IPSec/VPN Security Policy: Correctness, Conflict Detection and Resolution[1]

Zhi Fu, S. Felix Wu
Computer Science Department
North Carolina State University
{zfu, wu}@eos.ncsu.edu

He Huang, Kung Loh
Nortel Networks
{huanghe, kungloh}@nortelnetworks.com

Fengmin Gong, Ilia Baldine, Chong Xu
Advance Networking Research
MCNC
{gong, ibaldin, chong}@anr.mcnc.org

Abstract

IPSec (Internet Security Protocol Suite) is a typical policy-enabled networking service and its functions will be executed correctly only if its policies are correctly specified and configured. Manual IPSec policy configuration as currently in practice is inefficient and error-prone. An erroneous policy could lead to communication blockade or serious security breach. In addition, even if policies are specified correctly in each domain, the diversified regional security policy enforcement can create significant problems for end-to-end communication because of interaction or conflicts among policies in different domains. A policy management system is, therefore, demanded to systematically manage and verify various IPSec policies in order to ensure an end-to-end security service. This paper contributes to the development of an IPSec policy management system in two aspects: Firstly, we defined a high-level security requirement, which not only is an essential component to automate the policy specification process of transforming from security requirements to specific IPSec policies but also can be used as criteria to detect conflicts among IPSec policies, i.e. policies are not in conflict only if they satisfy all requirements. Secondly, we developed mechanisms to detect and resolve conflicts among IPSec policies in both intra-domain and inter-domain environment.

Keyword

IPSec Policy, Policy Conflict, Conflict Detection and Resolution, Security Requirement, Policy Hierarchy

1. Introduction

IPSec [1] is receiving widespread deployment to restrict access or selectively enforce security operations for VPN implementation etc. IPSec is a typical policy-enabled networking service in that IPSec functions will be executed correctly only if policies are correctly specified and configured. The process of handling either inbound or outbound traffic will consult the IPSec security policy database (SPD) to determine the treatment of the traffic.

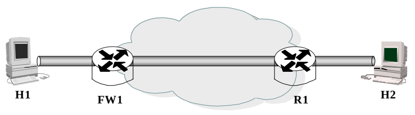
Although manually configuring the IPSec policy database may be fine for a small network, it is inefficient and error-prone for large distributed networking systems. Because of the growing number of secure Internet applications, IPSec policy deployment will be more and more complex in the near future. Therefore, a policy management system is clearly demanded to automatically and systematically configure and manage various IPSec policies.

Policy is to implement people or corporation's desired requirement. In a policy hierarchy [2], a requirement (high level policy) is an objective while implementation policies (low level policy)

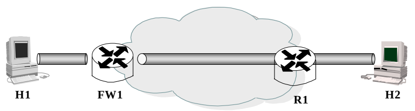
[1] This Research is supported in part by the U.S. Department of Defense Advanced Research Projects Agency under contract DABT63-97-C-0045 and in part by Nortel Networks.

are specific plans to meet the objective. One requirement might be satisfied by different sets of implementation policies. Therefore, the policy specification process is the process to transform from a requirement to specific implementation policies to realize the requirement. The current security policy proposals for IPSec [3,4,5] focus on policy rules that can be "deterministically" enforced by one or more network elements (i.e., PEP, Policy Enforcement Points). In other words, the security requirements of a policy domain have been manually transformed into LDAP Policy Framework rules. There is, therefore, a currently vague relationship between a desired security requirement and specific IPSec policies to realize the requirement. However, to manage policies for large distributed systems, it is desirable to separate requirement and policies because: 1) Policies are specific ways to implement requirements such that requirements are more static and policies are more dynamic. The separation allows requirement component to be reused while policies to be dynamically modified and improved without needs to alter the requirement component. 2) The separation permits automation of the process to transform from requirements to policies. 3) Explicitly specified requirements can be used as criteria to verify the correctness of low level policies.

At the first glance, it seems that requirement and IPSec policy may directly map to each other. Yet, we can use the following example to illustrate the difference between a security requirement and specific IPSec policies to fulfill the requirement.



a) One direct tunnel



b) Two chained tunnels

Figure 1: Security Requirement and IPSec Policies

In figure 1, if a sensitive communication from a host machine H1 to another host machine H2 requires to be encrypted during transmission anywhere from H1 to H2 except the firewall FW1, which is trusted to review content, then both of configurations shown in the figure 1 satisfy the requirement. In configuration a), H1 directly builds an encryption tunnel with H2 to protect the sensitive traffic while in b), two IPSec tunnels are chained at FW1, which will decrypt the traffic from cipher text back to plain text, then re-encrypt again for the second encryption tunnel. Similarly, more different chained tunnel configurations can satisfy the requirement if some other security gateways on the path are also trusted to review the content.

In a large distributed system or inter-domain environment, the diversified regional security policy enforcement can create significant problems for end-to-end communication. In the above example, suppose FW1 needs to examine traffic content for the purpose of intrusion detection and a policy is set up at FW1 to deny all encrypted traffic to enforce its content examination requirement. Yet, H1 and H2 build a direct tunnel without awareness of existence of the firewall and its policy rules. Therefore, all the traffic will be dropped by FW1. The scenario shows that each policy satisfies its corresponding requirement while all policies together can cause conflicts. In this case, if two chained tunnels are built as b) in figure 1, then both requirements are satisfied thus the traffic will go through with appropriate protection. However, end users have no idea about topology or policy information to make right choice of policy configurations. A policy management system should be responsible to provide assurance of end-to-end protection and transmission.

The following shows another scenario that each policy may be satisfying for individual requirement while all policies together cause violation:

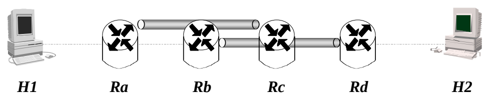


Figure 2: Overlapping tunnels

In this scenario, there are four routers Ra, Rb, Rc and Rd on the path from H1 to H2. Assume there are two requirements for the traffic from H1 to H2: one is integrity protection from Ra to Rc and the other is confidentiality protection from Rb to Rd. Two tunnels are built from Ra to Rc and from Rb to Rd accordingly. With the tunnels, the traffic will be encapsulated by Ra, then encapsulated again by Rb to send to Rd. When Rd decapsulates and finds the destination is Rc, Rd will send traffic back to Rc. Finally Rc will decapsulate and send traffic to its real destination H2. Although it is originally intended to encrypt traffic from Rc to Rd, the traffic is eventually sent in clear from Rc to Rd because of tunnel interaction.

