



ÉCOLE  
POLYTECHNIQUE  
DE BRUXELLES



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 Delhay

Directeur

Professeur Frédéric Robert

Superviseur

Sébastien Rabou (Barco Silex)

Service

BEAMS

Année académique

2014 - 2015

# Acknowledgements

Thanks a bunch of people here:

- Frédéric Robert for his support
  - Sébastien Rabou for his insight
  - Batien Heneffe for his extensive help
- Anthony Debruyne for one awesome LaTeX trick.

## **Abstract**

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

### **Implementation of high level cryptographic protocol using a SoC platform**

This is an abstract.

## Résumé

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

# Contents

Aknowledgements	i
Abstract	ii
Résumé	iii
Contents	iv
List of Figures	vi
<b>1 Introduction</b>	<b>1</b>
1.1 Challenge . . . . .	1
1.2 Network security . . . . .	1
<b>2 Technical background</b>	<b>2</b>
2.1 Operating system . . . . .	2
2.2 FPGA . . . . .	2
2.3 Cryptography . . . . .	3
2.4 Network and VPN implementation . . . . .	5
<b>3 Presentation</b>	<b>6</b>
3.1 Experimental setup . . . . .	6
<b>4 Implementation</b>	<b>8</b>
4.1 Software . . . . .	9
4.2 Offload . . . . .	12
<b>5 Test protocol and Prediction</b>	<b>14</b>
5.1 File transfer . . . . .	14
<b>6 Results and Analysis</b>	<b>15</b>
6.1 TLS connections . . . . .	15
6.2 Response time – latency . . . . .	15
6.3 File transfer . . . . .	16
<b>7 Discussion and conclusion</b>	<b>21</b>
7.1 Future work . . . . .	21

<b>A</b>	<b>Raw data</b>	<b>24</b>
A.1	TLS connections . . . . .	24
<b>B</b>	<b>OpenSSH patch</b>	<b>25</b>
<b>C</b>	<b>Toolchain</b>	<b>27</b>
<b>D</b>	<b>Cross-compilation</b>	<b>28</b>
D.1	OpenSSL . . . . .	28
D.2	OpenVPN . . . . .	28
D.3	nginx . . . . .	28
D.4	pure-ftpd . . . . .	28
D.5	Strongswan . . . . .	28
D.6	kernel . . . . .	28
<b>E</b>	<b>Stuff</b>	<b>29</b>
	<b>Bibliography</b>	<b>30</b>

# List of Figures

2.1	IPSec transport (a) and tunnel (b) overheads . . . . .	5
4.1	OS data paths . . . . .	8
4.2	OpenVPN packet flow as advertized on the openVPN wiki. . . . .	9
4.3	OpenSSL debugging benchmark . . . . .	11
4.4	IPSec user/kernel space workflow, using ping as a test case. . . . .	12
6.1	TLS connections per second . . . . .	15
6.2	Ping min/avg/max response time . . . . .	16
6.3	Ping min/avg/max response time . . . . .	17
6.4	Ping average comparison . . . . .	18
6.5	file transfer over an SSH tunnel. The background stripped bars are the CPU usage. . . . .	18
6.6	FTP file transfer over an OpenVPN tunnel. The background stripped bars are the CPU usage. . . . .	19
6.7	FTP file transfer over an IPSec tunnel. The background stripped bars are the CPU usage. . . . .	19
6.8	Comparison of file transfer methods . . . . .	20

# **Chapter 1**

## **Introduction**

### **1.1 Challenge**

### **1.2 Network security**



# Chapter 2

## Technical background

This chapter will address the technical ground inherent to this work. First comes an overview of Linux operating systems, the distinction between user and kernel mode, and the design of device drivers.

Follow a quick presentation of FPGAs and how they can be driven from the operating system.

### 2.1 Operating system

#### 2.1.1 Device Driver

Userspace I/O [9]

There are two ways to get the results from a hardware device: either by using interruption, or by actively polling the device, relentlessly asking it 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 routine registered by the driver [5, chap. 10]. The second case is the easiest and is always guaranteed to work, but won't let go off the processor willingly, loading it at 100%, and avoiding any other task to be executed. Hopefully, modern monolithic kernels such as the Linux kernel from 2.6 provide preemptive scheduling [14], that is the scheduler interrupts the running task and assigns the processor resources it used to an other one. Hence, systems with a lot of processes in need for CPU resources would not be stalled, but it would not change anything if the only process heavily requesting processor time is the one using the driver.

### 2.2 FPGA

Driving from the OS: basically, 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 operation system, we build a bunch of scatterlist in their 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 descriptor and synchronize them with the DMA, it does not have to care about them anymore, the DMA is now in charge to send them to the device where registers are ready to read the incoming 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.

## 2.3 Cryptography

### 2.3.1 Symetric cryptography

Talk about encryption, integrity and authentication.

#### 2.3.1.1 AES

Many modes, CBC is mainly used, GCM is great.

#### 2.3.1.2 SHA

Keyed signature algorithm, several versions in place. SHA-1 is depreciated, SHA-2 is widely used and SHA-3 is already defined and begins to be implemented.

### 2.3.2 Asymetric cryptography

Asymetric cryptography relies on a pair of keys: one private known only to the owner of the certificate, and one public available to anyone. Such cryptography uses two kinds of operations: encryption using the public key of the recipient and digital signature, which is an encryption using the private key of the sender.

#### 2.3.2.1 RSA

RSA is a public-key scheme proposed in 1978 by three MIT researchers who gave it their name [11]. 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 [13] and is defined as a precise key generation protocol allowing encryption and decryption. The keys can be generated by respecting a few steps:

1. randomly 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.  $e \cdot d \equiv 1 \pmod{\phi(n)}$

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**  $c = m^e \bmod n$

**Decryption**  $m = c^d \bmod n$

### 2.3.2.2 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 two type ([10], [7]):

- Static: the actors use their authenticated certificate to compute the shared secret.
- Ephemeral: 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. This mode is not advisable since it's vulnerable to man-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 transferred during past connections. However, if the secret is new for every new connection, the compromise of the shared secret does not jeopardize past communications. This is perfect forward secrecy: using a new key to protect the past.

