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

Thanks a bunch of people here: - Frédéric Robert for his support - Sébastien Rabou for his insight - Batien Heneffe for his extensive help

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

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 the operating system, followed by the cryptography and networking.

2.1 Operating system

2.2 Cryptography

2.2.1 Symetric cryptography

Talk about encryption, integrity and authentication.

2.2.1.1 AES

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

2.2.1.2 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.

2.2.2 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 ecnryption using the private key of the sender.

2.2.2.1 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. 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 $m = c^d \ mod \ n$

2.2.2.2 Diffie-Hellman

Diffie-Hellman is a secret key exchange protocol: two parties compute a shared secret ZZ that can be used as a symetric key during the following exchanges. It uses the same kind of operation as RSA, that is modular exponentiation. The RFC 2631 [10] decribes the protocol:

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. 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 [2].

2.3 Network and VPN implementation

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

2.3.1 SSL/TLS

2.3.2 IPSec

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 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.

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

4.1 Software

4.1.1 OpenVPN

```
/* Compress, fragment, encrypt and HMAC-sign an outgoing packet. */
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 4.1: openvpn compress then encrypt – sample from `forward.c`

4.1.2 OpenSSL

4.1.3 OpenSSH

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 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.

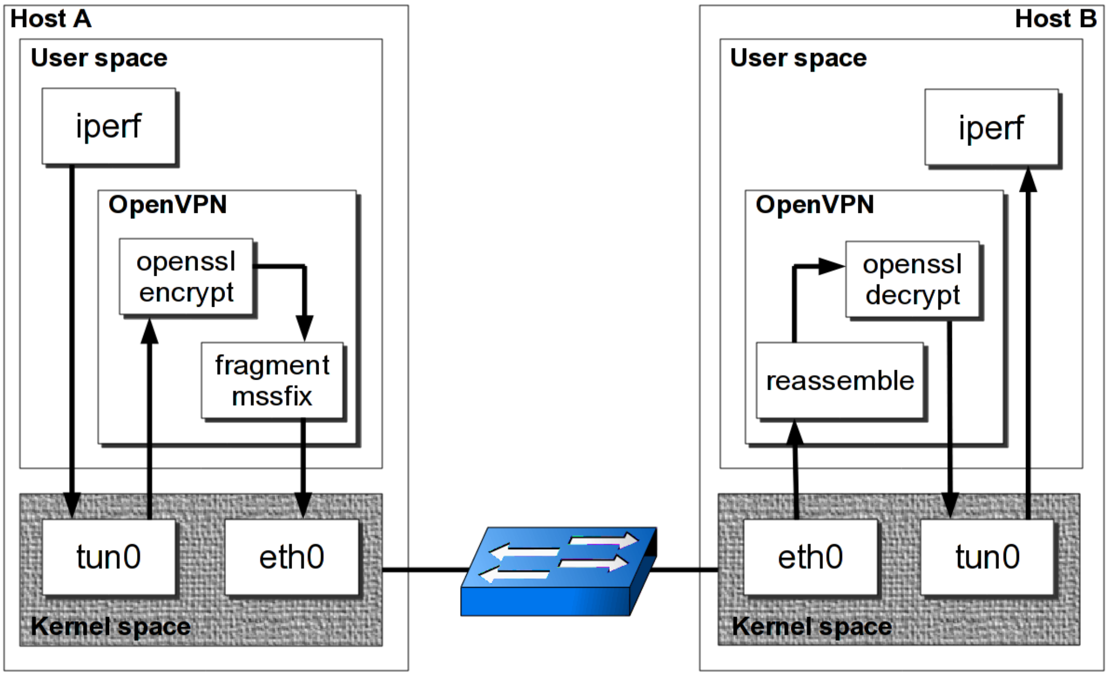

Figure 4.1: OpenVPN packet flow as advertized on the openVPN wiki.

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 4.2 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 [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 incomming packets.

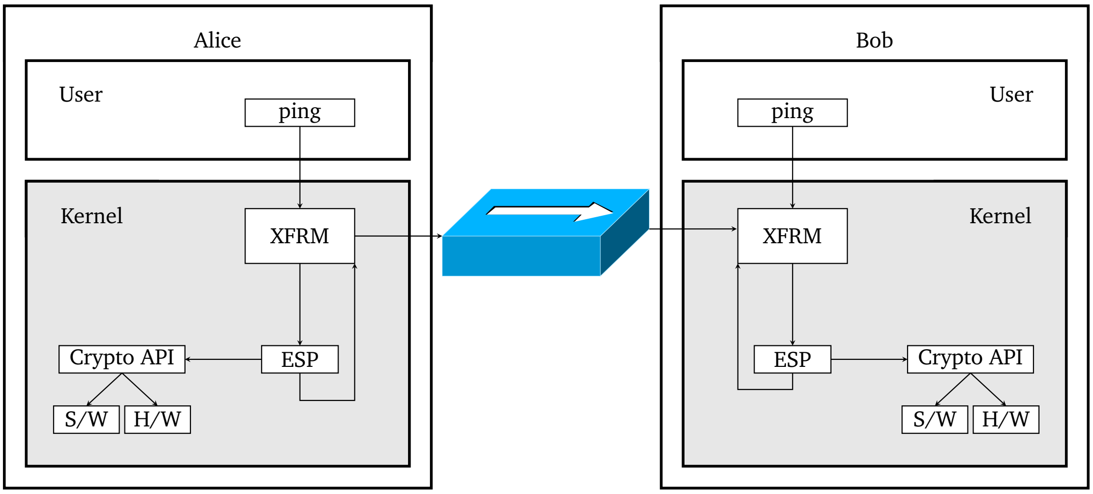

Figure 4.2: IPSec user/kernel space workflow, using `ping` as a test case.

4.1.5 Linux drivers

4.2 Offload

4.2.1 Silex engine

4.2.2 BA411E Driver

4.2.3 BA414E Driver

Chapter 5

Results

5.1 Response time – latency

5.2 TLS connections

5.3 File transfer

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 introduced in late April 2015 [14] 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 [1]. 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

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 B

Cross-compilation

B.1 OpenSSL

B.2 OpenVPN

B.3 nginx

B.4 pure-ftpd

B.5 Strongswan

B.6 kernel

Force kernel version

Change list of modules (show the final version)

Appendix C

Stuff

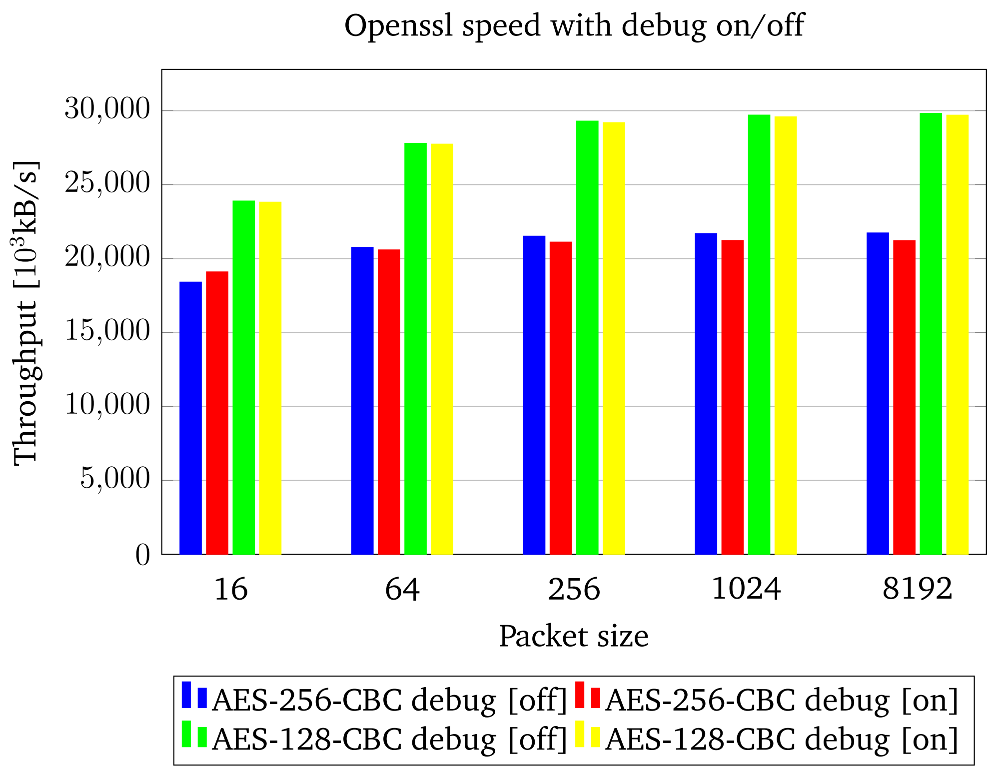

Figure C.1: Software benchmark of Openssl speed for AES mode CBC, with 128- and 256-bit keys, debugging flags (de)activated at compilation (`-fno-inline -g -marm`).

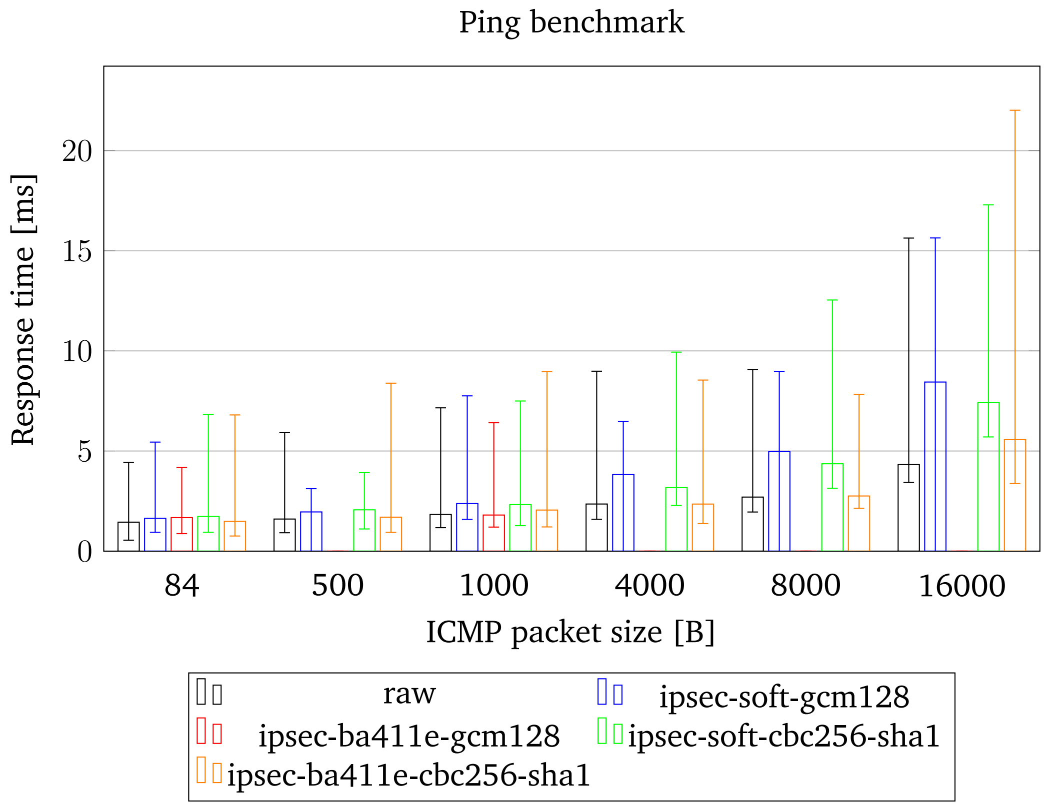