Therefore, an IPSec policy management system will need to not only systematically specify policies to fulfill requirements but also tackle the topological interaction and conflicts among IPSec policies. This paper contributes to the development of an IPSec policy management system in two aspects: First we specified security requirements in a high level. Then, we developed mechanisms to detect and resolve conflicts among IPSec policies to ensure secure end-to-end communications.

The remaining paper is organized as follows. In section 2, we define security requirements and their implementation policies. Section 3 develops an algorithm to systematically verify the correctness of policies. Section 4 discusses the policy resolution problem and solutions. Then section 5 talks about deployment issues by introducing Celestial system that the conflict detection and resolution mechanisms can be deployed in to provide end-to-end security service. Finally, section 6 presents some related work and section 7 concludes the paper and outlines future work.

2. Security Requirements and Implementation Policies

Requirement is high level objective while implementation policies are low level specific plans to meet the objective. One important task of IPSec policy management is to represent security requirements in a high level efficiently and unambiguously. We will first analyze the security requirements for IPSec policies.

2.1 Security Requirement Analysis

2.1.1 Access Control Requirement (ACR)

One fundamental function of security is to conduct access control that is to restrict access only to trusted traffic. A simple way to specify an ACR is:

flow id. → *deny | allow*

2.1.2 Security Coverage Requirement (SCR)

Another important function is to apply security functions to prevent traffic from being compromised during transmission across certain area, which requires the security protection to insulate the traffic from all links and nodes within the area. However, optionally, users can authorize certain nodes in the area to access content since some nodes on the path may need and be trusted to examine content. For example, a simple way to specify a SCR to protect traffic from *from* to *to* by a security function with certain strength could be:

flow id. → *enforce (sec-function, strength, from, to, excepted-nodes)*

2.1.3 Content Access Requirement (CAR)

Some nodes may need to access content of certain traffic, for example, a firewall with an intrusion detection system (IDS) may need to examine content to determine the characteristic of the traffic. Yet, one node is not able to view the content of traffic if an encryption tunnel is built across it. Similarly, there might be certain nodes that need to modify content for special processing but can not if authentication tunnels are built across them. However, one node can access the content if the tunnels are built with it as connecting point such that it can de-apply the security functions, access the content then apply the security function again for the next tunnel. We allow CAR to be explicitly specified such as to disallow IPSec tunnels to be built without the participation of the CAR nodes. A simple way to specify a CAR could be:

flow id, sec-function, against-nodes → *deny | allow*

2.1.4 Security Association Requirement (SAR)

Security Associations (SA) [1] need to be formed to perform encryption/authentication function. There might be needs to specify some nodes to desire/not desire to set up SA with some other nodes because of trust/distrust relationship. A simple way to specify a SAR could be:

flow id, SA-peer1, SA-peer2 → *deny | allow*

The above four requirements expressed the needs of IPSec users with respect to not only the access control and protection of traffic but also impacts and attributes of security enforcement.

2.2 Definitions of Security Requirements and Implementation Policies

Definition 1: IPSec/VPN policy can be specified in two different levels: the requirement level security policies (or security requirements in short) and the implementation level security policies (or security implementation policies in short). Two level security policies are the same in basic form as defined below but different in attributes and semantics as will be defined respectively below. For example, the IPSec policies that are installed in security gateways to operate on the passing traffic are implementation policies.

Definition 2: A security policy P is a rule of the following form: If _condition_ C then _action_ A,

$$P \;=\; C \;\rightarrow A.$$

Definition 3: The condition part of a security policy is composed of a set of sets $S_1, S_2, ..., S_N$, each of which is a finite set of values of a specific attribute, we call a selector, to associate certain traffic with a particular policy. The condition is met, or a packet is selected by a policy, if and only if each of the packet's value of a selector is an element of the corresponding set of the selector, which can be expressed by Cartesian product of the sets,

$$C \;=\; \mathop{\times}_{i=1}^{N} S_i$$

For example, if selector attributes of a policy are source address and destination address, then the traffic from *a* to *c* will be selected by the policy with condition of source address *{a,b}* and destination address *{c,d}* because

$$a \in \{a,b\} \qquad c \in \{c,d\} \qquad thus \qquad (a,c) \in \{a,b\} \times \{c,d\}$$

Therefore, selectors are defined as the attributes used to match packets with policies. We will specify selectors and their values in detail for requirement level and implementation level security policies respectively below.

Definition 4: The action part of a security policy is of form $a(t_1, t_2, ..., t_M)$ where *a* is an action type with *M* parameters that specify attributes of the action. One action type is for each policy. We will define each action type and associated parameters in detail for requirement level and implementation level security policies respectively below.

$$A \;=\; a(t_1, t_2, ..., t_M)$$

Definition 5: The requirement level security policies have the following selectors in the condition part:

flow identity
[sec-function against-list][2]
[SA-peer1 SA-peer2]

and have the following action types and parameters in the action part:

deny
allow
enforce (sec-function strength [algorithm] from to [except-list])[3]

In the condition part, each S_i is a finite set:

- *flow identity* is composed of 5~6 sub-selectors*: src-addr, dst-addr, src-port, dst-port, protocol, [user-id]* to identify the traffic flow;
- *sec-function against-list* is to specify the condition that certain security functions (e.g. authentication or encryption) are applied against particular nodes specified by the finite set *against-list*. This condition can be used in expressing the Content Access Requirement of certain nodes by denying certain security function(s) against them;
- *SA-peer1 SA-peer2* is to specify the condition that any node of the set of *SA-peer1* forms SA with any node in the set of *SA-peer2*. This condition can be used in expressing the Security Association Requirement by explicitly denying/allowing particular nodes to build association relationship.

