# Building an Application-aware IPsec Policy System

Heng Yin Haining Wang
*Department of Computer Science*
*The College of William and Mary*
*Williamsburg, VA 23187*
{hyin,hnw}@cs.wm.edu

## Abstract

As a security mechanism at the network-layer, the IP security protocol (IPsec) has been available for years, but its usage is limited to Virtual Private Networks (VPNs). The end-to-end security services provided by IPsec have not been widely used. To bring the IPsec services into wide usage, a standard IPsec API is a potential solution. However, the realization of a user-friendly IPsec API involves many modifications on the current IPsec and Internet Key Exchange (IKE) implementations. An alternative approach is to configure application-specific IPsec policies, but the current IPsec policy system lacks the knowledge of the context of applications running at upper layers, making it infeasible to configure application-specific policies in practice.

In this paper, we propose an application-aware IPsec policy system on the existing IPsec/IKE infrastructure, in which a socket monitor running in the application context reports the socket activities to the application policy engine. In turn, the engine translates the application policies into the underlying security policies, and then writes them into the IPsec Security Policy Database (SPD) via the existing IPsec policy management interface. We implement a prototype in Linux (Kernel 2.6) and evaluate it in our testbed. The experimental results show that the overhead of policy translation is insignificant, and the overall system performance of the enhanced IPsec is comparable to those of security mechanisms at upper layers. Configured with the application-aware IPsec policies, both secured applications at upper layers and legacy applications can transparently obtain IP security enhancements.

## 1 Introduction

Network-layer security protection is essential to Internet communications. No matter how secure the upper-layer protocols are, adversaries can exploit the vulnerability of the network-layer, such as IP spoofing [25] and IP fragmentation attacks [16], to sabotage end-to-end communications. The IP security (IPsec) protocol [17, 18, 19, 27] is a suite of protocols that secure data communications on the Internet at the network-layer. IPsec provides packet-level source authentication, data confidentiality and integrity, and supports perfect forward security. There are two major protocols in the IPsec protocol suite: the Authentication Header (AH) protocol and the Encapsulation Security Payload (ESP) protocol. The AH protocol provides source authentication and data integrity, while the ESP protocol provides data confidentiality and authentication. Internet Key Exchange (IKE) [13, 21] is the default key agreement protocol for the establishment of IPsec security associations (SAs), doing mutual authentication and choosing cryptographic keys.

However, while IPsec has been available for years, its usage is limited to the deployment of Virtual Private Networks (VPNs). Currently, the IPsec policy is not aware of the specific security requirements of Internet applications. It is static and coarse-grained, providing all or nothing security protection to Internet applications. Thus, in comparison with the success of security protocols and techniques deployed at the transport and application layers such as SSL/TLS and PGP, IPsec has been rarely used to provide end-to-end security protection for Internet applications. The major obstacles to the wide usage of IPsec beyond VPNs (i.e., providing end-to-end security services to Internet applications) are listed as follows.

- IPsec only provides rudimentary policy management support [8, 14], and IPsec lacks the knowledge of application context. Thus, the policy selector can only be the tuple of source/destination addresses, port numbers, and transport protocol type. The trust relationship and security policy between two participants must be set up in advance. Due to dynamic port numbers and unpredicted destination

IP addresses, configuring fine-grained application-specific policies beforehand is infeasible in practice. Note that many applications, including passive FTP data connections, DCOM/CORBA-based applications, and RTP-based streaming applications, even negotiate source/destination ports at runtime.

- There is no standard IPsec API, and implementing a user-friendly IPsec API requires many modifications on the current IPsec and IKE implementations. For instance, the current two-phase IKE protocol only performs host-oriented authentication, whereas user-oriented authentication is one basic requirement of a secured application (see more discussion in Section 2). Moreover, the wide usage of upper-layer security mechanisms such as SSL/TLS and SSH greatly dampens the demand for IPsec API.

- IPsec needs Public Key Infrastructure (PKI) to perform identity authentication in the public network environment, whereas PKI itself is not widely deployed yet. The lack of scalable authentication mechanisms is another major impediment to the wide use of IPsec.

The good news about IPsec is: (1) IPsec and IKE have been implemented on nearly all modern operating systems, and have been applied to the VPNs scenario maturely now; and (2) while no standard IPsec API exists, each implementation does have its own policy management interface.

In this paper, we explore an approach other than the IPsec API to facilitating the wide usage of IPsec. We attempt to provide application-aware IPsec service by introducing application context into IPsec policy model, while leaving the IPsec and IKE implementations intact. We propose an *application-aware IPsec policy system* as middleware to provide Internet applications with network-layer security protection. In order to bring application context into the IPsec policy model, a *socket monitor* detects the socket activities of applications and reports them to the *application policy engine*. Then, the application policy engine automatically generates the fine-grained IPsec policy in accordance with the application policy, and writes them into the IPsec Security Policy Database (SPD) via the existing IPsec policy management interface. [1] To ease policy configuration, we also propose an application specification language. Being simple, uniform, and extendable, the proposed specification language is essential to configure and distribute application-specific policies easily, and hence, reduces the management burden.

In addition to the feasibility issue, compared with a *potential* standard IPsec API, the application-aware policy system has the following advantages: (1) without any modification, Internet applications can transparently obtain end-to-end security protection at the network-layer by simply configuring application-specific policies; and (2) the application-aware policy system is built on the existing IPsec policy configuration interface, and no modifications on the current IPsec and IKE implementations are needed. Thus, it is much easier for IPsec vendors to support and deploy it.

We implement a prototype of the proposed IPsec policy system in Linux (Kernel 2.6), and evaluate the efficacy of our prototype in the testbed. The experimental results show that the overhead of policy translation is insignificant and overall system performance of the enhanced IPsec is comparable to those of security mechanisms at upper layers. More importantly, our experiments demonstrate that the application-aware IPsec policy system provides network-layer security enhancements (e.g., packet-level integrity protection) for secured applications and provides a variety of network-layer security services for legacy applications.

Note that IPsec is not a panacea to resolve all network-layer security problems. For example, IPsec cannot thwart the bandwidth-exhaustion Denial of Service (DoS) attacks [31] and IKE itself is vulnerable to fragmentation flooding attacks [16]. Seeking solutions to these problems is outside the scope of this paper. Although IPsec is able to protect broadcast and multicast traffic, the current key management protocols for IPsec can only work in one-to-one mode for now. Therefore, our proposed scheme applies to unicast communication only. The scenarios of broadcast and multicast are also beyond the scope of our research.

The remainder of this paper is organized as follows. Section 2 presents the background of this research and related work. Section 3 details the application-aware IPsec policy system. We describe the implementation of our prototype in Linux (Kernel 2.6) in Section 4. In Section 5, we conduct a series of experiments on our testbed to evaluate the proposed scheme. Finally, the paper concludes with Section 6.

## 2 Background and Related Work

Since IPsec is a layer-3 security protocol, it must be implemented in the kernel space. There are two auxiliary databases that IPsec consults with: Security Policy Database (SPD) and Security Association Database (SAD). Generally speaking, a security policy determines what kinds of services are to be offered to an IP flow, and a security association (SA) specifies how to process it.

Although the SAs can be manually created, in most cases they are created automatically by the key management program. The IKE [13] is the current IETF stan-

dard for key establishment and SA parameter negotiation. IKEv2 [15] has been proposed to replace the original IKE protocol for simplicity and strong DoS protection. Both are two-phase protocols. During the first phase, the two key management daemons authenticate each other and establish a secure channel between them (i.e IKE SA). Multiple Phase II SAs (i.e. IPsec SAs) can be negotiated through this secure channel, to amortize the cost of the Phase I negotiation.

