will be established beforehand, and then the simple `ftp` command will allow the client to fetch the file on the server. In order to use custom temporary security configuration with `scp`, the following command can be used:

```
1   $ ./scp -S /path/to/patched_openssh/bin/ssh -o Ciphers=aes256-cbc -o MACs=
        none@barco.com root@150.158.232.241:heavy.file .
```

All the results on the throughput and CPU usage take into account only the file transfer. The server authentication, the connection establishement and any other securoty negociation is ignored in this particular test case.

Chapter 5

Results and Analysis

This chapter gather the results of the use cases presented in chapter 4. Three use cases are considered: establishing TLS connections, exchanging ICMP requests to compute the latency and transfering a file over a secure channel. Depending on the case, several schemes are analysed: DHE-RSA, AES-CBC, AES-GCM and SHA-256.

It is to be noted that all the CPU usage values are taken from the single one loaded core. Although the platform is dual-core, all applications are used single threaded, either by design, such as OpenVPN, or by choice to obtain comparable results.

5.1 TLS connections

If the ten clients could connect instantaneously to the server every second, the maximum number of connections would be 600 per minute. However, a certain connection time has to be taken into account. Those are summarized in table 5.1.

		Connection time [s]
RSA-1024	soft	0.041921
	BA411E	0.020312
RSA-2048	soft	0.202945
	BA411E	0.039965
RSA-4096	soft	1.436743
	BA411E	0.183533

Table 5.1 – **OpenVPN connection time:** time necessary to establish an aes-256-cbc connection with DHE.

It already shows that when using the hardware, the connection latency is divided by 2 for low security RSA, and up to by 7.8 for higher security parameters. The figure 5.1 shows the number of TLS connections per minute for three RSA exponent sizes: 1024-, 2048- and 4096-bit. The higher the exponent size, the higher the performance boost.

	RSA-1024			RSA-2048			RSA-4096		
	Con.		CPU	Con.		CPU	Con.		CPU
Soft	445.4	x1.14	40.32	155.6	x2.70	92.14	19.6	x5.89	81.97
BA414E	509.3		13.29	420.9		4.82	115.5		4.34

Table 5.2 – **TLS connections per minute:** measures obtained with ten clients concurently connecting to an OpenVPN server.

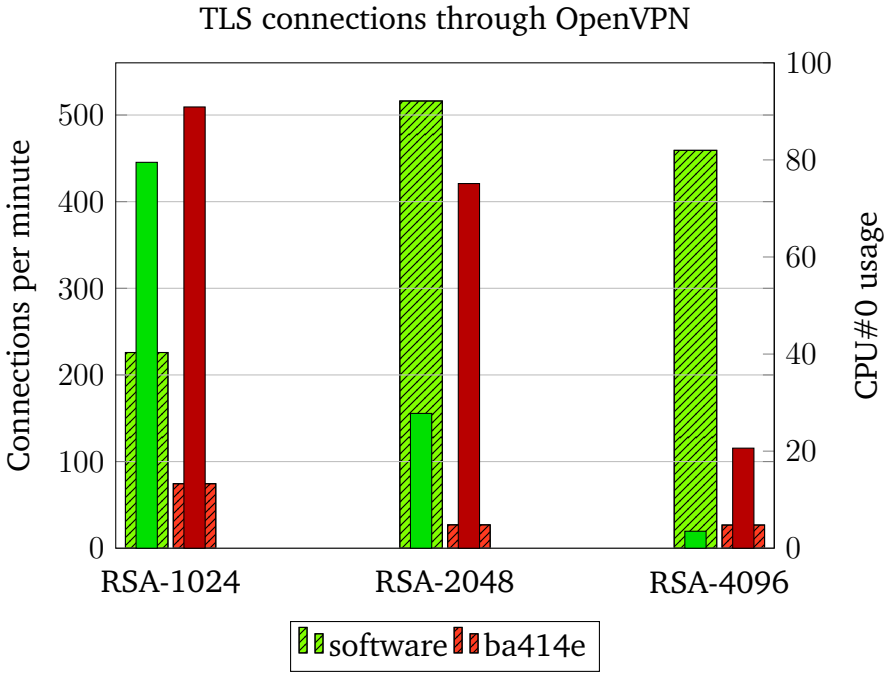


Figure 5.1 – **TLS connections per minute:** The background stripped bars are the CPU usage. Raw data in table 5.2.

For RSA-1024, the results are mitigated: a poor performance increase, but already less than half the CPU usage. It should be noted that at this point, the number of clients is probably too low to push the configuration to its limits. It is however an interesting comparison case with the next level of security: RSA-2048.
RSA-2048 is a much more common configuration, espacially since the NIST deprecated RSA-1024 in 2013. The full software implementation is visibly affected by the increase of the exponent size: the CPU usage doubles and the server processes three time less connections. At the same time, the hardware loses less than 20% connections for a third of the CPU usage.
The results obtained for RSA-4096 can be interpreted similarly to those of RSA-2048, except that the CPU usage is exactly the same for th hardware configuration. One way too look at those results is to directly compare the raw performance, and the hardware can then process almost six times more connections per minute than the software. However, this is only half of it, since it leaves the

CPU usage drop aside. If we look at the efficiency, the software can process 0.24 connection per percentage of CPU usage, whilst the hardware can process 24 of them. The efficiency is thus multipled by a factor 1000.

Such interesting results, particularly regarding the CPU usage, are possible because at least 87% of the operations are RSA and Diffie-Hellman operations, which are entirely offloaded in hardware. Nevertheless, OpenVPN still needs to proceed to some extra computations (such as SHA-1 integrity), and the hardware operations are not instantaneous, so the performance gain can only be that high.

5.2 Response time – latency

The following tests are conducted after the connection has been established, so as the clients do not need to undergo any new key negociation.

5.2.1 OpenVPN

The figures 5.3 and 5.2 shows the results for different payload sizes, which raw results are presented in table 5.3

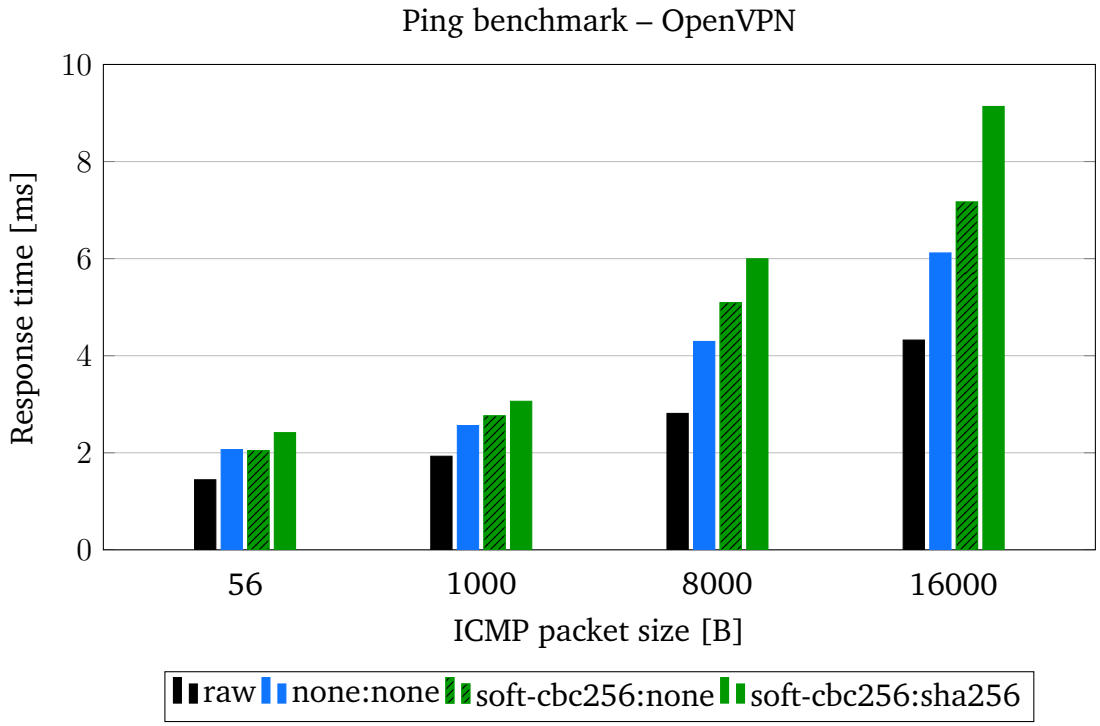


Figure 5.2 – **OpenVN: ping average response time – software:** Compares the response for different packet size when using a full-software implementation. OpenVPN adds up to 53% extra delay when not using encryption nor authentication (none:none).