In the action part, each t_j is a finite set:

- *sec-function* is to specify the security function(s) (e.g authentication or encryption) required for certain traffic;
- *strength* is to specify desired level of security protection such as ordinary, middle or high; optionally *algorithm* specifies the specific algorithms desired to use for the security protection;
- *from to* is to specify the areas outside the *from to* sets are to be protected against, for example, *from* (128.1.*.*) *to* (156.68.*.*) indicates the transmission going outside sub-domain 128.1.*.* before entering into sub-domain

[2] Attribute with [ ] is optional and can be specified to be empty.

[3] Each attribute will be specified as a finite set, which can be specified as wildcard, list of values, ranges or optionally preceded by not to express all but some etc. e.g. ip addresses, ip address ranges, or dns names can be used to specify particular nodes.

156.68.*.* needs to be protected. The enforcement agents, which form SA to protect the traffic, would be the border security gateways of the sub-domains. We use ***fromA toA*** to denote the agents of the *from to* sets. Also, *fromA* and *toA* can be used to determine the protection area that is between the two nodes. If a set only contains one member that is address of a specific node, then the specified node will be the enforcement agent. Optionally, *except-list* is to specify the nodes that are allowed to access content rather than being secured against.

The above definition of requirement specification is capable of expressing four security requirements analyzed in section 2.2 and is extensible for new security requirements in the future.

Definition 6: The implementation level IPSec security policies check various header fields to select a packet. Therefore, the implementation level security policies have selectors of all possible header fields of an IP packet in the condition part as follows:

src-addr, dst-addr, src-port, dst-port, proto, ah-hdr, esp-hdr, TOS, ah-next-hdr, ah-spi, etc.

and have the following action types and associated parameters in the action part:

deny
allow
ipsec-action (sec-prot, algorithm[4]*, mode, from, to)*

In the condition part, each S_i is a finite set used to match the header fields of a packet to the policy.

In the action part, each t_j is a single value except *algorithm:*

- *sec-prot* specifies either ah or esp;
- *algorithm* specifies all possible algorithms for ISAKMP to negotiate;
- *mode* specifies either transport or tunnel*;*
- *from to* specify two nodes to build an SA.

Implementation policies are to instruct certain security devices to set up specific SA and perform specific operations on the passing traffic. Therefore, in the definition, deterministic values will be assigned for the attributes of ipsec-action except *algorithm* of which multiple values can be specified for ISAKMP negotiation. Multiple protections with different security protocols can be specified separately in multiple ipsec-actions. Other kinds of representations would be equivalent and can be generalized by this definition.

[4] We use *algorithm* to also abstract other related attributes like *key-length* etc.

3 IPSec Policy Correctness and Conflict Detection

We call a set of implementation policies regarding a certain traffic flow ***correct*** iff the set of policies satisfies the set of requirements regarding the flow. We call a set of implementation policies regarding a certain traffic flow ***conflicting*** when the set of policies together does NOT satisfy all of the requirements regarding the flow, with the requirement satisfaction as defined below.

3.1 Security Requirement Satisfaction

3.1.1 Access Control Requirement Satisfaction

$Notation:$

- $path(x, f)$: node *x* is on the path of flow *f*
- $drop(x, f)$: node *x* drops flow *f*
- $R \leftarrow Q$: *R* is true if *Q* is true

Definition 7.1: $flow\, f \rightarrow deny$ is satisfied iff any node on the path of the flow *f* drops all packets of *f*.

$$R_{11}: flow\, f \rightarrow deny$$
$$R_{11} \leftarrow \exists x Path(x, f) \Lambda Drop(x, f)$$

Definition 7.2: $flow\, f \rightarrow allow$ is satisfied iff none of node on the path of the flow *f* drops the flow.

$$R_{12}: flow\, f \rightarrow allow$$
$$R_{12} \leftarrow \neg \exists x Path(x, f) \Lambda Drop(x, f)$$

3.1.2 Security Coverage Requirement Satisfaction

$Notation:$

- $\sec - link(f, x, sfunc, strg)$: Traffic flow *f* is protected by a security function *sfunc* with strength *strg* on the link from node *x* to the next node on the path.
- $\sec - node(f, x, sfunc, strg)$: Traffic flow *f* is protected by a security function *sfunc* with strength *strg* against the node *x*.

Definition 8: *flow f* → *enforce (sec-func, strength, from, to, except-list)* is satisfied iff

1) all the links within the protection area are secured against by all security functions specified in *sec-func* with strength equal or greater than the level specified in *strength*; and
2) all the nodes within the protection area are also secured against with the security functions and strength except the nodes in the *except-list*.

R_2 : *flow f* → *enforce(sec-func, strength, from, to, except-list)*[5]

$$R_2 \leftarrow \text{sec\_}links(f, \text{sec}-func, strength, fromA, toA) \ \& \ \text{sec\_}nodes(f, \text{sec}-func, strength, fromA, toA, except-list)$$

$$\text{sec\_}links(f, \text{sec}-func, strength, fromA, toA) \leftarrow \forall x Path(x, f) \Lambda (fromA \le x < toA) \Lambda \forall_{sfunc \in \text{sec}-func} \ \text{sec}-link(f, x, sfunc, strg) \Lambda (strg \ge strength)$$

$$\text{sec\_}nodes(f, \text{sec}-func, strength, fromA, toA, except-list) \leftarrow \forall x Path(x, f) \Lambda (fromA < x < toA) \Lambda (x \notin except-list) \Lambda \forall_{sfunc \in \text{sec}-func} \ \text{sec}-node(f, x, sfunc, strg) \Lambda (strg \ge strength)$$

From the definition, we can see a SCR contains protection requirements for every link and node in the specified area. If one requirement has some element requirements such that the requirement is satisfied iff all its elements are satisfied, we call the elements the ***sub-requirements*** of the requirement. If one requirement has property that, the satisfaction of which will be determined as a single unit, we call it an ***atomic requirement***. For example, a sub-requirement for a certain link or node with a certain security function is an atomic sub-requirement of a SCR, since it is either satisfied if the link or node is protected accordingly or violated otherwise and there is no partial satisfaction or violation. The verification of a non-atomic requirement can be accomplished by verifying if each of its atomic sub-requirements is satisfied.

