Alberto Ferrante

Security Association caching of a dedicated IPSec crypto-processor: dimensioning the cache and software interface

Supervisor: Prof. Roberto Negrini
(Politecnico di Milano)
Assistant supervisor: Dr. Jefferson Owen
(ST Microelectronics)

May 2002

Acknowledgements

This work is dedicated to my parents and all my relatives for their help and support throughout my life and my academic career, for which I will always be grateful.

I would like to thank Jeff Owen and Prof. Roberto Negrini for their fundamental help in developing this thesis.

I also want to thank my friends for playing such a fundamental role in my life, sharing with me the good and the bad moments. Among the others I want to especially thank Laura, Stefania, and Stephanie who also spent some of their time giving me precious suggestions on the language used in my thesis.

In addition, I want to thank all the people of ALaRI: the secretariat, the professors, and all my master's colleagues, both for the important contribution to this project and for having shared with me such an important and positive experience.

A special thanks to *2 Emme Progetti* that lent me a laser printer to make hardcopies of this thesis.

Last, but not least, I want to thank all my university fellows who gave me their friendship, shared with me some knowledge or had simply some good time with me.

Notes on the project

The first part of this project was developed during the 2000-2001 edition of the *Advanced Learning and Research Institute* (ALaRI) *Master of Engineering in Embedded System Design*, organized by *Università della Svizzera italiana* sited in Lugano (CH) in collaboration with *Politecnico di Milano* and *ETH Zürich* (CH).

That part of the project was also presented on October 10, 2001 at the Swiss conference *Technology Leadership Day 2001* at the *Ecole d'Ingénieurs et d'Architectes de Fribourg* (CH). The presentation, titled «*The Smart Card System Project: from "Plastic Money" to Mobile Transaction Support*», was made in collaboration with ing. Luca Mazzoni.

Table of Contents

List of figures

List of tables

Abstract

Security in a network environment is a very important requirement in many of the applications being developed today. This project's objective is to study the software interface needed for using a smart-card-like crypto-processor on a system running the IPSec protocol.

Security in such systems is becoming important as the use of smart cards is being increasingly adopted by the industry. In addition to that, with the increasing networking and openly interconnected environments, new dimensions are added to the security models. As the world head towards heterogeneous computing, security issues between different devices are becoming increasingly important. There need to be mechanisms in place to ensure that the transfer of information is carried out in a secure way. On the other side, an accurate study of the crypto-systems' performance is needed to allow them supporting the performance required in modern network environments.

1. Introduction

1.1. Why we do need security and cryptography

In today's world where our lives are becoming increasingly based on computers and information transmission, being able to store and transmit data in a secure way has became extremely important. E-commerce is a typical example of an application in which more security (data protection, authentication, and certification) is needed, but it is not the only one. For example we can think about the security issues in dealing with more common applications; such as, conversing on mobile, protecting sensitive information contained in medical smart cards, or simply sending confidential e-mails. In addiction, having increasing numbers of companies distributed all over the world, it has became exceedingly important to protect information sent between the various seat of the same company or, in other words, to create virtual private networks (VPN).

So, what can cryptography do to solve these kinds of problems? Cryptography can be used to protect and authenticate data to allow them to be stored and/or transmitted in the securest way possible.

With the advance of hardware in the last few years, it is becoming easier to break cryptographic codes generated by older encryption algorithms. A cryptographic code is said to be broken when someone is somehow able to read the information contained in it without being authorized (i.e. having the key in a legal way). Therefore new cryptographic algorithms have been developed taking into account their possible uses in emerging applications. Applications such as embedded systems like smart cards, cards where some memory and a processor are provided. This has added consequences in that these new algorithms should provide the possibility to run on low power and with high performance on small processors, while allowing a high level of data protection.

1.2. Cryptographic algorithms

There are two kinds of cryptographic algorithms, the symmetric and the public key algorithms. The former ones are faster and very secure, but need to have a pre-shared secret key. The latter ones are slower but not less secure (if the right key-dimension is chosen) and do not need to have a pre-shared secret key. A brief description of the two algorithm classes and a presentation of the Diffie-Hellman protocol are discussed below.

1.2.1. The symmetric key algorithms

These types of algorithms work by using a pre-shared secret key; essentially, some transformations involving that key are applied to the data to be codified. Some different working patterns exist for these algorithms, based on the application for which they have to be used. For example there is the CBC mode that is suitable for applications in which big blocks of data must be transmitted [MOV, section 1.5].

The most widely used of this class of algorithms is triple-DES (Digital Encryption Standard), a variation of the old (1977) DES. A new cryptographic algorithm called AES (Advanced Encryption Standard) was selected from competing candidates by NSA, replacing the triple-DES. In the near future AES will probably become the de-facto standard, as triple-DES is currently.

1.2.2. The public key algorithms

These kinds of algorithms solve the problem of having a pre-shared key by using asymmetric cryptography techniques. Two keys for every peer are needed, one that is called private and that is known only by the owner, and another called public and known to everyone that wants to communicate with that subject. Some transformations, based on the public key, are applied to every communication directed to that subject. This makes the data inaccessible to everyone that does not have the private key. As a matter of fact the inverse transformations cannot be applied by only knowing the public key [MOV, section 1.8].

Nowadays the most widely used algorithm of this class is RSA, but new algorithms, such as the ECC (Elliptic Curve Cryptography), have been developed and probably will become the dominating ones very soon.

1.2.3. The Diffie-Hellman protocol

Diffie-Hellman provides a solution to the key exchange problem by allowing two parties, never having met in advance or having shared keying material, to establish a shared key secret by exchanging messages over an open channel. The key is exchanged in the following way:

- The first peer *(A)* chooses a random secret called x, does some operations on it and sends the result *(h)* to *B*;
- The second peer *(B)* chooses a random secret called *y*, does some operations on it and sends the result *(k)* to *A*;
- B receives *h* from *A* and computes the key using *h* and *y*
- B receives *k* from *B* and computes the key using *k* and *x*

The key protection is given by the fact that the operations performed on *x* and *y* to obtain *h* and *k*, needs a lot of computational time to be inverted so that eventual third parties discovering the values of *h* and *k* would not anyway be able to discover the key in a reasonable amount of time.

The operations that can be applied to *x* and *y* to obtain *h* and *k* can be based either on elliptic curves or on exponentials in the discrete fields. The former case is based on the same principles of ECC, while the other is based on RSA.

See [MOV] on section 12.6 and [RHS] for more information about Diffie-Hellman and the key exchange procedures.

1.3. Authentication algorithms

These types of algorithms are used to certify that the information comes from a certain person and was not modified by someone else. This can be done by computing a hash function of the data and applying to that result an encryption algorithm (that can be a public or a symmetric key one) [MOV, section 1.7]. A hash function is one that, when applied to some data, provides a different brief code for every different packet of data, or, at least provides equal codes for different data with a really low probability [MOV, section 1.9].

1.4. The IETF IPSec protocol suite

IPSec is a protocol developed to provide secure communications on untrusted networks adding some security services to the ISO-OSI network level (i.e. to the IP protocol layer) [RFC-2401]. It is also the Internet Engineering Task Force (IETF) proposed standard for "layer 3 real-time communication security." IPSec can be thought of as a protocol that operates on top of Layer 3 (IP) but below layer 4 (TCP). This infers that it encrypts data independently of all others. If packets happen to be lost, the layer 4 sees only validated information [IEEE-1].

IPSec proposes three different security protocols:

- Authentication Header (AH);
- Encapsulating Security Payload (ESP);
- Internet Key Exchange (IKE).

The first is used to protect the IP headers, while the second is for protecting the content of the IP datagrams. The third protocol is used to perform the key exchange and the algorithm negotiation. The first two protocols can be combined in different ways to offer different levels of security services according to the established system's security policy.

The main method used for AH is applying an authentication algorithm on all the header's fields that are not changed during the packet transmission[1]. HMAC-MD5 and HMAC-SHA1 are the two alternatives proposed [RFC-2402] [RFC-2406] as minimum requirements for IPSec conformance, but new authentication algorithm such as HMAC-SHA-256, HMAC-SHA-348, and HMAC-SHA-512 [DRAFT-4] are under

[1] Some IP headers' fields are often changed when the datagrams go through different gateways before reaching their final destination.

development. In addition, there is another required-to-implement authentication algorithm that is the NULL algorithm: an algorithm that does nothing.

The ESP protocol is implemented by applying an encryption algorithm on the data to be transmitted, but not on the IP header (except from the case of IPSec tunneling, as explained later on in this paragraph). The required-to-implement algorithms for IPSec compliant implementations are triple DES and the NULL algorithm, but, since IPSec was designed with flexibility and extendibility in mind, other encryption methods, such as AES, can be added.

Both in AH and ESP a simple and efficient anti-reply mechanism is provided: a monotonically increasing 32-bit counter is used to implement this feature [RFC-2406]. Anti-reply is a process in which if someone were to intercept one of the packets exchanged by the two peers that are communicating, he could not use that packet to reply to one of the two peers to obtain reserved information (such as the symmetric key) – he would need to know the value of a field that is cryptographically encoded. Anti-reply is also called "partial sequence integrity".

So, to summarize:

- AH provides connectionless integrity, data origin authentication, and optional anti-replying service;
- ESP may provide confidentiality (using encryption) and may also provide connectionless integrity, data origin authentication, and anti-reply service if used in tunnel mode as explained later on in this document.

1.4.1. The concept of IPSec Security Association

An association in which either the AH or the ESP protocol (but not both) is used to communicate, is called IPSec Security Association (IPSec SA). The suffix "IPSec" is used to distinguish that kind of SAs from the Internet Security Association Key Management Protocol (ISAKMP) SAs that can be used only for key exchanging and algorithm negotiation. In this document, unless specified, all the SA-word occurrences will be related to IPSec SAs.

An IPSec Security Association is a one-way association between two peers, so, in order to have a bi-directional communication channel, the creation of two SAs is needed.

For providing particular protection services, multiple SAs can be employed; this is called a “SA bundle”. The order of the SA sequence is defined by the security policy, therefore if both AH and ESP are needed it will be necessary to create two SAs, one for AH and the other for ESP, so that these two SAs will be “nested” as required by the security policy.

We can note that, since a strong enough encryption algorithm is used, using ESP can offer the maximum level of protection available (see the ESP tunnel mode in the next section), so that nesting AH and ESP SAs seems unnecessary. As a matter of fact, AH is often seen as an additional, but not useful complication added to IPSec ([IEEE-1], [IEEE-2]). From what is stated in the IPSec RFCs, the AH protocol seems to have been kept for backward compatibility purposes.

1.4.2. The Transport and the Tunnel modes

Both tunnel and transport mode can be coupled with the AH or with the ESP protocol.

The use of tunnel mode allows the inner IP header to be protected, concealing the identities of the (ultimate) traffic source and destination. ESP padding can also be invoked to hide the real packet’s size.

The Transport mode provides protection only for the upper layer protocols. Hence, by using ESP, the IP header won’t be protected, while using AH only some selected IP header fields will be protected. As shown in Figure 1.1, the ESP transport mode protects the datagram’s data payload, while ESP tunnel mode protects both the IP headers and the data payload as shown in Figure 1.2.

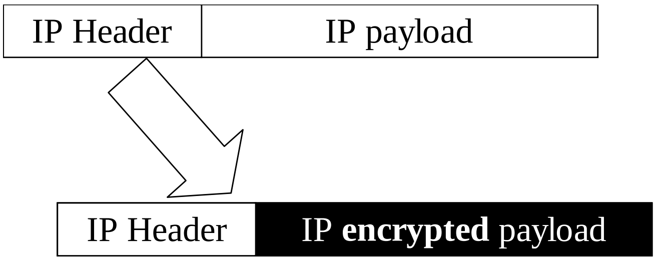


Figure 1.1: ESP in transport mode

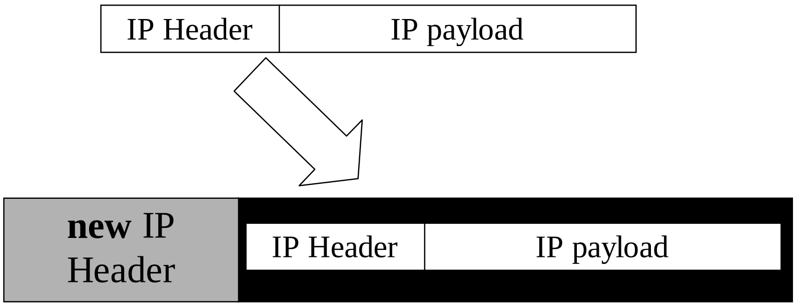


Figure 1.2: ESP in tunnel mode

In the same manner, Figure 1.3 shows the behavior of the AH protocol in transport mode: the IP header is the only protected (hashed) part there. In Figure 1.4 the behavior of the AH protocol in tunnel mode is displayed: there, both the IP header and the payload are protected (hashed). In both the figures the AH field represents the part added by the AH protocol (hash, SPI, …).

In the four figures shown here, the parts colored in black are the ones protected by cryptography or hashing; in AH the IP headers are never completely protected (i.e. some fields are not hashed, as explained before). All four figures are referred to IPSec used in combination with IP v.4. Slightly different figures can be drawn for IP v.6, since its structure allows a better IPSec integration. The effects obtained using tunnel and transport modes are exactly the same for both IP v.4 and IP v.6. The four figures shown here are only a simplified view of the IP datagrams used for IPSec. More detailed information about the IPSec modes and the exact composition of the datagrams can be found in [RFC-2402] and in [RFC-2407].

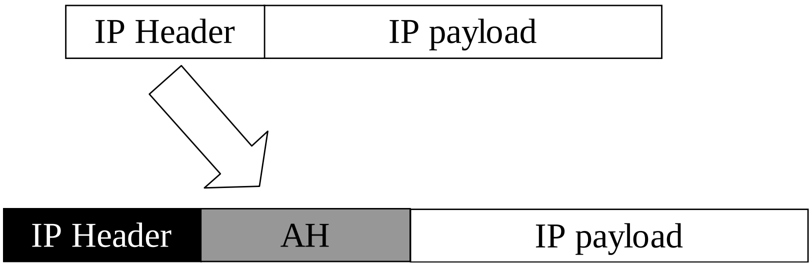


Figure 1.3: AH in transport mode

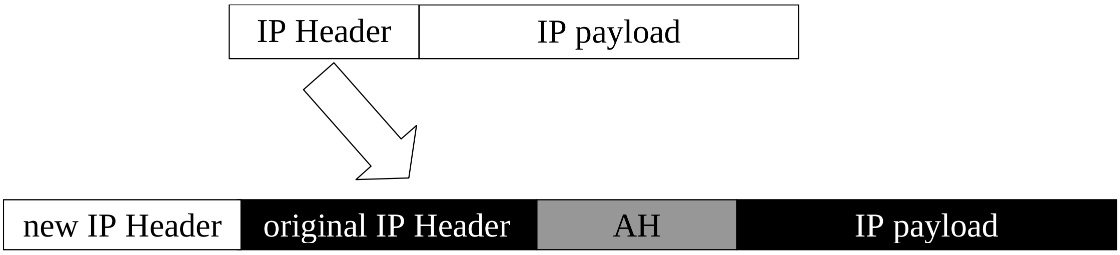


Figure 1.4 AH in tunnel mode

The (ESP) tunnel mode was primarily thought for being used in gateways or routers.

1.4.3. IKE: the key exchanging and the algorithm negotiation mechanisms

As stated before, a mechanism for symmetric key exchange and for algorithm negotiation is needed. It is important to note that the keys have to be exchanged in a secure way while the algorithm negotiation can be done in a non-protected way, which is the fundamental principle of cryptography (the strength of a cryptography algorithm is not given by hiding the algorithm itself, but by hiding the key). There is also a provision for protected negotiation in order to hide the identity of the peers or some other private information.

All mechanisms related to the creation of an IPSec SA must be done at the application layer and are described by the Internet Key Exchange (IKE) protocol. IKE is the interpretation of the Internet Security Association And Key Management Protocol (ISAKMP) in the IPSec domain. Therefore IKE is said to be the Domain of Interpretation (DoI) of ISAKMP. ISAKMP is a protocol describing how key exchange and algorithm negotiation is done over the Internet network.

The creation of an IPSec SA is completed in two phases, first an ISAKMP SA between the two peers is created (phase 1), and then that ISAKMP SA is used to negotiate the information about the IPSec SAs that have to be created (phase 2) [RFC-2409]. ISAKMP SAs are nothing more than a kind of secure tunnel for the creation of IPSec SAs.

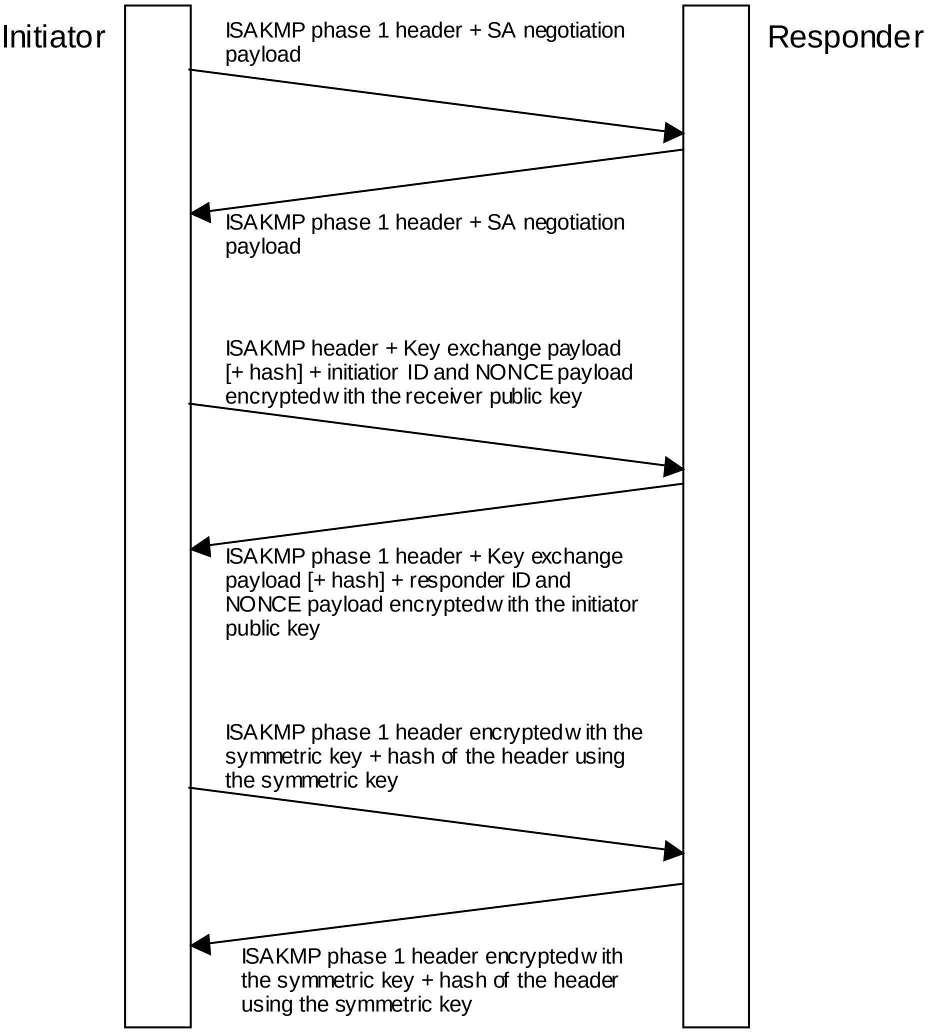

Initiator
Responder
ISAKMP phase 1 header + SA negotiation payload
ISAKMP phase 1 header + SA negotiation payload
ISAKMP header + Key exchange payload [+ hash] + initiatior ID and NONCE payload encryptedwith the receiver public key
ISAKMP phase 1 header + Key exchange payload [+ hash] + responder ID and NONCE payload encryptedwith the initiator public key
ISAKMP phase 1 header encryptedwith the symmetric key + hash of the header using the symmetric key
ISAKMP phase 1 header encryptedwith the symmetric key + hash of the header using the symmetric key

Figure 1.5: *IKE Phase 1* exchange

IKE provides several methods for the phase 1 negotiation with different levels of protection. The key exchange mechanism is based on the Diffie-Hellman algorithm. In Figure 1.5, one of the phase 1 negotiation method is shown: here a public key encryption algorithm is used for authentication, while the secure channel for the phase 2 is created during the second message exchange between the two peers, by using a symmetric encryption algorithm. The algorithms to be used are negotiated during the first message exchange between the initiator[2] and the responder: first the initiator sends, using the SA negotiation payload field, several complete algorithm proposals as defined by the system security policy, and then the responder puts the only proposal that it could accept in conformance with its security policy database (see section 1.4.4) into the same field of its reply message.

A phase 2 “quick mode” exchange is shown in Figure 1.6. The phase 2 accomplishes the creation of a pair of independent SAs, one for each communication direction. A new pair of IPSec SAs is created by exchanging only three messages: 1) the Initiator requests a new pair of IPSec SAs proposing the encryption and authentication algorithms for use in each of those SAs payload, 2) the Responder may accept one of the initiator’s proposals by always using the SA payload field, and 3) the Initiator confirms the creation of the two IPSec SAs. The last message exchange is done in a symmetric encrypted form using the recently exchanged key. After that message, the two peers are able to communicate through the two secure (unidirectional) channels created during that negotiation. The Initiator of phase 2 can be any of the two peers irrespective of which of the two was the Initiator during the phase 1.

[2] The initiator is the peer that propose to start an ISAKMP SA negotiation, while the other peer is called “responder”

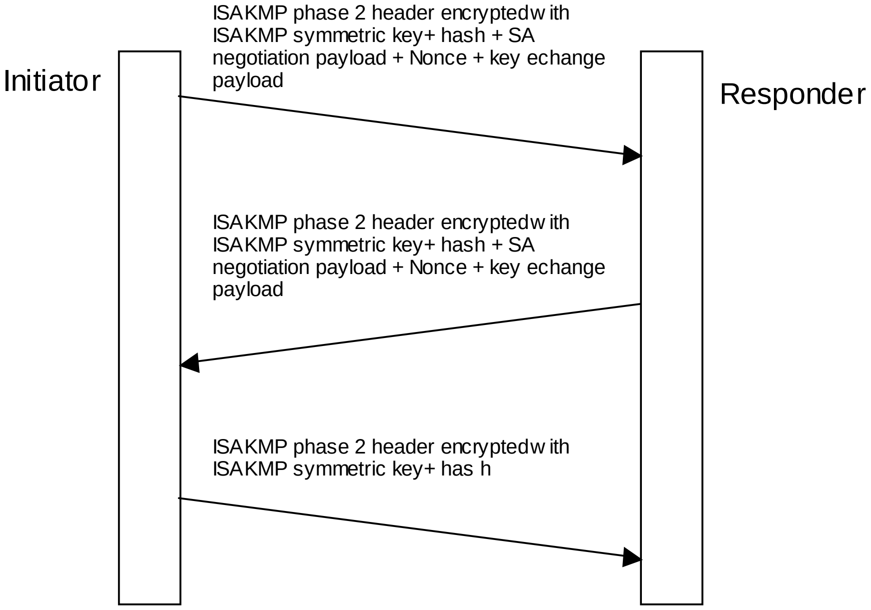

Initiator
Responder
ISAKMP phase 2 header encryptedw ith ISAKMP symmetric key+ hash + SA negotiation payload + Nonce + key echange payload
ISAKMP phase 2 header encryptedw ith ISAKMP symmetric key+ hash + SA negotiation payload + Nonce + key echange payload
ISAKMP phase 2 header encryptedw ith ISAKMP symmetric key+ has h

Figure 1.6: *IKE Phase 2 "quick mode"* exchange

Figure 1.7 shows the creation of an IPSec SA using the standard triple-DES and RSA algorithms. The first part of the figure represents the phase 1 negotiation, i.e. the creation of an ISAKMP SA, while the second part shows the IKE phase 2, i.e. the creation of an IPSec SA. This SA creation sequence is taken from the log file created by the FreeS/Wan IKE – IPSec implementation while it is running. The channel was created between two PCs in the ALaRI lab. FreeS/Wan is a reference IPSec implementation [FSWAN] we used to verify our understanding of the IPSec RFCs.

```
Apr 24 16:38:57 alari011 Pluto[565]: "affsts" #8: initiating Main Mode
Apr 24 16:38:57 alari011 Pluto[565]: | sending:
Apr 24 16:38:57 alari011 Pluto[565]: |   90 de 64 6c  77 45 48 2c  00 00 00 00  00 00 00 00
…………
Apr 24 16:38:57 alari011 Pluto[565]: | ICOOKIE:  90 de 64 6c  77 45 48 2c
Apr 24 16:38:57 alari011 Pluto[565]: | RCOOKIE:  94 5c 12 c3  94 74 a4 68
Apr 24 16:38:57 alari011 Pluto[565]: | peer:  c3 b0 b6 92
Apr 24 16:38:57 alari011 Pluto[565]: | sending:
Apr 24 16:38:57 alari011 Pluto[565]: |   90 de 64 6c  77 45 48 2c  94 5c 12 c3  94 74 a4 68
…………
Apr 24 16:38:57 alari011 Pluto[565]: | *received 180 bytes from 195.176.182.146:500 on eth0
Apr 24 16:38:57 alari011 Pluto[565]: |   90 de 64 6c  77 45 48 2c  94 5c 12 c3  94 74 a4 68
…………
Apr 24 16:38:57 alari011 Pluto[565]: | encrypting:
Apr 24 16:38:57 alari011 Pluto[565]: |   09 00 00 0c  01 00 00 00  c3 b0 b6 90  00 00 01 04
…………
Apr 24 16:38:57 alari011 Pluto[565]: | sending:
Apr 24 16:38:57 alari011 Pluto[565]: |   90 de 64 6c  77 45 48 2c  94 5c 12 c3  94 74 a4 68
…………
Apr 24 16:38:57 alari011 Pluto[565]: | *received 300 bytes from 195.176.182.146:500 on eth0
Apr 24 16:38:57 alari011 Pluto[565]: |   90 de 64 6c  77 45 48 2c  94 5c 12 c3  94 74 a4 68
…………
Apr 24 16:38:57 alari011 Pluto[565]: "affsts" #8: STATE_MAIN_I4: ISAKMP SA established
…………
Apr 24 16:38:58 alari011 Pluto[565]: "affsts" #9: initiating Quick Mode
RSASIG+ENCRYPT+TUNNEL+PFS
Apr 24 16:38:58 alari011 Pluto[565]: | encrypting:
Apr 24 16:38:58 alari011 Pluto[565]: |   01 00 00 18  af 7c 6e a8  19 9c 3d 3e  5d 50 9f f7
…………
Apr 24 16:38:58 alari011 Pluto[565]: | sending:
Apr 24 16:38:58 alari011 Pluto[565]: |   90 de 64 6c  77 45 48 2c  94 5c 12 c3  94 74 a4 68
…………
Apr 24 16:38:58 alari011 Pluto[565]: | *received 260 bytes from 195.176.182.146:500 on eth0
Apr 24 16:38:58 alari011 Pluto[565]: |   90 de 64 6c  77 45 48 2c  94 5c 12 c3  94 74 a4 68
…………
Apr 24 16:38:58 alari011 Pluto[565]: | encrypting:
Apr 24 16:38:59 alari011 Pluto[565]: |   00 00 00 18  b6 d0 c2 c8  c1 e7 b3 65  0f 2b b9 31
Apr 24 16:38:59 alari011 Pluto[565]: |   87 f8 f6 bc  d4 cc 6c 9b
Apr 24 16:38:59 alari011 Pluto[565]: | sending:
Apr 24 16:38:59 alari011 Pluto[565]: |   90 de 64 6c  77 45 48 2c  94 5c 12 c3  94 74 a4 68
…………
Apr 24 16:38:59 alari011 Pluto[565]: "affsts" #9: STATE_QUICK_I2: sent QI2, IPsec SA established
```

Figure 1.7: creation of an IPSec SA with Pluto

Also note that the KEY exchange can be done in a manual or in an automatic mode. The former consists of manually requesting a new key and should be used only on few occasions, for example during testing or in a very small VPN. The latter consists of automatically updating the keys when a 32-bit counter that is incremented every time the SA is used, reached its maximum; this is the mode that offers more reliability and security.

1.4.4. The Security Policy Database (SPD)

The Security Policy Database specifies what services are to be offered to each possible pair basing on its IP address and in what fashion [RFC-2401, section 4.4.1], so the SPD will contain a list of IP address with the corresponding security policies to be adopted.
The SPD must be consulted during the processing of all traffic, including non-IPSec traffic.

1.4.5. The Security Association Database (SAD)

The Security Association Database is a database in which stores all the information related to each opened SA [RFC-2401, section 4.4.3]. Each of them has to be univocally identified by the destination IP address, the IPSec protocol type, and the SPI, a 32-bit value used to distinguish among different SAs terminating at the same destination and using the same IPSec protocol.

1.4.6. Adding AES to IPSec

As there were no stable publicly available documents from the Internet Engineering Task Force (IETF) about this aspect of the project, our work is based on draft documents ([DRAFT-2] and [DRAFT-3]) and thus may have some interoperability problems with different IPSec (and IKE) implementations. This is due to the fact that in some cases we were forced to make certain specific choices not supported by any official documentation (see chapter 3).

1.4.7. Some notes about the available IPSec – IKE documentation

The (many) available RFCs are sometimes confusing giving the possibility of misunderstandings and different interpretations. This could cause interoperability problems as explained in a NIST document [NIST-1].

IKE is really complex to cover a high number of different use cases, and this causes some interpretation difficulties [IEEE-1].

1.5. Network attacks

Here a brief explanation of some kinds of computer attack is given. More detailed information can be found for example in [COM-1].

Network attacks are actions performed to take the control of a system (or a sub-network), to extract data from that system, or to make the system unusable. Network attacks are of increasing concern because of the number of organizations and users on the Internet and their increasing dependency on the Internet to carry out day-to-day business.

Network attacks can be done in many ways, often following some known procedures.

A very common kind of attack is the one called Denial of Service (DoS): here multiple systems are used to attack one or more victim systems. The goal of that kind of attack is to saturate the resources of the victim systems.

Other attacks can be specifically studied for routers: intruders can use poorly secured routers as platforms for generating attack traffic at other sites; routers can easily become victims of DoS attacks being designed to pass a large amount of data through them, but without the capability to handle the same amount of traffic directed to them.

There are also systems called “intrusion detection” systems: these are merely software developed to detect known operational patterns applied on the network. Usually once an attack has been discovered, intrusion detection systems deny these operation to continue and also advise the system administrator. The main problems associated with these types of software are due to the fact that new attack techniques are very often discovered, so that not all the attacks are recognized. On the other hand, these types of software can easily recognize some normal (non-malicious) operations performed on the system as attacks.

2. Description of the global system

The system under development in the ALaRI lab is composed of a smart card (or, more generally speaking, a crypto-processor) implementing some security functionality as described in section 2.1, and a host on which IPSec and IKE are installed and running. The system has to be able to support communications at 50Mbit/s. This system is connected to a network (i.e. Internet) and has to be able to communicate with other IPSec compliant systems that possibly support AES and ECC as described in section 2.3 See Figure 2.1 for a graphical representation of the system.

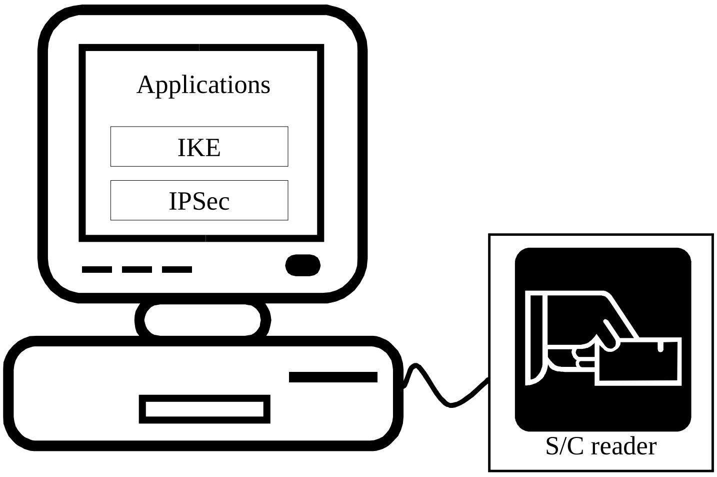


Figure 2.1: representation of the considered system

2.1. The hardware

The host on which IPSec and IKE are running can either be a PC or a different kind of machine (for example a gateway or a firewall), depending on the specific application of the system.

The smart card hardware interface is undefined and outside the scope of the project. It is assumed that the data transmission rate of the interface is enough to support the 50Mbit/s throughput.

The smart card must provide encryption and decryption of data packets using the symmetric AES algorithm and the public key ECC algorithm respectively, in addition to other services needed for the security associations' management. The key point is that both the symmetric and the private key of the public key cryptography algorithm should never exit from the smart card. Depending on the user scenario, it may not be necessary to hide both keys. In section 2.3 examples are provided as to how this feature can be used.

The protection of the keys is the main motivation that makes it necessary to implement the cryptographic algorithms in the smart card.

Slightly different requirements will be considered in chapter 6.

2.2. The software

The software running on the host consists of an implementation of the IKE protocol for key exchange and an implementation of the IPSec protocol. Both of these should be suited to take advantage of the smart card's functionality.

2.3. System usage scenarios

The system can be used in many different ways. One way that the smart card associated with IPSec can be used is to create various kinds of Virtual Private Networks (VPN) or to create a secure communication channel between two gateways. A VPN is a private network created over a public network (e.g. internet) using some security systems to provide confidentiality and authentication. Here follows the description of some of these systems.

2.3.1. The simplest VPN scenario: two users communicating through Internet

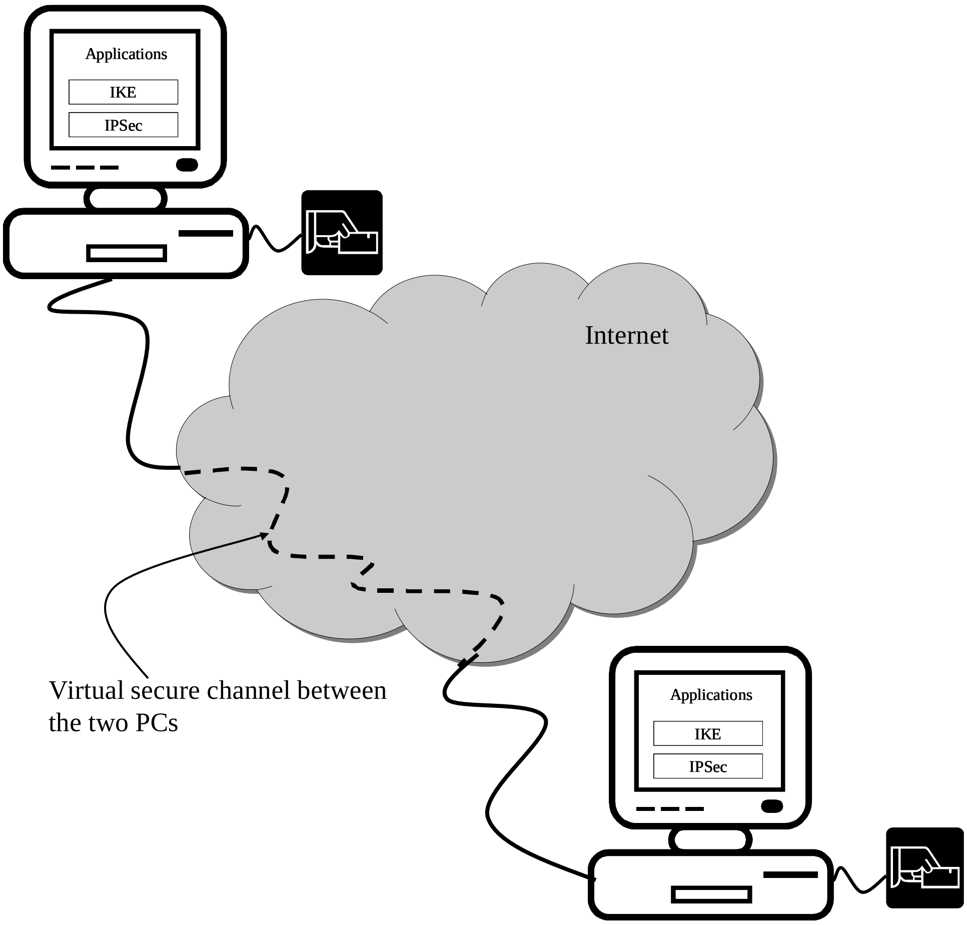


Figure 2.2: VPN scenario – 2 clients connected on a VPN through Internet

In this scenario a secure communication channel between two peers is established over an untrusted network like the Internet. The channel is called “virtual” because the Internet connectionless standard is still used, so no real permanent communication channel is provided.

In Figure 2.2, two PCs are represented but these can be substituted with other kinds of machines such as embedded systems, mobile devices, handheld devices, etc.

In this scenario the main use of the smart card is to provide authentication of the two users, so it is important that the ECC private key is stored in a protected manner in the smart card. This service is already provided by smart cards currently available on the market, but in these systems the private key has to exit from the smart card because the cryptography algorithms are not contained there. The smart card also provides other services such as symmetric encryption and authentication.

2.3.2. A more complex VPN scenario: a mobile user connected to his company's network through Internet

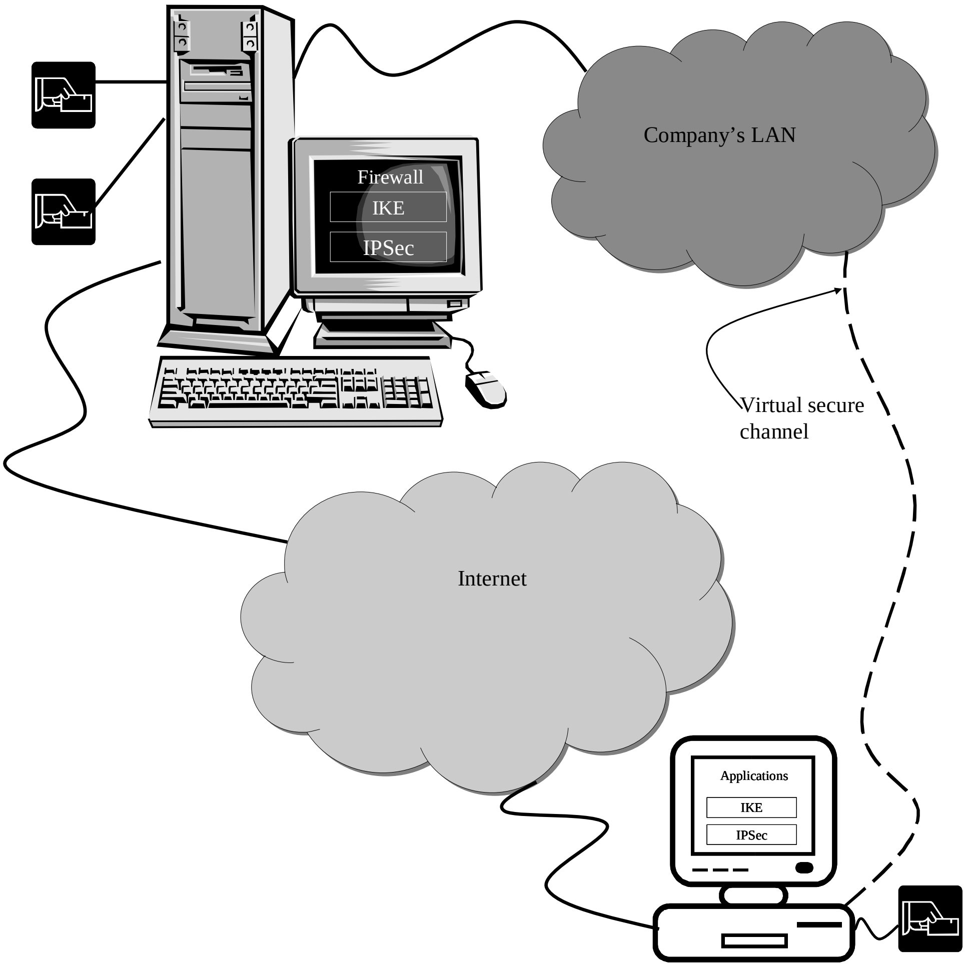


Figure 2.3: Client connected to his company's internal network in a secure way using Internet

As shown in Figure 2.3, the user is connected to the company's internal network via the Internet; this scenario is often called "road warrior". The company's LAN is protected by a firewall. In this case the use of such a system allows both users authentication and secure communication. Here the user is reliably connected to the company's LAN as if he was physically connected to the company's LAN[3]. In this case the secure channel is opened between the user's PC (or other kind of machine) and the firewall.

Here the smart card provides authentication when used on the PC side, and both authentication and data protection when used on the firewall side, so in this case it is important that both keys are kept secret.

Figure 2.4 shows that more than one smart card can (and should) be used at the firewall side to provide better performance. The smart card can also be substituted by a similar, but more powerful, crypto-processor, to create even better performance than the smart card alone as shown in chapter 6. Later in this document it is shown that network configurations such as the one here illustrated can expose the system to DoS attacks, if not carefully configured.

[3] This is not totally true: cryptography is only able to provide a certain level of security, depending on the adopted algorithm and on the key size, but it cannot guarantee, as any other protection mechanism, the total security of the data.

2.3.3. Data tunneling between two gateways

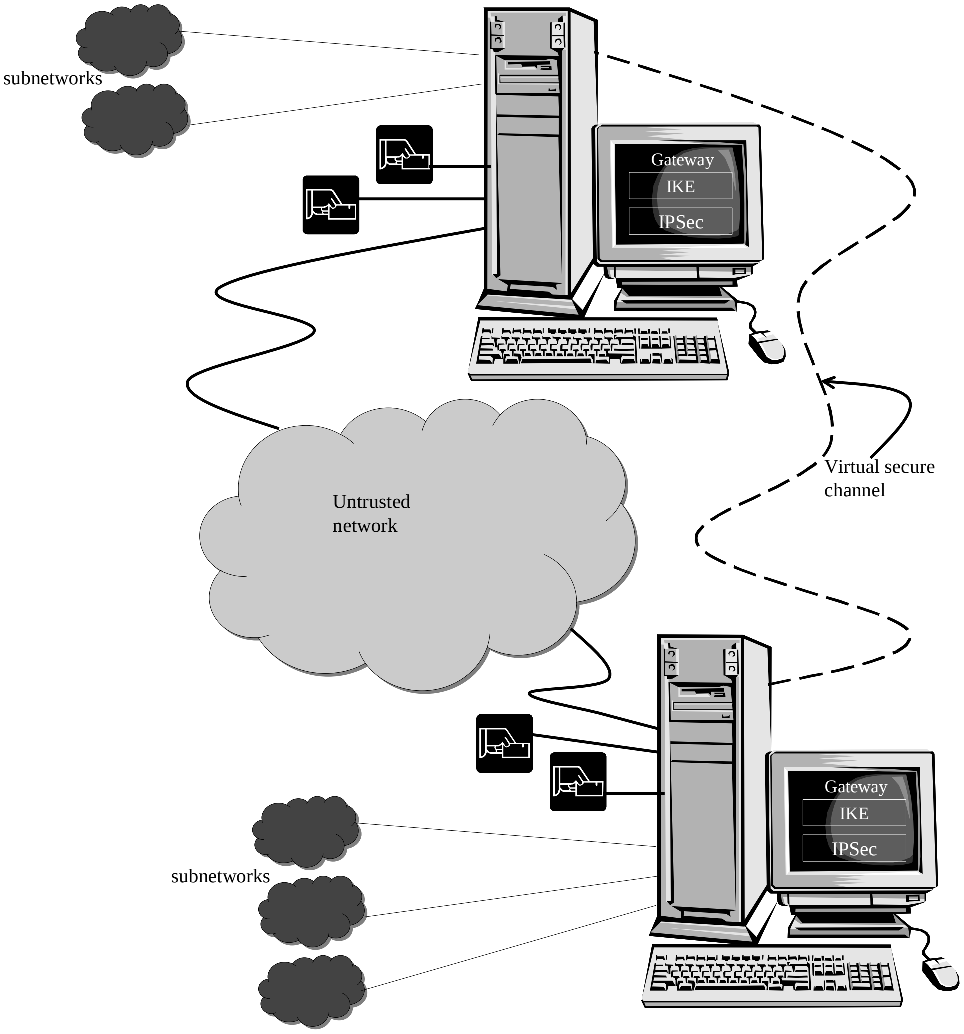


Figure 2.4: data tunneling between two gateways

In this case a secure channel is created to protect data transmitted between the two gateways, enabling multiple secure communications from different sub-networks at one time.

