ABSTRACT

ABHIJIT N. HAYATNAGARKAR. On Realizing Traffic-driven Security Association Establishment for IPSec. (Under the direction of Dr. S. Felix Wu.)

The rapid growth of the Internet in the past few years has led to an exponential increase in the network traffic. As more and more organizations connect to the Internet, the security of the network and the applications that use it has become an important concern in the Internet community. The IP Security architecture (IPSec), proposed by the Internet Engineering Task Force (IETF), is aimed at providing security services to the network traffic at the IP layer.

The key aspect of secure communication between two machines in IPSec is the establishment of a Security Association (SA). A Security Association is a one-way relation between the sender and the receiver that provides security services to the traffic carried on it. Current implementations of IPSec provide support for the establishment of only *static* SAs i.e. they require that the SAs be established *before* any other network traffic starts to flow between the sender and the receiver.

These static SAs may be sufficient for applications such as the Virtual Private Network (VPN), where only a few SAs are needed. But certain advanced security applications potentially require the establishment and teardown of a large number of SAs dynamically. SA-establishment is a computation-intensive job, and such advanced security applications would benefit if SAs are established only when (and if) there is network-traffic between the sender and the receiver. This thesis deals with the motivation, design, software implementation and the performance measurement of a traffic-driven approach to dynamic IPSec SA-establishment.

Towards this, the design and implementation of a utility program, called DIANA, is presented. DIANA adds traffic-driven SA-establishment functionality to an existing implementation of IPSec called FreeS/WAN. DIANA maintains a Security Policy Database (SPdb), which specifies the policies that determine the processing of all outbound IP traffic. DIANA provides traffic-driven SA-establishment by intercepting outgoing IP packets from the operating system kernel, matching them with policies specified in the SPdb and establishing the SAs if a matching policy is found.

This thesis also presents some performance measurements for IP interception and DIANA. These measurements indicate that for most applications (notably those that use the Transmission Control Protocol (TCP)), the overhead of the traffic-driven approach to dynamic SA-establishment is minimal.

ON REALIZING TRAFFIC-DRIVEN SECURITY ASSOCIATION ESTABLISHMENT FOR IPSEC

by

ABHIJIT N. HAYATNAGARKAR

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER SCIENCE

Raleigh

May, 1999

APPROVED BY:

Douglas S. Reeves
Dr. D. Reeves

Gary Rouskas
Dr. G. Rouskas

S. Felix Wu
Dr. S. Felix Wu
Chair of Advisory Committee

DEDICATION

To my parents and my sister, Pradnya.

BIOGRAPHY

Abhijit Hayatnagarkar is a Masters student in the Computer Science Department at North Carolina State University. He earned his Bachelor of Technology in Computer Science and Engineering from the Indian Institute of Technology, Mumbai, India, in 1997. He is currently a research assistant in the Secure and Highly Available Networking Group (SHANG) at North Carolina State University.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Felix Wu, for his constant support, encouragement and advice during my stay at North Carolina State University. Working with him has been a highly rewarding experience and I really enjoyed the many hours of brain storming, of which this thesis is a result. I would also like to thank Mr. Chandru Sargor, Member of Technical Staff at MCNC, for providing me valuable guidance in the design of DIANA. My special thanks to other members of the DECIDUOUS project – Mr. Ho-Yen Chang, Mr. Chien-Lung Wu and Mr. Ravindra Narayan – for providing me valuable tips throughout the duration of this thesis. I would also like to thank all the members of the SHANG group for helping me in numerous ways during my research assistantship. Finally, I would like to thank my thesis-committee members – Dr. D. Reeves and Dr. G. Rouskas – for the support and valuable time they have provided for this thesis.

Contents

List of Tables

List of Figures

List of Abbreviations

[AH]	Authentication Header
[API]	Application Program Interface
[ASIS]	Attack Source Identification System
[DECIDUOUS]	Decentralized Identification of Intrusion Sources
[ESP]	Encapsulating Security Payload
[IDS]	Intrusion Detection System
[IETF]	Internet Engineering Task Force
[IKE]	Internet key Exchange
[IP]	Internet Protocol
[IPSEC]	IP Security
[ISAKMP]	Internet Security Association and Key Management Protocol
[PGP]	Pretty Good Privacy
[RFC]	Request For Comments
[SA]	Security Association
[SET]	Secure Electronic Transaction
[SADB]	Security Association Database
[SPDB]	Security Policy Database
[TCP]	Transmission Control Protocol
[TLS]	Transport Layer Security
[UDP]	User Datagram Protocol
[VPN]	Virtual Private Networks

Chapter 1

Introduction

The Internet traffic has been increasing exponentially for the past few years [5], and is poised to grow even further. TCP/IP traffic is a major component of this Internet traffic. With business-critical applications now running on the Internet, the security of the network traffic has become an important consideration in the Internet community. Several ideas have been proposed to the Internet Engineering Task Force (IETF) for securing network communications. Chief among them are Transport Layer Security (TLS) and IP Security (IPSec). The IPSec architecture [9] and various other underlying protocols have been standardized by the IETF. This architecture provides security services to the network traffic at the IP layer.

As more and more corporations and businesses get connected to the Internet, dealing with network based intrusions becomes an important goal for the network security community. During the past few years, some progress has been made in the areas of attack prevention and intrusion detection technologies. These security services are very valuable to protect hosts from malicious attacks over the network by an intruder. However, identifying the sources of such attacks remains a difficult task. Today, the system administrator has to rely on tools such as `traceroute`, `finger` and `tcpdump` for tracing an attacking source.

Many times, the intruder is from a network in a different administrative domain. By the time the system administrators of different domains communicate and start to collaborate in the event of an attack, a sophisticated intruder may escape. What is needed is an automatic method for tracking down intrusion sources. The DECIDUOUS project discussed in chapter 3 provides such a framework for identifying network-based intrusion sources. DECIDUOUS uses the IPSec infrastructure to track these sources of network-based

intrusions.

A key concept in IPSec is the **Security Association**, which defines the parameters for secure communication between two machines. Advanced network-security applications such as DECIDUOUS would benefit if the establishment of Security Associations can be made **traffic-driven** i.e. if Security Associations are established only when (and if) there is network traffic between two communicating machines.

This thesis discusses the motivation, design and implementation of DIANA, a utility for traffic-driven Security Association establishment. Chapter 2 briefly discusses the architecture of IPSec. Chapter 3 looks at the DECIDUOUS framework and provides the motivation for traffic-driven Security Association establishment. Chapters 4 and 5 look at the design and implementation of DIANA. Chapter 6 presents some performance results of DIANA and chapter 7 concludes this thesis.

Chapter 2

IPSec Overview

The network security community has developed application-specific security mechanisms in a number of application areas. The most prominent are electronic-mail (PGP), client/server (Kerberos), web access (Secure Socket Layer) and electronic-commerce (SET). However, there are some security concerns that cut across all protocol layers and applications. For example, an organization might want to connect its geographically remote sites via Internet, yet prohibit others from reading the data being transferred between these sites. By implementing security at the IP level, such an organization can ensure security not only for applications having security mechanisms but also for applications which are security-ignorant. This results in a Virtual Private Network (VPN) for the organization as shown in figure 2.1. This chapter discusses the IP Security architecture (IPSec), standardized by IETF, which tries to achieve security at the IP layer.

2.1 Security Services

IPSec architecture [9] aims to provide security services like encryption and authentication for network traffic at the IP layer. It is designed to provide inter-operable, high quality, cryptographically-based security for IPv4 and IPv6. It provides security services such as:

- **Access Control**: Only authorized users should have access to a particular network service.

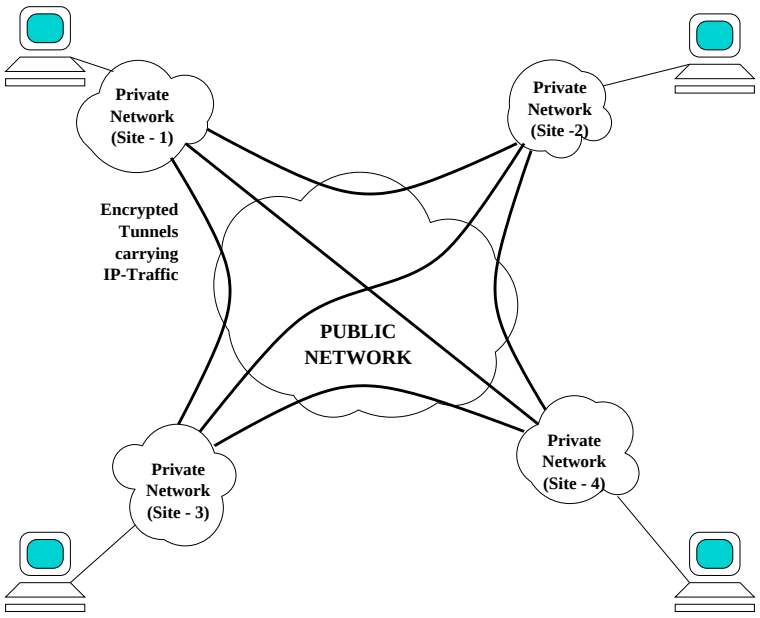


Figure 2.1: A VPN using IPSec in Tunnel Mode

- **Connectionless Integrity**: Integrity means that data should remain unaltered when it reaches the destination.
- **Data origin Authentication**: The receiver of data should be able to verify the origin of the data.
- **Protection against Replays**: If some data is sent back to the receiver, it should be able to identify that data as a replay.
- **Confidentiality**: No one other than the intended recipients should be able to read the data being transferred.

An IPSec implementation can operate in either a host or a security gateway environment. A *security gateway* refers to an intermediate system that implements IPSec protocols. It provides security to transit IP traffic. IPSec can be used to provide security between two hosts, between two security gateways or between a host and a security gateway.

2.2 Protocols

IPSec provides security services by means of two protocols: an authentication protocol called **Authentication Header (AH)** and a combined encryption/authentication protocol called **Encapsulating Security Payload (ESP)**. These protocols are supported by cryptographic key management procedures and protocols.

The IP Authentication Header (AH) [7] protocol provides access control, connectionless integrity, data origin authentication, and an anti-replay service.

The Encapsulating Security Payload (ESP) [8] protocol primarily provides confidentiality (encryption). It may also provide access control, connectionless integrity, data-origin authentication and an anti-replay service.

These protocols may be applied alone or in combination with each other to provide a desired set of security services. Each protocol can operate in two modes: **transport mode** and **tunnel mode**. Transport mode (figure 2.2) provides protection primarily for upper-layer protocols like TCP, UDP or ICMP. Tunnel mode (figure 2.1) provides protection to the entire tunneled IP packet.

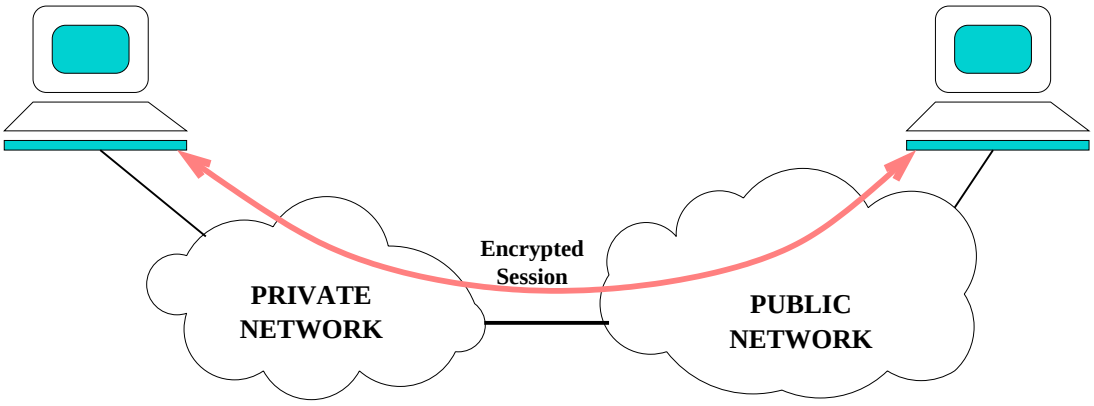


Figure 2.2: ESP in Transport Mode

These two security protocols use shared secret values (cryptographic keys). To put these keys and other parameters in place, IPSec requires support for both manual and automatic distribution of keys. IPSec specifies a *public-key cryptography* based approach called **Internet Key Exchange** (IKE) for automatic key-distribution.

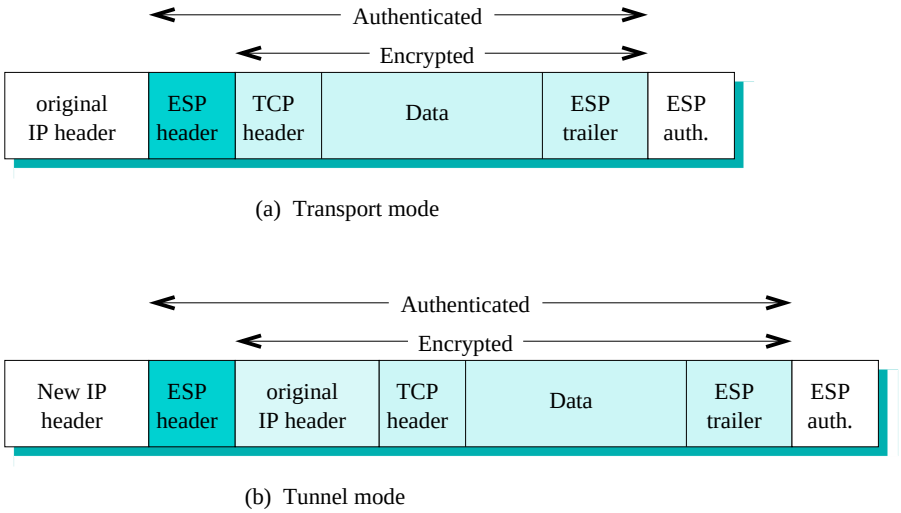


Figure 2.3: ESP in transport and tunnel mode for IPv4

2.3 Security Associations

A key concept that appears in both authentication and confidentiality mechanisms for IPSec is the **Security Association (SA)**. Both AH and ESP make use of SAs and a major function of IKE is the establishment and maintenance of Security Associations. *A Security Association is a simplex (one-way) relationship between the sender and the receiver that provides security services to the traffic carried on it.* An SA defines the algorithms, keys and other parameters for secure communication between two machines.

Two types of SAs are defined: **transport mode** and **tunnel mode**. A transport mode SA is a security association between two hosts. In the case of ESP, a transport mode SA provides security services only for the higher layer protocols, but not for the IP header. This is shown in figure 2.3. In the case of AH, the protection is also extended to selected portions of the IP header (figure 2.4).

A tunnel mode SA is essentially an SA applied to an IP tunnel. Whenever either end of an SA is a security gateway, the SA must be tunnel mode. For a tunnel mode SA, there are two IP headers: "inner" and "outer" (shown as *original* and *New* IP headers, respectively, in figures 2.3 and 2.4). The "inner" header specifies the ultimate destination of the packet and the "outer" header specifies the IPSec processing destination. The security protocol header appears between these two headers. If AH is employed in tunnel mode,

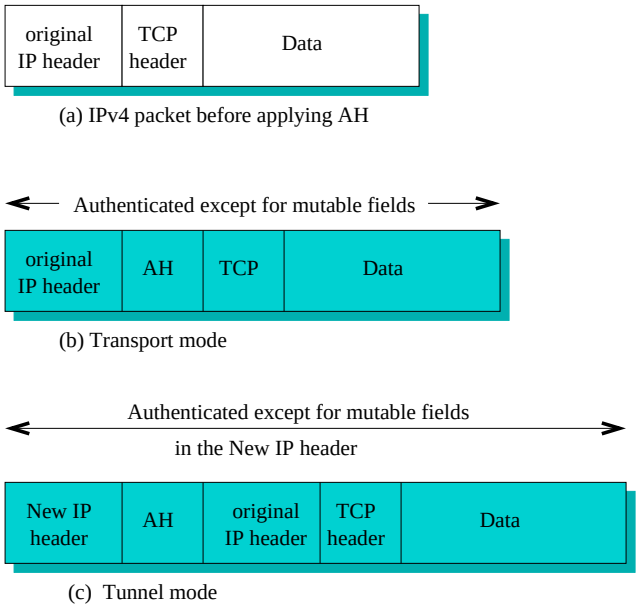


Figure 2.4: AH in transport and tunnel mode for IPv4

portions of the outer IP header are afforded protection, as well as all of the tunneled IP packet. If ESP is employed, the protection is afforded only to the tunneled packet, not to the outer header. These scenarios are shown in figures 2.3 and 2.4.

IPSec allows the user to control the **granularity** at which a security service is to be offered. For example, one can create a single encrypted tunnel to carry all the traffic between two security gateways or a separate encrypted tunnel can be created for each TCP connection between each pair of hosts communicating across these gateways. Fine granularity SAs are generally more vulnerable to traffic analysis than coarse granularity ones which carry traffic from many users.

2.4 Combining Security Associations

An individual SA is offered protection by *exactly* one security protocol: either AH or ESP, but not both. Sometimes, a security policy may require a combination of security services that is not achievable by a single protocol. To take care of such instances, IPSec defines ways to combine SAs into "bundles".

There are two ways in which SAs can be combined into bundles: **transport adjacency** and **iterated tunneling**. Transport adjacency refers to applying more than one security protocol to the same IP datagram, without invoking tunneling. Iterated tunneling refers to the application of multiple layers of security protocols effected through IP tunneling. This approach allows multiple levels of nesting, since each tunnel can originate or terminate at different IPSec site along the path. These two types of approaches can also be combined to give more possibilities. These combinations are discussed in detail in RFC 2401 which deals with the IPSec Architecture [9].

2.5 Security Association Databases

Two databases play a crucial role in the IPSec model: **Security Policy Database (SPdb)** and the **Security Association Database (SAdb)**. SPdb specifies the policies that determine the processing of all IP traffic, inbound or outbound, from a host or security gateway. These policies are typically determined by the system administrator at the security gateway or at the host. SAs are set up by the IPSec implementation depending on the policies in SPdb. SAdb contains parameters (keys, algorithms etc.) that are associated with each (*active*) SA. SAdb is typically a data-structure maintained by the IPSec implementation.

2.5.1 Security Policy Database

The *Security Policy Database* (SPdb) contains an ordered list of policy entries. Each policy entry consists of two parts: a *condition* and an *action*. The condition part consists of **selectors** which define the set of IP traffic encompassed by this policy entry. The action part specifies whether the traffic matching this policy will be bypassed, discarded or subject to IPSec processing. If the packets are to be IPSec processed, the details of security operations to be performed on packets which match this policy entry are also specified.

The *selectors* define the granularity of an SA. The following selectors are supported by IPSec architecture to facilitate the control of SA granularity:

- **Destination IP Address**: This is the destination address present in the outermost IP header of the packet. It can be a network address or a wild-card.

- **Source IP Address**: This is the source address present in the outermost IP header of the packet. It can be a network address or a wild-card.
- **Name**: This can either be a User ID or a System Name. One of the possible values is 'Opaque', which means that this field will not be considered while matching the policy entry with the packet.
- **Transport Layer Protocol**: This is obtained from the IPv4 "protocol" or the IPv6 "Next Header" fields.
- **Source and Destination Ports**: These may be individual TCP or UDP ports or wild-card values. A value of 'Opaque' is supported since the port numbers may not be available for all IP packets (e.g. IP packets encrypted in Tunnel Mode ESP).

The actual structure of SPdb and its management interface are left to the implementor.

2.5.2 Security Association Database

A *Security Association Database* (SAdb) contains entries which define the parameters associated with each (*active*) SA. Each SA has an entry in the SAdb. *If some policy in the SPdb does not currently point to an SA that is appropriate for a packet, then the IPSec implementation creates an appropriate SA for that policy.*

For incoming IP packets, each entry in the SAdb is indexed by the outer header's destination IP Address, IPSec protocol type (AH or ESP) and Security Parameters Index (SPI). The SPI is a 32-bit value used to distinguish among different SAs terminating at the same destination and using the same IPSec protocol. The following SAdb fields are used in IPSec processing:

- **Sequence Number Counter**: A 32-bit value used to generate the 'Sequence Number' field in AH or ESP headers.
- **Sequence Counter Overflow**: A flag indicating the overflow of the Sequence Number Counter.
- **Anti-Replay Window**: A 32-bit counter and a bit-map used to determine whether an inbound AH or ESP packet is a replay.
- **AH authentication algorithm, keys etc.**

- **ESP encryption algorithm, keys, initialization vector etc.**
- **ESP authentication algorithm, keys etc.**: This field can be NULL if the ESP Authentication service is not selected.
- **Lifetime of this SA**: A time interval after which an SA must be replaced with a new SA (and new SPI) or terminated, plus an indication of which of these actions should occur.
- **IPSec protocol mode: tunnel, transport or wild-card**. Indicates which mode of AH or ESP is applied to traffic on this SA. The "wild-card" value at the sending end means that the application has to specify the mode to the IPSec implementation.

Section 2.7 describes how these databases are used in IPSec processing.

2.6 Key Management

Secure communication requires that the sender and receiver share some secret keys. The IPSec Architecture supports two types of key management: **manual** and **automated**. In manual key management, the system administrator manually configures each system with its own keys and with keys of other communicating systems. An automated system on the other hand enables the on-demand creation and distribution of keys by software. The default automated key management protocol for IPSec is called ISAKMP/IKE. It defines the mechanisms and message formats for exchanging keys securely. It consists of the following two elements:

- **Internet Security Association and Key Management Protocol (ISAKMP)**: ISAKMP is designed to provide a flexible and extensible framework for establishing and managing Security Associations and cryptographic keys.
- **Internet Key Exchange (IKE)**: The purpose of IKE is to negotiate and provide authenticated keying material for security associations, in a protected manner.

2.6.1 ISAKMP

ISAKMP defines the procedures and packet formats to establish, negotiate, modify and delete SAs. The packet formats provide a consistent framework for transferring key

and authentication data which is independent of the key generation technique, encryption algorithm and authentication mechanism. It is distinct from key exchange protocols (such as IKE) in order to cleanly separate the details of security association management from the details of key exchange.

ISAKMP is not bound to any specific cryptographic algorithm, key generation technique, or security mechanism. This independence from specific security mechanisms and algorithms provides a forward migration path to better mechanisms and algorithms. ISAKMP provides protection against denial of service, replay, man-in-the-middle and connection hijacking attacks.

An SA-establishment between two entities using ISAKMP takes place in two phases. In **phase** 1, the two ISAKMP peers establish a secure, authenticated channel with which to communicate. This secure channel is called the ISAKMP-SA. ISAKMP utilizes *digital signatures*, based on public key cryptography, for authentication. In **phase** 2, SAs are negotiated on behalf of IPSec protocols. These SAs are called *IPSec*-SAs. A single phase 1 negotiation may be used for more than one phase 2 negotiation. A single phase 2 negotiation can request multiple SAs.

When an ISAKMP-SA is initially established, one side assumes the role of the initiator and the other side the role of the responder. Once the SA is established, both the original initiator and responder can initiate a phase 2 negotiation with the peer entity. Thus, ISAKMP-SAs are bidirectional in nature.