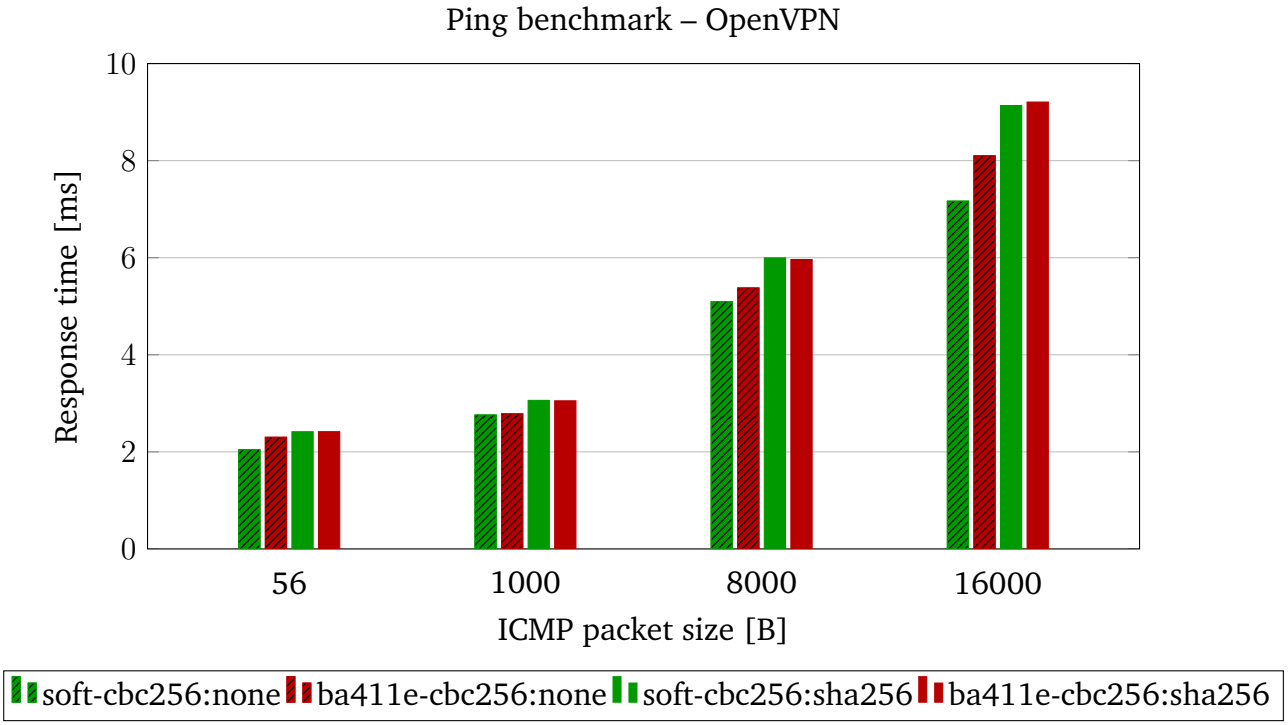


Figure 5.3 – **OpenVN: ping average response time – Software/Hardware:** When using encryption only, the transfers between the user space and the hardware make the hardware implementation perform worse than the software. Adding the authentication fills the gap by making the encryption synchronize with the authentication.

	56B	1000B	8000B	16000B
raw	1.444	1.929	2.811	4.322
none-none	2.066	2.561	4.293	6.117
soft-cbc256-none	2.044	2.760	5.092	7.166
ba411e-cbc256-none	2.301	2.783	5.377	8.009
soft-cbc256-sha256	2.415	3.061	5.997	9.135
ba411e-cbc256-sha256	2.416	3.052	5.963	9.207

Table 5.3 – **OpenVN: ping average response time:** the latency is given in milliseconds.

Firstly, is the impact of OpenVPN's operation on the delay: when no encryption nor authentication is used, the exchange is 33% to 53% longer. This is simply because the packet has to go through a virtual interface, hence even without the security time overhead, the latency is much longer.

When combining encryption and authentication, the software and hardware implementation are neck and neck. Eventhough the MAC is not offload, the hardware implementation could have performed better. The reason is not only all the

packets have to go through a virtual interface in the kernel, the payload also has to be sent to the hardware through the kernel. In the end, a packet will undergo two full round trips between the user and the kernel space, that is one more than a regular software implementation, plus a final transfer to the physical interface. It is interresting to note that for packets of 16000 bytes, the hardware adds an extra 12% to the latency when using only encryption, but only 1% when adding the authentication. In the second case, the synchronization between the confidentiality and the authentication slows the process down up to the point that the user/kernel space transfers are less an overhead than with confidentiality alone. If the MAC were to be done in hardware as well, the extra delay would probably be even larger as the data would have to be transfered to two different IP cores via two different drivers[1].
All those transfer make the offload useless in such a case in term of raw performance. The CPU usage is not even worth mentioning as the operation is only a few milliseconds long.

5.2.2 IPsec

The figures 5.5b and 5.5 studies the same payload size as OpenVPN, but with an extra software implementation of AES mode GCM. The raw results are in table 5.4.

	56B	1000B	8000B	16000B
raw	1.444	1.929	2.811	4.322
none-none	1.545	2.045	2.997	4.703
soft-cbc256-none	1.581	2.134	3.910	6.426
ba411e-cbc256-none	1.639	2.080	3.445	5.345
soft-cbc256-sha256	1.645	2.322	4.762	7.975
ba411e-cbc256-sha256	1.635	2.246	4.170	6.929
soft-gcm256	1.651	2.388	5.383	9.241

Table 5.4 – **IPsec: ping average response time:** the latency is given in milliseconds. Some results for the GCM mode in hardware are non-existant.

In this case, the time overhead imposed by IPsec is not larger than 9%, corroborating the results of Xenakis et al. [48].

When combining encryption and authentication, the hardware implementation steadily takes the advantage over the software with the encrease of the payload length. If they both have the same latency at default payload size, the hardware is up to 13% faster for 16000 bytes payloads.

As for the GCM mode, the support in the BA411E driver is highly experimental and is not functional enough to be integrated in those results. However, the process of GCM packets in hardware should be at least as fast as CBC packets.

[1] A unifed driver is under development, gathering the encryption and authentication in the same kernel module.

