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 protocol using a SoC platform

This is an abstract.

Résumé

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

Contents

List of Figures

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 [**?**].

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 [16]

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 [8, 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 [26], 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.

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 ther 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 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.

2.3 Cryptography

Cryptography is the corner stone of security. The four main goals are the following, as defined in [21]:

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: symmetric and asymmetric ciphers, message digests and digital signatures.

2.3.1 Message digest

A message digest is the result of a one-way mathematical function of a fixed size. Those hash functions are of two types [18]: 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:

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

Based on this design and adding a key to the process, the RFC 2104 [17] defines a MAC:

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

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.

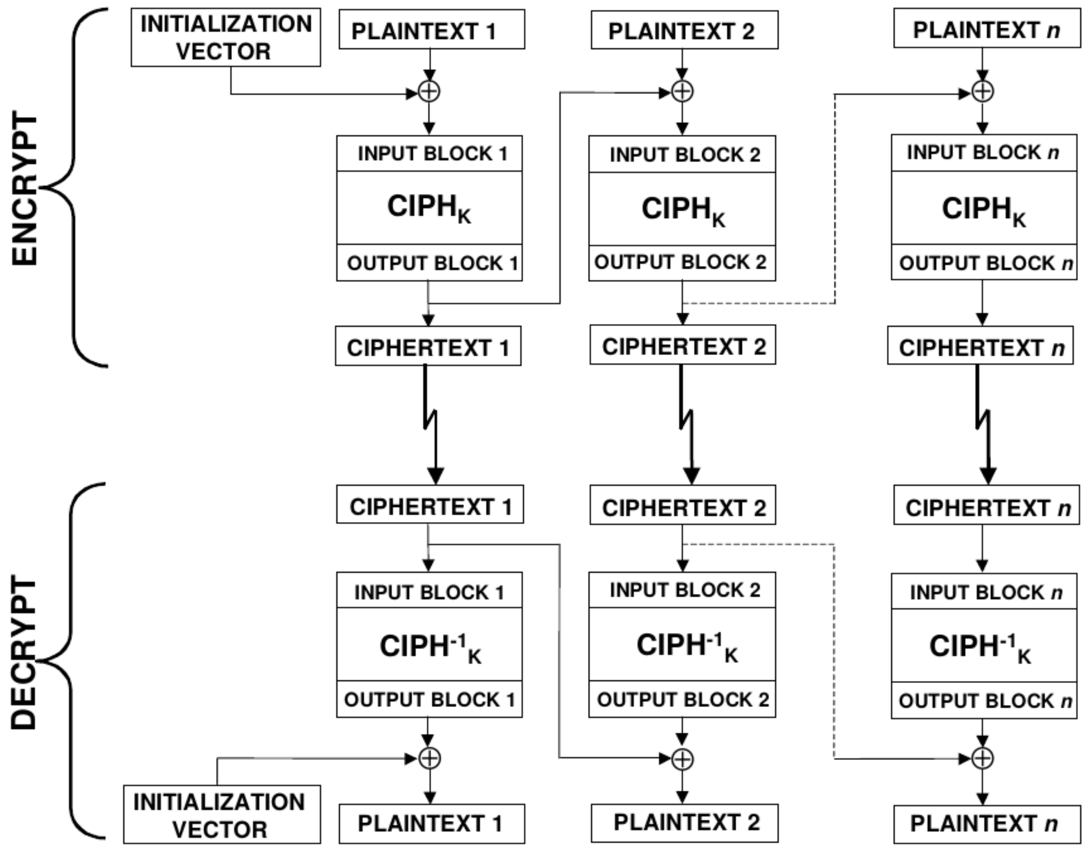


Figure 2.1 – **CBC encryption and decryption diagram:** taken from the NIST recommendation [10].

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.

Now rises the question of what and when authenticating. Bellare and Namprempre [4] 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 Symetric cryptography

Talk about encryption, integrity and authentication.

Look in depth into AES, NIST approved five modes [10]: CBC (Cipher Block Chaining), ECB (Electronic CodeBook), CFB (Cipher FeedBack), OFB (Output FeedBack) and CTR (Counter).

AEAD => Encryption and authentication: GCM.