Figure C.2: Ping min/avg/max response time for different packet sizes using IPsec. For each packet size, 1000 requests were flooded to the board, that is "*outputs packets as fast as they come back or one hundred times per second, whichever is more*", according to the `ping` command manual.

Bibliography

[1] 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`.

[2] 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.

[3] C. Michael Chernick, Charles Edington, III, Matthew J. Fanto, and Rob Rosenthal. SP 800-52. Guidelines for the Selection and Use of Transport Layer Security (TLS) Implementations. Technical report, Gaithersburg, MD, United States, 2005.

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

[5] Intel Corp. Using Intel® AES New Instructions and PCLMULQDQ to Significantly Improve IPSec Performance on Linux. Intel White Paper, August 2011.

[6] Neil Dunbar. IPsec networking standards — an overview. *Information Security Technical Report*, 6(1):35 – 48, 2001. ISSN 1363-4127. doi: http://dx.doi.org/10.1016/S1363-4127(01)00106-6.

[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] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: how secure is ssl?). Cryptology ePrint Archive, Report 2001/045, 2001. `http://eprint.iacr.org/`.

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

[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] 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`.

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

[16] J. Viega and D. McGrew. The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP). RFC 4106, RFC Editor, June 2005. URL `https://tools.ietf.org/rfc/rfc4106.txt`.

[17] 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.