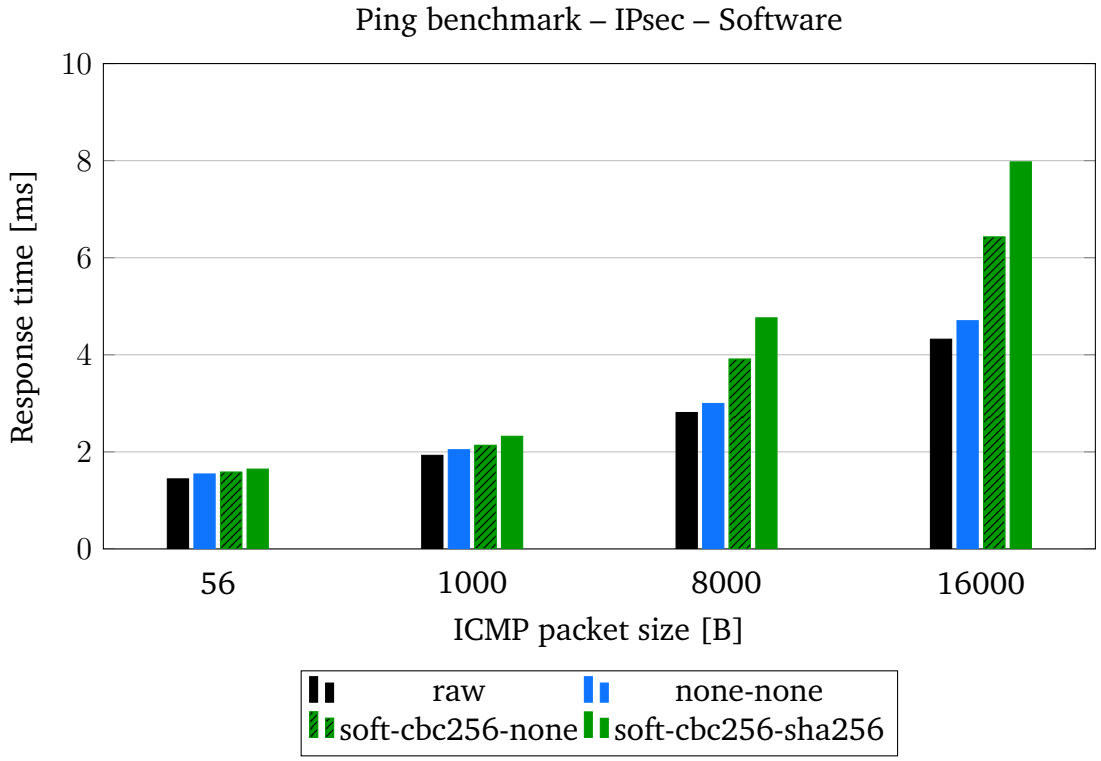


Figure 5.4 – **IPsec: ping average response time – Software:** The overhead imposed by IPsec when not using encryption nor authentication is 9%.

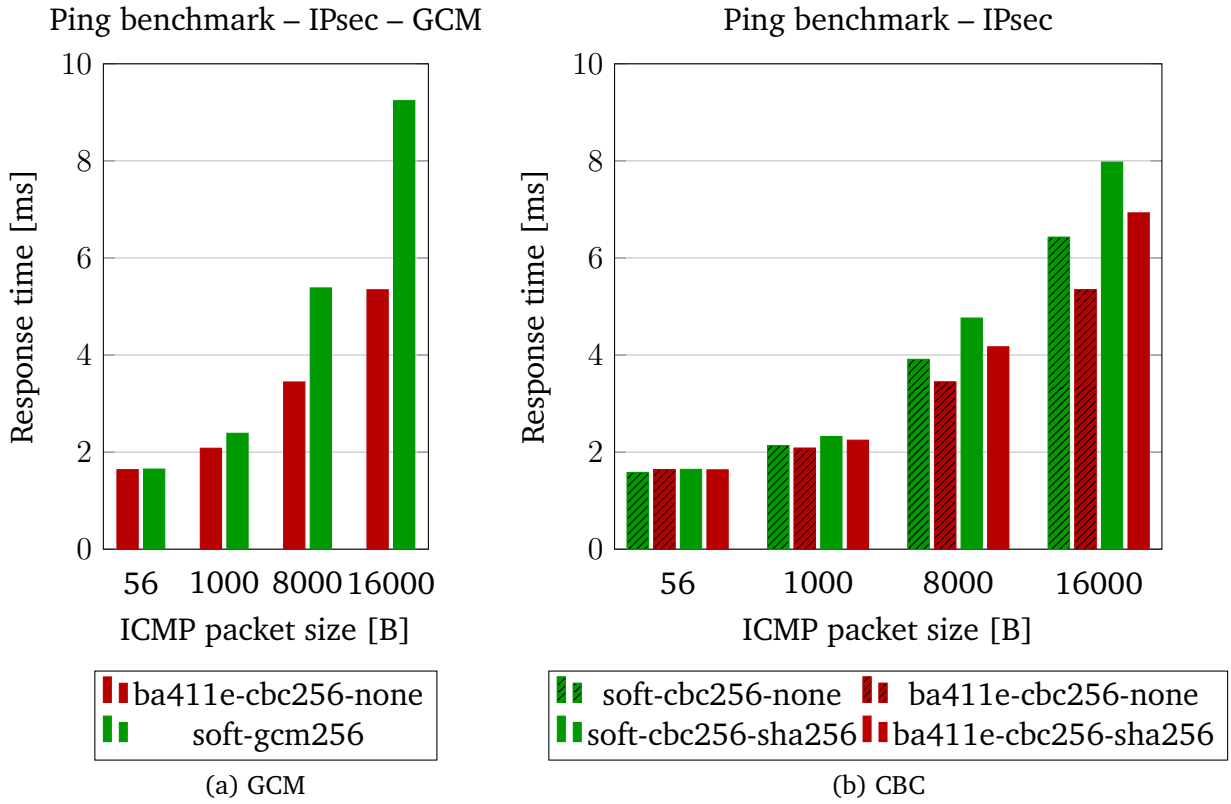
We can thus compare the software implementation of the GCM mode with a hardware implementation of CBC only (wihtout authentication). The figure 5.5a shows an improved delay by up to 42% when using the hardware.

5.2.3 Comparison

The figure 5.6 summarizes the results for OpenVPN and IPsec for the most realistic use case: the combination of encryption and authentication.

The first main difference is the overhead imposed by each method: 53% for OpenVPN and only 9% for IPsec. Even if those configurations are not realistic, it puts forward the advantage of directly working in the kernel as does IPsec.

As for the combination of encryption and authentication, IPsec is between 13% and 32% faster than OpenVPN. IPsec loses its advantage with the increase of the payload size, the time lost when moving around the data between the user and the kernel space being less important compared to the securoty operations. In hardware, as OpenVPN could not take advantage of the accelaration of the encryption, IPsec is even faster, ranging from 25% to 33% faster than OpenVPN.



(a) GCM

(b) CBC

Figure 5.5 – **IPsec: ping average response time – software/Hardware:**
(a) Processing GCM in hardware yields at least as good results as CBC. Comparing the CBC mode without authentication in hardware with the GCM mode in software thus makes sense.
(b) With or without authentication, the better results by the same margin.

5.3 File transfer

This use case studies the performance of a simple file transfer over three different secure implementations: OpenSSH, OpenVPN and IPsec. For each application, three encryption:authentication couples are considered: none:none, AES-256-CBC:none and AES-256-CBC:SHA-256.

5.3.1 OpenSSH

The figure 5.7 shows the results for a file transfer over an SSH tunnel. For this application, there is no none:none couple case inside the tunnel, as it would be the same as a transfer outside the tunnel. The raw results are gathered in table 5.5.

When only encryption is used, both implementation performs almost as well as outside the tunnel, but the software already saturates the CPU, whilst the hardware only uses half the same ressources. About 41% of the operations using