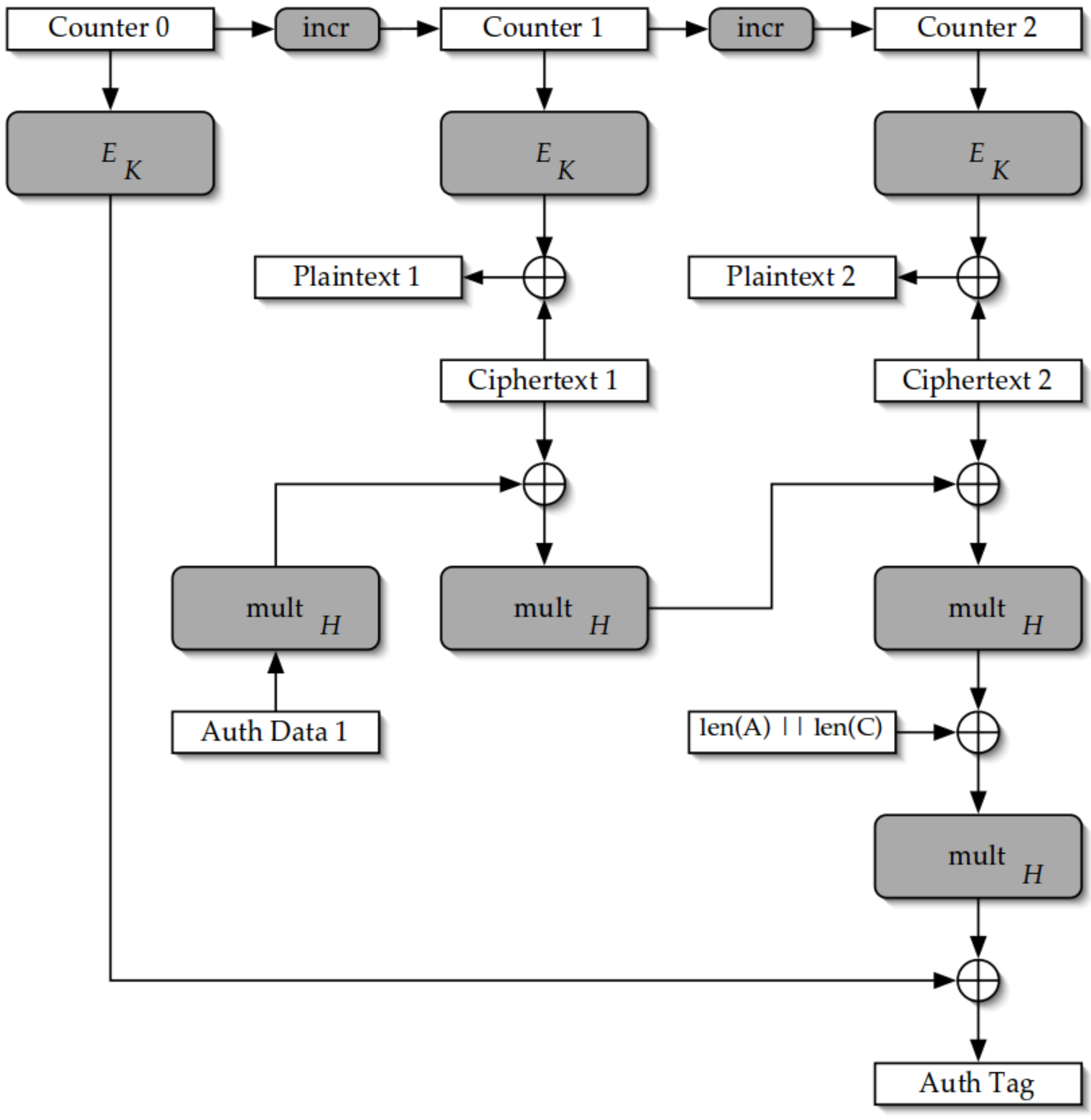


Figure 2.2 – **GCM encryption diagram:** taken from the NIST specification [20]. The ciphertext blocks are formed by *xor*-ing the encrypted counter and the plaintext. The tag is generated by a chain of ciphertext *xor*-ing with Galois field multiplicated data. The decryption works excatly the same way, except the plaintext and ciphertext are swapped.

2.3.3 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 ecnryption using the private key of the sender.

2.3.3.1 RSA

RSA is a public-key scheme proposed in 1978 by three MIT researchers who gave it their name [23]. 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 [25] 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 $m = c^d \ mod \ n$

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 symetric 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 ([22], [12]):

- 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 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 des 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):
   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 symetric 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, one signature verification and two modular exponentiations. Note that the client, B in our case, could have one signature less because RFC 5246 [29] 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 5 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 recommandations since 2013 [3].

2.4 Network and VPN implementation

The RFC 1122 [5] defines the TCP/IP and 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 [28], conceptually split the TCP/IP link layer into an additional physical layer.

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

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
Data 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.

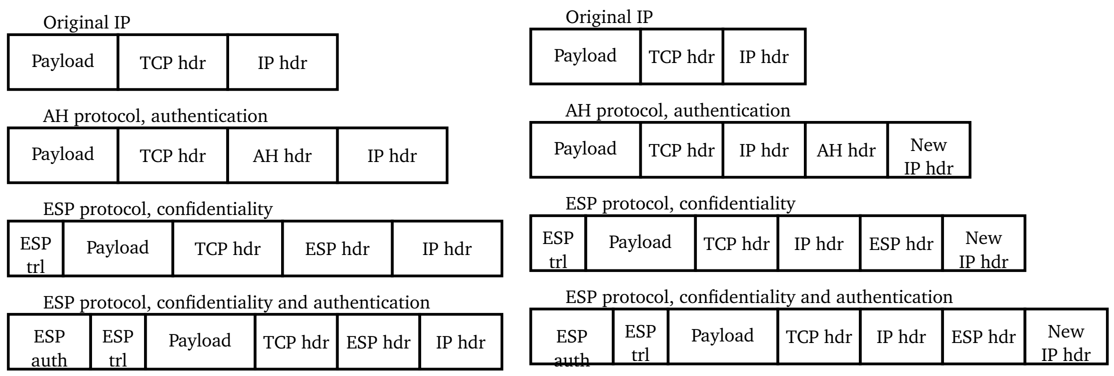
2.4.1 SSL/TLS

Application level security.

2.4.2 IPsec

Modification of the IP stack in the kernel space. IPsec is a network 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.

One of the main disadvantage compared to a user-space VPN is the difficulty to traverse NAT.



(a) Transport (b) Tunnel

Figure 2.3 – **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"