ISAKMP allows both initiator and responder to have some control during the negotiation process. The initiator maintains control by specifying a set of *proposals* based on the initiators local security policy. The responder, on receiving these proposals, selects the proposal best suited for its local security policy and returns this selection to the initiator. An **SA proposal** consists of the security protocols and associated security mechanisms to be used for the SA being negotiated.

ISAKMP specifies various payload formats and exchange types to accomplish SA negotiation. The payloads provide building blocks for constructing ISAKMP messages. The exchange types give the basic syntax of a message exchange and allow the transmission of SA, key exchange and authentication information. ISAKMP also specifies the processing for each payload. The details about this protocol can be found in [10].

2.6.2 IKE

The IKE protocol is based on Diffie Hellman algorithm [11]. It provides additional security by the use of *nonces* and *digital signatures* using public key cryptography. The use of nonces provides protection from replay-attacks.

IKE defines three modes for doing SA negotiation: main mode, aggressive mode and quick mode. The main mode and aggressive mode are used to establish an authenticated key exchange. The quick mode is used as a part of phase 2 of ISAKMP to derive keying material for IPSec-SAs. IKE supports the use of different **groups** for the Diffie-Hellman key exchange. The details about the modes and groups can be found in [4].

2.7 IP Traffic Processing

This section describes the IPSec processing performed on IP packets. The SPdb is consulted during the processing of all IP traffic (inbound and outbound).

2.7.1 Outbound IP Traffic Processing

Each outbound packet is compared against the SPdb entries to determine what processing is required for the packet. A "matching entry" in the SPdb may specify that the packet be discarded, allowed to bypass IPSec or IPSec-processed. If the packet is allowed to bypass IPSec, it continues through the "normal" processing. If IPSec processing is required, the packet is either mapped to an existing SA (or SA bundle) or a new SA (or SA bundle) is created for the packet. The creation of a new SA would involve the ISAKMP/IKE protocols discussed above. If a new SA is created, a corresponding entry is added to the SAdb. The packet is then processed (encrypted, authenticated etc.) depending on the appropriate entries in the SAdb and transmitted.

2.7.2 Inbound IP Traffic Processing

Prior to performing AH or ESP processing, any IP fragments are reassembled. The mapping of the IP datagram to the appropriate SA is simplified due to the presence of the SPI value in the AH or ESP header. The SPI value, destination IP address and security protocol (AH or ESP) uniquely determine an SA. If a "matching" SA is not found in the SAdb, the packet is dropped. If a "matching" SA is found, IPSec processing (authentication,

decryption etc.) is done. This step includes matching the packet's selectors to the selectors in the SA. After this, an incoming policy is found in the SPdb that matches the packet. This policy is used to check whether the required IPSec processing has been applied to the packet. After all these steps, the IP packet is forwarded or passed to the transport layer.

This chapter discussed the IPSec architecture. The basic aim of IPSec is to provide security to IP traffic. But, the IPSec infrastructure can also be used for other advanced applications. The next chapter discusses such an application, called DECIDUOUS, which uses the IPSec/ISAKMP mechanisms to track down sources of attack on the Internet.

Chapter 3

Motivation

The previous chapter looked at the IPSec architecture. This chapter focuses on an advanced security application called *DECIDUOUS* and provides motivation for the thesis. DECIDUOUS (**DEC**entralized **ID**entification of intr**U**sion s**OU**rce**S**) is a security management framework for identifying the sources of network-based intrusions. DECIDUOUS is built on top of IETF's IPSec/ISAKMP infrastructure, which was discussed in the previous chapter. It uses the IPSec authentication service in tunnel mode to trace the source of an attack. Such a sophisticated mechanism is needed because *the source IP address cannot be trusted*.

3.1 Network-based Intrusions

Network-based attacks are of two types: *hit-and-run* and *persistent*. Hit-and-run attacks require only one bad packet to cause damage to the system. It is difficult to trace down the attack source for hit-and-run attacks. By the time, the damage caused by such an attack is discovered, the intruder can hide himself or herself. One possible way to catch such attacks is to turn on *all* intrusion detection and authentication options *all* the time. But, this is very expensive and slows down the network performance even in the absence of attacks. The other type of attack, viz. *persistent*, requires the intruder to attack the system repeatedly. This makes it easier for the victim to trace down the intruder.

At present, system administrators use ad-hoc methods to trace down the attacker in the event of network-based intrusions. Typically, after detecting an intrusion, the network administrator would use tools like `tcpdump`, `traceroute` and `finger` and check various log

files on the system. If the suspected attack source is in another administrative network domain, he or she will contact the administrator in that domain for some help. This is time consuming, and it is not always effective in tracking down the attacker.

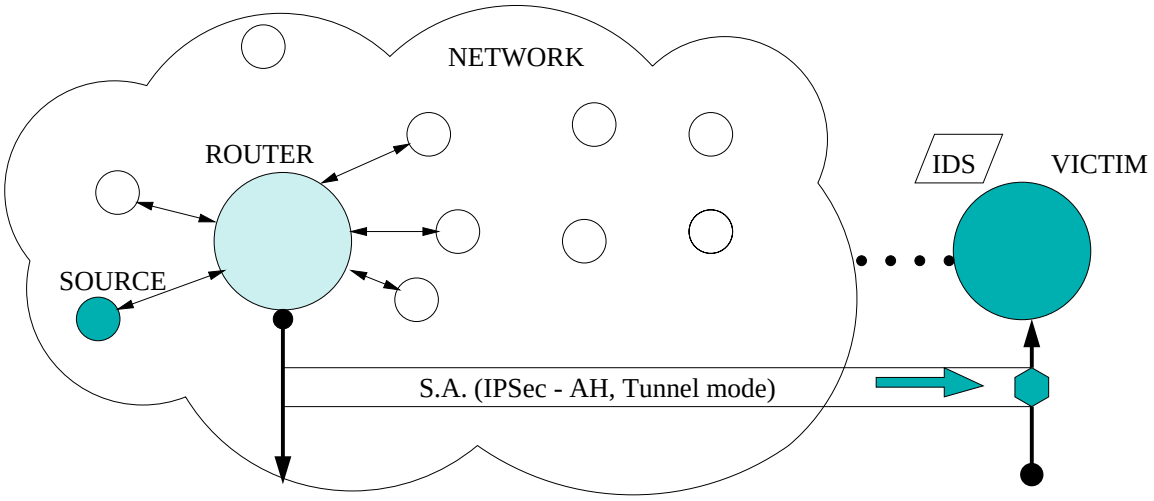
In order to deal with network-based intrusions systematically, three logical security components need to collaborate:

- **Intrusion Detection System (IDS)**: An Intrusion Detection System decides, with some probability, whether an observed network traffic is an **attack instance** or not.
- **Attack Source Identification System (ASIS)**: The IDS, in general, does not know the attack source(s) even after it detects an attack. This is so because the IP source address cannot be trusted. The objective of ASIS is to utilize the information provided by IDS to identify where the attacks are coming from. In DECIDUOUS, the ASIS module tries to force the attacker to reveal some new location information about the attack source(s) every time an attack instance is launched. Thus, it becomes very effective in the case of persistent attacks mentioned above.
- **Intrusion Damage Control System (IDCS)**: The objective of an IDCS is to control and repair the damage caused by the attacks detected by IDS. IDCS is very important in handling hit-and-run attacks. In particular, if IDCS can repair the damage caused by such attacks in real-time, the attacker will be forced to re-launch the attack or give up on attacking. If the attacker launches the attack multiple times, then this becomes a persistent attack. The ASIS will then be able to track down the attacker.

3.2 Attack Source Identification

As mentioned above, the main idea in DECIDUOUS is to use the IPSec authentication mechanisms to locate the source of attack-packets. The IPSec Authentication Header (AH) protocol in tunnel mode is well suited for this purpose.

For example, consider a victim having an IPSec-AH tunnel mode Security Association with some router. If the attack packet is authenticated correctly by the router, it *must* have been forwarded or generated by the router. This is the main principle on which the ASIS of DECIDUOUS is based. This situation is depicted in figure 3.1.



If the attack packet is authenticated correctly by the router,
it MUST have been forwarded or generated by the router.

Figure 3.1: ASIS Principle

If the victim has multiple Security Associations established with routers across the network, then, by looking at the authentication information in the attack packets, it can gain some information on the source of the attack. For example, consider the linear network topology in figure 3.2. Here, the victim has established Security Associations with routers A and B. If the attack packet is authenticated *only* by B, then the victim can conclude that the attack source is either from router A or router B or the network link between these two routers. (Note that this assumes correct routing behavior on the part of routers. i.e. *with correct network topology information, a good router will always forward the packets through the shortest path*).

This reasoning can be extended to general network topologies. Instead of using a single Security Association with one router, the victim would then establish Security Associations with a set of routers on a "cut" as shown in figure 3.3. A *d-distance cut* for a given host is a set of routers having a shortest path of length d from that host. Such a cut divides the network into three parts – the *cut*, the *inner part* and the *outer part*. The inner part is the set of routers having a shortest path of length less than d from the host. The outer part is the set of routers having a shortest path of length greater than d from the host. For example, consider a victim that has established tunnel-mode IPSec-AH

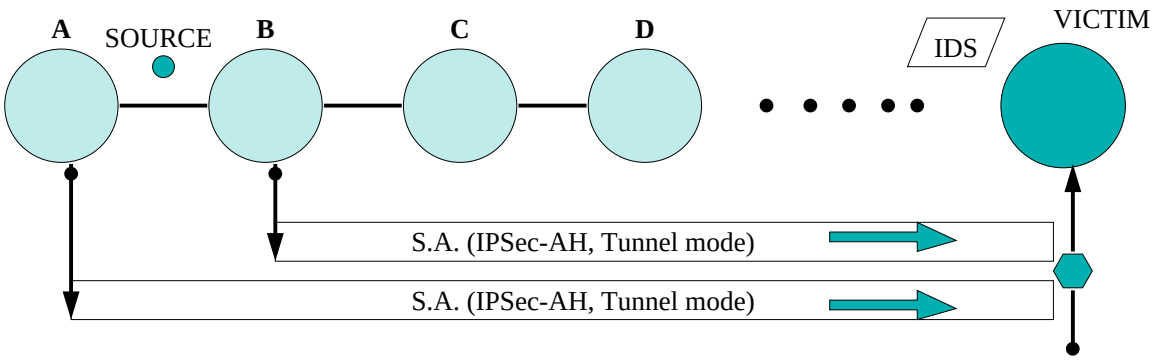


Figure 3.2: Locating an attack source

Security Associations with all routers on a *cut*. Now, depending on whether the attack packet is authenticated by some router on the cut or not, the victim can conclude whether the source of attack is on the cut, in the inner part or in the outer part. This situation is shown in figure 3.3. After this, taking different cuts and establishing appropriate Security Associations with routers on these cuts, the victim can narrow down the suspected area of the source of attack.

This method works very well if the network topology information is available to the victim. If the source of attack is in a different network domain, a collaboration protocol among DECIDUOUS ASIS daemons running in different network domains is needed. A discussion of such a protocol is outside the scope of this thesis.

DECIDUOUS is also capable of handling multiple attacking points (even if they coordinate) at the same time. More information on this can be found in [14].

As seen above, the approach of locating attack sources as used in DECIDUOUS, requires the victim to set up a large number of security associations. This poses a severe overhead for the victim. The concept of *traffic-driven* Security Association establishment described in the next section attempts to reduce this overhead.

3.3 Traffic-driven Security Association Establishment

To establish a Security Association, two communicating IPSec systems need to first securely exchange keys (assuming automatic key management). Then they need to update their respective Security Association Databases so that the IPSec traffic is pro-

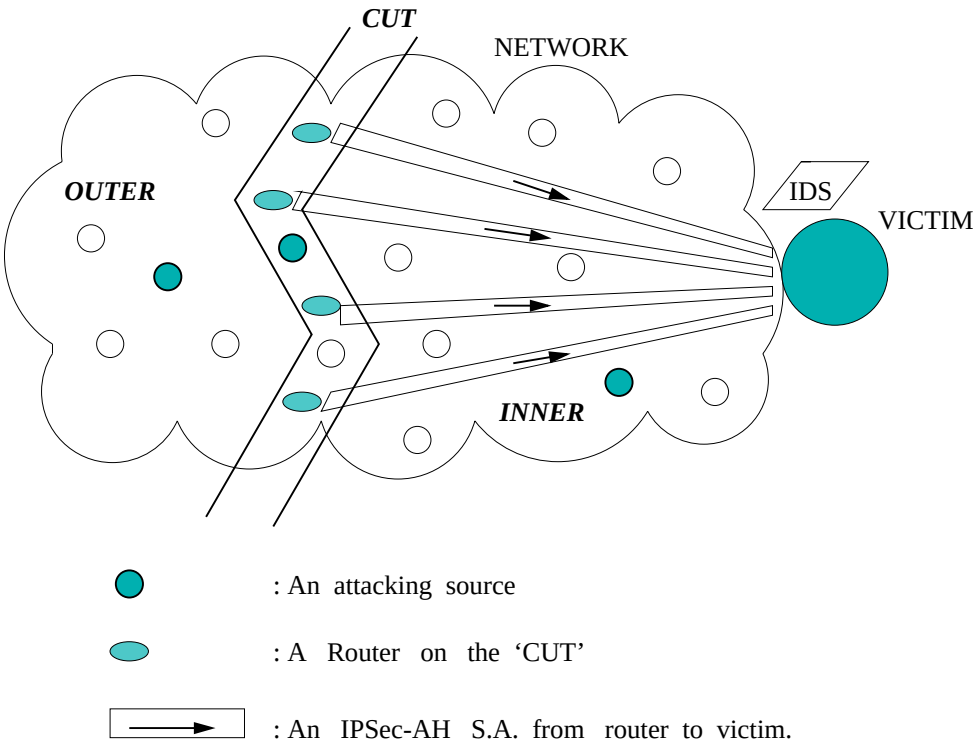


Figure 3.3: A *CUT* for locating attack sources

cessed properly. Though, IPSec is implemented in the kernel, the key-exchange mechanisms (ISAKMP/IKE) are typically implemented in the user-space. This means that there is a lot of overhead in setting up a Security Association, and maintaining its parameters.

Such an overhead is acceptable for most network applications, where only a few Security Associations are required and there is network traffic for most of the Security Associations established. But consider a network application, such as DECIDUOUS, which needs to communicate securely, but does not know in advance the source or destination of the network traffic. To cover all possible sources and destinations which may be of interest, such a network application will need to create a large number of Security Associations. This will incur unnecessary computational overheads. This will especially become significant, if only a few (possibly none) of the established Security Associations are ever used. If the IPSec system were to look into the future and determine which Security Associations will actually be used then it would create only those Security Associations. Unfortunately, the IPSec system has no way of knowing this. Thus, there is a wastage of resources in setting

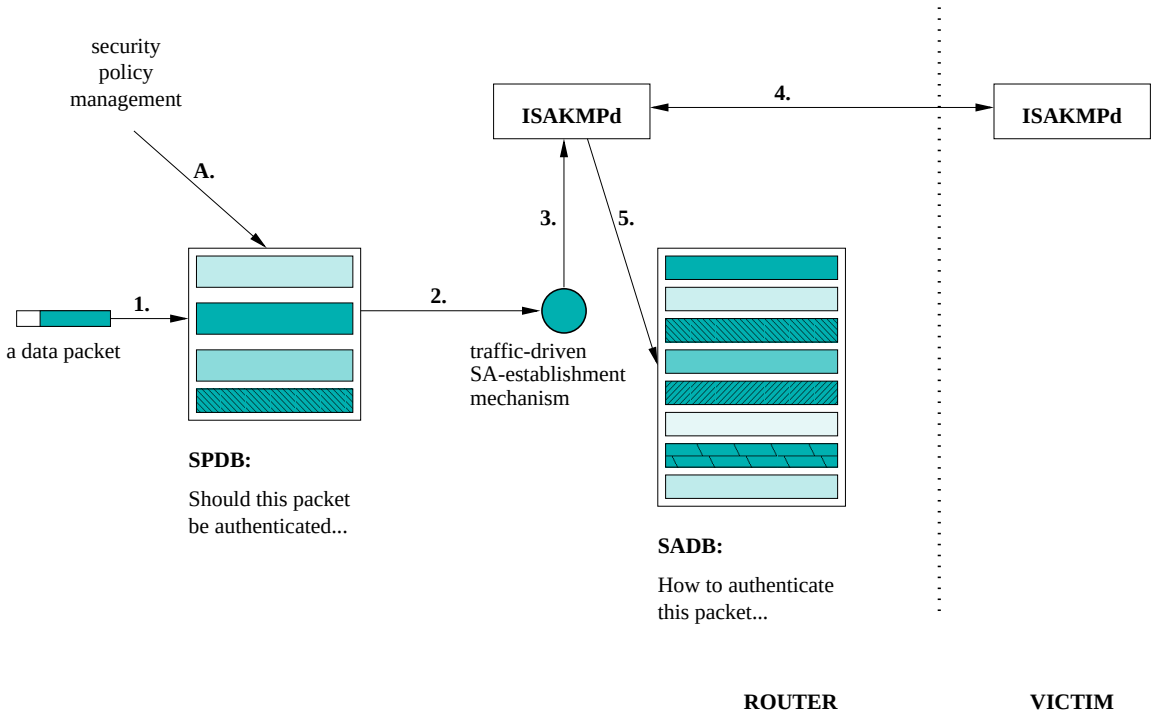


Figure 3.4: Traffic Driven SA Establishment

up Security Associations which will never be used.

Hence, for applications needing to create a lot of Security Associations, the *static* method of establishing SA's beforehand is insufficient. What is needed is a dynamic way to establish Security Associations whereby the SAs are established only when (and if) there is network traffic flowing for those Security Associations. The network traffic will thus act as a *trigger* for establishing Security Associations. This will eliminate the overheads associated with the creation of unnecessary Security Associations.

Such a scenario is shown in figure 3.4. Here, the router's security policy database contains policies specifying that packets destined to the victim need to be authenticated. When the first data packet arrives, the traffic-driven mechanism establishes an SA with the help of ISAKMP daemons. Then, the SADB is modified to reflect the newly created SA. Finally, the data packet is IPSec-processed according to the information in SADB and forwarded to the victim.

For the router to add an appropriate security policy to its SPDB, the victim needs to communicate with the router to set up the security policy. For this, a protocol called

Security Policy Update Protocol is needed. A discussion of such a protocol is beyond the scope of this thesis.

This thesis focuses on the development of a mechanism for *traffic-driven Security Association establishment*. There are two main issues in developing such a mechanism. The first issue is how to incorporate such a mechanism in the existing IPSec implementation. The second issue is that of performance – what is the overhead of dynamically setting up a Security Association and is this overhead low enough to justify introducing such a traffic-driven mechanism. The following two chapters discuss the design and implementation of DIANA, a utility program for establishing traffic-driven Security Associations. Chapter 6 looks at the performance characteristics and overheads of this mechanism.

Chapter 4

DIANA Design

This chapter discusses the design of a utility for establishing traffic-driven Security Associations. This utility is called DIANA (rhymes with the first two syllables of *Dyna*mic Security Association). This chapter first discusses the context of DIANA. Then, it looks at the current implementation support. Finally, it describes the design decisions and the high-level design of DIANA.

4.1 Context of DIANA

DIANA needs to establish traffic-driven Security Associations. For this, it needs to monitor outbound IP traffic. It must also have knowledge about the Security Policies in the Security Policy Database (SPdb). Finally, it should have some interface with Security Association-establishing mechanism within IPSec. Thus, the function of DIANA is to monitor outbound IP traffic and compare it with policies in SPdb for which Security Associations have not been established. If it finds some IP packet for which a Security Association needs to be created, it will trigger the IPSec/ISAKMP mechanism to establish that Security Association. This scenario is depicted in figure 4.1.

The network application (henceforth called the *Client User Process*) which needs to create multiple Security Associations just updates the SPdb. It has a well-defined interface for modifying the SPdb. By doing this, it can dynamically control the Security Policies. The Client User Process need not concern itself with whether the Security Associations are actually established or not. DIANA will see to it that these Security Associations are established if there is a *matching* traffic.

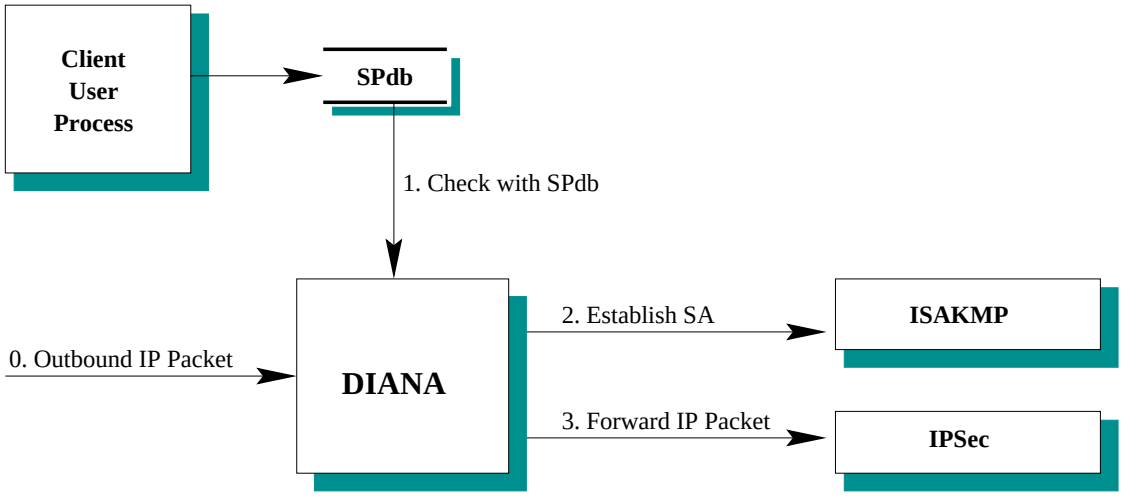


Figure 4.1: Context of DIANA

4.2 Current Implementation Support

An implementation of IPSec called *FreeS/WAN* [13] is available on the Internet. It supports Automatic Key Management by the use of ISAKMP daemons called *Pluto*. Two Plutos communicate with each other securely via an ISAKMP-Security Association by means of pre-shared keys. Plutos establish an IPSec-Security Association by exchanging session-keys and by informing IPSec about these keys and other Security Association parameters. The IPSec system then enters this information in its Security Association Database (SAdb). Pluto supports only tunnel modes for AH and ESP.

FreeS/WAN implementation is available for the Linux operating system. Pluto is implemented in user space while IPSec is implemented in the kernel. In this implementation a device called /dev/ipsec is created for user processes (like Pluto) to communicate with the kernel about IPSec information. Pluto does not initiate session-key negotiation with peer Pluto processes (on other machines). A program called *whack* has been provided to trigger ISAKMP negotiations. FreeS/WAN does not support traffic-driven Security Association establishment and one has to statically establish IPSec Security Associations before secure communication begins.

There is no Security Policy Database (SPdb) support in FreeS/WAN. The default policy is to allow packets if there is no Security Association already established with the

peer machine. The FreeS/WAN implementation scenario is shown in figure 4.2.