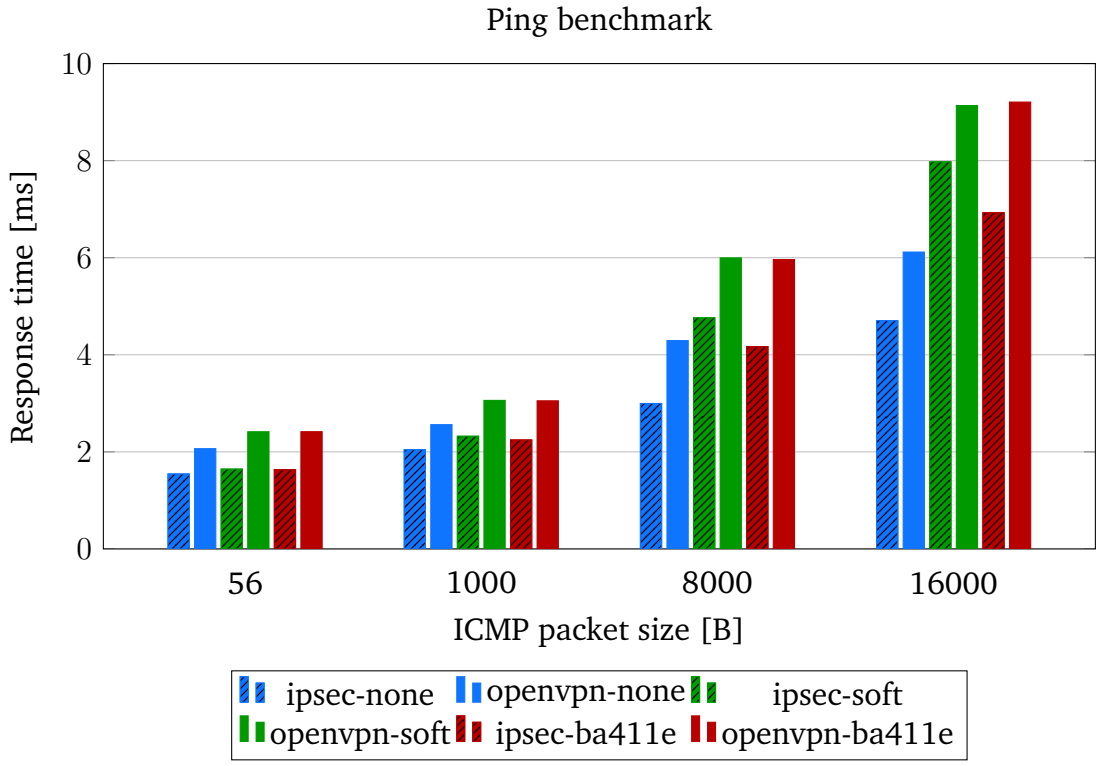


Figure 5.6 – **Ping average comparison:** All values are for AES-256-CBC with SHA-256. this graph shows two results: the comparison between IPsec and OpenVPN overhead without encryption nor authentication, and the performance of IPsec and OpenVPN in software and hardware. Globally, IPsec yields better results and propose the most significant hardware offload. Quick reading: the stripped bars are the IPsec results.

	none:none		aes256cbc:none		aes256cbc:sha256	
	Tp.	CPU	Tp.	CPU	Tp.	CPU
Out-of-tunnel	11.39	7.16	–	–	–	–
Software	–	–	10.89	96.26	8.19	89.29
BA414E	–	–	10.67	46.01	10.39	68.19

Table 5.5 – **File transfer over an SSH tunnel:** The throughput is in MB/s.

the CPU when accelarating with the hardware involve interuptions and kernel memory management.

Adding the authentication makes the software performance drop by 25%, as the CPU was already saturated without those extra operations. The hardware is also consistent, but as there were some ressources available, the throughput is merely affected, even if the MAC is not offloaded in hardware.

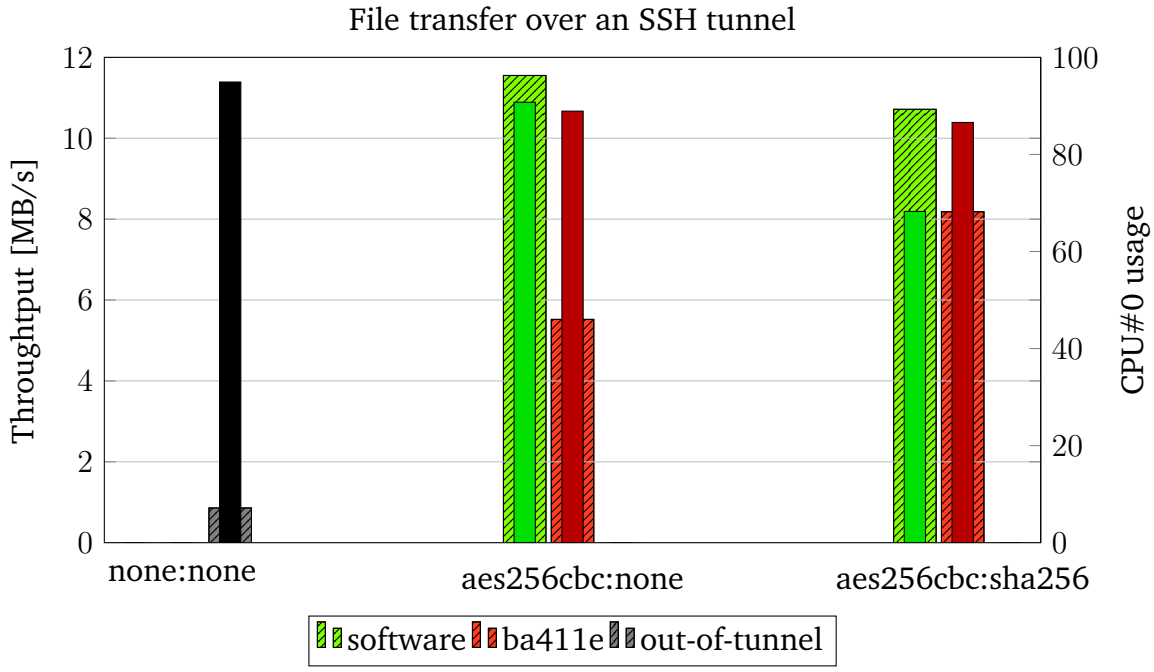


Figure 5.7 – **File transfer over an SSH tunnel:** The background stripped bars are the CPU usage.

5.3.2 OpenVPN

The figure 5.8 shows the performance of the file transfer over an OpenVPN tunnel. The data is gathered in table 5.6.

	none:none		aes256cbc:none		aes256cbc:sha256	
	Tp.	CPU	Tp.	CPU	Tp.	CPU
Out-of-tunnel	11.39	7.16	–	–	–	–
Inside tunnel	5.18	42.60	–	–	–	–
Software	–	–	4.78	76.60	3.87	76.03
BA414E	–	–	3.35	83.74	2.84	80.89

Table 5.6 – **FTP file transfer over an OpenVPN tunnel:** The throughput is in MB/s.

As it was already the case for the ping benchmark, the overhead imposed by the manipulation of OpenVPN is extremely heavy: the CPU usage jumps from 7.16% to 42.60%, and the throughput drops by 55%, from 11.39MB/s to 5.18MB/s.
Surprisingly enough, encrypting the data only changes the throughput by 8%, but the CPU usage is almost doubled. A fair interpretation would be that the encryption is no the bottleneck, the transfer bewteen the user and kernel mode, as well as the fragmentation, are. However for the hardware, the performance collapse to 3.35MB/s.

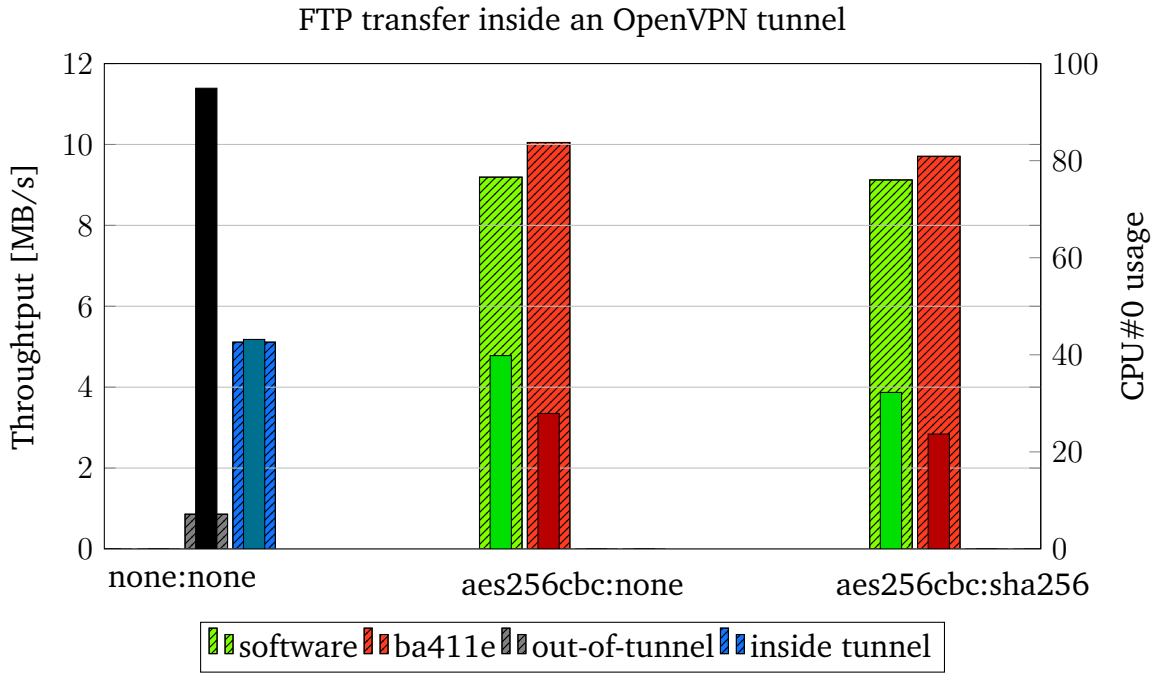


Figure 5.8 – **FTP file transfer over an OpenVPN tunnel:** The results inside the tunnel with no encryption nor authentication highlights the overhead OpenVPN adds to the transfer. From there, the throughput only change by a small extent. The poor hardware results also shows the heaviness of OpenVPN on hardware offloading. The background stripped bars are the CPU usage.