Currently IPsec/IKE has been implemented on major modern operating systems and network devices, but has not been widely used yet. One main reason is no standard IPsec API available for end-users [14]. The requirements for an IPsec API have been detailed in [29]. One of the requirements is to allow an application to authenticate a peer's identity, and then, make access control decisions. Nonetheless, it is difficult for the key management daemon (i.e., IKE) to expose the functionality of authentication and policy negotiation to applications.

At present IKE is only used for host-oriented authentication, but user-oriented authentication is usually needed by secured applications. To enable IKE with user-level authentication, Litvin et al. proposed a hybrid authentication mode for IKE [20]. This scheme enables a two-way authentication between a remote user and an IPsec device through challenge-response techniques. The IPsec device is authenticated via standard public-key techniques. At the end of phase I, the remote user is authenticated via an X-Auth exchange [10]. However, if we apply this authentication mode to the client/server scenarios, it cannot work with a server having multiple services. Since each service has its own authentication and access control policies, during phase I negotiation, the server-side IKE cannot know which service the remote user wants to access.

The Just Fast Keying protocol (JFK) [7] may overcome the problems mentioned above. Seeking simplicity and efficiency, it rejects the notion of two-phase negotiation. Thus, JFK may easily support user-oriented authentication. However, since JFK has not been deployed yet, whether JFK will replace IKE and be widely used is still unclear. Also, there are obstacles for integrating JFK into IPsec. For instance, JFK disallows extensions, thus it would be difficult to negotiate UDP encapsulation for NAT.

Although we still have no standard IPsec API, there are some preliminary implementations of IPsec APIs. McDonald [22] designed and implemented a simple IPsec API for BSD sockets. Applications can configure per-socket policy using *setsockopt*, which is extended to support IPsec policy options. This mechanism is available in BSD-family Unix systems. In [32], Wu et al. designed and implemented IPsec/PHIL interface to enable the controllability over which set of IPsec tunnels will be used to send a particular outgoing packet. Microsoft has integrated IPsec in its Windows 2000 and Windows XP products, but no official IPsec API has been published yet. A home-brewed IPsec API library [3] for Windows 2000 and above versions has been implemented by manipulating the local policy repository. Because of their very limited functionality, these IPsec APIs are not used in the real world.

Some research has been done to cope with the rudimentary policy support for IPsec. The IETF IP Security Policy Working Group [1] has been established for many years. Condell et al. [12] defined the Security Policy Specification Language (SPSL), a language designed to express IPsec security policies and IKE policies. While SPSL offers considerable flexibility in specifying IPsec security policies, it is only a low-level language. In contrast, the Application Policy Specification Language we propose is a high-level policy specification language for individual applications. Therefore, it is simpler, more concise, and more convenient to use. Blaze et al. [11] proposed an efficient policy management scheme for IPsec, based on the principles of trust management. It provides a simple language for describing and implementing policies, trust relationships, and credentials for IPsec. However, without the knowledge of application context, the scheme itself does not address the problem of protecting individual Internet applications directly.

FreeS/WAN proposed an extension to IPsec, which is called *opportunistic encryption* [26]. By putting the authentication information in the DNS (domain name service), any two FreeS/WAN gateways are able to encrypt their traffic without prior contact and configuration. This technique may move us toward a more secure Internet, allowing users to create an environment where message privacy is the default. However, encrypting all traffic would waste precious encryption bandwidth. In fact, a large portion of network traffic does not need strong confidentiality protection. Moreover, security gateways that originally serve their own organizations cannot get any benefit from doing opportunistic encryption, and therefore they lack incentive to support it.

Miltchev et al. [24] investigated the performance of IPsec using micro- and macro-benchmarks, and compared it against other secure data transfer mechanisms, such as SSL, *scp*, and *sftp*. Their experiment results have shown that IPsec outperforms all other popular schemes that try to accomplish secure network communications, because of faster processing and no handshake for each connection.

An alternative approach to transparently securing legacy applications is to tunnel their communications via TLS/SSL or SSH [6]. However, compared with these tunneling techniques, our application-aware IPsec policy system has the following advantages: (1) we can protect

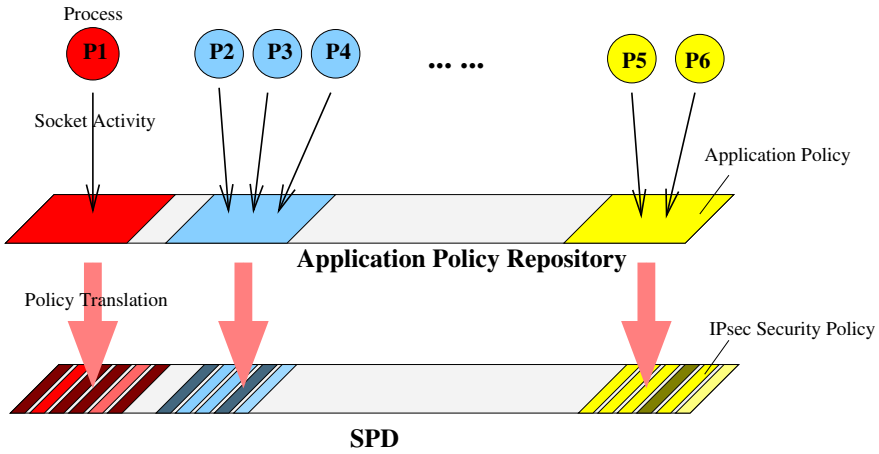



Figure 1: The Overview of the Application-aware IPsec Policy System.

the applications that negotiate port numbers at runtime, while the tunneled applications must have well-known service ports; (2) we can specify flexible security requirements, while tunneled applications have no options; (3) IPsec has inherent security and performance superiority over the tunneled security mechanisms.

## 3 System Design

Our research attempts to build middleware, called an application-aware IPsec policy system, between Internet applications and the IPsec/IKE implementations underneath. Through the proposed policy system, Internet applications can specify their requirements of network-layer security protection, and then rely on the underlying IPsec/IKE functionalities implemented at each end-host to obtain the desired network-layer security protection transparently. Note that the proposed application-aware IPsec policy model is preliminary but not a complete one, and it paves the path to further research work.

As shown in Figure 1, we have a high-level security policy abstraction in the context of applications: called *application security policy*. To have a high-level application security policy enforced, it must be translated into a low-level IPsec security policy. The policy translation is triggered by the the socket activities of a process, which are monitored and bound to an application security policy. Then, we create a fine-grained IPsec security policy, write it into SPD, and provide the specific protection for the communication over the particular socket at runtime. Note that IPsec manipulates each inbound or outbound IP packet by consulting with SPD and SAD.

The key management program (e.g. IKE, IKEv2) on each host is responsible to authenticate the hosts to each other, and create IPsec SAs automatically. However, we have not addressed the user-oriented authentication problem in this paper, which involves non-trivial modifications in IKE. Therefore, the identification authentication has to rely on Public Key Infrastructure (PKI). If PKI is not available, an alternative approach is to allow the mutually suspicious hosts to communicate with each other, as long as there is a security mechanism enforced in higher layer that performs secure identity authentication (e.g. TLS/SSL, SSH). We leave the support of the user-oriented authentication as our future work.

In the rest of this section, we present the architecture of the application-aware policy system, the necessary support from the current IPsec implementations, the application policy specification language, and runtime policy translation.