Hereunder is the generation 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):  
 $\mathbf{p}$  large prime number  
 $\mathbf{g}$  a generator of  $\mathbb{Z}_p^*$
2. Alice picks a random integer  $x_a$  and computes  $g^{x_a} \bmod p = y_a$ .
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  $y_a^{x_b} \bmod p = g^{x_a x_b} \bmod p = ZZ$ , the shared secret to use as a premaster key from which will be derived the symmetric key for further communications.
6. Bob sends  $y_b = g^{x_b} \bmod p$ , signing everything with his private certificate.
7. Alice checks the signature and computes the same shared secret:  $ZZ = y_b^{x_a} \bmod p = g^{x_a x_b} \bmod p$

If the server is Alice, it has to do at least one signature, one signature verification and two modular exponentiations. Note that the client, B in our case, could have one signature less because RFC 5246 [16] 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 prime numbers. We will see in chapter 6 that while a 1024-bit prime is easily manageable by full software implementation, hardware offloading become a necessity for 4096-bit primes. Moreover, 1024-bit parameter size, both RSA and Diffie-Hellman, are disallowed by the NIST recommendations since 2013 [3].

## 2.4 Network and VPN implementation

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

### 2.4.1 SSL/TLS

### 2.4.2 IPsec

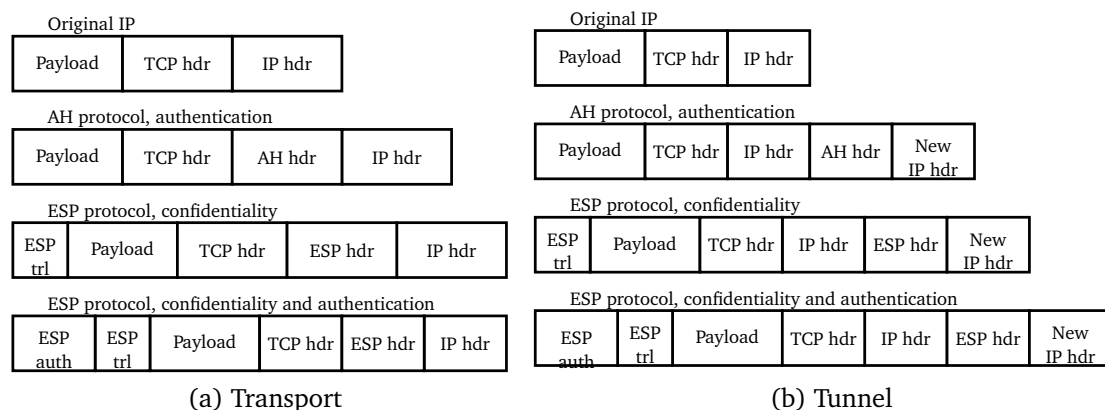


Figure 2.1 – IPsec transport (a) and tunnel (b) overheads: Tunnel mode adds a custom IP header and moves the AH/ESP header in front of the original IP header. Note: “trl” stands for “trailer”, “hdr” for “header”

# Chapter 3

## Presentation

Here we talk about the protocols, the platform. Show The OS stack (kernel/user)

### 3.1 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 together 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.

#### 3.1.1 x86 host

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

**OS** Ubuntu 12.04 LTS, kernel 3.16  
**CPU** Intel Core-i3 ... (two logical core out of four)  
**RAM** 1GB DDR3

#### 3.1.2 Altera Socrates SoCFPGA

**OS** Yocto project, kernel 3.14  
**CPU** Dual core ARM Cortex-A9, 800MHz  
**RAM** ...GB DDR3

**FPGA** Altera Cyclone V

### **3.1.3 ARM DS-5 Streamline**

### **3.1.4 Barco Silex' IPs**

# Chapter 4

## Implementation

This chapter will present how we implemented the protocols of chapter 2 on the board. The figure 4.1 show 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 application, 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 IPs to which the cryptographic operations will be offloaded and the associated drivers.

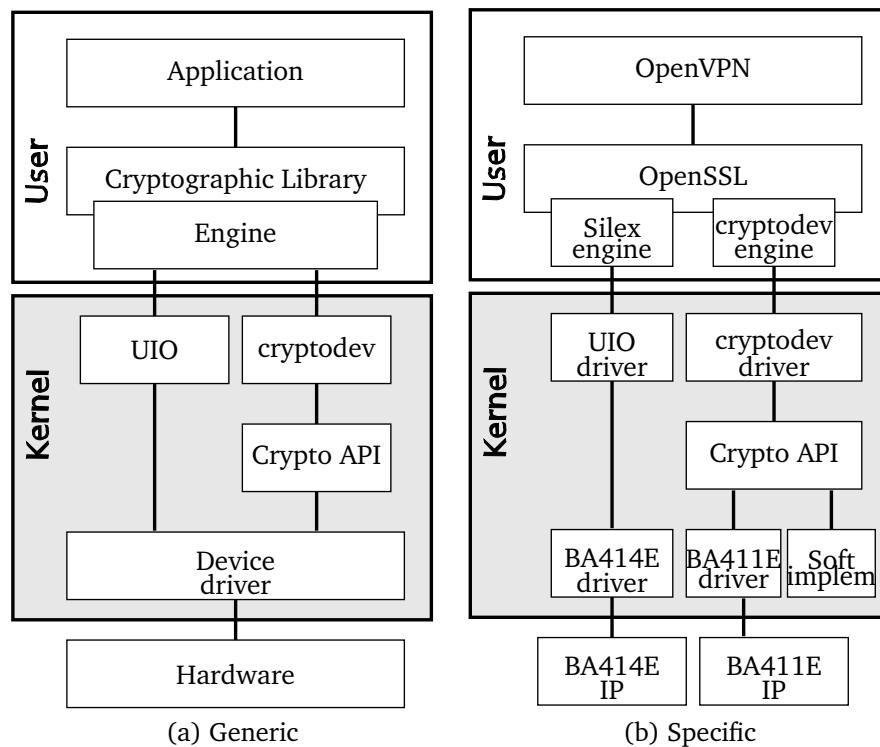


Figure 4.1 – OS data paths: (a) as a generic abstraction and (b) with some specific blocks replaced with custom implementations.