Adding a MAC computation aside the encryption lowers the performance in software and hardware by respectively 20% and 15%, for the same CPU usage as without authentication.

5.3.3 IPsec

The figure 5.9 gathers the results of the file transfer over the IPsec tunnel.

	none:none		aes256cbc:none		aes256cbc:sha256		aes256gcm	
	Tp.	CPU	Tp.	CPU	Tp.	CPU	Tp.	CPU
Out-of-tunnel	11.39	7.16	–	–	–	–	–	–
Inside tunnel	10.21	14.68	–	–	–	–	–	–
Software	–	–	8.83	63.74	6.47	74.64	5.09	89.66
BA414E	–	–	8.52	14.87	5.80	17.25	–	–

Table 5.7 – **FTP file transfer over an IPsec tunnel:** The throughput is in MB/s.

In this implementation, the IPsec overhead is only of 10%, close to the 9% of the ping overhead, and the CPU usage only double to reach 14.68%.

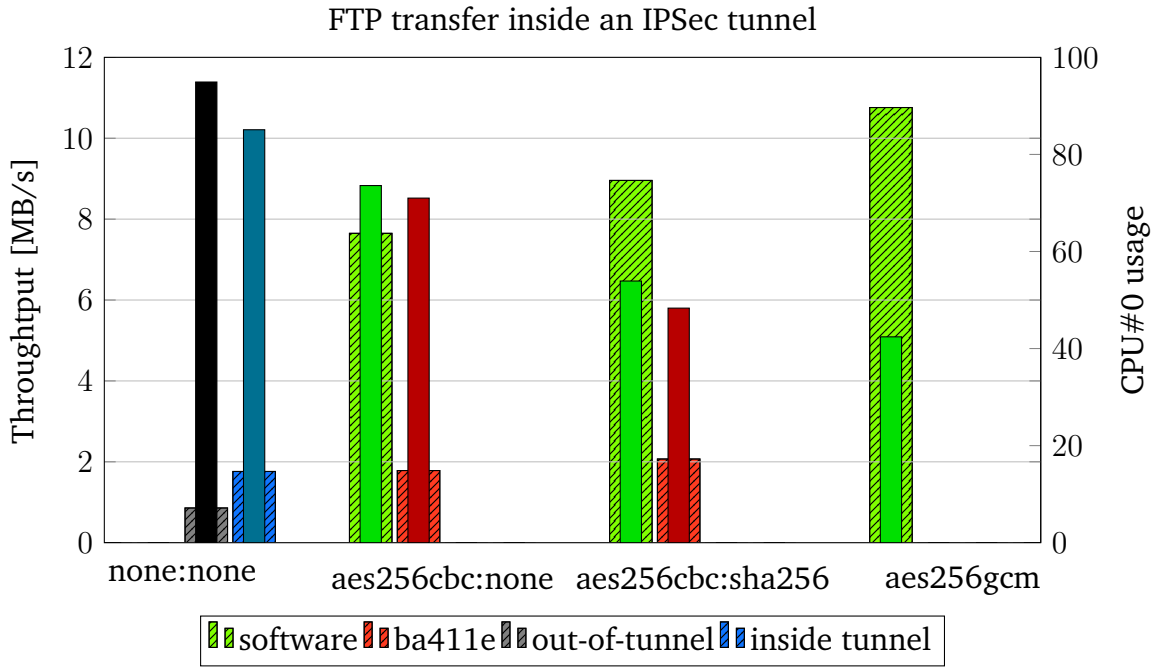


Figure 5.9 – **FTP file transfer over an IPsec tunnel:** The advantage of the hardware is a lower CPU usage, at the cost of a few percents lower throughput. The GCM mode is also tested to discuss its implementation in hardware. The background stripped bars are the CPU usage.

When encrypting the data, the CPU utilization explodes in the case of the software implementation, but stays at the same level for the hardware. As for the throughput, it lowers respectively by 14% and 17%. What could have been expected for the hardware implementation is a more stable throughput, but an increase of CPU usage for the few operations undergone by the BA411E driver, especially considering the fact that it uses active polling on the hardware. Thus, even if the hardware is the bottleneck, the CPU usage should be higher.
An explanation could be that the operating system prevent the driver to monopolizing the CPU with its polling and preempt it regularly, effectively lowering the ressource usage, but also limiting the performance.

Adding the authentication yields expected results for : an increase of CPU usage and lower throughput. Both implementation lose 2MB/s and a CPU usage increase of 20% and 16%.

In both cases, the hardware implementation exhibited slightly lower performance, but a CPU usage three to four times lower. On embedded platforms, this is as important as the raw performance, because lower CPU usage means underclocked CPU, resulting in a lower power consumption.

The softaware GCM results allows to open a discussion on this mode used in conjunction with IPsec. The GCM performance presented clearly shows a drop of

throughput and an increase of CPU usage, illustrating the fact that those operations are hard on the software. With an hardware offload, we could expect not only a drastic drop of the CPU usage, but an increase of throughput as well, since it's CPU-limited in those results. As we already discussed with the latency results in section 5.2.2, the results of a hardware implementation of GCM should be at least as good as those of AES-CBC wihtout authentication. If it were the case, we would increase the throughput by at least 67% and the decrease the CPU usage by at least 84%.

Note that the software results for the GCM are achieved using a C-based implementation of galois-field multiplications. As we saw in chapter 2, modern processor designers tend to add specialized instruction sets aimed at AES-GCM enhancement. Should further tests be conducted concerning IPsec paired with GCM, it would be wise to compare with an assembly implementation exploiting ARM NEON instruction set. Some are being developed [18, 14], but none have been committed to the Linux kernel repository yet.

5.3.4 Comparison

The figure 5.10 summarizes the file transfer performance for all the implementations, using the most realistic configuration tested: AES-256-CBC with SHA-256.

In this work, we have two objectives: the raw performance and the CPU usage. The first objetive is fulfilled by OpenSSH; even if it is a user-space application that needs to transfer the data through the kernel in order to use the hardware, it still outperforms the software by 27%.
The second objective is best fulfilled by IPsec with an efficiency of 0.34MB/s/% of CPU for the hardware, and 0.09MB/s/% of CPU for the software.

OpenVPN is however losing on all fronts. Not only it has the lowest throughput, but the CPU usage of the hardware is out the roof, especially when compared to the performance.
The table 5.8 shows to what size are fragmented the packets by OpenVPN and IPsec. A look at the figure 5.11 shows that fragmenting at small size is a bad idea, especially when the hardware is reached from the user space. OpenVPN sends one third of its packets at sizes that are highly ineffective, whilst IPsec manages it better and limit the small packets to less than 10%.

Some results have to be put into perspective with the fact that the implementation of SHA-256 is entirely C-based. A more recent one using assembly instructions optimized for the NEON SIMD instruction set of the ARMv7 core could be used and would most probably yield better results. The CPU usage of the software implementation would lower – even if not significantly – as for the hardware, it would be less limited by the software MAC counter part, and if the CPU usage would stay at the same level, we could expect a better throughput.