### 3.1 Architecture

The generic architecture of an application-aware policy system at an end-host is shown in Figure 2. In the network protocol stack, the *socket monitor* is installed atop the socket interface. Thus the socket monitor can observe the Internet socket activities within the context of application processes. In other words, it knows which application is operating on what kind of socket interface. The detailed information about the socket activities of the application, including the specific parameters, is forwarded to the *application policy engine*, which is a privileged program and can manipulate the local SPD and SAD. With that knowledge, the security policy engine fetches the corresponding application policy from the *application policy repository*, and prepares appropriate fine-grained IPsec security policies for the specific socket, and then writes them into the local SPD via the existing IPsec policy management interface. The socket monitor is a crucial component in our design, because it disseminates the application context into the IPsec policy model, making it possible to configure policy for each in-

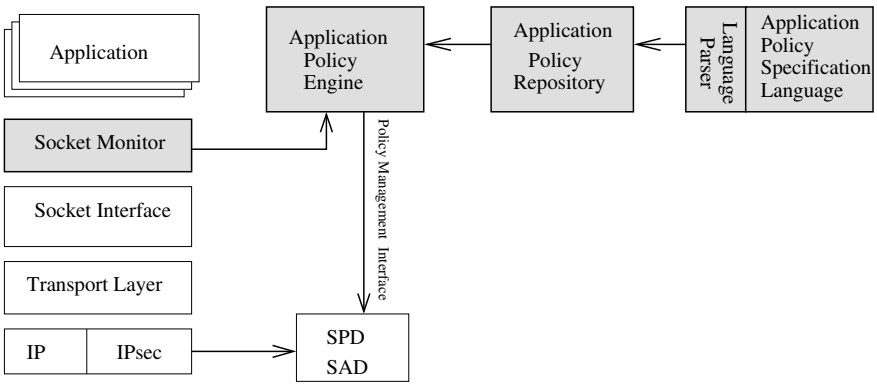



Figure 2: The Architecture of Application-aware IPsec Policy System.

dividual application.

To facilitate policy management, we further design an application policy specification language. The administrator specifies application policies using the proposed high-level specification language. A language parser converts the human-readable language into the internal data structures and stores them into the application policy repository. In comparison with the underlying IPsec security policy, the application policy is a higher-level application-oriented policy, which specifies the security requirements of the application. This high-level policy specification language is easy to write, and therefore alleviates the administrator's burden of policy configuration. Moreover, while there are potential conflicts and faults in the system-wide IPsec security policies, the policy translation effectively confine the potential policy conflicts and faults in the domain of an application — an error in one application policy can only affect that application. We describe the application policy specification language in detail in Section 3.3.

Triggered by the events of socket invocations from applications, the application policy engine executes the creation and deletion of the fine-grained IPsec security policies. For example, the event of an application calling *connect* would trigger the application policy engine to generate the necessary IPsec security policies for it, and the subsequent calling of *closesocket* would cause the deletion of these policies. We detail the socket-event-driven policy translation process in Section 3.4.

## 3.2 Support from Current IPsec Implementations

As an IPsec enhancement, our policy system cannot work independently without the appropriate support from the current IPsec implementations. In general, the support we need lies in three areas: the underlying policy management interface, the process of the IP packet that triggers the policy negotiation, and the setup of security level of a security policy. We describe the availability of the support in the current IPsec implementations and our design choices as follows.

- Each IPsec implementation has its own policy management interface, whether it is published or not. Since the KAME IPsec implementation [4] for BSD Unix families and the native IPsec implementation for Linux (Kernel 2.6), provide PF_KEY extensions for IPsec policy management [5, 23], we build our policy system on basis of the KAME-like IPsec implementations. The proposed application policy engine utilizes the underlying policy management interface to create and delete security policies. However, this does not imply that our policy system is restricted to the KAME-like IPsec implementations only. The proposed scheme can be integrated with other IPsec implementations, as long as they provide the basic policy management interfaces.

- When an outgoing IP packet triggers a policy negotiation, IPsec has to hold the IP packet until the completion of the negotiation, rather than drop it and return an error. This is because the upper-layer protocols usually cannot cope with this kind of error well. Thus, IPsec needs to handle the negotiation-triggering packets properly. According to our study, most of the current IPsec implementations are capable of providing this kind of support.

- Finally, IPsec should allow users to specify the security level of a security policy. Considering that a server sometimes needs to communicate with both IPsec clients and regular clients, we should have at least two levels: *mandatory* and *optional*. *Mandatory* indicates that the flows of this policy must be protected. *Optional* indicates that the establishment of an IPsec channel is optional for its flows. Once the IPsec channel has been established as mandatory or optional, the IP flow belonging to this chan-

nel must go through it. Many IPsec implementations do support optional security. For instance, the KAME IPsec implementation has three levels: *use*, *require*, and *unique*, and the native IPsec implementation of Windows 2000/XP also allows users to specify an optional policy.

## 3.3 Application Policy Specification Language

In order to facilitate the configuration of application policies, we define a simple application policy specification language. Figure 3 illustrates a sample of application policy described in the proposed language, and more realistic examples are shown in Section 5. The keyword *application* indicates the start of one application policy. Following the same line is the application name (or its path) or a list of application names that have the same policy setting. The body of the application policy consists of two classes of settings: one is network setting, and the other is protection setting.

```
application app1, app2
{
  network 192.168.0.0/24 trusted;
  network 192.168.2.0/24 untrusted;
  network www.abc.com    protected P1;
  network 0.0.0.0/0      protected P2;
  protection P1 {
    localport=123 remoteport=any
                  encryption mandatory;
    localport=any remoteport=any
                  authentication mandatory;
  }
  protection P2 {
    localport=any remoteport=any
                  authentication optional;
  }
}
```

Figure 3: A Sample of Application Policy

### 3.3.1 Network setting

The network setting classifies the IP address space into three categories: trusted, untrusted, and protected networks. In trusted networks (e.g., the local area network on which the end-host resides), their machines are trustworthy, and hence the IP traffic from and to the trusted networks will bypass IPsec processing. By contrast, the untrusted networks define a blacklist of machines, and the IP traffic from and to those machines will be discarded by IPsec. The machines of the protected networks are not trustworthy before the successful authentication of their identifications. After the approval of their identifications, the appropriate IPsec protocols (ESP or AH) are applied to their IP traffic to prevent eavesdropping and tampering.

The Backus Naur Form (BNF) of a network setting is given below:

```
network-def -> "network" address-range
               network-type ";"
address-range -> ipaddress "-" ipaddress
               | ipaddress "/" integer
               | ipaddress
network-type -> "trusted" | "untrusted"
               | "protected" string
```

The keyword *"network"* indicates that the following parts of this line is a network definition. The *address-range* can be a single IP address (or hostname) or a range of IP addresses specified by an IP address prefix and mask. The keywords *"trusted"*, *"untrusted"*, and *"protected"* indicate the specific category that the network belongs to. If the network is a protected, the protection-setting given below determines how communications are protected.

### 3.3.2 Protection setting

The protection setting defines how the IP traffic of an application is protected. Since an application usually uses unique (well-known) port numbers to distinguish different classes of IP flows, port-number-based protection setting is effective to appropriately secure different classes of IP flows belonging to the same application. Here we take FTP as an example, the local/remote port pair of (21, any) identifies FTP control connections, and the tuple (any, any) identifies the remaining FTP data connections (either passive or port mode). Then we may set encryption protection to FTP control connections, but authentication protection to FTP data connections. However, if an application negotiates source/destination port numbers at runtime for all its IP flows, we cannot tune the protection settings for the different IP flows based on (unknown) port numbers. Therefore, we have to use the tuple of (any, any) to cover all its IP flows and specify a unified protection for them. On the other hand, the setting of one protection policy per application is still acceptable if either security or performance requirements are not stringent.

The BNF form of a protection setting is given as follows:

```
protection-setting -> "protection" string
                      "{" protection-list "}"
protection-list -> protection-list protection-item
                 | protection-item
protection-item -> "localport=" portnumber
                   "remoteport=" portnumber
                   protection-type
                   protection-level ";"
portnumber -> integer | "any"
protection-type -> "encryption" | "authentication"
protection-level -> "optional" | "mandatory"
```

The keyword *"protection"* indicates that the following tokens form a protection setting. As a unique identifier, the subsequent string is the name of the protection setting and a referral for a network setting if the network is a protected. The brace encloses a list of protection items. The selector of each item is a tuple of local and remote port numbers, indicating the class of TCP connections or UDP flows. The protection type—"encryption or authentication"—indicates the specific IPsec protection required. Since ESP provides both data authentication and encryption services, the data authentication is implicitly enabled in the encryption service. The last parameter of a protection item is the protection level: "optional or mandatory", which determines whether the defined protection is optional or mandatory.