In this scenario, the most important service provided by the smart card is data encryption.

On gateways as well as on firewalls multiple smart cards could and should be used to enhance performance, or, as in the previous case, a crypto-processor can be utilized even more effectively than just the smart card.

3. The security policy

Before start working on the design of a security system, it is necessary to specify the level of security we need. This section, where the desired system's behavior is described, accomplishes that needing.

In the following subsection we describe the algorithms to be used during the key negotiation and during the communications, and how to use these algorithms. The key exchange mechanisms, and the support that the smart card should provide for IKE and IPSec are also discussed.

Since security is highly connected with the system's performance, some consideration about the latter topic are also provided in this part of the present document.

3.1. The key Exchange mechanism: creation of SAs

To create a pair of mono-directional IPSEC SAs (see the section 3.1.2 for further details about this), the following steps are needed ([RFC-2409], section 8):

- Phase1 : do a main mode exchange to create an ISAKMP SA
- Phase 2: do a quick mode exchange to establish the needed IPSec SAs (at least 2 mono-directional) (phase 2)
- delete the ISAKMP SA and its associated states.

3.1.1. Phase 1 Key exchange – ISAKMP SA

IKE Phase 1 creates a security channel to exchange SA information. This channel consist of a special bi-directional SA that is called ISAKMP SA. That security association is used only for creating new mono directional IPSec SAs and eventually it can be deleted. The maximum lifetime for an ISAKMP SA is of 24 hours.

According to [RFC-2408] and [RFC-2409], the key exchange policy has to be re-negotiated every time a new ISAKMP SA has to be established *(«The following* *attributes are used by IKE and are negotiated as part of the ISAKMP Security Association. […]*

encryption algorithm

hash algorithm

authentication method

information about a group over which to do Diffie-Hellman.

All of these attributes are mandatory and MUST be negotiated. In addition, it is possible to optionally negotiate a pseudo-random function ("prf").»).

The versions of Phase 1 negotiation techniques based on public key encryption (e.g. *Phase 1 authenticated with Public Key encryption*) will be implemented in our system through Elliptic Curve Cryptography (ECC).

The section of RFC 2409 quoted above and interpreted in our context means that each time it is necessary to exchange an ISAKMP key (that is to create a new ISAKMP SA), the tables related to a specific ECC curve have to be re-computed. This is necessary unless the curve’s parameters used during the previous ECC-based computation are the same as the one to be used for the new one.

We can also note that, once an ISAKMP SA has been created, we can use it for the creation of many other IPSec SAs by repeating IKE phase 2. As a matter of fact the PFS[4] mechanism ensures that the generated keys are independent. Therefore very few ECC encryption/decryption operations may be really needed (see [RFC-2409], section 5.5).

The ECC curve information are used in all the ECC based key exchanges, that is at least one time for each phase 1 negotiation and sometimes in phase 2.

According to [RFC-2408] section 4.3, security association modification within IKE is accomplished by creating a new SA and initiating communications using that new SA. Deletion of the old SA can be done anytime after the new SA is established.

Modification of an ISAKMP SA follows the same procedure as deletion of an ISAKMP SA.

[4] Perfect Forward Secrecy

According to [DRAFT-1] we have thirteen possible default groups for elliptic curve Diffie-Hellman that must be supported.

3.1.1.1. Phase 1 Key exchange procedure:

According to [RFC-2409] section 5, the following key exchange procedures are used:

- Phase 1 Authenticated With Signatures
- Phase 1 Authenticated With Public Key Encryption
    - Main mode
    - Aggressive mode
- Phase 1 Authenticated With a Revised Mode of Public Key Encryption
- Phase 1 Authenticated With a Pre-Shared Key

3.1.2. Phase 2 key exchange – IPSEC (AH-ESP) SA

The phase 2 ([RFC-2409], section 5.5) is used to exchange keys for IPSec sessions when an ISAKMP SA has already been created.

According to [RFC-2401] section 4.1, a SA is unidirectional. This means that to establish a standard bi-directional connection, two SAs have to be created.

If both the AH and the ESP protocols are applied to a traffic stream, two SAs (for each direction) will be created. Those SAs will be nested as specified in the security policy database.

In our system the IPSec SAs use the fast symmetric key encryption algorithm called AES for protecting the data. To prevent AES context switching inside the smart card (interleaving data blocks belonging to different SAs), the IPSec process on the host should send consecutive data blocks related to a single SA. This is recommended for optimizing the performance of the system, since AES works best on continuous streams of data.

Modification of an IPSec SA (phase 2 negotiation) follows the same procedure as the creation of a new IPSec SA, that is: a new SA is opened, traffic is moved on it and the old SA is deleted. The creation of the new SA is protected by the existing ISAKMP SA, so that there is no relationship between the old and the new IPSec SAs. A protocol implementation should begin using the newly created SA for outbound traffic and continue to support incoming traffic on the old SA until it is deleted or until traffic is received under the protection of the newly created SA.

3.1.2.1. Phase 2 key exchange mode: the quick mode

There are two possible implementation of the quick mode to be supported (see [RFC-2409], section 5.5): the base and the normal one. In our system we will always propose to use the phase 2 normal mode, which guarantees the Perfect Forward Secrecy of the exchanged keys (if someone discovered the AES key used for communicating the new keys information, he would not be able to discover the latter).

Using the quick mode, two mono-directional SAs are created, one for each end and the keys are derived from the information sent by the other peer.

3.2. Algorithms

IPSec RFCs state some required-to-implement algorithms for IPSec compliant implementations. As stated in section 2.1, our crypto-processor will not support the RSA and DES algorithms, while it will support the AES and ECC ones. For compatibility purposes, RSA and ECC will anyway have to be supported by software.

3.2.1. Public Key Elliptic Curve Criptography

For the ECC algorithm it is needed to support key-sizes up to 600 bits (NIST recommendation).

3.2.2. Simmetric Key AES (Rijndael) Criptography

In the early draft about the use of AES in IPSec [DRAFT-2], it was only required to support 128 bit-wide keys in CBC mode. In the new version of that draft ([DRAFT-3]) it is stated that both the key dimension and the number of algorithm rounds to be performed have to be negotiated. It is still not clear what fields of the IKE phase 1 and phase 2 are to be used for those negotiations.

3.2.3. Hash Algorithms

The mandatory to support hash algorithms are MD5 and SHA-1 ([RFC-2402], [RFC-2406], [RFC-2408]).

3.2.4. Authentication Algorithms (HMAC)

The mandatory to support authentication algorithms are HMAC-MD5 and HMAC-SHA-1 ([RFC-2402], [RFC-2406], [RFC-2408]); these two HMAC algorithms require the knowledge of the symmetric key, therefore they have to be implemented in the smart card.

Those authentication algorithms operate on entire packets of data: it should be investigated if block or stream implementations are possible.

3.2.5. Algorithm negotiation

During the SA negotiation, the encryption and authentication algorithms have to be agreed between the two peers. The initiator has to send some complete proposals (e.g. AES with HMAC-MD5), and the responder can accept one of them or propose a new one.

In our system a symmetric key for a SA (AES key) can only be exchanged using the ECC based Diffie-Hellman algorithm. This is the only method allowed by the smart card. As a matter of fact the key cannot be introduced in any other ways without exposing the card to security attacks.

Consequently it will not be allowed to use the ECC based Diffie-Hellman algorithm (and the ECC algorithm) to exchange secrets that are not AES symmetric keys: it would be possible to generate such a key, but not to retrieve it from the smart card.

According to the previous statements, when the initiator proposes the use of ECC for key exchanging, it will be able to continue the negotiation only if the responder has the AES algorithm available; if not, the negotiation has to be closed and restarted using a public key algorithm different from ECC (e.g. RSA).

The system acting as a responder will be able to accept only proposals corresponding to the previous conditions. Therefore it will be able to exchange an AES key only using

ECC and it will not be able to use ECC for other purposes. If the ECC algorithm is proposed for key exchanging but the initiator doesn't have AES available, the negotiation will be closed, or the null cryptographic algorithm accepted.

The curve parameters for ECC are to be negotiated as indicated in [DRAFT-1].

3.3. Support for a conceptually infinite number of SAs

Due to the fact that the AES keys are never to go out from the smart card, some information, such as the keys, would have to be stored in a local memory and kept there for all the SAs lifetime. As we know, the amount of memory we can put in a smart card (or in a crypto-processor) is really limited so that it is not possible to store there all the information we need. A solution to that problem is discussed in this section.

3.3.1. Smart card SA database

The number of SAs that the smart card can handle at a given time has an upper bound due to the limited amount of memory in the smart card itself. Once this limit has been reached, it is necessary to free some smart card's memory before creating new SAs. This can be done storing outside the data related to an active SA (for example the least recently used one) in a protected form (since the SA symmetric key has to be kept secret).

This mechanism is very similar to the "paging" technique applied in systems with virtual memory or to the replacement technique of cache memories: the main difference with respect to them is that the information stored out of the local memory must be protected in some way.

The data stored out of the smart card are saved in the host memory.

The swapping between the smart card and the host computer introduces a further data processing delay, but it gives the smart card access to the host memory and allows a conceptually unlimited number of opened SAs.

It is up to the IPSec/smart-card interface to manage that possibility in the best way, that is managing the SAs inside the smart card and freeing some memory position when needed. To reach the maximum efficiency of the system, the SAs to be taken in the smart card’s memory can be chosen taking into account the following remarks:

- the system administrator should set a limit to the number of SAs using the smart card, and assign a priority to them;
- the interface between the IPSec implementation and smart the card should monitor the SA traffic in order to keep the most used SAs inside the smart card;
- the number SAs opened at the same time should be carefully monitored to prevent DoS attacks. As a matter of fact allowing too many opened SAs can cause the system to spend more time swapping information between the smart card and the host than doing useful processing (i.e. data encryption/decryption).

On the other side the smart card has to provide to the host all the functionality needed for managing the SAs.

The SAs present in the smart card should be stored in a SA database (for example of 16 entries) addressable by an index: the IPSec host has to include it in each service request sent to the smart card.

To summarize, an entry in the smart card database is needed for each SA that has been created. That database will be kept in a non-accessible dedicated memory area of the smart card. When the smart card memory is full, the information related to an existing SA need to be stored in the host memory to allow the creation of a new SA. The information to be stored in the host memory, are to be protected through encryption.

When an already created SA needs to be restored in the smart card, the host will pass the encrypted information to the smart card. Then the smart card will perform a data integrity check (e.g. through a simple CRC computation), and will store back the decrypted information in a specified position of the SA database.

The details about the algorithms to be used for implementing the data storage described above are given in the following subsection.

3.3.2. AES session key

To protect the SA information that have to be stored outside the smart card, AES encryption can be performed. For that purpose an AES “session” key known only by the smart card can be used. The AES session key must be newly generated each time the smart card is plugged in the reader.

For obtaining the maximum possible level of security, a 256-bit wide AES key can be used.

Looking at the contents of the smart card SA-database records, we can see that the only field that really needs to be protected is the one containing the AES key. For authenticating the data contained in each smart card SA-database entry, a simple 16-bit CRC computation seems to be enough. As a matter of fact, we only need a consistency checking on the stored information and CRC can provide that service allowing very fast and simple hardware implementations. The result of the CRC computation can be encrypted together with the AES key.

To summarize, when an entry of the smart card SA-database needs to be stored in the host memory, first the CRC on the whole entry is computed, then the AES key and the CRC result are encrypted using the AES session key. At the end the encrypted SA-data and the other fields contained in the smart card SA-database entry (i.e. the IV and the AES algorithm settings for that SA) are stored out.

To understand the level of security provided by this solution, we have to take into account that:

- what is stored in encrypted form outside the smart card is, basically, the symmetric key that protects one SA;
- when the IPSec automatic key-refresh service (anti-reply service) is active, the lifetime of each SA cannot in any case exceed either 8 hours or a specified amount of exchanged data (2^{32} datagrams);
- when the key-refresh is manually performed – and that this procedure is usable only in very little VPNs – the smart card will be probably able to carry all the opened SAs in its memory. Only few of the opened SAs will possibly be stored on the host for a small time.

The AES robustness should anyway guarantee that the keys are protected “as they were kept in the smart card”. We can note that it would be more convenient for an attacker to try breaking the AES key related to the SA in which he is interested than the session key, since the keys used for IPSec SAs are usually smaller than the session key.

A special smart card command for refreshing the AES session key must be provided. Active SAs stored outside the smart card will not be anymore accessible after the execution of that command: this is because once the session key has been changed, all the AES keys stored on the host cannot be anymore decrypted. Therefore, when a session key refresh is needed, the IPSec communication module should monitor the number of opened SAs that are using the smart card and request that key refresh when the number of opened SAs is less or equal the number of available smart card SA-database positions. The technique described above can be used only on very small systems. A better mechanism should be studied for more complex systems using, for example, a high performance crypto processor, such the one described in chapter 6. A proposal for this kind of systems can be: once the key-refresh command is given, the old session key is stored in a reserved crypto-processor memory space and it is used only for the SAs that had been swapped out before renewing the session key. For the successive swap operations on the same SAs, the new session key must be used. The technique described above works only if the key-refresh command is given only once in the maximum lifetime for the SAs (8 hours). In that way, the old key needs to be kept for at least 8 hours, then it is no more useful, since the SAs opened before that time must be closed anyway. Using more than two session keys could overcome the problem of being able to perform only one session key refresh during the SAs maximum lifetime. Probably this is not necessary, since today it is almost impossible to break a 256-bit AES cryptographic code in a so short time.

3.3.3. Diffie-Hellman secret

As stated in section 1.2.3, a Diffie-Hellman secret needs to be stored in memory during each key exchange process. In accordance with the team developing the ECC hardware (see [CA-PO]), also using a 600-bit ECC key, 256 bits of memory space is needed to store the Diffie-Hellman secret (considering a maximum symmetric key dimension of

256 bits). This allows us to store the Diffie-Hellman secret related to each SA to be created, in the corresponding key field of the smart card SA cache.

It must be verified if that solution can cause any interoperability problem with the other existing IPSec implementations. If any of those problems arise, the following proposals can be evaluated as alternative solutions:

- use bigger smart card SA cache entries to allow the complete storage of the Diffie-Hellman secret (up to 600 bits). This solution can be suitable for small systems (like a smart card) in which very few cache entries should be present (e.g. 16).
- use a dedicate memory area in the smart card for the storage of the Diffie-Hellman secrets. The dimension of that memory area should be studied to avoid the system to stall because no positions in that area are available for the creation of a new SA. That memory space can also be managed as a cache, implementing a replacing algorithm. In the latter case, the data contained in the cache need to be encrypted before storage to avoid the host machine to be able to compute the symmetric key.

3.4. An IKE-IPSec security policy example: the case of a mobile user (“Road Warrior”)

3.4.1. IKE phase 1

The mode to be preferred in this phase depends on the application. The one that seems to be more suitable for the road warrior application is the *phase 1 authenticated with a revised mode of public key encryption* ([RFC-2409], section 5.3). The algorithms proposed are ECC as public key algorithm, SHA-1 as hash algorithm and AES as symmetric encryption algorithm. If these algorithms are not available, the classical RSA and triple-DES will be used.

128-bit wide keys (with the default number of rounds for that key dimension) will be proposed for the usage with the AES algorithm. As a matter of fact 128-bit wide keys seem to provide a sufficiently high level of protection, using less system resources than the ones used with wider keys. This solution also allow interoperability with all the systems which do not allow negotiation of the AES key-size.

For DES the key dimension cannot be negotiated.

For the elliptic curve cryptography, the ECC Group 8 is proposed. That group is based on a Galois Field GF[2^{283}] (see [DRAFT-1]). Unfortunately, that group is not mandatory-to-implement for IPSec compliant systems. Therefore, if the other peer does not have the ECC Group 8 available, the ECC Group 4 will be proposed instead.

The proposals for the phase 1 are summarized in Table 3.1.

MODE	**Public key algorithm**	**hash algorithm**	**Symmetric key algorithm**
phase 1 authenticated with a revised mode of public key encryption	ECC (Group 8 or 4)	SHA-1	AES (128-bit key)
phase 1 authenticated with a revised mode of public key encryption	ECC (Group 8 or 4)	MD5	AES (128-bit key)
phase 1 authenticated with a revised mode of public key encryption	RSA (Group 4)	SHA-1	3DES
phase 1 authenticated with a revised mode of public key encryption	RSA (Group 4)	MD5	3DES

Table 3.1: *IKE Phase 1* proposals for the “Road Warrior” network ordered by preference

3.4.2. *IKE phase 2*

As explained before, the mode to be preferred is the *main* one that guarantees PFS.

128-bit wide keys (with the default number of rounds for that key dimension) will be proposed for the usage with the AES algorithm. This is done because 128-bit wide keys seem to provide a high level of protection, using less system resources than the ones used with wider keys. This solution also allow interoperability with all the systems which do not allow negotiation of the AES key-size.

For DES the key dimension cannot be negotiated.

When our system acts as initiator in phase 2, it will do the proposals reported in Table 3.2.

Mode	**Private key algorithm**	**Authentication algorithm**
Main	AES (128-bit key)	HMAC-SHA1
Main	AES (128-bit key)	HMAC-MD5
Main	AES (128-bit key)	none
Quick	AES (128-bit key)	HMAC-SHA1
Quick	AES (128-bit key)	HMAC-MD5
Quick	AES (128-bit key)	none
Main	3-DES	HMAC-SHA1
Main	3-DES	HMAC-MD5
Main	3-DES	none
Quick	3-DES	HMAC-SHA1
Quick	3-DES	HMAC-MD5
Quick	3-DES	none

Table 3.2: *IKE Phase 2* proposals for the “Road Warrior” network ordered by preference

3.4.3. SA Protocol selection

For the road warrior application the protocol that seems to be more suitable is ESP in tunnel mode; in that way both data and headers are protected and authenticated without using any SA bundle.

4. The smart card – IPSec software interface

The IPSec implementation will be based on a smart card that will provide the AES and the ECC cryptography algorithms. IPSec is the only part of the system that can directly communicate with the smart card, therefore a driver needs to be written. That driver will have to manage various functions such as the SA data swapping, as described in the security policy (see chapter 3). IPSec will also have to provide any way for the key exchange application (IKE[5]) to communicate with the smart card.

The smart-card/IPSec interface was written referring to the IPSec suite of protocols documents, the security policy and the system's structure. The interface specification contains all the commands that the smart card should provide to the system and the functionality that the system should support to allow the smart card to work.

Only a description of the software interface is given here. As stated before, the development of the hardware interface between the smart card and the host is beyond the scope of this document.

4.1. Commands

Here follows the explanation of each command that needs to be present in the software interface, basing on the requirement of the system. A command summary is provided in Table 4.1 and in Table 4.2.

4.1.1. IPSec to smart card commands

In this section all the commands that can be invoked by IPSec are described. A complete list of those commands is provided in Table 4.1.

[5] Here IKE is intended as a separate part with respect to IPSec, this comes from the fact that IKE operates at the application level, while the other two IPSec protocols (AH an ESP) operates at network level, as explained in chapter 1.

4.1.1.1. login

It allows using the smart card after a PIN code check. This command makes the smart card generate the AES session key (see section 3.3.2).

If the given PIN code is not correct, the smart card will wait for 5s before accepting a new attempt. After 3 consecutive login errors, the smart card stops working and needs an hardware reset (pull-out).

4.1.1.2. refreshSessionKey

This command is used to refresh the session key used for storing the SA data on the host. The SA information stored outside the smart card before that operation will not be accessible anymore.

4.1.1.3. resetSC

It deletes the smart card SA database and brings the smart card state back to the login state. The AES session key is deleted too.

After having issued this command, all the previously opened SAs cannot be anymore used.

4.1.1.4. readSCStatus

This command allows IPSec to read the smart card internal status registers for testing purposes. This command will need to be further developed when the smart card layout will be completed. Anyway this command will be carefully thought to avoid revealing the keys in any case.

4.1.1.5. setSAStatus

This command is used to upload the SA information in a smart card memory location specified by the *position* parameter. The information sent are composed by:

- the IV and the AES algorithm settings in clear form;
- the AES key and the CRC result encrypted with the AES session key (see section 3.3).

The encrypted data are decrypted in the smart card and the CRC is then checked. If any part of the given data is wrong (i.e. the computed CRC does not correspond to the stored one), the smart card will give back an error code (see subsection 4.1.4).

4.1.1.6. getSAParameters

This command makes the smart card return the SA information corresponding to the given key memory position (specified by the *position* parameter). The data format is the same as described in subsection 4.1.1.5. This command is mainly needed when it is necessary to store the key on the host for freeing a smart card memory position.

4.1.1.7. genDH

This command makes the smart card generate a random number and store it in the given key memory position (specified by the *position* parameter). The smart card returns the result of the operation given by *k*p* where *k* is the randomly generated number and *p* is the curve point. Before giving this command, is necessary to use the *setECCinfo* command or to verify that the set ECC parameters are the correct ones using the *getECCinfo* command (a long ECC curve's computation time can be avoided). This command can be used to generate the Diffie-Hellman payload to send to the other peer during the key exchange phase of a SA creation.

From the point of view of the parameters:

- The key size is coded as follows: 0 corresponds to a 128-bit AES key; 1 corresponds to a 192-bit key; 2 corresponds to a 256-bit key.
- The AES modes are coded as in the IPSec RFCs.
- The number of rounds must be less or equal than 14.

This command accomplish with the hypothesis reported in section 3.3.3.

4.1.1.8. completeDH

This command makes the smart card generate the AES key applying the Diffie-Hellman procedure on the KE-DH parameter (key=*k*h*p*) and the parameter previously stored in the given key memory position (specified by the *position* parameter); after that computation the specified key memory position is overwritten by the newly generated key. Before giving this command, is necessary to use the *setECCinfo* command or to verify that the set ECC parameters are the correct ones using the *getECCinfo* command (a long ECC curve's computation time can be avoided). This command is used to generate the AES key during the key exchange phase of a SA creation.

The key size is coded as follows: 0 corresponds to a 128-bit AES key; 1 corresponds to a 192-bit key; 2 corresponds to a 256-bit key.

This command accomplish with the hypothesis reported in section 3.3.3.

4.1.1.9. deleteSA

This command deletes the information that are in the given smart card memory position (specified by the *postion* parameter).

4.1.1.10. symmdecrypt

This command makes the smart card decrypt the given data using the AES algorithm with the parameters contained in the given memory position (specified by the *position* parameter). Before using this command the AES parameters have to be stored into the corresponding memory position using the *setSAStatus* command (for SA information stored on the host) or the *completeDH* command (for a newly created SA). A parameter allows to specify whether a hash checking has to be performed or not on the data.

4.1.1.11. symmEncrypt

This command makes the smart card encrypt the given data using the AES algorithm with the parameters contained in the given memory position (specified by the *position* parameter). Before using this command the AES parameters have to be stored into the corresponding memory position using the *setSAStatus* command (for host stored SA information) or the *completeDH* command (for a newly created SA). A parameter allows to specify whether the data have to be hashed or not with the algorithm specified when the SA was created.

4.1.1.12. setECCInfo

This command sets the ECC parameters to be used for the next ECC-block operation(s). The parameters are: the polynomial length (n) and 4 n-bit numbers (A and B that are the curve parameters; x and y that are the coordinates of the curve's base point). If the given parameters are the same of the previous invocation of the *setECCInfo* command, the smart card will not perform the ECC curve computation that would be otherwise needed.

4.1.1.13. getECCInfo

This command makes the smart card returning the ECC parameters currently used by the ECC block.

4.1.1.14. getPubKey

Request the smart card certificate (containing the smart card public key) to be sent to another peer.

4.1.1.15. publicEncrypt

Encrypt a data packet using the public key set using the *setECCInfo* command. This is useful for encrypting data to be sent to another peer (whose public key is the one that has been set) using the ECC algorithm.

4.1.1.16. publicDecrypt

Decrypt a data packet using the smart card’s private key. This is useful for decrypting data encrypted with the smart card public key using the ECC algorithm.

4.1.1.17. hash

Makes the smart card compute (and give) the specified pseudo-random function (that can be MD5 or SHA-1) for the given data. This command can be useful during IKE phase 1.

4.1.1.18. genSymmSign

Makes the smart card compute (and give) the specified pseudo-random function (that can be HMAC-MD5 or HMAC-SHA-1) for the given data. This is useful for symmetric-key-based authentications.

4.1.1.19. genECDSASignature

This service signs using ECDSA (Elliptic Curve Digital Signature Algorithm) the SHA-1 HASH output that needs to have been previously computed. This service is used during *IKE Phase 1 authenticated with signatures*.

4.1.1.20. verifyECDSASignature

This service verifies the signature according to the ECDSA algorithm. The decryption of the message *m* and the SHA-1 hash computation need to be performed in advance. This service is used during *IKE Phase 1 authenticated with signatures*.

4.1.2. Smart card to IPSec commands

In this section all the commands that IPSec provides for the smart card are described. A complete list of those commands is provided in Table 4.2.

4.1.2.1. error

This command is used to return an error message to IPSec. The error codes are described in section 4.1.4.

4.1.2.2. loginResults

Used to communicate the results of a user login to the smart card driver.

4.1.2.3. testResults

Used to communicate the results of the internal self-tests or the status of internal circuitry to the smart card driver.

4.1.2.4. SCStatus

Used to give the content of the internal Status Register after a *readSCStatus* request.

4.1.2.5. SAParameters

This command is used to give to IPSec the SA information previously requested through the *getSAInfo* command. The data format is the same as described in subsection 4.1.1.5.

4.1.2.6. randomDH

This command is used to get to IPSec the symmetric key generation payload previously requested via the *getSAKey* command. The given key is the one corresponding to the specified smart card position number and it is in clear form.

4.1.2.7. symmdecryptedP

This command gets the AES decrypted packet corresponding to a previous decryption request made using the *symmdecrypt* command. The decrypted packet is given with the smart card position number in which the corresponding SA information are stored.

4.1.2.8. symmEncryptedP

This command gets the AES encrypted packet corresponding to a previous encryption request made using the *symmdecrypt* command. The decrypted packet is given with the smart card position number in which the corresponding SA information are stored.

4.1.2.9. ECCInfo

This command gives the current ECC parameters previously requested with the *getECCInfo* command.

4.1.2.10. ECCKey

This command gives the smart card public key previously requested with the *getPubKey* command.

4.1.2.11. ECCEncrypted

This command gives the ECC encrypted packet corresponding to a previous encryption request made using the *publicEncrypt* command. The encryption is made using the other peer’s public key previously set with the *setECCInfo* command.

4.1.2.12. ECCDecrypted

This command gets the ECC decrypted packet corresponding to a previous encryption request made using the *publicDecrypt* command. The decryption is made using the private key stored in the smart card.

4.1.2.13. hashResults

Gives the result of the pseudo-random function applied to the data passed with the *PRD* command.

4.1.2.14. symmSign

Gives the result of the HMAC-MD5 or HMAC-SHA-1 function applied to the data passed with the *getSymmSign* command.

4.1.2.15. ECDSASignature

Gives the ECDSA signature obtained from the data previously sent through a *genECDSASignature* command.

4.1.2.16. ECDSACheckRes

Gives the result of the *verifyECDSASignature* command.

4.1.2.17. confirmation

Confirm a previously requested action. The returned code is the command code of the requested action. The commands which need confirmation are the following ones:

- refreshSessionKey
- resetSC
- setSAStatus
- completeDH
- deleteSA
- setECCInfo

4.1.3. Command table

In Table 4.1 and in Table 4.2 are defined the codes and the data exchanged using the appropriate interface between the two peers. The commands that can be issued by IPSec (described in subsection 4.1.1) are shown in Table 4.1, while in Table 4.2 are shown the command that can be issued by the smart card (described in subsection 4.1.2).

Task N.	Task Name	Data Exchange	Operation	Notes
1	login	PIN	s.c. login. After the validation a new AES session key is generated	without doing this operation, the s.c. will refuse every request
2	refreshSessionKey		makes the s.c. generate a new AES session key	old SAs information stored in IPSec will not be usable anymore
3	resetSC		resets the s.c.: all the information contained in the s.c. are deleted and a new AES session key is generated	
5	readSCStatus	Register number	makes the s.c. give the specified register status	
6	setSAStatus	position, SA_info	sets the AES parameters for the indicated SA	
7	getSAParameters	position	requests the "position" SA stored into the s.c.	the given key is encrypted with AES using a key generated at the s.c. startup
8	genDH	position, dim. of the symmetric key, number of rounds, mode	makes the s.c. generate the random number (k), store it into the SA space and compute k*p where p is an ECC curve's point	
9	completeDH	position, dim. of the symmetric key, KE-DH	completes the DH computation doing p*k*h	with this command the SA creation process is completed
10	deleteSA	position	deletes the SA information from the s.c. memory	
11	symmdecrypt	position, data length, packet of ecnrypted data, hash flag,IV flag	decrypts data. The length of the data packet is given by the data length parameter and every packet sent as data after this command has and ID number	first the SA parameters have to be set
12	symmEncrypt	position, data length, packet of data,hash flag, IV flag	encrypts data. The length of the data packet is given by the data length parameter and every packet sent as data after this command has and ID number	first the SA parameters have to be set
13	setECCInfo	n, A, B, x, y [, public_key]	sets the parameters needed by the ECC	n=polynomial length; A,B=curve parameters; x, y=curve's base point.
14	getECCInfo		requests the parameters that ECC is using when this command is given	
15	getPubKey		requests the s.c. public key	
16	publicEncrypt	packet of data	encrypts with public key algorithm using a specified key and ECC polynomial	a setECCinfo command must be given first
17	publicDecrypt	packet of ecnrypted data	decrypts with public key algorithm using the private key stored into the s.c.	a setECCinfo command must be given first
18	hash	data1, data2, prf_fcn	generates the specified pseudo-random function of the given data	prf_fcn can be MD5 or SH-1
19	genSymmSign	sa#, data1, data2, hash_fcn	generates a signature using a symmetric key	hash_fcn can be HMAC-MD5 or HMAC-SH-1
20	genECDSASignature	data	generates an ECDSA signature	
21	verifyECDSASignature	data	checks the ECDSA signature of the given data	

Table 4.1: table of the command provided by the smart cards for IPSec

Task N.	Task Name	Data Exchange	Operation	Notes
22	error	error code	communicates an error to IPSec	
23	loginResults	results	replies to a login request	subsequent to a login command
24	testResults	results	gives the result of the s.c. test	subsequent to a testSC command
25	SCStatus	register content	gives the actual content of the specified s.c. register	subsequent to a readSCStatus command
26	SAParameters	position, info	gives the information about a SA previously requested by IPSec	subsequent to a getSAParameters command
27	randomDH	position, k	gives the randomly generated number for the D-H key exchange	subsequent to a genDH command
28	symmdecryptedP	position, decr_packet	gives the decrypted packet relative to a SA	subsequent to a symmdecrypt command
29	symmEncryptedP	position, encr_packet	gives the encrypted packet relative to a SA	subsequent to symmEncrypt command
30	ECCInfo	n, A, B, x, y [, public_key]	gives the information about the currently used ECC parameters	subsequent to getECCInfo command. For the return parameters refer to the ones of getECCInfo
31	ECCKey	Key	gives the s.c. public key previously requested	subsequent to getPubKey command
32	ECCEncrypted	encr_packet	gives an ECC encrypted data packet	subsequent to publicEncrypt command
33	ECCDecrypted	decr_packet	gives an ECC decrypted data packet	subsequent to publicDecrypt command
34	hashResults	data	gives the hash function	subsequent to PRD command
35	symmSign	signature	gives the symmetric signature	subsequent to genSymmSign command
36	ECDSASignature	signature	gives the ECDSA signature	subsequent to genECDSASign command
37	ECDSACheckRes	results		subsequent to verifyECDSASignature command
38	confirmation	ofWhich	confirms a previous command.	subsequent to refreshSessionKey, resetSC, setSAStatus, completeDH, deleteSA or setECCInfo commands. ofWhich is the code of one of these commands

Table 4.2: table of the command provided by IPSec for the smart card

4.1.4. *Error codes*

In Table 4.3 a list of all the possible error code that the smart card can return and an explanation of these error codes are given.

number	error type	optional parameters	Description
1	generic error		
2	Bad SA index	SA position	given when a command requests an operation on a SA location that has not been initialized
4	wrong AES packet	SA position	given when a wrong encrypted packet is passed as *setSAStatus* command parameter
5	tampered SA information	SA position	given when a tampered SA information is passed as *setSAStatus* command parameter: this is an auditable event
6	wrong DH number		given when using a *completeDH* a wrong KE-DH parameter is passed
7	AES parameters not set	SA position	given when some parameters for the AES algorithm are missing for a requested SA
8	wrong ECC info		given when one or more of the *setECCInfo* parameters are mistaken
9	ECC info not set		given when a *publicEncrypt* or *publicDecrypt* command is given without having set the ECC parameters first
10	wrong PRD parameter		given when a mistaken parameter is passed using the *PRD* command
11	wrong symmSign parameter		given when a mistaken parameter is passed using the *getSymmSign* command
12	wrong signature control parameter		given when a mistaken parameter is passed using the *signatureControl* command
13	login already done		given when someone tries to login the s.c. while it has already been done
14	can't refresh the Session Key		given when for some causes (e.g. key in use by the AES module) the AES session key cannot be modified

Table 4.3: error codes

4.2. Software communication protocol

Looking at the command table, three different command formats can be identified. Each of these formats has a different length from the other ones, depending on the parameters which are needed. We name those three command formats *A*, *B*, and *C*. The format *A* is composed of only one word; the format *B* supports one word for the command identifier and *n* words of data. The format *C* is composed of a word for the command identifier, one word for additional parameters, and *n* words of data.

4.2.1. Command Format A (1 word)

The command format A is composed of 4 bytes only, in which the command identifier and the needed command parameters are included. See Table 4.10 and Table 4.11 for a detailed list of format A commands.

Command Identifier (1 word)

Bit number	Function
31-24	Command ID
23-0	Parameters

Table 4.4: *A* command format

4.2.2. Command Format B (1+n word)

The command format B has a variable size, depending on the size of data to be transmitted. The first 4 bytes of the command are used to specify some parameters and how many packets of data follow the header. See Table 4.10 and Table 4.11 for a detailed list of format B commands.

Command Identifier (1 word)

Bit number	Function
31-24	Command ID
23-0	Parameters

Table 4.5: *B* command format – header

Data (n words)

Bit number	Function
n*8-0	Data

Table 4.6: *B* command format – data

4.2.3. Command Format C (1+1+n words)

The command format C is very similar to the format B, since its size depends on the data to be transmitted. The only difference is that the C command format allows two different sets of data to be sent using the same command; the first data packet is of fixed length, wile the second one is of variable length. The first 4 bytes of the command are used to specify some parameters and how many packets of data follows the header. See Table 4.10 and Table 4.11 for a detailed list of format C commands.

Command Identifier (1 word)

Bit number	Function
31-24	Command ID
23-0	Parameters

Table 4.7: *C* command format – header

Parameters(1 word)

Bit number	Function
31-0	Data

Table 4.8: *C* command format – data, 1st part

Data (n words)

Bit number	Function
n*8-0	Data

Table 4.9: *C* command format – data, 2nd part

4.2.4. Command format tables

The format of the commands described in section 4.1.1 is shown in Table 4.10, while the format of the commands described in section 4.1.2 is shown in Table 4.11.

In both tables the symbol “$\lceil x \rceil$ “is used. That symbol stands for “the smallest integer greater then or equal to *x*” (ceil).

Task Name	Command Format	Parameters number	Command words	Data Exchanged
Login	A	1	1	command code (bits 31-24), PIN value (bits 23-0)
refreshSessionKey	A	-	1	command code (bits 31-24)
ResetSC	A	-	1	command code (bits 31-24)
TestSC	t.b.d.	t.b.d.	t.b.d.	to be defined
readSCStatus	A	1		command code (bits 31-24), register number
setSAStatus	B	1+1	1+n	command code (bits 31-24), SA index (bits 7-0), crypted_info
getSAParameters	A	1	1	command code (bits 31-24), SA index (bits 7-0, bit 7 is MSB)
GenDH	A	1	1	command code (bits 31-24), mode (bits 23-20), number of rounds (bits 19-16), key size (bits 15-8), SA index (bits 7-0) [see section 4.1.1.7 for details about the parameters]
completeDH	B	1+1	1+n	packet 1: command code (bits 31-24), key size (bits 15-8), SA index (bits 7-0); packet 2: KE-DH
DeleteSA	A	1	1	SA index (bits 7-0)

symmDecrypt	C	1+4+1	1+1+n	packet 1: command code (bits 31-24), SA index (bits 7-0);
				packet 2: hash flag (bit 24), sign. append (bit 23), IV flag (bit16), data length (bit 15-0);
				next packets: encrypted data

symmEncrypt	C	1+4+1	1+1+n	packet 1: command code (bits 31-24), SA index (bits 7-0); packet 2: hash flag (bit 24), sign. append (bit 23), IV flag (bit16), data length (bit 15-0) next packets: clear data
SetECCInfo	B	1+4 [5]	1+⌈n/32⌉*2+⌈2n/32⌉*2 [+⌈n/32⌉]	packet 1: command code (bits 31-24), curve length (n) (bits 23-8), data packet length (bits 7-0) next packets: A, B, x, y [, public_key]
GetECCInfo	A	-	1	command code (bits 31-24),
GetPubKey	A	-	1	command code (bits 31-24),
PublicEncrypt	B	1	1+n	packet 1: command code (bits 31-24), length (bits 7-0) next packets: clear data
PublicDecrypt	B	1	1+n	packet 1: command code (bits 31-24), length (bits 7-0) next packets: encrypted data
Hash	B	3+2	1+n	packet 1: command code (bits 31-24), length (bit 23-16), length (bit 15-8), hash func. (bits 3-0) next packets: data1, data2
HashInsertSimmKey	A	1	1	command code (bits 31-24), SA index (bits 7-0)
GenSymmSign	B	3+2	1+n	packet 1: command code (bits 31-24), length (bit 23-16), Signature func. (bits 11-8), SA index (bits 7-0) next packets: data1, data2
genECDSASignature	B	1+1	1+n	packet 1: command code (bits 31-24), length(7-0) next packets: data
verifyECDSASignature	B	1+1	1+n	packet 1: command code (bits 31-24), length(7-0) next packets: data

Table 4.10: format table of the “IPSec to smart card” commands

Task Name	Command Format	Parameters number	Command words	Data Exchanged
Error	A	1	1	command code (bits 31-24), SA index (bits 15-8), error code (bits 7-0)
LoginResults	A	3	1	command code (bits 31-24), login counter (bits 10-8); status code (bits 7-4); results (bit 0);
TestResults	t.b.d.	t.b.d.	t.b.d.	to be defined
SCStatus	A	1	1	command code (bits 31-24), status code (bits 17-0);
SAParameters	B	1+1	1+n	command code (bits 31-24), SA index (bits 7-0), info
RandomDH	B	1+1	1+n	command code (bits 31-24), SA index (bits 7-0), k
SymmDecryptedP	B	1+1 [2]	1+n [+m]	Packet 1: command code (bits 31-24), SA index (bits 7-0); next packets: decr_data[,sign.]
SymmEncryptedP	B	1+1 [2]	1+n [+m]	Packet 1: command code (bits 31-24), SA index (bits 7-0); next packets: encr_data [,sign.]
ECCInfo	B	1+4 [5]	$1+\lceil n/32\rceil *2+\lceil 2n/32\rceil *2$ [$+\lceil m/32\rceil$]	packet 1: command code (bits 31-24), curve length (n) (23-8 bits), data packet length (bits 7-0) next packets: A, B, x, y, public_key]
ECCKey	B	1	1+n	packet 1: command code (bits 31-24), key length (23-8 bits), packet length (bits 7-0) next packets: Key
ECCEncrypted	B	1	1+n	packet 1: command code (bits 31-24), length (bits 7-0) next packets: encr_packet
ECCDecrypted	B	1	1+n	packet 1: command code (bits 31-24), length (bits 7-0) next packets: decr_packet
HashResults	B	1	1+n	packet 1: command code (bits 31-24), length (bits 7-0) next packets: data
SymmSign	B	1	1+n	packet 1: command code (bits 31-24), length (bits 7-0) next packets: signature
ECDSASignature	B	1	1+n	packet 1: command code (bits 31-24), length (bits 7-0) next packets: signature
ECDSACheckRes	B	1	1+n	packet 1: command code (bits 31-24), length (bits 7-0) next packets: results
Confirmation	A	1	1	command code (bits 31-24), executed command code (bits 7-0);

Table 4.11: format table of the “smart card to IPSec” commands

4.3. The communication function

The command format has an intuitive structure for command passing between IPSec and the smart card. Two functions, one for each side of the communication channel between IPSec and the smart card, will be provided. Those functions will have two arguments each. The former argument is the length of the latter one. The second argument is an array of 32-bit integers carrying the codified command with its parameters. The length given in the first argument is represented as the minimum number of 32-bit integers which are needed for containing the data to be sent.

5. Writing the software interface C++ code

In this chapter we discuss the project specification and a implementation of the software interface previously described.

The implementation here proposed has to be taken as a reference implementation, that should be modified and optimized before being used in a real life system. It should be modified to take into account the necessary synchronization issues that can raise between the various elements of the system. This cannot be done here, being the synchronization techniques specific for every considered system.

The programming language chosen for this implementation is C++.

The main objective of what proposed here is to provide a more clear view of the software interface and how it should be used.

5.1. UML class diagram

Figure 5.1 shows the UML class diagram specifying our software system; the figure also shows the *SmartCard* and the *iKE* classes, which are not developed by us. *SmartCard* is the class containing the software model of the smart card (or the interface to the hardware smart card), while *iKE* is the class representing the implementation of the IKE protocol. Figure 5.2 shows the detailed description of the *sC_driver* class.