RFC 7321 [19] defines the support for only three AES modes: CBC, CTR and GCM.

```
 0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ ----
|               Security Parameters Index (SPI)                 | ^Int.
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ |Cov-
|                      Sequence Number                          | |ered
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+---- | ----
|                    IV (optional)                              | ^ p | ^
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ | a | |
|                    Rest of Payload Data  (variable)           | | y |Conf.
~                                                               ~ | l |Cov-
|                                                               | | o |ered*
+               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ | a | |
|               |         TFC Padding * (optional, variable)    | v d | |
+-+-+-+-+-+-+-+-+         +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+---- | |
|                         |        Padding (0-255 bytes)        |     | |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+     | |
|                               |  Pad Length   | Next Header   |     v v
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+------------
|         Integrity Check Value-ICV   (variable)                |
~                                                               ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.4 – **ESP packet structure:** as defined in RFC 4303 [15].

Don't forget to talk about the IPsec overhead from [30].

Chapter 3

Implementation

This chapter will present how we implemented the protocols of chapter 2 on the board. The figure 3.1 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.

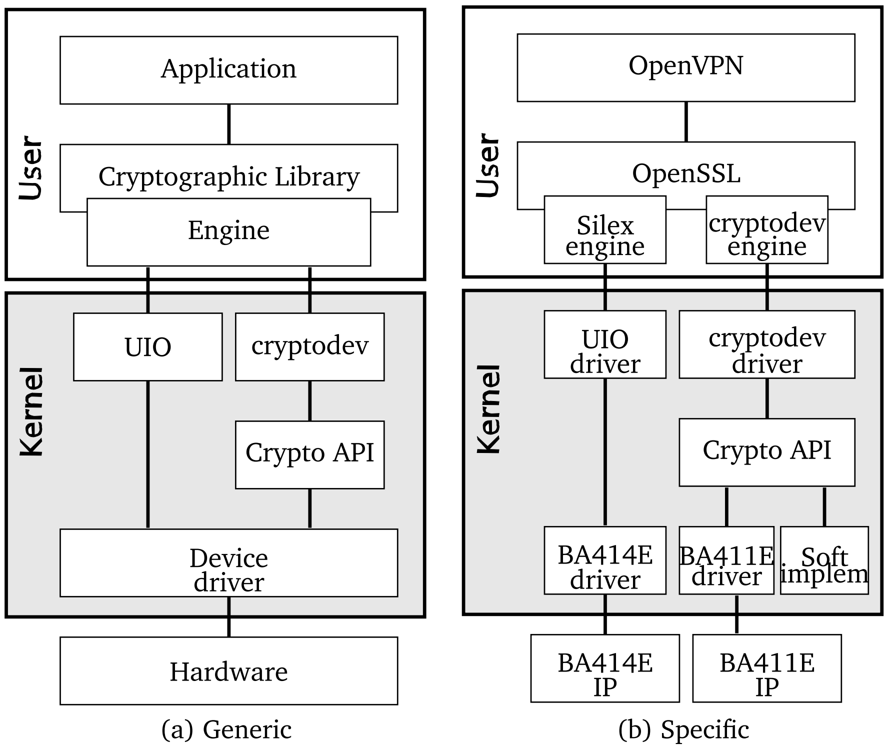


Figure 3.1 – **OS data paths:** (a) as a generic abstraction and (b) with some specific blocks replaced with custom implementations.

3.1 Software

3.1.1 OpenVPN

OpenVPN is also used by the Dutch government to secure all its communications [11].

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

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 which to push and from which to read the secured 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 are already 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.
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 part 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.

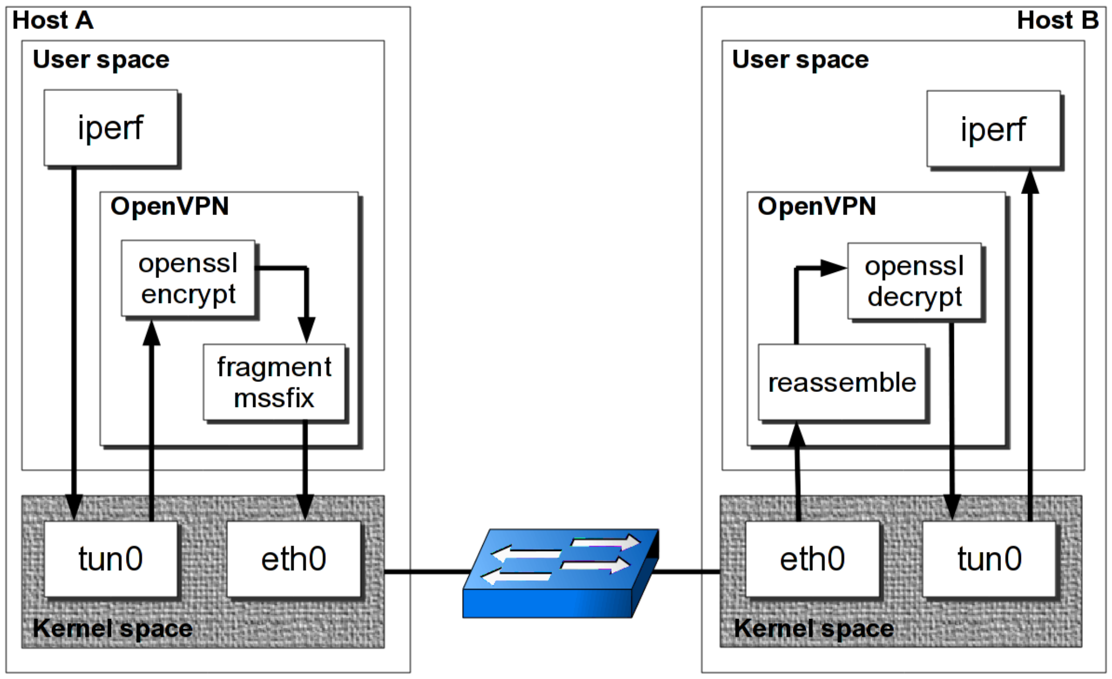


Figure 3.2 – **OpenVPN packet flow:** it shows two hosts using `iperf` inside an OpenVPN tunnel. This figure comes from the OpenVPN documentation [7] and has inverted the encryption and fragmentation.

OpenVPN relies on a cryptographic library, OpenSSL in our case, for all the security operations: encryption, signature, certificate management, etc.
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 [14].

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 liberty is a laste[1]. Even if it is not recommended, the user can still enable this mode by explicitly configuring it in the `sshd_config` options.

[1]A vulnerability note as been issued by Carnegie Mellon University Computer Emergency Response Team in early 2009 (last revision) [6], 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.

```c
void encrypt_sign (struct context *c, bool comp_frag)
{
  struct context_buffers *b = c->c2.buffers;
  const uint8_t *orig_buf = c->c2.buf.data;

  if (comp_frag){
  /* Compress the packet. */
    if (lzo_defined (&c->c2.lzo_compwork))
      lzo_compress (&c->c2.buf, b->lzo_compress_buf, &c->c2.lzo_compwork, &c->c2
          .frame);
  /* Fragment the packet. */
    if (c->c2.fragment)
      fragment_outgoing (c->c2.fragment, &c->c2.buf, &c->c2.frame_fragment);
  }