## 4.1 Software

### 4.1.1 OpenVPN

---

```

1  /* Compress, fragment, encrypt and HMAC-sign an outgoing packet. */
2  void encrypt_sign (struct context *c, bool comp_frag)
3  {
4      struct context_buffers *b = c->c2.buffers;
5      const uint8_t *orig_buf = c->c2.buf.data;
6
7      if (comp_frag){
8          /* Compress the packet. */
9          if (lzo_defined (&c->c2.lzo_compwork))
10             lzo_compress (&c->c2.buf, b->lzo_compress_buf, &c->c2.lzo_compwork, &c->c2
               .frame);
11         /* Fragment the packet. */
12         if (c->c2.fragment)
13             fragment_outgoing (c->c2.fragment, &c->c2.buf, &c->c2.frame_fragment);
14     }
15
16     /* Encrypt the packet and write an optional HMAC signature. */
17     openvpn_encrypt (&c->c2.buf, b->encrypt_buf, &c->c2.crypto_options, &c->c2.
               frame);
18 }

```

---

Listing 4.1 – openvpn compress then encrypt – sample from forward.c

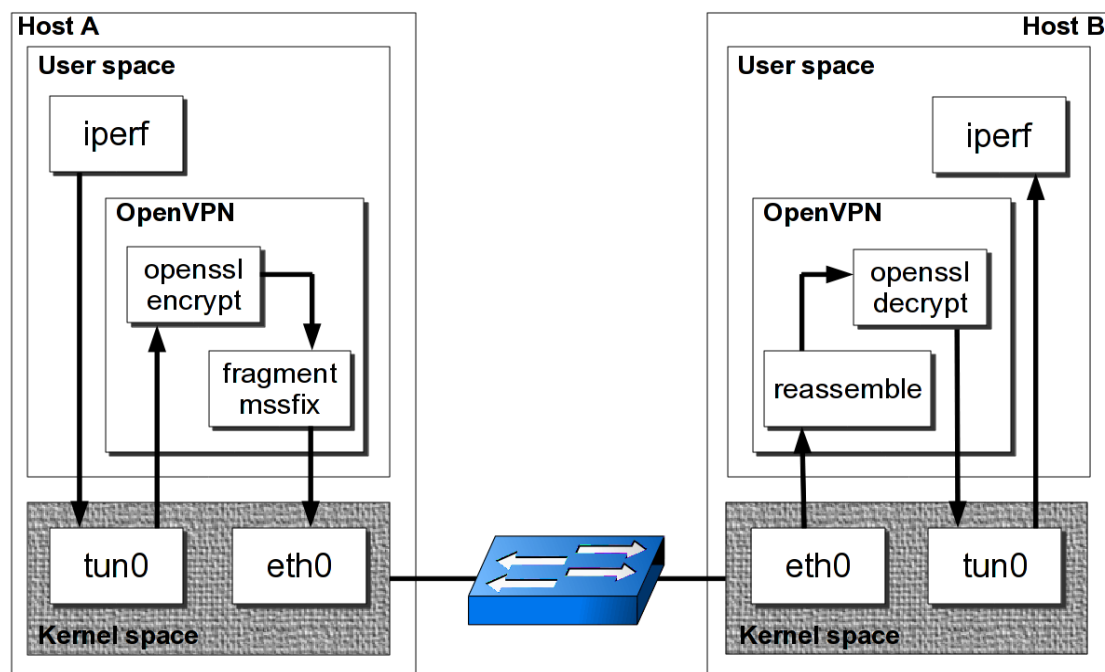


Figure 4.2 – OpenVPN packet flow as advertized on the openVPN wiki.:



### 4.1.2 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 liberty is a laste<sup>1</sup>. Even if it is not recommended, the user can still enable this mode by explicitly configuring it in the `sshd_config` options.

The second liberty has been taken to prevent the user to strip himself from data authenticity and integrity. However, in the case of testing and benchmarking, leaving the MAC behind can be interesting, especially if it is not offloaded in hardware such as the encryption in our case. In order to add this feature to OpenSSH, we wrote a patch to apply on OpenSSH 6.7, available in appendix [? ].

### 4.1.3 OpenSSL

It is to be noted that the present work uses an implementation with all the debug flags activated. Figure 4.3 shows that if it does have an impact on the performance, it is minimal: 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.

### 4.1.4 Strongswan

Strongswan is a full implementation of IPSec relying on the kernel drivers for the networking part, on the crypto API for the cryptographic part, and on user space crypto libraries for the connection negotiation. 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 painful. Strongswan as two advantages: it has a tremendous and exhaustive documentation, and its uer 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 4.4 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

---

<sup>1</sup>A vulnerability note as been issued by Carnegie Mellon University Computer Emergency Response Team in early 2009 (last revision) [4], in response to a research of University of London [1] presenting a plaintext-recovering attack against SSH when CBC mode is used, but the OpenSSH update took only place in october 2014.

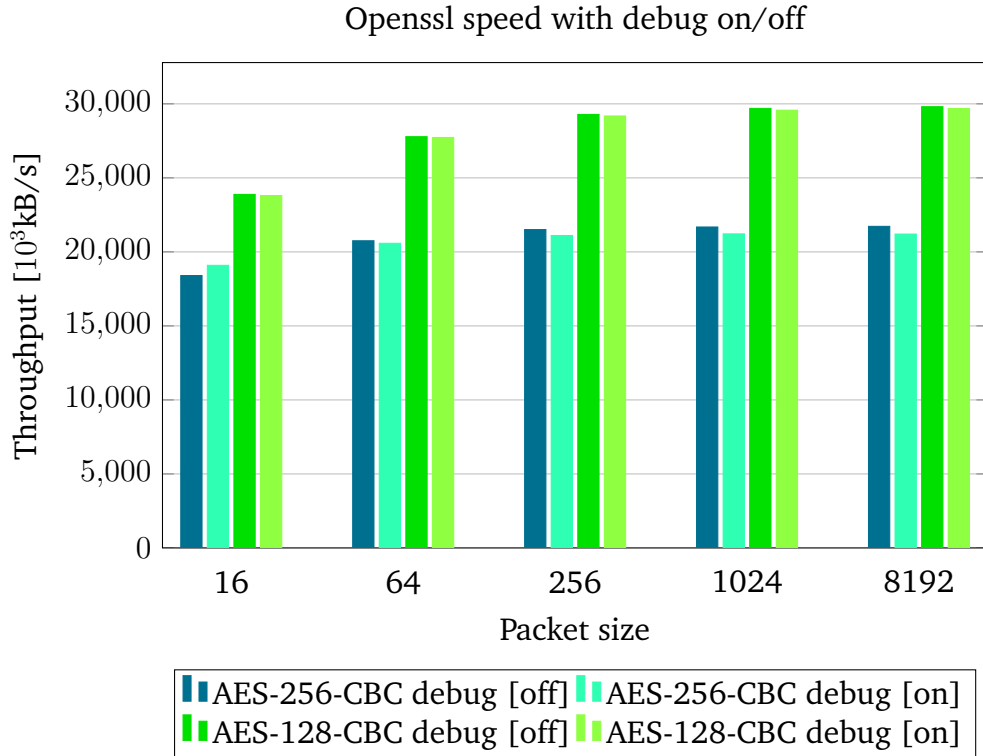


Figure 4.3 – **OpenSSL debugging benchmark:** Software benchmark of Openssl speed for AES mode CBC, with 128- and 256-bit keys, debugging flags (de)activated at compilation (`-fno-inline -g -marm`). The throughput difference ranges from 0.2% and 2.4% , and is more marked for larger keysize, as more debugging data needs to be generated.