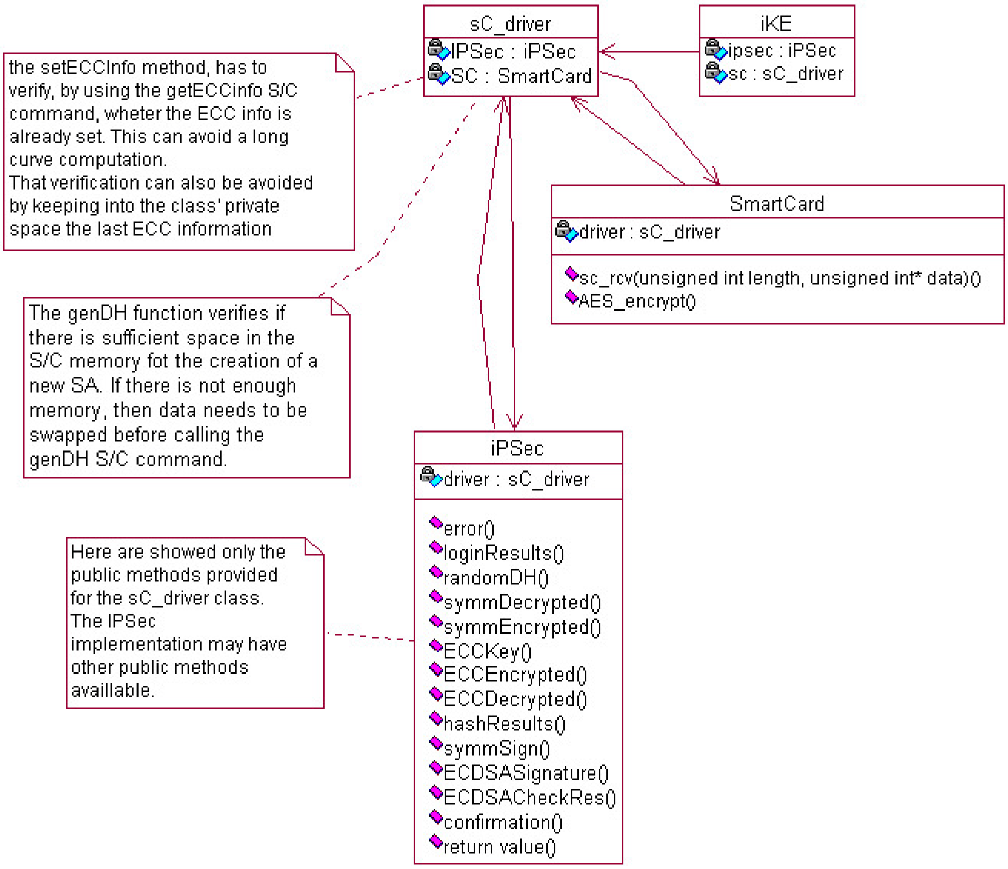
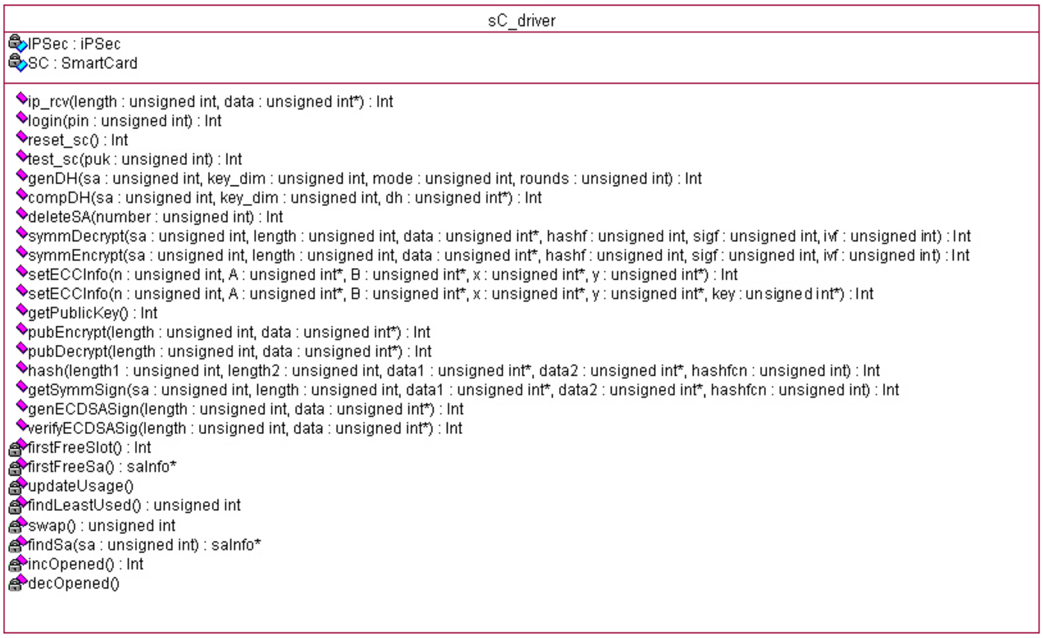

sC_driver
IPSec : iPSec
SC : SmartCard
iKE
ipsec : iPSec
sc : sC_driver
SmartCard
driver : sC_driver
sc_rcv(unsigned int length, unsigned int* data)()
AES_encrypt()
iPSec
driver : sC_driver
error()
loginResults()
randomDH()
symmDecrypted()
symmEncrypted()
ECCKey()
ECCEncrypted()
ECCDecrypted()
hashResults()
symmSign()
ECDSASignature()
ECDSACheckRes()
confirmation()
return value()
the setECCInfo method, has to verify, by using the getECCinfo S/C command, wheter the ECC info is already set. This can avoid a long curve computation. That verification can also be avoided by keeping into the class' private space the last ECC information
The genDH function verifies if there is sufficient space in the S/C memory fot the creation of a new SA. If there is not enough memory, then data needs to be swapped before calling the genDH S/C command.
Here are showed only the public methods provided for the sC_driver class. The IPSec implementation may have other public methods availlable.

Figure 5.1: UML class diagram of the system



sC_driver

IPSec : iPSec
SC : SmartCard

ip_rcv(length : unsigned int, data : unsigned int*) : Int
login(pin : unsigned int) : Int
reset_sc() : Int
test_sc(puk : unsigned int) : Int
genDH(sa : unsigned int, key_dim : unsigned int, mode : unsigned int, rounds : unsigned int) : Int
compDH(sa : unsigned int, key_dim : unsigned int, dh : unsigned int*) : Int
deleteSA(number : unsigned int) : Int
symmDecrypt(sa : unsigned int, length : unsigned int, data : unsigned int*, hashf : unsigned int, sigf : unsigned int, ivf : unsigned int) : Int
symmEncrypt(sa : unsigned int, length : unsigned int, data : unsigned int*, hashf : unsigned int, sigf : unsigned int, ivf : unsigned int) : Int
setECCInfo(n : unsigned int, A : unsigned int*, B : unsigned int*, x : unsigned int*, y : unsigned int*) : Int
setECCInfo(n : unsigned int, A : unsigned int*, B : unsigned int*, x : unsigned int*, y : unsigned int*, key : unsigned int*) : Int
getPublicKey() : Int
pubEncrypt(length : unsigned int, data : unsigned int*) : Int
pubDecrypt(length : unsigned int, data : unsigned int*) : Int
hash(length1 : unsigned int, length2 : unsigned int, data1 : unsigned int*, data2 : unsigned int*, hashfcn : unsigned int) : Int
getSymmSign(sa : unsigned int, length : unsigned int, data1 : unsigned int*, data2 : unsigned int*, hashfcn : unsigned int) : Int
genECDSASign(length : unsigned int, data : unsigned int*) : Int
verifyECDSASig(length : unsigned int, data : unsigned int*) : Int
firstFreeSlot() : Int
firstFreeSa() : saInfo*
updateUsage()
findLeastUsed() : unsigned int
swap() : unsigned int
findSa(sa : unsigned int) : saInfo*
incOpened() : Int
decOpened()

Figure 5.2: UML class diagram of the *sC_driver* class

Figure 5.3 shows the UML sequence diagram describing the typical behavior of the system. In that diagram all the operations related to a request of symmetric encryption are shown. A similar diagram can be drawn for all the other possible operations.

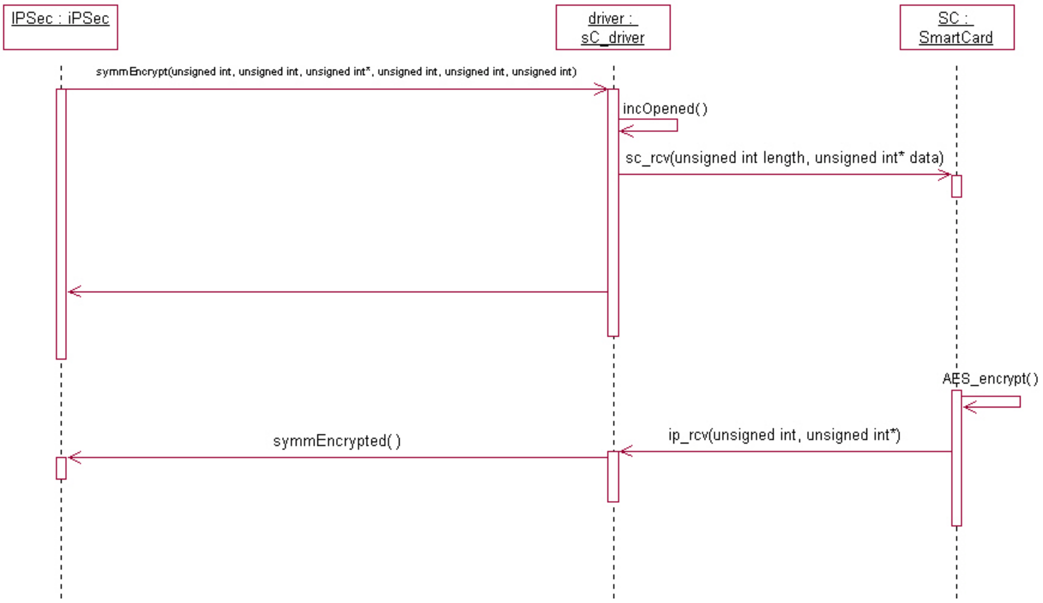

IPSec : iPSec
driver : sC_driver
SC : SmartCard
symmEncrypt(unsigned int, unsigned int, unsigned int*, unsigned int, unsigned int, unsigned int)
incOpened( )
sc_rcv(unsigned int length, unsigned int* data)
AES_encrypt( )
ip_rcv(unsigned int, unsigned int*)
symmEncrypted( )

Figure 5.3: UML sequence diagram of the system

5.2. Testing methods

Using an Object Oriented programming language, we choose to test our code by using the built in self test methodology ([FUG]): in each method of the written classes the required preconditions and the post-conditions will be checked through C++ assertions. This is used to ensure data consistency in the class, but a functional test of the system, as a class interoperability test is still needed.

5.3. The smart card software driver

The smart card driver represents a special part of the system.; it allows the IPSec implementation to communicate with the smart card and to hide certain smart card characteristics (e.g. the maximum number of available memory slots) to the IPSec implementation itself. The implementation issues related to this part of the system are described in the following paragraphs.

5.3.1. SA swap policy

In some cases, it will be necessary to swap out a SA from the smart card. We use the Least Recently Used (LRU) policy to select the SA to be stored outside the smart card. To implement that policy, it is necessary to keep track of the time when a SA was last used. This can be done using the counter technique ([TAN], pp.79-120): a variable for each SA stored in the smart card is updated every time one of the SAs is used. The update consists of decrementing the set of variables (one for each SA). Those variables are initialized at the maximum available value for the kind of chosen number representation (for example the maximum value available with a 32 bit-wide unsigned integer). Each time it is necessary to store a SA outside the smart card, the one with the lowest value of that variable is selected.

5.3.2. Synchronization issues

During the simulation of the system, it is necessary that only one smart card model is instantiated (we have to avoid that for every IPSec session created a smart card object is instantiated). This can be done by instantiating the driver during the IPSec initialization phase. Anyway, a check for the condition that only one smart card model has been instantiated, can be provided by a static counter, incremented and checked as a class constructor precondition at each driver instantiation.

The other thing we have to consider is that, if there are more than one IPSec sessions opened, only one of these can access the driver at a given time. For keeping under control this condition, we choose to use a counter. That counter will be checked at each driver's call done by an IPSec session. We have to note that the mechanism described before could be not sufficient in a real system and a more suitable technique should be implemented using the primitives offered by the considered operating system. Checking the counter as described before should be enough if the smart card driver is used coupled with a Linux IPSec implementation and compiled in the kernel (see [POM] in the "character device files" section). Those two conditions seem to be not too hard to respect being that all the known IPSec implementation for Linux run as kernel modules.

The smart card will manage the received commands using a FIFO queue, therefore we have to take into account that in some cases that queue can become full. For doing this we will use a counter of the free FIFO queue positions and we will return an error code to the IPSec calling function when that counter reach the 0 value. We can use a simple counter increment and decrement because, as stated before, only one driver instance can run at a given time.

Assuming that there will be only one smart card for every instance of the driver, the *ip_rcv* method implements no synchronization mechanism.

5.3.3. Data structures

Two different types of information have to be stored in the smart card driver:

- a map of the smart card memory, to know where a security association is allocated; we name this structure *"scSlot";*
- a list of the security associations that are using the smart card; we name this structure "*saInfo*".

The first structure is pretty small, since the smart card has a small amount of memory. The second one can be as big as we want, to store all the possible SAs that can be managed by the host.

For each element of the smart card memory we need to store two pieces of information: the number of the SA that is allocated in that slot and the usage counter as described in the paragraph above (Figure 5.4).

scSlot

```
saInfo* posArray;
unsigned int usage;
```

Figure 5.4: the *scSlot* structure

For each element of the SA list, we need to store three pieces of information: the SA number, the smart card memory cell where the SA is possibly allocated, and, optionally, the data swapped from the smart card for that SA (Figure 5.5). When a SA is not allocated to any smart card memory position, (i.e. that SA has been swapped out) we store in the corresponding field a position number obtained by the number of the last smart card memory slot plus one.

saInfo

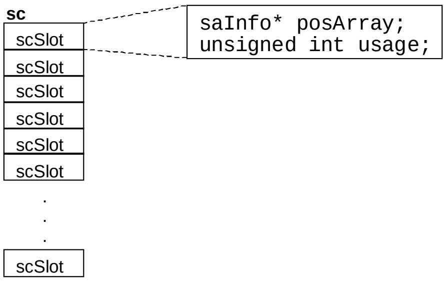
```
unsigned int sa;
unsigned int slot;
unsigned int* saved;
```

Figure 5.5: the *saInfo* structure

While it is clear that the structure described above can be organized in an array where each element is mapped on a smart card memory position, the same is not clear in the case of the SA list. Logically it would be better to organize the information in a dynamic structure like a list, but for speed purposes (we will need to search in that structure very often) using an array would a better choice. However an array does not provide the capability to grow as the requests of new SAs does. Therefore we choose to use an array of SA structures containing 40 times the number of available smart card memory slots. In that way we will be able to provide at most the equivalent of 40 opened SAs for each smart card memory slot. This could seem a limitation on the system’s capability, but we have to consider the fact that allowing too many opened SAs on the same smart card can force the system to loose more time swapping data between the smart card memory and the host than encrypting and decrypting data.

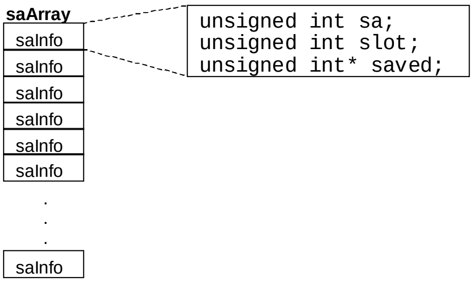
TO summarize, the two data structures considered are:

- *sc:* an array of *scSlot* structures (Figure 5.6); the length of this array corresponds to the number of memory slots available in the smart card (e.g. 16).
- *saArray:* an array of *saInfo* structures (Figure 5.7); the length of this array is 40 times the length of the sc array, (e.g. 16*40=640).



```
saInfo* posArray;
unsigned int usage;
```

Figure 5.6: the *sc* data structure



```
unsigned int sa;
unsigned int slot;
unsigned int* saved;
```

Figure 5.7: the *saArray* data structure

5.3.4. How the driver works

The smart card driver must perform two tasks :

- send and receive commands and data to the smart card
- swap SA data from the smart card.

In the first implementation of the driver, we will perform data swapping only when needed (e.g. when all the smart card memory slots are filled and there is the request of using a SA that is not in one of these slots), but a more efficient storage and swap policy can be implemented.

The driver receives the commands from the *iPSec* and from the *iKE* classes (through the *iPSec* one) and sends them to the smart card.

Here are the different kind of situations that the driver has to manage:

- When the driver receives the request to create a new SA it must first verify that there is enough space, both in the *saArray* array and in the smart card memory.

If there is no space in the *saArray*, the creation request must be refused. If the smart card memory is full, a swap will have to be performed. The smart card memory slot to swap must be chosen based on the swap policy previously described. After a free position is found or created through a swap, the driver can issue the command to create the new SA.

- When the driver receives a request to use the smart card with a specific SA (such as for the requests of data encryption or decryption), it must verify if the specified SA ha already been in a smart card memory slot; if not, it must retrieve the needed information from the host memory and put them in a smart card memory slot previously chosen. Once all these procedures are completed, the driver can send the desired command to the smart card.
- In all the other cases the driver must marshal the given command with its parameters and send the result to the smart card.

We have to remember that every time a SA is used the usage information have to be updated for each smart card memory slot.

All the commands are sent to the smart card or arrives from the smart card in the format described in the section 4.2, so that the data and the commands can be marshaled and put into an array of unsigned integers.

When the driver receives data from the smart card, it only has to pass them to IPSec by calling the suited function (chosen by looking at the command code).

The C++ code of this class can be found in Appendix A.

5.3.5. How IPSec should interact with the driver

When IPSec makes a request to the driver, it must take into account the fact that the driver itself may not be able to pass the given command to the smart card. This can happen when the command queue has became full. In that case an error code will be returned by the driver and IPSec will have to wait until a slot in the command queue will become free. For the first implementation of the driver, this should be done by retrying after a delay. Implementing a more suitable synchronization mechanism using the operating system primitives would be a good improvement for future releases.

We have to note that all these synchronization cannot be done in the driver because making the driver wait for some tasks would deny all the other processes to access the smart card.

6. The crypto-processor used on a router: study of the optimal SA cache dimension

In the first part of this document, we mostly referred to a crypto-system based on a smart card. In this section we will no more refer to a smart card, but to a very similar system built in a SoC[6]. Therefore here we will refer to a processor dedicated for cryptography, not considering anymore the limitations commonly considered for smart cards, such as the low power consumption. The core of our crypto-system can be therefore based on a relatively high performance embedded processor such as an ARM at 200MHz, while the main cryptographic functions can be implemented in hardware considering a higher clock frequencies than the one used in smart-card-like systems.

6.1. Reference system

In this part of the document we are referring to a system like the one shown in Figure 6.1. Here the IPSec host is coupled with a router; in that way all the communications between the machines belonging to the network in the dashed rectangle and the other ones, are protected through IPSec tunneling by the IPSec router. The machines in the left part of the scheme have to support IPSec for establishing secure connections with the machines protected by the IPSec-router. The IPSec-host has chosen to be protected by a firewall for diminishing the possibility of DoS attacks. Possibly the firewall should be coupled with an intrusion detection system (see section 1.5). Those choices about the network topology are anyway not influencing the results by the cache dimension study described in this chapter.

The additional requirements we are considering for the system are:

- Up to 200Mbit/s of throughput
- Up to 512 entries in the security processor’s cache

[6] System on a Chip.

All the other requirements, such the ones regarding the security of the keys are kept as stated in section 2.1.

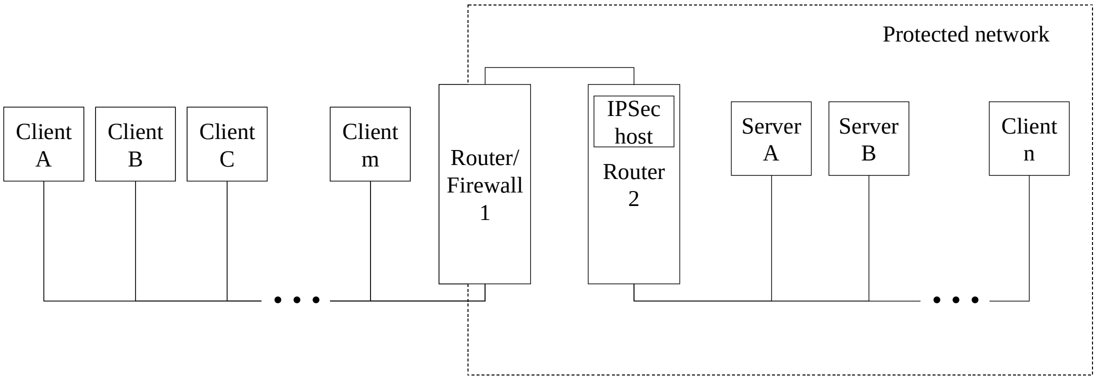


Figure 6.1: IPSec router reference system

As stated before, for such a system we can no more use a smart-card like security-processor, we prefer to consider a traditional integrated circuit, with the same structure and interface of the smart card we described in the previous sections of this document. When needed, we will refer to the data related to a standard 32-bit 66MHz PCI bus as hardware interface between the host and the crypto-processor, although a different interface could be chosen during the development of this project.

In this section we will refer to the various system's components as illustrated in Figure 6.2. The acronyms used in that figure are explained below:

- **SPD** is the Security Policy Database, containing all the information about the system's security policy
- **IP** is the usual IP layer, belonging to the usual protocol stack
- **SAD** is the Security Association Database, containing all the information about each opened SA (keys, settings, SA identifier, IV, …). It is really important to note that all the "sensible" information here are stored in encrypted form (AES, 256 bit-wide key) so that these information are readable only in the crypto-processor. See section 3.3 for further details.

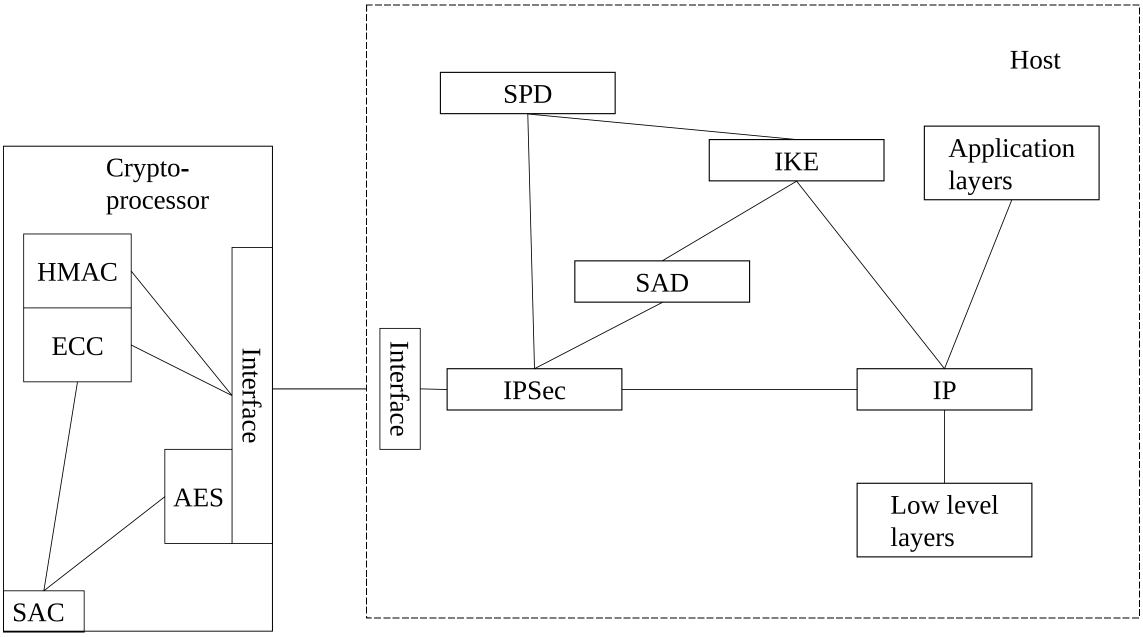


Figure 6.2: internal system representation

- **IPSec** is the block implementing all the IPSec communication functionality. For all the cryptography-related stuff it uses the Crypto-processor, through the *Interface* block.
- **IKE** is the block implementing the key exchange and the SA creation processes, basing on the data in the SPD. It uses the IP services and it fills the SAD fields once the SAs are created.
- The **Interface**:
    - on the host side, is the software interface between the IPSec software layer and the crypto-processor. It manages all the communications between these two blocks and it manages the SAC. As a matter of fact (see section 3.3 and chapter 4) the SAC is contained in the crypto-processor, but it is manager by the host Interface block.
    - On the crypto-processor side the interface manages the communications never giving out the keys in clear form.
- **Crypto-processor** is the macro-block implementing all the needed cryptographic algorithms (public and symmetric key encryption, authentication, D-H exchanges). This block contains:

- the **SAC,** that is the cached portion of the *SAD*.
- the **ECC** block, that implements in hardware or on software (see [CA-PO] for details) the ECC algorithm
- the **AES** block, that implements in hardware or in software (see [MAC-MAR] for details) the AES symmetric encryption algorithm
- the **HMAC** block, that implements the hashing algorithms

As we stated in the previous sections, the problem of dimensioning the SAC, can be seen as a cache dimensioning problem; for that cause, in the next parts of this document, all the occurrences of the word “cache” have the same meaning of the word “SAC”.

6.2. Simulation data

For all the simulations we used the data provided by the ITA site ([ITA-1], [ITA-2]). That data was obtained through *tcpdum*, a Unix tool for dumping the traffic of a system. The data was then modified by a script called *sanitize* for preserving the privacy of the people using that system. The typical data founded there contain in each line of the file a timestamp, the source and the destination IP addresses (modified for privacy), the source and the destination TCP ports, and the dimension of the datagram. The first few lines of the data file we used are shown in Figure 6.3.

```
0.010445 2 1 2436 23 2
0.023775 1 2 23 2436 2
0.026558 2 1 2436 23 1
0.029002 3 4 3930 119 42
0.032439 4 3 119 3930 15
0.049618 1 2 23 2436 1
0.052431 5 2 14037 23 2
0.056457 2 5 23 14037 2
0.057815 6 7 23 1502 414
0.072126 8 9 1023 513 0
```

Figure 6.3: first few lines of the data file

The data we considered represent the traffic between the Lawrence Berkley Laboratory and the rest of the world. That data are only about TCP traffic, but, considering this is the prevalent part of that system’s traffic, these data could be good for our simulations.

On the ITA site there are two tcpdump files taken from that system dumped in different day times and of a different length (the first is two hours long and the other is one hour long). Being the two dumps very similar from the point of view of throughput and number of datagrams managed per second, we chose to primarily use the longest one (around 1.8 million of rows, [ITA-1]). The throughput reached on that system is around 330kbit/s, so it is considerably lower than the maximum throughput desired for our system. This will be completely not influent when we will care about the *SAC* cache miss statistics, but it will be very important when we will care about the system timings. In that case we will need to scale all the timestamps for reaching the desired throughput of 200Mbit/s.

Unfortunately the data are about normal IP connections, so no information about the SA creation and closure are reported.

Another file (for each dump) containing the TCP SYN/FIN packets is also available.

6.3. Number of opened SA

Running a first program on the data file, we can obtain some graphs about the number of SAs needed at a given time for managing all the connections passing through the system. Having no information about the SA closure, we can try four different ways for keep the number of opened SAs under control during the simulations:

- no SA closure until they reach the maximum value for their sequence number (2^{32}) or the 8 hours limit;
- SAs are closed when unused for more than 30 minutes; this condition is checked every minute;
- when a TCP FIN packet sis received, the corresponding SA is closed;
- SAs are closed when unused for more than 30 minutes or when the system receives a TCP FIN packet.

The last three condition are here proposed for simulation purposes only, however the 30 minutes timeout can be evaluated as a condition to be applied in real systems too.

The first way is the one that would probably be used on a real system where the lifetime for each SA can be negotiated.

The number of SAs opened over time is shown in Figure 6.4. In that figure are considered the four cases previously illustrated: 30 min. timeout and not considering the FIN packets, 30min. timeout also considering the FIN packets, considering the FIN packets without any timeout, without any timeout and without considering the FIN packets. The simulation program used for obtaining that results is reported in Appendix B and it is explained in section 6.4.2.

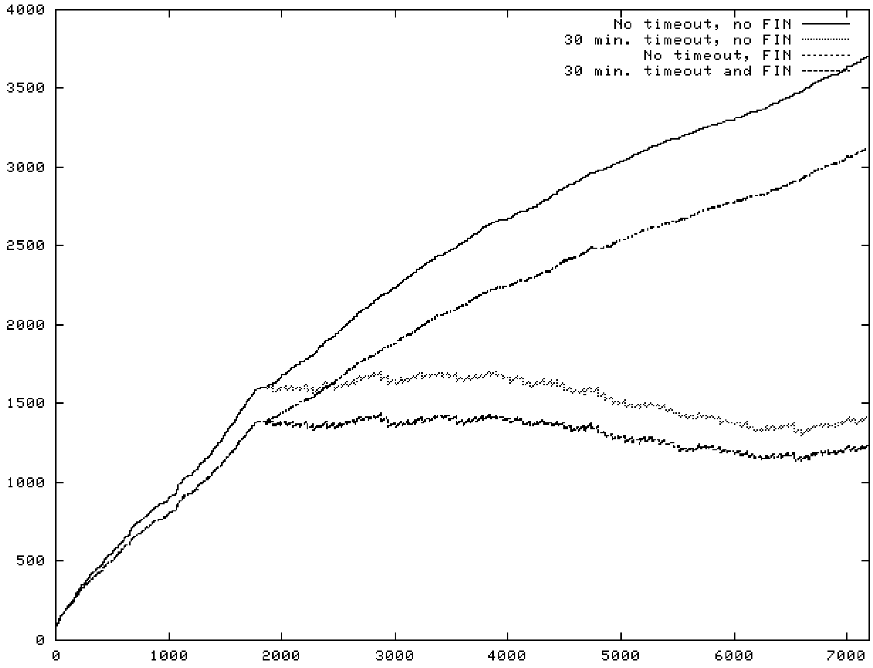


Figure 6.4: number of opened SAs over time

As can be easily noted, the number of SAs continues to grow if no closing policy is implemented or when only the TCP FIN packets are considered. The behavior of a real IPSec system should be slightly different, due to the possibility of setting an expiration time for each SA (anyway shorter than 8 hours). We stated that the behavior should be "slightly different" because specifying the expiration time is not mandatory. Moreover, when one of the two parties closes a SA, it can or cannot inform the other (see for example [FSWAN]). In the latter case, there can be anyway a lot of SAs opened and unused.

For understanding the behavior of the system in the four cases explained above, we can also look at the graphs representing the distribution of the creation of new SAs over 1 second intervals. These distribution are shown in Figure 6.5 (no SA closing policy adopted), Figure 6.6 (SAs are closed after being unused for more than 30min.), Figure 6.7 (SA closed when a TCP FIN packet is received), Figure 6.8 (SAs are closed after being unused for more than 30min. or when a TCP FIN packet is received).

Not considering the initial phase, where a high number of new SAs is created (around 35 in the first second), we can note that the creation of new SAs is a process pretty well distributed over the whole simulation time.

Using the TCP FIN packets for closing the SAs, causes more SAs to be newly opened in each second (because of SAs that need to be re-opened) and does not really limits the number of SA opened at a given time (see Figure 6.4). Those considerations are confirmed looking at the graphs representing how many times a SA is used before being closed. Those graphs are shown in Figure 6.9 (no timeout, not considering the TCP FIN packets), Figure 6.10 (30 min. timeout, not considering the FIN packets), Figure 6.11 (no timeout, using the FIN packets), and Figure 6.12 (30 min. timeout, using the FIN packets). In those graphs each SA is represented by a number on the abscissas axe.

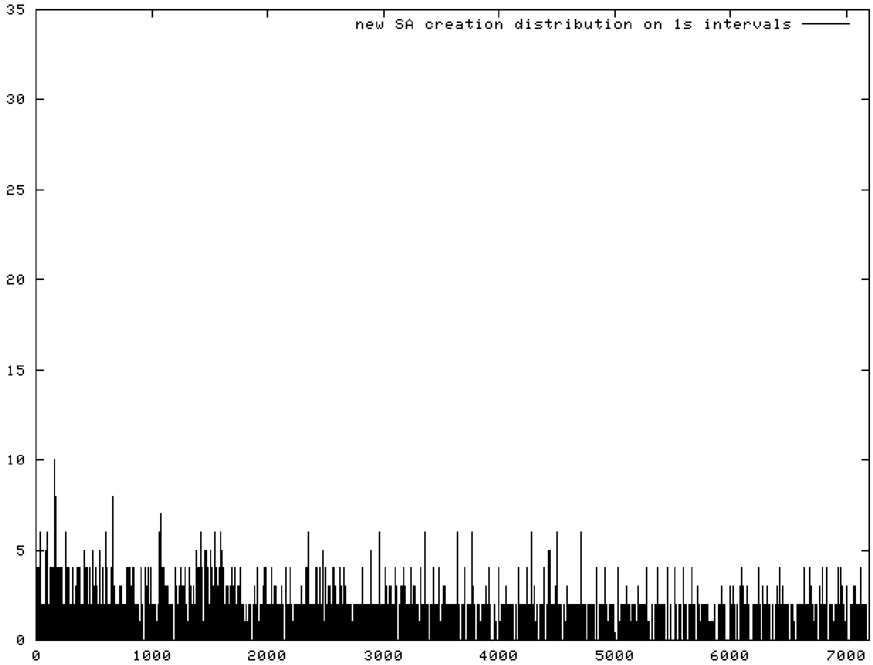


Figure 6.5: SA creation distribution over 1s intervals

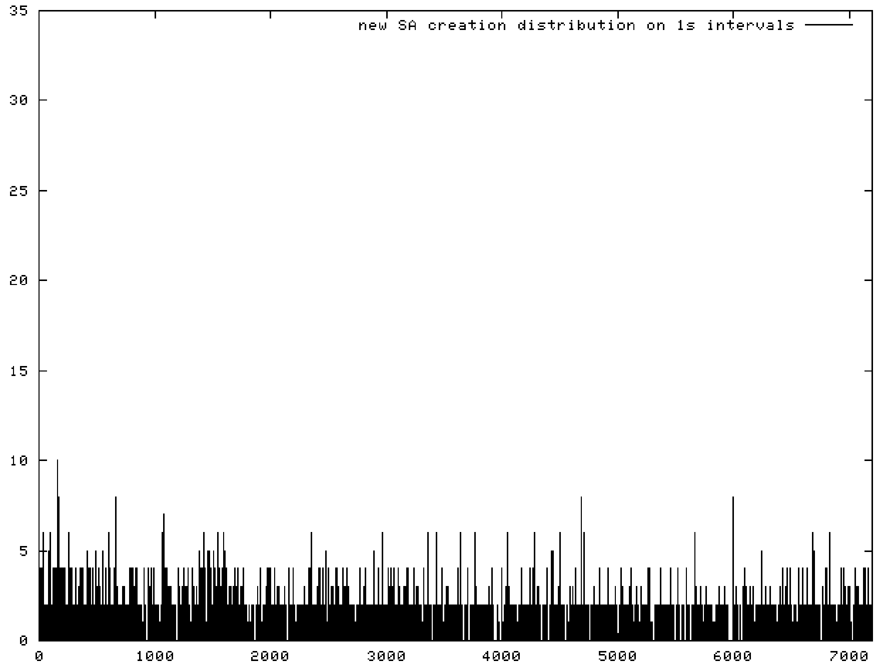


Figure 6.6: SA creation distribution over 1s intervals when a 30min. timeout is set on unused SAs

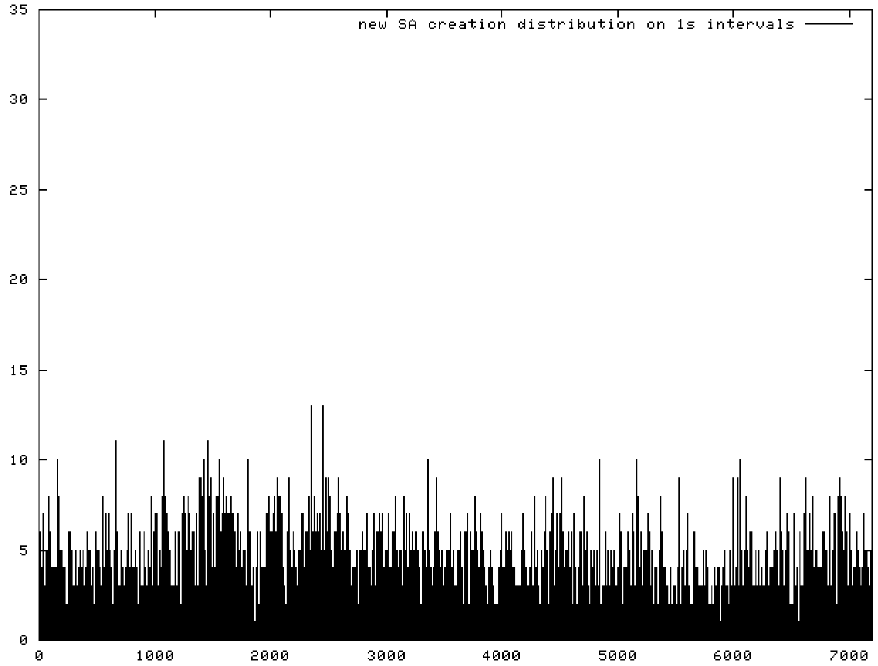


Figure 6.7: SA creation distribution over 1s intervals when the TCP FIN packets are used for closing the SAs

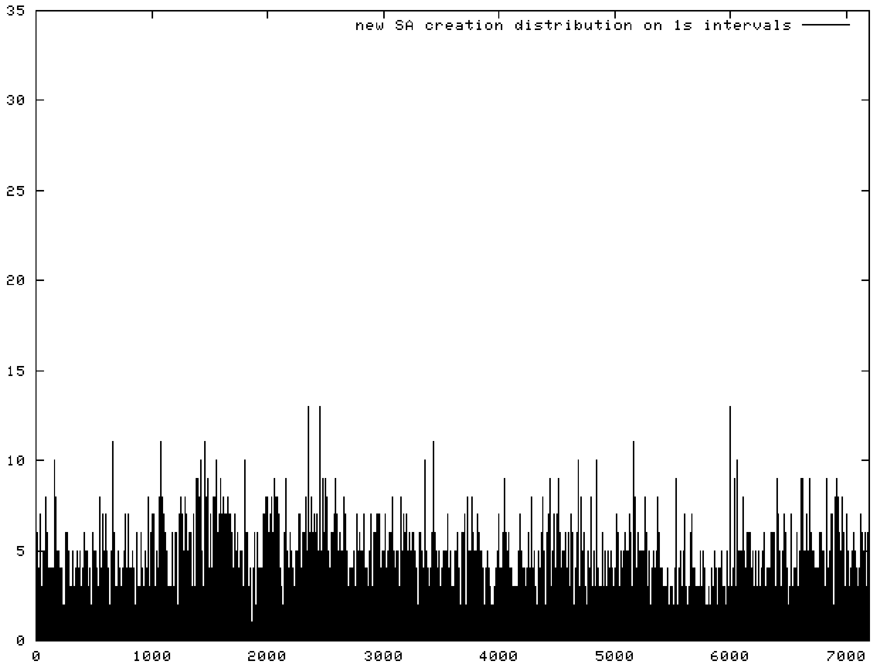


Figure 6.8: SA creation distribution over 1s intervals when a 30min. timeout on the unused SAs is set and the SA TCP FIN packets are used for closing the SAs

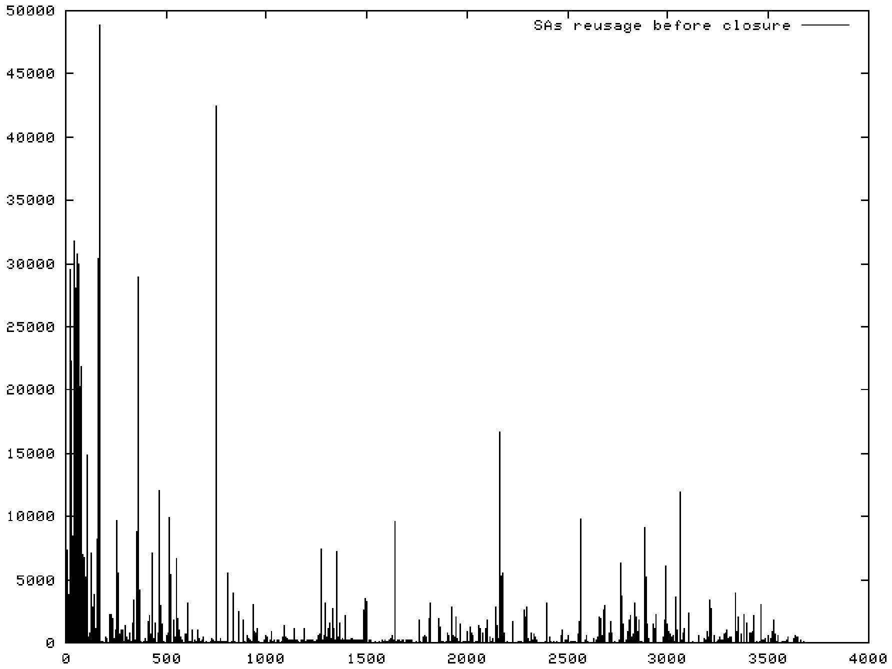


Figure 6.9: reuse of the SAs before being closed

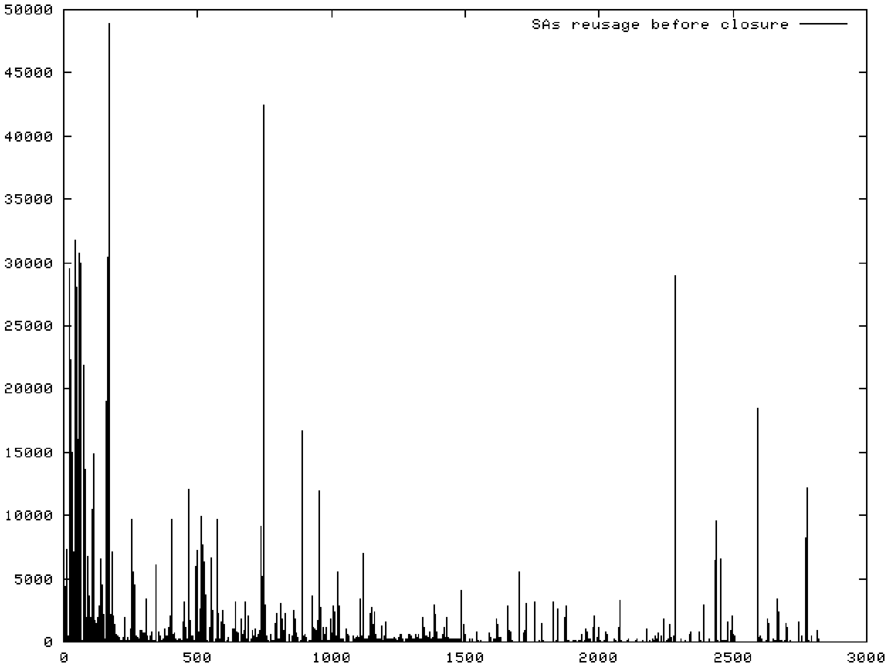


Figure 6.10: reuse of the SAs before being closed when a 30min timeout on the unused SAs is set

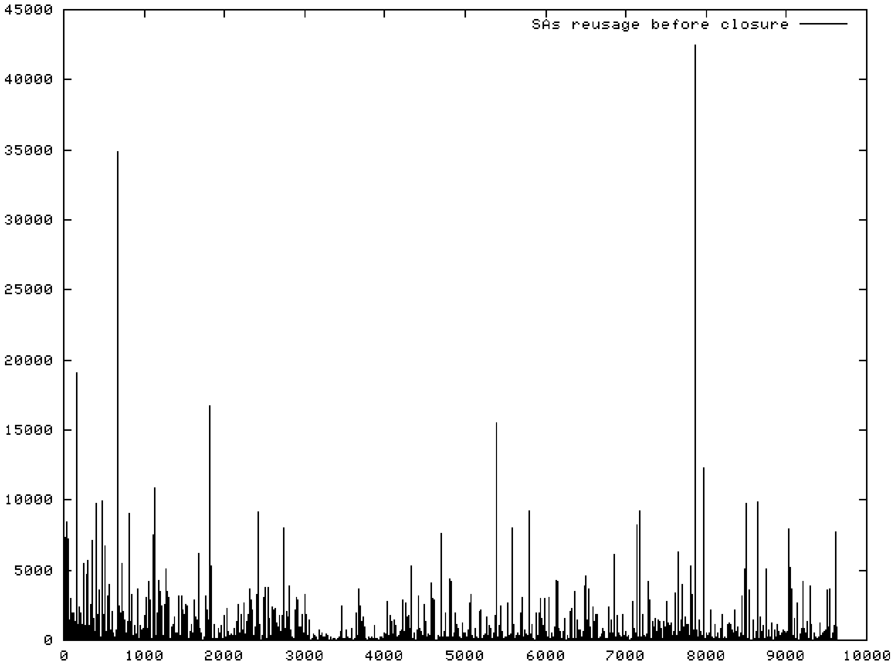


Figure 6.11: reuse of the SAs before being closed using the TCP FIN packets for closing the SAs

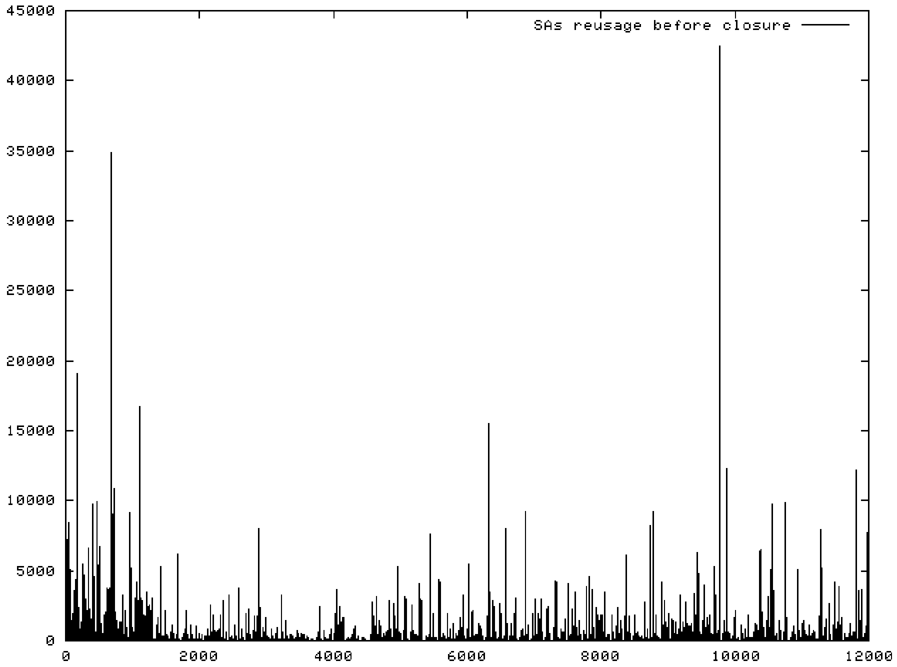


Figure 6.12: reuse of the SAs before being closed when a 30min timeout on the unused SAs is set and using the TCP FIN packets for closing the SAs

As can be noted looking at those graphs and at the results obtained from the simulations (see section 6.4.4), the highest SA reuse is obtained when no SA closing policy is adopted (average reuse of 483 times); slightly different results are obtained using the 30 minutes timeout (average reuse of 421). A consistent worsening of the SA reuse is obtained using the TCP FIN packets for closing the SAs (average reuse of 140 when using that policy alone and of 135 when combined with the 30 min. SA timeout). The different number of SAs shown in each of the four graphs is due to the fact that when using the TCP FIN packets or the 30 minutes timeout for the SAs closure, some SAs are later reopened with the same source and destination IP address (please note that the SAs are correctly considered different, since they are negotiated at different times, with different keys and, possibly, with different parameters).