  /* Encrypt the packet and write an optional HMAC signature. */
  openvpn_encrypt (&c->c2.buf, b->encrypt_buf, &c->c2.crypto_options, &c->c2.
      frame);
}
```

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.

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 interresting, 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 A.

3.1.3 OpenSSL

Offers an high-level interface called EVP to be used by other applications.

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 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.

3.1.4 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 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 painfull. Strongswan as two advantages: it has a tremendous and exhaustive documentation, and its uer interface is straightforward. Once configured, a simple `ipsec`

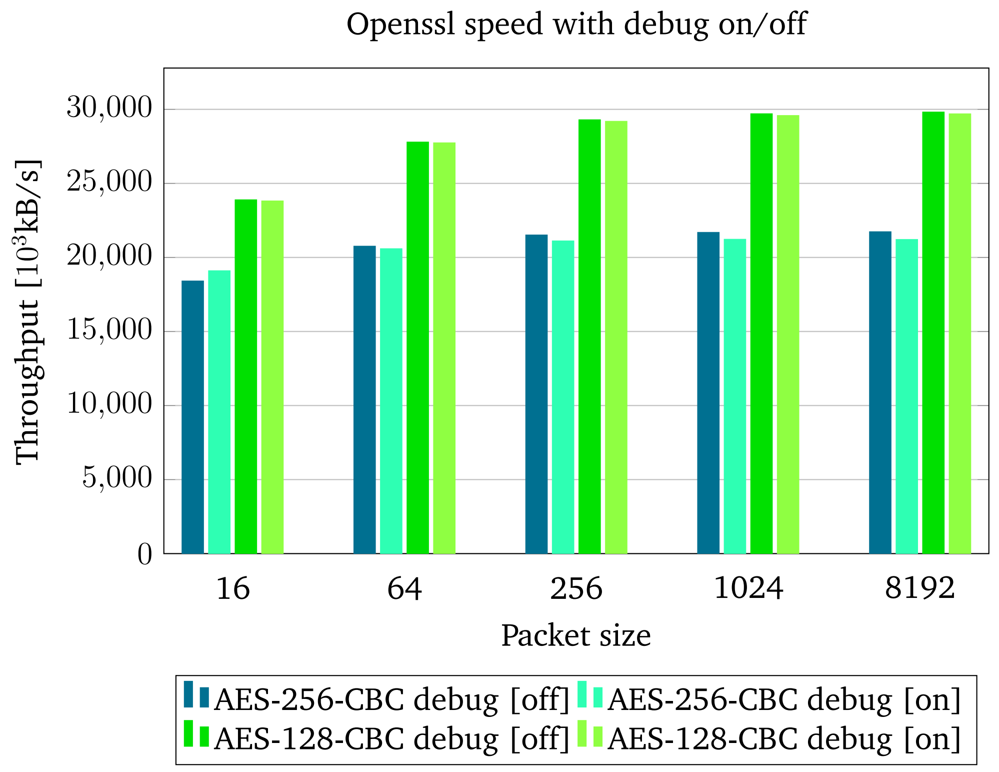


Figure 3.3 – **OpenSSL debugging benchmark:** Software benchmark of Openssl speed for AES mode CBC, with 128- and 256-bit keys, debugging flags (de)activated at compilation (`-fno-inline -g -marm`). The throughput difference ranges from 0.2% and 2.4% , and is more marked for larger keysize, as more debugging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

`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 VPNs [24]. 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 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

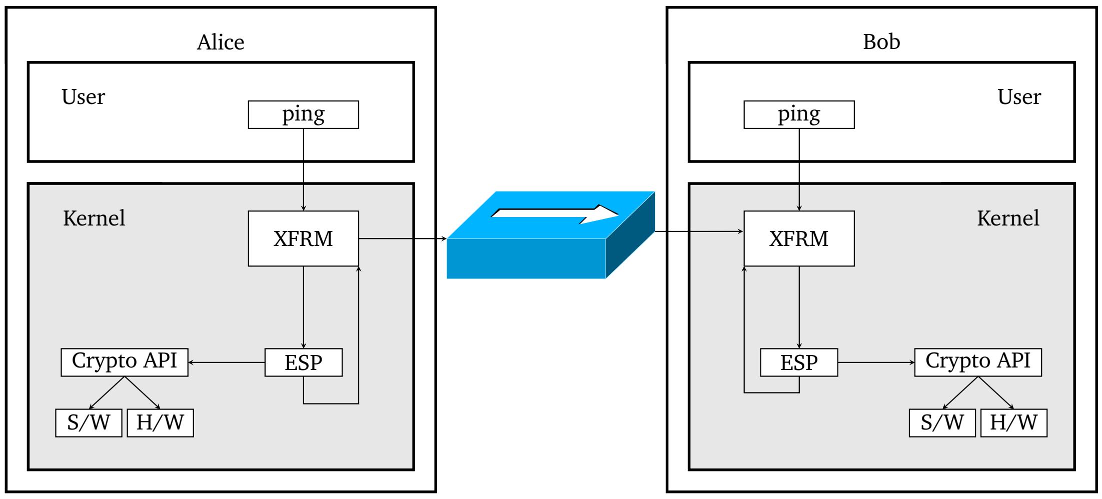


Figure 3.4 – **IPsec user/kernel space workflow, using `ping` as a test case.:**

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.
`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 3.1 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 3.1 – **Summary of the ciphers supported by two of Barco Silex' IPs.:**

3.2.1 BA411E Driver

Takes the rightern path, which is the cleanest because the most versatile. By pluging 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 6.1.

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.

3.2.2 BA414E Driver and Silex engine

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

[2]A request for comment patch as been submitted to the Linux kernel mailing list [27] in late April 2015, proposing a standard interface for public key encryption in the crypto API.

Chapter 4

Test protocol and Prediction

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 diretly 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 developpement and testings.

OS Ubuntu 12.04 LTS, kernel 3.16
CPU Intel Core-i5 (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

4.2 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.

For this use case, the board is configured in server mode, so that it can accept connections from any client. 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 **??**), 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.