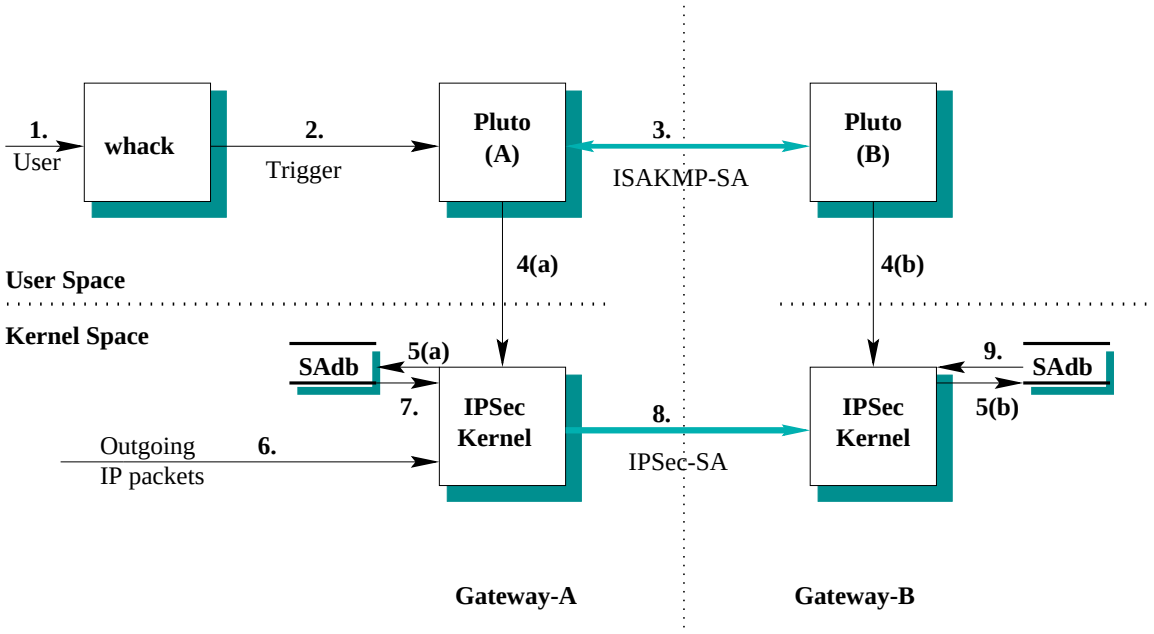


Figure 4.2: Current IPSec implementation of FreeS/WAN

The sequence of events as shown in figure 4.2 are:

1. User runs "Whack"
2. "Whack" triggers Pluto to establish Security Association
3. Pluto exchanges Security Association information with peer Pluto on another security gateway
4. The two Plutos on security gateways convey the Security Association information to their respective kernel-IPSec modules
5. IPSec kernels update their respective SAdbs with the new information
6. A *matching* packet arrives at security gateway A
7. The SAdb is consulted by the sender IPSec kernel to retrieve necessary Security Association information, and the packet is processed accordingly

8. The packet is sent on the IPSec Security Association

9. The receiver IPSec kernel uses information in its SAdb to process this incoming IP packet

4.3 IP Interception

DIANA was built on top of the existing FreeS/WAN implementation. As the function of DIANA is to trigger Security Association establishment on sensing appropriate IP traffic, DIANA took the place of "whack" in figure 4.2. Hence, the first design decision was to implement DIANA in the user space. Though this simplified the design of DIANA, it meant that an interception mechanism (in Linux) to transfer IP packets back and forth from the kernel is required. A discussion of the implementation of such an interception mechanism can be found in [15]. For the purpose of this thesis, we shall assume that the following mechanism exists for intercepting IP packets from the kernel:

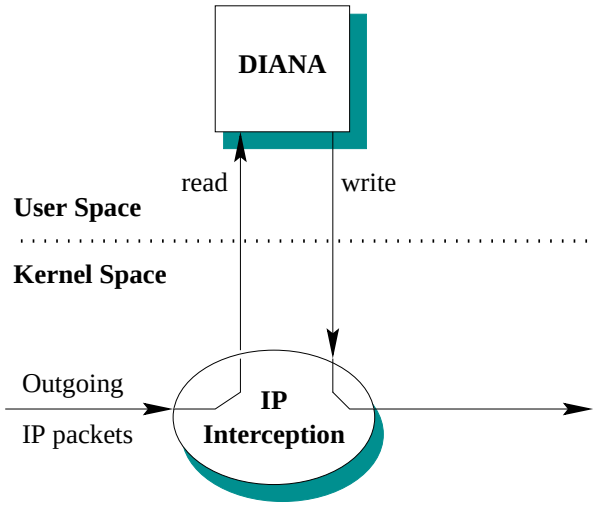


Figure 4.3: IP-packet interception

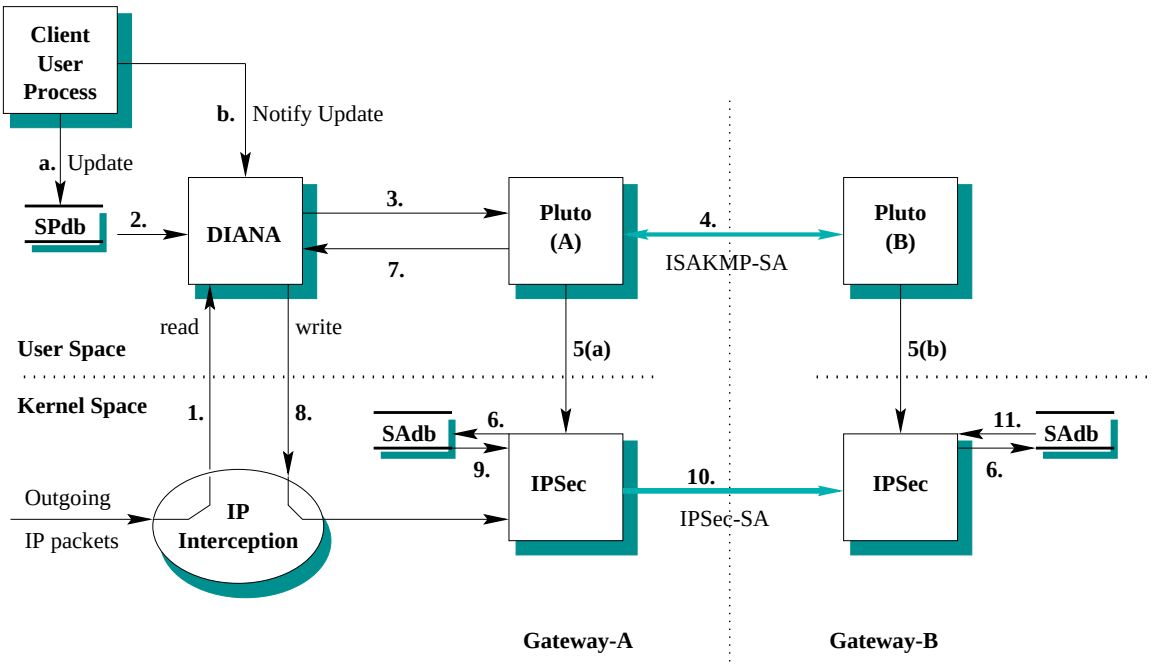


Figure 4.4: Interaction of DIANA with FreeS/WAN and Client User Process

4.4 Interaction of DIANA with FreeS/WAN and IP Interception

With the IP interception mechanism in place, DIANA can now intercept outgoing IP packets. Given an IP packet, DIANA looks into the SPdb to determine the security operations, if any, to be performed on the packet. If a *matching* Security Association with the required operations is already available or if the packet does not match any security policy, then DIANA just sends the packet back to the kernel. Otherwise, it communicates with Pluto for Security Association establishment. This scenario is shown in figure 4.4.

The *Client User Process* will asynchronously modify the SPdb directly. After modifying SPdb, it will notify DIANA of the change. These events are indicated by steps **(a)** and **(b)** in figure 4.4. The modification of SPdb might result in a Security Policy being removed from the SPdb. If a Security Association was established for such a policy, DIANA communicates with Pluto for its closure (teardown). If a new policy is added, DIANA just stores it in its internal data structures.

The meaning of the sequence of events 1..11 is as follows:

1. DIANA reads outbound IP packets by IP-intercept mechanism

2. DIANA looks in the SPdb to check if there is a policy that matches the current IP packet.

```
If ((there is a matching policy) and
    (a Security Association does not already exist for that policy,)
  goto step 3
else,
  goto step 8
```

3. Depending on the security needs of the policy, DIANA asks Pluto to establish an IPSec Security Association.

4. Pluto exchanges session-keys with peer ISAKMP Pluto daemon over an ISAKMP Security Association.

5. Pluto passes on the session-key information to the kernel and asks the kernel to establish the IPSec Security Association with the required security features.

6. IPSec stores the Security Association information in SAdb

7. Pluto replies back to DIANA. The reply can be one of the following :

   (a) There is already some IPSec Security Association in existence

   (b) Pluto created a new IPSec Security Association

   (c) Pluto was unable to create an IPSec Security Association

```
If the reply was (1) or (2), then
  goto step 8
else
  DIANA retries Security Association establishment 2-3 times, and
  prints appropriate error messages.  Even after such retries, if
  an IPSec Security Association cannot be established, then DIANA
  drops the packet.
```

8. DIANA sends the IP-packet back to the kernel via the IP interception write mechanism.

There are certain decisions which have influenced the design of DIANA. The remainder of this chapter discusses these design decisions. As was mentioned previously, DIANA is implemented in the user space to take the place of the process "whack" in figure 4.2. Apart from ease of implementation in user space, this decision allows de-coupling of DIANA from the Linux kernel code. This has an advantage that even if new versions of FreeS/WAN or Linux Kernel appear, DIANA would still work (with possibly very minimal modifications).

4.5 Security Policy Database

Currently, the FreeS/WAN implementation does not provide support for Security Policy Database. Eventually it is expected to support SPdb in the kernel and provide some mechanism (Application Program Interface or API) to update the SPdb from user space. In such a scenario, the *Client User Process* can directly modify the SPdb (which would now be in the kernel), and we can move some (or all) functionality of DIANA in kernel space.

In the current implementation of DIANA, SPdb is in user space. It exists as an ASCII file. The advantage of keeping it as ASCII file is that it helps in debugging. For example, the user can modify the SPdb manually (by using an editor, like 'emacs') and then inform DIANA about this modification. Diana also maintains a copy of SPdb in its internal data structures. When it receives an indication that the SPdb file has been modified, it updates these internal data structures.

The *Client User Process* just needs to know two things. The structure of the SPdb file and how to inform DIANA about the modification to the SPdb file. In the current implementation, each line of this file is either **(i)** a comment, **(ii)** a security policy or **(iii)** a blank line. Comment lines begin with '**#**' as their first character. A security policy has two parts: condition and action. The *condition* part describes rules for matching IP packets with policies. In DIANA, it is specified according to the *selectors* as given in the IPSec specification. The *action* part of the policy indicates the processing required for the packet. A security policy in the SPdb file would thus look like:

```
<policy_id> <src_addr> <dst_addr> <xpt_protocol> <src_port> <dst_port> \
<user_id> <peer_addr> <peer_port> <encry> <auth> <tunnel> <discard>
```

where,

- **(src_addr)**: IP address of the source (can be a subnet address). It is specified as address/mask.
- **(dst_addr)**: IP address of the destination (can be a subnet address). It is specified as address/mask.
- **(xpt_protocol)**: The transport protocol, obtained from IPv4 "Protocol" field (can be a wild-card).
- **(src_port)**: can be individual UDP or TCP port or wild-card
- **(dst_port)**: can be individual UDP or TCP port or wild-card
- **(user_id)**: this is not currently used
- **(peer_addr)**: address of the peer-gateway with which to establish Security Association.
- **(peer_port)**: UDP port at which the Pluto daemon of the peer-gateway will be listening (Default 500).
- **(encry)**: whether the packet should be encrypted [1: encryption, 0: no encryption]
- **(auth)**: whether the packet should be authenticated [1: authentication, 0: no authentication]
- **(tunnel)**: whether it's tunnel mode or transport mode [1: tunnel mode, 0: transport mode]
- **(discard)**: whether to discard the IP packet or not [1: discard, 0: process]

There should be a unique policy entry for each combination of `<(src_addr) (dst_addr) (xpt_protocol) (src_port) (dst_port) (user_id)>`.

The first implementation of DIANA checks the `src_port` and `dst_port` fields only for TCP and UDP protocols. For other transport-layer protocols, these fields are not considered while matching packets to policies. Each of the fields above is separated by blanks or tabs. The default policy is to allow packets to pass through.

4.6 Updating SPdb

Client User Process also needs to inform DIANA about the modification to the SPdb file. The SPdb file can also be manually modified by the network administrator (possibly for debugging purposes). As this modification is an asynchronous process, it would be better, if DIANA also updates its internal data structure for SPdb *only* when the SPdb file is modified. This can be done in two ways. One way is for DIANA to poll the SPdb file periodically and check if there is any change to it. The problem with this approach is that if the SPdb is not changed for a long time, DIANA would be unnecessarily doing work. Another way is for the *Client User Process* to notify DIANA of the change in SPdb. This approach was chosen for DIANA. Two approaches were considered for the *Client User Process* to notify DIANA:

1. **Signals**:

   In this approach, the *Client User Process* sends a `SIGHUP` signal to DIANA. A signal handler associated with DIANA updates its internal SPdb data structure when it gets a `SIGHUP` signal. Though this is a simple solution, it has one problem. What happens when `SIGHUP` is received when DIANA is in the middle of IP-packet processing? Inconsistencies might result as a result of a sudden change in SPdb data structure. To prevent this, a concurrency control mechanism is needed, whereby the signal handler will wait for the current IP packet to be processed before modifying the internal SPdb data structure.

2. **Sockets**:

   In this approach, the *Client User Process* sends a pre-formatted `UPDATE_SPDB` message to DIANA on a specified UDP port. DIANA uses a `select` statement to simultaneously listen to this port (for `UPDATE_SPDB` message) and intercept IP packets. When DIANA gets an `UPDATE_SPDB` message, it reads SPdb from the SPdb-file and updates its internal SPdb data structure. This approach avoids the need for concurrency control mechanisms, as `select` will take care that only one of IP-packet or `UPDATE_SPDB` message is processed by DIANA.

The *signal* mechanism for message passing between the Client User Process and DIANA was chosen, as it is easier for the user to modify the SPdb file and inform DIANA. Also, as the figure 4.4 indicates, DIANA needs to do four things simultaneously:

1. Update its internal SPdb data structure from time to time from the SPdb file
2. Listen to messages from Pluto
3. Intercept IP packets from the kernel and compare them with the security policies in the SPdb
4. Send IP packets back to the kernel

For this, DIANA is a multi-threaded implementation with four threads as described later (chapter 5) in this thesis.

4.7 Optimizations

As mentioned before, the implementation of DIANA in user space greatly simplifies its design. But, interception of all outbound IP packets from the kernel to the user space even if there is no Security Policy in SPdb, or even if Security Associations have been established for all policies, poses a severe overhead. Also, not every packet should trigger a Security Association-establishment message to be sent to Pluto. Only the first packet of a particular network traffic should trigger such a message.

To take care of these things, the internal SPdb data structures representing a Policy in DIANA have a `status` flag to indicate whether an IPSec Security Association has been set up for that policy or not. With this flag in place, DIANA will trigger Pluto for Security Association-establishment only if the status flag is not set. Thus, in figure 4.4, step 2, if there is a match between the IP packet and some policy and if the flag for the policy is set, it means that there already exists some IPSec Security Association for that policy and the control just passes to step 8, i.e. the IP packet is returned to the kernel.

Once flags are set for all the policies in the SPdb, IP packets need not be intercepted from the kernel. This is done by '*closing*' the interception mechanism. If some new policy is added to the SPdb, then the interception mechanism is '*opened*' again. This has been implemented in DIANA.

A better optimization can be sought, if there is a filtering mechanism in IP interception. Such a filter in the kernel would allow user processes to specify rules for matching IP packets. The filter will then intercept only those packets which match some rule. Thus, if packets can be filtered based on *IPSec selectors*, then, only those packets will be intercepted

for which there is some policy available *and* there is no IPSec Security Association available for that policy. If the Security Association-establishment is fast, then such packets will indeed be very few in number. All other packets will not be intercepted. In particular, those IP packets for which an IPSec Security Association has already been established will not be intercepted. This filtering mechanism is not currently supported by the IP interception mechanism and hence has not been implemented in DIANA.

The current implementation of Pluto does not reply back. So, step 7 in figure 4.4 is not supported. To support this, some part of Pluto was modified so that it informs DIANA whether the IPSec Security Association was established or not.

These were the design issues for implementing the traffic-driven Security Association establishment mechanism. The next chapter deals with the implementation of DIANA.

Chapter 5

DIANA Implementation

This chapter discusses the implementation of DIANA, a utility for traffic-driven Security Association establishment. As was mentioned in the previous chapter, DIANA is implemented in the user space. Hence it needs an IP interception mechanism to read IP packets from the kernel and write them back to the kernel from the user space.

This chapter first looks at the data structures used in the implementation. Then it discusses the overall architecture of DIANA. This is followed by a discussion of the interfaces required for communication between DIANA and other entities (such as Client User Process, Pluto, Kernel). Finally, a pseudo-code for various components of DIANA is presented.

5.1 Data Structures

DIANA needs to maintain quite a few data structures. The most prominent among them are the SPdb policies, queue-messages and queues. The use of these data structures is given in the architectural design.

Data Structure for internal form of SPdb Policies:

```
struct Policy {

  unsigned int policy_id,  /* unique ID of the policy */

  struct in_addr src_addr, /* source address in network order */
  struct in_addr src_mask,

  struct in_addr dst_addr, /* destination address in network order */
  struct in_addr dst_mask,
```

```
    boolean xpt_protocol_wildcard;
    /* TRUE == wildcard i.e. any protocol
     * FALSE == no wildcard i.e. specific protocol */
    u_char xpt_protocol, /* transport layer protocol */

    boolean src_port_wildcard;
    /* TRUE == wildcard i.e. any port
     * FALSE == no wildcard i.e. specific port */
    u_int16_t src_port, /* transport layer source port in
                         * network order */

    boolean dst_port_wildcard;
    /* TRUE == wildcard i.e. any port
     * FALSE == no wildcard i.e. specific port */
    u_int16_t dst_port, /* transport layer destination port in
                         * network order */

    u_int16_t user_id, /* not used currently */

    struct in_addr peer_addr, /* address of the peer security
                               * gateway in network order */

    u_int16_t peer_port, /* network order */

    boolean encry,   /* TRUE -> encrypt, FALSE -> don't encrypt */

    boolean auth,    /* TRUE->authenticate, FALSE->don't authenticate */

    boolean tunnel,  /* TRUE -> tunnel mode, FALSE -> transport mode */

    boolean discard, /* TRUE->discard IP packet, FALSE->don't discard */

    boolean status   /* tells whether the policy is ACTIVE, PENDING
                      * or INACTIVE */

    boolean mark     /* used while updating SPdb tables */

    unsigned int try_count; /* used while counting number of tries
                             * in establishing a Security Association*/
}
```

SPdb Table: This data structure is a container of policies which can be linked to form a list.

```
struct SPdb_table {

  Policy *p;              /* the policy */
  Queue  *q;              /* the packet-queue associated
                             * with the policy */
  struct SPdb_table *next;
}
```

Data Structure for IP-packet header:

```
struct ip, as defined in /usr/include/netinet/ip.h
```

Whack Message: Message format for message from DIANA to Pluto (taken from `whack.h` file in FreeS/WAN implementation):

```
#define WHACK_MAGIC (('w' << 24) + ('h' << 16) + ('k' << 8) + 1)

struct whack_message {

  unsigned int magic; /* used for version compatibility with pluto */

  bool whack_options;      /* for WHACK_OPTIONS: */

  unsigned int debugging;

  bool whack_initiate;     /* for WHACK_INITIATE: */

  bool
    explicit_local_client, /* whether the local and peer clients are*/
    explicit_peer_client;  /* explicitly specified or not */

  struct in_addr           /* network order */
    peer,
    local_client_net, local_client_mask,
    peer_client_net, peer_client_mask;

  u_int16_t peer_port;     /* network order */

  unsigned int goal;   /* encrypt, authenticate, tunnel, transport */

  bool whack_shutdown;     /* for WHACK_SHUTDOWN */
};
```

Diana Message: This is the format for messages from Pluto to DIANA.

```
struct diana_message {
  unsigned int status; /* 0 = FAILURE = could not establish IPSec SA,
                        * 1 = SUCCESS = IPSec SA already in existence
                        *               or new IPSec SA created */
  Policy policy;
}
```

Message format for queue items:

```
struct Qmesg {
  unsigned int type;  /* Update_SPdb, IP_Pkt, Pluto_Resp */
  void * dat;         /* Null, IP packet*, diana_message* */
  struct Qmesg *next;
}
```

Data structure for queue:

```
struct Queue {

  int length;             /* length of the queue */

  Qmesg *first;           /* first element of the queue */
  Qmesg *last;            /* last element of the queue */

  pthread_mutex_t mutex;  /* a mutex variable to ensure that only
                           * a single thread accesses the queue at
                           * a given time */

  pthread_cond_t not_empty, empty; /* conditional variables to avoid
                                    * busy waits for readers of the
                                    * queue */
}
```

These are the data structures in the DIANA implementation. The next section discusses the architecture of DIANA.

5.2 Architecture

DIANA is a multi-threaded implementation with 4 threads: **(1)** DIANA Controller, **(2)** Pluto Reader, **(3)** Net Input and **(4)** Net Output. The Signal Handler is a part of DIANA Controller. To avoid concurrency issues, a queuing mechanism for message exchange is implemented. There are two queues – InQ and OutQ. DIANA Controller reads

messages from InQ and writes messages to OutQ. The following diagram illustrates the architecture of DIANA.