The SA negotiation is here considered to be done only when needed, while applying “IKE Phase 2 quick mode” negotiations, two SAs are opened at the same time, one for each direction of the communication. See section 6.4.8 for further details about this topic.

As stated before, on a real system the policy to be adopted would probably be the one considering a timeout on the unused SAs. That timeout should be chosen basing on a profiling of the real system's traffic and used in conjunction with the SAs lifetime data. The better solution would be to check if the SA chosen to be closed due to the expired timeout is or is not considered alive by the other peer. In our simulation we will anyway consider all the four described cases, using an upper bound for the dimension of the SAD of 4,000 records. In that way the SAD is large enough for not limiting the number of opened SAs at any time during the simulations.

6.4. Cache dimension study without considering the crypto-processor delays

In this section we illustrate the simulation we wrote and we ran to find the best dimension to be adopted for the SAC. None of the delays introduced by the crypto-system are here considered.

A LRU replacing policy ([P-H], pp. 380-402) is adopted for the SAC elements. For the same cache, we choose to use a completely associative structure ([P-H] , pp. 380-402).

6.4.1. Space needed for the SAC

In the crypto processor only few information are really needed for processing the data related to each SA. Studying the SAD fields that we need to also keep in the SAC, and their dimension, we are able to compute how many space the SAC require on the crypto-processor chip. The computation is done considering the worst possible case.

Considering an ISAKMP SA, the required fields are shown in Table 6.1.

Field	***Length (worst case)***
AES key	256 bits
AES encryption algorithm, IV mode	8 bits
symmetric key size, number of rounds, authentication algorithm	8 bits
IV	128 bits
Total	400 bits = **50 bytes**

Table 6.1: memory space needed in the SAC for each ISAKMP SA

While considering an IPSec SA we obtain the result shown in Table 6.2.

Field	***Length (worst case)***
AH/ESP – key size, authentication algorithm	8 bits
AH/ESP – symmetric key	256 bits
ESP encryption algorithm, number of rounds, IV mode	8 bits
IV	128 bits
Total	400 bits = **50 bytes**

Table 6.2: memory space needed in the SAC for each IPSec SA

We can note that in both cases we need 50 bytes for each SA to be stored in the SAC. We choose to use, mainly for expansibility purposes, a dimension of 64 bytes for each SAC entry.

We can now compute the SAC dimensions related to the different numbers of SAC-entries chosen, as shown in Table 6.3.

Number of entries	***Dimension (bytes)***
16	1024 (1kb)
32	2048 (2kb)
64	4096 (4kb)
128	8192 (8kb)
256	16384 (16kb)
512	32768 (32kb)

Table 6.3: space needed for the SAC depending on the number of entry chosen

6.4.2. Designing the simulation

The simulation we need to write in this phase has only to keep track of what happens in the cache, when the data read from the file are used as input for the system. This means that no timings need to be taken into account. Looking at the source and destination IP addresses read from the data file, the program must be able to determine if the considered SA is already in cache or not, and to put it in the SAC when necessary. Doing those operations the statistics about cache misses and cache distribution over time can be computed.

6.4.3. The simulation program

The simulation is written in C and it works in the following way:

- It reads one line from the TCP dump file.
- It checks if a SA for the source and the destination IP addresses which has been read from the file has already been opened. If not, it opens a new one. If that SA has already been opened, it updates the SA usage counter and the timestamp of the last usage of that SA. The timestamps used here are the ones taken from the data file.
- It checks if the SA is already in the SAC, if not two different things can happen:
    - there is a free entry in the SAC. In that case the free entry is used for loading the information related to the considered SA.
    - there is not a free entry in the SAC. In that case the least recently used SAC entry is found and stored out of the SAC. The information related to the considered SA are then loaded in the freed SAC position.
- During all the operations the appropriate counters (cache misses, SA number,…) are kept up to date.

The program source code can be found in Appendix B.

6.4.3.1. Data structure used

The main data structures used are two, one for the SAD and one for the SAC.

Those two data structures are simply array of records. We choose to manage the data structures in the simplest possible way, so we choose to use unsorted arrays accessed in a sequential way. Obviously, for a real system better data structures and access methods should be studied, but discussing those topics is beyond the scope of this document.

As stated before, the data structures are based on records (C structures), the first one is *SADel* which is designed to contain all the information related to a SA (the one we need in our simulation), while the other is *SACel* that is designed to contain all the information related to each cached element.

The SADel and the SACel structures are shown in Figure 6.13 and in Figure 6.14.

The SADel structure contains the following fields:

- *sourceIP:* an integer field used for storing the source IP address (as explained in section 6.2, the IP addresses were modified for privacy concerns);
- *destIP:* an integer field used for storing the destination IP address;
- *cached:* the position in the *SAC* where the current element of *SAD* is possibly stored;
- *counter:* the SA usage counter. Once this field reach its maximum value (2^{32}), the SA have to be closed as explained in section 1.4.3;
- *time:* the timestamp of the SA last usage.

SADel

```
int       sourceIP;
int       destIP;
int       cached;
unsigned  counter;
double    time;
```

Figure 6.13:*SADel* structure

SACel

```
int       sourceIP;
int       destIP;
double    time;
long      countUsed;
```

Figure 6.14: *SACel* structure

The *SACel* structure is composed by the following fields:

- *sourceIP:* an integer field used for storing the source IP address, as in *SADel*;
- *destIP:* an integer field used for storing the destination IP address, as in *SADel*;
- *time:* the timestamp of the last usage of the cache position;
- *countUsed:* a counter for computing the statistics on each cache entry reuse.

As wrote before, the two data structure used in the simulation are array composed by the two C structures shown above. The SAD elements are contained in the *SAD* array defined in Figure 6.15 below and shown in Figure 6.17.

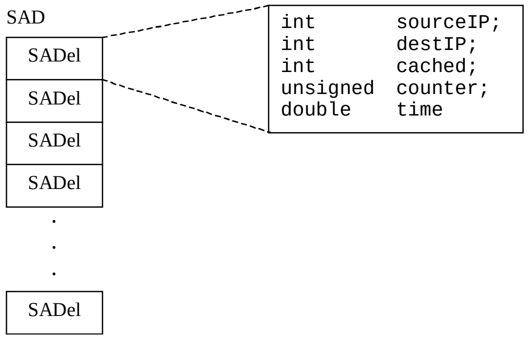
```
struct SADel SAD[MAX_SA];
```

Figure 6.15: *SAD* array definition

The SAC entries are contained in the *SAC* array defined in Figure 6.16 and shown in Figure 6.18.

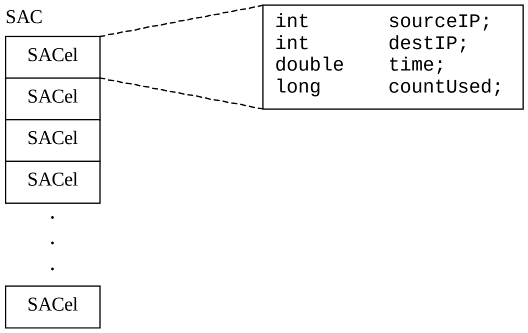
```
struct SACel SAC[CACHE_SIZE];
```

Figure 6.16: *SAC* array definition



```
int       sourceIP;
int       destIP;
int       cached;
unsigned  counter;
double    time
```

Figure 6.17: the *SAD* array



```
int       sourceIP;
int       destIP;
double    time;
long      countUsed;
```

Figure 6.18: the *SAD* array

For the meaning of the *CACHE_SIZE* and the *MAX_SA* constants, please refer to section 6.4.3.2.

A temporary variable called *datagram* is used in each program cycle. That variable is based on a data structure called *dataT* and shown in Figure 6.19. That variable is used to store the values read from the data file.

dataT

```
double time;
int sourceIP;
int destIP;
int sourceTCP;
int destTCP;
int bytes;
```

Figure 6.19: *dataT* structure

The fields of the *dataT* structure are explained below :

- *time:* is the field used to store the datagram’s timestamp;
- *sourceIP:* is the field used to store the datagram’s source IP address as explained for the *SACel* and the *SADel* structures;
- *destIP:* is the field used to store the datagram’s destination IP address;

- *sourceTCP:* is the field used to store the originating TCP port; this field is unused in the program;
- *destTCP:* is the field used to store the destination TCP port; this field is unused in the program;
- *bytes:* this field stores the number of bytes of each datagram. At each value read from the file we decided to add 34, that is the number obtained by the length of the IP v.4 headers (20 bytes) and the length of the TCP headers (14 bytes).

6.4.3.2. Simulation options

The options related to each simulation are given through C constants (C *define* directive), the available options are:

- *CACHE_DIMENSION:* defines the cache dimension (number of entries).
- *MAX_SA:* defines the maximum number of SAs allowed on the host.
- *USE_FIN_PACKETS:* defines whether or not to use the TCP FIN packets. The FIN packets are used to close the SAs when that constant is defined.
- *CLOSE_UNUSED:* defines whether or not to use the timeout on the opened SAs. That option is enabled when this constant is defined.
- *CLOSE_TIME:* defines the SA timeout to use (when the *CLOSE_UNUSED* option is active).
- *CHECK_TIME:* defines the checking time for SAs exceeding the timeout (when the *CLOSE_UNUSED* option is active).
- *PRINT_INSTANT_STATISTICS:* when defined it makes the program print the number of opened SAs and the number of cache misses on the standard output.
- *PRINT_CACHE_DISTRIB:* when defined it makes the program print the distribution of the cache misses over 1s intervals.
- *PRINT_SA_DISTRIB:* when defined it makes the program print the distribution of the opened and closed SAs over 1s intervals.
- *PRINT_CACHE_REUSE:* when defined it makes the program print the number of reuse for each SAC element before its closure.

- *PRINT_SA_REUSE:* when defined it makes the program print the number of reuse for each SA before its closure.
- *QUICK_MODE:* when defined it makes the program to threat the SAs as created with *IKE phase 2 – quick mode* (two SAs are created each time a new SA is needed for having a bi-directional communication channel).

6.4.4. The results of the simulations

In this paragraph the results obtained running the simulations varying the cache dimension are reported. Those results are presented through graphs representing the cache misses over time, and through the output provided by the program at the end of each simulation run. The numerical output of the program provides the following information:

- *Total number of datagrams analyzed:* the total number of datagrams taken from the data file. This parameter is constant for all the simulation runs being that we use the same data file for all of them.
- *Average dimension of datagrams:* this is the average dimension in byte of the analyzed datagrams. The dimension of each datagram is obtained adding 34 to the packet's dimension read from the data file. Those 34 bytes are given by 20 bytes of IP headers and 14 bytes of TCP headers. As a matter of fact the dimension of the headers are not reported in the data file.
- *Average data rate:* this is the average throughput, obtained dividing the sum of the dimension of all the datagram passed through the system, by the ending time of the simulation.
- *Average connections managed per second:* this parameters reports how many IP datagrams are managed in each second by the system and it is obtained dividing the number of datagrams passed through the system by the total simulation time.
- *Average reuse of each SA:* this gives the average number of time that each SA is reused during its lifetime.

- *Total cache misses:* this gives the total number of cache misses happened during the simulation run
- *Compulsory misses:* this gives the number of unavoidable cache misses happened during the simulation run. The unavoidable cache misses are the ones happening the first time the SAs are used.
- *Avoidable cache misses:* this gives the total number of avoidable misses happened during the whole simulation. With “avoidable misses” we mean the misses that can be avoided having a cache big enough for containing all the SAD entries. Those are the misses happening for each SA after it has been used for the first time.
- *Average reuse of each cache position before replacing:* this gives the average number of times that each cache entry is reused before being discarded.

In the following subsections the obtained results are shown.

6.4.4.1. Not closing any SA

In this case we report the results of the simulations ran considering 16, 32, 64, 128, 256, and 512-entry caches. In Figure 6.20, Figure 6.21, Figure 6.22, and Figure 6.23 are represented the results obtained with the last four cache dimensions as total cache misses over time, compulsory cache misses over time, and avoidable (total minus compulsory) cache misses over time. The other curve shown in each graph represents the number of SAs opened over time.

Please note that in all those four figures, the curve representing the number of SAs overlaps the one representing the number of compulsory cache misses.

Since the first part of the numerical results obtained here (the data rate, the SA reuse, the number of analyzed datagrams, and the number of connections managed per second) is always the same for each simulation run performed in this subsection, we will report it only in the first case shown.

Here follow the numerical results obtained for a 16-enty cache:

```
Total number of datagrams analyzed 1789994
Average dimension of datagrams (34bytes of header): 170.36bytes
Average data rate: 330.876kbit/s
Average connections managed per second: 248.61
Average reuse of each SA: 483.13

Total cache misses: 798593 (44.61%)
Compulsory misses: 3705
Avoidable cache misses: 794888 (44.41%)
Average reuse of each cache position before replacing 2.24
```

Here follow the numerical results obtained for a 32-enty cache:

```
Total cache misses: 382498 (21.37%)
Compulsory misses: 3705
Avoidable cache misses: 378793 (21.16%)
Average reuse of each cache position before replacing 4.68
```

In Figure 6.20 a graphic representation of the results obtained for a 64-entry cache is shown. The numerical results are shown below:

```
Total cache misses: 124369 (6.95%)
Compulsory misses: 3705
Avoidable cache misses: 120664 (6.74%)
Average reuse of each cache position before replacing 14.39
```

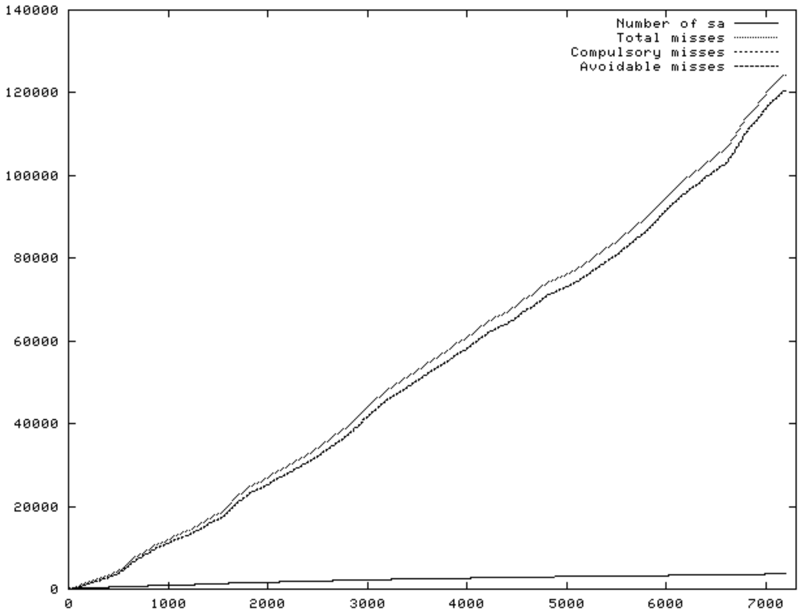


Figure 6.20: simulation results over time with a 64-entry cache

In Figure 6.21 a graphic representation of the results obtained for a 128-entry cache is shown. The numerical results are shown below:

```
Total cache misses: 25122 (1.40%)
Compulsory misses: 3705
Avoidable cache misses: 21417 (1.20%)
Average reuse of each cache position before replacing 71.25
```

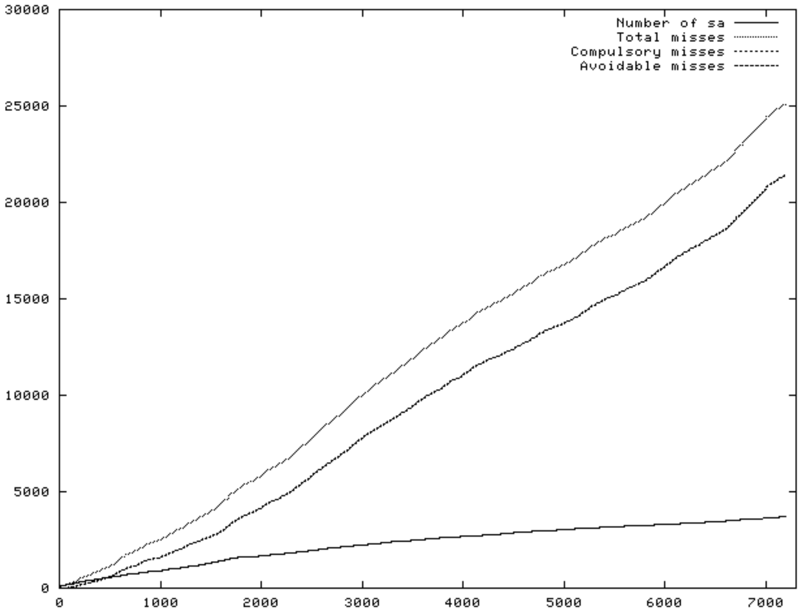


Figure 6.21: simulation results over time with a 128-entry cache

In Figure 6.22 a graphic representation of the results obtained for a 256-entry cache is shown. The numerical results are shown below:

```
Total cache misses: 11892 (0.66%)
Compulsory misses: 3705
Avoidable cache misses: 8187 (0.46%)
Average reuse of each cache position before replacing 150.52
```

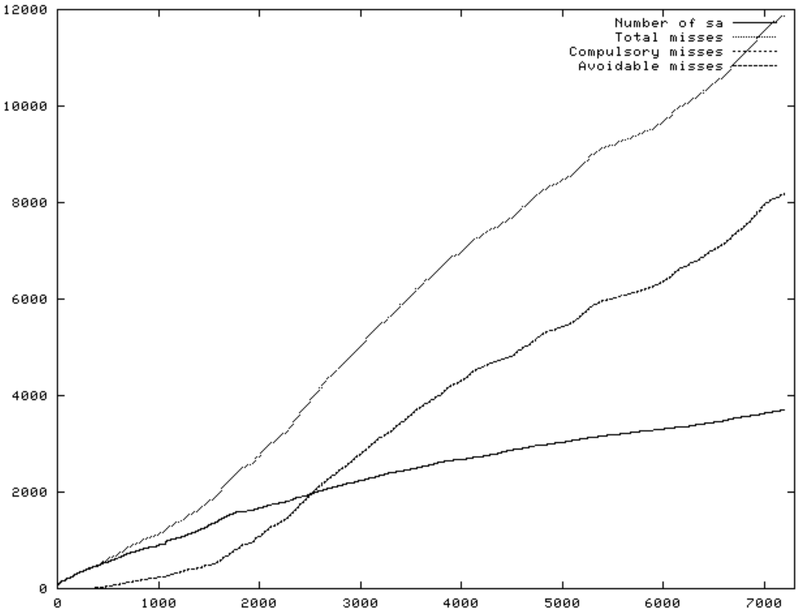


Figure 6.22: simulation results over time with a 256-entry cache

In Figure 6.23 a graphic representation of the results obtained for a 512-entry cache is shown. The numerical results are shown below:

```
Total cache misses: 8056 (0.45%)
Compulsory misses: 3705
```

Avoidable cache misses: 4351 (0.24%)
Average reuse of each cache position before replacing 222.19

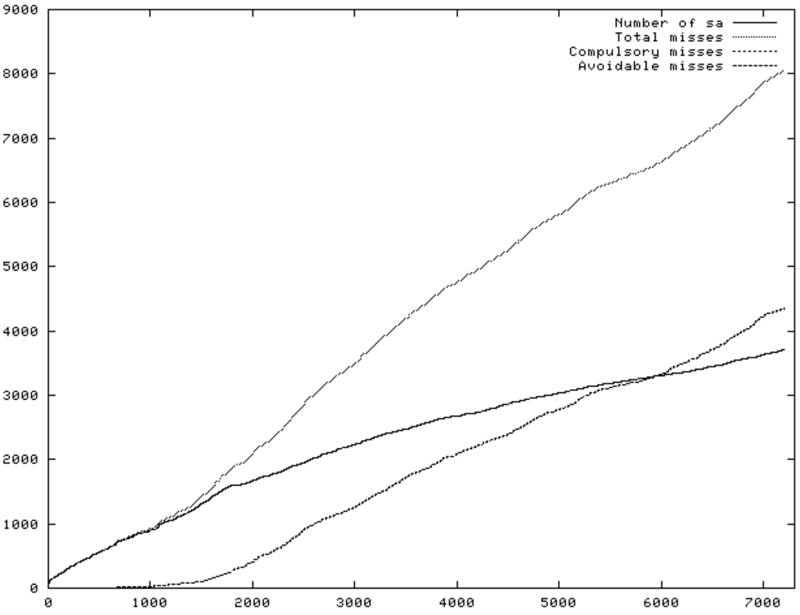


Figure 6.23: simulation results over time with a 512-entry cache

The obtained results show us that 16-entry or 32-entry caches are fairly not useful causing a too high number of avoidable cache misses. The differences between a 64-entry cache and a 128-entry cache are quite high (the number of avoidable cache misses is divided by 5.6). Less differences can be obtained using a 256-entry cache: with respect to a 128-entry cache the number of avoidable cache misses is divided by 2.6. It seems that using a 512-entry cache gives very few benefits (with respect to a 256-entry cache) from the point of view of avoidable cache misses.

6.4.4.2. Closing the SAs exceeding a 30 min. timeout

In this case, as in the following ones, we report the results of the simulations done with 64, 128, 256 entry caches only, being those the more significant ones. In Figure 6.24, Figure 6.25, and Figure 6.26 are shown the results obtained with those three cache dimensions as total cache misses over time, compulsory cache misses over time, and avoidable (total-compulsory) cache misses over time. The other curve shown in each graph represents the number of SAs opened over time.

Since the first part of the numerical results obtained here (the data rate, the SA reuse, the number of analyzed datagrams, and the number of connections managed per second) is always the same for each simulation run performed in this subsection, we will report it only in the first case shown.

In Figure 6.24 a graphic representation of the results obtained for a 64-entry cache is shown. The numerical results are shown below:

```
Total number of datagrams analyzed 1789994
Average dimension of datagrams (34bytes of header): 170.36bytes
Average data rate: 330.876kbit/s
Average connections managed per second: 248.61
Average reuse of each SA (before closing): 421.77

Total cache misses: 124369 (6.95%)
Compulsory misses: 4244
Avoidable cache misses: 120125 (6.71%)
Average reuse of each cache position before replacing 14.39
```

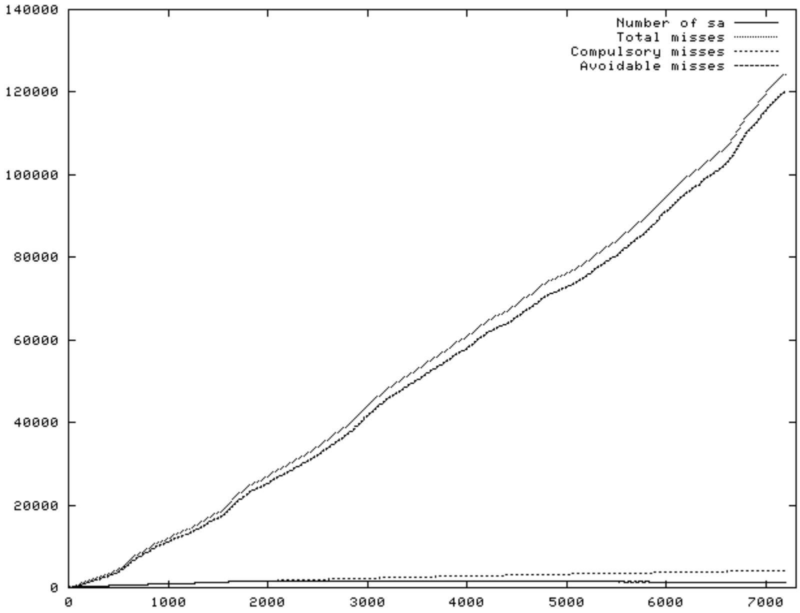


Figure 6.24: simulation results over time with a 64-entry cache, considering a 30min. timeout on the unused SAs

In Figure 6.25 a graphic representation of the results obtained for a 128-entry cache is shown. The numerical results are shown below:

```
Total cache misses: 25122 (1.40%)
Compulsory misses: 4244
Avoidable cache misses: 20878 (1.17%)
Average reuse of each cache position before replacing 71.25
```

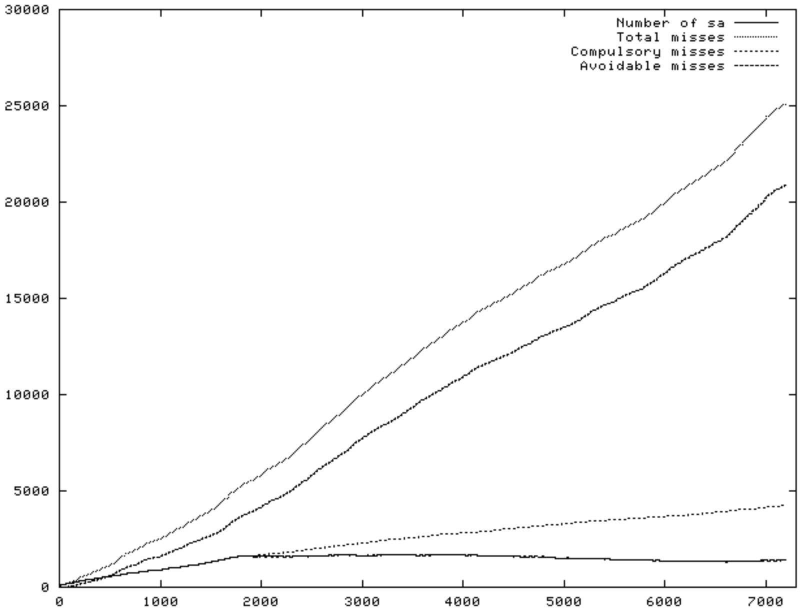


Figure 6.25: simulation results over time with a 128-entry cache, considering a 30min. timeout on the unused SAs

In Figure 6.26 a graphic representation of the results obtained for a 256-entry cache is shown. The numerical results are shown below:

```
Total cache misses: 11892 (0.66%)
Compulsory misses: 4244
Avoidable cache misses: 7648 (0.43%)
Average reuse of each cache position before replacing 150.52
```

We can note that, using a 30 minutes timeout for the unused SAs, we obtain slightly less avoidable cache misses and the same number of total cache misses, with respect to the same cache dimensions without considering any closure policy. The number of compulsory cache misses grows due to the fact that some SAs which has been closed for having exceeded the timeout need to be re-opened later in the simulation (see also section 6.3).

The fact that this policy is very efficient with respect to SA reusing and system's resources is also shown by the average reuse of each SA that is very similar to the one obtained with no timeout set.

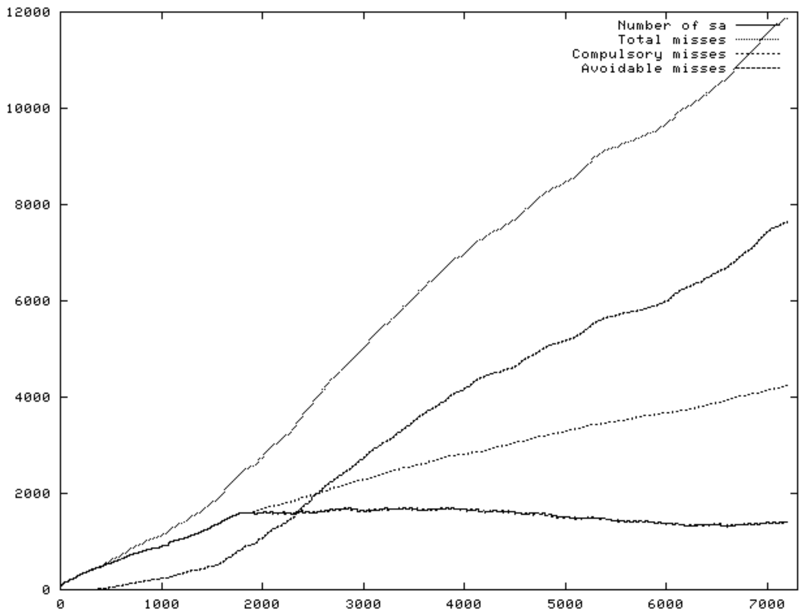


Figure 6.26: simulation results over time with a 256-entry cache, considering a 30min. timeout on the unused SAs

6.4.4.3. Using the TCP FIN packets

In Figure 6.27 and Figure 6.28 are represented the results obtained with a 128 and a 256-entry caches. Those graphs show the number of total cache misses over time, the number of compulsory cache misses over time, and the number of avoidable (total-compulsory) cache misses over time. The other curve shown in each graph represents the number of SAs opened over time.

Since the first part of the numerical results obtained here (the data rate, the SA reuse, the number of analyzed datagrams, and the number of connections managed per second) is always the same for each simulation run performed in this subsection, we will report it only in the first case shown.

The numerical results for a 64-entry cache are shown below:

```
Total number of datagrams analyzed 1789994
Average dimension of datagrams (34bytes of header): 170.36bytes
Average data rate: 330.876kbit/s
Average connections managed per second: 248.61
Average reuse of each SA (before closing): 140.29

Total cache misses: 128601 (7.18%)
Compulsory misses: 12759
Avoidable cache misses: 115842 (6.47%)
Average reuse of each cache position before replacing 13.92
```

In Figure 6.27 a graphic representation of the results obtained for a 128-entry cache is shown. The numerical results are shown below:

```
Total cache misses: 31804 (1.78%)
Compulsory misses: 12759
Avoidable cache misses: 19045 (1.06%)
Average reuse of each cache position before replacing 56.28
```

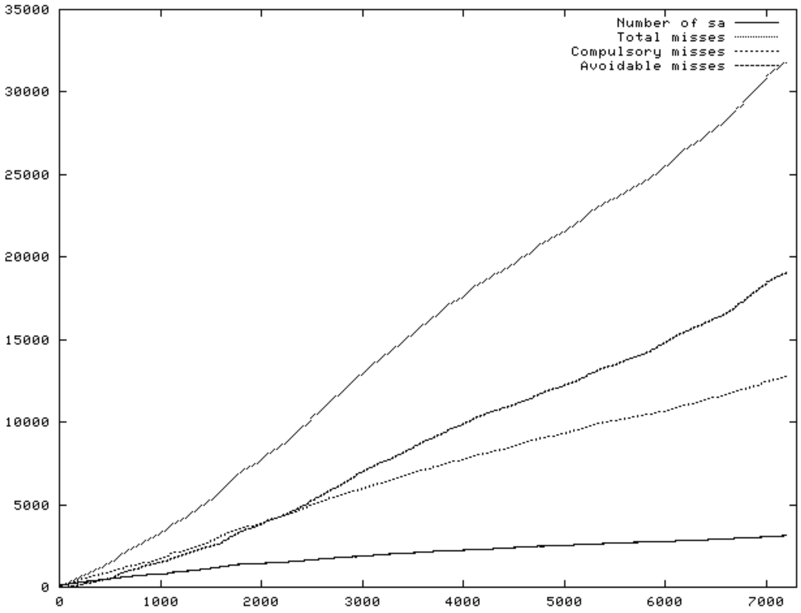


Figure 6.27: simulation results over time with a 128-entry cache, using the TCP FIN packets for closing the SAs

In Figure 6.28 a graphic representation of the results obtained for a 256-entry cache is shown. The numerical results are shown below:

```
Total cache misses: 19113 (1.07%)
Compulsory misses: 12759
Avoidable cache misses: 6354 (0.35%)
Average reuse of each cache position before replacing 93.65
```

The policy of using the TCP FIN packet for deciding whether to close a SA seems to be not very efficient. It allows a slightly diminishing of the avoidable cache misses due to the lower number of opened SAs at a given time, but it causes a dramatic increase of the compulsory cache misses due to the fact that some SAs closed for having received a TCP FIN packet often need to be reopened. The bad behavior provided by the usage of this technique is also confirmed by the average reuse of each SA before being closed: in this case the result obtained is really different from the one obtained with no closure policy or with the closure policy based on a timeout.

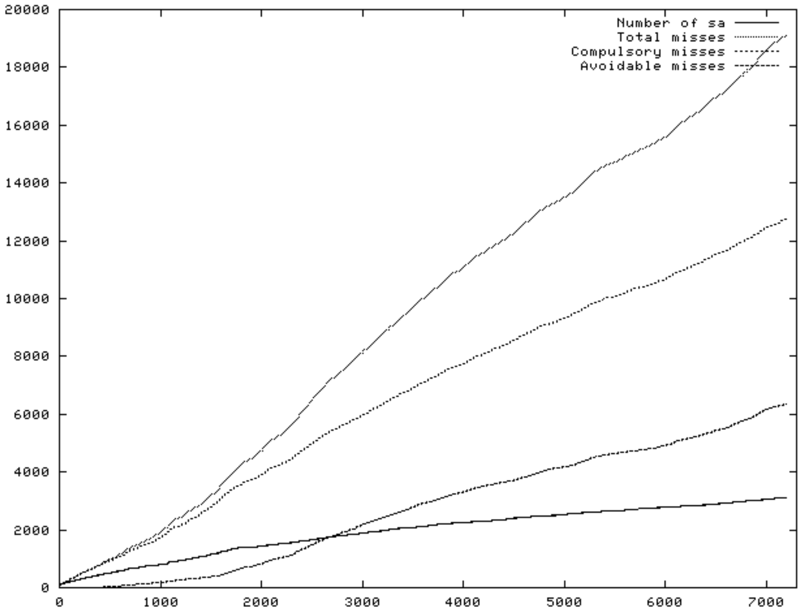


Figure 6.28: simulation results over time with a 256-entry cache, using the TCP FIN packets for closing the SAs

6.4.4.4. Closing the SAs exceeding a 30 min. timeout and using the TCP FIN packets

In Figure 6.29 and Figure 6.30 are displayed the results obtained with a 128 and a 256-entry caches. Those graphs show the number of total cache misses over time, the number of compulsory cache misses over time, and the number of avoidable (total-compulsory) cache misses over time. The other curve shown in each graph represents the number of SAs opened over time.

Since the first part of the numerical results obtained here (the data rate, the SA reuse, the number of analyzed datagrams, and the number of connections managed per second) is always the same for each simulation run performed in this subsection, we will report it only in the first case shown.

The numerical results for a 64-entry cache are shown below:

```
Total number of datagrams analyzed 1789994
Average dimension of datagrams (34bytes of header): 170.36bytes
Average data rate: 330.876kbit/s
Average connections managed per second: 248.61
Average reuse of each SA (before closing): 135.27

Total cache misses: 128601 (7.18%)
Compulsory misses: 13233
Avoidable cache misses: 115368 (6.45%)
Average reuse of each cache position before replacing 13.92
```

In Figure 6.29 a graphic representation of the results obtained for a 128-entry cache is shown. The numerical results are shown below:

```
Total cache misses: 31804 (1.78%)
Compulsory misses: 13233
Avoidable cache misses: 18571 (1.04%)
Average reuse of each cache position before replacing 56.28
```

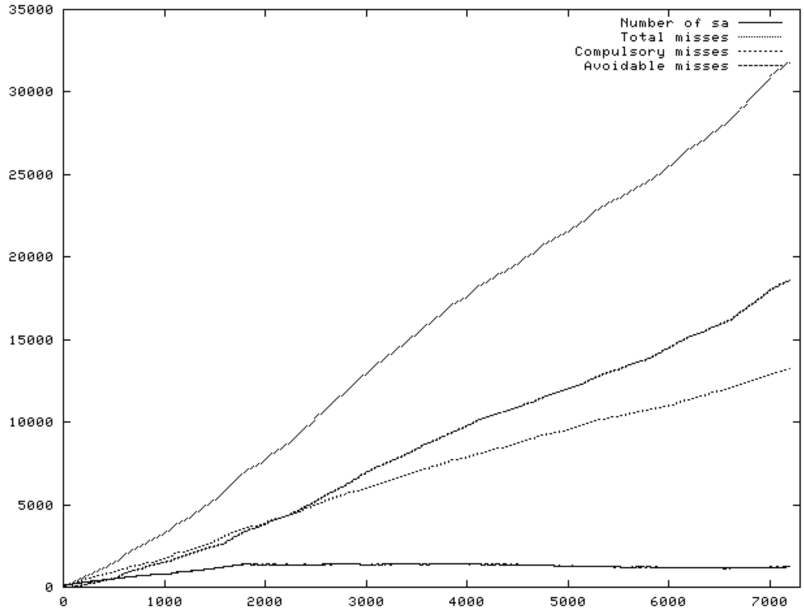


Figure 6.29: simulation results over time with a 128-entry cache, when a 30min. timeout on the unused SAs is set and using the TCP FIN packets for closing the SAs

In Figure 6.30 a graphic representation of the results obtained for a 256-entry cache is shown. The numerical results are shown below:

```
Total cache misses: 19113 (1.07%)
Compulsory misses: 13233
Avoidable cache misses: 5880 (0.33%)
Average reuse of each cache position before replacing 93.65
```

In this case are again pretty evident the bad effects caused by closing the SAs basing on the TCP FIN packets. This technique should never be used.

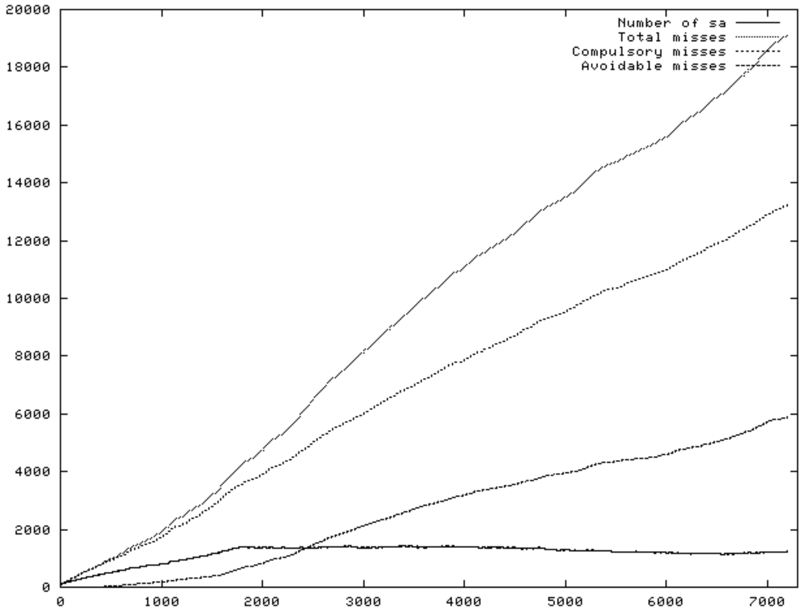


Figure 6.30: simulation results over time with a 256-entry cache, when a 30min. timeout on unused SAs is set and using the TCP FIN packets for closing the SAs

6.4.4.5. Conclusions

In all the four examined cases the results about the avoidable cache misses are very similar, however obtained with different combinations of total and compulsory misses. The data about the reuse of each SAC element before being discarded is different in the four considered cases, strictly depending it on the SA reuse (see section 6.3 for further considerations about that topic).

Consideration about the number of avoidable cache misses leads us to prefer the SAC to be composed of 128 elements (16kb) or of 256 elements (32kb). Further considerations can be done also considering the SA creation phase, as reported in section 6.4.7.

6.4.5. Reuse of each cache entry

Before proceeding it is really important to verify if we can obtain any benefits caching the SAs. This can be done by evaluating the data about the reuse of each cache entry. As a matter of fact we can obtain very few cache misses, but, if each cache element were reused very few times, we would have no real benefits using that cache.

In the results of the simulations (section 6.4.4) it is reported the average reuse of each cache entry before being discarded. From that data we can see that using a cache seems to be a very good idea. As a matter of fact, in each considered case, each cache entry is used many times (in average 71 times for a 128-entry cache and 150 for a 256-entry cache when no SA closing policy has been adopted).

The reuse of each cache entry is shown in Figure 6.31 (128-entry cache and no timeout), Figure 6.32 (256-entry cache and no timeout), and Figure 6.33 (128-entry, 30 minutes timeout). In those graphs each number shown on the abscissas axe represents a cache entry replace. For example, the number 1 on that axe represents the first cache replacement occurred. The numbers shown on the ordinates axe represent the number of times each entry had been reused before a replacement.

The different shapes of the three graphs are due to the different order of the replacements obtained considering different cache sizes. The different number of bars shown in those graphs is due to the different number of cache replacements obtained considering different cache dimensions.