Note that the optional policy in the proposed policy system needs resource isolation support at the kernel level; otherwise, it may be vulnerable to malicious attacks. The reason is that for an optional policy, an IPsec packet and a non-IPsec packet that match this optional policy would both be accepted. This opens a hole to attackers. Since the traffic in plaintext may still go through without packet-level authentication, an attacker can inject spoofed IP packets to abuse the victim's resources at the kernel level, which are shared among different applications. Fortunately, fine-grained resource isolation and accurate resource accounting have been proposed and implemented [9, 30] to prevent resource abuse and shield the privileged traffic from the other traffic. Implementation of the fine-grained resource isolation in the kernel space is the subject of our future work.

## 3.4 Runtime policy translation

Runtime policy translation is essential to the application-aware IPsec policy system. Based on the socket activities of an application, the application policy engine must translate the application security policies into the underlying IPsec policies. The translation needs to observe two principles. One principle is exclusiveness, and there are two levels of exclusiveness. The first-level exclusiveness requires that the security policies created for one application will not interfere with those for other applications. This guarantees that a configuration error in one application policy will not be propagated outside the domain of this application. The second-level exclusiveness requires that the security policies created for one class of traffic will not influence other class of traffic within the same application. Note that the most straightforward approach to achieving this two-level exclusiveness is to create the specific IPsec policy for each particular socket. The other principle is completeness, which requires that all the targeted traffic should be under protection. For UDP-based applications, each UDP datagram should be protected; and for TCP-based applications, the whole TCP connection should be protected, including handshaking messages, keepalive messages, and connection closing messages.

One difficulty of achieving completeness is that we need to know which local port the socket is going to be used beforehand, since the security policy of the socket should be configured before network operations begin. However, a dynamic socket—the socket not explicitly bound to a local port—would only be bound to a local port during the first network operation. We cannot predict the local port of the dynamic socket. Fortunately, the *bind* interface is able to bind a socket to an available local port if its caller sets the argument of local port number as 0. By calling *bind* with local port number "0", we can explicitly bind a dynamic socket to an available local port before configuring the security policies of the dynamic socket.

The general policy translation procedure is as follows. When the socket monitor intercepts an invocation of a socket function (e.g. *connect*), it retrieves the socket address information (IP address and port number) of that socket, knowing which application is currently making this call. Then, it passes the socket address, the information of application (e.g name or path), and the socket function name to the application policy engine. Based on the information of application, the application policy engine locates the corresponding application policy from the application policy repository. Depending on which socket function is called, the application policy engine will create or delete the underlying IPsec security policy for the particular socket with the knowledge of the socket address information. In the rest of the section, we describe the translation process in more detail for TCP-based and UDP-based communication, respectively.

### 3.4.1 TCP-based communication

As shown in Figure 4, we use an FTP application as an example to detail the translation procedure for TCP-based communication. Since the source and destination port numbers of a passive FTP data connection are determined at runtime, without our policy system, it is impossible to configure application-specific IPsec policy for FTP applications. In contrast, the application-aware IPsec policy system enables us to configure simple application polices for FTP client and server programs, respectively. The policy for the client-side program *ftp* specifies two different networks: 1.1.1.0/24 is the local trusted network, and the other is protected network. The protected network has the following security policy: the IP packets with remote port number 21 (FTP control connection) must be encrypted, and the other traffic (FTP data connection) should be authenticated. The policy for

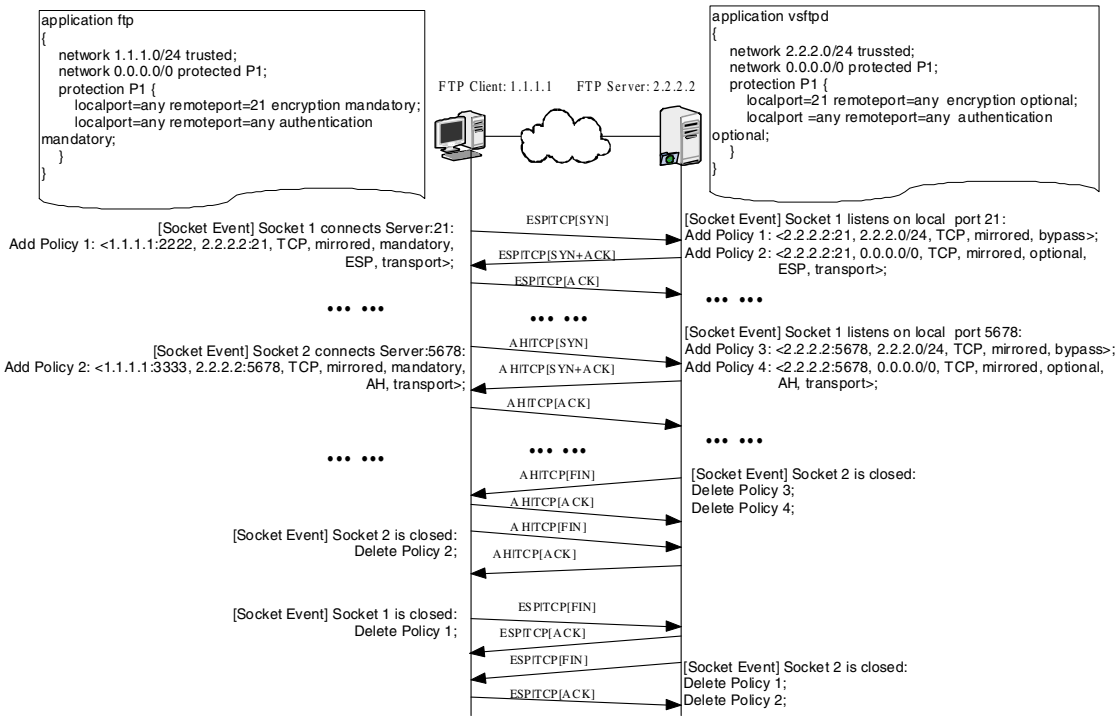



Figure 4: Runtime Policy Translation for TCP-based Communication.

the server program *vsftpd* also defines two different networks: 2.2.2.0/24 is the local trusted network, and the other is the protected network. The policy of the protected network is as follows: the traffic with the local port number 21 may be encrypted in option, and the other traffic may be authenticated as an option too.

When the server process *vsftpd* calls *listen* on a particular socket, we need to set up the appropriate security policies to protect the prospective incoming connections. At that moment, we do not know where a TCP connection comes from. So, we create a mirrored security policy for each network defined in the application policy. Here a mirrored security policy is a policy for both inbound and outbound processing. If the underlying IPsec implementation does not support mirrored security policy, individual inbound and outbound policies can be created instead. If the network is trusted, a bypass policy is created; if the network is untrusted, a discard policy is created; if the network is protected, then the protection item that matches the local port of this socket is retrieved from the protection setting of the network. In this example, we create a bypass policy (Policy1) for the trusted network, and an optional ESP policy (Policy2) for the protected one.

When the client-side process *ftp* calls *connect* to open one FTP control connection with the server, we need to prepare a mirrored security policy for this particular connection, since we have the knowledge of local/remote addresses and port numbers. The attribute of the network, in which the remote address is located, determines the configuration of its IPsec policy. If it is trusted or untrusted, then a bypass or discard policy is created; otherwise, the protection item that matches both the local and the remote port will be retrieved from the protection setting of this network, and an ESP or AH policy is created accordingly. In this example, the server's address locates in the client's protected network, and the first protection item matches the connection. So, we create a mandatory ESP policy (Policy1) for the TCP connection between the local address (1.1.1.1) and local port (2222) and the server's address and port (2.2.2.2 and 21). After policy configuration, the TCP packets of this connection are protected. If SAs of this policy have not been created, the first SYN packet will trigger the IKE process to negotiate and create SAs on both sides.