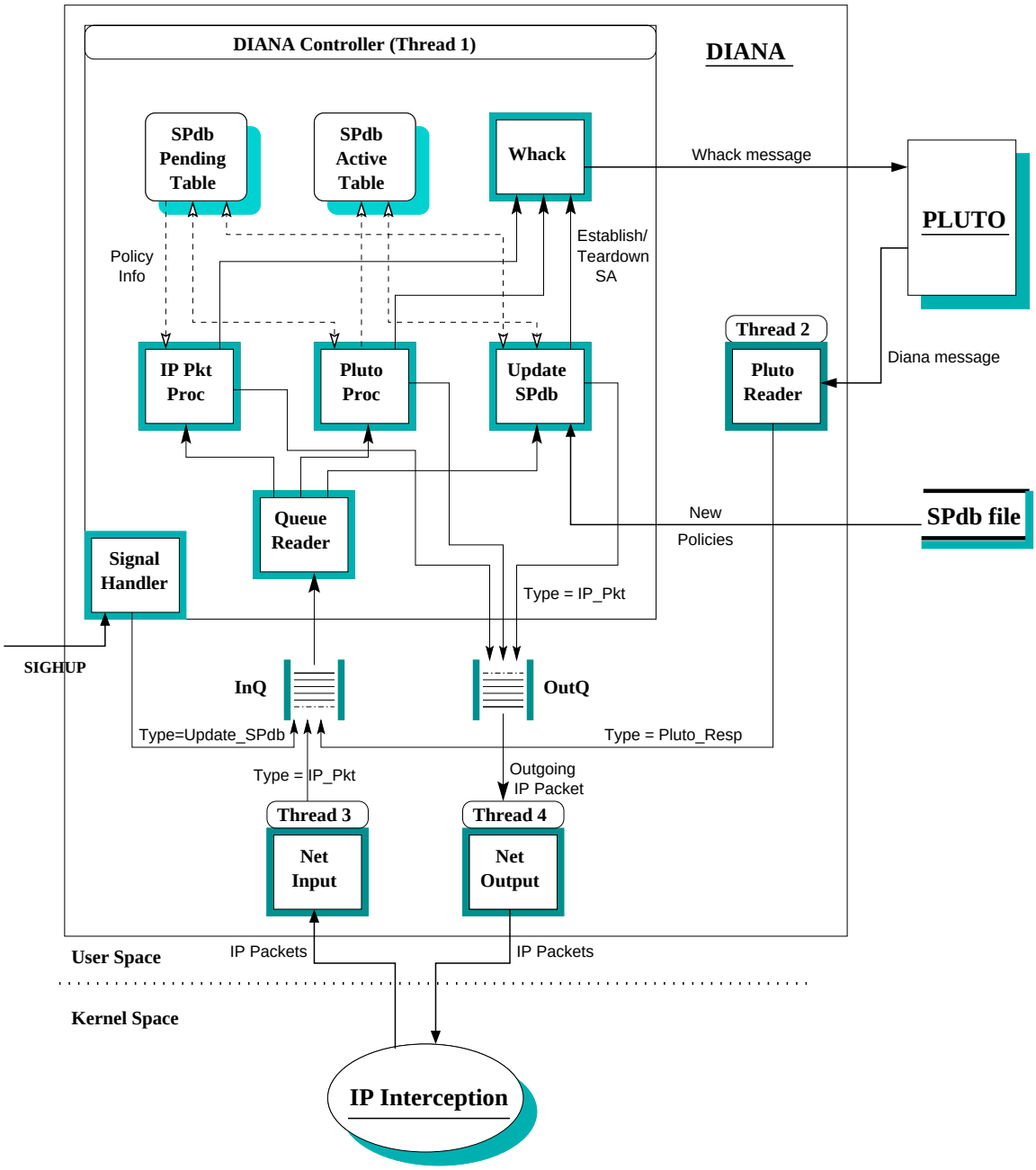


Figure 5.1: Architecture of DIANA

Note: The arrows indicate passing of messages and control between modules.

The different modules of DIANA in figure 5.1 are as follows:

- **InQ**: This is a queue for input to the DIANA controller. Net Input, Signal Handler and Pluto Reader send messages to this queue. These messages are read by the Queue Reader part of DIANA Controller.
- **Net Input**: This thread interfaces with the kernel via the IP interception mechanism to read IP packets from the kernel. For each packet, this module composes a Qmesg of type IP_Pkt and sends it to InQ.
- **Net Output**: This thread reads Qmesg messages of type IP_Pkt from the OutQ, extracts IP packets from them and sends the IP packets to the kernel by the write mechanism of IP interception.
- **Signal Handler**: This is a part of the DIANA controller thread. It catches the `SIGHUP` signal and sends an *Update_SPdb* message to InQ.
- **Pluto Reader**: This thread reads status messages from Pluto. After receiving a *diana_message* from Pluto, it composes a *Qmesg* and puts it into InQ.
- **Queue Reader**: This module reads *Qmesg* messages from InQ. Depending on the type of the message, it invokes one of the functions: IP_pkt_proc, Pluto_proc or Update_spdb.
- **IP Pkt Buffer**: This buffer (not shown in the figure), is a *logical* list of queues, each of which contains IP packets. Each queue corresponds to packets which satisfy a given policy in the SPdb, and for which a Security Association is being established. When the SA is established, the corresponding queue is emptied and the packets are sent back to the kernel. IP packets are added to this buffer by the IP_pkt_proc function and queues are removed by Pluto_proc or Update_SPdb. In the actual implementation, each queue of this buffer is grouped along with the corresponding policy in the SPdb tables.
- **SPdb Pending Table**: This is a table of policies having status as either `PENDING` or `INACTIVE`. A policy status of `PENDING` implies that a security association is being established for that policy. A policy status of `INACTIVE` indicates that no packet was received corresponding to that policy and hence no Security Association exists for the policy.

- **SPdb Active**: This is a table of policies which have status as `ACTIVE`. A policy status of `ACTIVE` implies that a security association has been established corresponding to the policy. The SPdb has been separated into two tables Pending and Active so that IP packets are matched only with policies in the SPdb Pending table, thus achieving some performance benefit.

- **OutQ**: This is a queue of processed IP packets encapsulated in *Qmesg* data structure, which are to be written back to the kernel.

- **IP Pkt Proc**: This module is responsible for processing IP packets within DIANA.

- **Pluto Proc**: This module is responsible for processing diana_messages from Pluto.

- **Update SPdb**: This module is responsible for updating the internal SPdb data structure (SPdb Pending and SPdb Active tables) from the SPdb file.

5.3 Interfaces

DIANA interfaces with the Client User Process, Pluto and the Kernel. The interfaces for each of these are as follows:

- **Client User Process**:

  When the Client User Process modifies the SPdb, it sends a `SIGHUP` signal to DIANA. A signal handler in DIANA catches this signal. The signal handler adds a *Qmesg* of type Update_SPdb to InQ as described in the Architectural Design above. DIANA then reads the SPdb file. The structure of the SPdb file is given in the previous chapter.

- **Pluto**:

  DIANA needs to tell Pluto to establish/teardown Security Associations. This communication is done via UDP socket `PLUTO_PORT` + 1 (`PLUTO_PORT` is defined in the Pluto implementation and its default value is 500) by sending a "whack_message" in the pre-specified format. Pluto replies DIANA about the status on the UDP port `PLUTO_PORT` + 2 by a "diana_message". This tells DIANA whether the IPSec SA was created or not. The data structures for these messages are given in the previous chapter.

- **Kernel**:

  DIANA interfaces with the Linux kernel by the IP interception `read` and `write` mechanisms for reading and writing back IP packets.

The four threads of DIANA communicate with each other by means of two queues: InQ and OutQ. The use of these queues also achieves concurrency control amongst the threads. The modules within DIANA controller communicate with each other via various messages that are described later in the pseudo-code.

5.4 Procedures

This section describes the sequence of events that occur in DIANA when either it gets a signal from Client User Process, or it receives an IP packet from the IP interception module.

SPdb related:

1. When DIANA starts up, it spawns four threads: DIANA controller, Pluto Reader, Net Input and Net Output. The DIANA controller reads SPdb into its internal data structure: SPdb Pending table.

2. When the Client User Process modifies the SPdb file, it sends a SIGHUP signal to DIANA.

3. The Signal Handler in DIANA catches the signal and sends a Qmesg of type Update_SPdb to InQ.

4. When the Update_SPdb message reaches the front of InQ, the Queue Reader part of DIANA reads it and calls the Update_spdb function.

5. The Update_spdb function updates the internal SPdb data structures – SPdb Active and SPdb Pending tables – by re-reading the SPdb file.

6. If some `ACTIVE` or `PENDING` policy has been deleted, the Update_spdb function sends a corresponding "Teardown" message to whack function. If there is a corresponding non-empty queue associated with the policy, the queue is emptied and its contents (IP packets) are sent back to the kernel by writing them to OutQ.

IP packet processing related:

1. Net Input reads an IP packet from the kernel, composes a Qmesg of type IP_Pkt and sends it to InQ.

2. When this Qmesg reaches the front of InQ, the Queue Reader module reads it and calls IP_Pkt_Proc function.

3. The IP_Pkt_Proc function extracts the header information from the IP packet and matches it with policies from the SPdb Pending table.
   If a match is not found, the IP packet is written back to OutQ. Eventually, Net Output reads this packet from the OutQ and sends it to the kernel.
   If a match is found **and** the status of the corresponding policy is PENDING, the IP Packet is added to the packet queue corresponding to the policy.
   If a match is found **and** the status of the corresponding policy is INACTIVE, an "Establish_SA" (whack) message is sent to Pluto, the packet is added to the queue corresponding to the policy and the status of the policy is changed to `PENDING`.

4. Pluto replies status to DIANA by diana_message. The Pluto Reader thread reads this message, composes a Qmesg of type Pluto_Resp and sends it to InQ. The Queue Reader eventually reads this message and calls the Pluto_Proc function.

5. The Pluto_Proc function checks the reply from Pluto. If the reply was `SUCCESS`, i.e. an SA was established, then it writes back the IP packets in the corresponding queue to the kernel. It also moves the corresponding policy from SPdb Pending table to the SPdb Active Table and sets its status to `ACTIVE`.
   If the reply from Pluto was `FAILURE`, i.e. could not establish an SA, then Pluto_Proc sends the "Establish_SA" message to Pluto a few more times. If Pluto is still not able to establish the required Security Associations, the IP packets of the corresponding queue are dropped, the queue is removed from the IP Pkt Buffer and the status of the policy is set to `INACTIVE`. An appropriate error message is output for troubleshooting purposes.

This is how the usual processing takes place in DIANA. DIANA is supposed to run as a daemon process, hence special care has been taken to avoid memory leaks in the implementation. The next section gives the pseudo code for the various functions in DIANA.

5.5 Pseudo Code

```
DIANA main()
{
   Initialize InQ and OutQ
   Process command line options and parameters, if any
   Spawn three threads Pluto Reader, Net Input and Net Output
   Continue as DIANA controller
}

DIANA controller()
{
   Setup the signal handler for SIGHUP signal
   Initialize the SPdb Active and SPdb Pending tables
   Read SPdb from file into SPdb Pending table
   Call the function Queue Reader
}

Signal Handler()
{
   Catch SIGHUP signal
   Send Qmesg of type 'Update_SPdb' to InQ
}

Net Input()
{
   Loop forever

      Read IP packet from Kernel by IP interception read mechanism
      Compose a Qmesg of type IP_Pkt and send it along with the IP
         packet to InQ

   End Loop
}

Net Output()
{
   Loop forever

      Read Qmesg from OutQ
      Extract IP packet from this Qmesg
      Send the IP packet to kernel

   End Loop
}
```

```
Pluto Reader()
{
   Loop forever

      Read diana_messages from Pluto from specified UDP port
      Compose and send Qmesg of type 'Pluto_Resp' along with
       the diana_message to InQ

   End Loop
}

Queue Reader()
{
   Loop forever:

      Read a Qmesg from InQ
      switch (Qmesg.type)

         case Update_SPdb:
             call the Update_spdb function

         case Pluto_Resp:
             call the Pluto_proc function with Qmesg.dat as parameter

         case IP_Pkt
             call the IP_pkt_proc function with Qmesg.dat as parameter

      end switch
    End loop
}

Update_spdb()
{
  Read SPdb-file into internal buffer SPdb_buf
  Compare all entries of SPdb_buf with all entries of the SPdb
   Active table

  If some entry matches, remove it from SPdb_buf and mark it in
   SPdb Active table

  If there are unmarked entries in SPdb Active table,
   for each such entry do:
```

```
      Send a 'Teardown' message to whack function
      Delete that entry from SPdb Active table
      /* assume that Pluto always succeeds in Teardown */
    end for

  Unmark all entries in SPdb Active table

  Compare all entries of SPdb_buf with all entries of the SPdb
   Pending Table

  If some entry matches, remove it from SPdb_buf and mark it in
   SPdb Pending table

  If there are unmarked entries in SPdb Pending table,
   for each such entry do:

      If the status of the entry is INACTIVE, delete the entry

      If the status of the entry is PENDING then

         Send a 'Teardown' message to whack function
         Remove corresponding queue from IP Pkt Buffer and send
          the packets in the queue to OutQ
         Delete the entry from SPdb Pending table

      end If
    end for

   Unmark all entries in SPdb Pending table

   If there are some entries left in SPdb_buf, add corresponding
    entries in SPdb Pending table with status = INACTIVE

   Free space from SPdb_buf

   If there are no entries in SPdb Pending table and interception
    mechanism is open then
      close interception mechanism
   else
      if the interception mechanism is closed
         open interception mechanism
   end If
}
```

```
Pluto_proc()
{
   switch (diana_message.status)

   case SUCCESS:
      Move the corresponding policy-entry from SPdb Pending table to
       SPdb Active table and set its flag to ACTIVE
      Empty the corresponding packet queue from IP Pkt Buffer and
       send the packets to OutQ
      If there are no entries in SPdb Pending table, close the IP
       interception mechanism

   case FAILURE:
      Check if number of tries == MAXTRIES /* 2 or 3 */
      If yes, then

              Empty the corresponding queue from IP Pkt Buffer and
               discard the packets
              Set the flag of the corresponding policy in SPdb Pending
               table to INACTIVE
              Give an Error message

      else /* number of tries != MAXTRIES */
              number of tries = number of tries + 1
              send an 'Establish_SA' message to whack function
      endif
   end switch
}

IP_pkt_proc()
{
   Extract IP packet header
   Match IP packet with policies in SPdb Pending table by calling
    match function

   If there is no match, send the packet to OutQ
   If there is a match then

      If the discard flag for the policy is set then
             discard the packet

      else if the flag on the policy is PENDING
             add the packet to corresponding queue in IP Packet Buffer
```

```
      else if the flag on the policy is INACTIVE
             create a new queue in IP Pkt Buffer corresponding to
               the policy
             add packet to this queue
             change flag on policy to PENDING
             Send an 'Establish_SA' message to whack function
      endif
   endif
}

match()
{
 match packet with policies in SPdb Pending table
 If a match is found, return TRUE along with the matched policy
 If a match is not found, return FALSE
}

whack(arg)
{
  Compose a whack_message.  Set the encryption/authentication
  parameters depending on the corresponding matched policy and
  whether arg is 'Teardown' or 'Establish_SA'.

  Send the whack_message to Pluto on pre-specified port
}
```

This chapter discussed the implementation of DIANA. The actual `C` source-code for DIANA is given in Appendix A. To summarize, DIANA is a multi-threaded application, which handles concurrency between threads by means of message queues. It communicates with three entities in the system: Client User Process, IP interception module and Pluto (ISAKMP process). By intercepting IP packets, matching them with SPdb policies, queuing them and by communicating with Pluto, DIANA performs traffic-driven Security Association establishment.

The next chapter presents performance measurement results when DIANA is used in a network environment.

Chapter 6

Performance Measurement

As remarked at the end of chapter 3, the traffic-driven approach to SA-establishment will be useful only if the overhead of dynamically setting up an SA is "reasonable". This chapter discusses the performance measurements for DIANA. First it describes the experimental setup in which the measurements were taken. Then it looks at the overhead of intercepting IP packets from kernel to the user space. Next, it presents the time required by Pluto to establish ISAKMP SAs and IPSec SAs. Finally, it presents a series of timing measurements for DIANA.

6.1 Experimental Setup

The measurements were taken with the configuration shown in figure 6.1. `SQUEEZE` and `JINAO` act as routers, between whom an SA is setup. The traffic is ICMP traffic generated by `STONE`, at various rates, by using `ping`. The environment is a controlled one in the sense that there is no other traffic, except OSPF "hello" messages, on the links.

`STONE` is a PC running Linux (Kernel version 2.0.35) having a 266 MHz Pentium II processor and 32 MB RAM. It is used only to generate ICMP traffic.

`SQUEEZE` is a PC running Linux (Kernel version 2.0.35) having a 450 MHz Pentium III processor, 128 MB RAM and 128 MB virtual memory. IP interception mechanism has been installed in its kernel. This kernel also supports FreeS/WAN (version 0.90) implementation of IPSec. `SQUEEZE` also runs Pluto as the ISAKMP/IKE implementation and DIANA for traffic-driven SA establishment. This machine is used as a *security gateway*.

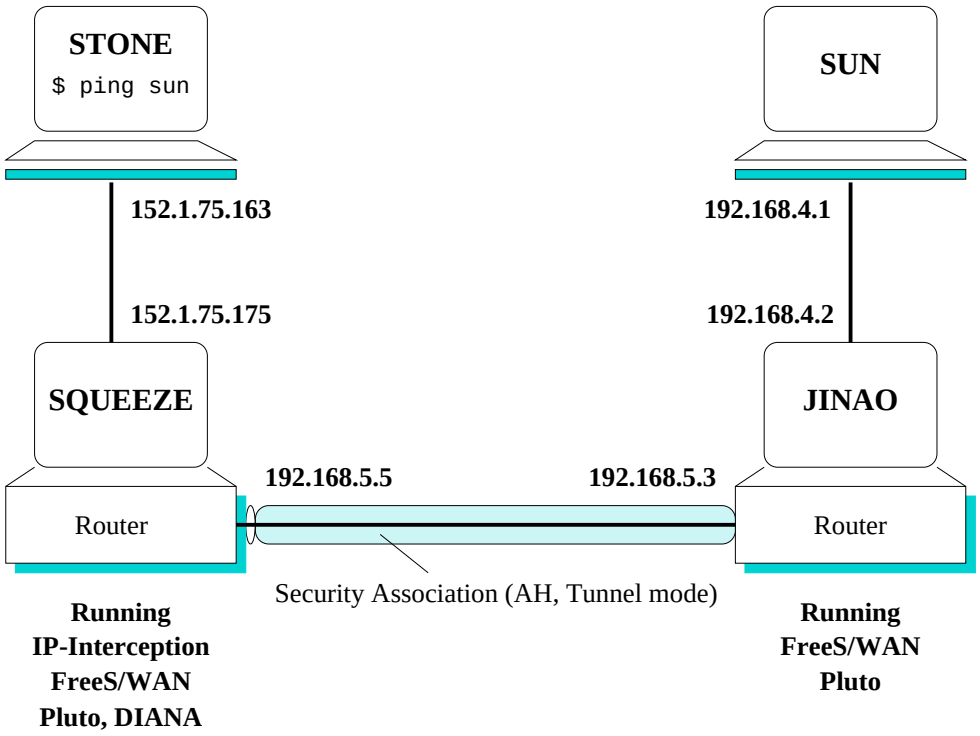


Figure 6.1: Experimental Setup

`JINAO` is also a PC running Linux (Kernel version 2.0.34) having a Pentium processor. Its kernel supports FreeS/WAN implementation of IPSec. `JINAO` also runs Pluto as the ISAKMP/IKE implementation. This machine is also used as a *security gateway*.

`SUN` is a PC running FreeBSD 2.2.5 having a i386 processor. It replies back to the ICMP echo requests from `STONE`.

Initially, Plutos (ISAKMP daemons) are started on `SQUEEZE` and `JINAO`. Then, DIANA is started on `SQUEEZE`. An appropriate policy is specified in the SPdb at `SQUEEZE` to establish an SA as soon as traffic starts to flow between `STONE` and `SUN`. Then, traffic is generated at `STONE` and transmitted. The first packet of this traffic sets up an SA between `SQUEEZE` and `JINAO`. After the SA has been set up, all traffic from `STONE` to `SUN` passes through this SA.

The various events that take place at `SQUEEZE` are (for more information on modules within DIANA, please refer chapter 5):

- **A**. The Net Input module of DIANA receives a packet from the IP interception mechanism.
- **B**. The Net Input module of DIANA puts the packet in InQ.
- **C**. The Packet Reader module of DIANA reads the packet from InQ and sends it to IP Pkt Proc module.
- **D**. The IP Pkt Proc module either sends the packet to OutQ, or sends an establish-SA message to Pluto and stores packet in the queues associated with the matching policy entry in SPdb Pending Table.
- **E**. Pluto establishes an SA and replies back to DIANA.
- **F**. The Pluto Proc module of DIANA, after receiving a SUCCESS diana_message from Pluto, empties the corresponding policy packet-queue into OutQ.
- **G**. Net Output module of DIANA reads a packet from OutQ and sends it to the kernel.

Readings were taken at `SQUEEZE` to measure the following:

1. Overhead of IP interception

2. Time required for Pluto to complete Phase 1 ISAKMP-SA negotiation with peer Pluto.

3. Time required for Pluto to complete Phase 2 IPSec-SA negotiation with peer Pluto.

4. Overhead of diverting all IP packets via. DIANA, without SA-establishment.

5. Delay experienced by a packet due to processing within DIANA and waiting for SA establishment.

These results are summarized in table 6.1. The remaining sections of this chapter explain each result in detail.

Table 6.1: Summary of performance measurements for DIANA

Delay Type	**Mean**	**Standard Deviation**
Overhead of IP Interception	29 μsec	3 μsec
Phase-1 ISAKMP negotiation time	224150 μsec	51937 μsec
Phase-2 ISAKMP negotiation time	30540 μsec	5610 μsec
DIANA processing delay (1 policy)	84 μsec	5 μsec

6.2 Overhead of Linux IP-interception

In this experiment, IP packets were intercepted from the kernel and immediately written back. The time taken by each IP packet to go from the kernel to the user-space and back to the kernel was measured from the kernel. The results are shown in figure 6.2. As table 6.1 shows, the mean time for an IP packet was 29 μsec and the standard deviation was 3 μsec.

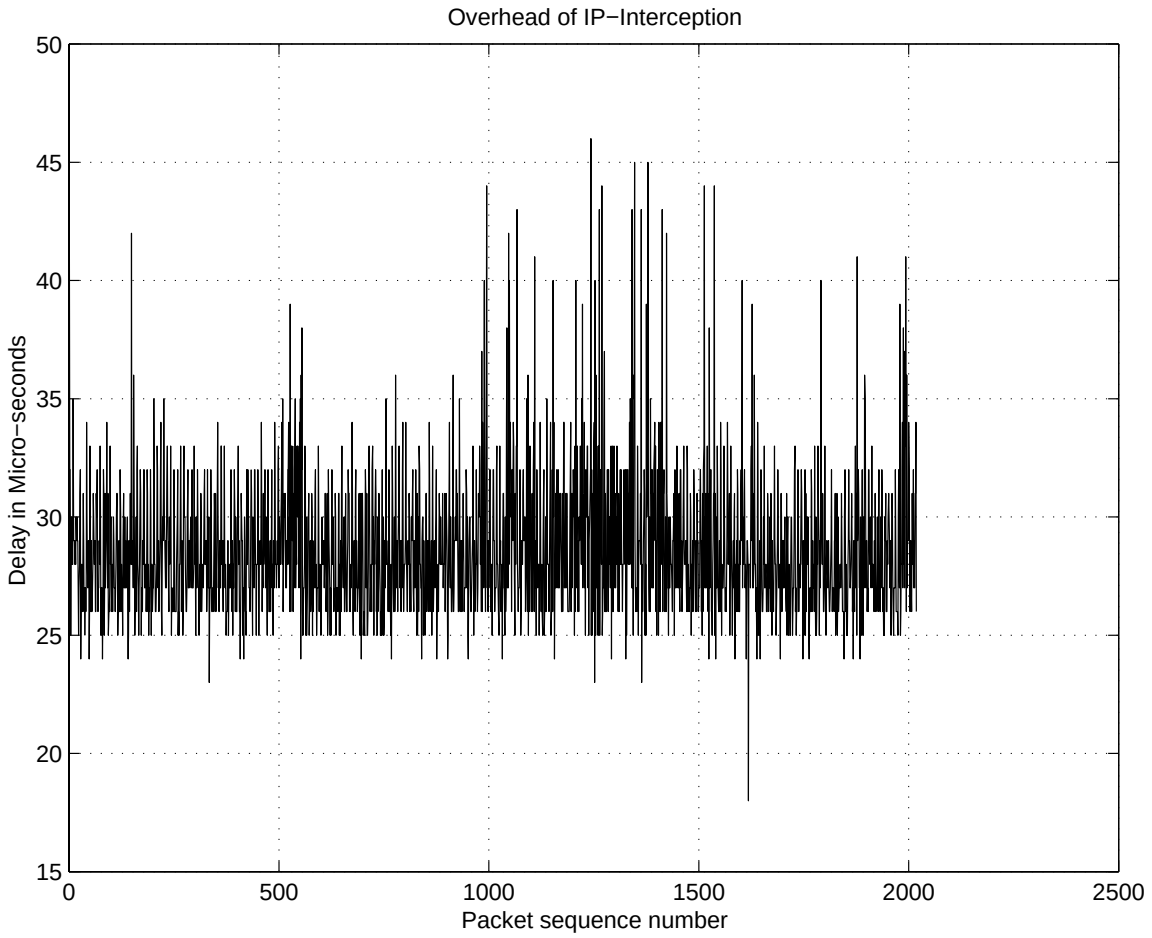


Figure 6.2: Overhead of IP Interception

6.3 Time required to establish an ISAKMP SA

As described in chapter 2, in ISAKMP, the SA negotiation takes place in two phases. In the first phase, an ISAKMP-SA is established, which provides security to further ISAKMP message exchanges. This phase involves the use of digital signatures by public key cryptography, and hence takes a longer time than the phase 2 IPSec-SA establishment.