VPNs [12]. Its name comes from the fact that the kernel transforms packet frames to incorporate IPsec security. Depending on the configuration, XFRM uses 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 incoming packets.

#### 4.1.5 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.

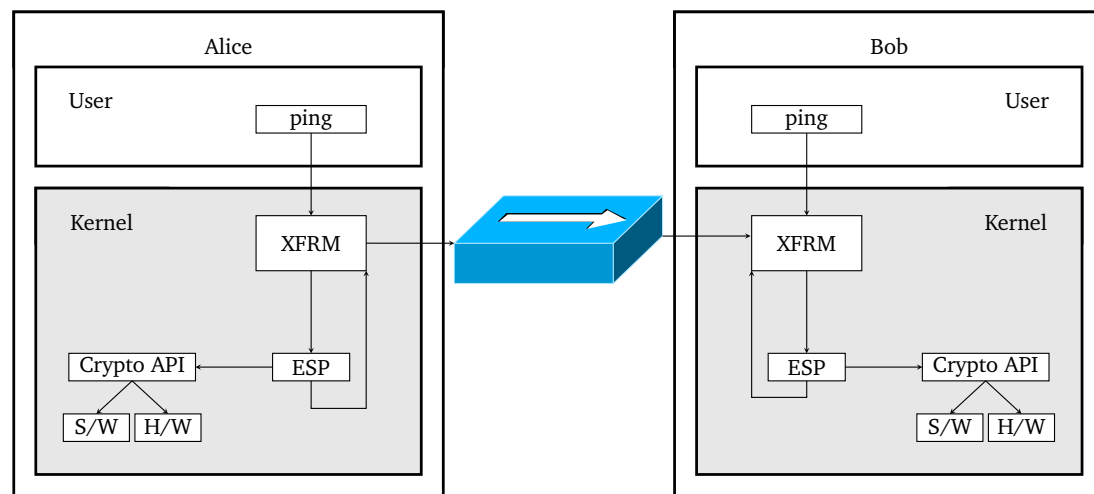


Figure 4.4 – IPsec user/kernel space workflow, using ping as a test case.:

`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 counterpart 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.

## 4.2 Offload

The table 4.1 summarizes the ciphers supported by the two Barco Silex' IPs used in this work.

IP	Ciphers
BA414E	RSA, DH, DSA, EC
BA411E	AES modes CBC, CTR, GCM, CCM, CFB, OFB, CTS, ECB

Table 4.1 – Summary of the ciphers supported by two of Barco Silex' IPs.:

### 4.2.1 BA411E Driver

Takes the rightern path, which is the cleanest because the most versatile. By plugging the driver into the 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 application 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 that, when called on other kernel drivers, put the kernel in panic mode and require the system to reboot. An alternative is under development and is further discussed in the last part of this work, in section 7.1.

Be it IRQ or polling, the inner workings are the same and follow the exact same canvas as the default software implementation the different AES modes.

### 4.2.2 BA414E Driver and Silex engine

At the moment of development, there is no asymetric cryptography interface in the crypto API<sup>2</sup>. 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 4.1b. As presented in section 2.1.1, we will need to rely on an UIO driver to access the device from the user space.

---

<sup>2</sup>A request for comment patch as been submitted to the Linux kernel mailing list [15] in late April 2015, proposing a standard interface for public key encryption in the crypto API.

# Chapter 5

## Test protocol and Prediction

### 5.1 File transfer

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

---

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

---

# Chapter 6

## Results and Analysis

### 6.1 TLS connections

This benchmark is done only with OpenVPN. 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 crypto API, this use case should however be tested immediatly.

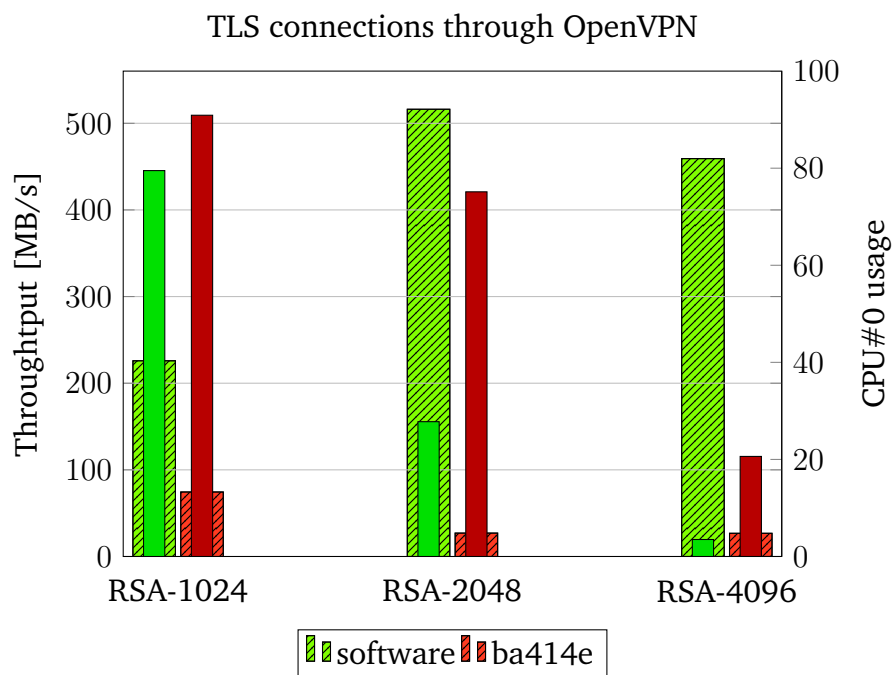


Figure 6.1 – **TLS connections per second**: The background stripped bars are the CPU usage. Raw data in table A.1.