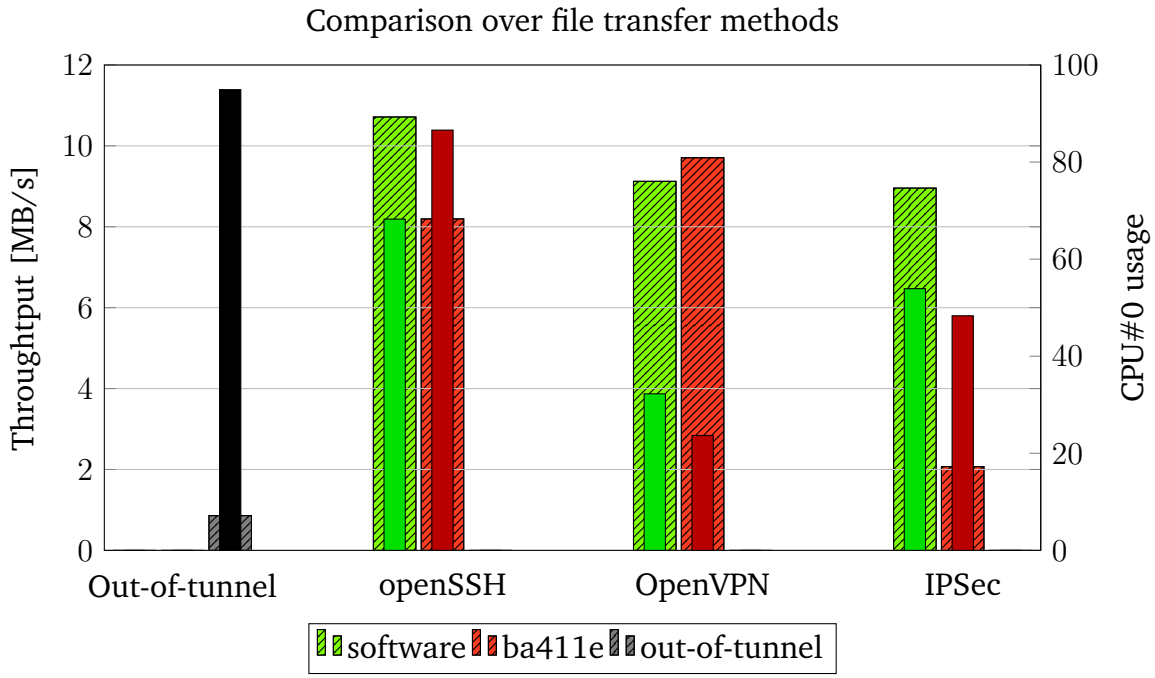


Figure 5.10 – **Comparison of file transfer methods:** Out of the three methods, OpenSSH is to be prefered is the raw performance is the objective, but IPsec offers an higher effeciency. At the same time, OpenVPN is worse on all fronts. The background stripped bars are the CPU usage.

	Length	Frequency
OpenVPN	80 – 159	33.04%
	1280 – 2559	66.90%
	other	0.06%
IPsec	80 – 159	9.42%
	1280 – 2559	89.84%
	other	0.74%

Table 5.8 – **Packet size frequency for an FTP transfer:** OpenVPN is less effective in its fragmentation, producing more packets of a smaller size, which perform worse on hardware.

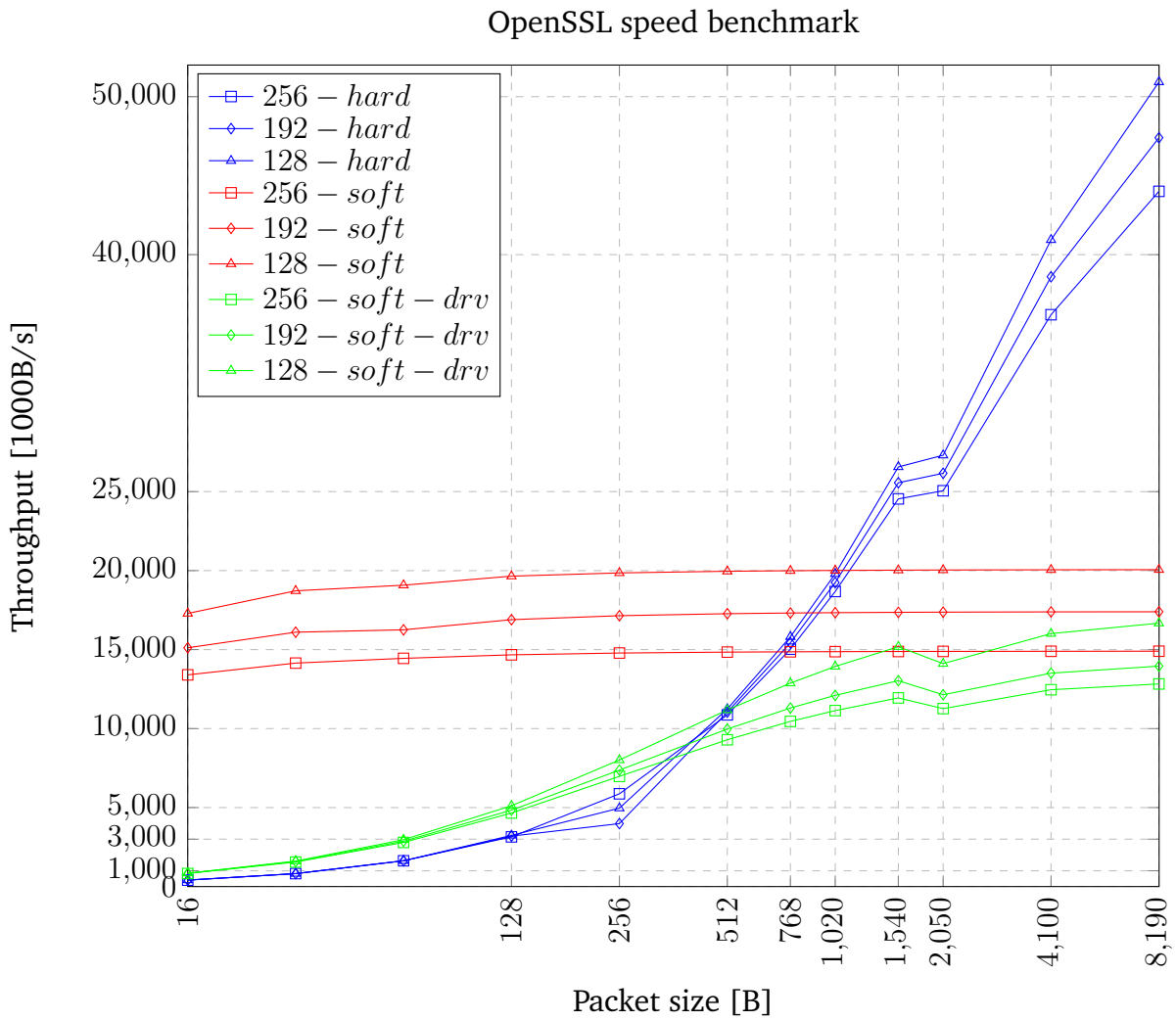


Figure 5.11 – **OpenSSL benchmark for AES-256-CBC:** compare a run of `openssl speed` for three implementation of AES: in software by OpenSSL (soft), in software by the standard Linux kernel module (soft-drv) and in hardware (hard). The last two have to go through cryptodev in order to be reached from the user space.

Chapter 6

Discussion and conclusion

6.1 Future work

Although the present work presents some promissing results, the implementation can certainly be improved in several ways and some further experiments should be conducted.

Driver improvement The driver of the BA411E can be made less ressources hungry by improving the initialisation of the descriptors and their linking, but the gain would not be significant enough to justify the time investment at this point. A better alternative would be to avoid descriptors altogether by modifiying the interface with the IP so that it can use the scatterlist directly. We would then spare a lot of DMA mapping instruction and thus some precious cycles on the software side.

As we already remarked in **??**, the use cases involving IPsec were conducted using a previous revision of the driver still activelly polling the IP for its results. A better and cleaner way to proceed is to use interruption routines, as shown in **??**. However, the kernel does not support their current implementation and panics upon usage. If one were to be willing to spend the time replacing the active polling by clean asynchronous interruptions, he should be aware of the overhead imposed by an interruption. In some cases, when the operation is just a few clock cycle long for the IP, an active polling could still be the better way to go. A more thorough comparison of the mutual trade-off deserves some investigation, and as a starting point, the packet size could be treated as a branching point between the two solutions.