```bash
#!/bin/bash

timeout=120
rsa=1024
remote_ip=150.158.232.208

for i in `seq 1 10`; do
  timeout ${timeout} openvpn --config client_${i}_${rsa}.ovpn --verb 2 --remote
      ${remote_ip} &
done
```

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

4.3 File transfer

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

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

For the IPsec use case, the cipher/authentication pair null/null will be used to quantify the overhead of the encapsulation. It should however not be forgotten that it is not to be used as a production configuration. As the RFC 7321 [19, pg. 7] sates: “Note that while authentication and encryption can each be ‘NULL’, they MUST NOT both be ‘NULL’”.

Chapter 5

Results and Analysis

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 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.

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

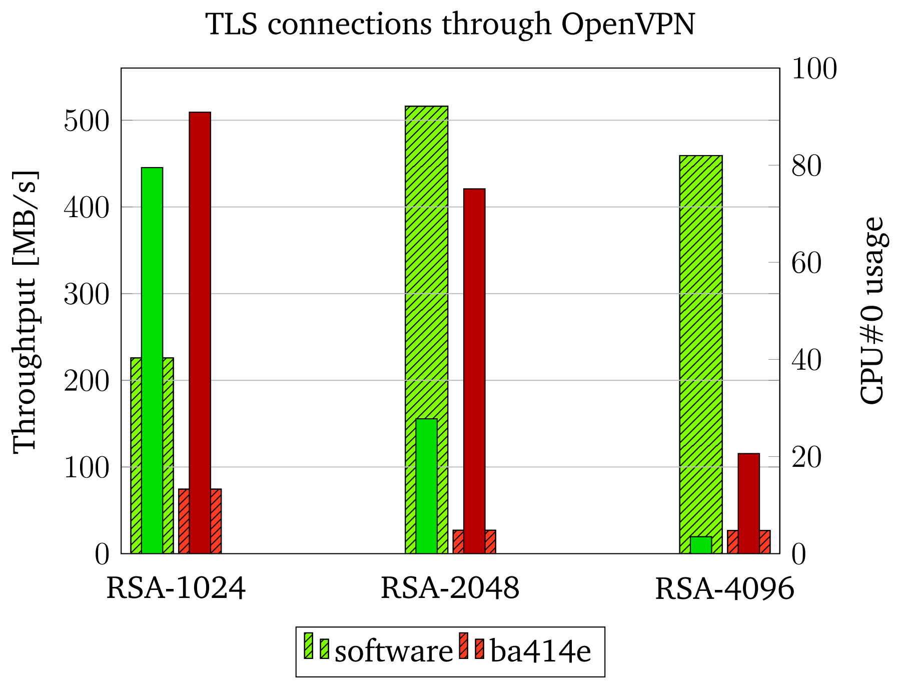


Figure 5.1 – **TLS connections per minute:** The background stripped bars are the CPU usage. Raw data in table 5.2.

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

After the establishement of a connection.

5.2.1 OpenVPN

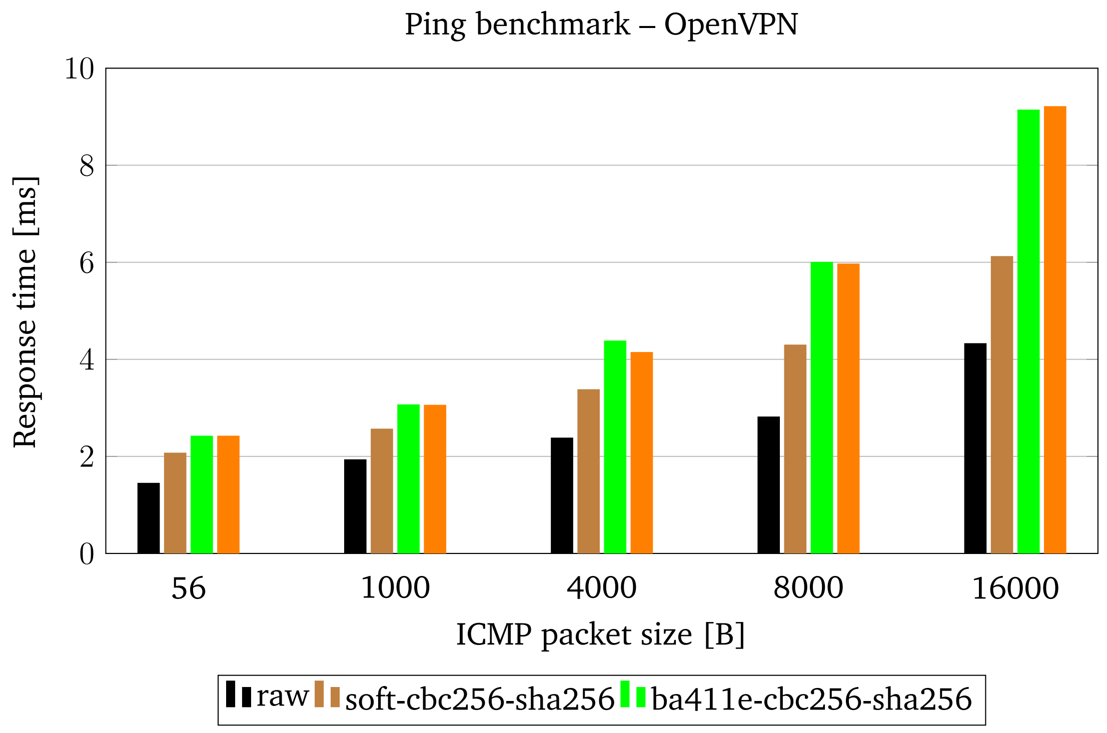


Figure 5.2 – **Ping min/avg/max response time:** for different packet sizes using OpenVPN

5.2.2 IPsec

5.2.3 Comparison

5.3 File transfer

5.3.1 openSSH

5.3.2 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 hihgly ARM-optimzed assembly implementation of SHA-256, it shows that the bottleneck is on the hardware side.