3.1.3 Content Access Requirement Satisfaction

[5] *from to* might be specified as sets or sub-domains etc. *fromA toA* as denoted in definition 5 are two specific nodes to determine the protection area, which is between the two nodes. In addition, we did not explicitly list *algorithm* here since satisfaction of *algorithm* can be verified in a similar way.

Definition 9: *flow f, sec-func against-list* → *deny* is satisfied iff all nodes in the *against-list* can access the traffic content that is not secured by any of the function in *sec-func.*

$$R_3 : flow f, \text{sec}-func \quad against-list \rightarrow deny$$

$$R_3 \leftarrow \forall x Path(x, f) \Lambda (x \in against-list) \Lambda \forall_{strg} \forall_{sfunc \in \text{sec}-func} \neg \text{sec}-node(f, x, sfunc, strg)$$

It is composed of atomic sub-requirements of access requirement of each node in the *against-list.*

Although *flow f, sec-func against-list* → *allow* is satisfied unconditionally, it can be used in conjunction with *flow f, sec-func against-list* → *deny* to specify some CAR. For instance, a requirement that all nodes except *SG1* need to access content of flow *f* can be specified as:

flow f, encryption against SG1 → *allow*
*flow f, encryption against ** → *deny*

3.1.4 Security Association Requirement Satisfaction

Notation:

➢ $peer(f, x_1, x_2)$: node *x1* and *x2* form a SA peer for flow *f.*

Definition 10: *flow f, SA-peer1 SA-peer2* → *deny* is satisfied iff none of node in *SA-peer1* set up SA with any of node in *SA-peer2* for flow *f.*

$$R_4 : flow f, SA-peer1 \quad SA-peer2 \rightarrow deny$$

$$R_4 \leftarrow \forall x \forall y (x \in SA-peer1) \Lambda (y \in SA-peer2) \Lambda \neg peer(f, x, y)$$

It is composed of atomic sub-requirements of peer requirement of each pair specified in *SA-peer1 SA-peer2* sets.

Although *flow f, SA-peer1 SA-peer2* → *allow* is satisfied unconditionally, it can be used in conjunction with *flow f, SA-peer1 SA-peer2* → *deny* to specify some SAR in a similar way as exemplified in the last subsection.

3.2 IPSec Policy Processing

The IPSec policies installed in security gateways will be consulted in processing either inbound or outbound traffic. As specified in [1], the IPSec policy will be processed at a particular node as follows:

- For inbound traffic, if the action in the policy for the traffic is *deny*, then the traffic is dropped; if *allow*, then forward the traffic. If it is the destination of the outer tunnel, then it needs to de-apply the security function. For tunnel mode, it also decapsulates to remove outer header before forward;
- For outbound traffic, if the policy is with action of *ipsec-action (sec-prot, alg, mode, from, to)*, then the node will apply the corresponding security function. For tunnel mode, it also encapsulates an outer header with new source and destination address to be addresses of tunnel entry and tunnel exit nodes. Finally it will forward the packets;
- All the forwarding is only based on destination address of outer header.

Traffic sometimes might be sent back and forth because of tunnel interaction as illustrated with figure 2 in the introduction part.

3.3 Policy Verification Algorithm

As we illustrated in the introduction, regionally enforced policies together might interact or cause conflicts. It is important for a policy management system to be able to detect those conflicts. A ***Policy Verification/ Conflict Detection Problem*** can be defined as follows: Given a set of security requirements regarding a particular traffic flow $\{\mathrm{Re}q_1, \mathrm{Re}q_2, ..., \mathrm{Re}q_K\}$ and a set of implementation policies regarding the flow $\mathrm{Im}p_1, \mathrm{Im}p_2, ... \mathrm{Im}p_N$ that are installed in nodes along a linear path with N nodes $Node_1, Node_2, ... Node_N$. Verify the correctness of the set of implementation polices.

From the correctness definition, we need to verify the satisfaction of the requirements thus to verify satisfaction of all their atomic sub-requirements. Therefore, we need to obtain information as follows:

- Access Control Requirement: we only need to verify whether any node has policy to deny the traffic flow.
- Security Coverage Requirement: we need information about security function and strength for each link and node to verify if required areas have been covered with appropriate functions and strength.
- Content Access Requirement: we need information about whether certain security function is applied against certain nodes to verify whether the nodes can access content or not. The information needed by CAR is a subset of information needed by SCR.
- Security Association Requirement: we need information about SA peers specified by the policies to see whether all of them are allowed by requirements.

Two points need to be emphasized before proceeds to the verification algorithm: 1) Transmission at a link or a node is subject to protection of all the security functions that are applied but not de-applied yet when the traffic travels to the link or node. 2) Since traffic may travel to one link or node more than once, the security protections are only the weakest one of all the trips to the link or node. To illustrate the two points, we use an example as shown below.

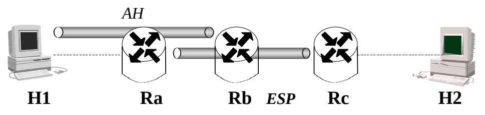


Figure 3: Calculating Security Coverage

In this simple five nodes linear topology, traffic is to be sent from H1 to H2. First traffic is tunneled to Rb with authentication. Then before it reaches tunnel exit Rb, it is tunneled by Ra and send to Rc with encryption. Since authentication function is applied at node A and has not been de-applied yet at tunnel from Ra to Rc, the link Ra-Rb and Rb-Rc are subject to protection of both authentication and encryption. Then the encryption will be de-applied at node Rc and the traffic will be sent back to Rb along Rb-Rc link with protection of authentication only. Then Rb will de-apply the authentication function. At this moment, no any security function still applies such that the third time the traffic travels the link Rb-Rc under no protection, which is the weakest one of the three trips.