### 6.2 Response time – latency

After the establishment of a connection.

### 6.2.1 OpenVPN

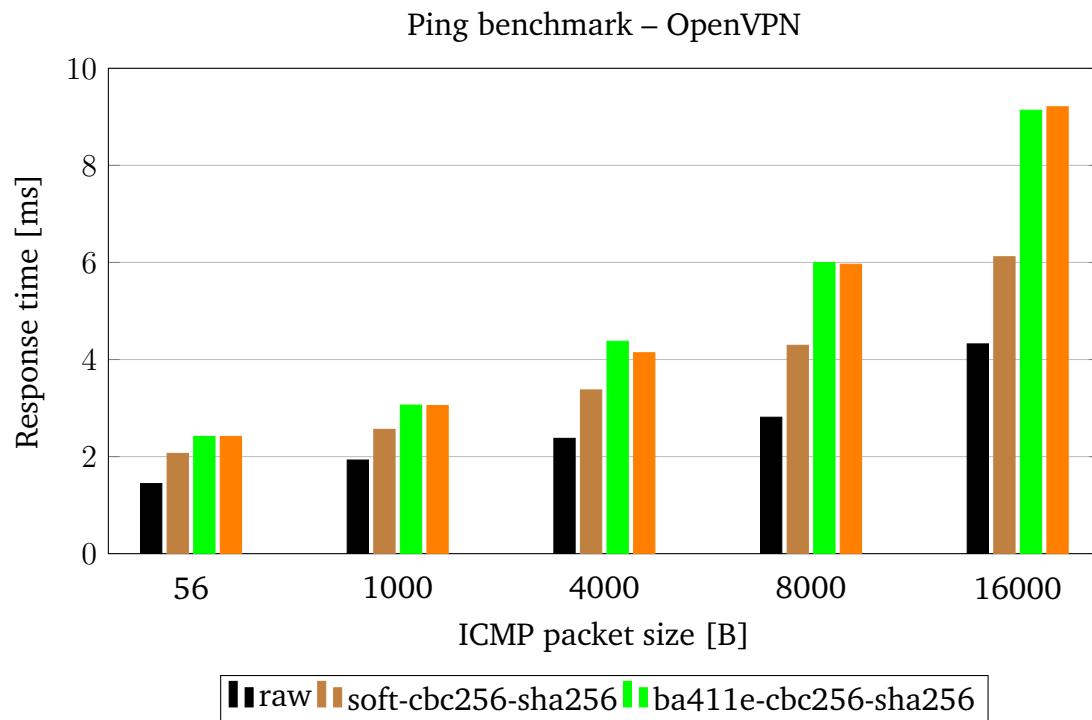


Figure 6.2 – Ping min/avg/max response time: for different packet sizes using OpenVPN

### 6.2.2 IPsec

### 6.2.3 Comparison

## 6.3 File transfer

### 6.3.1 openSSH

### 6.3.2 OpenVPN

We can see that adding a MAC computation aside the encryption merely lowers the performance when using the hardware. Even though OpenSSL uses here an highly ARM-optimized assembly implementation of SHA-256, it shows that the bottleneck is on the hardware side.

Indeed,

### 6.3.3 IPsec

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

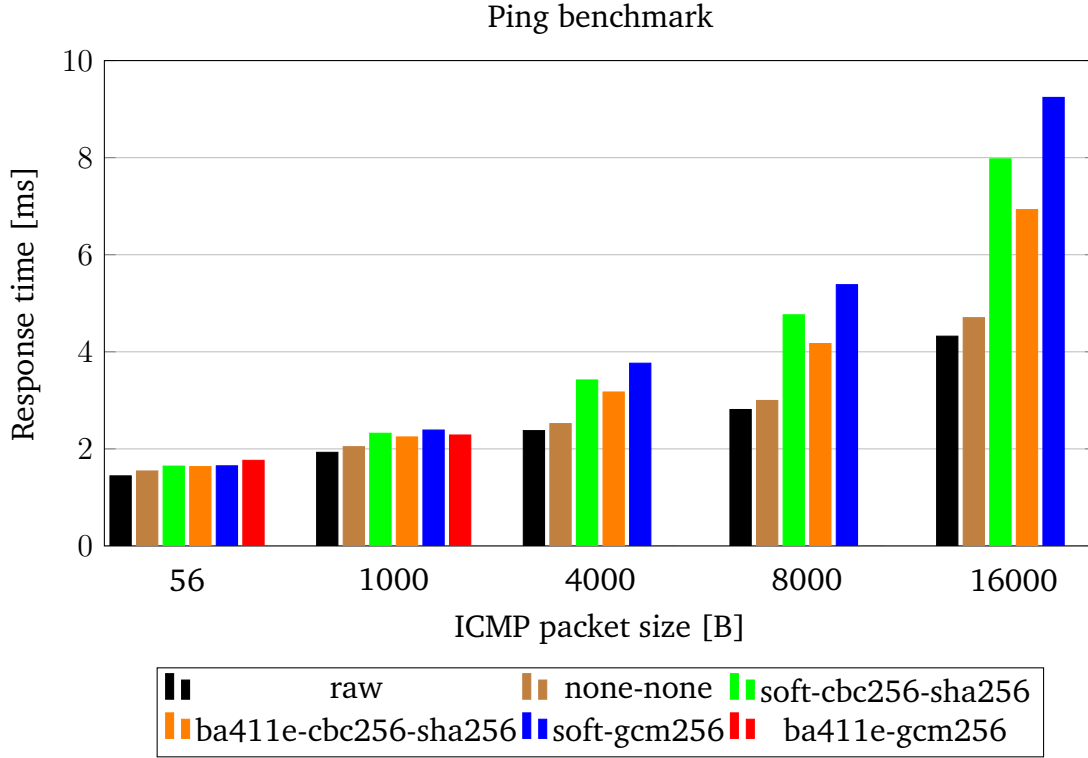


Figure 6.3 – **Ping min/avg/max response time:** for different packet sizes using IPsec. For each packet size, 1000 requests were flooded to the board, that is "outputs packets as fast as they come back or one hundred times per second, whichever is more", according to the ping command manual.

be used and would most probably yield better results. The CPU usage of the software implementation would drop – even if not significantly – as for the hardware, it would be less limited by the software MAC counter part, and if the CPU usage could stay at the same level, we could expect a better throughput.