Indeed,

5.3.3 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

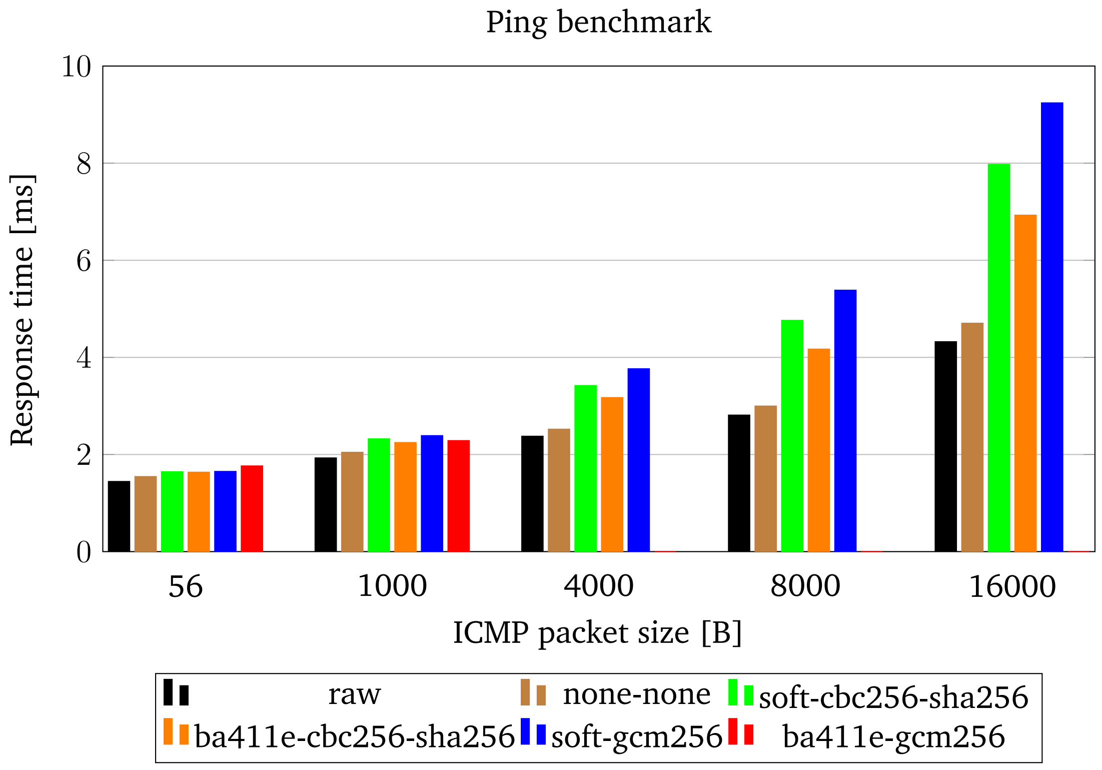


Figure 5.3 – **Ping min/avg/max response time:** for different packet sizes using IPsec. For each packet size, 1000 requests were flooded to the board, that is "*outputs packets as fast as they come back or one hundred times per second, whichever is more*", according to the `ping` command manual.

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 usinf 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 [13, 9], but none have been committed to the Linux kernel repository yet.

5.3.4 Comparison

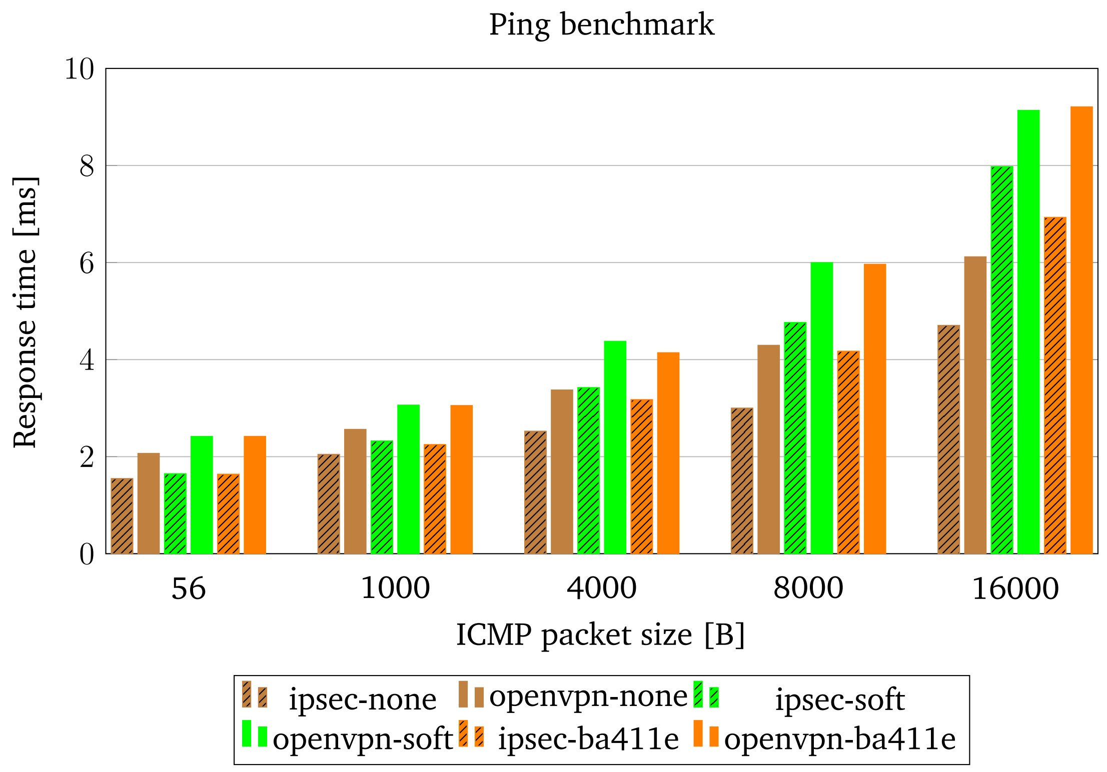


Figure 5.4 – **Ping average comparison:** All values are for aes-256-cbc with sha-256

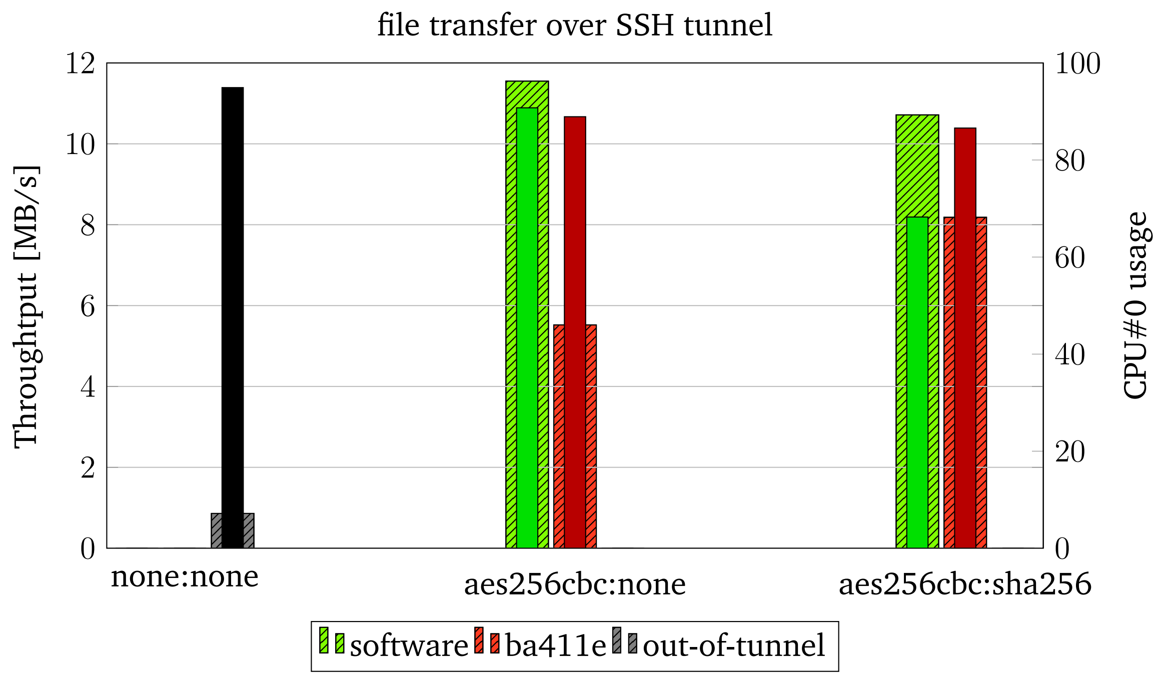


Figure 5.5 – **file transfer over an SSH tunnel. The background stripped bars are the CPU usage.:**

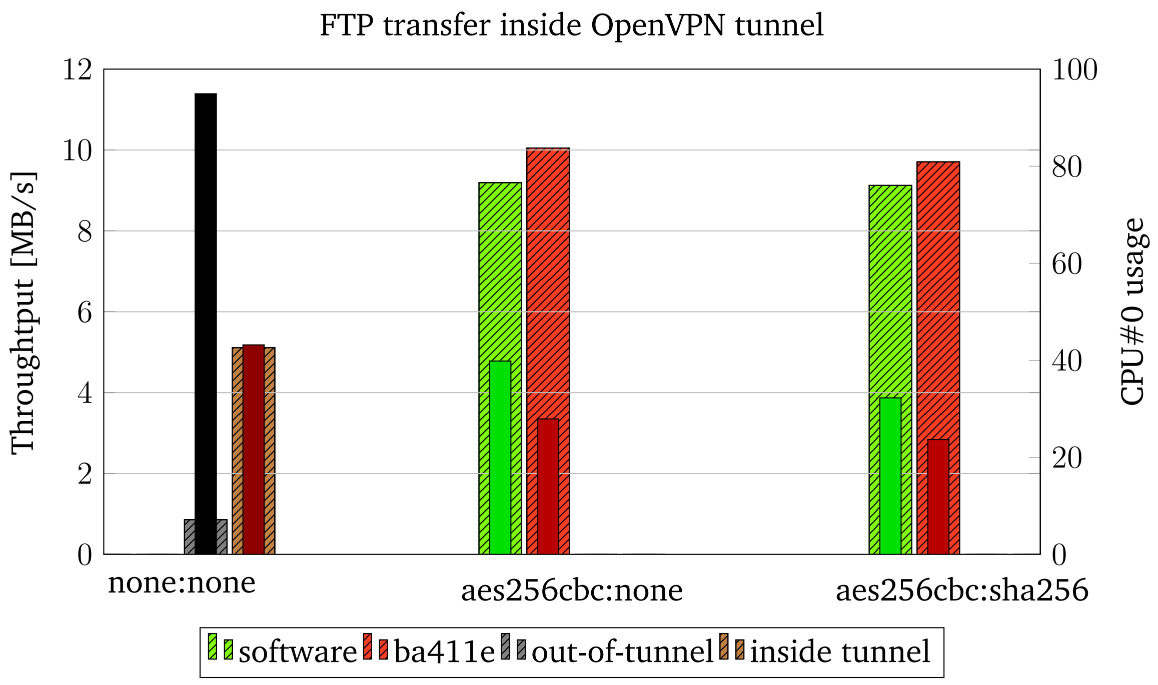


Figure 5.6 – **FTP file transfer over an OpenVPN tunnel. The background stripped bars are the CPU usage.:**

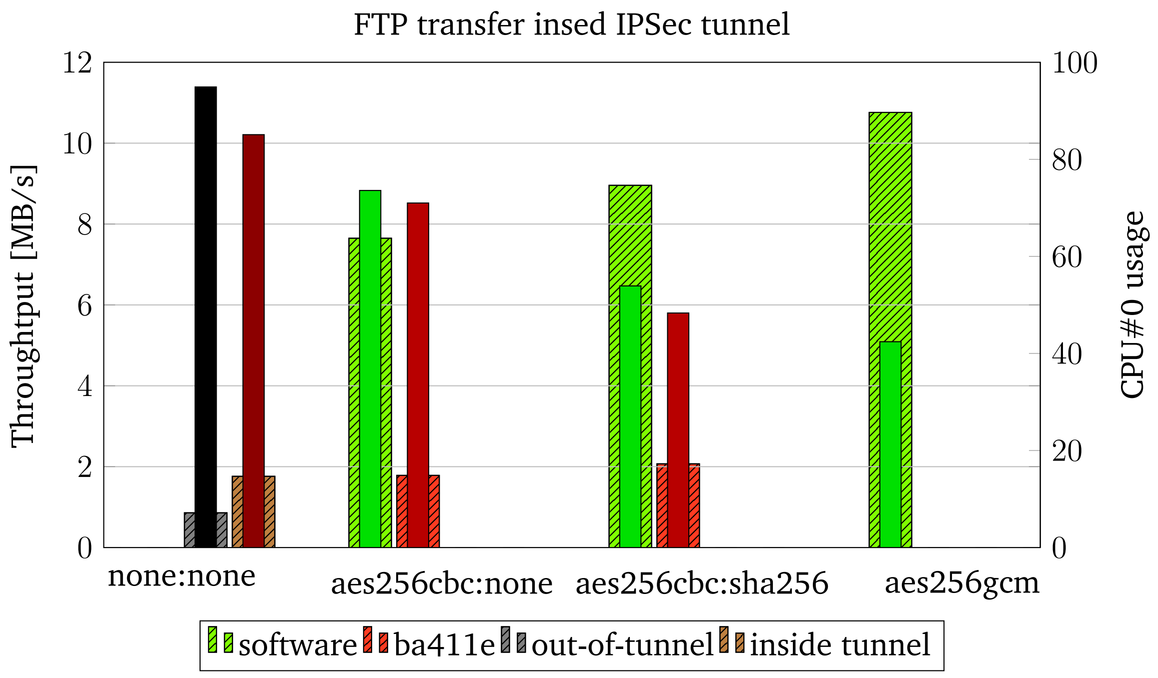


Figure 5.7 – **FTP file transfer over an IPsec tunnel. The background stripped bars are the CPU usage.:**

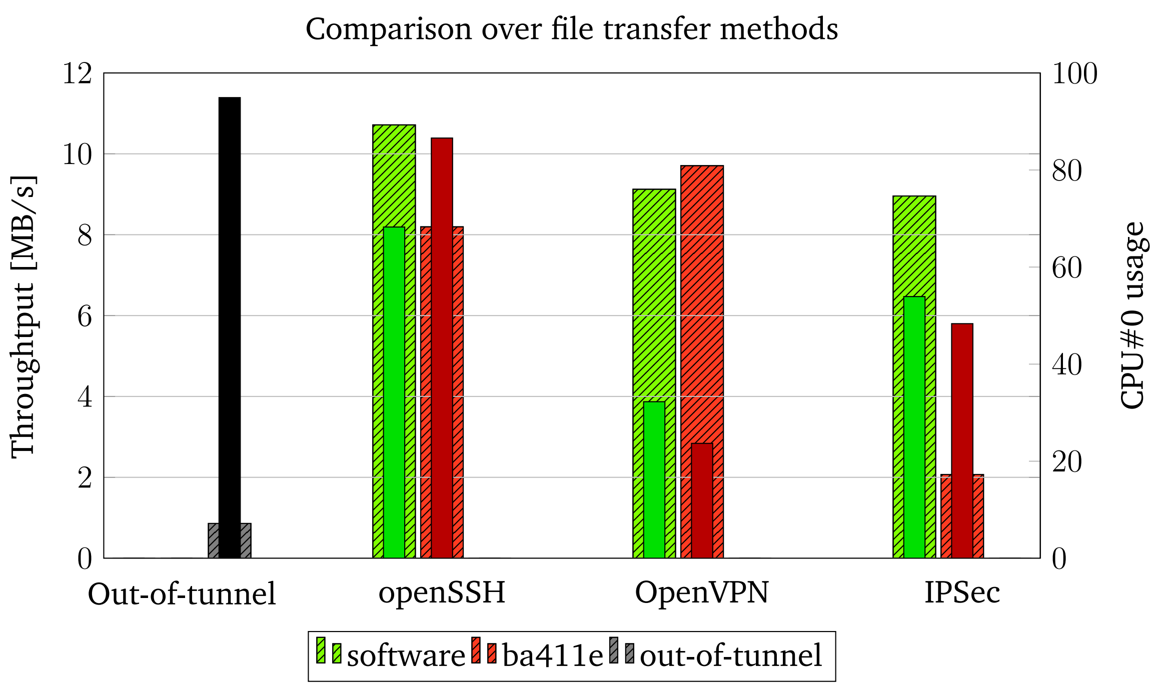


Figure 5.8 – **Comparison of file transfer methods:** The background stripped bars are the CPU usage.

Chapter 6

Discussion and conclusion