Based on the above analysis, we know a packet may be traveling in many different ways rather than simply hop-by-hop ahead because of IPSec processing. However, the processing at each node, which was described in section 3.2, is with fixed number of operations and can be easily simulated. In the verification algorithm, we will simulate IPSec policy processing to follow a packet's trip step-by-step from source to destination as well as record security protection and related information of each link and node at every step. As described in section 3.2, tunnel mode processing will encapsulate an outer header with addresses of tunnel entry and tunnel exit as the new source and destination address. We will use a stack to simulate the nested header such that new header will be pushed into the stack upon encapsulation and popped out upon decapsulation. In addition to header information, we also push security protection information associated with the tunnel into the stack when it is applied and pop it out when it is de-applied. Therefore, the security protection for a link or node is all those security functions on the stack at the moment that the traffic comes to it.

In the following, we will present an algorithm to follow the packets' trip and collect necessary information along the path.

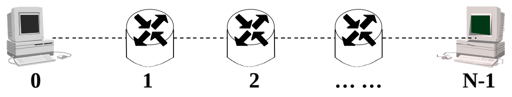


Figure 4: A linear path with N nodes

There are *N* nodes on the path. *Imp[n]* is the policy of the corresponding node *n*. We need to use action information of each policy to calculate required information while action part can be represented by:

Imp[n].action = deny | allow |
ipsec -> (sec-prot, alg, mode, from, to)

in which *ipsec* point to a link list of one or more ipsec actions. There is only one action type in one policy though we allow multiple actions with the same type and different parameters in one policy.

We also need the following data structures in the algorithm:

- *sec_link[N]* is an array of link list, each of which is to mark what security protection covers link from $Node_n$ to $Node_{n+1}$. One link might be subject to multiple protections, *e.g. sec_link[n] = esp cast -> ah hmac5 .*
- *sec_node[N]* is an array of link list, similar to the above, each of which is to mark what security protections are against the corresponding node.
- *SA_peer[M][2]* is used to record all SA peers in the policies.
- *A stack S* is used to store series of *(sec-prot, alg, from, to)* and simulate encapsulation/ decapsulation, security function application/ de-application. *top = 0* initially and the destination address of encapsulated outer header will always be destination address *to* on the top of the stack *S*. Keep in mind that the security protection of a link or node at the time of packet traveling to it is all *sec-prot alg* on the stack and the real security protection of a link or node is only the weakest one among all the trips to the link or node.

Algorithm 1: Policy Processing Along The Path

```
top = 0      // stack is empty
m = 0
sec_link[N] = sec_node[N] = null  // no travel yet
n = 0      // from the first node

while  ( n < N)       // at node n
{
   // inbound processing
   if (Imp[n].action == deny)
        report and exit

   // calculate what the node n is secured against
   if ( top == 0)
        sec_node[n] = 0  // no protection
   else
   {    //decapsulates first if tunnel exit
        while ( S[top].to == Node[n])
        {     pop (S)
              top - -
        }

        // current protections against the node are all
        // those on the stack and the real one is always
        // the weaker one of this trip and previous trips.
        if (protection in stack is weaker than in
                        sec_node[n])
         insert all (sec-prot, alg) on stack into sec_node[n]
   }

   // Outbound processing
   // encapsulate and record SA information for ipsec
   if ( Imp[n].action == ipsec)
   {    push (sec-prot, alg, from, to) series into the stack
        record all pairs ( from, to) in SA_peer
   }
   // send packet out; record protection for each link
   if (top == 0)
   {     sec_link[n] = 0 // no protection
         n ++            // forward
   }
   else if ( S[top].to > Node[n])       // forward
   {  if (protection in stack is weaker than in sec_link[n])
         insert all (sec-prot, alg) on stack into sec_link[n]
      n ++
   }
   else      // backward and travel the link n to n-1
   {if (protection in stack is weaker than in
              sec_link[n-1])
      insert all (sec-prot, alg) on stack into sec_link[n-1]
     n - -
   }
}
```

We call a move from one node to another node a ***step***. Since traffic might be tunneled back to visit some nodes multiple times, there might be more steps than nodes on the path. The above algorithm simulates the processing at each node and follows every step of the trip. If the total number of steps for the trip is *L*, the time complexity will be *O(L)*. Now we first analyze the number of the total steps *L* at worst case.

With a group of nested tunnels, the traffic might be sent back and forth several times. If the group of tunnels start from node *n*, then after the traffic being carried by all those tunnels and come back to the exit of the first tunnel, it will at least have moved ahead one hop to *n+1*[6]. From node *n*, there are at most *(N-n)* nested tunnels in this group. Each tunnel can span at most *(N-n)* nodes. For each tunnel, the steps taken to carry traffic back and forth then is at most $2 \times (N-n)$. Therefore, for every forward move from *n* to *n+1*, there are at most $2\times(N-n)\times(N-n)$ steps. Then, the total steps are at most $\underset{n}{\ } 2\times(N-n)\times(N-n)$ that is $O(N^3)$.

The above analysis is based on assumption that there is no reverse tunnel to carry the traffic. If there is reverse tunnel, there is possibility that a packet may be sent back and forth in an infinite loop because of inappropriate policy setting. The algorithm should detect and report error for situations where the policies make traffic travel too long time.

With the collected information by the above algorithm, we can verify each requirement one by one. As analyzed previously, we have four types of requirements: ACR, SCR, CAR and SAR, which specify the requirements regarding access control, security coverage along the path, content accessibility of certain nodes on the path and security associations for the protection along the path. The number of ACR and CAR will be proportional to the number of nodes on the path. The number of SCR and SAR will be at most $O(N^2)$ since SCR might requires to cover from any node to any other node and SAR might requires relationship between any node and any other node. Then the total number of relevant requirements for a specific flow is at most $O(N^2)$. Each requirement can be verified in *O(N)* with the collected information. Therefore, the total time to verify all requirements is at worst $O(N^3+N^3)=O(N^3)$.

[6] Actually, if there is nesting, then the first tunnel must at least span two links. For simplicity, we assume it only moves one hop each time.

4. IPSec Policy Conflict Resolution

Once we found conflicts, next step is to find ways to resolve them. Ideally, we want to satisfy all the security requirements. However, there may be circumstances that in no way can all requirements be simultaneously satisfied. Violation of any requirement will cause some damage. If there has to be some damage, then our goal is to find a set of policies that minimize the possible damage. We will first discuss the mapping from the requirements to implementation policies.