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. Note that they 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 set 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 [8, 6], but none have been committed to the Linux kernel repository yet.

### 6.3.4 Comparison



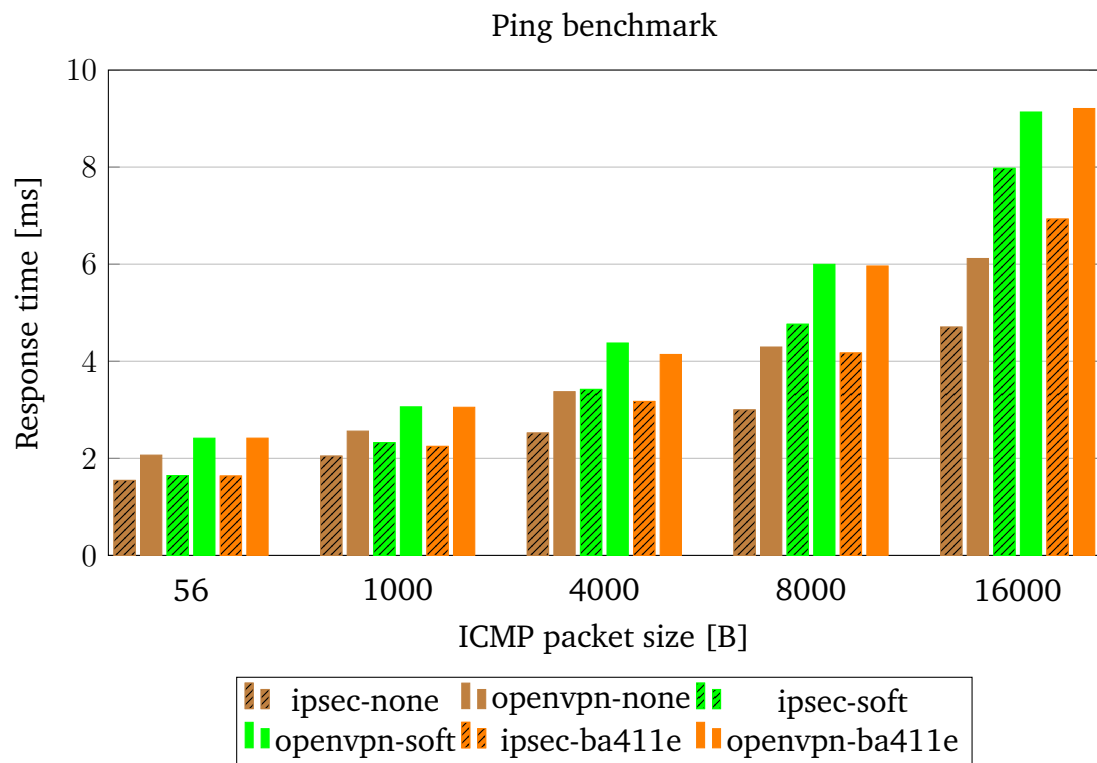


Figure 6.4 – Ping average comparison: All values are for aes-256-cbc with sha-256

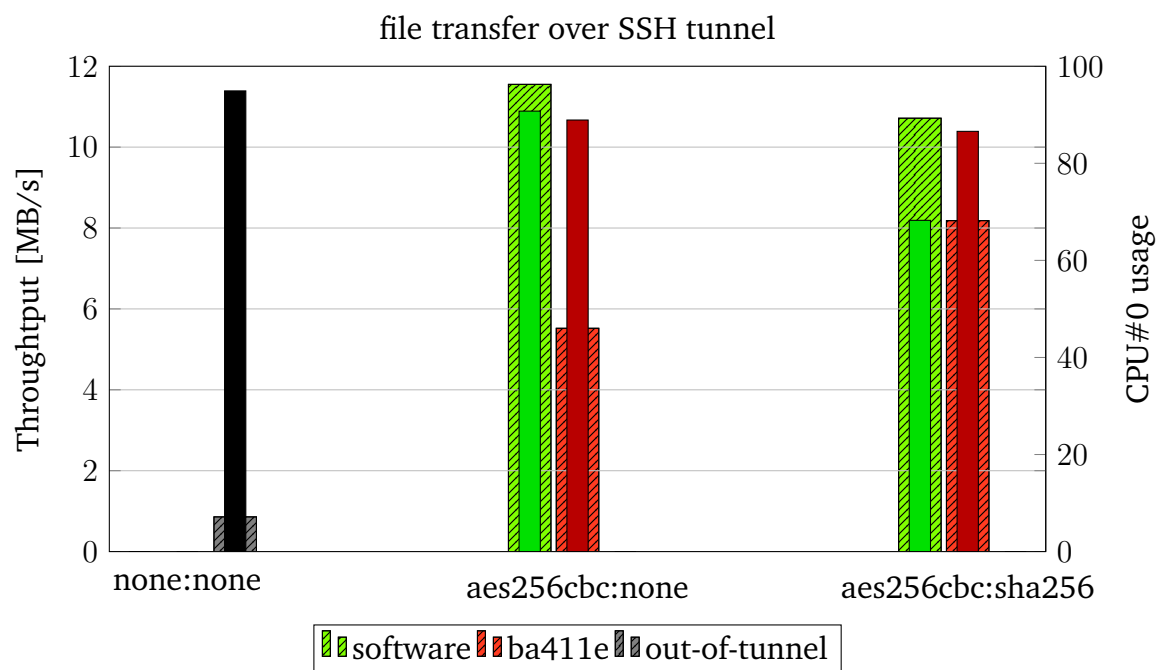


Figure 6.5 – file transfer over an SSH tunnel. The background striped bars are the CPU usage.:

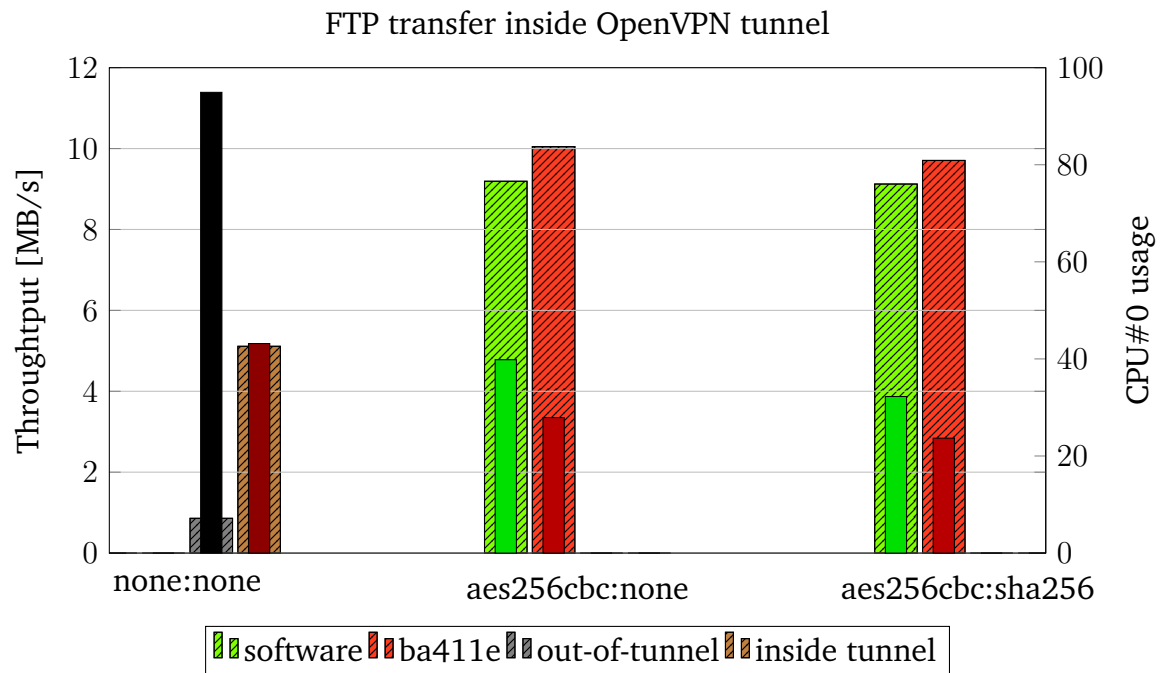


Figure 6.6 – FTP file transfer over an OpenVPN tunnel. The background stripped bars are the CPU usage.:

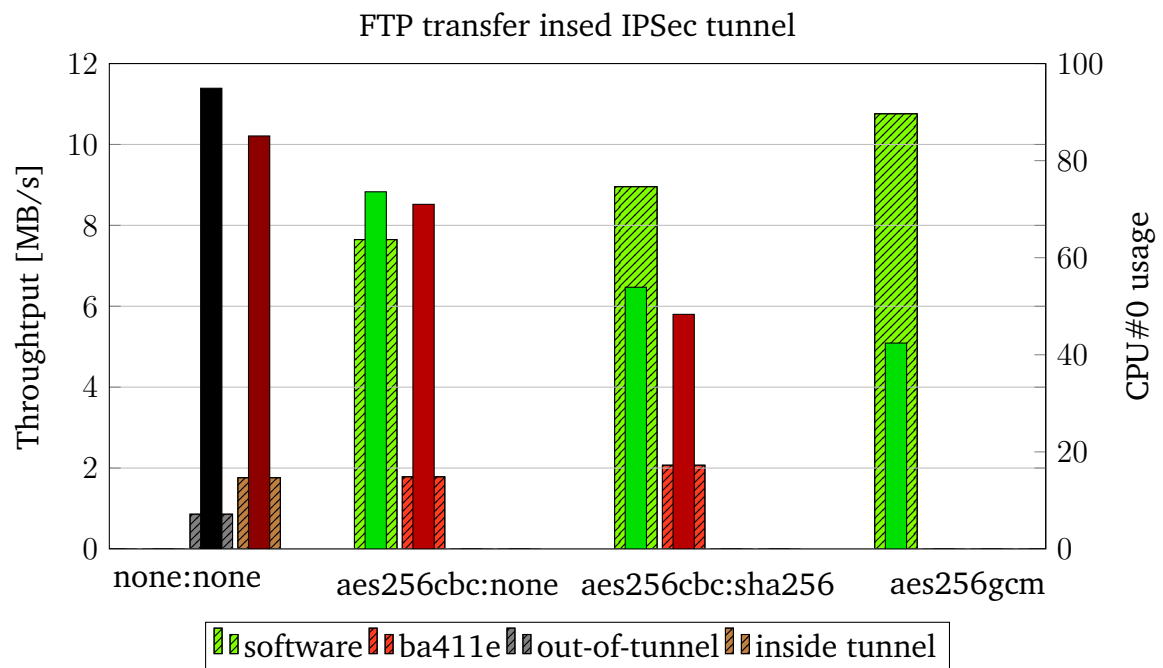


Figure 6.7 – FTP file transfer over an IPsec tunnel. The background stripped bars are the CPU usage.:

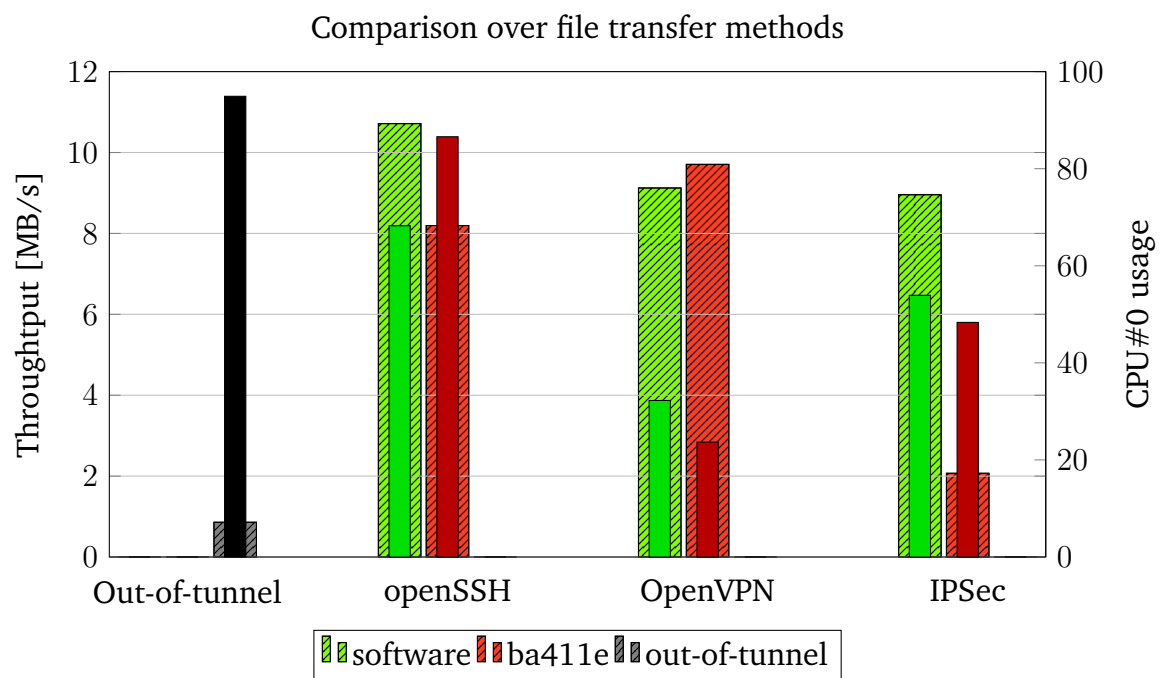


Figure 6.8 – Comparison of file transfer methods: The background stripped bars are the CPU usage.

# Chapter 7

## Discussion and conclusion

### 7.1 Future work

Although the present work presents some promising 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 resources 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 modifying 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 actively 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 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 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 development. 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 [15] and is still under request for comments.

**Conditional offloading in cryptodev** The figure ?? 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 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 interesting 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<sup>1</sup> 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 [2]. 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 symmetric and asymmetric encryption ciphers 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 some tricks and patches allowed us to bypass this requirement. The implementation of the GCM mode, combining encryption and authentication, showed us

---

<sup>1</sup>Jumbo frames have an ethernet MTU of 9000 bytes, whilst standard frames are set to 1500.

that stopping relying on the software implementation of MACs would be a huge step forward.

# Appendix A

## Raw data

### A.1 TLS connections

	RSA-1024		RSA-2048		RSA-4096	
	Con.	CPU	Con.	CPU	Con.	CPU
Soft	445.4	40.32	155.6	92.14	19.6	81.97
BA414E	509.3	13.29	420.9	4.82	115.5	4.34

Table A.1 – **TLS connections per minute**: measures obtained with ten clients concurrently connecting to an OpenVPN server.

# Appendix B

## OpenSSH patch

---

```
1 diff -rupN openssh-6.7p1_vanilla/digest.h openssh-6.7p1/digest.h
2 --- openssh-6.7p1_vanilla/digest.h 2014-07-03 13:25:04.000000000 +0200
3 +++ openssh-6.7p1/digest.h 2015-02-27 11:59:53.508495697 +0100
4 @@ -28,7 +28,8 @@
5 #define SSH_DIGEST_SHA256 3
6 #define SSH_DIGEST_SHA384 4
7 #define SSH_DIGEST_SHA512 5
8 -#define SSH_DIGEST_MAX 6
9 +#define SSH_DIGEST_NONE 6
10 +#define SSH_DIGEST_MAX 7
11
12 struct sshbuf;
13 struct ssh_digest_ctx;
14 diff -rupN openssh-6.7p1_vanilla/digest-libc.c openssh-6.7p1/digest-libc.c
15 --- openssh-6.7p1_vanilla/digest-libc.c 2014-07-02 07:28:03.000000000 +0200
16 +++ openssh-6.7p1/digest-libc.c 2015-02-27 12:01:48.420494935 +0100
17 @@ -113,6 +113,16 @@ const struct ssh_digest digests[SSH_DIGE
18     (md_init_fn *) SHA512Init,
19     (md_update_fn *) SHA512Update,
20     (md_final_fn *) SHA512Final
21 + },
22 + {
23 +     SSH_DIGEST_NONE,
24 +     "none@barco.com",
25 +     0,
26 +     0,
27 +     0,
28 +     NULL,
29 +     NULL,
30 +     NULL
31     }
32 };
33
34 diff -rupN openssh-6.7p1_vanilla/digest-openssl.c openssh-6.7p1/digest-openssl.
35 c
36 --- openssh-6.7p1_vanilla/digest-openssl.c 2014-07-17 01:01:26.000000000 +0200
37 +++ openssh-6.7p1/digest-openssl.c 2015-02-27 12:00:24.812495489 +0100
38 @@ -59,6 +59,7 @@ const struct ssh_digest digests[] = {
39     { SSH_DIGEST_SHA256, "SHA256", 32, EVP_sha256 },
40     { 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 C

## Toolchain

Talk here about linux linaro and shit. This does not need to be long, just explain which version was used, what environment variable to export.

See if this does not enter in conflict with the "cross-compilation" appendix.

# Appendix D

## Cross-compilation

**D.1**   **OpenSSL**

**D.2**   **OpenVPN**

**D.3**   **nginx**

**D.4**   **pure-ftpd**

**D.5**   **Strongswan**

**D.6**   **kernel**

Force kernel version

Change list of modules (show the final version)

# **Appendix E**

## **Stuff**

# Bibliography

- [1] 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.
- [2] 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>.
- [3] 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.
- [4] CERT. Vulnerability note vu#958563: Ssh cbc vulnerability. <http://www.kb.cert.org/vuls/id/958563>, 2009. Carnegie Mellon University.
- [5] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. ISBN 0596005903.
- [6] 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.
- [7] 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.
- [8] Conrado P. L. Gouvêa and Julio López. Implementing gcm on armv8. In *Lecture Notes in Computer Science*. Springer International Publishing, 2015.
- [9] Hans J. Koch. Userspace I/O drivers in a realtime context. Linutronix GmbH, 2011.
- [10] E. Rescorla. Diffie-Hellman Key Agreement Method. RFC 2631, RFC Editor, June 1999. URL <http://tools.ietf.org/rfc/rfc2631.txt>.
- [11] 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.

- [12] Rami Rosen. *Linux Kernel Networking*. Apress, 2014. ISBN 978-1-4302-6196-4.
- [13] RSA Laboratories. PKCS #1 v2.2: RSA Cryptography Standard. Technical report, October 2012.
- [14] 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](http://www2.comp.ufscar.br/~helio/kernel_2.6/inside_kernel-2.6.html), September 2003. Originaly published on IMB website.
- [15] 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>.
- [16] J. Viega and D. McGrew. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 45246, RFC Editor, August 2008. URL <https://tools.ietf.org/rfc/rfc5246.txt>.