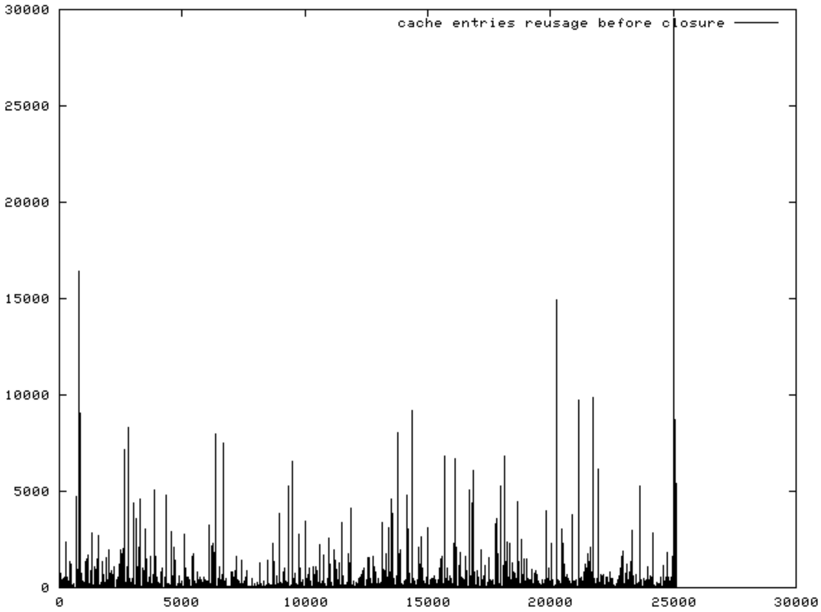


Figure 6.31: reuse of each cache entry before being discarded on a 128-entry cache

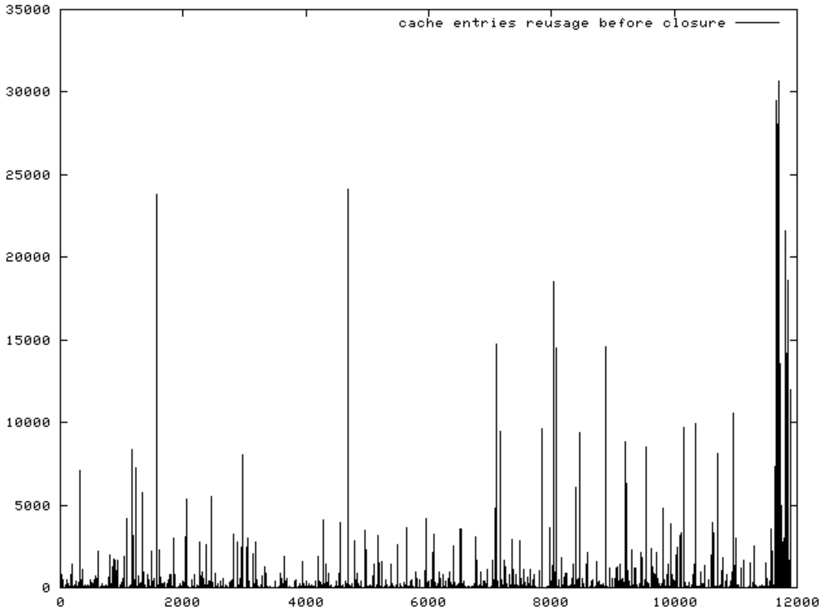


Figure 6.32: reuse of each cache entry before being discarded on a 256-entry cache

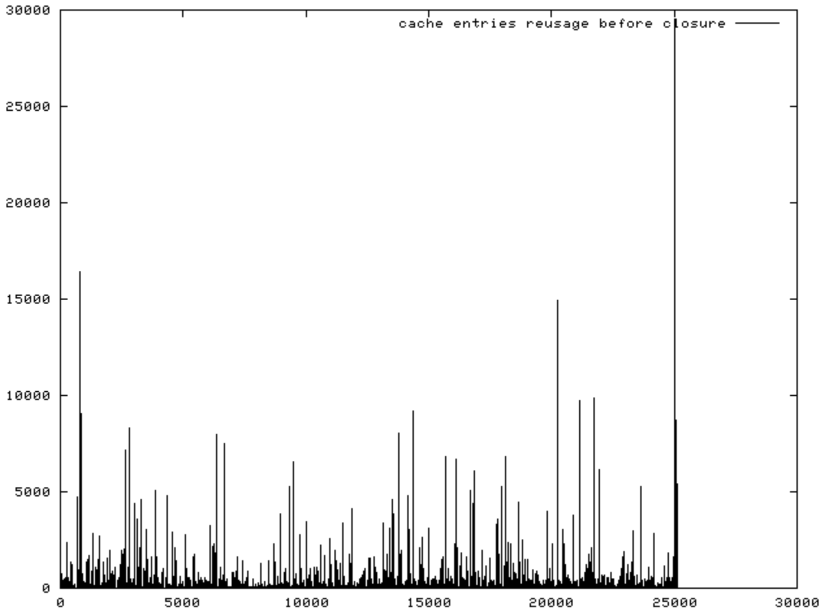


Figure 6.33: reuse of each cache entry before being discarded on a 128-entry cache, when a 30min. timeout on the unused SAs is set

From those figures we can also note that there are some SAs remaining in the cache for a very long time, having a very high reuse.

6.4.6. Using a different cache replace policy

We tried using a random replace policy ([P-H] , pp. 380-402) instead of the LRU one and the results we obtained was considerably worse. This confirm us that there really is a temporal locality that can be exploited using the LRU replacing policy and an adequate number of cache elements.

6.4.7. Taking into account the SA creation phase

Since the data files we have do not contain any data about SAs creation, the results shown in the previous section are about a system where all the SAs are supposed to have already been created.

Looking at the diagrams which show the distribution of the newly opened SAs and which are reported in section 6.3 (Figure 6.5, Figure 6.6, and Figure 6.7), we could see that the creation of new SAs is pretty uniformly distributed over time. This means that, for simulation purposes only, we can try reserving a certain number of entries in the SAC for the creation of new SAs. Looking at the diagrams it seems that reserve 20 cache position could be a good (and probably conservative) choice. This is done just for causing a SAC space diminishing like the one that would have been caused by the creation of new SAs during the normal working of a real system, since this operation also uses the SAC memory space. The simulation can be run again using 108 and 236 as SAC dimensions.

In Figure 6.34 are shown the results obtained with a 108-entry cache without using any SA closing policy. The average results follow here:

```
Total number of datagrams analyzed 1789994
Average dimension of datagrams (34bytes of header): 170.36bytes
Average data rate: 330.876kbit/s
Average connections managed per second: 248.61
Average reuse of each SA: 483.13

Total cache misses: 34625 (1.93%)
Compulsory misses: 3705
Avoidable cache misses: 30920 (1.73%)
Average reuse of each cache position before replacing 51.70
```

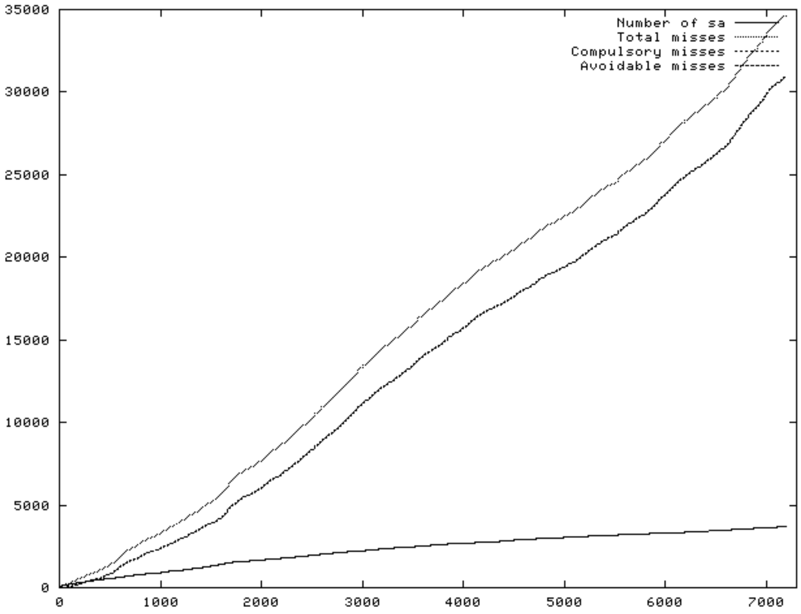


Figure 6.34: simulation results over time with a 108-entry cache

In Figure 6.35 are shown the results obtained with a 236-entry cache without using any SA closing policy. The average results follow here:

```
Total cache misses: 13027 (0.73%)
Compulsory misses: 3705
```

`Avoidable cache misses: 9322 (0.52%)`
`Average reuse of each cache position before replacing 137.41`

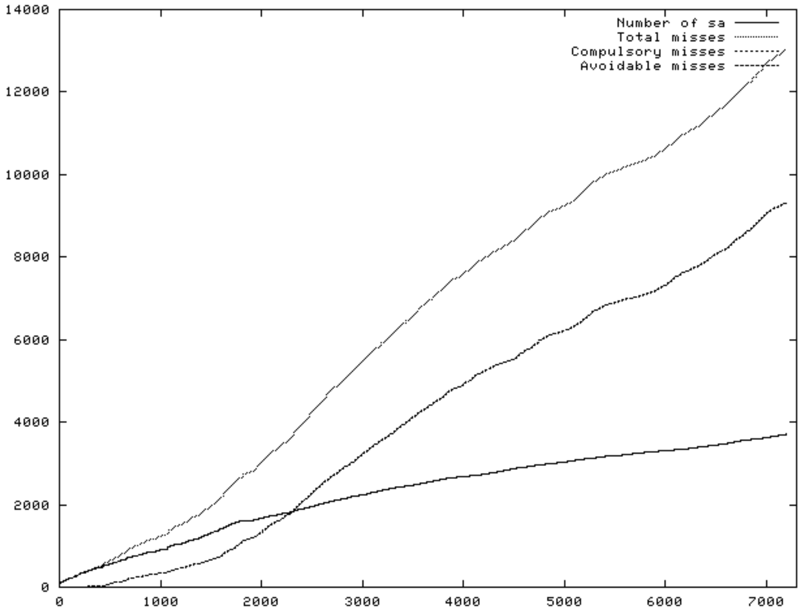


Figure 6.35: simulation results over time with a 236-entry cache

The results obtained for the 236-entry cache are pretty much the same of the ones obtained with the 256-entry cache, while the diminishing of the number of entries influences more the 128-entry cache. As a matter of fact throwing out 20 entries form the 128-entry cache, we left the 15,6% of the elements, while for the 256-entry cache we have a reduction of only the 7.8%.

6.4.8. Considering the SAs created as in “IKE Phase 2 – quick mode”

In the previously discussed simulation, we did not care about the fact that, when IKE Phase 2 quick mode is performed, two SAs are created at the same time (see section 3.4.2). Those two SAs are created to accomplish the needing of bi-directional communications. The results shown before was obtained creating each SA the first time it was needed, without considering the communications in the opposite direction.

In this subsection we show the results obtained running a slightly modified version of the simulation program. Here, when a new SA is opened, another SA between the same two IP addresses but in the opposite direction is automatically opened too.

We have to note that in this section we are not adding any information about the SA creation phase (such as functionalities or timings) to the simulation, we only force the system to a slightly different usage of the cache that should be closest to the real one.

The obtained results (without considering any SA closure policy) are shown below.

Considering a 64-entry cache we obtain:

```
Total cache misses: 124852 (6.97%)
Compulsory misses: 3806
Avoidable cache misses: 121046 (6.76%)
```

Considering a 108-entry cache we obtain:

```
Total cache misses: 34890 (1.95%)
Compulsory misses: 3806
Avoidable cache misses: 31084 (1.74%)
```

Considering a 128-entry cache we obtain:

```
Total cache misses: 25326 (1.41%)
Compulsory misses: 3806
Avoidable cache misses: 21520 (1.20%)
```

Considering a 256-entry cache we obtain:

```
Total cache misses: 12071 (0.67%)
Compulsory misses: 3806
Avoidable cache misses: 8265 (0.46%)
```

We can note that the obtained results about the cache misses are pretty much the same as obtained in the previous sections. We can note that the number of compulsory misses is a little bit increased here. This is due to the fact that some SAs that would not need to be opened are anyway opened. As a matter of fact, in the previous simulations, only the needed SAs was opened, while here a pair of SAs is anyway opened whether or not a bi-directional channel is needed.

The results about the number of created SAs when no closure policy is adopted are also pretty much the same as the ones shown in section 6.3. That distribution is displayed in Figure 6.36. The only data which slightly change are the ones about the reuse of the SAs and of the cache entries. As a matter of fact, using this SA creation procedure, we force the system to open some not-needed (and not used) SAs, making the SA reuse and the cache entry reuses to lower. For example the average SA reuse goes from 483 to 470.

Since the SAs are created in pairs, an alternative cache structure can be thought. We can think about using cache entries which can contain the information of a pair of SAs each. That solution would give no advantage in our system, being that the information contained in each single-SA cache entry (mainly the AES key and the IV) must anyway be contained in the new cache structure too. Therefore we would obtain no memory saving and we would lower the flexibility of the cache. That technique would also introduce a further complication for managing the ISAKMP SAs that are bi-directional and do not need a double cache entry.

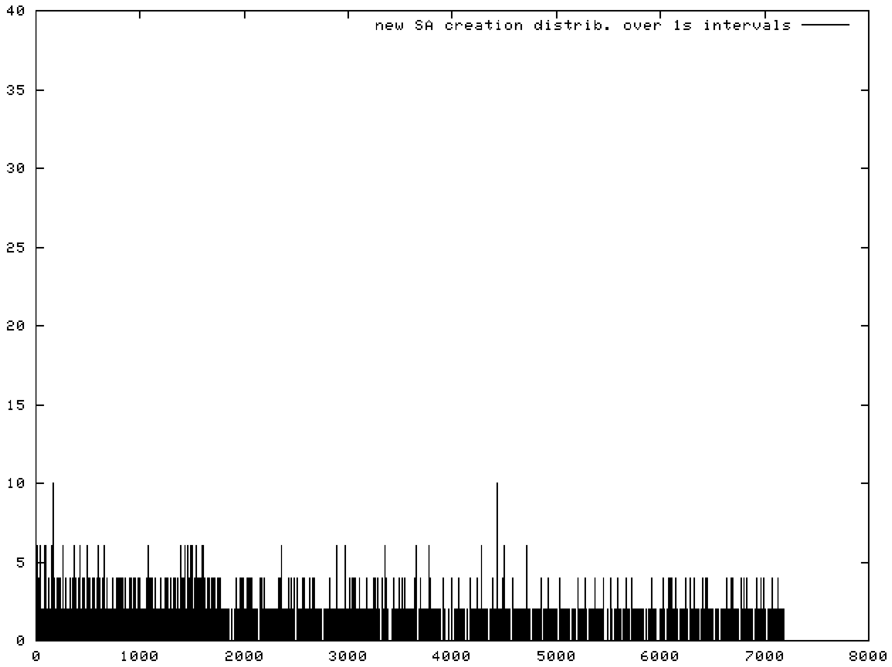


Figure 6.36: SA creation distribution over 1s intervals when *IKE Phase 2 quick mode* procedure is used for opening the SAs

6.5. Simulating the delay introduced by the crypto-processor

To simulate the delay introduced by the crypto-processor, we need to compute the delays introduced by the various operations which need to be performed. These operations are: the data transfer between the host and the smart card, the encryption or decryption of the data, the time needed for dealing with an element not present in the SAC or with an element present in the SAC, the storing time of a SAC element. Those delays will then be used in the simulation program to compute the time that each packet needs to pass through the system.

6.5.1. Computation of the delays

6.5.1.1. Host–smart card data transfer

As stated before, we suppose to use a 32-bit standard 66MHz PCI bus as hardware interface between the host and the crypto-processor. Therefore we will use the data taken by the PCI specification to compute the communication channel delays.

The PCI bus in burst mode needs one cycle for the address and n divided by 4 cycles for the data, where n is the number of bytes to be transferred. Being the bus at 66MHz, each cycle takes 1 over $66*10^6$ that is $1.515*10^{-8}$s. We consider as initial time one cycle, since there should be no conflict on the data bus. As a matter of fact the bus is dedicated to the crypto processor and all the transfer requests are sequentially done.

The time we need to transfer something on the bus, is given by:

$$D_{trans}(b) = D_{initial} + \left(\left\lceil \frac{b}{4} \right\rceil + 1 \right) * t_{cycle}$$ [7]

where:

- $\boldsymbol{D_{initial}}$ is the time needed to start the communication. We can assume for it the value of $1.515*10^{-8}$s.
- ***b*** is the number of bytes we have to transfer: this parameter can be obtained by the command table and in each case we are considering here it is given by the number of data byte plus 4 (one 32-bit words of command code and options). The first time the SA is used (depending on the mode used for encryption) the IV should also need to be loaded in the smart card. In that case 16 (equivalent to 128 bits) need be added to the data dimension.
- $\boldsymbol{t_{cycle}}$ is the time needed by a PCI bus cycle that is, as stated before, $1.515*10^{-8}$s.

The delay introduced by the data transferring is given by:

$$D_{trans}(b) = 1.515*10^{-8} + \left(\left\lceil \frac{b}{4} \right\rceil + 1 \right) * 1.515*10^{-8}$$

When the data to be encrypted or decrypted are to be transferred through a *symmEncrypt* or a *symmDecrypt* command, a further 32-bit word of command code must be added, since those command are of the *C* format (see section 4.2.4). In those cases the previous formula become:

[7] The symbol $\lceil x \rceil$ means “the smallest integer greater than or equal to x” This is equivalent to the C function called *ceil.*

$$D_{trans}(b) = 1.515 * 10^{-8} + \left(\left\lceil \frac{b}{4} \right\rceil + 2 \right) * 1.515 * 10^{-8}$$

6.5.1.2. Encryption/decryption

This delay can be computed considering the worst-case-time needed to process a data block, the number of blocks to be processed, and a fixed initial delay. The initial delay is needed for setting up the AES hardware with the key to be used (key enrolling). We can use a formula like the following one:

$$D_{enc}(c) = D_{setup} + \left\lceil \frac{8 * c}{128} \right\rceil * t_{enc} = D_{setup} + \left\lceil \frac{c}{16} \right\rceil * t_{enc}$$

where:

- $\boldsymbol{D_{setup}}$ is the initial fixed delay
- $\boldsymbol{c}$ is the number of bytes to be processed. Here we consider the worst possible case, so we also take into account also the IP header dimension: this corresponds to consider the ESP IPSec protocol used in tunnel mode.
- $\boldsymbol{t_{enc}}$ is the time we need for the encryption of one data packet.

Looking at the AES fast hardware implementation studies done at ALaRI (see [MAC-MAR] for more details), we can assume the following values for the previous variables:

$\boldsymbol{D_{setup}}$**:**	17 clock cycles * cycle time
$\boldsymbol{t_{enc}}$**:**	22 clock cycles * cycle time

The decryption process is slower than the encryption one and for that reason we choose to use the data relative to the slowest process.

Considering the same studies on AES we can see that running the AES hardware at 50MHz we can obtain a throughput of 290Mbit/s with a continuous flux of information (i.e. giving to that hardware a continuous flux of data related to the same AES key). In our system the flux will not be continue. Therefore we will have to choose a higher clock rate than 50MHz. The values for D_{setup} and t_{enc} considering various clock rates are reported in Table 6.4.

Clock rate	**50MHz**	**55MHz**	**60MHz**	**70MHz**
D_{setup}	$3.4*10^{-7}$	$3.09*10^{-7}$	$2.83*10^{-7}$	$2.42*10^{-7}$
t_{enc}	$4.4*10^{-7}$	$4*10^{-7}$	$3.67*10^{-7}$	$3.14*10^{-7}$

Table 6.4: encryption timings for different AES hardware clock rates

6.5.1.3. Element present in the SAC

In this case the delay introduced is very low. If a SA element were cached, the SAC position number would be stored in the SAD, that would anyway be consulted. The delay can be considered to be $5*10^{-9}$ sec.

6.5.1.4. Element not present in the SAC

In this case there are two different possibilities: there is a free SAC slot or there is not a free SAC slot.

In the former case, the delay is given by the sum of the time needed to transfer the SA data (from the host to the crypto-processor) and of the time needed to decrypt the key and to check the CRC. In the latter case, we must also add the store time of the discarded SAC element (the one that need to be stored out for freeing a memory position in the crypto-processor). We consider that our system uses the simple 16-bit CRC algorithm to check the consistency of the data stored out of the smart card. The CRC is computed over the contents of a SAC element, then the result is encrypted with the key and stored on the host. The time needed for comparing the CRC result with the stored CRC can be neglected, since that operation is performed on two bytes only.

We have:

$$D_{miss} = D_{enc}(k_d/8) + D_{trans}(b) + D_{crc}(b) =$$
$$= D_{setup} + \left\lceil \frac{k_d+16}{128} \right\rceil * t_{enc} + 1.515*10^{-8} + \left(\left\lceil \frac{b}{4} \right\rceil + 1 \right) * 1.515*10^{-8} + D_{crc}(b)$$

$$D_{store} = D_{enc}(k_e/8) + D_{trans}(b) + D_{crc}(b) + D_{trans}(0) =$$
$$= D_{setup} + \left\lceil \frac{k_e+16}{128} \right\rceil * t_{enc} + 2*1.515*10^{-8} + \left(\left\lceil \frac{b}{4} \right\rceil + 2 \right) * 1.515*10^{-8} + D_{crc}(b)$$

where:

- ***D_{setup}*** and ***t_{enc}*** are the same ones considered in the computation of the encryption delay.
- ***k_e*** and ***k_d*** are respectively the key length (bit) of the SA to be stored out from the crypto-processor and the key length of the SA to be put in the SAC. Both the k_e and the k_d values are considered to be 256 (the maxim length of an AES key). The 16 bits added in the formula are the ones related to CRC result.
- ***b*** is the number of bytes to be exchanged between the host and the crypto processor to load the information about a SA.
- ***D_{crc}*** is the delay introduced by the data "hashing" function chosen, the CRC. The CRC needs to operate on *b-2* bytes, since 2 bytes are generated by the CRC itself:

  $$D_{crc}(b) = (b-2) * t_{crc}$$

  In the previous formula ***t_{crc}*** is the time we need to apply CRC to a byte. In CRC hardware implementations a throughput of 200Mbit/s can be easily reached, so that t_{crc} can be considered to be 8 over $200*10^6$ that is $4*10^{-8}$.

The $D_{trans}(0)$ delay is introduced to take into account that a store request must be done by the host through the *getSAinfo* command.

Considering all the timings introduced above, the D_{miss} and D_{store} formulas become:

$$D_{miss} = D_{enc}(k_d/8) + D_{trans}(b) + D_{crc}(b) =$$

$$= D_{setup} + \left\lceil \frac{k_d + 16}{128} \right\rceil * t_{enc} + 1.515 * 10^{-8} + \left(\left\lceil \frac{b}{4} \right\rceil + 1 \right) * 1.515 * 10^{-8} + D_{crc}(b)$$

$$D_{store} = D_{enc}(k_e/8) + D_{trans}(b) + D_{crc}(b) =$$

$$= D_{setup} + \left\lceil \frac{k_e + 16}{128} \right\rceil * t_{enc} + 2 * 1.515 * 10^{-8} + \left(\left\lceil \frac{b}{4} \right\rceil + 2 \right) * 1.515 * 10^{-8} + D_{crc}(b)$$

b can be assumed to be 66 (64 bytes taken from the SAC plus 2 of CRC). The CRC is applied only to the 64 bytes of the SAC entry. The two delays can now be considered to be:

$$D_{miss} = D_{setup} + \left\lceil \frac{256+16}{128} \right\rceil * t_{enc} + 1.515 * 10^{-8} + \left(\left\lceil \frac{b}{4} \right\rceil + 1 \right) * 1.515 * 10^{-8} + (b-2) * 4 * 10^{-8} =$$
$$= D_{setup} + 3 * t_{enc} + 2.87 * 10^{-7} + 2.56 * 10^{-6}$$

$$D_{store} = D_{setup} + \left\lceil \frac{256+16}{128} \right\rceil * t_{enc} + 2 * 1.515 * 10^{-8} + \left(\left\lceil \frac{b}{4} \right\rceil + 2 \right) * 1.515 * 10^{-8} + (b-2) * 4 * 10^{-8} =$$
$$= D_{setup} + 3 * t_{enc} + 3.18 * 10^{-7} + 2.56 * 10^{-6}$$

As stated before, when there is no free slot in the SAC, the delay is given by D_{miss}+D_{store}.

6.5.1.5. Conclusions about the delays

Considering the results obtained in the previous subsections, we can state that:

- Each operation on the smart-card introduces a delay given by:

  *2*D_{trans}+D_{enc}*

- Each cache access introduces a fixed delay D_{acc},
- Each cache miss introduces an additional delay of
  - D_{miss}, if there is a free slot in the SAC
  - D_{miss}+D_{store}, if there is no room for other SA in the SAC and a SA have to be swapped out

When the value of the parameters has been fixed, D_{miss}, D_{store}, and D_{acc} are constant. D_{trans} and D_{enc} depend on the number of bytes contained in the datagram to be processed.

6.5.2. *Designing the simulation*

The simulation we need to write here has still to provide a trace of the cache usage, but the computation of the delays introduced by the various operations done in the system must also be taken into account. In the previous section we explained what parts of the system arise those delays and how to compute them, therefore our simulation will have to use those information to compute the time needed by each datagram to pass through the system, basing on the operations to be performed on that datagram.

6.5.3. The simulation program

The C program used for the simulation including the delays is reported in Appendix C. It is based on the one used in the section 6.4 and explained in the subsection 6.4.2.

6.5.3.1. The main data structures

The *SAC* and *SAD* data structures we use here are the same ones we used for the simulation without delays. We explained them in section 6.4.3.1 and we showed them in Figure 6.17 and in Figure 6.18.

The *dataT* structure we use here was slightly modified to store some additional information we need to compute the delays. The new *dataT* structure is shown in Figure 6.37.

dataT

```
double time;
double theorTime;
int sourceIP;
int destIP;
int sourceTCP;
int destTCP;
int bytes;
```

Figure 6.37: the *dataT* structure

The new field added here is called *theorTime* and it is used to store the theoretical time at which that datagram would have been processed if there were no delays due to IPSec (the communication on the system where the data was taken is based on IP only). The *time* field is now used to store the real timestamp assigned to the considered datagram. As a matter of fact the original timestamps can need some modifications. This is due to the fact that the delays introduced by the system can make the datagrams to take longer processing times than the ones allowed by the original timestamps of the successive datagrams.

6.5.3.2. How the simulation program works

As stated before, the simulation we considered here works pretty much in the same way of the one described in section 6.4.3.1. Here the main differences are that the datagram processing delays are added for each executed operation. The delays are computed using the formulas explained in section 6.5.1.

When a SA is already cached the delay introduced is given by the data transfer between the host and the crypto-processor and by the encryption/decryption time of the data. When a SA is not cached also the delays due to the cache miss are added to the previous ones. If the sum of the introduced delays added to the datagram's timestamp is higher than the next datagram's timestamp, the latter is delayed. Therefore it takes a new timestamp composed by the previous datagram's timestamp plus the processing time and a little interval of time between the two packets.

In this way we are able to compute the throughput that can be obtained by the system. That throughput can be compared with the theoretical throughput, obtained considering the original timestamps read in the data file. The throughput is computed basing on time intervals set by the *PRINT_THROUGHPUT_DISTRIB* C constant. That constant has been set to 0.25s. Therefore, the real throughput is computed each time an interval of time higher that 0.25s is detected between the timestamp of the new datagram considered and a timestamp stored in memory (corresponding to the last throughput computation). The throughput is computed as the number of bytes processed in that interval divided by the interval length. For obtaining the theoretical throughput, the same operations are done considering the theoretical timestamps of the same datagrams. An average value of the throughput is computed at the end of the simulation, while instant values can be saved in a file for successive computations.

Here the results are put in separate files allowing to obtain more results per simulation run. The files that can be obtained are described in the following section.

6.5.3.3. Simulation parameters

The simulation options can be set through C constants (C *define* directive). The available options are:

- *CACHE_DIMENSION:* defines the cache dimension.
- *MAX_SA:* defines the maximum number of SAs allowed on the host.
- *USE_FIN_PACKETS:* defines whether or not to use the TCP FIN packets. The FIN packets are used when this constant is defined.
- *SCALE_TIME:* this parameter is used for scaling all the timestamps to obtain the desired theoretical throughput. For reaching an average throughput around 200Mbit/s we have to set this parameter to 0.00161.
- *CLOSE_UNUSED:* defines whether or not to use the a timeout on the opened SAs. The option is enabled when this constant is defined.
- *CLOSE_TIME:* defines the SA timeout to use (when the previous option is active).
- *CHECK_TIME:* defines the checking time for SAs exceeding the timeout.
- *MIN_DISTANCE:* defines the minimum time that should pass between two successive datagrams. This parameter is used when a datagram has to be delayed; in that case the datagram’s timestamp is obtained as the finishing processing time of the precedent datagram plus *MIN_DISTANCE.* This parameter is set to $1*10^{-12}$s.
- *PRINT_CACHE_DISTRIB:* when defined it makes the program save the distribution of the cache misses over 0.25s intervals into the *“cacheDistrib.txt”* file. That file contains the timestamps in the first column, the number of total cache misses in the second one, and the number of compulsory cache misses in the third one.
- *PRINT_THROUGHPUT_DISTRIB:* when defined it makes the program save the throughput obtained by the system and computed over the intervals specified by the constant itself into the “*throughput.txt*” file. That file contains the

timestamps in the first column and the throughput, that correspond to that timestamps, in the second column.

- *CHANNELL:* it is the corresponding to the t_{cycle} parameter described in section 6.5.1.1. Here is set to 1.515*10^{-8}s.
- *CHANNELL_INITIAL:* it is the $D_{initial}$ parameter used in section 6.5.1.1. Here is set to 1.515*10^{-8}s.
- *CACHED_DELAY:* it is the delay introduced by a SA that is already in cache as described in section 0. Here this constant is set to 5*10^{-9}
- *ENC_TIME:* it is the value corresponding to the t_{enc} parameter described in section 6.5.1.2 (see the following section for indication about the value set here).
- *ENC_SETUP:* it is the value corresponding to the D_{setup} parameter described in section 6.5.1.2 (see the following section for indication about the value set here).
- *SAINFO_LEN:* it is the number of bytes to be transferred between the host and the crypto-processor when a cache miss or a store operation occurs. The parameter is set to 66 bytes as described in section 6.5.1.4.
- *KEY:* it is the number of blocks to be encrypted. The value 3 is obtained by the two blocks which compose the key, plus a block for the CRC result as described in 6.5.1.4.
- *CRC_TIME:* it is the time we need to compute the CRC over a byte. As described in section 6.5.1.4, we can choose a value of 4*10^{-8}s for that parameter.
- *QUICK_MODE:* when defined it makes the program to threat the SAs as created with *IKE phase 2 – quick mode* (two SAs are created each time a new SA is needed for having a bi-directional communication channel).

6.5.4. The results of the simulations

All the simulations here were run setting the *SCALE_TIME* parameter to 0.00161 to obtain a theoretical average throughput (on all the datagrams analyzed) around 200Mbit/s (200.7Mbit/s).

We have to take into account that in this phase we scaled the timestamps to reach the desired throughput, but, in a real system, that throughput is usually reached in a different way. There usually is a higher number of opened connections (SAs, in our case). This may introduce some approximations from the point of view of SA managing and caching. Probably a better estimation of these effects could be done only using data coming from a real 200Mbit/s system.

Analyzing the results of the simulations we will see that the encryption time parameters can influence the cache dimension we need. A cache dimension that guarantees the desired throughput using a certain clock rate for the AES hardware can fail providing the same performance using a lower clock rate.

The theoretical throughput shown in each graph of this section is obtained running the simulation program with all the timing parameters set to 0. This would be the throughput of the system when no delay were introduced by the crypto system (or if the delays are smaller than the minimum interval between successive datagrams).

The throughput is everywhere computed over 0.25s intervals[8]: using different interval widths can give different throughput curve shapes, always leading to the same results, since the two curves plotted in the graphs are obtained using the same value for this parameter.

6.5.4.1. Running the AES hardware at 50MHz

As explained in section 6.5.1.2, considering a 50MHz clock for the AES hardware, the *ENC_TIME* and *ENC_SETUP* parameters have to be respectively set to $4.4 * 10^{-7}$ and to 3.410^{-7}.

The throughput obtained for a 64-entry cache, and for a 256-entry cache is reported in Figure 6.38 and in Figure 6.39. From that figures can be pointed out that the cache dimensions influence very few the throughput of the system in this situation. As a matter of fact the two graphs differ very few while the cache dimension in the two cases differ for a factor of four. We can state that in this case the cache dimension parameter

[8] The throughput is always obtained computing the number of bytes going through the system over an interval of time.

“is dominated” by the encryption time parameter that partially hides the effects of the former parameter.

No one of the cache dimensions here considered can guarantee the desired throughput with such a slow cryptographic hardware.

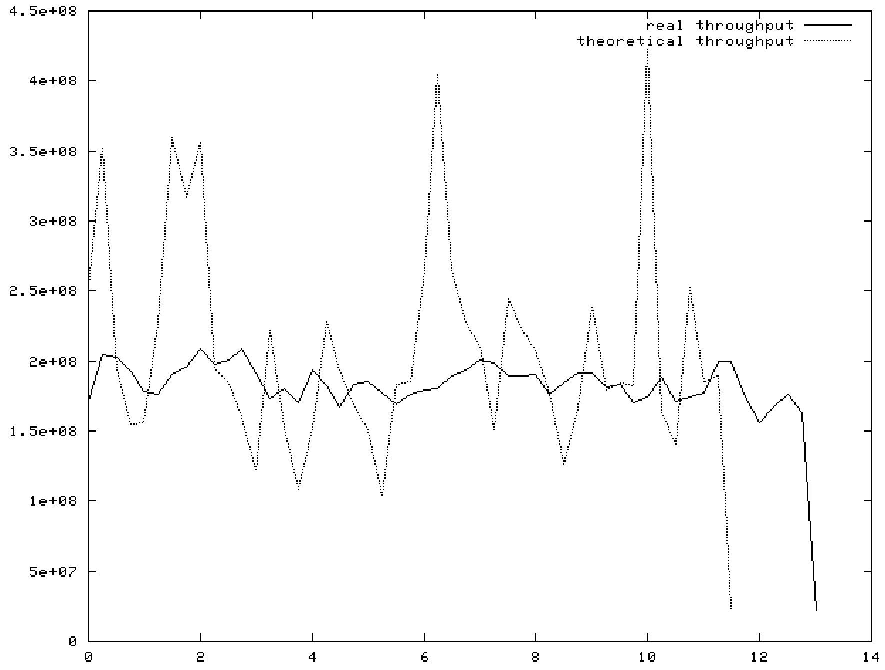


Figure 6.38: throughput obtained using a 64-entry cache and an AES hardware running at 50MHz

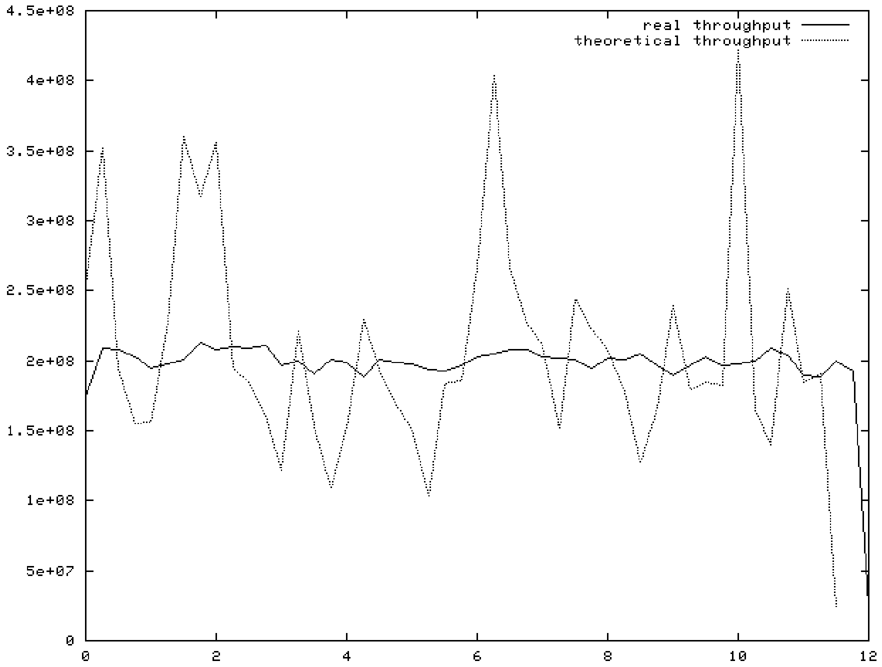


Figure 6.39: throughput obtained using a 256-entry cache and an AES hardware running at 50MHz

6.5.4.2. Running the AES hardware at 55MHz

As explained in section 6.5.1.2, considering a 55MHz clock for the AES hardware, the *ENC_TIME* and *ENC_SETUP* parameters have to be respectively set to $4*10^{-7}$ and to $3.09*10^{-7}$.

The throughput obtained for a 64-entry cache, for a 128-enty cache, and for a 256-entry cache is respectively reported in Figure 6.40, in Figure 6.41 and in Figure 6.42. From that figures it can be pointed out that using a 64 cache does not allow to obtain the required throughput. A 128-entry and a 256-entry cache seem allowing the system to almost provide the desired throughput. This can also be seen by the numerical results obtained from the simulations: the average throughput obtained for the 64-entry cache is 188.65Mbit/s, the one obtained for the 128-entry cache is 198.10Mbit/s, and the one obtained for the 256-entry cache is 199.15Mbit/s. The average theoretical throughput is 200.70Mbit/s.

The performance would probably be more safely guaranteed with a higher AES hardware clock.

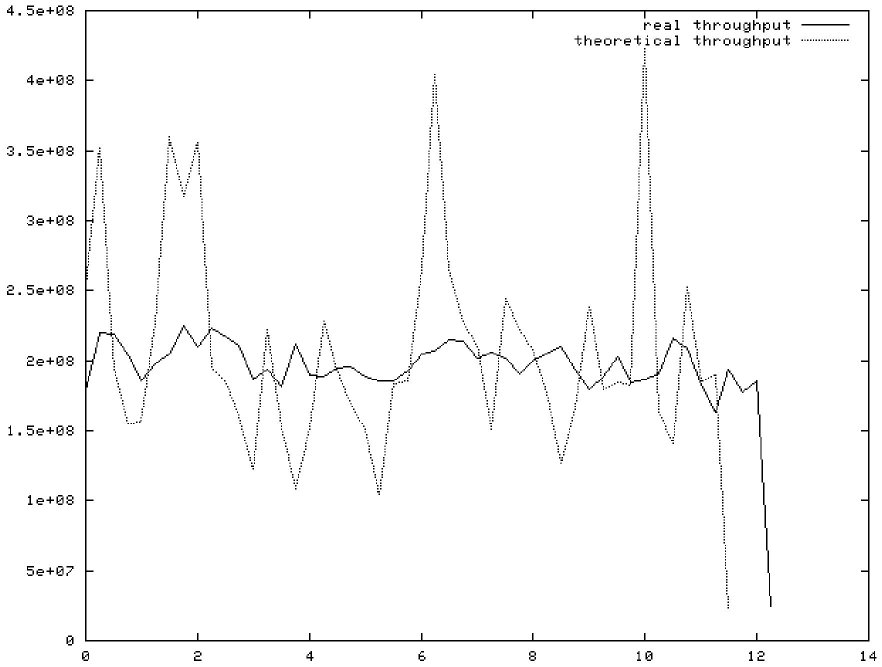


Figure 6.40: throughput obtained using a 64-entry cache and an AES hardware running at 55MHz

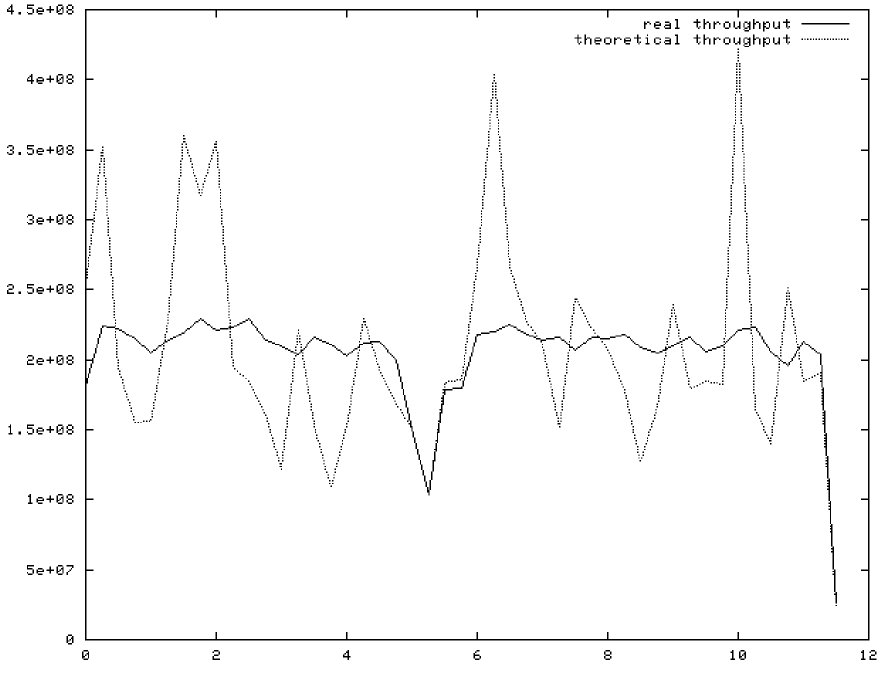


Figure 6.41: throughput obtained using a 128-entry cache and an AES hardware running at 55MHz

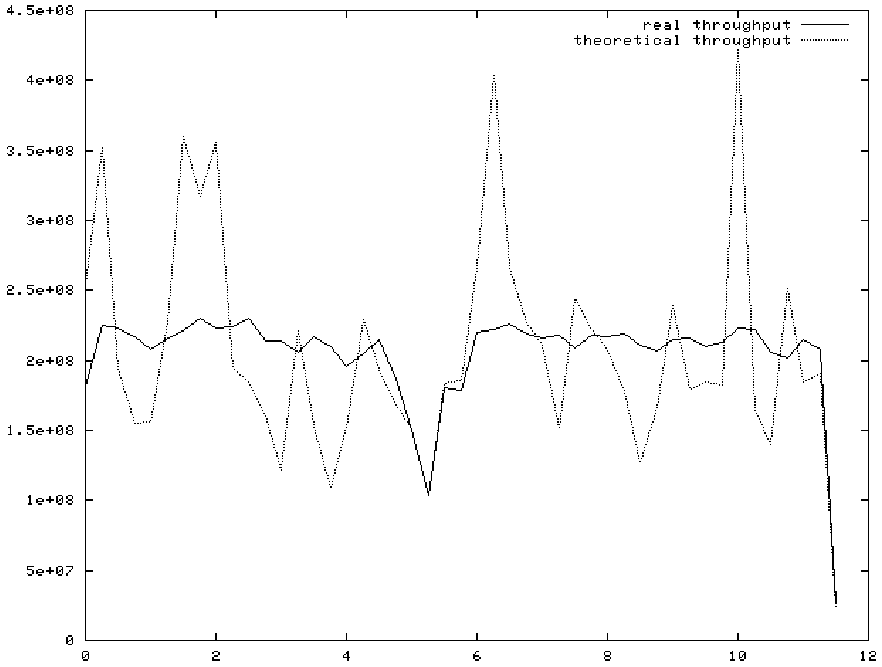


Figure 6.42: throughput obtained using a 256-entry cache and an AES hardware running at 55MHz

6.5.4.3. Running the AES hardware at 60MHz

As explained in section 6.5.1.2, considering a 60MHz clock for the AES hardware, the *ENC_TIME* and *ENC_SETUP* parameters have to be respectively set to $3.67*10^{-7}$ and to $2.83*10^{-7}$.

The throughput obtained for a 64-entry cache, for a 128-entry cache, and for a 256-enty cache is respectively reported in Figure 6.43, in Figure 6.44, and in Figure 6.45. From that graphs we can point out that using a 60MHz clock rate for the AES hardware we can reach the desired performance using a 128-entry cache. A 64-entry cache still cannot be used.

The numerical data for the throughput here obtained are 196.35Mbit/s for the 64-entry cache, 200.63Mbit/s for the 128-entry cache, and 200.65Mbit/s for the 256-entry cache.