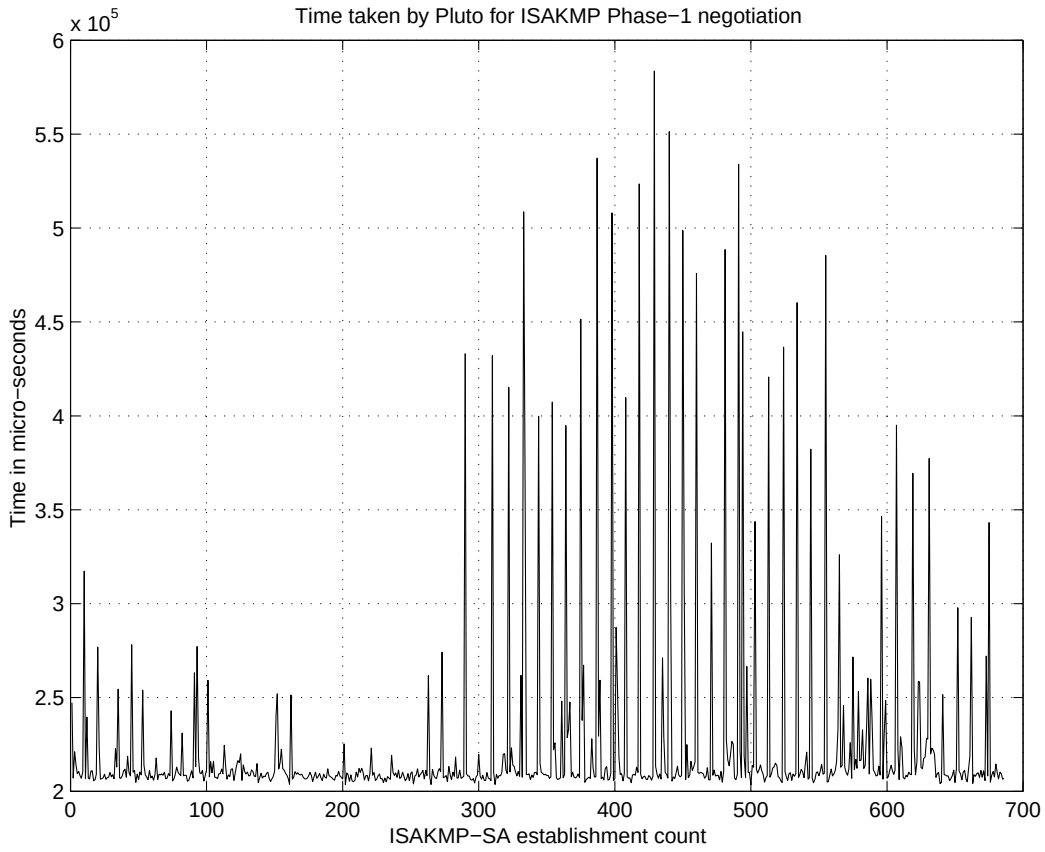


Figure 6.3: Time for phase 1 ISAKMP negotiations

These readings were taken by measuring the time it takes from DIANA to trigger Pluto for the first time and getting back a SUCCESS reply. This includes the time for phase 1 *and* phase 2. By subtracting the phase 2 times from the total, we get the time taken by Pluto for phase 1 SA-establishment. These results are shown in figure 6.3. As shown in table 6.1, the mean time for phase 1 is 224150 μsec with a standard deviation of 51937 μsec.

6.4 Time required to establish an IPSec SA

An IPSec SA is established during phase 2 of ISAKMP negotiations. It just involves exchanging security parameters for IPSec-SA. This does not involve public key based cryptographic algorithms, and hence takes considerably less time. These readings were taken from within Pluto and are shown in figure 6.4. As shown in table 6.1, the mean time for phase 2 is 30540 μsec with a standard deviation of 5610 μsec. The mean value is about seven times less than the corresponding value for phase 1 ISAKMP negotiations.

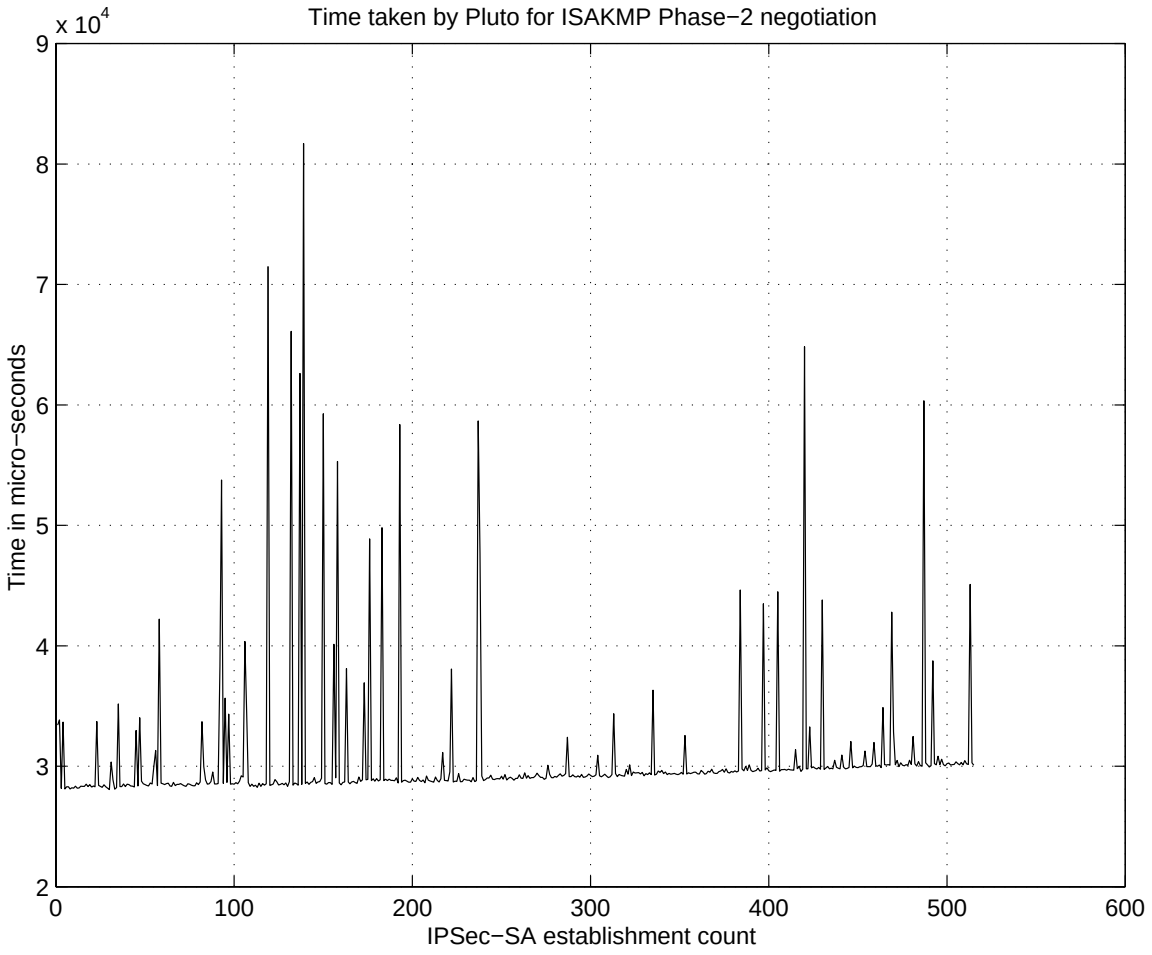


Figure 6.4: Time for phase 2 ISAKMP negotiations

6.5 Overhead of passing an IP packet through DIANA

This experiment was designed to measure the processing delay within DIANA. The packet rate was kept to 1 packet per second, so the queuing delay was minimal. The policies in the SPdb did not match the packets. In the first part of the experiment, the SPdb contained only one policy entry. The results of this first part are shown in figure 6.5. The mean time for a packet to pass through DIANA with one policy entry in SPdb is 84 μsec with a standard deviation of 5 μsec.

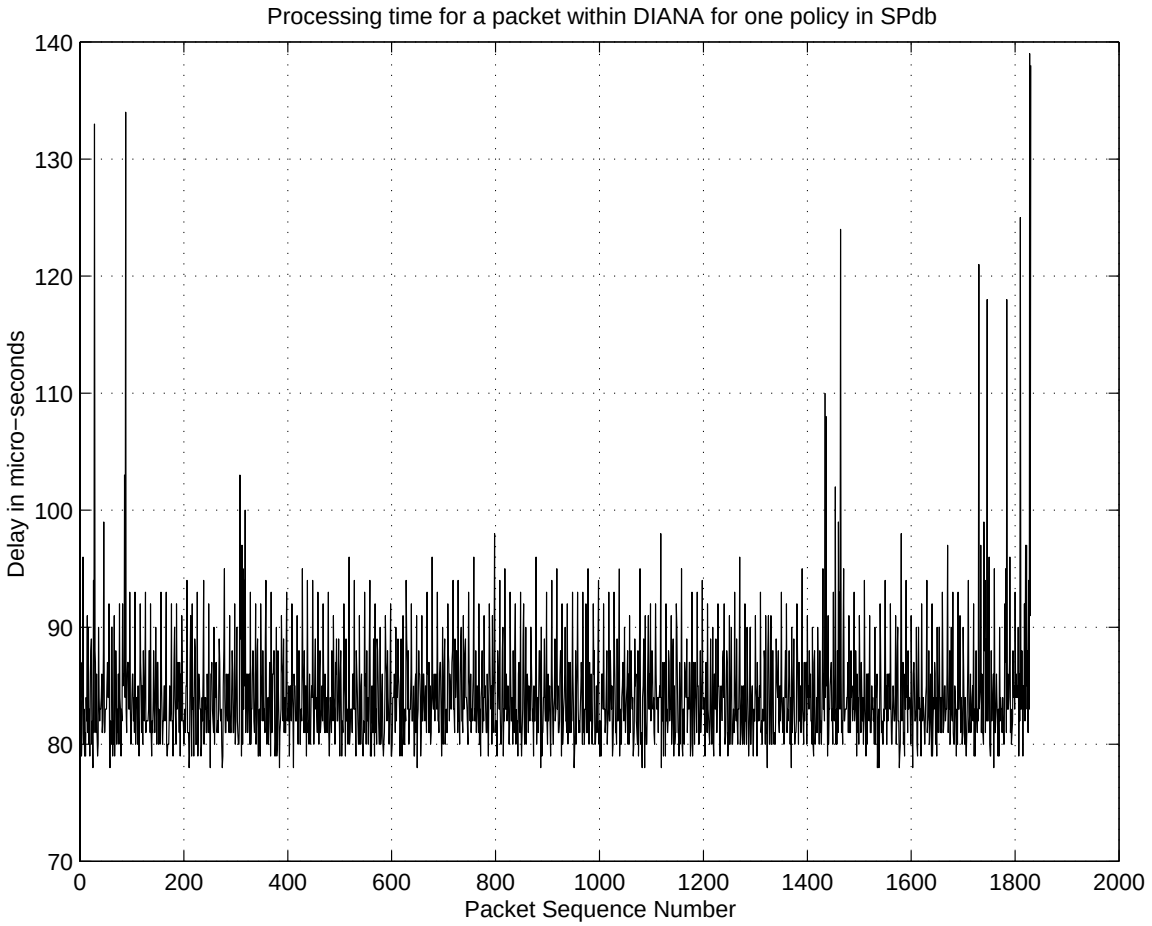


Figure 6.5: Processing delay in DIANA with one policy in SPdb

In the second part of this experiment, mean values of processing delays within DIANA were measured by varying the number of policies in SPdb. The results are shown

in table 6.2. From the table it is clear that the delay increases with the number of policies in SPdb. Currently, DIANA implements a linked list to store the policy entries. The performance with multiple policies may improve if a hash-table is used instead of a linked list to store these entries.

Table 6.2: Mean and Standard Deviation of processing delay in DIANA for different number of policies in SPdb

Number of Policies	**Mean processing delay**	**Std. dev.**
1	84 μsec	5 μsec
10	94 μsec	10 μsec
100	109 μsec	23 μsec
1000	248 μsec	53 μsec
10000	2775 μsec	59 μsec

6.6 IP Packet delay within DIANA

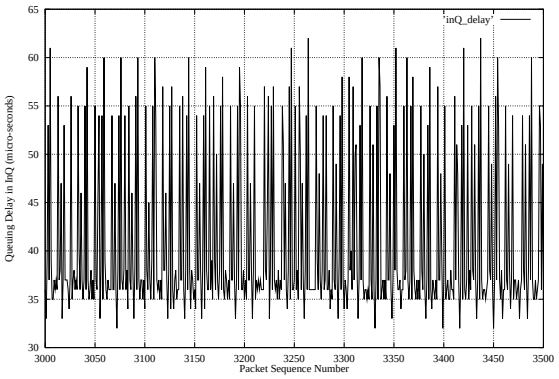
This experiment was designed to measure various queuing delays when a packet passes through DIANA. Recall that there are 3 queues in DIANA through which a packet can pass: the InQ, the OutQ and a queue associated with the matching policy entry (called the SPdbQ). This experiment was conducted in two parts. In the first part, the rate of IP packets was kept at 1 packet per second. In the second part, the rate was increased to 10 packets per second. The SAs were deleted periodically so that queuing delays in SPdbQ could be measured.

Part 1: Packet rate = 1 packet per second

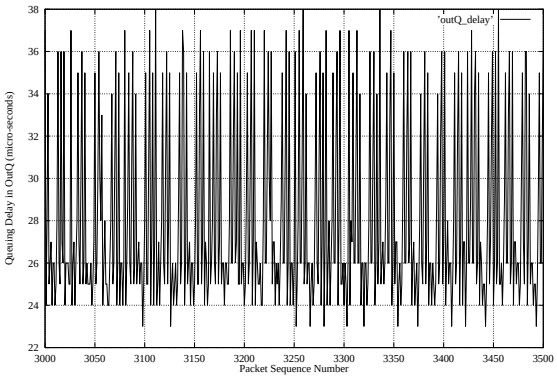
This measurement was taken with two policies in the SPdb – one matching the traffic and the other not matching the traffic. The packets were generated at a rate of 1 packet per second. As this rate is sufficiently low, there was little or no accumulation of packets in the queues within DIANA. These results are shown in figure 6.6. The graphs show only a subset of the packets for which readings were taken. The mean and standard deviation of these delays are given in table 6.3.

Table 6.3: Summary of various delays for packets in DIANA when packet rate = 1 packet/second

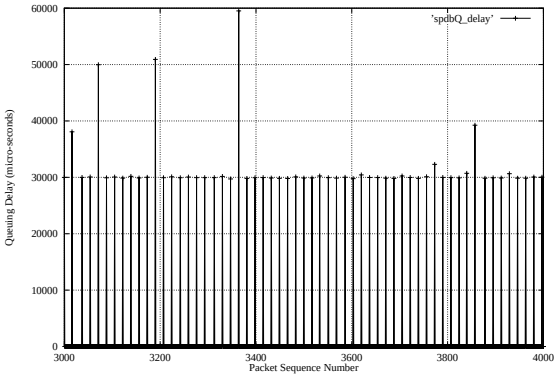
Delay Type	**Mean**	**Standard Deviation**
Delay in InQ	$41\mu sec$	10 μsec
Delay in OutQ	$29\mu sec$	5 μsec
Delay for "triggering" packets in queue with 'matching' policy	32829 μsec	5824 μsec
Total Delay for "triggering" packets	32922 μsec	5826 μsec
Delay for "non-triggering" packets in queue with 'matching' policy	0 μsec	0 μsec
Total Delay for "non-triggering" packets	79 μsec	14 μsec



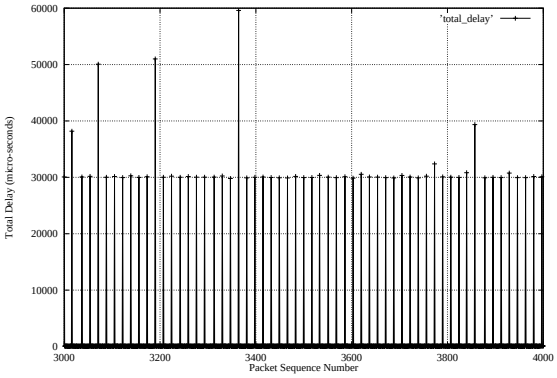
(a) Packet Delays in InQ



(b) Packet Delays in OutQ



(c) Packet Delays in queue in SPdb

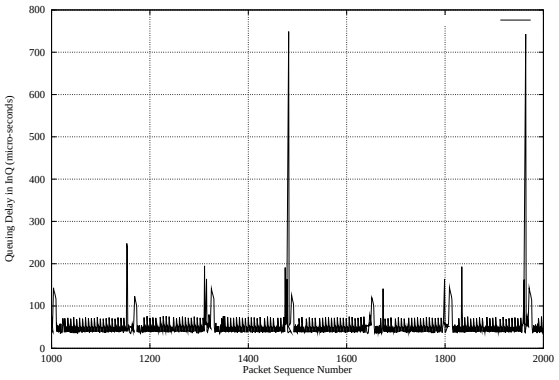


(d) Total Delay for a packet in DIANA

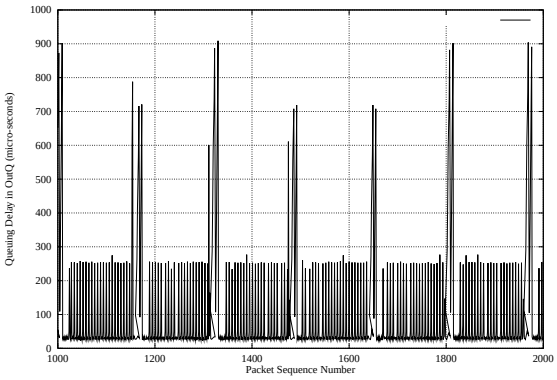
Figure 6.6: Various Delays in DIANA when packet rate = 1 packet/sec

Table 6.4: Summary of various delays for packets in DIANA when packet rate = 10 packets/second

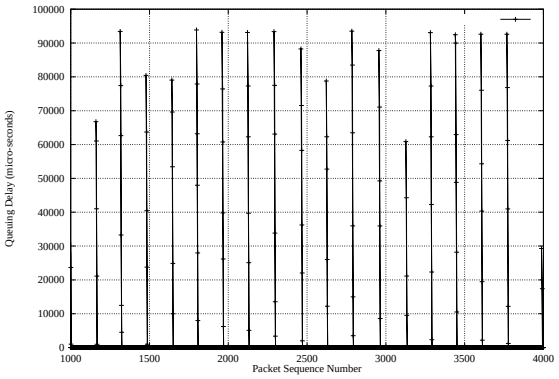
Delay Type	**Mean**	**Standard Deviation**
Delay in InQ	42 μsec	26 μsec
Delay in OutQ	43 μsec	83 μsec
Delay for packets in SPdbQ which waited this queue	46063 μsec	30596 μsec
Total Delay for packets which waited in the SPdbQ	39089 μsec	32422 μsec
Total Delay for packets which did not wait in the SPdbQ	87 μsec	66 μsec



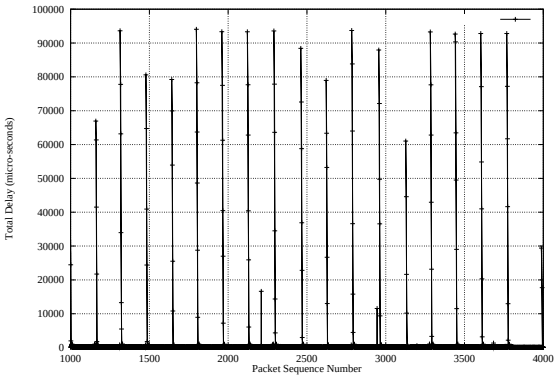
(a) Packet Delays in InQ



(b) Packet Delays in OutQ



(c) Packet Delays in queue in SPdb



(d) Total Delay for a packet in DIANA

Figure 6.7: Various Delays in DIANA when packet rate = 10 packets/sec

Part 2: Packet rate = 10 per second

This measurement was also taken with two policies in the SPdb – one matching the traffic and the other not matching the traffic. The packets were generated at a rate of 10 packets per second. As this rate is relatively higher, there was some accumulation of packets in the queue associated with the matching policy (called the SPdbQ) while the SA was being established. These results are shown in figure 6.7. The graphs show only a subset of the packets for which readings were taken. The mean and standard deviation of these delays are given in table 6.4.

Note that the spikes in these graphs indicate the measurements for packets which cause, or wait for, SA-establishment. This experiment shows that the major component of the delay experienced by a packet in DIANA is the time it is buffered while waiting for the SA to be established. The processing overhead of DIANA is minimal in this scenario. If the rate of packets is high, the initial packets will be queued in the SPdbQ while the SA is being established.

6.7 Throughput Measurements

Table 6.5: Throughput Measurements for IP Interception and DIANA

Experimental Conditions at SQUEEZE	**Mean Throughput** (packets/second)	**Standard Deviation** (packets/second)
No IP-interception No DIANA	5480	39
Only IP-interception	5490	17
DIANA with 1 non-matching policy in SPdb	3477	180
DIANA with 100 non-matching policies in SPdb	3494	172
DIANA with 10000 non-matching policies in SPdb	353	1

This experiment was designed to measure the throughput of IP traffic when it passes through the IP interception mechanism and DIANA. UDP (User Datagram Protocol) messages were sent from `JINAO` to `STONE` (see figure 6.1) at a very high rate. The rate of packets arriving at Stone was measured to give the throughput. For each reading, 100000

packets, each of size 100 bytes (including the IP and UDP headers), were sent from `JINAO` to `STONE`. Each mean value in table 6.5 is a mean of 10 readings. The IP interception mechanism and DIANA were run on `SQUEEZE`.