Afterwards, when the client initiates an FTP data connection, similar procedures happen on the server and the client. Upon the *listen* call at the server side, we create a bypass policy (Policy3) for the trusted network, and an optional AH policy (Policy4) for the protected network. Upon the *connect* call at the client side, we create a mandatory AH policy (Policy2) for this connection. After the socket is closed, the socket monitor notifies the application policy engine to delete the security policies related to the socket.

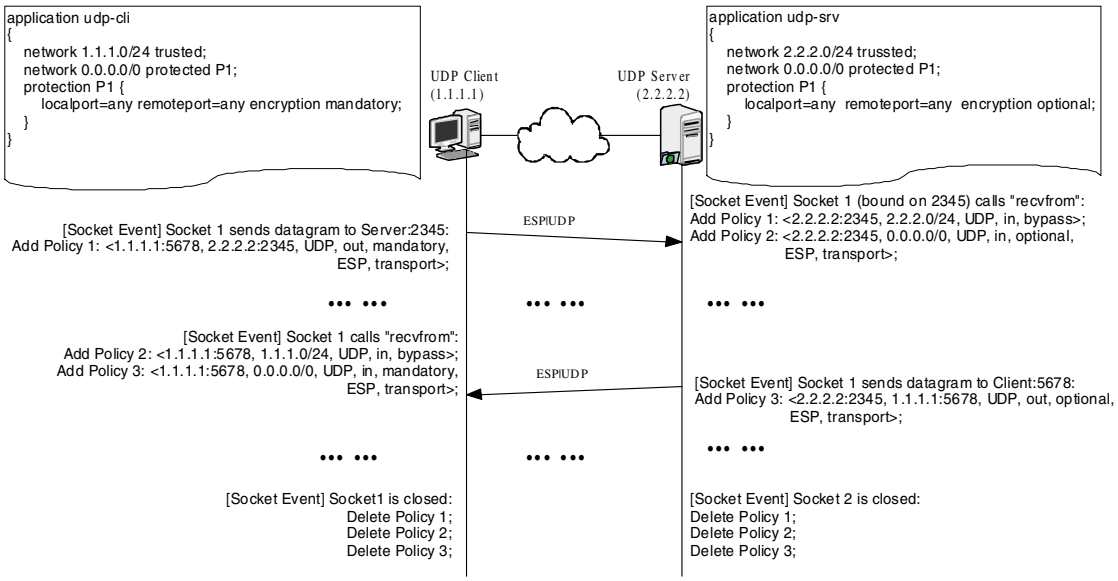



Figure 5: Runtime Policy Translation for UDP-based Communication.

### 3.4.2 UDP-based communication

In contrast to connection-oriented TCP-based communication, in which a stream socket is bound to communicate with a fixed destination, a datagram socket can talk with multiple destinations simultaneously. Therefore, the policy translation of UDP is more complicated than that of TCP.

Figure 5 shows an example of translation process for UDP-based communication. When the server process *udpsrv* calls *recvfrom* to receive a datagram from the socket that bound to local port 2345, the policy engine retrieves the corresponding application policy. Then, for each network defined in this application policy, it creates the appropriate IPsec policy. Here we create a bypass policy (Policy1) for the trusted network, and an optional ESP policy (Policy2) for the protected network. The procedure is similar to that of *listen* in TCP-based communication. The difference is that for *listen* we create mirrored policies, but for *recvfrom* we only create inbound policies.

When the client process *udp-cli* calls *sendto* to send a datagram to the server, the application policy engine creates an outbound policy if the policy has not been created before. The destination address and port number are the server's, while the source address and port number are fetched from the socket descriptor. This procedure is similar to that of *connect* in TCP-based communication. The difference is that for *connect* we create mirrored policies, but for *sendto* we only create outbound policies. In this example, we create a mandatory ESP policy (Policy1).

After that, the server returns a datagram to the client, and similar procedures happen on both sides, as shown in Figure 5. Upon the *recvfrom* call at the client side, we create two inbound security polices: one bypass policy (Policy2) for the trusted network, and one mandatory ESP policy (Policy3) for the protected network. Upon the *sendto* call at the server side, we create an optional outbound ESP policy (Policy3).

Recall in TCP, once a socket descriptor is closed, the application policy engine will remove all security policies related to it. However, it would be problematic in UDP if we only remove security policies after a socket is closed. It is possible that even if a socket only talks with a few destinations simultaneously, a large number of policies may have been created for the socket after a long time due to the accumulation effect. Our solution is to define a TTL (time-to-live) for each outbound UDP policy, and delete it when its TTL expires.

In this example, we only describe the policy translation performed on *recvfrom* and *sendto*. UDP-based applications can also use *send* to transmit a datagram after *connect* is called. The translation procedure for *send* is the same as the one for *sendto*.

## 4 Implementation

We have implemented a prototype of our application-aware IPsec policy system in Linux (Kernel 2.6). There is a native IPsec implementation in the Linux kernel after version 2.5.47, which is similar to KAME implementation [4] in the BSD variants such as FreeBSD, NetBSD and OpenBSD. The user-level utilities [2], including *racoon* (an IKE daemon), PF_KEYv2 library

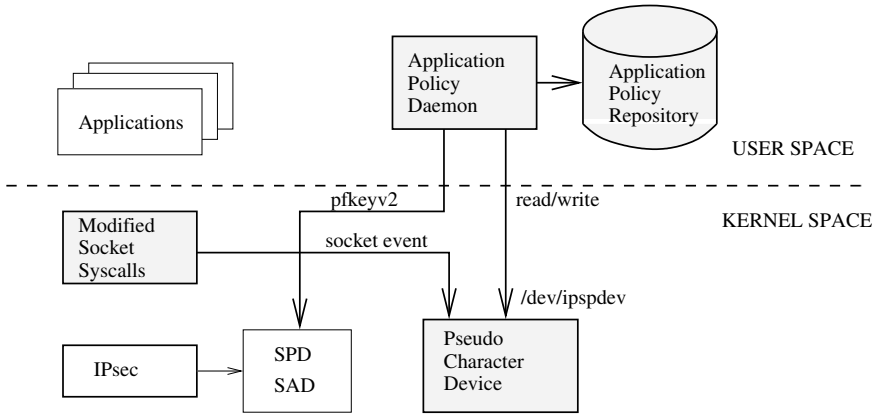



Figure 6: The Prototype in Linux (Kernel 2.6).

*pfkeyv2*, and *setkey* (a tool for policy configuration), have also been ported to Linux kernel 2.6.

The structure of this prototype is illustrated in Figure 6. The function of the socket monitor is implemented by modifying socket system calls, and the application policy engine is implemented as a user-level daemon. The communication between the user-level policy daemon and the kernel is through a pseudo character device. The socket monitor enqueues the socket events into the pseudo character device */dev/ipspdev*, and the policy daemon uses regular file reading operation interfaces (i.e. `read`) to retrieve these socket events from */dev/ipspdev* and writing operation (i.e. `write`) to acknowledge them after translation. The policy daemon creates and deletes security policies via the interface defined in the *pfkeyv2* library. The socket monitor can detect which application is running based on the current process id *pid*, which is retrieved from the appropriate entry in the */proc* filesystem.

As an independent kernel module, the kernel-space portion, including the socket monitor and the pseudo character device, only consists of 500 lines of C code. We also notice that the socket monitor could be implemented in user space in some operating systems. For example, in Windows systems it can be implemented as a Winsock2 Layered Service Provider (a dynamic link library). So, the programming in kernel space is minimal.