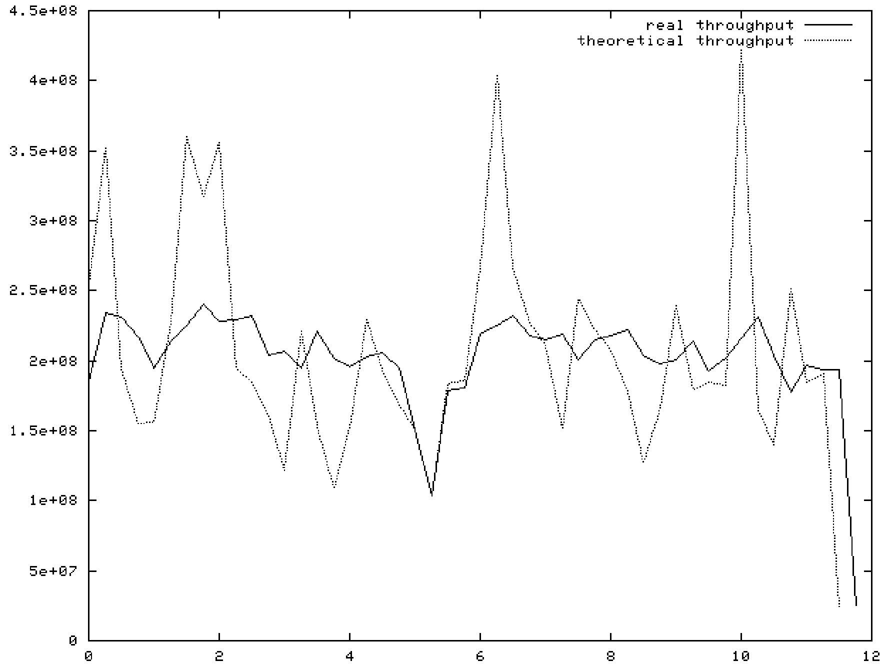


Figure 6.43: throughput obtained using a 64-entry cache and an AES hardware running at 60MHz

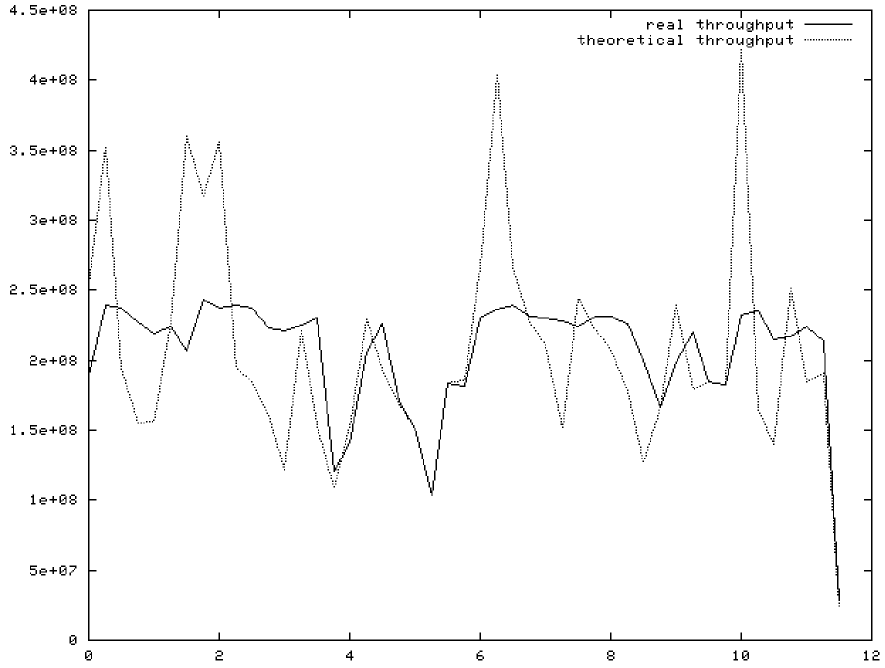


Figure 6.44: throughput obtained using a 128-entry cache and an AES hardware running at 60MHz

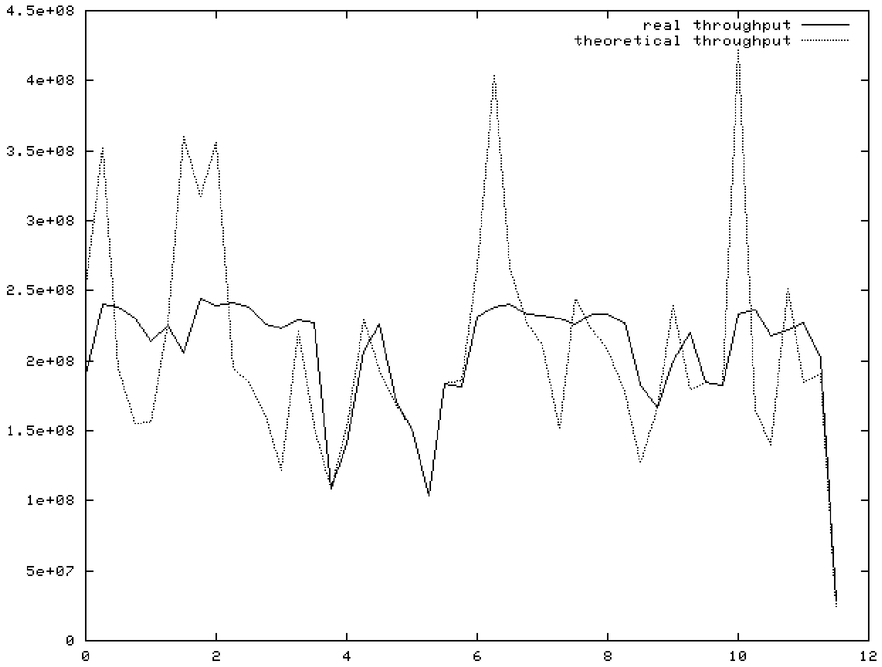


Figure 6.45: throughput obtained using a 256-entry cache and an AES hardware running at 60MHz

6.5.4.4. Running the AES hardware at 70MHz

As explained in section 6.5.1.2, considering a 70MHz clock for the AES hardware, the *ENC_TIME* and *ENC_SETUP* parameters have to be respectively set to $3.14*10^{-7}$ and to $2.42*10^{-7}$.

The throughput obtained for a 64-entry cache and for a 128-entry cache is respectively reported in Figure 6.46 and in Figure 6.47. Here it can be seen that running the AES hardware at 70MHz, a 64-entry cache can be successfully used too. The numerical data for the throughput obtained with that cache dimension is 200.69Mbit/s.

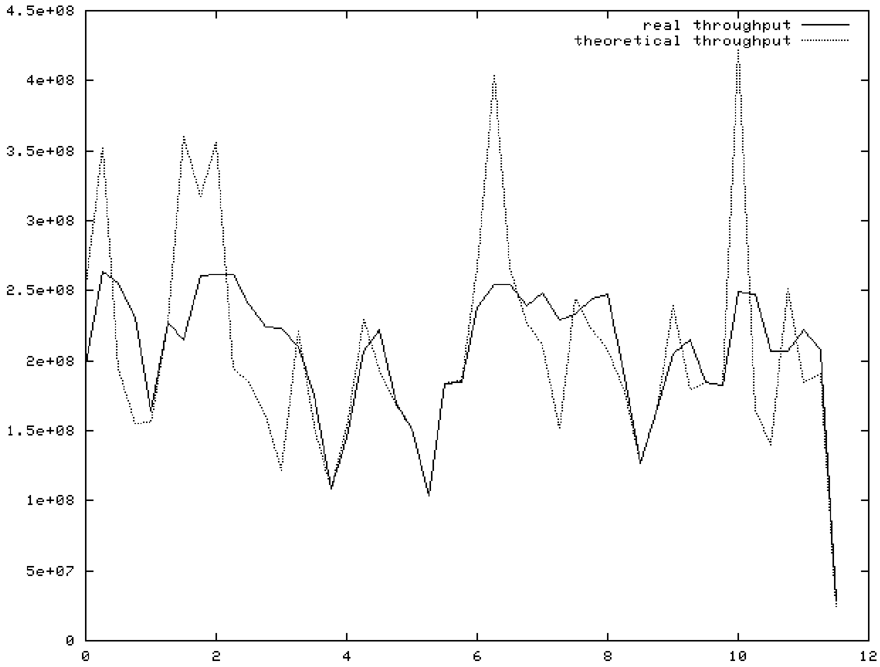


Figure 6.46: throughput obtained using a 64-entry cache and an AES hardware running at 70MHz

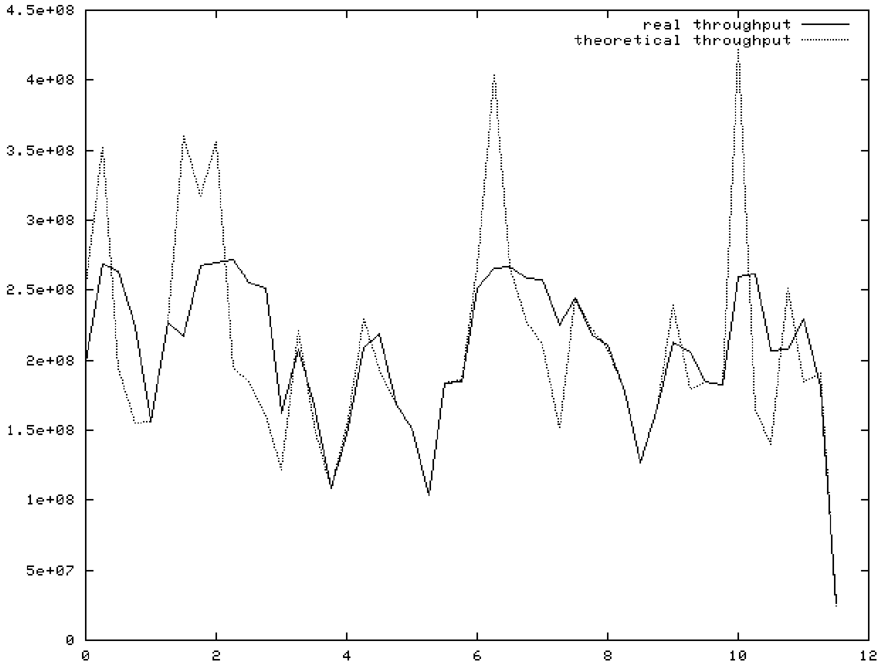


Figure 6.47: throughput obtained using a 128-entry cache and an AES hardware running at 70MHz

6.5.4.5. Conclusions about the results obtained with this simulation

From the results obtained in the previous sections, we can point out that the cache dimension we should use to obtain the required performance, strictly depends on the clock rate used for the AES hardware.

The AES clock and the cache dimension should be chosen also taking into account that a performance margin is needed for safely guarantee the results with respect to all the approximations we introduced writing the simulation.

Using a 64-entry cache with a 70MHz AES hardware seems not to be a good idea, both for the higher clock rate to be used and because using a 64-entry cache can left less margin for the SA creation process (as explained in section 6.4.7).

The better solutions seem to be two: the one that is composed by a 128-entry cache coupled with a 60MHz AES hardware or the one composed by a 256-entry cache coupled with a 60MHz AES hardware. Since the results obtained with those two cache dimensions are almost the same and that the former solution needs half the memory needed by the latter solution, the cache dimension to be chosen should be the first one.

6.5.4.6. Using a 30 min. timeout on the unused SAs

Using a timeout on the unused SAs as explained in section 6.4.4.2, does not change the results about the throughput presented above. As a matter of fact, that timeout is not useful here, because the simulation last for less than 12s. The timeout would have to be scaled by the same factor of the timestamps, but this would mean to set a timeout of only 2.9s, that would not have any meaning in a real system. As we saw in the previous sections, introducing that kind of SA closure policy does not have bad effects on cache misses (as a matter of fact it lower the number of avoidable misses), so it could not have bad effects on throughput here.

No results considering the TCP FIN packets are shown here, considering that solution not good from the point of view of SA reuse.

6.5.4.7. Considering some cache position reserved for SA creation

As done in section 6.4.7, we consider here the cache dimension slightly diminished for taking into account the cache positions that in a real system would be used for creating the SAs. The throughput obtained with a 108-entry cache (simulating a 128-entry cache with 20 busy cache positions) and a 60MHz AES hardware is shown in Figure 6.48. As we can see from that figure, the system is still able to support the desired throughput.

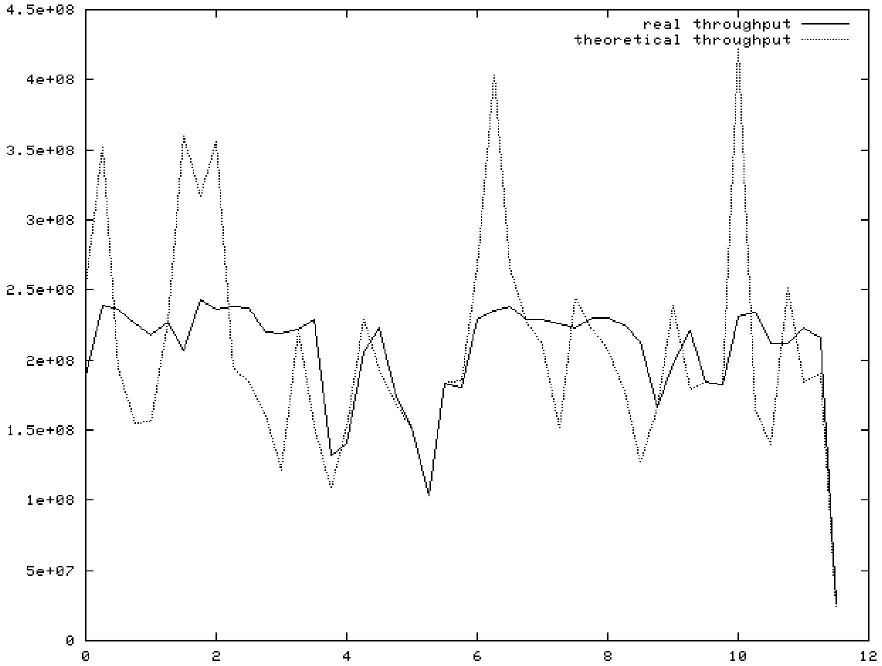


Figure 6.48: throughput obtained using a 108-entry cache and an AES hardware running at 60MHz

6.5.5. Considering the behavior of IKE Phase 2 in quick mode

In the previously shown simulations, we did not care about the fact that, when *IKE Phase 2 - quick mode* is applied, two SAs are created at the same time (see sections 3.4.2 and 6.4.8).

In this subsection we will show the results obtained running a slightly modified version of the simulation program. In that version of the simulation program, when a new SA is opened, another SA between the same two IP addresses but in the opposite direction is automatically opened too.

Also in this case, no information about the time needed for the creation of the security associations are introduced.

The results obtained here are pretty much the same as the ones shown before. A graphical representation of the results obtained considering a 60MHz AES hardware, is shown in Figure 6.49, Figure 6.50, and in Figure 6.51. In those figures a 128-entry cache, a 256-entry cache and a 108-entry cache are respectively considered. The throughput obtained for a 128-entry cache is 200.63Mbit/s. With a 256-entry cache we can obtain a throughput of 200.65Mbit/s.

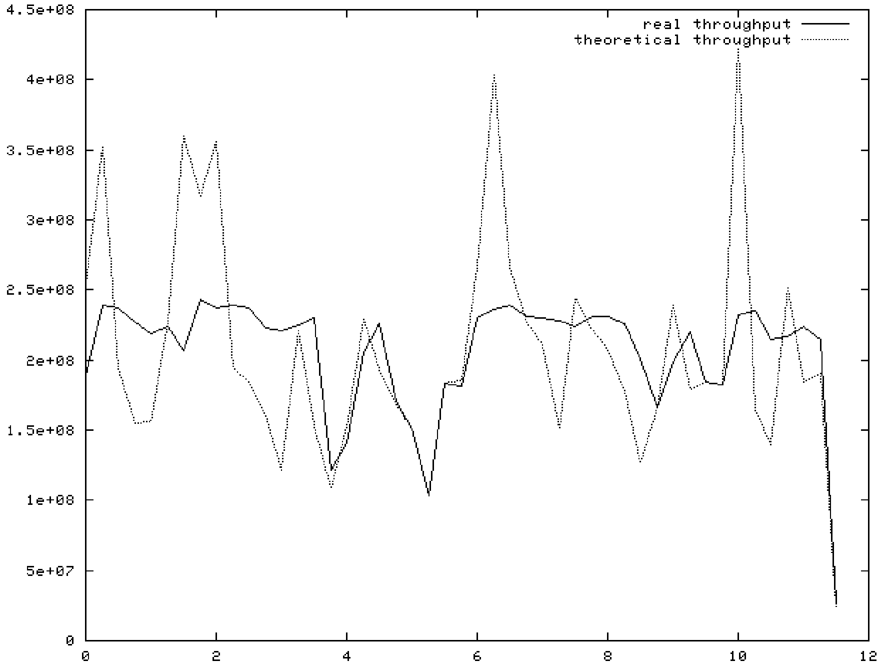


Figure 6.49: throughput obtained using a 128-entry cache and an AES hardware running at 60MHz – SA creation as in *IKE Phase 2 Quick Mode*

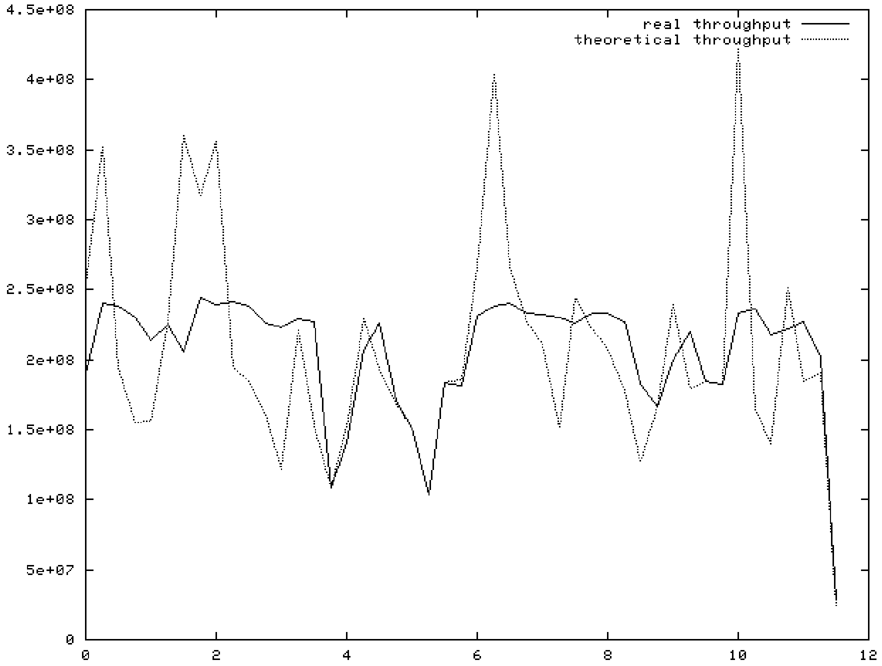


Figure 6.50: throughput obtained using a 256-entry cache and an AES hardware running at 60MHz – SA creation as in *IKE Phase 2 Quick Mode*

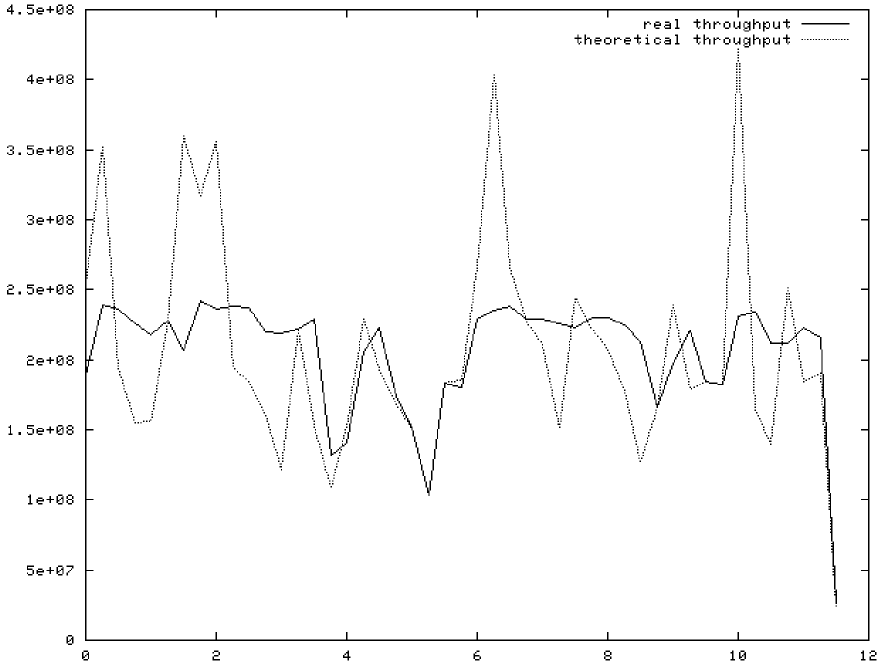


Figure 6.51: throughput obtained using a 108-entry cache and an AES hardware running at 60MHz – SA creation as in *IKE Phase 2 Quick Mode*

6.6. Adding the delays due to the SA creation phase

In this section we analyze the possibility to add some information about the delay introduced by the SA creation phase (IKE phase 1 and 2) to the simulation. The fact that the data we have do not contain any information about that phase (see section 6.2), makes the work done here to be less accurate than in the previous sections.

Considerations about the way the phase 1 of IKE is performed, lead us to consider the IKE phase 2 only. This means we are supposing that the IKE phase 1 has already been performed so that the ISAKMP SAs have already been created. Unfortunately, not having real data about the creation of the ISAKMP SAs, introducing guesses on the phase 1 would have introduced too much uncertainty without giving any (useful) additional information about cache usage. The ISAKMP SAs have a maximum lifetime of 24 hours (see [NIST-2]), therefore supposing the SAs to have already been created can be a not so bad choice. IKE phase 1 is based on public-key cryptography, a component of our system that does not use cache. In IKE phase 2 public-key cryptography is still used for Diffie-Hellman exchanges, but the cache is also used to store the “half generated” keys and the ISAKMP SAs information. With “half-generated key” we mean the Diffie-Hellman secret that must be stored while waiting for the other peer’s Diffie-Hellman payload.

In the following subsection a description of the delays that must be considered building this simulation is given.

6.6.1. Description of the delays introduced by the IPSec SA creation phase

6.6.1.1. Diffie-Hellman key generation delays

The delay introduced by this operation can be considered to be around 400µs for each of the two needed phases (the Diffie-Hellman secret generation and the key completion operations). The time needed for the ECC curve computation (assumed to be around 100µs) must be also added to that delay. In [CA-PO], better results are given for hardware-software implementations of the key generation algorithm, but only considering 155-bit-wide ECC keys. Here we consider a 600-bit-wide ECC key.

Therefore we consider the time needed for the key generation to be the longest time taken by the software implementation of the algorithm running with a 155-bit ECC key (400μs).

The key computation delay described above, should be considered twice for each SA creation process, being that for each *IKE phase 2 quick mode* process, a pair of independent SAs is created and that those two SAs have keys derived from different key material. The ECC curve computation must be considered only once for each of the two phases, since the two key generations previously discussed are performed on the same curve and one after the other.

6.6.1.2. Transferring the public-key material on the bus

Each time the command to generate a key is invoked, we suppose that the ECC parameters have to be set. Considering a 600-bit ECC key, we need to transfer 526 bytes on the data channel between the host and the crypto-processor (see Table 4.10). Basing on what stated in section 6.5.1.1, to transfer 526 bytes we need

$$1.515*10^{-8}*(1+1+132)=2*10^{-6}s$$

When the key-completion command is invoked, the delay introduced by transferring the key material received by the other peer should also be considered.

6.6.1.3. The network delays

The delays that need to be considered in this phase are composed not only by the ones introduced by the crypto-system (key generation, cache misses, …), but also by the network delays. As a matter of fact the SA creation phase is based on some exchanges between the two peers who are negotiating a pair of SAs as described in section 1.4.3. The minimum number of ask/reply exchanges that must be considered here are two (a message from the initiator to the responder, a reply, and another message from the initiator to the responder), since we are considering only the phase 2 of IKE. It can be easily noted that the delays introduced by the network are pretty much not predictable and are the ones that have the main impact on the SA creation time (they are higher than the ones introduced by the crypto-system, often by some order of magnitude).

As first step we analyzed the data file we have, to obtain some information about the time necessary for an ask/reply exchange between each of the peers. To do that we used the C program reported in Appendix D. In Figure 6.52 is reported a graph that displays the average reply time between each peer, while in Figure 6.53 the same graph is reported with the lowest part of the ordinates axe magnified.

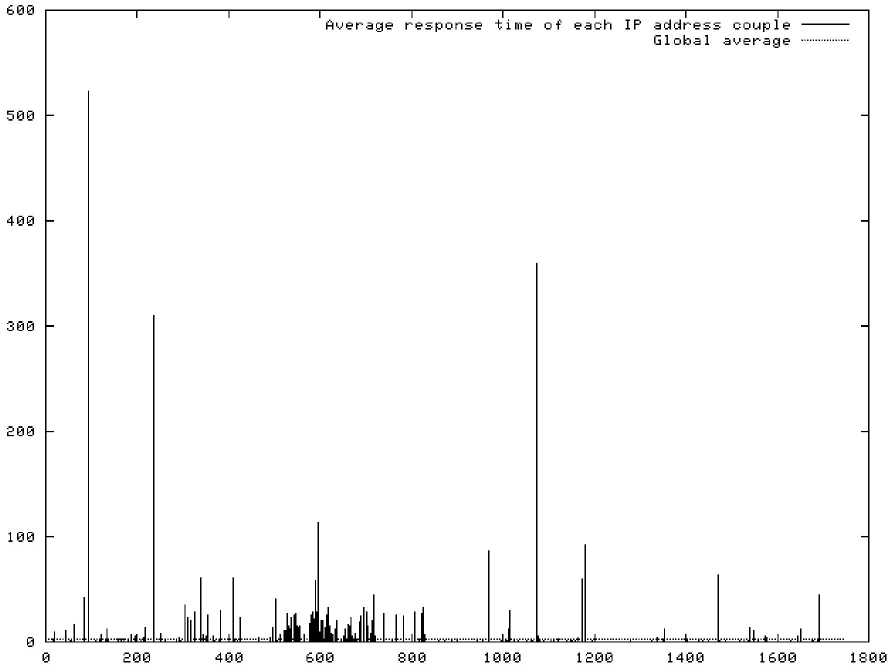


Figure 6.52: average reply time for each pair of peers

Using the same C program we was also able to obtain the minimum reply time between each of the peers as reported in Figure 6.54, and the maximum reply time between each of the peers as reported in Figure 6.55.

That data was obtained considering the time that intercourses between a data transmission from a source to a destination an the corresponding reply (a datagram going from the one that was previously the destination to the ones that was previously the source). When two datagrams going from a source to a destination without a reply between them are found, the oldest datagram's timestamp is thrown away. We have to note that there are 101 peers (over around 1,800) which do not have any time statistic, this is due to the fact that the communication pattern we considered never occurs for that peers.

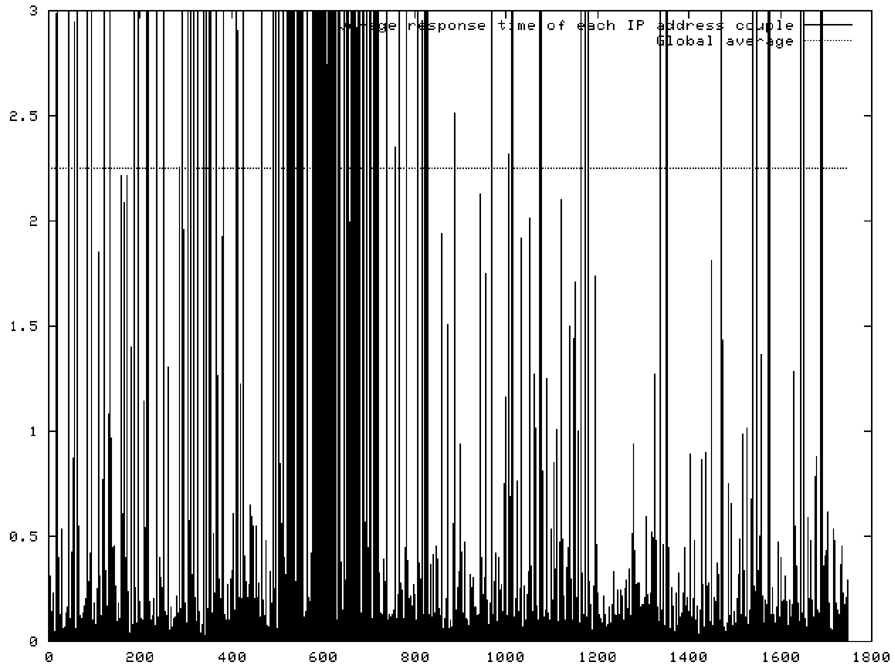


Figure 6.53: average reply time between each pair of peers with the part of the ordinate axe between 0 and 3 magnified

The average reply time obtained is 2.2s.

These network delays was obtained without scaling the timestamps with any factor. From our stand point, scaling the network delays is not correct, and using the values as they are is not correct too. More than that, deciding what kind of values to use can be hard. As a matter of fact using the average values can be a good choice, but being this a worst case estimation, we should use the maximum ones. Any choice we did would led us to introduce enormous delays having an uncertainty that is often higher than the delays introduced by the whole crypto-system. We are talking about delays of some seconds (around 4s in average) in a system that can process 1.8 million of datagrams in les than 12s! We have also to remember that we are working on a model, and adding that delays would change the system behavior, leading us far from the indication given by the real timestamps we have in the data file.

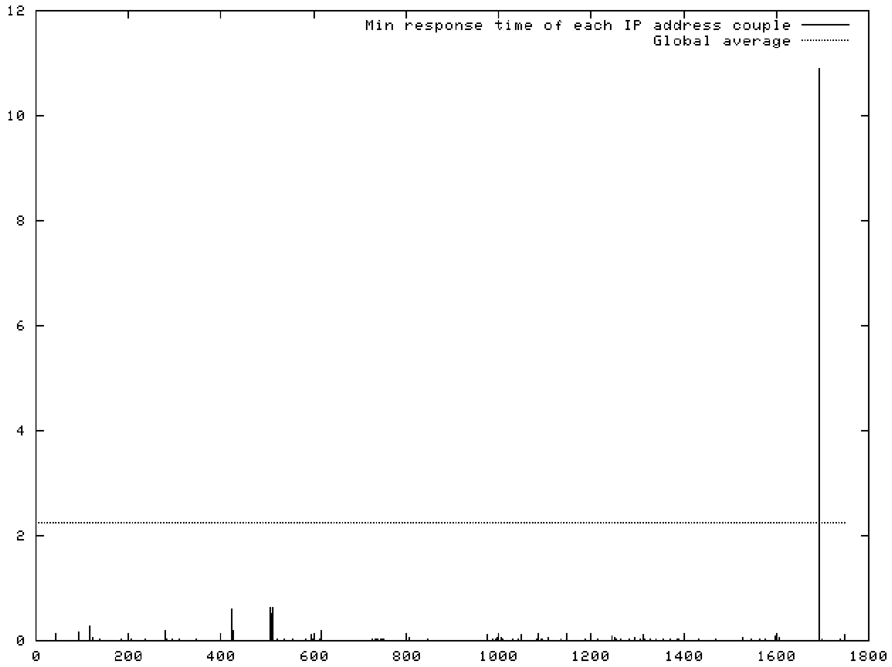


Figure 6.54: minimum reply time between each pair of peers

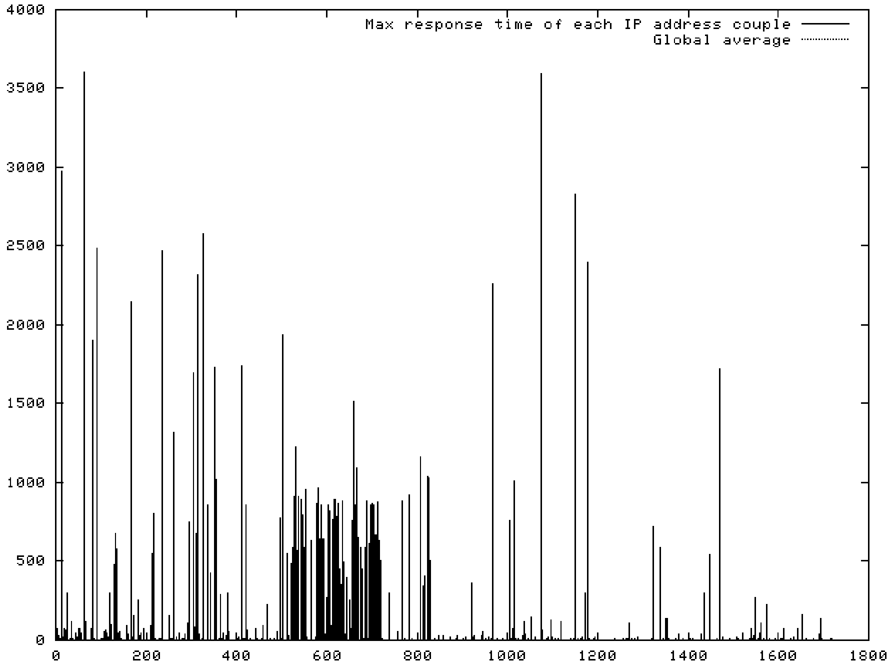


Figure 6.55: maximum reply time between each pair of peers

6.6.2. Conclusions about the SA creation delays

As stated before, the main delays that must be introduced for considering the SA creation phase, strictly depend on unpredictable events, such as the network communications. Moreover, thinking about how the simulation should work, it can be pointed out that the datagrams would need to be reordered to keep the simulated system running while the SAs are created. Not reordering the packets would mean that, for each SA-pair creation, the system would stop for around 4s waiting for the key generation process to be completed. In both cases we would consider a system behavior far from reality. As a matter of fact in the first case the datagram reordering would dramatically increase the temporal locality of the packets, lowering the number of avoidable cache misses; in the latter case we would have a simulation reproducing a system stalling at each SA-pair creation.

From our stand point the only way for obtaining a reasonable simulation including the SA creation phase, would be to take some data from a real IPSec system. Using such data, would also allow to study the effects of a SA closure policy on the performance of the system. As a matter of fact, when a SA is closed and needs to be reopened, the SA creation mechanism (IKE phase 2) needs to be used.

The previous considerations lead us to decide not to write any further simulation based on the available data.

6.7. Technical details related to the simulations

The simulation programs was compiled using the GNU gcc compiler (2.96 modified by Red Hat) under Red Hat Linux 7.2 (with a 2.4.18 kernel). The only gcc option used was *"-O3"* that makes the compiler perform the best possible code optimization for speed it can do.

The PC used to run the simulations is based on an AMD Athlon XP1700+ (1.46GHz) processor. Each simulation run described in section 6.4 takes around 4 seconds; the ones described in section 6.5 take around 7 seconds each.

All the graphs shown in this chapter were obtained using the *gnuplot* tool.

6.8. Choiching the optimal cache dimension

From the results shown in the previous sections, we can state that the desired throughput can be obtained using a 128-entry or a 256-entry cache. Obviously, those performance are provided only using the right clock rate for the AES hardware. Although some indications about the values to be used for that parameter are shown here, the study of its optimal value is beyond the scope of this document.

Also changing the other system's parameters may impact on the optimal cache dimension to be chosen. For example, considering a CRC hardware two order of magnitude slower than the considered one would lead us point out that also the 256-entry cache is not enough to support the desired throughput.

For choosing the right cache dimension, all the system parameters have to be considered, and a performance margin should be taken, considering all the approximations made writing the simulations. Using the parameters shown in the previous sections, the optimal cache dimension seems to be of 128 entries. That solution does not guarantee the best reuse of the cache entries before discard, but it seems not to provide so different performance from the 256-entry cache, while it allows to save of 16kb of memory on the crypto-processor.

6.9. Results validation

With "validation" we mean the use of a different set of data to verify the obtained results, as normally done in the model identification field. As stated before we have available another (shorter) set of data taken from the same system in a different daytime ([ITA-2]). Using that set of data, the results obtained are pretty much the same as the ones obtained with the other data file. In Figure 6.56 is reported (for example) the throughput obtained with a 128-entry cache and a 60MHz AES hardware. In that case the required average throughput is of 210.9Mbit/s, as the obtained one. The throughput that can be reached here is higher than in the previous cases because the considered traffic has a slightly different shape. Less but bigger datagrams per second are here processed. With this kind of traffic the AES hardware is able to work in better conditions allowing a higher throughput.

A further validation of the conducted studies should be done using sets of data taken from other systems.

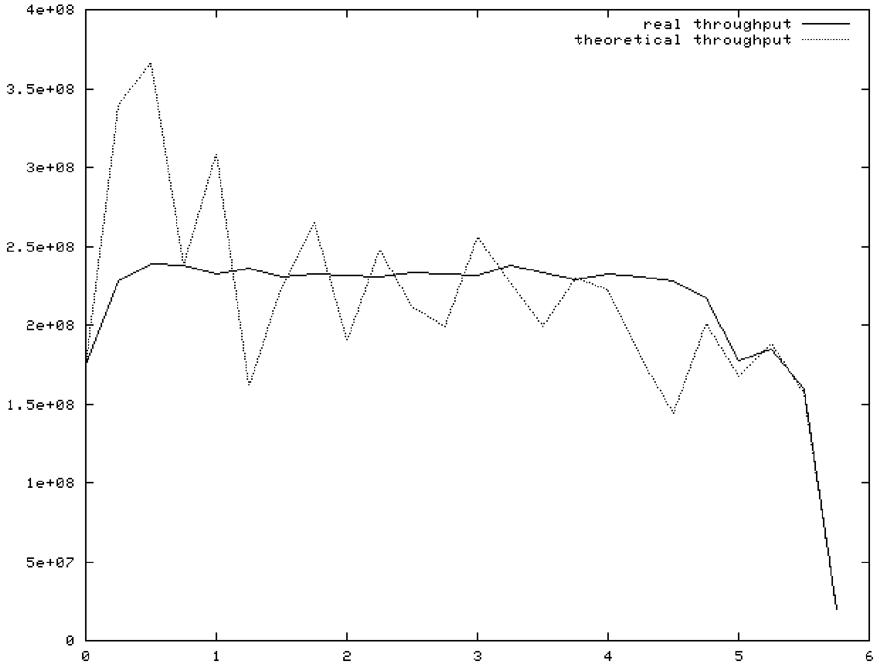


Figure 6.56: throughput obtained using a 128-entry cache and an AES hardware running at 60MHz with a different set of data

7. Conclusions

Defining the security policy and the smart card software interface is a good starting point for future development of the project. Furthermore, defining the smart card software interface lead us to develop a quite deep understanding of the IKE and IPSec protocols.

Studying the data taken from a real system, we were also able to understand the dimension of the memory needed in a high-performance crypto-processor to support IPSec in a high throughput system. Some data about the SA closure policy to be used were also obtained.

8. Possible future improvements of the system

8.1. Testing and verification of the software interface

The smart card software interface described here need to be tested in a real IPSec implementation, mainly to verify that the provided commands allow the system to work in the right way. This can be done only when both a complete software model of the smart card and an IPSec implementation based on that model will be ready.

That test would also allow to discover possible interoperability problems with other IPSec implementations, for example conducting the on-line test provided on the NIST website ([NIST-1]).

8.2. Further studies about the SA cache

Further studies about the crypto-processor’s SA-database can be conducted using data taken from real IPSec systems. In that way we would be able to see the effects of the SA creation phase on the system’s performance and on the cache usage.

More precise data about cache misses can be also obtained running the same simulation program we used on data taken from a real 200Mbit/s system. This would allow to run the simulation without scaling the timestamps, so that less approximations were introduced in the results.

Further studies should also be done on the behavior of the SA cache with respect to the SA creation process.