Registering public key verification with the Linux Crypto API As we saw in **??**, the driver is already capable of offloading a large portion of public key operations to the IP, but only with very specific libraries at the time being – openSSL in our case. The next step is to register the very same operations with the Linux Crypto API so it can be used without having to rely on a custom openSSL engine. This feature would require to work very closely to the linux kernel developpe-

ment. Indeed, if the signature verification using public key cryptography has been available in the kernel since 2013, a public key encryption API has only been proposed in late April 2015 [42] and is still under request for comments.

Conditional offloading in cryptodev The figure 5.11 clearly shows a threshold on the packet size from which the hardware has a clear advantage, and below which the user mode software implementation is to go for. Using this breakeven point, one could patch the cryptodev cryptodev engine of OpenSSL to branch on the packet size, using the hardware if the packet is large enough, or fallback on the default software implementation otherwise. It would probably not be as trivial and beneficial as it may sound:

- the encryption contexts would need to be synchronized between the hardware and the software;
- as the breakeven point is around 1024kB, the performance for a network application would be very close to those of a full software implementation, knowing that the ethernet frame size, the MTU, is set by default at 1500 bytes.

Such a conditional offloading would be interresting for applications involving mainly very large packets and a few periodic smaller ones, like large data transfer between two hosts on a infrastructure supporting ethernet jumbo frames[1] with periodic ICMP heartbeat.

Disk encryption As the hardware is better used with larger blocks of data, disk encryption could be an interesting application to look into.

Cryptographic libraries OpenSSL is not the only cryptographic library available; GnuTLS is also a very popular alternative and supports cryptodev engines too.

However, one library definitely worth to keep an eye on is mbed TLS, formally known as PolarSSL, recently bought by ARM [3]. We can expect the future releases of this library to be more optimized for ARM platforms, and maybe the software footprint and overhead to be reduced.

Cryptodev If patches adding the GCM support to cryptodev have already been released, those are not compatible with Barco Silex' driver. Adapting the interface would open the GCM hardware offload to the whole user space applications park.

MAC offloading While the symetric and asymetric encryption ciphers IPs are usable from the operating system, the IP computing MACs does not have a usable driver yet. Wherever there is encryption, authentication is also needed. As such, any real day-to-day use case can not be fully offloaded to hardware yet, even if

[1] Jumbo frames have an ethernet MTU of 9000 bytes, whilst standard frames are set to 1500.

some tricks and patches allowed us to bypass this requirement. The implementation of the GCM mode, combining encryption and authentication, showed us that stopping relying on the software implementation of MACs would be a huge step forward.

Appendix A

OpenSSH patch

```
diff -rupN openssh-6.7p1_vanilla/digest.h openssh-6.7p1/digest.h
--- openssh-6.7p1_vanilla/digest.h 2014-07-03 13:25:04.000000000 +0200
+++ openssh-6.7p1/digest.h 2015-02-27 11:59:53.508495697 +0100
@@ -28,7 +28,8 @@
 #define SSH_DIGEST_SHA256 3
 #define SSH_DIGEST_SHA384 4
 #define SSH_DIGEST_SHA512 5
-#define SSH_DIGEST_MAX  6
+#define SSH_DIGEST_NONE  6
+#define SSH_DIGEST_MAX  7

 struct sshbuf;
 struct ssh_digest_ctx;
diff -rupN openssh-6.7p1_vanilla/digest-libc.c openssh-6.7p1/digest-libc.c
--- openssh-6.7p1_vanilla/digest-libc.c 2014-07-02 07:28:03.000000000 +0200
+++ openssh-6.7p1/digest-libc.c 2015-02-27 12:01:48.420494935 +0100
@@ -113,6 +113,16 @@ const struct ssh_digest digests[SSH_DIGE
    (md_init_fn *) SHA512Init,
    (md_update_fn *) SHA512Update,
    (md_final_fn *) SHA512Final
+ },
+ {
+   SSH_DIGEST_NONE,
+   "none@barco.com",
+   0,
+   0,
+   0,
+   NULL,
+   NULL,
+   NULL
   }
 };

diff -rupN openssh-6.7p1_vanilla/digest-openssl.c openssh-6.7p1/digest-openssl.
    c
--- openssh-6.7p1_vanilla/digest-openssl.c 2014-07-17 01:01:26.000000000 +0200
+++ openssh-6.7p1/digest-openssl.c 2015-02-27 12:00:24.812495489 +0100
@@ -59,6 +59,7 @@ const struct ssh_digest digests[] = {
```

```
38    { SSH_DIGEST_SHA256, "SHA256",  32, EVP_sha256 },
39    { SSH_DIGEST_SHA384, "SHA384", 48, EVP_sha384 },
40    { SSH_DIGEST_SHA512, "SHA512",  64, EVP_sha512 },
41  + { SSH_DIGEST_NONE,  "none@barco.com", 0, EVP_md_null },
42    { -1,    NULL,  0,  NULL },
43   };
44
45  diff -rupN openssh-6.7p1_vanilla/mac.c openssh-6.7p1/mac.c
46  --- openssh-6.7p1_vanilla/mac.c 2014-05-15 06:35:04.000000000 +0200
47  +++ openssh-6.7p1/mac.c 2015-02-27 12:01:00.204495255 +0100
48  @@ -87,6 +87,7 @@ static const struct macalg macs[] = {
49    { "hmac-ripemd160-etm@openssh.com", SSH_DIGEST, SSH_DIGEST_RIPEMD160, 0, 0,
        0, 1 },
50    { "umac-64-etm@openssh.com",  SSH_UMAC, 0, 0, 128, 64, 1 },
51    { "umac-128-etm@openssh.com",  SSH_UMAC128, 0, 0, 128, 128, 1 },
52  + { "none@barco.com",      SSH_DIGEST, SSH_DIGEST_NONE, 0, 0, 0, 0 },
53
54    { NULL,       0, 0, 0, 0, 0, 0 }
55   };
```

Appendix B

Configuration files

B.1 OpenVPN

```
# client.ovpn -- OpenVPN configuration file
lport 11101

dev tap01

ifconfig 10.4.0.101 255.255.255.0

tls-client

ca awesome-ca_4096.crt
cert client_1_4096-cert.crt
key client_1_4096-cert.key

reneg-sec 1

verb 2

#tls-cipher TLS-ECDHE-ECDSA-WITH-AES-256-CBC-SHA
tls-cipher TLS-DHE-RSA-WITH-AES-256-CBC-SHA

# sha256 or none
auth sha256

# AES-256-CBC or none
cipher AES-256-CBC
```

Listing B.1 – OpenVPN client configuration file.

```
# server.ovpn -- OpenVPN configuration file
dev tap1

ifconfig 10.4.0.1 255.255.255.0

tls-server

```

```
8  # dh4096.pem, dh2048.pem or dh1024.pem
9  dh dh4096.pem
10
11 # Make sure the CA and server certificates/private keys are of the right size:
       1024, 2048 or 4096 bits.
12 ca awesome-ca.crt
13 cert server_4096-cert.crt
14 key server_4096-cert.key
15
16 # Server does not renegociate the connection.
17 reneg-sec 0
18
19 verb 2
20
21 mode server
```

Listing B.2 – OpenVPN server configuration file.

B.2 Strongswan

```
1  # ipsec.conf - strongSwan IPsec configuration file
2
3  config setup
4
5  conn %default
6      ikelifetime=60m
7      keylife=20m
8      rekeymargin=3m
9      keyingtries=1
10     keyexchange=ikev2
11
12 conn host-host
13     esp=aes256-sha256!
14 #    esp=aes256-null!
15 #    esp=aes256gcm16!
16 #    esp=null!
17     ike=aes256-sha256-modp4096
18     left=150.158.232.242
19     leftcert=client_1_4096-cert.crt
20     leftid="C=AU, ST=Some-State, O=BARCO, CN=client_1_4096" # Should match the '
           subject' field of the certificate. To display it, you can do 'openssl
           x509 -text -in path/to/certificate -noout'.
21     leftfirewall=yes
22     right=150.158.232.241
23     rightid="C=AU, ST=Some-State, O=BARCO, CN=server_4096"
24     auto=add
```

Listing B.3 – Strongswan configuration file. Left is the local host, right the remote.

Bibliography

[1] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz. Extensible Authentication Protocol (EAP). RFC 3748, RFC Editor, June 2004. URL http://tools.ietf.org/rfc/rfc3748.txt.