In terms of security policy, like the KAME implementation, the native IPsec implementation in Linux kernel 2.6 supports three security levels: *use*, *require*, and *unique*. The *use* level indicates that the kernel can utilize an SA if it is available, otherwise the kernel just stays in normal operation. This security level can be used for an optional policy. The *require* level indicates that SA is a must whenever the kernel sends a packet matched with the policy. The *unique* level is similar to *require*. In addition, it allows the policy to bind with the unique out-bound SA. Both the *require* and *unique* levels can be used for a mandatory policy. However, since we create fine-grained security policies, using *unique* security level may cause frequent policy negotiation, leading to performance degradation. Besides, we have no such strict requirement that no policies can share an SA. Therefore, we choose the *require* level for a mandatory policy.

## 5 System Evaluation

The purpose of our system evaluation is two-fold: (1) to demonstrate the efficacy of the proposed policy system in security enhancement and protection; and (2) to measure the overhead of the proposed policy system. We employ a simple setup for the testbed, which consists of two PCs connected by a 100Mbps Ethernet. The server machine has two Pentium-4 2.8GHZ CPUs, and 512MB memory, and the client machine has one Pentium-4 2.8GHZ CPU and 256MB memory. Both have Redhat 9.0 installed with Linux kernel 2.6.7.

We conduct file-transfer across the testbed using secure-version and legacy-version applications (*sftp* and *ftp*), respectively. With respect to the different running applications on the testbed, we divide our experiments into two classes: the experiments running *sftp* and the experiments running *ftp*. As shown in [28], the median size of Web objects is 2KB, the median size of P2P objects is 4MB, and about 5% of Kazaa objects are over 100MB. Therefore, we apply two typical workloads in our experiments: the small-file workload and the large-file workload. The small-file workload consists of downloading 5000 different small files, each of which is 2KB long; while the large-file workload consists of downloading a single large file of 100MB. The performance metric we used here is end-to-end file transfer time.

In the extreme case of downloading a large number of small files in FTP, we intend to amplify the overhead of policy translation. On the other hand, downloading a single large file amortizes the overhead incurred by policy translation, and demonstrates the basic processing overhead of IPsec in comparison with other security mechanisms. Equipped with the IPsec and the proposed policy system, the total overhead of an application can be classified into the following three categories: the base overhead of the application, the additional overhead incurred by IPsec processing, and the additional overhead incurred by the policy system. In order to evaluate these three kinds of overheads, we measure the end-to-end file transfer time under three different protections: no protection at all, IPsec protection with the direct IPsec policy configuration, and the application-aware IPsec policy system protection.

## 5.1 Security Enhancement for Secured Applications

For those secured applications that already have identification authentication, data confidentiality and integrity protection, the application data has already been encrypted and authenticated at the upper layers. Thus, the packet-level authentication provided by IPsec is effective enough to counter various network-layer attacks such as IP spoofing and enhance the security of these applications.

```
application sshd
{
    network 192.168.1.0/24 protected Enh;
    network 0.0.0.0/0 trusted;
    protection Enh {
        localport=any remoteport=any
                    authentication mandatory;
    }
}
```

```
application ssh, sftp, scp
{
    network 192.168.1.0/24 protected Enh;
    network 0.0.0.0/0 trusted;
    protection Enh {
        localport=any remoteport=any
                    authentication mandatory;
    }
}
```

Figure 7: A Policy Example for SSH Server and Client.

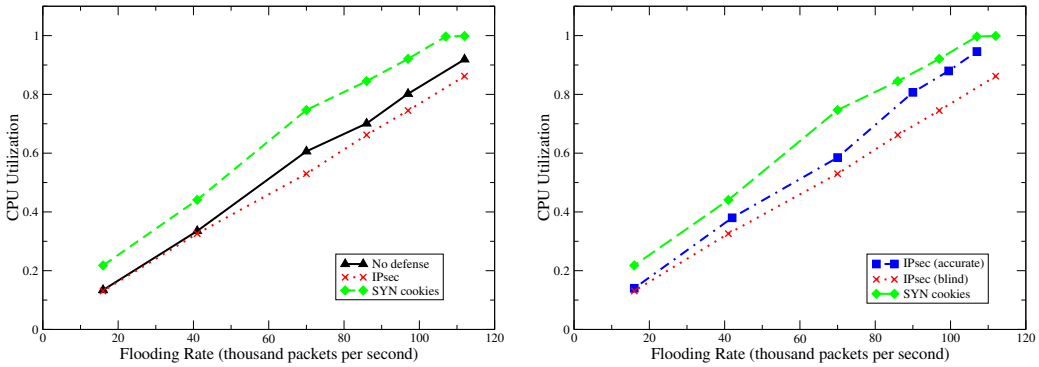
Here we use *ssh* as an example. Figure 7 shows the application policies for the SSH server daemon *sshd* and the client processes *ssh*, *sftp*, and *scp*. The policies are simple: the local area network 192.168.1.0/24 is a protected network with mandatory authentication.

To demonstrate the virtue of the network-layer security enhancement, we launch SYN flooding attacks from the client machine targeting at the SSH service port 22 of the server. The spoofed source IP addresses are randomly chosen from a private network of 10.0.0.0/4, and the server's responses to these non-existing addresses are redirected to a third machine that drops these bogus packets directly. We configure the server with three different settings: no protection, IPsec protection, and SYN cookies. The SYN flooding rate varies from 15,000 to 112,000 packets per second that is the maximum flooding rate we can reach in the 100Mbps Ethernet.

We observe that without protection of IPsec or SYN cookies, the server is easily clogged and no legitimate SSH connection can be established under the minium flooding rate of 15,000 packets per second. Both SYN cookies and IPsec are effective defense mechanisms, since a legitimate TCP connection can be established even under the maximum flooding rate of 112,000 packets per second. Nevertheless, SYN cookies consume considerably more CPU cycles than IPsec. In particular, when the flooding rate exceeds 100,000 packets per second, a legitimate SSH client cannot login even though the TCP connection has been established, due mainly to the shortage of CPU cycles and the expensive cryptographical computation thereafter. The CPU utilization under different protection settings is depicted in Figure 8 (a). The extra CPU cycles used by SYN cookies are due to cookie generation and sending responses with each spoofed SYN requests. By contrast, IPsec drops spoofed SYN requests directly and only accepts AH packets.

A potentially more effective attack against IPsec would be to flood spoofed AH packets, which may exhaust the victim's CPU resource on verifying MACs of AH packets. The challenge of launching such an attack is the necessity of knowing the detailed information of the victim's inbound SA, including the source IP address, SPI and current position of anti-replay window. Otherwise, the spoofed AH packets without correct combination of source IP address, SPI and sequence number will be easily sifted out by IPsec via lightweight checking. Thus, as shown in Figure 8 (b), with IPsec protection the number of CPU cycles burnt by blind AH flooding attacks (i.e., spoofing without correct detailed information about inbound SA) is similar to that of SYN flooding attacks. To amplify the effect of AH flooding attacks, we flood spoofed AH packets with the correct combination of source IP address, SPI and sequence number. Figure 8 (b) clearly demonstrates that the CPU overhead incurred by IPsec is still less than that of SYN cookies for protecting SYN flooding attacks, and it is no surprise that IPsec takes more CPU cycles in defending such attacks than blind AH flooding attacks.

The performance of *sftp* under different protections is listed in Table 1. With application policy protection, the elapsed time of *sftp* increases from 20.945s to 22.493s for the small-file workload, and from 9.355s to 9.452s for the large-file workload. For both workloads, the performance degradation induced by the the packet-level authentication for *sftp* is insignificant. In the SSH protocol, only one TCP connection needs to be established for each login session, and all the subsequent data (user-commands and files) is transferred over the single TCP




(a) SYN Flooding (b) AH Flooding

Figure 8: The CPU Utilization under SYN and AH Flooding Attacks

connection. Thus, the policy translation overhead for that single TCP connection is negligible compared to the total file transfer time. We will evaluate the policy translation overhead in Section 5.2.