Bibliography

[BOR] Michael S. Borella – 3Com Corp., *Methods and Protocols for Secure Key Negotiation Using IKE*, IEEE Network, July/August 2000

[CA-PO] F. Cassoli, F. Polloni, *Public-key Exchange Implementation*, ALaRI Master technical report, July 2001

[COM-1] *Security & Privacy*, a supplement to the April 2002 issue of IEEE Computer

[COM-2] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, S. Weingart, *Building the IBM 4758 Secure Coprocessor*, IEEE Computer, October 2001, pp. 57-66

[DEIT] H.M. Deitel, P.J. Deitel, *C++ How to program*, Prentice Hall, 1994

[DRAFT-1] S. Blake-Wilson, Y. Poeleuev, M. Salte, *Additional ECC Groups for IKE*, www.ietf.org, March 2001 – expires September 2001

[DRAFT-2] S. Frankel, S. Kelly, R. Glenn – *The AES Cipher Algorithm and Its Use With IPsec*, www.ietf.org, November 2000 – expires May 2001

[DRAFT-3] S. Frankel, S. Kelly, R. Glenn – *The AES Cipher Algorithm and Its Use With IPsec*, www.ietf.org, November 2001 – expires May 2002

[DRAFT-4] National Institute of Standards and Technology, *Secure Hash Standard*, www.nist.gov, 2001

[FSWAN] J. Gilmore, H. Spencer, R. Guy Briggs, H. Redelmeier, S. Harris, C. Schmeing, H. Daniel – *Linux FreeS/Wan*; www.xs4all.nl/~freeswan/download.html, 2000

[FUG] Alfonso Fuggetta, *WebBook*, www.cefriel.it/~alfonso, 2001

[IEEE-1] R. Perlman, C. Kaufman – *Key exchange in IPSec: Analysis of IKE*, IEEE Internet Computing, Nov-Dec 2000

[IEEE-2] A.D. Keromytis, J. Ioannidis, J. M. Smith – *Implementing IPSec*, IEEE, 1997

[ITA-1] Internet Traffic Archive, ita.ee.lbl.gov/html/contrib/LBL-TCP-3.html, 2002

[ITA-2] Internet Traffic Archive, ita.ee.lbl.gov/html/contrib/LBL-PKT.html, 2002

[MAC-MAR] M. Macchetti, S. Marchesin, *AES algorithm analysis, implementation an profiling*, ALaRI Master technical report, July 2001

[MOV] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of applied cryptography*, CRC press, 1996

[NIST-1] R. Glenn, S. Frankel, D. Montgomery – *IPSec-WIT: the NIST IPSec Web-based Interoperability Test System*, www.nist.gov

[NIST-2] *NIST PlutoPlus, An IKE Reference Implementation for Linux*, www.antd.nist.gov/plutoplus/

[P-H] D.A. Patterson, J.L. Hennessy, *Struttura e progetto dei calcolatori*, Zanichelli, 1995

[POM] O. Pomerants, *Linux kernel module programming guide*, www.doc-linux.co.uk/LDP/LDP/lkmpg, 1999

[RFC-2401] S. Kent, *Security architecture for the internet protocol*, www.ietf.org/rfc/rfc2401.txt, November 1998

[RFC-2402] R. Atkinsons, *IP Authentication Header*, www.ietf.org/rfc/rfc2402.txt, November 1998

[RFC-2406] R. Atkinsons, *IP Encapsulating Security Payload*, www.ietf.org/rfc/rfc2406.txt , November 1998

[RFC-2407] D. Piper, *The internet IP security domain of interpretation for ISAKMP*, November 1998; www.ietf.org/rfc/rfc2407.txt

[RFC-2408] D. Maughan, M. Schertler et al., *Internet security association and key management protocol (ISAKMP)*, www.ietf.org/rfc/rfc2408.txt, November 1998

[RFC-2409] D. Carrel, *The internet key exchange (IKE)*, www.ietf.org/rfc/rfc2409.txt, November 1998

[RFC-2451] R. Pereira, *The ESP CBC-Mode Cipher Algorithms*, www.ietf.org/rfc/rfc2451.txt, November 1998

[RHS] R. Schroeppel, H. Orman, S. O'Malley, *Fast key exchange with elliptic curve systems*, citeseer.nj.nec.com/schroeppel95fast.html, 1995

[TAN] Andrew S. Tanenbaum, *I moderni sistemi operativi*, Prentice Hall International, 1994

[VAS] Alavoor Vasudevan, *C++ programming HOW-TO*, www.linuxdoc.org/HOWTO/C++Programming-HOWTO.html, 2001

Appendix A. The smart card interface code

A.1. sC_driver.h

```
//sC_driver.h
// header definition of the smart card software interface for IPSec

//made by Alberto Ferrante
//May-June 2001

#include<limits.h>

#ifndef __SC_DRIVER__
#define __SC_DRIVER__

//maximum number of slots available into the S/C
#define MAX_SC_SLOTS 16
//maximum number of SA available = MAX_SC_SLOTS*40
#define MAX_SA MAX_SC_SLOTS*40
//length of the S/C command queue
#define MAX_SC_QUEUE 10


//smart card command code definitions (8 bits for each command)
#define LOGIN                        1
#define REFRESH_SESSION_KEY          2
#define RESET_SC                     3
#define TEST_SC                      4
#define READ_SC_STATUS               5
#define SET_SA_STATUS                6
#define GET_SA_PARAMETERS            7
#define GEN_DH                       8
#define COMPLETE_DH                  9
#define DELETE_SA                    10
#define SYMM_DECRYPT                 11
#define SYMM_ENCRYPT                 12
#define SET_ECC_INFO                 13
#define GET_ECC_INFO                 14
#define GET_PUB_KEY                  15
#define PUBLIC_ENCRYPT               16
#define PUBLIC_DECRYPT               17
#define HASH                         18
#define GEN_SYMM_SIGN                19
#define GEN_ECDSA_SIGNATURE          20
#define VERIFY_ECDSA_SIGNATURE       21

//driver command code definitions (8 bits for each command)
#define SC_ERROR              22
#define LOGIN_RESULTS         23
#define TEST_RESULTS          24
#define SC_STATUS             25
#define SA_PARAMETERS         26
#define RANDOM_DH             27
#define SYMM_DECRYPTED_P      28
#define SYMM_ENCRYPTED_P      29
#define ECC_INFO              30
#define ECC_KEY               31
#define ECC_ENCRYPTED         32
#define ECC_DECRYPTED         33
#define HASH_RESULTS          34
#define SYMM_SIGN             35
#define ECDSA_SIGNATURE       36
#define ECDSA_CHECK_RES       37
#define CONFIRMATION          38

//error codes
#define GENERIC                      1
#define BAD_SA_INDEX                 2
#define WRONG_AES_PACKET             4
#define TAMPERED_SA_INFORMATION      5
```

```
#define WRONG_DH_NUMBER              6
#define AES_PARAMETERS_NOT_SET       7
#define WRONG_ECC_INFO               8
#define ECC_INFO_NOT_SET             9
#define WRONG_PRD_PARAMETER          10
#define WRONG_SYMM_SIGN_PARAM        11
#define WRONG_SIGNATURE_PARAM        12
#define LOGIN_ALREADY_DONE           13
#define CANNOT_REFRESH_S_KEY         14

//hashfcn parameter
#define HMACMD5              1
#define HMACSH1              2

//methods error codes
#define OK            0
#define GEN_ERROR     -1
#define TOO_OPENED    -2
#define SC_QUE_FULL   -3

struct Info{
       unsigned int sa; //sa number into the SADB
       unsigned int slot; //sc slot in which the SA has been allocated, if any
       unsigned int* saved; //space for saving info for swapping
};
typedef Info saInfo;



struct  Slot{
       saInfo* posArray;
       unsigned int usage;
};
typedef Slot scSlot;


class sC_driver {
       public:
              sC_driver(iPSec ipsec, SmartCard SC); //constructor
              ~sC_driver(); //destructor

              int ip_rcv(unsigned int length, unsigned int* data);
              int login(unsigned int pin);
              int reset_sc();
              int test_sc(unsigned int puk);
              int genDH(unsigned int sa, unsigned int key_dim, unsigned int mode,
                            unsigned int rounds);
              int compDH(unsigned int sa, unsigned int key_dim, unsigned int* dh);
              int deleteSA(unsigned int number);
              int symmDecrypt(unsigned int sa, unsigned int length, unsigned int* data,
                            unsigned int hashf, unsigned int sigf, unsigned int ivf);
              int symmEncrypt(unsigned int sa, unsigned int length, unsigned int* data,
                            unsigned int hashf, unsigned int sigf, unsigned int ivf);
              int setECCInfo(unsigned int n, unsigned int* A, unsigned int* B,
                            unsigned int* x, unsigned int* y);
                            //overloaded method: version without the key parameter
              int setECCInfo(unsigned int n, unsigned int* A, unsigned int* B, unsigned int* x,
                            unsigned int* y, unsigned int* key);//version with the key parameter
              int getPublicKey();
              int pubEncrypt(unsigned int length, unsigned int* data);
              int pubDecrypt(unsigned int length, unsigned int* data);
              int hash(unsigned int length1, unsigned int length2, unsigned int* data1,
                            unsigned int* data2, unsigned int hashfcn);
              int genSymmSign(unsigned int sa, unsigned int length, unsigned int* data1,
                            unsigned int* data2, unsigned int hashfcn);
              int genECDSASign(unsigned int length, unsigned int* data);
              int verifyECDSASig(unsigned int length, unsigned int* data);


       private:
              SmartCard SC;
              iPSec ipsec; /*       the ipsec process that inizialized the driver;
                                    if there are more than one IPSec instances,
                                    it will be necessary to store it into the saInfo
                                    structure so that each opened SA has an associated
                                    IPSec instance. In that case, the genDH method
                                    should also have the pointer to the IPSEc object
                                    as parameter and the ip_rcv method should search
```

```
                                        into the saArray array the object to use.
                                */

                scSlot sc[MAX_SC_SLOTS];
                saInfo saArray[MAX_SA];

                unsigned int opened_dev;
                static unsigned int istantiated=0;

                unsigned int freeSlots;
                unsigned int freeSa;
                unsigned int freeQue;

                unsigned int saved_n;
                unsigned int* saved_A;
                unsigned int* saved_B;
                unsigned int* saved_x;
                unsigned int* saved_y;
                unsigned int* saved_key;

                int firstFreeSlot();
                saInfo* firstFreeSa();
                void updateUsage();
                unsigned int findLeastUsed();
                unsigned int swap();
                saInfo* findSa(unsigned int sa);
                int incOpened();
                void decOpened();
};


#endif //__SC_DRIVER__
```

A.2. sC_driver.cpp

```
//sC_driver.cpp
// smart card software interface for IPSec


//made by Alberto Ferrante
//May-June 2001

#include <limits.h>
#include <math.h>
#include <stdlib.h>
#include "sC_Driver.h"

#ifndef NDEBUG
#include <assert.h>
#endif;


sC_driver::sC_driver(iPSec ipsec, SmartCard SC){
//constructor
//pre: ipsec!=NULL && SC!=NULL && istantiated == 0
//post: this.ipsec!=NULL && this.SC!=NULL && istantiated == 1

        #ifndef NDEBUG
                assert(ipsec!=NULL);
                assert(SC!=NULL);
                assert(istantiated ==0);
        #endif;


        unsigned int i=0;

        if (istantiated == 0){
                istantiated++;
                opened_dev=0;
                freeSlots=MAX_SC_SLOTS;
                freeSa=MAX_SA;
                freeQue=MAX_SC_QUEUE;

                this.ipsec=ipsec;
                this.SC=SC;
                for (i=0; i<MAX_SA; i++){
                        saArray[i].scSlot=MAX_SC_SLOTS;
                        saArray[i].sa=0;
                }

                for (i=0; i<MAX_SC_SLOTS; i++){
                        sa[i].posArray=NULL;
                        sa[i].usage=0;
                }

                current=0;
        }

        #ifndef NDEBUG
                assert(this.ipsec!=NULL);
                assert(this.SC!=NULL);
                assert(istantiated == 1);
        #endif;
}


sC_driver::~sC_driver(){
//destructor
//pre: none
//post: none
}


int sC_driver::ip_rcv(unsigned int length, unsigned int* data){
//method for receiving data from the S/C
//pre: length>0 && data!=NULL
//post: none

        #ifndef NDEBUG
                assert(length>0);
                assert(data!=NULL);
        #endif;

        unsigned int command=data[0]>>24;

        switch(command){
                case SC_ERROR:
                        ipsec.error(data[0]&0x000000FF);
                        break;

                case LOGIN_RESULTS:
                        ipsec.loginResults(data[0]&0x00000700>>8, data[0]&0x000000F0>>4,
                                        data[0]&0x00000001);
                                        //counter, status code, result
                        break;

                case TEST_RESULTS:
                        //still to be defined!!!
                        break;

                case SC_STATUS:
                        //unused so far
                        break;

                case SA_PARAMETERS:
                        //unused so far
                        break;

                case RANDOM_DH:
                        ipsec.randomDH(data[0]&0x000000FF, data+1); //SA index, data
                        break;

                case SYMM_DECRYPTED_P:
                        ipsec.symmDecrypted(data[0]&0x000000FF, data+1); //SA index, data
                        break;

                case SYMM_ENCRYPTED_P:
                        ipsec.symmEncrypted(data[0]&0x000000FF, data+1); //SA index, data
                        break;

                case ECC_INFO:
                        //unused so far
                        break;
```

```
               case ECC_KEY:
                      ipsec.ECCKey(data[0]&0x000000FF, (data[0]&0x00FFFF00)>>8, data+1);
                                     //length, key length (bits), data
                      break;

               case ECC_ENCRYPTED:
                      ipsec.ECCEncrypted(data[0]&0x000000FF, data+1); //length, data
                      break;

               case ECC_DECRYPTED:
                      ipsec.ECCDecrypted(data[0]&0x000000FF, data+1); //length, data
                      break;

               case HASH_RESULTS:
                      ipsec.hashResults(data[0]&0x000000FF, data+1); //length, data
                      break;

               case SYMM_SIGN:
                      ipsec.symmSign(data[0]&0x000000FF, data+1); //length, data
                      break;

               case ECDSA_SIGNATURE:
                      ipsec.ECDSASignature(data[0]&0x000000FF, data+1); //length, data
                      break;

               case ECDSA_CHECK_RES:
                      ipsec.ECDSACheckRes(data[0]&0x000000FF, data+1); //length, data
                      break;

               case CONFIRMATION:
                      ipsec.confirmation(data[0]&0x000000FF);
                      break;

               default:
                      break;
        }
        return OK;
}


int sC_driver::login(unsigned int pin){
//method for calling the LOGIN command
//pre: none
//post: none

        if(incOpened()==1){
               unsigned int code=0;

               code=(((unsigned int)LOGIN)<<24)|pin;
               if (freeQue==0) return SC_QUE_FULL; else freeQue--;
               SC.sc_rcv(1, &code);
               freeQue++;

               decOpened();
               return OK;
        }else return TOO_OPENED;
}



int sC_driver::reset_sc(){
//method for calling the RESET command
//pre: none
//post: none

        if(incOpened()==1){
               unsigned int code=0;

               code=(((unsigned int)RESET_SC)<<24);
               if (freeQue==0) return SC_QUE_FULL; else freeQue--;
               SC.sc_rcv(1, &code);

               freeQue++;

               decOpened();
               return OK;
        }else return TOO_OPENED;
}
```

```
int sC_driver::test_sc(unsigned int puk){
//method for calling the TEST_SC command
//pre: none
//post: none

        if(incOpened()==1){
                unsigned int code=0;

                code=(((unsigned int)TEST_SC)<<24)|puk;
                if (freeQue==0) return SC_QUE_FULL; else freeQue--;
                SC.sc_rcv(1, &code);
                freeQue++;

                decOpened();
                return OK;
        }else return TOO_OPENED;
}
```

```
int sC_driver::genDH(unsigned int sa, unsigned int key_dim, unsigned int mode, unsigned int rounds){
//method for calling the GENDH command
//pre: sa!=0 && (key_dim==128||key_dim==192||key_dim==256) && rounds <15
//post: none


        #ifndef NDEBUG
                assert(sa!=0);
                assert(key_dim==128||key_dim==192||key_dim==256);
                assert(rounds<15);
        #endif;

        if(incOpened()==1){
                int slotnum;
                unsigned int code;
                unsigned int key=0;

                switch(key_dim){ //the key dimension is transformed in the corresponding code
                        case 196:
                                key=1;
                                break;
                        case 256:
                                key=2;
                                break;
                        default:
                                key=0;
                                break;
                }

                if (freeSa>0){
                        freeSa--;
                        if(freeSlots>0){
                                freeSlots--;
                                slotnum=firstFreeSlot();
                        }else{
                                if (freeQue==0) return SC_QUE_FULL; else freeQue--;
                                                        //verifies if it is possible to perform the swap
                                slotnum=swap();
                                freeQue++;
                        }
                        sc[slotnum].posArray=firstFreeSa();
                        (sc[slotnum].posArray)->slot=slotnum;
                        updateUsage();
                        sc[slotnum].usage=UINT_MAX;
                        code=(((unsigned int)GEN_DH)<<24)|(mode<<20)|(rounds<<16)
                                |(key<<8)|(unsigned int)slotnum;
                        if (freeQue==0) return SC_QUE_FULL; else freeQue--;
                        SC.sc_rcv(1, &code);
                        freeQue++;

                        decOpened();
                        return OK;
                }
```

```
               else{
                      decOpened();
                      return GEN_ERROR;
               }
       }else return TOO_OPENED;
}


int sC_driver::compDH(unsigned int sa, unsigned int key_dim, unsigned int* dh){
//method for calling the COMPLETE_DH command
//pre: sa!=0 && (key_dim==128||key_dim==192||key_dim==256) & dh!=NULL
//post: none

       #ifndef NDEBUG
               assert(sa!=0);
               assert(key_dim==128||key_dim==192||key_dim==256);
               assert(dh!=NULL);
       #endif;

       if(incOpened()==1){
               saInfo* pos=NULL;
               unsigned int slotnum=0;
               unsigned int length=(unsigned int)ceil(((float)key_dim)/32);
               unsigned int* payload=new(unsigned int[length+1]);
               unsigned int i=0;
               unsigned int key=0;

               switch(key_dim){ //the key dimension is transformed in the corresponding code
                      case 196:
                             key=1;
                             break;
                      case 256:
                             key=2;
                             break;
                      default:
                             key=0;
                             break;
               }

               if((pos=findSa(sa))!=NULL){
                      if((slotnum=pos->slot)==MAX_SC_SLOTS){
                                           //the sa is not in the S/C and there is a free slot
                             if(freeSlots>0){
                                    freeSlots--;
                                    slotnum=firstFreeSlot();
                             }
                             else{  //the sa is not in the S/C performs a swap and there are
                                    //no free slots
                                    if(freeQue==0) return SC_QUE_FULL; else freeQue--;
                                                  //verify if it is possible to perform the swap
                                    slotnum=swap();
                                    freeQue++;
                             }
                             sc[slotnum].posArray=pos;
                             pos->slot=slotnum;
                      }
                      updateUsage();
                      sc[slotnum].usage=UINT_MAX;

                      //prepares the data to send to the S/C
                      payload[0]=(((unsigned int)COMPLETE_DH)<<24)|(key<<8)|(unsigned int)slotnum;
                      for(i=0;i<length;i++) payload[i+1]=dh[i];

                      if(freeQue==0) return SC_QUE_FULL; else freeQue--;
                      SC.sc_rcv(length+1, payload);
                      freeQue++;

                      decOpened();
                      return OK; //operation correctly executed
               }
               else {
                      decOpened();
                      return GEN_ERROR;
               }
       }else return TOO_OPENED;
}
```

```
int sC_driver::deleteSA(unsigned int sa){
//method for calling the DELETESA command
//pre: sa!=0
//post: none

        #ifndef NDEBUG
                assert(sa!=0);
        #endif;

        if(incOpened()==1){
                saInfo* pos=NULL;
                unsigned int code=0;

                if((pos=findSa(sa))!=NULL){
                        if(pos->slot<MAX_SC_SLOTS){
                                code=(((unsigned int)DELETE_SA)<<24)|(unsigned int)pos->slot;
                                if (freeQue==0) return SC_QUE_FULL; else freeQue--;;
                                SC.sc_rcv(1, &code);
                                freeQue++;

                                sc[pos->slot].usage=0;
                                freeSlots++;
                        }
                        pos->slot=MAX_SC_SLOTS;
                        pos->sa=0;
                        delete(pos->saved);
                        freeSa++;
                }
                return OK;
        }else return TOO_OPENED;
}



int sC_driver::symmDecrypt(unsigned int sa, unsigned int length, unsigned int* data, unsigned int
hashf,
                          unsigned int sigf, unsigned int ivf){
//method for calling the SYMMDECRYPT command
//pre: sa!=0 && length>0 && data!=NULL
//post: none

        #ifndef NDEBUG
                assert(sa!=0);
                assert(length>0);
                assert(data!=NULL);
        #endif;

        if(incOpened()==1){
                saInfo* pos=NULL;
                unsigned int slotnum=0;
                unsigned int* payload=new(unsigned int[length+2]);
                unsigned int i=0;

                if((pos=findSa(sa))!=NULL){
                        if((slotnum=pos->slot)==MAX_SC_SLOTS){
                                                //the sa is not in the S/C and there is a free slot
                                if(freeSlots>0){
                                        freeSlots--;
                                        slotnum=firstFreeSlot();
                                }
                                else{   //the sa is not in the S/C performs a swap and there are
                                        //no free slots
                                        if(freeQue==0) return SC_QUE_FULL; else freeQue--;
                                                //verify if it is possible to perform the swap
                                        slotnum=swap();
                                        freeQue++;
                                }
                                sc[slotnum].posArray=pos;
                                pos->slot=slotnum;
                        }
                        updateUsage();
                        sc[slotnum].usage=UINT_MAX;

                        //prepares the data to send to the S/C
                        payload[0]=(((unsigned int)SYMM_DECRYPT)<<24)|(unsigned int)slotnum;
                        payload[1]=hashf<<24|sigf<<23|ivf<<16|length;
                        for(i=0;i<length;i++) payload[i+2]=data[i];
                        if(freeQue==0) return SC_QUE_FULL; else freeQue--;
                        SC.sc_rcv(length+2, payload);
                        freeQue++;

                        decOpened();
                        return OK; //operation correctly executed
                }
                else {
                        decOpened();
                        return GEN_ERROR;
                }
        }else return TOO_OPENED;

}



int sC_driver::symmEncrypt(unsigned int sa, unsigned int length, unsigned int* data,
                                unsigned int hashf, unsigned int sigf, unsigned int ivf){
//method for calling the SYMMENCRYPT command
//pre: sa!=0 && length>0 && data!=NULL
//post: none


        #ifndef NDEBUG
                assert(sa!=0);
                assert(length>0);
                assert(data!=NULL);
        #endif;

        saInfo* pos=NULL;
        unsigned int slotnum=0;
        unsigned int* payload=new(unsigned int[length+2]);
        unsigned int i=0;

        if(incOpened()==1){
                if((pos=findSa(sa))!=NULL){
                        if((slotnum=pos->slot)==MAX_SC_SLOTS){
                                if(freeSlots>0){ //the sa is not in the S/C and there is a free slot
                                        freeSlots--;
                                        slotnum=firstFreeSlot();
                                }
                                else{   //the sa is not in the S/C performs a swap and there are
                                        //no free slots
                                        if (freeQue==0) return SC_QUE_FULL; else freeQue--;
                                                //verify if it is possible to perform the swap
                                        slotnum=swap();
                                        freeQue++;
                                }
                                sc[slotnum].posArray=pos;
                                pos->slot=slotnum;
                        }
                        updateUsage();
                        sc[slotnum].usage=UINT_MAX;

                        //prepares the data to send to the S/C
                        payload[0]=(((unsigned int)SYMM_ENCRYPT)<<24)|(unsigned int)slotnum;
                        payload[1]=hashf<<24|sigf<<23|ivf<<16|length;
                        for(i=0;i<length;i++) payload[i+2]=data[i];

                        if (freeQue==0) return SC_QUE_FULL; else freeQue--;
                        SC.sc_rcv(length+2, payload);
                        freeQue++;

                        decOpened();
                        return OK; //operation correctly executed
                }
                else{
                        decOpened();
                        return GEN_ERROR;
                }
        }else return TOO_OPENED;

}
```

```
int sC_driver::setECCInfo(unsigned int n, unsigned int* A, unsigned int* B, unsigned int* x, unsigned
int* y){
                                             //n is the length in BITS
//method for calling the SETECCINFO command
//pre: n>0 && A!=NULL && B!=NULL && x !=NULL && y!=NULL
//post: none

        #ifndef NDEBUG
                assert(n>0);
                assert(A!=NULL);
                assert(B!=NULL);
                assert(x!=NULL);
                assert(y!=NULL);
        #endif;

        if(incOpened()==1){
                unsigned int curveLen=(unsigned int)ceil(((float)n)/32);
                unsigned int baseLen=(unsigned int)ceil(((float)n)*2/32);
                unsigned int payLen=curveLen*2+baseLen*2+1;
                unsigned int* payload=new(unsigned int[payLen]);
                unsigned int i=0;
                unsigned int notequals=0;

                //verifies if the data has already been loaded into the S/C to avoid
                //unuseful computations
                if (n==saved_n){
                        for(i=0;i<curveLen && notequals==0;i++){
                                if (A[i]!=saved_A[i]) notequals++;
                                if (B[i]!=saved_B[i]) notequals++;
                        }
                        for(i=0;i<baseLen && notequals==0;i++){
                                if (x[i]!=saved_x[i]) notequals++;
                                if (y[i]!=saved_y[i]) notequals++;
                        }
                }
                else notequals=1;

                if (notequals!=0){
                        //prepares the arrays to store the ECC info for successive comparings
                        delete(saved_A); saved_A=new(unsigned int[curveLen]);
                        delete(saved_B); saved_B=new(unsigned int[curveLen]);
                        delete(saved_x); saved_x=new(unsigned int[baseLen]);
                        delete(saved_y); saved_y=new(unsigned int[baseLen]);
                        delete(saved_key); saved_key=NULL;
                        saved_n=n;

                        //prepares the data to send to the S/C
                        payload[0]=(((unsigned int)SET_ECC_INFO)<<24)|n<<8|payLen;
                        for(i=0;i<curveLen;i++) payload[i+1]=saved_A[i]=A[i];
                        for(i=0;i<curveLen;i++) payload[i+curveLen+1]=saved_B[i]=B[i];
                        for(i=0;i<baseLen;i++) payload[i+curveLen*2+1]=saved_x[i]=x[i];
                        for(i=0;i<baseLen;i++) payload[i+curveLen*2+baseLen+1]=saved_y[i]=y[i];
                        if (freeQue==0) return SC_QUE_FULL; else freeQue--;
                        SC.sc_rcv(payLen, payload);
                        freeQue++;
                }
                decOpened();
                return OK;
        }else return TOO_OPENED;
}



int sC_driver::setECCInfo(unsigned int n, unsigned int* A, unsigned int* B,
                              unsigned int* x, unsigned int* y, unsigned int* key){
                                      //n is the length in BITS
//method for calling the SETECCINFO command passing also the public key of the other peer
//pre: n>0 && A!=NULL && B!=NULL && x !=NULL && y!=NULL && key!=NULL
//post: none

        if(incOpened()==1){
                unsigned int curveLen=(unsigned int)ceil(((float)n)/32);
                unsigned int baseLen=(unsigned int)ceil(((float)n)*2/32);
                unsigned int payLen=curveLen*3+baseLen*2+1;
                unsigned int* payload=new(unsigned int[payLen]);
                unsigned int i=0;
                unsigned int notequals=0;
```

```
               //verifies if the data has already been loaded into the S/C to avoid
               //unuseful computations
               if (n==saved_n && saved_key!=NULL){
                      for(i=0;i<curveLen && notequals==0;i++){
                             if (A[i]!=saved_A[i]) notequals++;
                             if (B[i]!=saved_B[i]) notequals++;
                             if (key[i]!=saved_key[i]) notequals++;
                      }
                      for(i=0;i<baseLen && notequals==0;i++){
                             if (x[i]!=saved_x[i]) notequals++;
                             if (y[i]!=saved_y[i]) notequals++;
                      }
               }
               else notequals=1;

               if (notequals!=0){
                      //prepares the arrays to store the ECC info for successive comparings
                      delete(saved_A); saved_A=new(unsigned int[curveLen]);
                      delete(saved_B); saved_B=new(unsigned int[curveLen]);
                      delete(saved_x); saved_x=new(unsigned int[baseLen]);
                      delete(saved_y); saved_y=new(unsigned int[baseLen]);
                      delete(saved_key); saved_key=new(unsigned int[curveLen]);
                      saved_n=n;

                      //prepares the data to send to the S/C
                      payload[0]=(((unsigned int)SET_ECC_INFO)<<24)|n<<8|payLen;
                      for(i=0;i<curveLen;i++) payload[i+1]=saved_A[i]=A[i];
                      for(i=0;i<curveLen;i++) payload[i+curveLen+1]=saved_B[i]=B[i];
                      for(i=0;i<baseLen;i++) payload[i+curveLen*2+1]=saved_x[i]=x[i];
                      for(i=0;i<baseLen;i++) payload[i+curveLen*2+baseLen+1]=saved_y[i]=y[i];
                      for(i=0;i<curveLen;i++) payload[i+curveLen*2+baseLen*2+1]=saved_key[i]=key[i];
                      if (freeQue==0) return SC_QUE_FULL; else freeQue--;
                      SC.sc_rcv(payLen, payload);
                      freeQue++;
               }
               decOpened();
               return OK;
       }else return TOO_OPENED;
}



int sC_driver::getPublicKey(){
//method for calling the TEST_SC command
//pre: none
//post: none

       if(incOpened()==1){
               unsigned int code=0;

               code=(((unsigned int)GET_PUB_KEY)<<24);
               if (freeQue==0) return SC_QUE_FULL; else freeQue--;
               SC.sc_rcv(1, &code);
               freeQue++;
               decOpened();
               return OK;
       }else return TOO_OPENED;
}



int sC_driver::pubEncrypt(unsigned int length, unsigned int* data){
//method for calling the PUBENCRYPT command
//pre: length>0 && data!=NULL
//post: none

       #ifndef NDEBUG
               assert(length>0);
               assert(data!=NULL);
       #endif;

       if(incOpened()==1){
               unsigned int* payload=new(unsigned int[length+1]);
               unsigned int i=0;

               payload[0]=(((unsigned int)PUBLIC_ENCRYPT)<<24)|length;
               for(i=0;i<length;i++) payload[i+1]=data[i];
```

```
                if (freeQue==0) return SC_QUE_FULL; else freeQue--;
                SC.sc_rcv(length+1, payload);
                freeQue++;
                decOpened();
                return OK;
        }else return TOO_OPENED;
}



int sC_driver::pubDecrypt(unsigned int length, unsigned int* data){
//method for calling the PUBDECRYPT command
//pre: length>0 && data!=NULL
//post: none

        #ifndef NDEBUG
                assert(length>0);
                assert(data!=NULL);
        #endif;

        if(incOpened()==1){
                unsigned int* payload=new(unsigned int[length+1]);
                unsigned int i=0;

                payload[0]=(((unsigned int)PUBLIC_DECRYPT)<<24)|length;
                for(i=0;i<length;i++) payload[i+1]=data[i];

                if (freeQue==0) return SC_QUE_FULL; else freeQue--;
                SC.sc_rcv(length+1, payload);
                freeQue++;
                decOpened();
                return OK;
        }else return TOO_OPENED;
}



int sC_driver::hash(unsigned int length1, unsigned int length2, unsigned int* data1,
                        unsigned int* data2, unsigned int hashfcn){
//method for calling the HASH command
//pre: length1>0 && length2>0 && data1!=NULL && data2!=NULL && (hashfcn==1 || hashfcn==2)
//post: none

        #ifndef NDEBUG
                assert(length1>0);
                assert(length2>0);
                assert(data1!=NULL);
                assert(data2!=NULL);
                assert(hashfcn==1 || hashfcn==2);
        #endif;

        if(incOpened()==1){
                unsigned int* payload=new(unsigned int[length1+length2+1]);
                unsigned int i=0;

                payload[0]=(((unsigned int)HASH)<<24)|length1<<16|length2<<8|hashfcn;
                for(i=0;i<length1;i++) payload[i+1]=data1[i];
                for(i=0;i<length2;i++) payload[i+length1+1]=data2[i];

                if (freeQue==0) return SC_QUE_FULL; else freeQue--;
                SC.sc_rcv(length1+length2+1, payload);
                freeQue++;
                decOpened();
                return OK;
        }else return TOO_OPENED;
}



int sC_driver::genSymmSign(unsigned int sa, unsigned int length, unsigned int* data1,
                                unsigned int* data2, unsigned int hashfcn){
//method for calling the GETSYMMSIGN command
//pre: sa!=0 && length >0 & data1!=NULL && data2!=NULL && (hashfcn==1 || hashfcn==2)
//post: none

        #ifndef NDEBUG
                assert(sa!=0);
                assert(length>0);
```

```
		assert(data1!=NULL);
		assert(data2!=NULL);
		assert(hashfcn==1 || hashfcn==2);
	#endif;

	if(incOpened()==1){
		unsigned int* payload=new(unsigned int[length*2+1]);
		unsigned int i=0;

		payload[0]=(((unsigned int)GEN_SYMM_SIGN)<<24)|length<<16|hashfcn<<8|sa;
		for(i=0;i<length;i++) payload[i+1]=data1[i];
		for(i=0;i<length;i++) payload[i+length+1]=data2[i];

		if (freeQue==0) return SC_QUE_FULL; else freeQue--;
		SC.sc_rcv(length*2+1, payload);
		freeQue++;
		decOpened();
		return OK;
	}else return TOO_OPENED;
}



int sC_driver::genECDSASign(unsigned int length, unsigned int* data){
//method for calling the GENECDSASIGN command
//pre: length>0 && data!=NULL
//post: none

	#ifndef NDEBUG
		assert(length>0);
		assert(data!=NULL);
	#endif;

	if(incOpened()==1){
		unsigned int* payload=new(unsigned int[length+1]);
		unsigned int i=0;

		payload[0]=(((unsigned int)GEN_ECDSA_SIGNATURE)<<24)|length;
		for(i=0;i<length;i++) payload[i+1]=data[i];

		if (freeQue==0) return SC_QUE_FULL; else freeQue--;
		SC.sc_rcv(length+1, payload);
		freeQue++;
		decOpened();
		return OK;
	}else return TOO_OPENED;
}


int sC_driver::verifyECDSASig(unsigned int length, unsigned int* data){
//method for calling the VERIFYECDSASIGN command
//pre: length>0 && data!=NULL
//post: none

	#ifndef NDEBUG
		assert(length>0);
		assert(data!=NULL);
	#endif;

	if(incOpened()==1){
		unsigned int* payload=new(unsigned int[length+1]);
		unsigned int i=0;

		payload[0]=(((unsigned int)VERIFY_ECDSA_SIGNATURE)<<24)|length;
		for(i=0;i<length;i++) payload[i+1]=data[i];

		if (freeQue==0) return SC_QUE_FULL; else freeQue--;
		SC.sc_rcv(length+1, payload);
		freeQue++;
		decOpened();
		return OK;
	}else return TOO_OPENED;
}
```

```
//******************************private methods************************************



int sC_driver::firstFreeSlot(){
//computes the first free mem slot into the S/C, if any
//pre: sc!=NULL
//post: slot<sc+MAX_SC_SLOTS || slot==NULL

        #ifndef NDEBUG
                assert(sc!=NULL);
        #endif;

        scSlot* slot=sc;
        int pos=0;

        while(pos<MAX_SC_SLOTS && slot->posArray!=0){
                slot++;
                pos++;
        }


        if (slot=sc+MAX_SC_SLOTS) pos=-1;

        #ifndef NDEBUG
                assert(pos<MAX_SC_SLOTS || pos==-1);
        #endif;

        return pos;
}


saInfo* sC_driver::firstFreeSa(){
//computes the first free empty saArray position, if any
//pre: saArray!=NULL
//post: num<sc+MAX_SA || num==NULL

        #ifndef NDEBUG
                assert(saArray!=NULL);
        #endif;

        saInfo* num=saArray;

        while(num<(saArray+MAX_SA) && num->sa!=0) num++;


        if (num=saArray+MAX_SA) num=NULL;

        #ifndef NDEBUG
                assert(num<saArray+MAX_SA|| num==NULL);
        #endif;

        return num;
}


void sC_driver::updateUsage(){
//update the usage info for all the sa contained in S/C mem slots
//pre: none
//post: none

        scSlot* slot;

        for(slot=sc; slot<sc+MAX_SC_SLOTS;slot++) slot->usage--;
}


unsigned int sC_driver::findLeastUsed(){
//return the index of the least used sa
//pre: none
//post: pos<MAX_SC_SLOTS

        scSlot* slot=sc;
        unsigned int    i=0,
                        pos=0,
                        u=UINT_MAX;
```

```
	while(i<MAX_SC_SLOTS){
		if(slot->usage<u){
			pos=i;
			u=slot->usage;
		}
		if (u==0) break;
		i++; slot++;
	}

	#ifndef NDEBUG
		assert(pos<MAX_SC_SLOTS);
	#endif;

	return pos;
}


unsigned int sC_driver::swap(){
//perform the swap of an sa and return the index of the freed slot
//pre: none
//post: pos<MAX_SC_SLOTS

	unsigned int pos=findLeastUsed();
	unsigned int code=(((unsigned int)GET_SA_PARAMETERS)<<24)|pos;
	SC.sc_rcv(1, &code);//the room in the command queue should be assured by previous controls
	(sc[pos].posArray)->slot=MAX_SC_SLOTS;
		//the S/C will call the ip_rcv method giving the SA parameters to be saved

	#ifndef NDEBUG
		assert(pos<MAX_SC_SLOTS);
	#endif;

	return pos;
}


saInfo* sC_driver::findSa(unsigned int sa){
//return the address of the searched sa
//pre: sa>0
//post: pos==NULL || pos>=sc && pos<sc+MAX_SA

	#ifndef NDEBUG
		assert(sa>0);
	#endif;

	//first we search into the sc array... maybe we're lucky!:-))
	scSlot* slot=sc;

	while (slot!=NULL && slot<sc+MAX_SC_SLOTS && (slot->posArray)->sa!=sa) slot++;
	if (slot!=NULL && slot<sc+MAX_SC_SLOTS){

		#ifndef NDEBUG
			assert(slot->posArray==NULL||slot->posArray>=saArray &&
				slot->posArray<saArray+MAX_SA);
		#endif;

		return slot->posArray;
	}

	//if not found into the sc array we search into the saArray
	saInfo* pos=saArray;
	while (pos!=NULL && pos<saArray+MAX_SA && pos->sa!=sa) pos++;

	if  (pos!=NULL || pos>=saArray+MAX_SA) pos=NULL;

	#ifndef NDEBUG
		assert(pos==NULL||pos>=saArray && pos<saArray+MAX_SA);
	#endif;

	return pos;
}

int sC_driver::incOpened(){
//increments the number of opened devices checking that is never higher than one
//pre: none
//post: none
```

```
        if(opened_dev==0){
                opened_dev++;
                return 1;
        }
        else return 0;

}


void sC_driver::decOpened(){
//decrements the number of opened devices
//pre: none
//post: none
        opened_dev--;
}
```