[2] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against ssh. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09. IEEE Computer Society, 2009. doi: 10.1109/SP.2009.5.

[3] ARM. *ARM buys Leading IoT Security Company Offspark as it Expands its mbed Platform*. February 2015. http://www.arm.com/about/newsroom/arm-buys-leading-iot-security-company-offspark-as-it-expands-its-mbed-platform.php.

[4] Barco Silex. Multi-purpose aes ip core (cbc, ctr, cfb, ...). http://www.barco-silex.com/ip-cores/encryption-engine/BA411E-FLEX, . Visited May 27, 2015.

[5] Barco Silex. Scalable aes-gcm/gmac/ctr ip core. http://www.barco-silex.com/ip-cores/encryption-engine/BA415, . Visited May 27, 2015.

[6] Elaine Barker and Allen Roginsky. SP 800-131A. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. Technical report, Gaithersburg, MD, United States, January 2011.

[7] R. Barnes, M. Thomson, A. Pironti, and A. Langley. Deprecating Secure Sockets Layer Version 3.0 draft-ietf-tls-sslv3-diediedie-03. Internet-Draft draft-ietf-tls-sslv3-diediedie-03.txt, IETF Secretariat, April 2015.

[8] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In *Advances in Cryptology — ASIACRYPT 2000*. Springer Berlin Heidelberg, 2000.

[9] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, RFC Editor, October 1989. URL http://tools.ietf.org/rfc/rfc1122.txt.

[10] CERT. Vulnerability note vu#958563: Ssh cbc vulnerability. `http://www.kb.cert.org/vuls/id/958563`, 2009. Carnegie Mellon University.

[11] Community. Cryptodev-linux module . `http://cryptodev-linux.org/`. February 2015 for the last version.

[12] OpenVPN community. Optimizing performance on gigabit networks. https://community.openvpn.net/openvpn/wiki/Gigabit_Networks_Linux, 2011. visited May 25, 2015.

[13] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. ISBN 0596005903.

[14] Danilo Câmara, Conrado P. L. Gouvêa, Julio López, and Ricardo Dahab. Fast software polynomial multiplication on arm processors using the neon engine. In *Security Engineering and Intelligence Informatics*. Springer Berlin Heidelberg, 2013.

[15] Morris Dworkin. SP 800-38A. Recommendation for Block Cipher Modes of Operation. Technical report, Gaithersburg, MD, United States, 2001.

[16] Sheila E. Frankel, Karen Kent, Ryan Lewkowski, Angela D. Orebaugh, Ronald W. Ritchey, and Steven R. Sharma. SP 800-77. Guide to IPsec VPNs. Technical report, Gaithersburg, MD, United States, 2005.

[17] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, RFC Editor, August 2011. URL `http://tools.ietf.org/rfc/rfc6101.txt`. Republication of the last version of the protocol from Novermber 18, 1996.

[18] Conrado P. L. Gouvêa and Julio López. Implementing gcm on armv8. In *Lecture Notes in Computer Science*. Springer International Publishing, 2015.

[19] ISO/IEC. *ISO/IEC 7498-1. Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. ISO/IEC, second edition, 1996.

[20] Steffan Karger. Really fix cipher none patch. `https://community.openvpn.net/openvpn/attachment/ticket/473/0001-Really-fix-cipher-none.patch`, December 2014.

[21] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, RFC Editor, October 2014. URL `http://tools.ietf.org/rfc/rfc7296.txt`.

[22] S. Kent. IP Authentication Header. RFC 4302, RFC Editor, December 2005. URL `https://tools.ietf.org/rfc/rfc4302.txt`.

[23] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303, RFC Editor, December 2005. URL `http://tools.ietf.org/rfc/rfc4303.txt`.

[24] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, RFC Editor, December 2005. URL `https://tools.ietf.org/rfc/rfc4301.txt`.

[25] Hans J. Koch. Userspace I/O drivers in a realtime context. Linutronix GmbH, 2011.

[26] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor, February 1997. URL `http://tools.ietf.org/rfc/rfc2104.txt`.

[27] J. Lau, M. Townsley, and I. Goyret. Layer Two Tunneling Protocol - Version 3 (L2TPv3). RFC 3931, RFC Editor, March 2005. URL `http://tools.ietf.org/rfc/rfc3931.txt`.

[28] Olivier Markowitch. Infof405 – computer security. Lecture notes, 2013.

[29] D. McGrew and P. Hoffman. Cryptographic Algorithm Implementation Requirements and Usage Guidance for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 7321, RFC Editor, August 2014. URL `http://tools.ietf.org/rfc/rfc7321.txt`.

[30] D. A. McGrew and J. Viega. The galois/counter mode of operation (gcm). *NIST Modes Operation Symmetric Key Block Ciphers*, 2005.

[31] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, first edition, 1996.

[32] James Morris. The Linux Kernel Cryptographic API. *Linux Journal*, 2003.

[33] Bodo Möller, Thai Duong, and Krysztof Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback. Security advisory, Google, September 2014.

[34] B. Patel, B. Aboba, W. Dixon, G. Zorn, and S. Booth. Securing L2TP using IPsec. RFC 3193, RFC Editor, November 2001. URL `http://tools.ietf.org/rfc/rfc3193.txt`.

[35] Kenneth G. Paterson. A cryptographic tour of the IPsec standards. *Information Security Technical Report*, 11(2):72 – 81, 2006. ISSN 1363-4127.

[36] E. Rescorla. Diffie-Hellman Key Agreement Method. RFC 2631, RFC Editor, June 1999. URL `http://tools.ietf.org/rfc/rfc2631.txt`.

[37] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 draft-ietf-tls-tls13-05. Internet-Draft draft-ietf-tls-tls13-05.txt, IETF Secretariat, March 2015.

[38] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. doi: 10.1145/359340.359342.

[39] Rami Rosen. *Linux Kernel Networking*. Apress, 2014. ISBN 978-1-4302-6196-4.

[40] RSA Laboratories. PKCS #1 v2.2: RSA Cryptography Standard. Technical report, October 2012.

[41] Anand K. Santhanam. Towards Linux 2.6 – A look into the workings of the next new kernel. `http://www2.comp.ufscar.br/~helio/kernel_2.6/inside_kernel-2.6.html`, September 2003. Originaly published on IMB website.

[42] Tadeusz Struk. [patch rfc 0/2] crypto: Introduce public key encryption api. Request for comments on the Linux Kernel mailing list, April 2015. URL `https://lkml.org/lkml/2015/4/30/846`.

[43] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, 4th edition, 2014.

[44] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall Professional Technical Reference, 5th edition, 2011.

[45] Henk C. A. van Tilborg and Sushil Jajodia, editors. *Encyclopedia of Cryptography and Security*. Springer US, 2011. ISBN 978-1-4419-5905-8.

[46] Serge Vaudenay. Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology — EUROCRYPT 2002*. Springer Berlin Heidelberg, 2002.

[47] J. Viega and D. McGrew. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008. URL `https://tools.ietf.org/rfc/rfc5246.txt`.

[48] Christos Xenakis, Nikolaos Laoutaris, Lazaros Merakos, and Ioannis Stavrakakis. A generic characterization of the overheads imposed by IPsec and associated cryptographic algorithms. *Computer Networks*, 50(17):3225 – 3241, 2006. ISSN 1389-1286.

Glossary

AEAD Authenticated Encryption with Associated Data.
AH Authentication Header.

CBC Cipher Block Chaining.

DH Diffie-Hellman.
DHE Ephemeral Diffie-Hellman.
DMA Direct Memory Access.

ESP Encapsulating Security Payload.

FPGA Field-Prgrammable Gate Array.

GCM Galois Counter Mode.

ICV Integrity Check Value.
IKE Internet Key Exchange.
IRQ Interruption Request.
ISR Interrupt Service Routine.

MAC Message Authentication Code.
MDC Manipulation Detection Code.

PKI Public Key Infrastructure.

SA Security Association.
SADB Security Association Database.
SN Sequence Numbe.
SPD Security Policy Database.
SPI Security Parameter Index.

TFC Traffic Flow Confidentiality.

UIO Userspace Input/Output.

VPN Virtual Private Network.