From the table, it is clear that the IP-interception mechanism does not introduce any significant overhead in terms of throughput. (Note that the readings indicate an *increase* in throughput when IP interception is introduced; but this is on account of the randomness in packet-generation rate at `JINAO`.) The results also show that DIANA causes a reduction in throughput. The throughput of IP traffic, with DIANA, does not differ much if the number of policies are within a few hundred. (Note again that the readings indicate an *increase* in throughput when the number of policies in SPdb in DIANA are increased from 1 to 100. But, this is on account of the randomness in packet-generation rate at `JINAO`.) If the number of policies is large, for example 10000, the reduction in throughput is substantial. This decrease in throughput corresponding to an increase in the number of policies in SPdb is due to the fact that DIANA uses a linear linked-list to store the policies in SPdb. A performance comparable to the case of 1 policy in SPdb can be obtained even for a large number of policies in SPdb by using a *hash table* data structure instead of a linked-list.

6.8 Summary of Results

As seen above, the overhead of IP-interception is negligible compared to the times for the phase 1 and phase 2 SA-negotiations by Pluto. If a filtering mechanism is present in IP-interception, then, only the first few packets of a traffic stream would be intercepted and the delay on account of DIANA-processing for remaining packets would be nil. But, the overhead of matching packets to policies will be passed on to the kernel.

If an ISAKMP-SA already exists between the communicating peers, then they need to carry out only phase 2 IPSec SA negotiation. As seen above, this delay is about 30 milli-seconds for the given experimental setup. This delay might be tolerable at the beginning of a traffic session. Even if the ISAKMP-SA were to be established using phase 1 negotiation, the initial delay would be about 0.2 seconds, which may still be tolerable for most applications (notably those using TCP). In TCP, for example, the initial connection-setup packets will trigger the SA-establishment process, and subsequent traffic will not incur additional delays waiting for the SA to be established.

Chapter 7

Conclusion

7.1 Summary

The basic IPSec architecture has already been standardized by the IETF. Though this architecture has been there for some time now, the only prominent application so far has been the Virtual Private Network (VPN). Most current implementations of IPSec support only *static* SA-establishment. Though this type of SA-establishment is suitable for applications like VPN, which need to setup the SAs only once, it is not very useful for advanced applications like DECIDUOUS, which require the establishment of a large number of SAs. Such applications require SAs to be established only when (and if) there is network traffic between two communicating entities.

The IPSec architecture does mention the need for a *traffic-driven* approach to SA-establishment; but none of the current implementations supports it. This thesis was the first such attempt to realize dynamic traffic-driven SA-establishment for IPSec and show its feasibility. The thesis primarily focused on the design, implementation and performance measurement of a traffic-driven approach to SA-establishment for IPSec. Towards this, a utility called DIANA was implemented and tested. The performance measurements of DIANA, Pluto and IP-interception mechanism show that the overhead of the traffic-driven approach to SA-establishment is tolerable for most applications.

The results in chapter 6 indicate that it takes about 0.2 seconds to establish a phase 1 SA with a machine exactly one hop away. If the source is sending packets at a very high rate, this might mean that a large number of packets will be queued in DIANA. But, if the initial rate of packets is low, the number of packets queued will be less. The latter

is more likely as most of the Internet traffic is TCP traffic and TCP performs a three-way handshake at the time of connection-setup. In such a case, only one packet is likely to be queued in DIANA while the SA is established.

DIANA can be used by any advanced application which needs to establish a large number of SAs. An example of such an application called DECIDUOUS was described in chapter 3.

7.2 Future Work

7.2.1 Filtering mechanism for IP Interception

As mentioned in chapter 4, DIANA intercepts *all* packets to the user-space. Though the interception of the first few packets of a network traffic is acceptable, that of subsequent packets is not. This is so, because the first few packets (in TCP) are connection-setup packets, which can tolerate a delay introduced by DIANA. But the subsequent packets can arrive at a very high rate, and may be dropped by DIANA if the queues within DIANA become full. This will affect the throughput of the connection, especially due to the slow-start in TCP.

To avoid this, DIANA should attempt to intercept only the first few packets of a network traffic session. This can be done, if a filtering mechanism is implemented in the IP-interception mechanism, whereby DIANA can control when (and which) packets are to be intercepted. Such an interception mechanism has been recently implemented in the Computer Science Department at North Carolina State University. The next step for DIANA would be to use this mechanism to avoid unnecessary interception of IP packets.

7.2.2 Data Structure for SPdb

Currently, DIANA uses a linked-list to store policies in the SPdb. This takes a time linearly proportional in the number of policies to search a matching policy for a given packet. For a large number of policies, this matching of policies to packets introduces an appreciable delay as shown in the previous chapter. The performance of DIANA can improve if a data structure like the *hash table* is used instead of linked-list for storing policies in the SPdb.

7.2.3 Kernel Implementation of SPdb

As was mentioned in chapter 4, the current FreeS/WAN implementation of IPSec does not support SPdb in the kernel. It is expected to do so in the future releases. It is also expected to provide an API for the modification of SPdb from the user space. When this is done, the functionality of DIANA can be moved to the kernel.

7.2.4 Support for IPv6

DIANA currently supports only IPv4 packets. As future network traffic will also include IPv6 packets, DIANA needs to recognize and handle these packets. Some parts of DIANA, notably the functions that match packets to policies, need to be modified for this.

Bibliography

[1] Comer, D. E. *Internetworking with TCP/IP, Volume* 1 - *Principles, Protocols and Architecture, Third Edition* Prentice Hall, 1995.

[2] Comer, D. E. *Internetworking with TCP/IP, Volume* 3 - *Client-Server Programming and Applications, BSD socket version.* Prentice Hall, 1993.

[3] DecIdUouS Home Page. `http://shang.csc.ncsu.edu/deciduous/`. May 1999.

[4] Harkins, D., Carrel, D. The Internet Key Exchange (IKE). RFC 2409, November 1998.

[5] Internet Domain Survey. `http://www.nw.com/zone/WWW/report.html`. January 1999.

[6] Kaufman, C., Perlman, R. and Speciner, M. *Network Security, Private Communication in a Public World.* Prentice Hall, 1995.

[7] Kent, S., Atkinson, R. IP Authentication Header. RFC 2402, November 1998.

[8] Kent, S., Atkinson, R. IP Encapsulating Security Payload (ESP). RFC 2406, November 1998.

[9] Kent, S., Atkinson, R. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[10] Maughan, D., Schertler, M., Schneider, M., Turner, J. Internet Security Association and Key Management Protocol (ISAKMP). RFC 2408, November 1998

[11] Stallings, W. *Cryptography and Network Security, Principles and Practice, Second Edition.* Prentice Hall, 1999.

[12] Stevens, W. R. *Unix Network Programming, Networking APIs: Sockets and XTI, Volume 1, Second edition.* Prentice Hall, 1998.

[13] The Linux FreeS/WAN Project. `http://www.xs4all.nl/freeswan`. 14 April 1999.

[14] Wu, S. F., Sargor, C., et.al. Designing DECIDUOUS: Decentralized Source Identification for Network-Based Intrusions. Technical report, Department of Computer Science, North Carolina State University, April 1998.

[15] Wu, S. F., Wu, C. Divert Sockets and IP-interception on Linux. Technical report, Department of Computer Science, North Carolina State University, 1999.