Table 1: Experiments of Protecting SSH

| | **small-file case** (s) | **large-file case** (s) |
|---|---|---|
| *sftp* | 20.945 | 9.355 |
| *sftp* with AH | 21.776 | 9.409 |
| *sftp* with App Policy | 22.493 | 9.452 |

## 5.2 Security Protection for Legacy Applications

Many Internet applications are not secure, such as FTP, HTTP, Telnet, SMTP, POP3, etc. Their traffic, including sensitive authentication information (e.g. username and password), is transmitted in plaintext. Most of them are still widely used. By specifying application-aware IPsec policies, these legacy applications can transparently obtain network-layer security protection. Furthermore, compared with other security mechanisms deployed at the upper layers, the network-layer security provided by IPsec protects these legacy applications against various network-layer attacks.

Here we use the FTP protocol as an example. There are two kinds of TCP connections in FTP: FTP control connections and FTP data connections. The login information (username and password) and all user-commands are transferred over the FTP control connection; while the real file data and directory information are transferred over the FTP data connection. Therefore, the FTP control connection must be encrypted, and the FTP data connection can be encrypted or authenticated according to the specific circumstances. Here, we give two different suites of policies. The left suite of application policies in Figure 9 is for FTP server process *vsftpd* and client process *ftp*. Since it is configured to encrypt all FTP traffic, we simply call it the secure policy. The right suite of application policies is configured to encrypt the FTP control connection but only authenticate the FTP data connection. Thus, we call it the fast policy.

Table 2: Experiments of Securing FTP protocol

| | **small-file case** (s) | **large-file case** (s) |
|---|---|---|
| FTP | 4.838 | 9.837 |
| FTP with AH/ESP | 6.133 | 9.983 |
| FTP with Fast Policy | 11.909 | 10.062 |
| FTP with ESP | 7.383 | 14.262 |
| FTP with Secure Policy | 13.131 | 14.282 |

The performance of FTP under the different protections is listed in Table 2. With the security protection, the file-transfer delay of the large-file workload increases from 9.837s to up to 14.282s, which is mainly caused by the overhead of IPsec processing. In the case of the small-file workload with the security protection, the file-transfer delay increases from 4.838s to up to 13.131s, which is caused by both the overhead of IPsec processing and the overhead of policy processing. Note that since multiple security policies may share a single SA in our implementation, the overhead of SA establishment is negligible even in the case of the small-file workload. Figure 10 plots the proportion of the application policy processing overhead to the whole file-transfer overhead under the different workloads. The percentage of the policy processing overhead in the small-file workload is 48.5% for the fast policy and 43.8% for the secure policy, respectively. Compared with the large-file workload case, the large portion of overhead incurred by the policy system is due to the following two reasons.

```
application vsftpd
{
   network 192.168.1.0/24 protected P1;
   network 0.0.0.0/0 trusted;
   protection P1 {
      localport=any remoteport=any
                        encryption mandatory;
   }
}
```

```
application ftp
{
   network 192.168.1.0/24 protected P1;
   network 0.0.0.0/0 trusted;
   protection P1 {
      localport=any remoteport=any
                        encryption mandatory;
   }
}
```

(a) Secure Policy

```
application vsftpd
{
   network 192.168.1.0/24 protected P1;
   network 0.0.0.0/0 trusted;
   protection P1 {
      localport=21 remoteport=any
                        encryption mandatory;
      localport=any remoteport=any
                        authentication mandatory;
   }
}
```

```
application ftp
{
   network 192.168.1.0/24 protected P1;
   network 0.0.0.0/0 trusted;
   protection P1 {
      localport=any remoteport=21
                        encryption mandatory;
      localport=any remoteport=any
                        authentication mandatory;
   }
}
```

(b) Fast Policy

Figure 9: A Policy Example for FTP Server and Client.

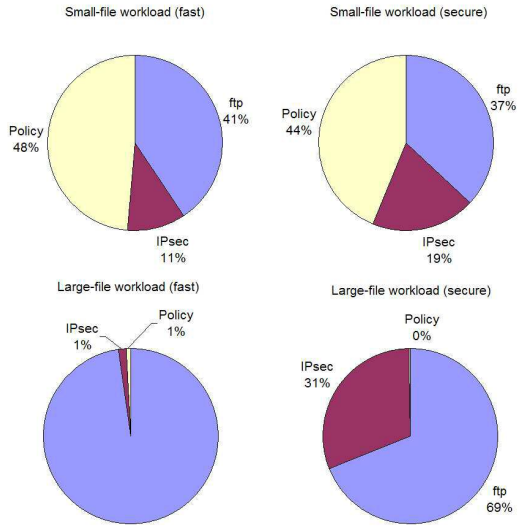



Figure 10: The Overhead of Application Policy Processing in Securing FTP

- In the FTP protocol, a new TCP connection has to be established for each file to be transferred, and our policy system needs to create appropriate IPsec policies for each TCP connection before it is to be established and delete them after it is closed.
- The file size of small-file workload and the short propagation delay in the LAN environment amplify the overhead of policy system.

Note that the overhead of policy processing per connection is stable, if not constant, and is independent of the file size and the propagation delay. By dividing the additional overhead caused by the policy processing (about 5.7s) with the number of TCP connections (5000), we can derive the cost of policy processing for one TCP connection, which is about 1.1ms. Compared to the end-to-end delay of 20-200ms in the WAN environment, the policy processing overhead per connection is negligible.

In our implementation, the socket monitor residing in the kernel space needs to report the socket activities to the user-space policy engine. Thus, the communication between kernel and user space and the synchronization between the policy engine and the processes being monitored induce the major overhead of the policy system. However, even in the low-latency (less than 1ms) LAN environment with the small-file workload that amplifies the overhead of the proposed policy system, the performance of FTP protected by either the fast policy or the secure policy outperforms that of *sftp* with a large margin. Note that for a fair comparison, we choose the same encryption algorithm (3DES) and authentication algorithm (HMAC_SHA1) for SSH and IPsec. Therefore, we can conclude that the overall performance of our policy system is satisfactory in both scenarios.

To demonstrate that the application policy for FTP is correctly enforced, we dump the FTP traffic between the server and the client, which is shown in Appendix A.

## 6 Conclusion

In this paper, we presented an application-aware IPsec policy system as a flexible middleware to provide Internet applications with network-layer security protection. Since IPsec is at the network layer and lacks knowledge of application context, the current IPsec policy is rigid and coarse-grained, providing all or nothing security protection to different Internet applications. To make the IPsec policy system flexible and application-aware, we installed a socket monitor at the network stack of end hosts. The socket monitor detects the socket activities of Internet applications, and passes them to the application policy engine. Then, the application policy engine translates the corresponding application policies into the underlying security policies via the existing policy management interface. Moreover, we defined an application policy specification language to alleviate administrator's burden of configuring and distributing application policies in different platforms. We have implemented a prototype of the proposed policy system in Linux (Kernel 2.6) and evaluated its efficacy in the testbed. Our experiments have shown that utilizing the application-aware IPsec policy system, both secured applications and

legacy applications can obtain the end-to-end security enhancement or protection transparently. Furthermore, the overhead of policy translation has insignificant impact upon the end-to-end transfer delay over the Internet.

## 7 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. We would also like to thank William Bynum and Jianping Pan for helpful feedback.

## References

[1] Ip security policy working group. `http://www.ietf.org/html.charters/ipsp-charter.html`.

[2] IPsec tools. `http://ipsec-tools.sourceforge.net`.

[3] IPsec2k library. `http://sourceforge.net/projects/ipsec2k`.