There are *except-list* in SCR such that the nodes in *except-list* are not necessarily secured against. In an example shown below, a SCR requires to protect certain traffic from H1 to H2. Between H1 and H2, there are Ra and Rb that are in the *except-list*. Then all the following configurations satisfy the SCR (we assume every tunnel is with appropriate security function and strength):

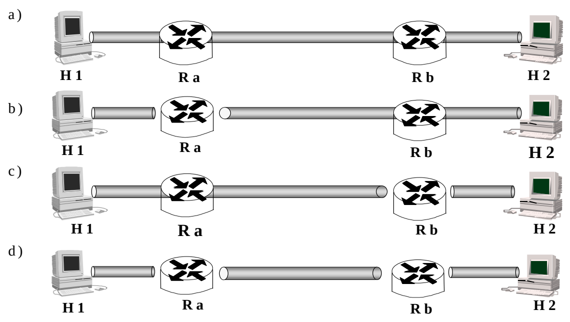


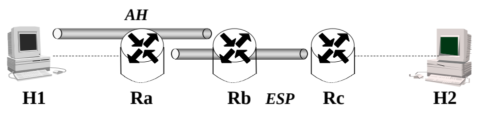
Figure 5: Four different configurations that satisfy one SCR

From the figure 5, we can see that one SCR can be implemented by one tunnel or chain of tunnels that connect to each other in which the connecting nodes can de-apply the security and access to content before apply the security function again for the next tunnel.

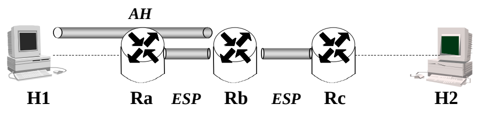
Sometimes it is advantageous to build several chained tunnels to implement one SCR rather than one direct tunnel for the preference of other requirements. For instance, in the above figure, if Rb is required to examine the content, then Rb has to be a connecting node of tunnels to be able to access the content. In another example, Rb also requires to examine content. Additionally, H1 and Rb is not suitable to set up SA as specified in one

SAR, while H1 and Ra, Ra and Rb are allowed to build SA, then we can use Ra, Rb as connecting nodes to build three SAs rather than one for the SCR.

In addition to satisfaction of CAR and SAR, another reason to use several tunnels instead of one is to resolve overlapping as illustrated below.



a) Overlapping causes SCR violation



b) Breaking the lower tunnel up eliminates the overlap

Figure 6: Breakup to resolve overlapping

In the example shown in the figure 6, as we explained and calculated in section 3.3, the configuration will make traffic to be sent back from Rc to Rb, then sent in clear from Rb ahead which may violate the SCR for the link Rb-Rc. If the lower tunnel is broken up as shown in b), then the traffic will not be sent back and the link Rb-Rc will be protected by the tunnel from Rb to Rc. The reason that we do NOT build additional tunnel from Rb to Rc to resolve lack of security coverage caused by overlapping is that the additional tunnel does not compensate the security coverage. In the above example, even though we can build additional ESP tunnel from Rb to Rc, the traffic sent back from Rc to Rb is still not encrypted.

Sometimes we may prefer to break up at the nodes that not in the *except-list*, which violates SCR for this node but may be in favor of other requirements. The problem is then to evaluate the tradeoff and find one configuration that results in minimal total damage. Since each tunnel can be implemented by an IPSec policy with ipsec-action, the policy resolution problem becomes that, from all set of policies of alternative chained tunnel configurations, choose one that minimizes damage caused by violation of requirements. We can resolve ACR separately since IPSec policies with deny or allow action can directly implement ACR and the resolution of ACR conflicts will become the resolution among different ACR themselves. Therefore, in the following, we only focus on resolution problem for SCR, CAR and SAR requirements.

We quantify the damage associated with any requirement violation as a non-negative value called ***penalty***. If one requirement is not atomic, there might be different penalties associated with violation of each of its atomic sub-requirement. For example, a CAR may specify that *Ra*, *Rb* need to examine content while the penalty that *Ra* is not able to access content might be much greater than that of *Rb*.

In the ***IPSec Policy Resolution/Optimization Problem***, we are given a set of SCR, CAR or SAR requirements $\{\mathrm{Re}\,q_1, \mathrm{Re}\,q_2, ..., \mathrm{Re}\,q_K\}$ that equals an atomic sub-requirement set of M members (M ≥ K) $\{AR_1, AR_2, ..., AR_M\}$ and a set of penalties $\{Pnlt_1, Pnlt_2, ..., Pnlt_M\}$ when the corresponding atomic requirement is violated as well as a linear topology with N nodes $Node_1, Node_2, ... Node_N$. A subset of requirements might be SCR requirements $\{SCR_1, SCR_2, ..., SCR_{K_1}\}$ where $K_1 \le K$. Each SCR_i requires certain security function to cover area from $fromA_i$ to toA_i. Each SCR_i can be implemented by building one single tunnel or a chain of tunnels that connecting at some nodes between $fromA_i$ and toA_i [7]. We can represent each alternative configuration for each SCR as a subset of node set $\{x : fromA_i < x < toA_i\}$ such that the subset of the nodes adding $fromA_i$ as the first and toA_i as the last can form a chain of tunnels and SAs. The tunnels built for all SCRs together may cause violation of some CAR, SAR or SCR (e.g. because of overlapping). Therefore, the problem is to find K_1 subsets, where each subset i can be selected from set $\{x : fromA_i < x < toA_i\}$, such that all tunnels built based on the subsets together will cause minimum total penalty. With an optimal set of subsets, we can easily translate them into a set of implementation policies to be installed on the path. A zero penalty indicates that all requirements are satisfied by the optimal configuration. The following shows a policy resolution problem example with three overlapping tunnels:

[7] We will build one tunnel or chain of tunnels with the required security functions and strength. Since security function and strength can be easily mapped to certain security protocol and algorithms in implementation, in the resolution problem, we mainly focus on tunnel configuration layout, i.e. to determine the connecting nodes of chained tunnels.