6.1 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 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 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 developpement. 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 [27] 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 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 [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 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 some tricks and patches allowed us to bypass this requirement. The implementation of the GCM mode, combining encryption and authentication, showed us

[1]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

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[] = {
  { SSH_DIGEST_SHA256, "SHA256",  32, EVP_sha256 },
  { SSH_DIGEST_SHA384, "SHA384", 48, EVP_sha384 },
  { SSH_DIGEST_SHA512, "SHA512",  64, EVP_sha512 },
+ { SSH_DIGEST_NONE,  "none@barco.com", 0, EVP_md_null },
  { -1,    NULL,  0,  NULL },
 };

diff -rupN openssh-6.7p1_vanilla/mac.c openssh-6.7p1/mac.c
--- openssh-6.7p1_vanilla/mac.c 2014-05-15 06:35:04.000000000 +0200
+++ openssh-6.7p1/mac.c 2015-02-27 12:01:00.204495255 +0100
@@ -87,6 +87,7 @@ static const struct macalg macs[] = {
  { "hmac-ripemd160-etm@openssh.com", SSH_DIGEST, SSH_DIGEST_RIPEMD160, 0, 0,
      0, 1 },
  { "umac-64-etm@openssh.com",  SSH_UMAC, 0, 0, 128, 64, 1 },
  { "umac-128-etm@openssh.com",  SSH_UMAC128, 0, 0, 128, 128, 1 },
+ { "none@barco.com",      SSH_DIGEST, SSH_DIGEST_NONE, 0, 0, 0, 0 },

  { NULL,       0, 0, 0, 0, 0, 0 }
 };
```

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] 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.

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

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

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

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

[9] 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.

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

[11] Fox-IT. OpenVPN-NL. `https://openvpn.fox-it.com/`. Consulted on the 2015-05-24.

[12] 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.

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

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

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

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

[17] 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`.

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

[19] 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`.

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

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

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

[23] 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.

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

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

[26] 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.

[27] 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`.

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

[29] 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`.

[30] 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.