Appendix A

DIANA Source Code

 # file: README
 # This file lists the files in the DIANA source code.

Makefile	: Compilation procedures and dependencies
diana.h	: Header file for DIANA
diana_controller.c	: Code for Diana-Controller thread
install_ipsec_sa.c	: Code to modify 'kernel.c' file in Pluto
ip_pkt_proc.c	: Code for IP_Pkt_Proc module within Diana-Controller
main.c	: The main file for DIANA which starts four threads
net_interface.c	: Code for interfacing with the IP interception mechanism
pluto_proc.c	: Code for Pluto_Proc module within Diana-Controller
pluto_reader.c	: Code for the Pluto_Reader thread which reads diana_messages from Pluto
queue.c	: Various functions for the 'queue' data-structure
queue_reader.c	: Code for Queue_Reader module within Diana-Controller
read_spdb.c	: Code for reading policies from SPdb-file and converting them to the internal data-structures
update_spdb.c	: Code for Update_SPdb module within Diana-Controller
whack.c	: Code for sending messages to Pluto

```
# file: Makefile
# This is a makefile for DIANA
#

CC = gcc

FREESWANLIBDIR=../lib
FREESWANINCLS= -I$(FREESWANLIBDIR)
FREESWANLIB=$(FREESWANLIBDIR)/libfreeswan.a

CFLAGS = -g -Wall -Wmissing-prototypes #-O

HDRDIRS = $(FREESWANINCLS)

CPPFLAGS = $(HDRDIRS) -DDEBUG $(BYTE_ORDER) -DKLIPS -DDIANA

ALLFLAGS = $(CPPFLAGS) $(CFLAGS)

RM = /bin/rm
RMFLAGS = -f

.SUFFIXES:
.SUFFIXES: .c .o

DISTSRC = diana.h diana_controller.c ip_pkt_proc.c \
        main.c net_interface.c pluto_proc.c pluto_reader.c \
        queue.c queue_reader.c read_spdb.c update_spdb.c whack.c

OBJSDIANA = diana_controller.o ip_pkt_proc.o main.o net_interface.o \
        pluto_proc.o pluto_reader.o queue.o queue_reader.o read_spdb.o \
        update_spdb.o whack.o $(FREESWANLIB)

dianad: $(OBJSDIANA)
        $(CC) -o dianad $(OBJSDIANA) -lpthread

clean:
        $(RM) dianad *.o

.c.o:
        $(CC) $(ALLFLAGS) -c $<

# This rule is not for production use
$(FREESWANLIB):
        cd $(FREESWANLIBDIR) ; $(MAKE)

# other dependencies
diana_controller.o: diana_controller.c
diana_controller.o: diana.h
ip_pkt_proc.o: ip_pkt_proc.c
ip_pkt_proc.o: diana.h
main.o: main.c
main.o: diana.h
```

```
net_interface.o: net_interface.c
net_interface.o: diana.h
pluto_proc.o: pluto_proc.c
pluto_proc.o: diana.h
pluto_reader.o: pluto_reader.c
pluto_reader.o: diana.h
queue.o: queue.c
queue.o: diana.h
queue_reader.o: queue_reader.c
queue_reader.o: diana.h
read_spdb.o: read_spdb.c
read_spdb.o: diana.h
update_spdb.o: update_spdb.c
update_spdb.o: diana.h
whack.o: whack.c
whack.o: diana.h
```

```
/* file diana.h
 * This is the header file for DIANA */
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/time.h>
#include <unistd.h>
#include <signal.h>

#include "../pluto/constants.h"
#include "../pluto/whack.h"

#define MAXQLEN 1024        /* maximum length of queues in DIANA controller */
#define MAXBUFLEN 1000      /* maximum buffer size for storing IP packet */
#define MAXTRIES 3          /* maximum number of tries for establishing a
                             * security association */
#define MAC_HDR_SIZE 14     /* size of the MAC header */

typedef int boolean;
#define FALSE 0
#define TRUE  1

#define INACTIVE 0
#define ACTIVE   1
#define PENDING  2

#define FAILURE 0
#define SUCCESS 1

#define Update_SPdb 0
#define IP_Pkt      1
```

```
#define Pluto_Resp  2

#define Teardown     0
#define Establish_SA 1

#define PID_FILE "/var/run/diana_pid" /* contains pid of DIANA process */
#define SPDB_FILE "spdb.txt"          /* contains policies */

/* struct Policy */
typedef struct {
  unsigned int policy_id;           /* unique ID of the policy */
  struct in_addr local_client_net; /* source address in network order */
  struct in_addr local_client_mask;

  struct in_addr peer_client_net;  /* destination address in network order */
  struct in_addr peer_client_mask;

  boolean xpt_protocol_wildcard;   /* TRUE -> wildcard i.e. any protocol
                                    * FALSE -> no wildcard i.e.
                                    * specific protocol */
  u_char xpt_protocol;              /* transport layer protocol */

  boolean src_port_wildcard;       /* TRUE  -> wildcard i.e. any port
                                    * FALSE -> no wildcard */
  u_int16_t src_port;              /* transport layer source port in
                                    * network order */
  boolean dst_port_wildcard;       /* TRUE -> wildcard i.e. any port
                                    * FALSE -> no wildcard, specific port */
  u_int16_t dst_port;              /* transport layer destination port in
                                    * network order */
  u_int16_t  user_id;              /* not currently used */
  struct in_addr peer_addr;         /* address of the peer security gateway in
                                    * network order */
  u_int16_t peer_port;              /* UDP port of peer security gateway with
                                    * which Pluto will communicate */
  boolean encry;                   /* Whether to do encryption
                                    * TRUE  -> encrypt
                                    * FALSE -> do not encrypt */
  boolean auth;                    /* Whether to do authentication
                                    * TRUE  -> authenticate
                                    * FALSE -> do not authenticate */
  boolean tunnel;                  /* Whether tunnel mode or transport mode
                                    * TRUE  -> tunnel mode
                                    * FALSE -> transport mode */
  boolean discard;                 /* Whether to discard this packet or not
                                    * TRUE  -> discard packet
                                    * FALSE -> do not discard packet */
  int status;                      /* Whether policy is ACTIVE, PENDING
                                    * or INACTIVE */
  boolean mark;                     /* used while updating SPdb tables
                                    * TRUE  -> marked
                                    * FALSE -> unmarked */
  unsigned int try_count;           /* used to count number of tries in
                                    * establishing a Security Association */
} Policy;
```

```
/* struct diana_message, message format for message from Pluto to DIANA */
typedef struct {
  unsigned int status;      /* FAILURE -> could not establish IPSec SA
                            * SUCCESS -> IPSec SA already in existence
                            *            or new IPSec SA created */
  Policy policy;
} diana_message;

/* message format for queue items */
typedef struct Qmesg_t {
  unsigned int type;        /* Update_SPdb, IP_Pkt, Pluto_Resp */
  unsigned int mesgid;      /* used for performance evaluation */
  struct timeval tv;
  void *dat;                /* Null, IP packet, diana_message  */
  struct Qmesg_t *next;
} Qmesg;

/* queue data structure */
typedef struct {
  int length;               /* length of the queue */
  Qmesg *first;             /* first element of the queue */
  Qmesg *last;              /* last element of the queue */
  pthread_mutex_t mutex;    /* to ensure that only a single thread
                            * accesses the queue at a given time */
  pthread_cond_t  not_empty;/* to avoid busy waits for readers of queue */
  pthread_cond_t empty;     /* used for optimization ... OutQ should be flushed
                            * before the net_output thread is cancelled */
} Queue;

/* struct SPdb_table */
typedef struct SPdb_table_t {
  Policy *p;
  Queue *q;
  struct SPdb_table_t *next;
} SPdb_table;

/* packet header information data structure */
typedef struct {
  u_int32_t src_addr;
  u_int32_t dst_addr;
  u_int8_t  xpt_protocol;
  u_int16_t src_port;
  u_int16_t dst_port;
} Header;

/* functions */
extern void diana_controller (Queue *InQ, Queue *OutQ, char *spdb_filename);
extern void queue_reader      (Queue *InQ, Queue *OutQ, SPdb_table *SPdb_Active,
                           SPdb_table *SPdb_Pending, char *spdb_filename);
extern Qmesg *dequeue         (Queue *queue);
extern void update_spdb       (Queue *InQ, Queue *OutQ,SPdb_table **SPdb_Active,
                           SPdb_table **SPdb_Pending, char *spdb_filename);
extern void pluto_proc        (Queue *OutQ, SPdb_table **SPdb_Active,
                           SPdb_table **SPdb_Pending, diana_message *mesg);
```

```
extern void ip_pkt_proc  (Queue *OutQ, SPdb_table *SPdb_Pending, Qmesg *mesg);
extern void compare      (SPdb_table ** SPdb_buf, SPdb_table *table);
extern void whack        (int request, Policy *p);
extern void delete_entry (SPdb_table **prev, SPdb_table **elem,                    150
                          SPdb_table **table);
extern void unmark_entries      (SPdb_table *table);
extern void appendq             (Queue *q1, Queue *q2);
extern boolean match_policy     (Policy *policy1, Policy *policy2);
extern void extract_header_info (Header *hdr, unsigned char *packet);
extern boolean match_packet     (SPdb_table **matched_entry,
                                 Header hdr, SPdb_table *table);
extern void flushq    (Queue **queue);
extern void initq     (Queue *queue);
extern void enqueue   (Qmesg *mesg, Queue *queue);                                 160
extern int read_spdb  (SPdb_table **policy_table, char *spdb_filename);
extern int print_spdb (SPdb_table *policy_list);
extern void * diana_sig_handler    (int sig);
extern void * pluto_reader         (void *arg);
extern void * net_input            (void *arg);
extern void * net_output           (void *arg);
extern int isspacestr              (char *line);
extern boolean open_taps           (Queue *InQ);
extern boolean close_taps          (void);
extern void thread_cleanup_routine (void *arg);                                    170
```

```
/* file: diana_controller.c
 * This file contains the code for DIANA controller thread
 * Pseudo code:
 *              Initialize IP Packet Buffer, SPdb Active table and
 *              SPdb Pending table
 *              Enable SIGHUP signal handler
 *              Read SPdb from file into internal data structures
 *              Call the function Queue Reader
 */
                                                                                   10
#include "diana.h"

static Queue *InQ_g; /* global variable used so that signal handler can
                      * access the variable InQ */

/* function diana_sig_handler */
void * diana_sig_handler(int sig) {
  Qmesg *mesg;
  mesg = (Qmesg *) malloc (sizeof (Qmesg)) ;
  mesg->type = Update_SPdb;                                                        20
  mesg->dat  = NULL;

  if (sig == SIGHUP) {
    enqueue (mesg, InQ_g);
    signal(SIGHUP, (__sighandler_t) diana_sig_handler);
  }
  return (NULL);
} /* end of diana_sig_handler */
```

```
/* function diana_controller */
void diana_controller (Queue *InQ, Queue *OutQ, char *spdb_filename) {

  SPdb_table *SPdb_Active = NULL, *SPdb_Pending = NULL;

  /* initialize the global variable InQ_g */
  InQ_g = InQ;

  /* set signal handler */
  signal(SIGHUP, (__sighandler_t) diana_sig_handler);

  /* read SPdb from file into SPdb_Pending table */
  read_spdb(&SPdb_Pending, spdb_filename);

  if (SPdb_Pending == NULL) {
     sleep(1);   /* a hack to let other threads catch up */
     close_taps();
  }

  /* Pass control to the function queue_reader */
  queue_reader(InQ, OutQ, SPdb_Active, SPdb_Pending, spdb_filename);

} /* end diana_controller() */
```

```
/* file: install_ipsec_sa.c
 * This file contains bits of code which is to be added to the 'kernel.c' file
 * in the Pluto implementation so that Pluto replies back to DIANA
 */

#ifdef DIANA
#include "../diana/diana.h"
#else
#include "constants.h"
#endif

.....
.....

#ifdef DIANA
/* function reply_diana()
 * send a reply message to DIANA about whether the SA-establishment
 * was successful or not.
 * Pseudo-code:
 *        compose a 'diana_message' from status and state information
 *        setup udp socket for writing
 *        send 'diana_message' on the socket */
void reply_diana (bool status, struct state *st, unsigned int dummy) {
  diana_message dmesg;
  struct sockaddr_in *peer;
  unsigned long pluto_port = IKE_UDP_PORT + 2;

  /* first compose a 'diana_message' */
```

```
memset (&dmesg, '\0', sizeof(dmesg));

if (status == TRUE)
  dmesg.status = SUCCESS;
else
  dmesg.status = FAILURE;

dmesg.policy.local_client_net  = st->st_myuser.sin_addr;
dmesg.policy.local_client_mask = st->st_myusernetmask.sin_addr;
dmesg.policy.peer_client_net   = st->st_peeruser.sin_addr;
dmesg.policy.peer_client_mask  = st->st_peerusernetmask.sin_addr;

dmesg.policy.user_id = dummy;

peer = (struct sockaddr_in *) (& (st->st_peer));
dmesg.policy.peer_addr = peer->sin_addr;
dmesg.policy.peer_port = (peer->sin_port >> 8) | (peer->sin_port << 8);

if ( (st->st_goal & GOAL_ENCRYPT) == 0 )
  dmesg.policy.encry = FALSE;
else
  dmesg.policy.encry = TRUE;

if ( (st->st_goal & GOAL_AUTHENTICATE) == 0 )
  dmesg.policy.auth = FALSE;
else
  dmesg.policy.auth = TRUE;

if ( (st->st_goal & GOAL_TUNNEL) == 0 )
  dmesg.policy.tunnel = FALSE;
else
  dmesg.policy.tunnel = TRUE;

dmesg.policy.discard = FALSE;

/* now setup udp socket and send the 'diana_message' to DIANA */
{
  int sock = socket(PF_INET, SOCK_DGRAM, 0); /* protocol 0 for IP */
  struct sockaddr_in sin;

  if (sock == -1)
    {
      perror("function reply_diana(): socket() failed");
      exit(1);
    }

  memset(&sin, '\0', sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
  sin.sin_port = htons(pluto_port);

  if (sendto(sock, &dmesg, sizeof(dmesg),
             0, (struct sockaddr *)&sin, sizeof(sin)) == -1)
    {
      perror("function reply_diana(): sendto() DIANA failed");
```

```
        exit(1);
      }

    close (sock);

  } /* end send message to DIANA */
                                                                    90
} /* end reply_diana() */
#endif /* DIANA */

bool
install_ipsec_sa(struct state *st, bool initiator)
{
    unsigned int dummy;

    /* do routing commands, if we can */
    if (!do_route(st, TRUE))                                        100
#ifdef DIANA
      {
        reply_diana(FALSE, st, 1);
#endif /* DIANA */
        return FALSE;
#ifdef DIANA
      };
#endif /* DIANA */

#ifdef KLIPS                                                        110
    if (no_klips)
    {
#ifdef DIANA
      {
        int retval = do_eroute(-1, st, EMT_SETEROUTE)
          && setup_ipsec_sa(-1, st, initiator);
        reply_diana (retval, st, 2);
        return retval;
      }
#else /* DIANA */                                                   120
      return do_eroute(-1, st, EMT_SETEROUTE)
        && setup_ipsec_sa(-1, st, initiator);
#endif /* DIANA */
    }
    else
    {
        int fd = open("/dev/ipsec", O_WRONLY);
        bool res;

        if (fd < 0)                                                 130
        {
            log_errno("open() of /dev/ipsec failed in install_ipsec_sa()");
#ifdef DIANA
            reply_diana(FALSE, st, 3);
#endif /* DIANA */
            return FALSE;
        }
```

```
        dummy = 8;
        res = do_eroute(fd, st, EMT_SETEROUTE);                                  140

        if (res)
        {
            dummy = 12;
            res = setup_ipsec_sa(fd, st, initiator);
            if (!res) {
                dummy = 16;
                (void)do_eroute(fd, st, EMT_DELEROUTE);
            }
        }                                                                        150

        close(fd);      /* rain or shine */

#ifdef DIANA
        reply_diana(res, st, dummy);
#endif /* DIANA */
        return res;
    }
#else /* KLIPS */
    DBG(DBG_CONTROL, DBG_log("if I knew how, I'd do_eroute() and setup_ipsec_sa()"));   160
#ifdef DIANA
    reply_diana(FALSE, st, 5); /* is this OK ? */
#endif /* DIANA */
    return TRUE;
#endif /* KLIPS */
}

.....
.....
```

```
/* file: ip_pkt_proc.c
 * This file contains the code for the function ip_pkt_proc() */

#include "diana.h"
#include <netinet/tcp.h>
#include <netinet/udp.h>

void ip_pkt_proc (Queue *OutQ,  SPdb_table *SPdb_Pending,  Qmesg *mesg) {

  Header hdr;                  /* holder for IP-header information */            10
  unsigned char *packet;         /* buffer to hold packet */
  SPdb_table *matched_entry;     /* table entry containing matched policy */

  packet = (unsigned char *) (mesg->dat);

  extract_header_info (&hdr, packet);

  /* match_packet should match ranges of IP values */
  if ( (match_packet (&matched_entry, hdr, SPdb_Pending)) == FALSE )
    /* send the packet to OutQ */                                              20
    enqueue (mesg, OutQ);
```

```
  else /* a matched policy is found */ {
    /* if discard flag is set for the policy, discard the packet */
    if (matched_entry->p->discard == TRUE) {
      /* move the dat pointer so that it includes the MAC header */
      mesg->dat -= MAC_HDR_SIZE;
      free (mesg->dat);
      free (mesg);
    }
    /* else if the SA is being established, store the packet in corresponding
     * queue
     */
    else if (matched_entry->p->status == PENDING) {
      /* first check if the queue-length has been reached */
      if (matched_entry->q->length == MAXQLEN) {

        /* discard the packet */
        /* move the dat pointer so that it includes the MAC header */
        mesg->dat -= MAC_HDR_SIZE;
        free (mesg->dat);
        free (mesg);
      }
      else
        enqueue (mesg, matched_entry->q);
    }
    /* else store the packet in the corresponding queue, update status of
     * policy and send an 'Establish_SA' message to whack function
     */
    else /* status == INACTIVE */ {
      enqueue (mesg, matched_entry->q);
      matched_entry->p->status = PENDING;
      matched_entry->p->try_count = 1;
      whack (Establish_SA, matched_entry->p);
    }
  } /* end else matched entry is found */
} /* end ip_pkt_proc() */

/* function extract_header_info()
 * extracts fields from the ip-header and tcp/udp header necessary for
 * diana-processing */

void  extract_header_info (Header *hdr, unsigned char *packet) {
  struct ip *ip_hdr;
  ip_hdr = (struct ip *) packet;

  hdr->src_addr = ip_hdr->ip_src.s_addr;
  hdr->dst_addr = ip_hdr->ip_dst.s_addr;
  hdr->xpt_protocol = ip_hdr->ip_p;

  /* if protocol is tcp or udp, get the src and dst ports */
  if (ip_hdr->ip_p == IPPROTO_TCP) {
    struct tcphdr *xpt_hdr;
    xpt_hdr = (struct tcphdr *) (packet + ip_hdr->ip_hl);
    hdr->src_port = xpt_hdr->source;
    hdr->dst_port = xpt_hdr->dest;
  }
```

```
  if (ip_hdr->ip_p == IPPROTO_UDP) {
    struct udphdr *xpt_hdr;
    xpt_hdr = (struct udphdr*) (packet + ip_hdr->ip_hl);
    hdr->src_port = xpt_hdr->source;
    hdr->dst_port = xpt_hdr->dest;
  }

} /* end extract_header_info() */

/* function match_packet()
 * matches packet header with policy entries in 'table'
 * returns TRUE and the pointer to the matched entry if an entry is found
 * returns FALSE if entry is not found
 * matches range of IP addresses */
boolean match_packet (SPdb_table **matched_entry, Header hdr,
                      SPdb_table *table) {
  *matched_entry = table;

  if ((hdr.xpt_protocol == IPPROTO_TCP) || (hdr.xpt_protocol == IPPROTO_UDP))
    while ((*matched_entry) != NULL) {
      Policy *p;
      p = (*matched_entry)->p;

      if ( ( (hdr.src_addr & p->local_client_mask.s_addr)
             == (p->local_client_net.s_addr & p->local_client_mask.s_addr))
           && ( (hdr.dst_addr & p->peer_client_mask.s_addr)
                == (p->peer_client_net.s_addr & p->peer_client_mask.s_addr))
           && ( (p->xpt_protocol_wildcard) ||
                (hdr.xpt_protocol == p->xpt_protocol) )
           && ( (p->src_port_wildcard) || (hdr.src_port  == p->src_port) )
           && ( (p->dst_port_wildcard) || (hdr.dst_port  == p->dst_port) )
           )
        break;
      else
        *matched_entry = (*matched_entry)->next;
    } /* end while */

  else /* if protocol is neither TCP nor UDP, don't match ports */
    while (*matched_entry != NULL) {
      Policy *p;
      p = (*matched_entry)->p;

      if ( ( (hdr.src_addr & p->local_client_mask.s_addr)
             == (p->local_client_net.s_addr & p->local_client_mask.s_addr))
           && ( (hdr.dst_addr & p->peer_client_mask.s_addr)
                == (p->peer_client_net.s_addr & p->peer_client_mask.s_addr))
           && ( (p->xpt_protocol_wildcard) ||
                (hdr.xpt_protocol == p->xpt_protocol) )
           )
        break;
      else
        *matched_entry = (*matched_entry)->next;
    } /* end while */
```

```
  if (*matched_entry == NULL)
    return FALSE;
  else
    return TRUE;
} /* end match_packet () */
```

```
/* file: main.c
 * This is the starting file of DIANA.
 * Pseudo code:
 *              Spawn 4 threads: DIANA controller, Pluto Reader, Net Input
 *                               and Net Output */

#include "diana.h"

void main(int argc, char *argv[]) {

  pthread_t t2, t3, t4;
  pthread_attr_t *attr;
  Queue InQ, OutQ;
  char spdb_filename[] = SPDB_FILE;  /* currently hardcoded, can be input from
                                      * command line */
  FILE *fp;

  /* process command line options and parameters (not yet) */

  /* initialize queues */
  initq(&InQ);
  initq(&OutQ);

  /* write the pid of process in PID_FILE */
  fp = fopen (PID_FILE, "w+");
  fprintf(fp, "%d", getpid());
  fclose(fp);

  /* spawn 3 threads */
  attr = NULL;

  /* thread pluto_reader */
  if ( pthread_create (&t2, attr, pluto_reader, &InQ) != 0 ) {
    perror("diana main : Error while creating thread t2");
    exit(1);
  }
  pthread_detach (t2);

  /* thread net_input */
  if ( pthread_create (&t3, attr, net_input, &InQ) != 0 ) {
    perror("diana main : Error while creating thread t3");
    exit(1);
  }
  pthread_detach (t3);

  /* thread net_output */
  if ( pthread_create (&t4, attr, net_output, &OutQ) != 0 ) {
```

```
    perror("diana main : Error while creating thread t4");
    exit(1);
  }
  pthread_detach (t4);

  /* the current thread will continue as DIANA controller */
  diana_controller(&InQ, &OutQ, spdb_filename);

} /* end main() */
```

```
/* file: net_interface.c
 * This file contains code for the threads Net Input and Net Output
 */

#include "diana.h"
#include <string.h>


/* global variables for this file */
static pthread_t net_inp_thread, net_out_thread;
static int tap_open;
static int fdr, fdw;
static unsigned int counter = 0;

/* Pseudo code for Net Input:
 *      Loop indefinitely:
 *          Read IP packets from kernel by IP interception read mechanism
 *          Compose and send a Qmesg messge of type 'IP_Pkt' along with
 *          the IP packet to InQ
 *      End Loop */

void * net_input (void *arg) {
  int oldtype;
  Queue *InQ;
  Qmesg *mesg;
  unsigned char readbuf [MAXBUFLEN + 1] , *pktbuf;
  unsigned int rl;

  tap_open = 1;
  net_inp_thread = pthread_self();

  /* setting type to DEFERRED means that the thread is killed
   * at a cancellation point
   */
  pthread_setcanceltype (PTHREAD_CANCEL_DEFERRED, &oldtype);

  InQ = (Queue *) arg;

  /* open taph for reading */
  if ((fdr = open("/dev/taph", O_RDONLY))<0) {
    perror("Net Input: failure in opening /dev/taph for reading ");
    exit (1);
  };
```

```
  /* Loop indefinitely */
  while (TRUE) {
    /* initialize mesg */
    mesg = (Qmesg *) malloc (sizeof (Qmesg));
    mesg->type = IP_Pkt;
                                                                        50
    /* read IP packet */

    /* the pthread_testcancel() calls are included below, as the current
     * version of LinuxThreads does not support the read() system call
     * as a cancellation point as mentioned in the POSIX specification of
     * threads.  Please see man pthread_cancel.
     */
    pthread_testcancel();
    if ( (rl = read (fdr, readbuf, MAXBUFLEN)) < 0) {
      pthread_testcancel();                                             60
      perror ("Net Input: Error reading from /dev/taph ");
    }
    else {
      pktbuf = (unsigned char *) malloc ( (rl + 1) * sizeof (unsigned char) );
      readbuf[rl]='\0';
      memcpy (pktbuf, readbuf, rl + 1);

      /* Note: The packet read contains MAC header (14 bytes)
       * So we need to readjust the readbuf buffer so that it points
       * to the ip-packet */                                           70
      pktbuf += MAC_HDR_SIZE;
      mesg->dat = (void *) pktbuf;

      enqueue (mesg, InQ);
    }
  } /* end while */
} /* end net_input() */

/* Pseudo code for Net Output:
 *       Loop indefinitely:                                           80
 *            Read Qmesg messages from OutQ
 *            Extract IP packets from these messages
 *            Send the IP packets to kernel
 *       End loop */
void * net_output (void *arg) {
  Queue *OutQ;
  int oldtype;
  Qmesg *mesg;
  unsigned char *writebuf;
  u_int16_t pktlen;                                                     90

  net_out_thread = pthread_self();

  /* setting type to DEFERRED means that the OutQ will be first flushed
   * before the thread is cancelled. */
  pthread_setcanceltype (PTHREAD_CANCEL_DEFERRED, &oldtype);

  OutQ = (Queue *) arg;
```

```
  /* open tapl for writing */
  if ((fdw = open("/dev/tapl", O_WRONLY))<0) {
    perror ("Net Output: failure in opening /dev/tapl for writing ");
    exit(1);
  };

  /* Loop indefinitely */
  while (TRUE) {
    /* read Qmesg from OutQ */
    mesg = dequeue (OutQ); /* dequeue implements a conditioned wait */

    /* extract IP packet from Qmesg */
    writebuf = (unsigned char *) (mesg->dat);

    /* find the length of IP packet from the ip-header */
    memcpy (&pktlen, writebuf + 2, 2);
    /* the following is needed due to byte ordering */
    pktlen = (pktlen << 8) | (pktlen >> 8);

    /* re-adjust the writebuf buffer, so that it again includes the
     * MAC header (note that this memory location is available)
     */
    writebuf -= MAC_HDR_SIZE;
    pktlen   += MAC_HDR_SIZE;

    /* write the IP packet to kernel */
    if ( ( write (fdw, writebuf, pktlen )) < 0)
      perror ("Net Output(): Error writing to /dev/tapl ");
    else {
      free (writebuf);
      free (mesg);
    }
  } /* end while */
} /* end Net Output */

boolean open_taps(Queue *InQ) {

  if (tap_open == 0) {
    if ( pthread_create (&net_inp_thread, NULL, net_input, InQ) != 0 ) {
        perror ("diana open_taps(): Error while creating thread net_input\n");
        exit(1);
    }
     pthread_detach (net_inp_thread);
     tap_open = 1;
  }
  return TRUE;
}

/* close only input tap */
boolean close_taps() {

  if (tap_open == 1) {
    /* cancel threads */
    if ( pthread_cancel (net_inp_thread) != 0 )
```

```
      perror ("close_taps(): Error while canceling thread net_input ");

    /* close taps */
    close(fdr);
    tap_open = 0;
  }
  return TRUE;                                                          160
}
```

```
/* file: pluto_proc.c
 * This file contains the code for function pluto_proc()
 */

#include "diana.h"

void pluto_proc (Queue *OutQ,  SPdb_table **SPdb_Active,
                 SPdb_table **SPdb_Pending, diana_message *mesg) {

  SPdb_table *curr, *prev;                                              10

  /* find the policy in SPdb_Pending table which matches mesg->policy */
  prev = NULL;
  curr = *SPdb_Pending;

  while (curr != NULL) {
    if  ( ( match_policy (&(mesg->policy), curr->p) == TRUE )
        && (curr->p->status == PENDING) )
      break;
    prev = curr;                                                        20
    curr = curr->next;
  }

  /* Now either curr == NULL, or curr points to some matched policy.
   * if curr == NULL, we need to do nothing.  This can happen, for
   * example, if the policy got deleted before we got a reply from Pluto. */

  if (curr != NULL) {
    /* Now switch depending on whether Pluto was able to create a Security
     * Association or not                                               30
     */
    switch (mesg->status) {

    case SUCCESS: /* Pluto was able to create a Security Association */
      /* remove the policy from SPdb_Pending table */
      if (prev != NULL)
        prev->next = curr->next;
      else
        *SPdb_Pending = curr->next;
                                                                        40
      /* Set the status of the policy to ACTIVE and add it to SPdb_Active
       * table */
      curr->p->status = ACTIVE;
      curr->next = *SPdb_Active;
```

```
      *SPdb_Active = curr;

      /* Append the corresponding packet queue to OutQ */
      appendq (curr->q, OutQ);
      curr->q = NULL;

      /* if there are no more entries in SPdb_Pending table, close the
       * IP interception mechanism */
      if (*SPdb_Pending == NULL) {
        close_taps(); /* this function implemented in net_interface.c */
      }
      break;

    case FAILURE: /* Pluto was not able to create a Security Association */

      /* check if number of tries == MAXTRIES */
      if (curr->p->try_count == MAXTRIES) {
        SPdb_table *next;

        /* discard IP packets from corresponding queue, and free memory */
        flushq (&(curr->q));

        /* set the flag of the policy to INACTIVE */
        curr->p->status = INACTIVE;

        /* reset the try_count value */
        curr->p->try_count = 0;

        /* Give an error message */
        fprintf (stderr, "DIANA: Could not create SA for policy:\n");
        next = curr->next;
        curr->next = NULL;
        print_spdb (curr);
        printf ("dummy = %d\n", mesg->policy.user_id);
        curr->next = next;  /* this is just for reusing print_spdb() */
      }
      else {
        (curr->p->try_count)++;

        /* send an 'Establish_SA' message to 'whack' */
        whack (Establish_SA, curr->p);
      }
    } /* end switch */
  } /* end if curr != NULL */

  /* free memory held up by mesg : this is done by the calling (queue_reader)
   * function  */
} /* end pluto_proc() */

/* function match_policy.
 * if two policies match, return TRUE, else return FALSE */

boolean match_policy (Policy *p1, Policy *p2) {

 if ((p1 == NULL) && (p2 == NULL))
```

```
   return TRUE;                                                          100

 if ((p1 != NULL) && (p2 != NULL)) {
  if ( (p1->local_client_net.s_addr == p2->local_client_net.s_addr)
       && (p1->local_client_mask.s_addr == p2->local_client_mask.s_addr)
       && (p1->peer_client_net.s_addr == p2->peer_client_net.s_addr)
       && (p1->peer_client_mask.s_addr == p2->peer_client_mask.s_addr)
       /*       && (p1->xpt_protocol == p2->xpt_protocol)
        *       && (p1->src_port == p2->src_port)
        *       && (p1->dst_port == p2->dst_port)
        *       && (p1->user_id  == p2->user_id)                        110
        */
       && (p1->peer_addr.s_addr == p2->peer_addr.s_addr)
       && (p1->peer_port == p2->peer_port)
       && (p1->encry == p2->encry)
       && (p1->auth == p2->auth)
       && (p1->tunnel == p2->tunnel)
       && (p1->discard == p2->discard)
       )
    return TRUE;
  }                                                                      120
  return FALSE;
} /* end match_policy () */
```

```
/* file pluto_reader.c
 * This file contains the function pluto_reader()
 * This function runs indefinitely as a separate thread
 * Pseudo-code:
 *     Loop indefinitely:
 *          Read diana_messages from Pluto from specified UDP port
 *          Compose and send Qmesg of type 'Pluto_Resp' along with
 *                diana_message to InQ
 *     End loop */
                                                                         10
#include <string.h>  /* for memset */
#include "diana.h"
#include "../pluto/defs.h"

void * pluto_reader (void *arg) {

  Queue *InQ;
  diana_message *mesg;
  int n;
  unsigned long pluto_port = IKE_UDP_PORT + 2; /* IKE_UDP_PORT is defined in   20
                                        * constants.h */
  struct sockaddr_in sin;
  int s;

  InQ = (Queue *) arg;

  s = socket (PF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if (s == -1) {
    perror("DIANA: socket() failed in pluto_reader() ");
```

```
    exit(1);
  }

  mksin (sin, htonl(INADDR_ANY), pluto_port);
  if (bind (s, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
    perror ("DIANA: bind() failed in pluto_reader() ");
    exit(1);
  }

  /* loop indefinitely */
  while (TRUE) {
    mesg = (diana_message *) malloc (sizeof (diana_message));

    /* read messages from Pluto */
    n = read (s, mesg, sizeof(*mesg));

    if (n == -1) {
      perror ("DIANA: read() failed in pluto_reader() ");
      continue;
    };

    if (n != sizeof (*mesg)) {
      fprintf (stderr, "DIANA: truncated message from Pluto: got %d bytes, \
              expected %d. Ignored ", n, (int) sizeof(*mesg));
      continue;
    };

    /* compose a Qmesg and send it to InQ */
    {
      Qmesg *elem;
      elem = (Qmesg *) malloc (sizeof (Qmesg));
      elem->type = Pluto_Resp;
      elem->dat =  mesg;
      elem->next = NULL;

      enqueue (elem, InQ);
    }
  } /* end while */
} /* end pluto_reader() */
```

```
/* file: queue.c
 * This file contains operations on queues : enqueue, appendq, dequeue, flushq
 * As queues are shared amongst threads, we have to employ concurrency
 * mechanisms.  'dequeue' also implements a conditioned wait.  The wait
 * is signalled by 'enqueue' and 'appendq'.
 *
 */

#include "diana.h"

/* Thread cleanup routine: this unlocks mutex variable in case the
 * thread is cancelled when it has locked a mutex variable */
```

```
void thread_cleanup_routine (void *arg) {
  Queue *queue;

  queue = (Queue *) arg;
  pthread_mutex_unlock (&(queue->mutex));
  flushq (&queue);
} /* end thread_cleanup_routine */                                    20

/* function initq()
 * initialize a queue
 * Assumes that memory has already been allocated for the queue */
void initq (Queue *queue) {
  if (queue != NULL) {
    queue->length = 0;
    queue->first  = NULL;
    queue->last   = NULL;
    pthread_mutex_init (&(queue->mutex), NULL);                       30
    pthread_cond_init (&(queue->not_empty), NULL);
    pthread_cond_init (&(queue->empty), NULL);
  }
} /* end function initq() */

/* function enqueue()
 * add mesg to queue
 * Assume queue is not NULL */
void enqueue (Qmesg *mesg, Queue *queue) {
                                                                      40
  if (mesg != NULL) {

    pthread_mutex_lock (&(queue->mutex)) ; /* set lock */

    if (queue->last == NULL) {
      queue->first = mesg;
      queue->last = mesg;
      mesg->next = NULL;
    }
    else {                                                            50
      queue->last->next = mesg;
      mesg->next = NULL;
      queue->last = mesg;
    }

    queue->length += 1;

   /* signal to some other thread that might be waiting on the
    * queue, that the queue now contains some element */
    pthread_cond_signal (&(queue->not_empty));                        60

    pthread_mutex_unlock (&(queue->mutex)); /* release lock */
  } /* end if mesg != NULL */
} /* end enqueue() */


/* function appendq()
 * append one q1 to q2
```

```
 * secure locks only on q2 */
void appendq (Queue *q1, Queue *q2) {

  if ( (q1 != NULL) && (q1->first != NULL) ) {
    pthread_mutex_lock (&(q2->mutex));  /* set lock */

    if (q2->last == NULL) {
      q2->first = q1->first;
      q2->last  = q1->last;
    }
    else {
      q2->last->next = q1->first;
      q2->last = q1->last;
    }

    q2->length += q1->length;

    /* free memory space occupied by q1 */
    pthread_mutex_destroy(&(q1->mutex));
    pthread_cond_destroy(&(q1->not_empty));
    free(q1);

  /* signal to some other thread that may be waiting on the
   * queue, that the queue now contains some element */
    pthread_cond_signal (&(q2->not_empty));
    pthread_mutex_unlock (&(q2->mutex)); /* release lock */
  } /* end if q1 != NULL */
} /* end appendq() */


/* function dequeue()
 * remove and the front element of the queue
 * wait for an element if the queue is empty */
Qmesg * dequeue (Queue *queue) {

  Qmesg *mesg;

  /* The thread might receive a cancel message when it is waiting on
   * pthread_cond_wait().  So, a cleanup routine is necessary to
   * free the mutexes held by the thread.  Please read manpages for
   * pthread_cancel() and pthread_cleanup_push() for more information.
   */
  pthread_cleanup_push ( thread_cleanup_routine, queue );

  pthread_mutex_lock (&(queue->mutex));

  while (queue->first == NULL) /* wait while the queue is empty */ {

   /* signal to some other thread who might be waiting for
    * for this queue to get empty */
    pthread_cond_signal (&(queue->empty));

    pthread_cond_wait (&(queue->not_empty), &(queue->mutex));
  }
```

```
  mesg = queue->first;
  queue->first = mesg->next;
  mesg->next = NULL;

  if (queue->first == NULL)
    queue->last = NULL;
                                                                        130
  queue->length -= 1;
  pthread_mutex_unlock (&(queue->mutex));
  pthread_cleanup_pop (0);

  return (mesg);
} /* end dequeue() */

/* function flushq()
 * free all memory associated with the queue elements.  do not free the
 * queue and its mutex and condition fields. */                          140
void flushq (Queue **queue) {

  Queue *q;
  Qmesg *mesg, *prev;

  pthread_mutex_lock (&((*queue)->mutex));
  q = *queue;
  mesg = q->first;

  while (mesg != NULL) {                                                 150
    if (mesg->dat != NULL) {
      mesg->dat -= MAC_HDR_SIZE;
      free (mesg->dat);
    }
    prev = mesg;
    mesg = mesg->next;
    free (prev);
  }

  q->first = NULL;                                                       160
  q->last  = NULL;
  q->length = 0;

  pthread_mutex_unlock (&((*queue)->mutex));
} /* end flushq() */
```

```
/* file: queue_reader.c
 * This file contains code for queue_reader() part of diana_controller thread
 * Pseudo code:
 *      Loop indefinitely:
 *          Read a Qmesg from InQ
 *          switch (Qmesg->type) {
 *              case Update_SPdb:
 *                  call the Update_SPdb function
 *              case Pluto_Resp:
 *                  call the Pluto_proc function with Qmesg->dat as parameter   10
```

```
 *              case IP_Pkt:
 *                  call the IP_pkt_proc function with Qmesg->dat as parameter
 *          } end switch
 *      End loop
 */

#include "diana.h"

void queue_reader ( Queue *InQ, Queue *OutQ, SPdb_table *SPdb_Active,
                    SPdb_table *SPdb_Pending, char *spdb_filename ) {
  Qmesg *mesg;

  /* Loop indefinitely */
  while (TRUE) {
    /* Read a Qmesg from InQ */
    mesg = dequeue (InQ);  /* dequeue implements a conditioned wait */

    switch (mesg->type) {
    case Update_SPdb:
      /* call update_spdb function */
      update_spdb (InQ, OutQ, &SPdb_Active, &SPdb_Pending,
                   spdb_filename);
      free(mesg);
      break;

    case Pluto_Resp:
      /* call pluto_proc function */
      pluto_proc (OutQ, &SPdb_Active, &SPdb_Pending, mesg->dat);
      if (mesg != NULL) {
         free(mesg->dat);
         free(mesg);
      }
      break;

    case IP_Pkt:
      /* call ip_pkt_proc function */
      ip_pkt_proc (OutQ, SPdb_Pending, mesg);
      /* memory held by mesg is freed by ip_pkt_proc() */
      break;
    } /* end switch */
  } /* end while */
} /* end queue_reader() */
```

```
/* file: read_spdb.c
 * This file contains the code for read_spdb() and print_spdb() functions */

#include <stdio.h
#include <stdlib.h>
#include <ctype.h>       /* required for function isspace() */
#include <freeswan.h>    /* required for format conversions */
#include "diana.h"

#define MAX_LINE_SZ 256 /* the max. number of characters in a line in the SPdb
```

```
                    * file that are read while parsing the SPdb + 1 */

/* function isspacestr
 * returns 1 if the string contains only whitespace,
 *           else returns 0 */
int isspacestr(char *line) {
/* Assumes that the string 'line' is null-terminated */
  char *s;
  s = line;
                                                                        20
  /*  printf("! isspacestr: "); */
  while (*s != '\0') {
    /* printf("%c.",*s); */
    if (!isspace(*s)) {
      /* printf("\n"); */
      return(0);
    }
    s++;
  } /* end of while loop */
  /*  printf("\n!blank-line detected\n"); */                            30
  return(1);

} /* end of isspacestr() */

/* function read_spdb()
 * This function reads in the Security Policy Database from a file
 * The SPdb is read into the variable policy_table
 * The file from which it is read is the parameter FP
 * This function assumes that the parameter policy_table is NULL when
 * it is called.  It's upto the calling function to take care of freeing  40
 * memory of the old policy_list. */
int read_spdb (SPdb_table **policy_table, char *spdb_filename) {
  char line[MAX_LINE_SZ]; /* assume that a line in a file is at most
                           * MAX_LINE_SZ characters (defined in read_spdb.h) */

  Policy *policy;
  Queue *queue;
  SPdb_table *element = NULL, *prev;

  char local_client[32],                                                50
       peer_client[32],
       peer[16],
       wildcard_char[16], dummy_char[16];
  FILE *fp;

  fp = fopen ( spdb_filename, "r");

  while (fgets(line, MAX_LINE_SZ, fp) != NULL) {
    /* Note: fgets adds a '\0' after the string which it reads */
    /* Check if the line is a comment */                                60
    if (line[0] == '#')
      continue;

    /* Check if the line is a blank line */
    if (isspacestr(line))
```

```
  continue;

/* input a policy, and add it to the policy_list */
policy = (Policy *) malloc (sizeof(Policy));

/* first check if xpt_protocol is a wildcard */
sscanf(line,"%u%s%s%s", &(policy->policy_id),
       local_client, peer_client, wildcard_char);

if (wildcard_char[0] == '*') {
  policy->xpt_protocol_wildcard = TRUE;

  /* now check if src_port is a wildcard */
  sscanf(line,"%u%s%s%s%s", &(policy->policy_id),
         local_client, peer_client, dummy_char, wildcard_char);

  if (wildcard_char[0] == '*') {
    policy->src_port_wildcard = TRUE;

    /* now check if dst_port is a wildcard */
    sscanf(line,"%u%s%s%s%s%s", &(policy->policy_id), local_client,
           peer_client, dummy_char,dummy_char, wildcard_char);

    if (wildcard_char[0] == '*') {
      policy->dst_port_wildcard = TRUE;

      sscanf(line,"%u%s%s%s%s%d%s%u%d%d%d%d",
             &(policy->policy_id), local_client, peer_client, dummy_char,
             dummy_char, dummy_char, &(policy->user_id),
             peer, &(policy->peer_port),
             &(policy->encry), &(policy->auth), &(policy->tunnel),
             &(policy->discard));
    }
    else {
      policy->dst_port_wildcard = FALSE;

      sscanf(line,"%u%s%s%s%u%d%s%u%d%d%d%d",
             &(policy->policy_id), local_client, peer_client, dummy_char,
             dummy_char, &(policy->dst_port), &(policy->user_id),
             peer, &(policy->peer_port),
             &(policy->encry), &(policy->auth), &(policy->tunnel),
             &(policy->discard));
    }
  }
  else {
    policy->src_port_wildcard = FALSE;

    /* now check if dst_port is a wildcard */
    sscanf(line,"%u%s%s%s%u%s", &(policy->policy_id),
           local_client, peer_client, dummy_char,
           &(policy->src_port), wildcard_char);

    if (wildcard_char[0] == '*') {
      policy->dst_port_wildcard = TRUE;
```

```
      sscanf(line,"%u%s%s%s%u%s%d%s%u%d%d%d%d", &(policy->policy_id),
             local_client, peer_client, dummy_char,
             &(policy->src_port), dummy_char, &(policy->user_id),
             peer, &(policy->peer_port),
             &(policy->encry), &(policy->auth), &(policy->tunnel),
             &(policy->discard));
    }
    else {
      policy->dst_port_wildcard = FALSE;

      sscanf(line,"%u%s%s%s%u%u%d%s%u%d%d%d%d", &(policy->policy_id),
             local_client, peer_client, dummy_char,
             &(policy->src_port), &(policy->dst_port), &(policy->user_id),
             peer, &(policy->peer_port),
             &(policy->encry), &(policy->auth), &(policy->tunnel),
             &(policy->discard));
    }
  }
}
else {
  policy->xpt_protocol_wildcard = FALSE;

  /* now check if src_port is a wildcard */
  sscanf(line,"%u%s%s%d%s", &(policy->policy_id),
         local_client, peer_client,
         &(policy->xpt_protocol), wildcard_char);

  if (wildcard_char[0] == '*') {
    policy->src_port_wildcard = TRUE;

    /* now check if dst_port is a wildcard */
    sscanf(line,"%u%s%s%d%s%s", &(policy->policy_id),
           local_client, peer_client,
           &(policy->xpt_protocol), dummy_char, wildcard_char);

    if (wildcard_char[0] == '*') {
      policy->dst_port_wildcard = TRUE;

      sscanf(line,"%u%s%s%d%s%s%d%s%u%d%d%d%d", &(policy->policy_id),
             local_client, peer_client, &(policy->xpt_protocol),
             dummy_char, dummy_char, &(policy->user_id),
             peer, &(policy->peer_port),
             &(policy->encry), &(policy->auth), &(policy->tunnel),
             &(policy->discard));
    }
    else {
      policy->dst_port_wildcard = FALSE;

      sscanf(line,"%u%s%s%d%s%u%d%s%u%d%d%d%d", &(policy->policy_id),
             local_client, peer_client, &(policy->xpt_protocol),
             dummy_char, &(policy->dst_port), &(policy->user_id),
             peer, &(policy->peer_port),
             &(policy->encry), &(policy->auth), &(policy->tunnel),
             &(policy->discard));
    }
```

```
  }
  else {
    policy->src_port_wildcard = FALSE;

    /* now check if dst_port is a wildcard */                                    180
    sscanf(line,"%u%s%s%d%u%s", &(policy->policy_id),
           local_client, peer_client,
           &(policy->xpt_protocol), &(policy->src_port), wildcard_char);

    if (wildcard_char[0] == '*') {
      policy->dst_port_wildcard = TRUE;

      sscanf(line,"%u%s%s%d%u%s%d%s%u%d%d%d%d", &(policy->policy_id),
             local_client, peer_client, &(policy->xpt_protocol),
             &(policy->src_port), dummy_char, &(policy->user_id),                190
             peer, &(policy->peer_port),
             &(policy->encry), &(policy->auth), &(policy->tunnel),
             &(policy->discard));
    }
    else {
      policy->dst_port_wildcard = FALSE;

      sscanf(line,"u%s%s%d%u%u%d%s%u%d%d%d%d", &(policy->policy_id),
             local_client, peer_client, &(policy->xpt_protocol),
             &(policy->src_port), &(policy->dst_port), &(policy->user_id),      200
             peer, &(policy->peer_port),
             &(policy->encry), &(policy->auth), &(policy->tunnel),
             &(policy->discard));
    }
  }
}

/* Note:atosubnet() and atoaddr() are declared in freeswan.h */

/*    printf("!local_client = %s\n", local_client); */                          210
atosubnet(local_client,0,&(policy->local_client_net),
          &(policy->local_client_mask));

/*    printf("!peer_client = %s\n", peer_client); */
atosubnet(peer_client,0,&(policy->peer_client_net),
          &(policy->peer_client_mask));

/* printf("!peer = %s\n", peer); */
atoaddr(peer,0,&(policy->peer_addr));
                                                                               220
policy->status = INACTIVE;
policy->mark   = FALSE;
policy->try_count = 0;

prev = element;
element = (SPdb_table *) malloc (sizeof (SPdb_table));
queue   = (Queue *) malloc (sizeof (Queue));

queue->first = NULL;
queue->last  = NULL;                                                           230
```

```
    queue->length = 0;
    pthread_mutex_init (&(queue->mutex), NULL);
    pthread_cond_init (&(queue->not_empty), NULL);
    pthread_cond_init (&(queue->empty), NULL);

    element->p = policy;
    element->q = queue;
    element->next = NULL;

    if (prev == NULL)                                                    240
      *policy_table = element; /* hook the policy_table to the first element */
    else
      prev->next = element;

  } /* end of while() */

  fclose(fp);
  return(1);
} /* end of read_spdb() */
                                                                         250

/* A test function for printing policy_list */
int print_spdb (SPdb_table *policy_list) {
  SPdb_table *elem;

  for(elem = policy_list; elem != NULL; elem = elem->next) {
    fprintf(stdout, "%d ",elem->p->policy_id);
    fprintf(stdout, "%#x/",elem->p->local_client_net.s_addr);
    fprintf(stdout, "%#x ",elem->p->local_client_mask.s_addr);
    fprintf(stdout, "%#x/",elem->p->peer_client_net.s_addr);             260
    fprintf(stdout, "%#x ",elem->p->peer_client_mask.s_addr);
    if (elem->p->xpt_protocol_wildcard == TRUE)
        fprintf(stdout, "TRUE ");
    else
        fprintf(stdout, "FALSE ");
    fprintf(stdout, "%d ",elem->p->xpt_protocol);
    if (elem->p->src_port_wildcard == TRUE)
        fprintf(stdout, "TRUE ");
    else
        fprintf(stdout, "FALSE ");                                       270
    fprintf(stdout, "%d ",elem->p->src_port);
    if (elem->p->dst_port_wildcard == TRUE)
        fprintf(stdout, "TRUE ");
    else
        fprintf(stdout, "FALSE ");
    fprintf(stdout, "%d ",elem->p->dst_port);
    fprintf(stdout, "%d ",elem->p->user_id);
    fprintf(stdout, "%#x ",elem->p->peer_addr.s_addr);
    fprintf(stdout, "%d ",elem->p->peer_port);
    fprintf(stdout, "%d ",elem->p->encry);                               280
    fprintf(stdout, "%d ",elem->p->auth);
    fprintf(stdout, "%d ",elem->p->tunnel);
    fprintf(stdout, "%d ",elem->p->discard);
    if (elem->p->status == ACTIVE)
      fprintf(stdout, "A ");
```

```
    else if (elem->p->status == INACTIVE)
      fprintf(stdout, "I ");
    else if (elem->p->status == PENDING)
      fprintf(stdout, "P ");
    /* fprintf(stdout, "%d",elem->p->mark); */                    290
    fprintf(stdout, "\n");
  } /* end of for loop */
  return(1);
} /* end of print_spdb() */
```

```
/* file: update_spdb.c
 * This file contains the code for update_spdb() and free_spdb() functions
 */

#include "diana.h"

void update_spdb (Queue *InQ, Queue *OutQ, SPdb_table **SPdb_Active,
                  SPdb_table **SPdb_Pending, char *spdb_filename) {

  SPdb_table *SPdb_buf, *elem, *prev;                             10

  /* Read SPdb-file into SPdb_buf */
  SPdb_buf = NULL;
  read_spdb (&SPdb_buf, spdb_filename);

  /* Compare all entries of SPdb_buf with all entries of SPdb_Active table.
   * if some entry matches, delete it from SPdb_buf and mark it in SPdb_Active
   */
  compare (&SPdb_buf, *SPdb_Active);
                                                                  20
  /* For each unmarked entry in SPdb_Active do: */
  prev = NULL;
  elem = *SPdb_Active;
  while (elem != NULL) {

    if (elem->p->mark == FALSE) /* unmarked */ {

      /* send a 'Teardown' message to whack function */
      whack (Teardown, elem->p);
                                                                  30
      /* delete that element from SPdb_Active table */
      delete_entry (&prev, &elem, SPdb_Active);
    }
    else {
      prev = elem;
      elem = elem->next;
    }
  } /* end while */

  /* Unmark all entries in SPdb_Active */                         40
  unmark_entries (*SPdb_Active);

  /* Compare all entries of SPdb_buf with all entries of SPdb_Pending table.
```

```
 * if some entry matches, delete it from SPdb_buf and mark it in SPdb_Pending
 */
compare (&SPdb_buf, *SPdb_Pending);

/* For each unmarked entry in SPdb_Pending do: */
prev = NULL;
elem = *SPdb_Pending;                                                    50

while (elem != NULL) {
  if (elem->p->mark == FALSE) /* unmarked */ {
    if (elem->p->status == PENDING) {
      /* send a 'Teardown' message to whack function */
      whack (Teardown, elem->p);

      /* append the corresponding queue of packets to OutQ so that
       * the packets are sent back to kernel
       */                                                               60
      appendq (elem->q, OutQ);
      elem->q = NULL;
    }

    /* Delete the entry from SPdb Pending table */
    delete_entry (&prev, &elem, SPdb_Pending);
  }
  else {
    prev = elem;
    elem = elem->next;                                                   70
  } /* end if */
} /* end while */

/* Unmark all entries from SPdb_Pending table */
unmark_entries (*SPdb_Pending);

/* If there are some entries left in SPdb_buf, add corresponding entries
 * in SPdb Pending table (default status == INACTIVE) */

/* first go to the end of SPdb_buf */                                    80
elem = SPdb_buf;

if (elem != NULL) {
  while (elem->next != NULL)
    elem = elem->next;

  /* Now elem points to the last element of SPdb_buf */
  elem->next = *SPdb_Pending;
  *SPdb_Pending = SPdb_buf;
}                                                                        90
/* elem == NULL means that SPdb_buf is NULL, so do nothing */

/* if there are no entries in SPdb_Pending table, then close interception
 * mechanism */
if (*SPdb_Pending == NULL)
  close_taps();
else  /* if interception mechanism is already closed, open it */
  open_taps(InQ);
```

```
} /* end update_spdb() */

void delete_entry (SPdb_table **prev, SPdb_table **elem, SPdb_table **table) {

  if (*prev == NULL) {
    *table = (*elem)->next;
    free ((*elem)->p);
    free (*elem);
    *elem = *table;
  }
  else {
    (*prev)->next = (*elem)->next;
    free ((*elem)->p);
    free (*elem);
    *elem = (*prev)->next;
  }

} /* end delete_entry() */

void unmark_entries (SPdb_table *table) {
  SPdb_table *elem;

  for (elem = table; elem != NULL; elem = elem->next)
    elem->p->mark = FALSE;
}

/* function compare()
 * Compare all entries of SPdb_buf with all entries of table.
 * if some entry matches, delete it from SPdb_buf and mark it in table.
 * does check for duplicate entries. */

void compare (SPdb_table **SPdb_buf, SPdb_table *table) {
  SPdb_table *buf_prev, *buf_curr, *table_curr;

  buf_prev = NULL;
  buf_curr = *SPdb_buf;

  while (buf_curr != NULL) {

    boolean match = FALSE;
    for (table_curr = table; table_curr != NULL;
         table_curr = table_curr->next) {

      Policy *b, *t;
      b = buf_curr->p;
      t = table_curr->p;

      /* if the two policies b and t are equal */
      if ( (b->local_client_net.s_addr == t->local_client_net.s_addr)
           && (b->local_client_mask.s_addr == t->local_client_mask.s_addr)
           && (b->peer_client_net.s_addr == t->peer_client_net.s_addr)
           && (b->peer_client_mask.s_addr == t->peer_client_mask.s_addr)
           && (b->xpt_protocol == t->xpt_protocol)
           && (b->src_port == t->src_port)
```

```
          && (b->dst_port == t->dst_port)
          /*      && (b->user_id  == t->user_id) */
          && (b->peer_addr.s_addr == t->peer_addr.s_addr)
          && (b->peer_port == t->peer_port)
          && (b->encry == t->encry)
          && (b->auth == t->auth)
          && (b->tunnel == t->tunnel)                                    160
          && (b->discard == t->discard)
          ) {

        /* mark entry in table */
        t->mark = TRUE;
        match   = TRUE;
      } /* end if */
    } /* end for */

    if (match == TRUE) {                                                 170
      delete_entry (&buf_prev, &buf_curr, SPdb_buf);
      match = FALSE;
    }
    else {
      buf_prev = buf_curr;
      buf_curr = buf_curr->next;
    }
  } /* end while */
} /* end compare() */
```

```
/* file: whack.c
 * This file contains the 'whack' function, which sends a message
 * to Pluto.  This initiates ISAKMP exchange in Pluto for the
 * establishment/teardown of a Security Association.
 *
 * Pseudo-code:
 *      Compose a 'whack-message' from the policy in the parameter, depending
 *              on whether a 'Teardown' or 'Establish_SA' has been requested.
 *      Send the 'whack-message' to Pluto on the specified port
 *                                                                       10
 */

#include <ctype.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <freeswan.h>
#include "diana.h"

void whack (int request, Policy *policy) {                               20
  struct whack_message mesg;
  unsigned long pluto_port = IKE_UDP_PORT + 1;
  /* IKE_UDP_PORT is defined in constants.h */

  /* first compose the 'whack_message' */
  memset (&mesg, '\0', sizeof(mesg));
```

```
  mesg.magic = WHACK_MAGIC;
  mesg.peer_port = htons (policy->peer_port);

  if (request == Establish_SA)                                          30
    mesg.whack_initiate = TRUE;          /* establish SA */
  else
    mesg.whack_initiate = FALSE;         /* teardown SA */

  mesg.peer = policy->peer_addr;
  mesg.local_client_net = policy->local_client_net;
  mesg.local_client_mask = policy->local_client_mask;
  mesg.peer_client_net = policy->peer_client_net;
  mesg.peer_client_mask = policy->peer_client_mask;
                                                                        40
  mesg.explicit_local_client = TRUE;
  mesg.explicit_peer_client  = TRUE;
  mesg.whack_shutdown = FALSE;

  if (policy->encry == TRUE)
    mesg.goal |= GOAL_ENCRYPT;

  if (policy->auth == TRUE)
    mesg.goal |= GOAL_AUTHENTICATE;
                                                                        50
  if (policy->tunnel == TRUE)
    mesg.goal |= GOAL_TUNNEL;

  mesg.debugging = DBG_NONE;
  mesg.whack_options = FALSE;

  /* Now send the whack message to Pluto on udp port: IKE_UDP_PORT + 1
   * This code taken from whack.c file from freeswan-0.90/pluto/whack.c */
  {
    int sock = socket(PF_INET, SOCK_DGRAM, 0);                          60
    struct sockaddr_in sin;

    if (sock == -1)
        perror("whack: socket() failed");

    memset(&sin, '\0', sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
    sin.sin_port = htons(pluto_port);
                                                                        70
    if (sendto(sock, &mesg, sizeof(mesg),
               0, (struct sockaddr *)&sin, sizeof(sin)) == -1)
        perror("sendto() failed");
    close (sock);
  } /* end send message to Pluto */
} /* end whack() */
```