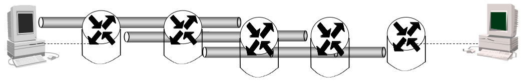


Figure 7: A group of three tunnels

In this scenario, traffic will be sent from H1 then Ra then Rb then to Re which will decapsulate and send it back to Rd which will decapsulate again and send it back to Rc which then forward traffic ahead in clear. To resolve lack of security coverage or violation of other requirements, the first tunnel may be broken at node Ra or Rb, the second tunnel may be broken at node Rb or Rc and the last tunnel may be broken at node Rc or Rd. Totally there will be $2^2 \times 2^2 \times 2^2 = 64$ possible configurations including the original one.

If there is K_1 SCRs and the members of each subset can be selected from a set with *J* nodes, then the total number of possible configurations are $2^J \times 2^J \times ... \times 2^J = 2^{J \times K_1}$. The solution space can be expressed as $(x_1, x_{2,} ..., x_{J \times K_1})$ where $x_i = 1 | 0$, value 1 represents breakup at a certain node while 0 represents not. However, most time we do not need to break one tunnel up if the tunnel already satisfies the requirement. We only test different breakup configurations when violation occurs. Therefore, we can start from building one tunnel for each SCR first. If one tunnel plan cause no conflict, then it is perfect. Otherwise we will try different breakup plans for those problematic tunnels to search for optimal configurations. To find those tunnels that need to be resolved, we can first define the following tunnel relationship types:

- ***Two Tunnels*** with tunnel end points $(from_i, to_i)$ and $(from_j, to_j)$. Tunnel *i* is ***containing*** *j* iff $from_i < from_j < to_j < to_i$.
- ***Two tunnels*** with tunnel end points $(from_i, to_i)$ and $(from_j, to_j)$ are ***overlapping*** iff $from_i < from_j < to_i < to_j$.
- ***Two tunnels*** are ***nesting*** with each other iff they are either containing or overlapping.
- ***A group of tunnels are nesting*** iff every tunnel is nesting with at least one other tunnel in the group.
- ***Two groups of tunnels are disjoint*** iff none of tunnel in one group is nesting with any tunnel in the other group.

Nested tunnels need to be considered as a whole in seeking optimal breakup plans because breaking one up may cause additional overlapping among nested tunnels. Those disjoint groups can be considered separately because any kind of breakup for one group will not have any effect on the other group. Having these in mind, we can group those tunnels with corresponding requirements and resolve each group separately. We only try different breakup plans for those groups that caused violations and leave others as they are. Resolution for some groups might be very easy. For instance, if one group is only with one tunnel that only violated one CAR of one node Ra, then the only work to do is to compare and make a decision whether or not break the single tunnel up at the node Ra. However, at the worst case, given a group of tunnels, we may need to test all possible configurations before an optimal one can be determined.

Among those optimization problems with solution as a n-tuple $(x_1, x_{2,} ..., x_n)$, ***backtracking*** [7] is a commonly used algorithm. The basic idea of the *backtracking* algorithm is to continuously build and test partial vector $(x_1, x_{2,} ..., x_i)$ to see if it can possibly lead to an optimal solution. If not, then all possible values of latter part of vector $(x_{i+1}, x_{i+2,} ..., x_n)$ can be ignored entirely. The process can be also illustrated by constructing a solution tree. A bounding function is used to test whether a branch has any chance to lead to an optimal solution and a node is killed immediately without generating any of its children when it is found to be with no chance to success. Then it will go back to an upper layer node to construct other branches of the solution tree. Depth first node generation with bounding functions is called *backtracking*.

When we search for optimal configurations for nested tunnels, with *backtracking* algorithm, we can calculate the penalty with partial configuration to help to kill non-optimal breakup plans at its earliest stage. The verification algorithm developed in section 3 can be easily modified to calculate the total penalty for a set of policies. The idea of using *backtracking* here is that if some portion of a configuration already cause penalty greater than a so-far-minimal penalty, then we will not investigate any other portion of this configuration further, which may greatly reduce the number of configurations that are really calculated.

We can use the example in figure 7 to show how we may kill one branch earlier. In this example, when verification algorithm goes to the second layer, it finds that, no matter how the third layer tunnel is configured, the traffic will be sent from Rd to Rc then send in clear from Rc, which may have violated certain atomic SCR with big penalty. We also can calculate how many penalties there are for violation of CAR and SAR at each stage. If the penalty already exceeds the penalty of a known configuration, it will be killed for non-optimality without investigating deeper layers. A simple backtracking algorithm can be written as follows:

Algorithm 2: Backtracking for Policy Resolution

```
// Backtracking can be implemented by recursively call
// the following procedure. The former k-1 elements x(1),
// x(2), … x(k-1) have been assigned when it comes to k

// initial penalty and configuration
penalty = init_pnlt;  conf = init_conf

void POL_RES(k)
{    if (k <= K)    // the length of the solution vector is K
         for every possible value of x(k)
            if  ( (p1=penalty_comp(x(1), x(2), …, x(k)) )<
                   penalty )
               if   ( k == K)    // it is a complete solution
                     penalty = p1       // update the penalty
                     conf = x(1), x(2), …, x(k)
               else
                    POL_RES(k+1)    // test next layer
}
```

The complexity of the above backtracking algorithm mainly depends on two factors: 1) the time to calculate the penalty; 2) the number of branches that not being killed. We may greatly improve efficiency if we initially have a known configuration with a small penalty that can help to kill more branches earlier. Combining heuristic and random mechanisms, we may first choose an initial configuration with a small although not smallest penalty. We can first sort the set of requirement penalties and find those largest penalties. Then we randomly choose dozens of configurations that avoided the large penalty violation. For example, if one largest penalty caused by violation of a SAR, then we select initial value only from the configurations that do not build the undesirable SA. Then from the randomly selected dozens of configurations, we use penalty calculation algorithm to find out the one with smallest penalty to be as our initial penalty.