Appendix B. Cache behavior simulation without delays

```
//
//file: simul-no-delay.c
//
//simulation of a crypto-processor SA cache using a completely
//associative cache with the LRU policy. Cache misses study only.
//
//written by Alberto Ferrante, April 2002



#include <stdio.h>
#include <string.h>

//program compilation option defines
//#define USE_FIN_PACKETS
//#define CLOSE_UNUSED
//#define PRINT_INSTANT_STATISTICS
//#define PRINT_CACHE_DISTRIB
//#define PRINT_SA_DISTRIB
//#define PRINT_CACHE_REUSE
//#define PRINT_SA_REUSE
//#define QUICK_MODE


//dimension of the data structures
#define CACHE_SIZE 256
#define MAX_SA 4000
#define ROW_LENGTH 70
#define MAX_DEL 1000

//scale factor for the timestamps; 0.00161 for reaching 2Mbit/s
#define SCALE_TIME 1

//timeout for an unused connection
#define UNUSED_TIMEOUT 1800
//checking interval for unused connections
#define CHECK_TIME 60

struct dataT{  //data taken from file
	double time;
	int sourceIP;
	int destIP;
	int sourceTCP;
	int destTCP;
	int bytes;
};

struct SADel{ //element of the SAD
	int sourceIP;
	int destIP;
	int cached;
	unsigned counter;
	double time;
};

struct SACel{  //element of the SAC
	int sourceIP;
	int destIP;
	double time;
	long countUsed;
};


struct synFin{
	int sourceIP;
	int destIP;
	double time;
	int used;
};
```

```
char row[ROW_LENGTH];  //row of the data file
char rowFin[ROW_LENGTH];  //row of the syn/fin data file
FILE *infile; //input file
FILE *infileFin; //input file for TCP Syn/Fin
struct dataT datagram; //data related to each IP datagram
struct SADel SAD[MAX_SA]; //Security Association DB
struct SACel SAC[CACHE_SIZE]; //Cached Security Association DB

struct SADel deletedSA[MAX_DEL]; //for taking track of the discarded SAs
struct SADel *last;  //last element of the deleted SA list

struct synFin fin;

long	discarded,  //counts the discared SAs
	discardedOnce,  //counts the number of SAs discarded only once
	countUnused, //counts the number of SA are discarded because unused
	howMany, //counts the number of opened SA
	prevMissPrint, //number of misses previously printed
	prevCompulsoryPrint,  //number of comp. missess prev. printed
	cacheReuse, //sum of the reuse data for each cache entry
	saReuse, //sum of the reuse of each SA
	saNum, //number of SAs
	cacheMisses,  //total number ofa cache misses
	compulsoryMisses, //total number of comp. cache misses
	totalDatagrams,  //total number of datagrams processed
	sa,
	saDistr,  //used for printing the SA distrib. over time
	oldSa, //used for printing the stastistics
	oldMiss, //used for printing the stastistics
	oldComp, //used for printing the stastistics
	closed;

double	saTime; //used for printing the SA cache distrib.

long bytes;  //total bytes processed

double	discardTime, //for computing statistics on discarded SAs
	shortestDiscardTime,
	longestDiscardTime,
	unusedTime;


double toDouble(char* base, char* end){  //converts a string to a double
	double tmp=*base-48;
	int div=10;

	while(++base<end && *base!='.'){
		tmp=tmp*10+(*base)-48;
	}
	while(++base<end){
		tmp=tmp+((float)(*base-48))/div;
		div*=10;
	}

	return tmp;
}

int toInt(char* base, char* end){  //converts a string to an integer
	int tmp=(*base)-48;

	while(++base<end){
		tmp=tmp*10+(*base)-48;
	}

	return tmp;
}

void fill(char row[ROW_LENGTH]){ //fills the structure datagram with the data taken by file
	char *tmp=row;
	char *base=row;
	char *end;
	int length=strlen(row);

	/*setting datagram.time*/
	if ((end=strchr(row, ' '))<row+length){
		datagram.time=toDouble(base, end)*SCALE_TIME;
		base=end;
```

```
        }

        /*setting datagram.sourceIP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.sourceIP=toInt(base, end);
                base=end;
        }

        /*setting datagram.destIP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.destIP=toInt(base, end);
                base=end;
        }

        /*setting datagram.sourceTCP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.sourceTCP=toInt(base, end);
                base=end;
        }

        /*setting datagram.destTCP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.destTCP=toInt(base, end);
                base=end;
        }

        /*setting datagram.bytes*/
        end=row+length-1;
        base++;
        datagram.bytes=34+toInt(base, end);
}

struct SADel* searchSA(struct SADel* DB, struct SADel *end, int source, int dest){
                                             //searches an element in a SAD-like array
        struct SADel* tmp=DB;

        while (tmp<end){
                if(tmp->sourceIP==source && tmp->destIP==dest)
                        break;
                else
                        tmp++;
        }

        if (tmp<end)
                return tmp;
        else
                return NULL;
}


struct SACel* searchCache(int source, int dest){ //searches an element in the SAC DB
        struct SACel* tmp=SAC;

        while (tmp<SAC+CACHE_SIZE){
                if(tmp->sourceIP==source && tmp->destIP==dest)
                        break;
                else
                        tmp++;
        }

        if (tmp<SAC+CACHE_SIZE)
                return tmp;
        else
                return NULL;
}


struct SACel* replace(){ //cache replace with RLU policy
        struct SACel* tmp;
        struct SACel* which=SAC;
        struct SADel* sa;

        for (tmp=SAC+1; tmp<SAC+CACHE_SIZE; tmp++)
                if (tmp->time<which->time)
                        which=tmp;

        cacheReuse+=which->countUsed;
#ifdef PRINT_CACHE_REUSE
```

```
        printf("%d\n", which->countUsed);
#endif

        if((sa=searchSA(SAD, SAD+MAX_SA, which->sourceIP, which->destIP))!=NULL)
                sa->cached=CACHE_SIZE;

        return which;
}


int addToCache(int newSA, int cachePos, int source, int dest){  //add an SA to the SAC
                                        //newSA==1 -> the SA is new, so no cache entry could exist
        struct SACel* where;

        if (newSA||cachePos>=CACHE_SIZE){
                cacheMisses++;
#ifdef PRINT_CACHE_DISTRIB
                if (datagram.time>=saTime+1){  //prints the distribution of the cache
                                                //misses over 1 sec intervals
                        printf("%lf %d %d\n", saTime, cacheMisses-prevMissPrint,
                                        compulsoryMisses-prevCompulsoryPrint);
                        saTime=(long)datagram.time;
                        prevMissPrint=cacheMisses;
                        prevCompulsoryPrint=compulsoryMisses;
                }
#endif //PRINT_CACHE_DISTRIB
                if (newSA)
                        compulsoryMisses++;
                if ((where=searchCache(0, 0))==NULL)
                        where=replace();
                where->sourceIP=source;
                where->destIP=dest;
                where->time=datagram.time;
                where->countUsed=1;
                return ((where-SAC));
        }
        else{
                SAC[cachePos].time=datagram.time;
                SAC[cachePos].countUsed++;
                return cachePos;
        }

}


void removeFromCache(struct SADel *sa){  //remove a SA from the SAC

        if (sa->cached<CACHE_SIZE){
                SAC[sa->cached].sourceIP=0;
                SAC[sa->cached].destIP=0;
                SAC[sa->cached].time=0;
                cacheReuse+=SAC[sa->cached].countUsed;
#ifdef PRINT_CACHE_REUSE
                printf("%d\n", SAC[sa->cached].countUsed);
#endif
                SAC[sa->cached].countUsed=0;
                sa->cached=CACHE_SIZE;
        }
}


struct SADel* unused(){  //find a SA to discard if all the slots in the SAD are full
        struct SADel* tmp=SAD;
        struct SADel* found=SAD;
        struct SADel* which;

        for (tmp=SAD+1; tmp<SAD+MAX_SA; tmp++){
                if ((datagram.time-tmp->time)>(datagram.time-found->time))
                        found=tmp;
        }

        if((which=searchSA(deletedSA, last, found->sourceIP, found->destIP))!=NULL){
                which->counter++;
                discarded++;
                discardTime+=datagram.time-found->time;
                if (datagram.time-found->time<shortestDiscardTime)
                        shortestDiscardTime=datagram.time-found->time;
                if (datagram.time-found->time>longestDiscardTime)
```

```
                        longestDiscardTime=datagram.time-found->time;
                which->time=datagram.time;
        }
        else{
                if (last<deletedSA+MAX_DEL){
                        last->sourceIP=found->sourceIP;
                        last->destIP=found->destIP;
                        last->counter=1;
                        last->time=datagram.time;
                        last++;
                }else printf("no more room for deleted SAs!");
        }
        sa--;

        return found;
}


struct SADel* addSA(int source, int dest){  //add a new SA
        struct SADel* which;

        if ((which=searchSA(SAD, SAD+MAX_SA, source, dest))!=NULL){
                which->counter++;
                which->time=datagram.time;
                which->cached=addToCache(0, which->cached, datagram.sourceIP, datagram.destIP);
        }
        else{
                sa++;
#ifdef QUICK_MODE
                sa++;
#endif
#ifdef PRINT_SA_DISTRIB
                //prints the distribution of the new SA over 1 sec intervals
                if (datagram.time<saTime+1){
                        saDistr++;
#ifdef QUICK_MODE
                        saDistr++;
#endif
                }
                else{
                        printf("%lf %d %d\n", saTime, saDistr, closed);
                        saTime=(long)datagram.time;
                        saDistr=2;
                        closed=0;
                }
#endif //PRINT_SA_DISTRIB

#ifdef QUICK_MODE
                //adds SAs as in IKE phase 2 - quick mode
                if ((which=searchSA(SAD, SAD+MAX_SA, 0, 0))!=NULL){
                        which->sourceIP=datagram.destIP;
                        which->destIP=datagram.sourceIP;
                        which->counter=1;
                        which->time=datagram.time;
                        which->cached=addToCache(1, CACHE_SIZE, datagram.destIP, datagram.sourceIP);
                }
                else{
                        which=unused();
                        removeFromCache(which);
                        countUnused++;
                        unusedTime+=datagram.time-which->time;
                        which->sourceIP=datagram.destIP;
                        which->destIP=datagram.sourceIP;
                        which->counter=1;
                        which->time=datagram.time;
                        which->cached=addToCache(1, CACHE_SIZE, datagram.destIP, datagram.sourceIP);
                }
#endif  //QUICK_MODE


                if ((which=searchSA(SAD, SAD+MAX_SA, 0, 0))!=NULL){
                        which->sourceIP=datagram.sourceIP;
                        which->destIP=datagram.destIP;
                        which->counter=1;
                        which->time=datagram.time;
                        which->cached=addToCache(1, CACHE_SIZE, datagram.sourceIP, datagram.destIP);
                }
                else{
                        which=unused();
                        removeFromCache(which);
                        countUnused++;
                        unusedTime+=datagram.time-which->time;
                        which->sourceIP=datagram.sourceIP;
                        which->destIP=datagram.destIP;
                        which->counter=1;
                        which->time=datagram.time;
                        which->cached=addToCache(1, CACHE_SIZE, datagram.sourceIP, datagram.destIP);
                }
        }

        return which;
}


void close(int source, int dest){  //closes a SA basing on the addresses given
        struct SADel* which;

        if ((which=searchSA(SAD, SAD+MAX_SA, source, dest))!=NULL){
                removeFromCache(which);
                which->sourceIP=0;
                which->destIP=0;

#if defined(USE_FIN_PACKETS) || defined (CLOSE_UNUSED)
                saReuse+=which->counter;
                saNum++;
#ifdef PRINT_SA_REUSE
                printf("%d %d\n", saNum, which->counter);
#endif //PRINT_SA_REUSE
#endif //USE_FIN_PACKETS || CLOSE_UNUSED

                which->counter=0;
                which->time=0;
                sa--;
#ifdef PRINT_SA_DISTRIB
                closed++;
#endif
        }
}


void closeNumber(struct SADel* which){  //close a SA based on the pointer passed

        if (which!=NULL && which>=SAD && which<SAD+MAX_SA){
                removeFromCache(which);
                which->sourceIP=0;
                which->destIP=0;

#if defined(USE_FIN_PACKETS) || defined (CLOSE_UNUSED)
                saReuse+=which->counter;
                saNum++;
#ifdef PRINT_SA_REUSE
                printf("%d %d\n", saNum, which->counter);
#endif //PRINT_SA_REUSE
#endif //USE_FIN_PACKETS || CLOSE_UNUSED

                which->counter=0;
                which->time=0;
                sa--;
#ifdef PRINT_SA_DISTRIB
                closed++;
#endif
        }
}

int fillFin(char row[ROW_LENGTH]){ //fills the structure datagram with the data taken by file
        char *tmp=row;
        char *base=row;
        char *end;
        int length=strlen(row);

        /*setting fin.time*/
        if ((end=strchr(row, ' '))<row+length){
                fin.time=toDouble(base, end)*SCALE_TIME;
                base=end;
        }
        /*setting fin.sourceIP*/
        if ((end=strchr(++base, ' '))<row+length){
                fin.sourceIP=toInt(base, end);
                base=end;
        }

        /*setting fin.destIP*/
        if ((end=strchr(++base, ' '))<row+length){
                fin.destIP=toInt(base, end);
                base=end;
        }

        /*setting fin.sourceTCP*/
        if ((end=strchr(++base, ' '))<row+length){
                base=end;
        }

        /*setting fin.destTCP*/
        if ((end=strchr(++base, ' '))<row+length){
                base=end;
        }

        if ((end=strchr(++base, ' '))<row+length){
        }
        fin.used=0;
        return (*base=='F');
}


void checkFin(){  //checks the SAs to be closed when a FIN packet is received
        char* tmp;
        if (datagram.time>=fin.time){
                if (fin.used==0){
                        close(fin.sourceIP, fin.destIP);
                        fin.used=1;
                }
                while (fgets(rowFin, ROW_LENGTH, infileFin)!=NULL && datagram.time>=fin.time){
                        while (!fillFin(rowFin)){
                                if((tmp=fgets(rowFin, ROW_LENGTH, infileFin))==NULL)
                                        break;
                        }
                        if (tmp!=NULL && datagram.time>=fin.time){
                                close(fin.sourceIP, fin.destIP);
                                fin.used=1;
                        }
                }
        }
}

int main(void){
        struct SADel* tmp;
        struct SADel* tmpSAD;
        struct SACel* tmp1;
        unsigned i=0;
        long lastCheck;
        long oldestConnTime;

        saReuse=0;
        saNum=0;
        lastCheck=0;
        oldestConnTime=0;
        cacheReuse=0;
        closed=0;
        bytes=0;
        sa=0;
        saDistr=0;
        saTime=0;
        shortestDiscardTime=1e6;
        longestDiscardTime=0;
        discardTime=0;
        discarded=0;
        last=deletedSA;
        howMany=0;
        unusedTime=0;
        countUnused=0;
        cacheMisses=0;
        compulsoryMisses=0;
        prevMissPrint=0;
        prevCompulsoryPrint=0;
        totalDatagrams=0;
        fin.used=1;
        for (tmp=SAD; tmp<SAD+MAX_SA; tmp++){
                tmp->sourceIP=0;
                tmp->destIP=0;
                tmp->time=0;
                tmp->counter=0;
                tmp->cached=CACHE_SIZE;
        }
        for (tmp=deletedSA; tmp<deletedSA+MAX_DEL; tmp++){
                tmp->sourceIP=0;
                tmp->destIP=0;
                tmp->time=0;
                tmp->counter=0;
        }

        for(tmp1=SAC; tmp1<SAC+CACHE_SIZE; tmp1++){
                tmp1->sourceIP=0;
                tmp1->destIP=0;
                tmp1->time=0;
                tmp1->countUsed=0;
        }


        infile=fopen("/home/alberto/tesi/simulations/lbl-tcp-3/lbl-tcp-3.tcp", "r");
#ifdef USE_FIN_PACKETS
        infileFin=fopen("/home/alberto/tesi/simulations/lbl-tcp-3/lbl-tcp-3.sf", "r");
#endif
        /*infile=fopen("/home/alberto/tesi/simulations/lbl-pkt-4/lbl-pkt-4.tcp", "r");
#ifdef USE_FIN_PACKETS
                infileFin=fopen("/home/alberto/tesi/simulations/lbl-pkt-4/lbl-pkt-4.sf", "r");
#endif*/

        while (fgets(row, ROW_LENGTH,infile)!=NULL){
                fill(row);
                totalDatagrams++;
                bytes+=datagram.bytes;
                tmp=addSA(datagram.sourceIP, datagram.destIP);
                if (tmp!=NULL && tmp->counter==0){
                        printf("SA closed: max SA usage reached");
                        closeNumber(tmp);
                }
#ifdef USE_FIN_PACKETS
                checkFin();
#endif

#ifdef CLOSE_UNUSED
                if (datagram.time>=lastCheck+CHECK_TIME){//each CHEC_TIME sec, checks and closes
                                                        //all the SA not used for more than
                                                        //UNUSED_TIMEOUT secs
                        if (datagram.time-oldestConnTime>=UNUSED_TIMEOUT){
                                oldestConnTime=datagram.time;
                                for(tmpSAD=SAD; tmpSAD<SAD+MAX_SA; tmpSAD++){
                                        if(tmpSAD->sourceIP!=0 && tmpSAD->destIP!=0){
                                                if (datagram.time-tmpSAD->time>=UNUSED_TIMEOUT){
                                                        /*printf("%d %d %lf %lf\n",
                                                                tmpSAD->sourceIP,
                                                                tmpSAD->destIP,
                                                                datagram.time, tmpSAD->time);*/
                                                        closeNumber(tmpSAD);
                                                }
                                                else if (tmpSAD->time<oldestConnTime)
                                                        oldestConnTime=tmpSAD->time;
                                        }
                                }
                        }
                        lastCheck=(long)datagram.time;
                }
#endif  //CLOSE_UNUSED

#ifdef PRINT_INSTANT_STATISTICS
                //prints the info about opened SA and cache misses for each datagram
                if(oldSa!=sa || oldMiss!=cacheMisses || oldComp!=compulsoryMisses){
                        oldSa=sa;
                        oldMiss=cacheMisses;
                        oldComp=compulsoryMisses;
                        printf("%lf %d %d %d\n", datagram.time, sa, cacheMisses, compulsoryMisses);
                }
#endif //PRINT_INSTANT_STATISTICS

                /*printf("%lf %d %d %d %d *** %s\n ",     datagram.time, datagram.sourceIP,
                                                        datagram.destIP, datagram.sourceTCP,
                                                        datagram.destTCP, datagram.bytes, row);*/
        }

        fclose(infile);
#ifdef USE_FIN_PACKETS
        fclose(infileFin);
#endif

        /*howMany=0;
        for (i=0; i<MAX_DEL && !(deletedSA[i].sourceIP==0 && deletedSA[i].destIP==0); i++){
                howMany+=deletedSA[i].counter;
                if(deletedSA[i].counter==1)
                        discardedOnce++;
        }*/

#if !defined(USE_FIN_PACKETS) && !defined (CLOSE_UNUSED)
        saReuse=0;
        for (i=0; i<MAX_SA; i++){
                if (SAD[i].sourceIP!=0 && SAD[i].destIP!=0){
                        saReuse+=SAD[i].counter;
#ifdef PRINT_SA_REUSE
                        printf("%d %d\n", i, SAD[i].counter);
#endif //PRINT_SA_REUSE
                }
        }
#endif //USE_FIN_PACKETS || CLOSE_UNUSED

#if defined(USE_FIN_PACKETS) || defined (CLOSE_UNUSED)
        for (i=0; i<MAX_SA; i++){
                if (SAD[i].sourceIP!=0 && SAD[i].destIP!=0){
                        saReuse+=SAD[i].counter;
#ifdef PRINT_SA_REUSE
                        printf("%d %d\n", i, SAD[i].counter);
#endif //PRINT_SA_REUSE
                }
        }
        saNum+=sa;
#endif //USE_FIN_PACKETS || CLOSE_UNUSED

        for(tmp1=SAC; tmp1<SAC+CACHE_SIZE; tmp1++){
                if (tmp1->sourceIP!=0 && tmp1->destIP!=0){
                        cacheReuse+=tmp1->countUsed;
#ifdef PRINT_CACHE_REUSE
                        printf("%d\n", tmp1->countUsed);
#endif
                }
        }

        fprintf(stderr, "\n***************************************************************\n");
        fprintf(stderr, "***************************************************************\n");
        fprintf(stderr, "max number of SA: %d; cache size %d\n", MAX_SA, CACHE_SIZE);
#ifdef USE_FIN_PACKETS
        fprintf(stderr, "Using FIN TCP packets for SA closing\n");
#endif

#ifdef CLOSE_UNUSED
        fprintf(stderr, "Closing connections after an unused timeout of %ds (checking each %ds)\n",
                                                                UNUSED_TIMEOUT, CHECK_TIME);
#endif
        fprintf(stderr, "***************************************************************\n\n");

        /*
        fprintf(stderr, "SAs discarded more than one time: %d\n", discarded);
        fprintf(stderr, "SAs discarded once: %d\n", discardedOnce);
        */

        fprintf(stderr, "Total number of datagrams analyzed %d\n", totalDatagrams);
        fprintf(stderr, "Average dimension of datagrams (34bytes of header): %.2fbytes\n",
                                        (float)bytes/(float)totalDatagrams);
        fprintf(stderr, "Average data rate: %.3fkbit/s\n", 8*(float)(bytes)/datagram.time/1024);
        fprintf(stderr, "Average connections managed per second: %.2f\n",
                (float)(totalDatagrams)/datagram.time);
```

```
#if !defined(USE_FIN_PACKETS) && !defined (CLOSE_UNUSED)
        fprintf(stderr, "Average reuse of each SA: %.2f\n", (float)saReuse/(float)sa);
#endif //!USE_FIN_PACKETS || !CLOSE_UNUSED

#if defined(USE_FIN_PACKETS) || defined (CLOSE_UNUSED)
        fprintf(stderr, "Average reuse of each SA (before closing): %.2f\n",
                                                     (float)saReuse/(float)saNum);
#endif //USE_FIN_PACKETS || CLOSE_UNUSED

        /*
        fprintf(stderr, "average reuse of discarded SAs: %.2f\n", (float)howMany/(float)i);
        fprintf(stderr, "shortest time between two discards of \"the same\" SA: %.2lfs\n",
                                                                    shortestDiscardTime);
        fprintf(stderr, "longest time between two discards of \"the same\" SA: %.2lfs\n",
                                                                    longestDiscardTime);
        fprintf(stderr, "average time between two discards of \"the same\" SA: %.2lfs\n",
                                                     discardTime/(float)discarded);
        fprintf(stderr, "average unused time between replacement: %.2lfs\n",
                                              unusedTime/(float)countUnused);
        */

        fprintf(stderr, "\nTotal cache misses: %d (%.2f\%)\n",      cacheMisses,
                                      100*(float)cacheMisses/(float)totalDatagrams);
        fprintf(stderr, "Compulsory misses: %d\n", compulsoryMisses);
        fprintf(stderr, "Avoidable cache misses: %d (%.2f\%)\n", cacheMisses-compulsoryMisses,
                                      100*(float)(cacheMisses-compulsoryMisses)/(float)totalDatagrams);

        fprintf(stderr, "Average reuse of each cache position before replacing %.2f\n",
                               (float)cacheReuse/(float)cacheMisses);

        return 0;
}
```

Appendix C. Cache behavior simulation considering the crypto processor delays

```
//
//file: simul-delay.c
//
//simulation of a crypto-processor SA cache using a completely
//associative cache with the LRU policy. Study of the
//delay introduced by the crypto processor, regardless of the
//SA creation phase
//
//written by Alberto Ferrante, April 2002


#include <stdio.h>
#include <string.h>
#include <math.h>

//program compilation option defines
//#define USE_FIN_PACKETS
//#define CLOSE_UNUSED
//the value defined for PRINT_THROUGHPUT_DISTRIB is also used for timing the printing
#define PRINT_THROUGHPUT_DISTRIB 0.25
#define PRINT_CACHE_DISTRIB

//#define QUICK_MODE

//dimension of the data structures
#define CACHE_SIZE 128
#define MAX_SA 4000
#define ROW_LENGTH 70
#define MAX_DEL 1000

//scale factor for the timestamps; 0.00161 for reaching 200Mbit/s of throughput
#define SCALE_TIME 0.00161

//minimum distance between two datagram
#define MIN_DISTANCE 1e-12

//timeout for an unused connection
#define UNUSED_TIMEOUT 1800
//checking interval for unused connections
#define CHECK_TIME 60

//delay parameters
#define CHANNELL 1.515e-8
#define CHANNELL_INITIAL 1.515e-8
#define CACHED_DELAY 5e-9
//100MHz AES hardware
//#define ENC_TIME 2.2e-7
//#define ENC_SETUP 1.7e-7
//80MHz AES hardware
//#define ENC_TIME 2.75e-7
//#define ENC_SETUP 2.125e-7
//70MHz AES hardware
//#define ENC_TIME 3.14e-7
//#define ENC_SETUP 2.42e-7
//60MHz AES hardware
#define ENC_TIME 3.67e-7
#define ENC_SETUP 2.83e-7
//55MHz AES hardware
//#define ENC_TIME 4e-7
//#define ENC_SETUP 3.09e-7
//50MHz AES hardware
//#define ENC_TIME 4.4e-7
//#define ENC_SETUP 3.4e-7
#define SAINFO_LEN 66
#define KEY 3
#define CRC_TIME 4e-8

struct dataT{  //data taken from file
```

```
        double time;
        double theorTime;
        int sourceIP;
        int destIP;
        int sourceTCP;
        int destTCP;
        int bytes;
};

struct SADel{ //element of the SAD
        int sourceIP;
        int destIP;
        int cached;
        unsigned counter;
        double time;
};

struct SACel{  //element of the SAC
        int sourceIP;
        int destIP;
        double time;
};


struct synFin{  //Syn Fin packet data
        int sourceIP;
        int destIP;
        double time;
        int used;
};



char row[ROW_LENGTH];  //row of the data file
char rowFin[ROW_LENGTH];  //row of the syn/fin data file
FILE *infile; //input file
FILE *infileFin; //input file for TCP Syn/Fin
#ifdef PRINT_THROUGHPUT_DISTRIB
FILE *outTdistrib;
#endif
#ifdef PRINT_CACHE_DISTRIB
FILE *outcachedistrib;
#endif
struct dataT datagram; //data related to each IP datagram
struct SADel SAD[MAX_SA]; //Security Association DB
struct SACel SAC[CACHE_SIZE]; //Cached Security Association DB

struct SADel deletedSA[MAX_DEL]; //for taking track of the discarded SAs
struct SADel *last;  //last element of the deleted SA list

struct synFin fin;

long    discarded,  //counts the discared SAs
        discardedOnce,  //counts the number of SAs discarded only once
        countUnused, //counts the number of SA are discarded because unused
        howMany, //counts the number of opened SA
        delayed,  //counts the number of SAs that are delayed due to processing time
        prevMissPrint, //number of misses previously printed
        prevCompulsoryPrint,  //number of comp. missess prev. printed
        cacheMisses,  //total number ofa cache misses
        compulsoryMisses, //total number of comp. cache misses
        totalDatagrams,  //total number of datagrams processed
        sa,
        saDistr,  //used for printing the SA distrib. over time
        oldSa, //used for printing the stastistics
        oldMiss, //used for printing the stastistics
        oldComp; //used for printing the stastistics

double  saTime, //used for printing the SA cache distrib.
        delay,  //keeps track of the delay introduced by the datagram processing
        lastTimestamp,  //timestamp of the last datagram analizied
        procTime; //total time used for processing

long bytes;  //total bytes processed

double  discardTime, //for computing statistics on discarded SAs
        shortestDiscardTime,
        longestDiscardTime,
        unusedTime;


double toDouble(char* base, char* end){  //converts a string to a double
        double tmp=*base-48;
        int div=10;

        while(++base<end && *base!='.'){
                tmp=tmp*10+(*base)-48;
        }
        while(++base<end){
                tmp=tmp+((float)(*base-48))/div;
                div*=10;
        }

        return tmp;
}

int toInt(char* base, char* end){  //converts a string to an int
        int tmp=(*base)-48;

        while(++base<end){
                tmp=tmp*10+(*base)-48;
        }

        return tmp;
}

void fill(char row[ROW_LENGTH]){ //fills the structure datagram with the data taken by file
        char *tmp=row;
        char *base=row;
        char *end;
        int length=strlen(row);

        /*setting datagram.time*/
        if ((end=strchr(row, ' '))<row+length){
                datagram.time=toDouble(base, end)*SCALE_TIME;
                datagram.theorTime=datagram.time;
                base=end;
        }

        /*setting datagram.sourceIP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.sourceIP=toInt(base, end);
                base=end;
        }

        /*setting datagram.destIP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.destIP=toInt(base, end);
                base=end;
        }

        /*setting datagram.sourceTCP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.sourceTCP=toInt(base, end);
                base=end;
        }

        /*setting datagram.destTCP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.destTCP=toInt(base, end);
                base=end;
        }

        /*setting datagram.bytes*/
        end=row+length-1;
        base++;
        datagram.bytes=34+toInt(base, end);
}

struct SADel* searchSA(struct SADel* DB, struct SADel *end, int source, int dest){
                                             //searches an element in a SAD-like array
        struct SADel* tmp=DB;

        while (tmp<end){
                if(tmp->sourceIP==source && tmp->destIP==dest)
                        break;
                else
                        tmp++;
        }

        if (tmp<end)
                return tmp;
        else
                return NULL;
}


struct SACel* searchCache(int source, int dest){ //searches an element in the SAC DB
        struct SACel* tmp=SAC;

        while (tmp<SAC+CACHE_SIZE){
                if(tmp->sourceIP==source && tmp->destIP==dest)
                        break;
                else
                        tmp++;
        }

        if (tmp<SAC+CACHE_SIZE)
                return tmp;
        else
                return NULL;
}


struct SACel* replace(){ //cache replace with LRU policy
        struct SACel* tmp;
        struct SACel* which=SAC;
        struct SADel* sa;

        for (tmp=SAC+1; tmp<SAC+CACHE_SIZE; tmp++)
                if (tmp->time<which->time)
                        which=tmp;
        if((sa=searchSA(SAD, SAD+MAX_SA, which->sourceIP, which->destIP))!=NULL)
                sa->cached=CACHE_SIZE;

        //store delay
        delay+=(double)ENC_SETUP+(double)KEY*(double)ENC_TIME+(double)(SAINFO_LEN-2)*CRC_TIME+
                (ceil((double)SAINFO_LEN/4)+1)*(double)CHANNELL+(double)CHANNELL_INITIAL*2+
                (double)CHANNELL;

        return which;
}


int addToCache(int newSA, int cachePos, int source, int dest){  //add an SA to the SAC
                                          //newSA==1 -> the SA is new, so no cache entry could exist
        struct SACel* where;

        delay+=CACHED_DELAY;
        if (newSA||cachePos>=CACHE_SIZE){
                //miss delay
                delay+=(double)ENC_SETUP+(double)KEY*(double)ENC_TIME+(double)(SAINFO_LEN-2)*CRC_TIME+
                        (ceil((double)SAINFO_LEN/4)+1)*(double)CHANNELL+(double)CHANNELL_INITIAL;
                cacheMisses++;
#ifdef PRINT_CACHE_DISTRIB
                if (datagram.time>=saTime+0.10){  //prints the distribution of the cache
                                                //misses over 1 sec intervals
                        fprintf(outcachedistrib, "%lf %d %d\n", saTime, cacheMisses-prevMissPrint,
                                        compulsoryMisses-prevCompulsoryPrint);
                        saTime=datagram.time;
                        prevMissPrint=cacheMisses;
                        prevCompulsoryPrint=compulsoryMisses;
                }
#endif //PRINT_CACHE_DISTRIB

                if (newSA)
                        compulsoryMisses++;
                if ((where=searchCache(0, 0))==NULL)
                        where=replace();

                /*printf("pos %d: in %d %d %lf, out %d %d %lf\n", where-SAC, datagram.sourceIP,
                                datagram.destIP, datagram.time, where->sourceIP,
                                where->destIP, where->time);*/
                where->sourceIP=source;
                where->destIP=dest;
                where->time=datagram.time;
                return ((where-SAC));
        }
        else{
                SAC[cachePos].time=datagram.time;
                /*printf("Update pos %d: in %d %d %lf, out %d %d %lf\n", cachePos, datagram.sourceIP,
                                datagram.destIP, datagram.time, SAC[cachePos].sourceIP,
                                SAC[cachePos].destIP, SAC[cachePos].time);*/
                return cachePos;
        }

}


void removeFromCache(struct SADel *sa){  //remove a SA from the SAC

        if (sa->cached<CACHE_SIZE){
                SAC[sa->cached].sourceIP=0;
                SAC[sa->cached].destIP=0;
                SAC[sa->cached].time=0;
                sa->cached=CACHE_SIZE;
        }
}


struct SADel* unused(){  //find a SA to discard if all the slots in the SAD are full
        struct SADel* tmp=SAD;
        struct SADel* found=SAD;
        struct SADel* which;

        for (tmp=SAD+1; tmp<SAD+MAX_SA; tmp++){
                if ((datagram.time-tmp->time)>(datagram.time-found->time))
                        found=tmp;
        }

        if((which=searchSA(deletedSA, last, found->sourceIP, found->destIP))!=NULL){
                which->counter++;
                discarded++;
                discardTime+=datagram.time-found->time;
                if (datagram.time-found->time<shortestDiscardTime)
                        shortestDiscardTime=datagram.time-found->time;
                if (datagram.time-found->time>longestDiscardTime)
                        longestDiscardTime=datagram.time-found->time;
                which->time=datagram.time;
        }
        else{
                if (last<deletedSA+MAX_DEL){
                        last->sourceIP=found->sourceIP;
                        last->destIP=found->destIP;
                        last->counter=1;
                        last->time=datagram.time;
                        last++;
                }else printf("no more room for deleted SAs!");
        }
        sa--;

        return found;
}


double process(int first){  //stub for simulating the packet processing

        return  (((double)CHANNELL_INITIAL+
                (double)(ceil((float)(datagram.bytes+16*first)/4)+2)*(double)CHANNELL)+
                ((double)CHANNELL_INITIAL+
                (double)(ceil((float)(datagram.bytes)/4)+1)*(double)CHANNELL)
                +(double)ENC_SETUP+(double)ENC_TIME*ceil((double)datagram.bytes/16));
}


struct SADel* addSA(int source, int dest){  //adds a new SA
        struct SADel* which;

        delay=0;
        if ((which=searchSA(SAD, SAD+MAX_SA, source, dest))!=NULL){
                which->counter++;
                which->cached=addToCache(0, which->cached, datagram.sourceIP, datagram.destIP);
                delay+=process(0);
        }
        else{
                //SA creation as in IKE Phase 2 quick mode
#ifdef QUICK_MODE
                sa++;
                if ((which=searchSA(SAD, SAD+MAX_SA, 0, 0))==NULL){
                        which=unused();
                        removeFromCache(which);
                        countUnused++;
                        unusedTime+=datagram.time-which->time;
                }
                which->sourceIP=datagram.destIP;
                which->destIP=datagram.sourceIP;
                which->counter=1;
                which->cached=addToCache(1, CACHE_SIZE, datagram.destIP, datagram.sourceIP);
#endif //QUICK_MODE

                sa++;
                if ((which=searchSA(SAD, SAD+MAX_SA, 0, 0))==NULL){
                        which=unused();
                        removeFromCache(which);
                        countUnused++;
                        unusedTime+=datagram.time-which->time;
                }
                which->sourceIP=datagram.sourceIP;
                which->destIP=datagram.destIP;
                which->counter=1;
                which->cached=addToCache(1, CACHE_SIZE, datagram.sourceIP, datagram.destIP);
                delay+=process(1);
        }
        which->time=datagram.time+delay;
        lastTimestamp=which->time;
        procTime+=delay;
        SAC[which->cached].time=which->time;

        return which;
}


void close(int source, int dest){  //closes an SA
        struct SADel* which;

        if ((which=searchSA(SAD, SAD+MAX_SA, source, dest))!=NULL){
                removeFromCache(which);
                which->sourceIP=0;
                which->destIP=0;
                which->counter=0;
                which->time=0;
                sa--;
        }
}


void closeNumber(struct SADel* which){  //closes an SA already ahving its pointer

        if (which!=NULL && which>=SAD && which<SAD+MAX_SA){
                removeFromCache(which);
                which->sourceIP=0;
                which->destIP=0;
                which->counter=0;
                which->time=0;
                sa--;
        }
}


int fillFin(char row[ROW_LENGTH]){ //fills the structure fin with the data taken by the sf file
        char *tmp=row;
        char *base=row;
        char *end;
        int length=strlen(row);

        /*setting fin.time*/
        if ((end=strchr(row, ' '))<row+length){
                fin.time=toDouble(base, end)*SCALE_TIME;
                base=end;
        }

        /*setting fin.sourceIP*/
        if ((end=strchr(++base, ' '))<row+length){
                fin.sourceIP=toInt(base, end);
                base=end;
        }

        /*setting fin.destIP*/
        if ((end=strchr(++base, ' '))<row+length){
                fin.destIP=toInt(base, end);
                base=end;
        }

        /*setting fin.sourceTCP*/
        if ((end=strchr(++base, ' '))<row+length){
                base=end;
        }

        /*setting fin.destTCP*/
        if ((end=strchr(++base, ' '))<row+length){
                base=end;
        }

        if ((end=strchr(++base, ' '))<row+length){
        }
        //if(*base=='F')printf("%lf %d %d %c\n", fin.time, fin.sourceIP, fin.destIP, *base);
        fin.used=0;
        return (*base=='F');
}


void checkFin(){  //checks the connections to be closed
        char* tmp;

        if (datagram.theorTime>=fin.time){
                if (fin.used==0){
                        close(fin.sourceIP, fin.destIP);
                        fin.used=1;
                        /*printf("%lf %lf %d %d\n", fin.time, datagram.theorTime,
                                        fin.sourceIP, fin.destIP);*/
                }
                while (fgets(rowFin, ROW_LENGTH, infileFin)!=NULL && datagram.theorTime>=fin.time){
                        while (!fillFin(rowFin)){
                                if((tmp=fgets(rowFin, ROW_LENGTH, infileFin))==NULL)
                                        break;
                        }
                        if (tmp!=NULL && datagram.theorTime>=fin.time){
                                close(fin.sourceIP, fin.destIP);
                                /*printf("%lf %lf %d %d\n", fin.time, datagram.theorTime,
                                                fin.sourceIP, fin.destIP);*/
                                fin.used=1;
                        }
                }
        }
}

int main(void){
        struct SADel* tmp;
        struct SADel* tmpSAD;
        struct SACel* tmp1;
        unsigned i=0;
        long lastCheck;
        long oldestConnTime;
        double  throughTime,
                throughTheorTime;
        long throughBytes;

//initialization
        throughBytes=0;
        throughTheorTime=0;
        throughTime=0;
        bytes=0;
        delayed=0;
        procTime=0;
        sa=0;
        saDistr=0;
        saTime=0;
        shortestDiscardTime=1e6;
        longestDiscardTime=0;
        discardTime=0;
        discarded=0;
        last=deletedSA;
        howMany=0;
        unusedTime=0;
        countUnused=0;
        cacheMisses=0;
        compulsoryMisses=0;
        totalDatagrams=0;
        fin.used=1;
        for (tmp=SAD; tmp<SAD+MAX_SA; tmp++){
                tmp->sourceIP=0;
                tmp->destIP=0;
                tmp->time=0;
                tmp->counter=0;
                tmp->cached=CACHE_SIZE;
        }
        for (tmp=deletedSA; tmp<deletedSA+MAX_DEL; tmp++){
                tmp->sourceIP=0;
                tmp->destIP=0;
                tmp->time=0;
                tmp->counter=0;
        }

        for(tmp1=SAC; tmp1<SAC+CACHE_SIZE; tmp1++){
                tmp1->sourceIP=0;
                tmp1->destIP=0;
                tmp1->time=0;
        }

//opening the files
        infile=fopen("/home/alberto/tesi/simulations/lbl-tcp-3/lbl-tcp-3.tcp", "r");
#ifdef USE_FIN_PACKETS
        infileFin=fopen("/home/alberto/tesi/simulations/lbl-tcp-3/lbl-tcp-3.sf", "r");
#endif
/*      infile=fopen("/home/alberto/tesi/simulations/lbl-pkt-4/lbl-pkt-4.tcp", "r");
#ifdef USE_FIN_PACKETS
                infileFin=fopen("/home/alberto/tesi/simulations/lbl-pkt-4/lbl-pkt-4.sf", "r");
#endif*/

#ifdef PRINT_THROUGHPUT_DISTRIB
        outTdistrib=fopen("res/throughput.txt", "w");
#endif

#ifdef PRINT_CACHE_DISTRIB
        outcachedistrib=fopen("res/cacheDistr.txt", "w");
#endif


        while (fgets(row, ROW_LENGTH,infile)!=NULL){
                fill(row);
                if (datagram.time<lastTimestamp){
                        datagram.time=lastTimestamp+MIN_DISTANCE;
                }
                //printf("%lf %lf\n", datagram.time, lastTimestamp);
                totalDatagrams++;
                bytes+=datagram.bytes;
                tmp=addSA(datagram.sourceIP, datagram.destIP);
                if (datagram.time!=datagram.theorTime){
                        delayed++;
                }

#ifdef PRINT_THROUGHPUT_DISTRIB
                if (datagram.time<throughTime+PRINT_THROUGHPUT_DISTRIB){
                        throughBytes+=datagram.bytes;
                }
                else{
                        fprintf(outTdistrib, "%.2lf %.2lf %.2lf %.2lf\n", throughTime,
                                8*(double)throughBytes/(datagram.time-throughTime), throughTheorTime,
                                8*(double)throughBytes/(datagram.theorTime-throughTheorTime));
                        throughTime=datagram.time;
                        throughTheorTime=datagram.theorTime;
                        throughBytes=datagram.bytes;
                }
#endif
```

```
                if (tmp!=NULL && tmp->counter==0){
                        printf("SA closed: max SA usage reached");
                        closeNumber(tmp);
                }

#ifdef USE_FIN_PACKETS
                checkFin();
#endif

#ifdef CLOSE_UNUSED
                if (datagram.time>=lastCheck+CHECK_TIME){//each CHEC_TIME sec, checks and closes
                                                        //all the SA not used for more than
                                                        //UNUSED_TIMEOUT secs
                        if (datagram.time-oldestConnTime>=UNUSED_TIMEOUT){
                                oldestConnTime=datagram.time;
                                for(tmpSAD=SAD; tmpSAD<SAD+MAX_SA; tmpSAD++){
                                        if(tmpSAD->sourceIP!=0 && tmpSAD->destIP!=0){
                                                if (datagram.time-tmpSAD->time>=UNUSED_TIMEOUT){
                                                        /*printf("%d %d %lf %lf\n",
                                                                tmpSAD->sourceIP,
                                                                tmpSAD->destIP,
                                                                datagram.time, tmpSAD->time);*/
                                                        closeNumber(tmpSAD);
                                                }
                                                else if (tmpSAD->time<oldestConnTime)
                                                        oldestConnTime=tmpSAD->time;
                                        }
                                }
                        }
                        lastCheck=(long)datagram.time;
                }
#endif  //CLOSE_UNUSED

        }


//computing some statistics on the discarded SAs
        /*howMany=0;
        for (i=0; i<MAX_DEL && !(deletedSA[i].sourceIP==0 && deletedSA[i].destIP==0); i++){
                howMany+=deletedSA[i].counter;
                if(deletedSA[i].counter==1)
                        discardedOnce++;
        }*/


#ifdef PRINT_THROUGHPUT_DISTRIB
        fprintf(outTdistrib, "%.2lf %.2lf %.2lf %.2lf\n", throughTime,
                (double)throughBytes/(datagram.time-throughTime), throughTheorTime,
                (double)throughBytes/(datagram.theorTime-throughTheorTime));
#endif


//closing the files
        fclose(infile);
#ifdef USE_FIN_PACKETS
        fclose(infileFin);
#endif

#ifdef PRINT_THROUGHPUT_DISTRIB
        fclose(outTdistrib);
#endif

#ifdef PRINT_CACHE_DISTRIB
        fclose(outcachedistrib);
#endif

//printing the results to standard error
        fprintf(stderr, "\n************************************************************\n");
        fprintf(stderr, "************************************************************\n");
        fprintf(stderr, "max number of SA: %d; cache size %d\n\n", MAX_SA, CACHE_SIZE);
        fprintf(stderr, "Time scale factor: %lf\n", SCALE_TIME);
        fprintf(stderr, "Channel initial time: %.3es\n", CHANNELL_INITIAL);
        fprintf(stderr, "Channell transfer time (burst of 4 bytes): %.3es\n", CHANNELL);
        fprintf(stderr, "Encryption setup time: %.3es\n", ENC_SETUP);
        fprintf(stderr, "Encryption time for a 128-bit block: %.3es\n", ENC_TIME);
        fprintf(stderr, "Cache access time: %.3es\n", CACHED_DELAY);
        fprintf(stderr, "CRC generation time for a byte: %.3es\n", CRC_TIME);
#ifdef USE_FIN_PACKETS
```

```
        fprintf(stderr, "Using FIN TCP packets for SA closing\n");
#endif

#ifdef CLOSE_UNUSED
        fprintf(stderr, "Closing connections after an unused timeout of %ds (checking each %ds)\n",
                                                            UNUSED_TIMEOUT, CHECK_TIME);
#endif
        fprintf(stderr, "***************************************************************\n\n");

        fprintf(stderr, "Total number of datagrams analyzed %d\n", totalDatagrams);
        fprintf(stderr, "Average dimension of datagrams (34bytes of header): %.2fbytes\n",
                                      (float)bytes/(float)totalDatagrams);
        fprintf(stderr, "Average data rate: %.3fkbit/s\n", 8*(float)(bytes)/lastTimestamp/1024);
        fprintf(stderr, "Theoretical average data rate: %.3fkbit/s\n",
                              8*(float)(bytes)/datagram.theorTime/1024);
        fprintf(stderr, "Total processing time %lfs\n", procTime);
        fprintf(stderr, "Average processing time %lfs\n", procTime/(double)totalDatagrams);
        fprintf(stderr, "Average connections managed per second: %.2f\n",
                                      (float)(totalDatagrams)/datagram.time);
        fprintf(stderr, "\nTotal cache misses: %d (%.2f\%)\n",      cacheMisses,
                                      100*(float)cacheMisses/(float)totalDatagrams);
        fprintf(stderr, "Compulsory misses: %d\n", compulsoryMisses);
        fprintf(stderr, "Avoidable cache misses: %d (%.2f\%)\n", cacheMisses-compulsoryMisses,
                                      100*(float)(cacheMisses-compulsoryMisses)/(float)totalDatagrams);
        fprintf(stderr, "\nNumber of delayed datagrams %d (%.2lf\%)\n", delayed,
                                                     (float)delayed*100/(float)totalDatagrams);


        return 0;
}
```

Appendix D. Network timing evaluation

```
//
//file: network.c
//
//program for computing statistics on the network delays
//
//written by Alberto Ferrante, April 2002

#include <stdio.h>
#include <string.h>
#include <limits.h>

//dimension of the data structures
#define ROW_LENGTH 70
#define MAX_EL 4000


struct dataT{  //data taken from file
       double time;
       int sourceIP;
       int destIP;
       int sourceTCP;
       int destTCP;
       int bytes;
};

struct statStruct{
       int sourceIP;
       int destIP;
       long counter;
       double maxTime;
       double minTime;
       double last;
       double sumTime;
       int replied;
};

char row[ROW_LENGTH];  //row of the data file
FILE *infile; //input file
struct dataT datagram; //data related to each IP datagram
struct statStruct conn [MAX_EL]; //
unsigned last;

double toDouble(char* base, char* end){
       double tmp=*base-48;
       int div=10;

       while(++base<end && *base!='.'){
              tmp=tmp*10+(*base)-48;
       }
       while(++base<end){
              tmp=tmp+((float)(*base-48))/div;
              div*=10;
       }

       return tmp;
}

int toInt(char* base, char* end){
       int tmp=(*base)-48;

       while(++base<end){
              tmp=tmp*10+(*base)-48;
       }

       return tmp;
}

void fill(char row[ROW_LENGTH]){ //fills the structure datagram with the data taken by file
       char *tmp=row;
       char *base=row;
       char *end;
```

```
        int length=strlen(row);

        /*setting datagram.time*/
        if ((end=strchr(row, ' '))<row+length){
                datagram.time=toDouble(base, end);
                base=end;
        }

        /*setting datagram.sourceIP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.sourceIP=toInt(base, end);
                base=end;
        }

        /*setting datagram.destIP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.destIP=toInt(base, end);
                base=end;
        }

        /*setting datagram.sourceTCP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.sourceTCP=toInt(base, end);
                base=end;
        }

        /*setting datagram.destTCP*/
        if ((end=strchr(++base, ' '))<row+length){
                datagram.destTCP=toInt(base, end);
                base=end;
        }

        /*setting datagram.bytes*/
        end=row+length-1;
        base++;
        datagram.bytes=34+toInt(base, end);
}

struct statStruct* searchConn(struct statStruct* DB, struct statStruct *end, int source, int dest){
                                             //searches an element in a SAD-like array
        struct statStruct* tmp=DB;

        while (tmp<end){
                if((tmp->sourceIP==source && tmp->destIP==dest) ||
                        (tmp->sourceIP==dest && tmp->destIP==source))
                        break;
                else
                        tmp++;
        }

        if (tmp<end)
                return tmp;
        else
                return NULL;
}


void addConn(){
        struct statStruct* which;

        if ((which=searchConn(conn, conn+last, datagram.sourceIP, datagram.destIP))==NULL){
                which=searchConn(conn, conn+MAX_EL, 0, 0);
                which->sourceIP=datagram.sourceIP;
                which->destIP=datagram.destIP;
                last++;
        }

        if ((which->sourceIP==datagram.sourceIP && which->replied==-1) ||
            (which->sourceIP==datagram.destIP && which->replied==1)){
                which->sumTime+=datagram.time-which->last;
                if (which->maxTime<datagram.time-which->last)
                        which->maxTime=datagram.time-which->last;
                if (which->minTime>datagram.time-which->last)
                        which->minTime=datagram.time-which->last;
                which->last=datagram.time;
                which->counter++;
                which->replied=0;
        }
        else{
                if (which->sourceIP==datagram.sourceIP)
                        which->replied=1;
                else
                        which->replied=-1;

                which->last=datagram.time;
        }

}

int main(void){
        struct statStruct* tmp;
        long connections;
        long zero;
        double totalTime;
        double maxTime;
        double minTime;

        last=0;
        connections=0;
        totalTime=0;
        maxTime=0;
        minTime=7200;
        zero=0;

        for (tmp=conn; tmp<conn+MAX_EL; tmp++){
                tmp->sourceIP=0;
                tmp->destIP=0;
                tmp->maxTime=0;
                tmp->counter=0;
                tmp->last=0;
                tmp->sumTime=0;
                tmp->replied=0;
                tmp->minTime=7300;
        }

        infile=fopen("/home/alberto/tesi/simulations/lbl-tcp-3/lbl-tcp-3.tcp", "r");
        //infile=fopen("/home/alberto/tesi/simulations/lbl-pkt-4/lbl-pkt-4.tcp", "r");

        while (fgets(row, ROW_LENGTH,infile)!=NULL){
                fill(row);
                addConn();
        }

        fclose(infile);

        for (tmp=conn; tmp<conn+last; tmp++){
                if (tmp->sourceIP!=0 && tmp->destIP!=0 && tmp->counter>0){
                         printf("%d %d %d %.4lf %.4lf %.4lf\n", tmp->sourceIP, tmp->destIP,
                                 tmp->counter, tmp->maxTime, tmp->minTime, tmp->sumTime/tmp->counter);
                         connections++;
                         totalTime+=tmp->sumTime/tmp->counter;
                         if (tmp->maxTime>maxTime)
                                 maxTime=tmp->maxTime;
                         if (tmp->minTime<minTime)
                                 minTime=tmp->minTime;
                }
                if (tmp->counter==0)
                        zero++;
        }

        printf ("Average reply time: %.4lf\n", totalTime/connections);
        printf ("Max reply time: %.4lf\n", maxTime);
        printf ("Min reply time: %.4lf\n", minTime);
        printf ("Datagram without any reply: %d\n", zero);

        return 0;
}
```