Appendix B

User Manual

This Appendix contains the instructions for installing and using DIANA.

B.1 Installation

DIANA requires that the FreeS/WAN (version 0.90) implementation of IPSec be installed in an appropriate directory. Let this this directory be called `FREESWANDIR`. For example, if `freeswan-0.90` has been installed in `/usr/src`, then,

```
FREESWANDIR = /usr/src/freeswan-0.90
```

Install all files of DIANA given in Appendix A in a sub-directory of `FREESWANDIR` called `diana`. Let this directory be called `DIANADIR`. Thus,

```
DIANADIR = FREESWANDIR/diana
```

Compile DIANA by doing a '`make`' from within `DIANADIR`. This will create an executable file called '`dianad`' in `DIANADIR`.

B.1.1 Modification of Pluto

Pluto needs to be modified so that it replies back to DIANA. This involves modifying the files '`kernel.c`', '`kernel.h`' and '`Makefile`' in the '`FREESWANDIR/pluto/`' directory. The steps involved are as follows:

- **kernel.c**:
  1. Replace the line

     ```
     #include <constants.h>
     ```

     by the lines

     ```
     #ifdef DIANA
     #include "../diana/diana.h"
     #else
     #include "constants.h"
     #endif
     ```

2. Insert the function '`reply_diana`' as given in the `DIANADIR/install_ipsec_sa.c` file, along with the '`#ifdef DIANA`' statements.
3. Replace the function '`install_ipsec_sa`' by the function '`install_ipsec_sa`' as given in '`DIANADIR/install_ipsec_sa.c`'.

- **kernel.h**: Add the following lines at the end of the file:

```
#ifdef DIANA
extern void reply_diana (bool , struct state *, unsigned int);
#endif
```

- **Makefile**: Add '`-DDIANA`' to the '`CFLAGS`' variable.

Re-compile and re-install Pluto by using `make` from within `FREESWANDIR/pluto/`. For more information on compiling and installing Pluto, please refer to the `README` file in the '`FREESWANDIR/pluto/`' directory.

B.2 User Guide

Running DIANA for traffic-driven SA establishment is simple. Just follow the following steps:

1. Install the `freeswan-0.90` IPSec kernels on machines between whom an SA establishment is desired. Please follow the instructions given in the `freeswan-0.90` documentation for installation.
2. Boot these machines with the IPSec kernels
3. Configure IPSec devices and attach them to the desired interfaces on these machines. Please read the `freeswan-0.90` documentation for information on how to configure and attach IPSec devices.
4. Start the Pluto daemon. Please read the Pluto '`man` page' for more details on how to run Pluto.
5. Make sure that appropriate routing paths exist between all the machines of interest.
6. Edit the `spdb.txt` file in `DIANADIR` and set appropriate policies in it.
7. Start DIANA by executing '`./dianad`' at the command prompt from within `DIANADIR`.

This will start the traffic-driven SA-establishment process. When *matching* traffic starts to flow, DIANA will establish appropriate SAs. Currently, Pluto does not support SA-teardown, so the SAs will not be removed automatically (unless they time-out). The user will have to manually remove the SAs from SAdbs. Please read the `freeswan-0.90` documentation on how to remove SAs from the SAdb. DIANA can be stopped by sending it a `SIGINT`, a `SIGTERM` or a `SIGKILL` signal.