Although *backtracking* can vary greatly in time complexity for different problem instances, for a lot instances in large scale, backtracking indeed can find out solution in very short time. Monte Carlo [7] method can be used to estimate the efficiency of the backtracking algorithm for a specific instance. Besides *backtracking* algorithm, other algorithms like ***branch and bound*** [7]**,** ***genetic algorithm*** [8] etc. can also be used for policy optimization problem. *Genetic algorithm* normally could get a good solution very fast but can not guarantee the optimality of the solution.

5. Celestial – An Inter-domain Security Management System

In our conflict detection and resolution algorithms, we need information about requirements and implementation policies as well as the route path for the flow. For intra-domain policy management, the required information might be obtained from a central policy server of the domain. For inter-domain communication, an inter-domain security management system like Celestial system [9] is needed to collect and manage the information.

Celestial system aims to provide reliable and scalable end-to-end security services using multiple distributed security mechanisms. In Celestial system, Security Management Agent (SMA) is to sit in management plane of any SMA-enabled node (switch, router, security gateway etc.) and is responsible for coordinating all security-related activities on a network system. Inter-Domain SMA Coordination Protocol (ISCP) [10] provides the transport function for security service negotiation and reservation in order for the Celestial system to gather relevant information and manage security services end-to-end. In Celestial, security context establishment is done in two phases. In the discovery phase, the application’s service request is distributed along the communication path and the service capabilities/policies of the nodes along the path are collected. Then the SMA who is authoritative to the receiver will determine an optimal configuration plan using certain policy resolution algorithm based on the collected information, and then invokes the reservation phase that distributes assignments to the nodes selected for providing the security services. Refreshing messages are periodically sent to collect updated policy and path information and distribute new reservation/ assignment information, which helps the system dynamically adapt to changes and maintain adequate security service for users.

6. Related Work

Another research on end-to-end IPSec policy management is Policy Based Security Management (PBSM) system [11,12] developed in BBN. PBSM is a distributed systems with Policy Servers (PS) that can manage IPSec security policies for multiple domains. The system answers end-to-end security service query by merging policies among Policy Servers (PS). Along with PBSM, they developed Security Policy Specification Language (SPSL) [3]. However, the potential conflicts and topological interactions have not been analyzed in their work. In addition, without distinguishing the requirement level and implementation level security policies, SPSL itself can not ensure the correctness of policy specification.

The needs of separating high-level requirements and low-level policies were addressed in [2,6]. Our work applied the concepts to a specific policy service by defining IPSec security requirements in a high level. Some recent work [13,14] analyzed two types of conflicts: one is co-existence of both positive and negative policies, which can be detected by checking syntax; the other one is application specific conflicts. In this research, we analyzed IPSec specific conflicts caused by topological interaction etc. The developed algorithm can detect all possible conflicts among IPSec policies in a distributed environment. The consistency analysis of security policies in [15] focuses on access control policy while our work focuses on topologically interacted IPSec policies.

7. Conclusion and Future Work

With the increasing scale the IPSec is deployed in, a policy management system is demanded to systematically specify and manage various IPSec policies. To support automation of policy specification and conflict detection for policy management, in this research, we first defined security requirements in a high level. Then, we developed mechanisms to systematically detect and resolve IPSec policy conflicts.

In this research, we focus on conceptually centralized conflict detection and resolution in which we resolve policies when all relevant information is collected. Next step we will work on a decentralized collaboration model in which conflicts can be detected in a distributed manner. Furthermore, QoS specific policy conflict issues will be researched.

REFERENCE

[1] S. Kent, R. Atkinson, *Security Architecture for the Internet Protocol*, RFC-2401, Internet Society, Network Working Group, Nov. 1998

[2] J. D. Moffett and M. S. Sloman, *Policy Hierarchies for Distributed Systems Management*, IEEE Journal on Selected Areas in Communication, vol. 11, pp. 1404-1414, 1993

[3] M. Condell, C. Lynn, J. Zao, *Security Policy Specification Language*, Internet Draft, <draft-ietf-ipsp-spsl-00.txt>, March, 2000

[4] J. Jason, *IPsec Configuration Policy Model*, Internet Draft <draft-ietf-ipsp-config-policy-model-00.txt>, March, 2000

[5] R. Pereira, P. Bhattacharya, *IPSec Policy Data Model*, Internet Draft <draft-ietf-ipsec-policy-model-00.txt>, Feb. 1998

[6] J. D. Moffett, *Requirements and Policies*, Position paper for Policy Workshop 1999

[7] E. Horowitz, S. Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press Inc.,1978.

[8] M. Gen, R. Cheng, *Genetic Algorithms & Engineering Optimization,* Wiley-Interscience, 2000

[9] C. Xu, F. Gong, I. Baldine, C. Sargor, F. Jou, S. F. Wu, Z. Fu, H. Huang, *Celestial Security Management System*, DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings, Volume: 1, 1999, Page(s): 162 -172 vol.1

[10] Z. Fu, H. Huang, T. Wu, S. F. Wu, F. Gong, C. Xu, I. Baldine, *ISCP: Design and Implementation of An Inter-Domain Security Management Agent (SMA) Coordination Protocol*. Proceedings, NOMS 2000, Pages 565-578.

[11] L.A. Sanchez, M.N. Condell, *Security Policy System*, Internet Draft, <draft-ietf-ipsec-sps-00.txt>, Nov. 1998

[12] J. Zao, L. Sanchez, M. Condell, C. Lyn, M. Fredette, P. Helinek, P. Krishnan, A. Jackson, D. Mankins, M. Shepard, S. Kent, *Domain Based Internet Security Policy Management*, DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings ,1999, Pages: 41 -53 vol.1

[13] E.C. Lupu and M. Sloman. *Conflict Analysis for Management Polcies.* Proc. 5th IFIP/IEEE International Symposium on Integrated Network Management, pages 430-443, 1997

[14] E.C. Lupu and M. Sloman. *Conflicts in Policy-Based Distributed Systems Management.* IEEE Transaction on Software Engineering. Vol. 25, No. 6, pages 852-869, Nov./Dec. 1999

[15] L. Cholvy and F. Cuppens. *Analyzing Consistency of Security Policies.* IEEE Symposium on Security and Privacy, 1997, Proceedings, Pages: 103-112