[4] KAME Project. `http://www.kame.net`.

[5] PF_KEY Extensions for IPsec Policy Management in KAME Stack. `http://www.kame.net/newsletter/20021210`.

[6] Stunnel–universal ssl wrapper. `http://www.stunnel.org`.

[7] AIELLO, W., BELLOVIN, S. M., BLAZE, M., CANETTI, R., IOANNIDIS, J., KEROMYTIS, A. D., AND REINGOLD, O. Efficient, DoS-Resistant, Secure Key Exchange for Internet Protocols. In *ACM Conference on Computer and Communication Security (CCS'02)* (Washington D.C, USA, November 2002).

[8] ARKKO, J., AND NIKANDER, P. Limitations of IPsec Policy Mechanisms. In *Security Protocols, Eleventh International Workshop* (Cambridge, UK, April 2003).

[9] BANGA, G., DRUSCHEL, P., AND MOGUL, J. Resource containers: A new facility for resource management in server systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)* (New Orleans, LA, February 1999).

[10] BEAULIEU, S., AND PEREIRA, R. Extended Authentication with IKE (XAUTH). *Internet Draft, Internet Engineering Task Force* (Oct 2001).

[11] BLAZE, M., IOANNIDIS, J., AND KEROMYTIS, A. D. Trust management for IPsec. *ACM Transactions on Information and System Security (TISSEC) 5*, 2 (2002), 95–118.

[12] CONDELL, M., LYNN, C., AND ZAO, J. Security Policy Specification Language. *Internet Draft, Internet Engineering Task Force* (October 1998).

[13] HARKINS, D., AND CARREL, D. The Internet Key Exchange (IKE). *RFC 2409, Internet Engineering Task Force* (November 1998).

[14] IOANNIDIS, J. Why don't we still have IPsec, Dammit. In *Invited talk at USENIX Security Symposium '02* (August 2002).

[15] KAUFMAN, C. The Internet Key Exchange (IKEv2) Protocol. *Internet Draft, Internet Engineering Task Force* (August 2004).

[16] KAUFMAN, C., PERLMAN, R., AND SOMMERFELD, B. DoS protection for UDP-based protocols. In *ACM conference on Computer and Communication Security (CCS'03)* (Washington D.C, USA, October 2003), pp. 2–7.

[17] KENT, S., AND ATKINSON, R. IP Authentication Header. *RFC 2402, Internet Engineering Task Force* (November 1998).

[18] KENT, S., AND ATKINSON, R. IP Encapsulating Security Payload (ESP). *RFC 2406, Internet Engineering Task Force* (November 1998).

[19] KENT, S., AND ATKINSON, R. Security architecture for the internet protocol. *RFC 2401, Internet Engineering Task Force* (November 1998).

[20] LITVIN, M., SHAMIR, R., AND ZEGMAN, T. A Hybrid Authentication Mode for IKE. *Internet Draft, Internet Engineering Task Force* (June 2001).

[21] MAUGHAN, D., SCHNEIDER, M., AND SCHERTLER, M. Internet Security Association and Key Management Protocol (ISAKMP). *RFC 2408, Internet Engineering Task Force* (November 1998).

[22] MCDONALD, D. A Simple IP Security API Extension to BSD Sockets. *Internet Draft, Internet Engineering Task Force* (November 1996).

[23] METZ, C., AND PHAN, B. PF_KEY Key Management API Version 2. *RFC 2367, Internet Engineering Task Force* (October 2001).

[24] MILTCHEV, S., IOANNIDIS, S., AND KEROMYTIS, A. D. A Study of the Relative Costs of Network Security Protocols. In *USENIX Annual Technical Conferences, Freenix Track* (Monterey, CA, June 2002), pp. 41–48.

[25] MORRIS, R. T. A weakness in the 4.2bsd unix TCP/IP software. In *Computing Science Technical Report 117, AT&T Bell Laboratories* (Murray Hill, NJ, February 1985).

[26] Opportunistic Encryption. `http://www.freeswan.org`.

[27] PIPER, D. The Internet IP Security Domain of Interpretation. *RFC 2407, Internet Engineering Task Force* (November 1998).

[28] SAROIU, S., GUMMADI, K., DUNN, R., GRIBBLE, S., AND LEVY, H. An analysis of internet content delivery systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)* (Boston, MA, December 2002).

[29] SOMMERFELD, W. Requirements for an IPsec API. *Internet Draft, Internet Engineering Task Force* (June 2003).

[30] SPATSCHECK, O., AND PETERSON, L. Defending against denial of service attacks in Scout. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)* (New Orleans, LA, February 1999).

[31] WANG, X., AND REITER, M. Mitigating bandwidth-exhaustion attacks using congestion puzzles. In *ACM conference on Computer and Communication Security (CCS'04)* (Washington D.C, USA, October 2004).

[32] WU, C. L., WU, S., AND NARAYAN, R. IPSEC/PHIL (Packet Header Information List): Design, Implementation, and Evaluation. In *IEEE International Conference on Computer Communication and Networks '01* (October 2001).

## Notes

[1] At present each IPsec vendor has a very different policy management interface. Note that if it does not exist in one operating system, it will be much easier to build such interface than a standard IPsec API.

## A Efficacy of Application Policy for FTP

Using *ethereal*, we dump the FTP traffic between the server and the client, and show that the application policy for FTP is correctly enforced (see Figure 11). The left figure shows the FTP traffic without any protection, in which the sensitive information like username and password can be easily fetched. The right figure shows the FTP traffic protected by the fast policy, in which the FTP control connection is protected by ESP as predicted, and hence the confidentiality is guaranteed.

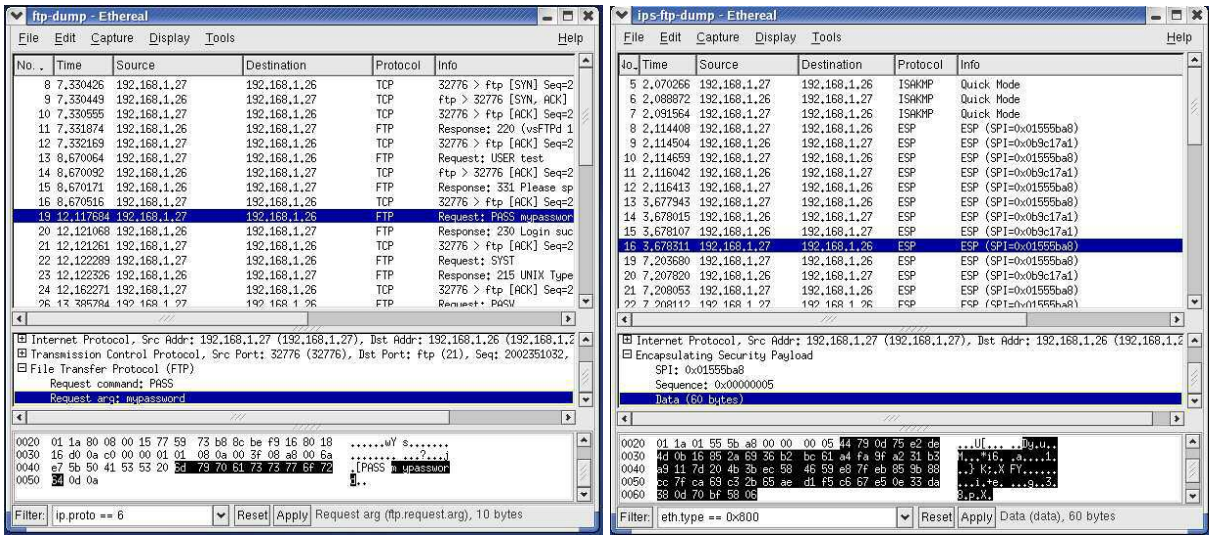



Figure 11: Screenshots of Unprotected and Protected FTP Connections