Automatic Generation of IPSec/VPN Security Policies In an Intra-Domain Environment

Zhi (Judy) Fu[1] and S. Felix Wu[2]

[1]Networks and Infrastructure Research Lab, Motorola, 1301 East Algonquin Rd. MS IL02-2246, Schaumburg, IL 60196, USA.
Email: jfu@labs.mot.com (Was Computer Science Dept., North Carolina State University)
[2]3057 Engineering II, Computer Science Dept., University of California, Davis, One Shields Avenue, Davis, CA 95616, USA
Email:wu@cs.ucdavis.edu

Abstract: IPSec [1] policies are widely deployed in firewalls or security gateways to protect information property. The security treatment (e.g. deny, allow or encrypt etc.) of all inbound or outbound traffic will be determined by the security policies, and thus it is critical for policies to be specified and configured correctly. IPSec policies are manually configured to individual security gateway in current practice, which could be very inefficient and error-prone. In this research, we focus on two questions: 1) How to ensure policy correctness? 2) How to systematically specify correct policies instead of manually configuring? Apparently, policies are correct if they do what they are wanted to do. However, there is vague relationship between what they are wanted and what they really do. In our research, we clearly defined a higher level policy, called security requirement, and clearly defined their satisfaction. Therefore, policies are correct only if they satisfy all requirements. Furthermore, we designed algorithms to automatically generate correct policies given security requirements. People can specify their requirements at a high level without concerning specific low level parameters, and then correct low level policies will be automatically generated. The automation can not only save tremendous administrative labor but also guarantee the policies are correct.

Keyword: Security Policy Management, IPSec Policy, Security Policy Specification, Security Requirement, Firewall

1. Introduction

IPSec (Suite of protocols for IP layer Security) [1] policies are widely deployed in firewalls or security gateways to restrict access or selectively enforce security operations. We first illustrate a typical scenario of intra-domain communications for a large distributed organization.

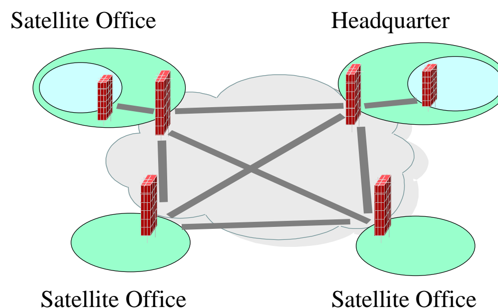


Figure 1: Scenario for intra-domain site-to-site communications

In figure 1, a large organization is composed of multiple distributed sites, which can communicate with each other through VPN tunnels. Each site can have firewalls with specific security policies to protect their property. Furthermore, some sub-domains, like financial department etc., can have their own firewall with their specific policies to protect their sensitive data.

The security treatment (e.g. deny, allow or encrypt etc.) of all inbound or outbound traffic will be determined by the security policies, and thus it is critical for policies to be specified and configured correctly. IPSec policies are manually configured to individual security gateway in current practice, which

could be very inefficient and error-prone. For a large distributed organization with complex hierarchy and many security gateways or firewalls, there will be excessive amount of work to specify and configure security policies for each node. Small or subtle errors in the process may expose massive holes in the overall security of the network. Furthermore, as we will analyze in next section, even if each policy is correct individually, the interactions among policies might cause unexpected security breach, which is very difficult to check even with careful and experienced administrators. Although various policy issues attracted a lot of attention [2,3,4], two important problems have not been carefully studied so far: **1) How to ensure correctness of security policies? 2) What are ways to systematically specify and distribute correct policies instead of manually configuring each node?**

Policies are correct if they do what they are wanted to do. Current IPSec specification can be so specific that it is hard to understand what it is really intended for. It is realized in IPSec security policy working group of IETF that we need a higher-level policy language with well-defined and clear semantics. In our research, we clearly defined a higher level policy, we call security requirements, whose functions can be clearly understood and rigorously proven. The low-level policies are correct only when they satisfy all requirements. In addition, we developed a system, in which people can specify their security requirements at a higher level to a central policy database, and correct low-level policies will be automatically generated and distributed to appropriate nodes to enforce. The automation can not only save administrators tremendous labor but also guarantee the correctness of low-level policies.

In summary, this research presents the following contributions:

- We carefully analyzed potential problems in policy enforcement due to interactions;
- We formally defined implementation-independent security requirement at a higher level;
- We clearly defined correctness of policies;
- We developed scalable algorithms to systematically generate policies to satisfy desired security requirements.

The rest of the document is organized as follows. Section 2 analyzes potential policy problems and discusses requirement for policy management, which motivated the definition of security requirements at a higher level in section 3. Then section 4 develops three different approaches to determine policies based on requirements and analyzes their pros and cons. Finally section 5 summarizes the research and outlines future directions.

2. IPSec Policy Management: Problem Analysis and Requirement Statement

2.1 A Brief Overview Of IPSec And IPSec Policies

IPSec, standardized in IETF, defines a suite of protocols such as ESP, AH, ISAKMP, IKE etc., which forms a security architecture to protect traffic at IP layer. The basic functions of IPSec are access control and selective security enforcement, i.e. only selected IP packets are allowed to pass and only selected IP packets are protected with specific security function. The selection of IP packets and specification of security actions are defined in IPSec policies. An IPSec policy consists of two parts: condition and action, i.e. if condition is met, then the action will be taken. The condition in a policy maps values with header fields in selecting an IP packet. For example, if (src=A, dst=B) is in condition part of a policy, then all IP packets with source address A and destination address B in IP header will be selected. There are generally three actions: deny, allow, ipsec_action, meaning the selected packet should be dropped, allowed to pass or applied with certain security function as specified in ipsec_action. Typically the attributes of an ipsec_action include security protocols (encryption or authentication), algorithms, mode (we explain it below), the node "from" at which the security enforcement starts and the node "to" at which the security enforcement ends. Combining condition and action, a simple representation of policy can be like (src=A, dst=B → allow). This kind of representation is informally used in our policy examples in the following subsections. To simplify policy processing, policies can be specified to be order-dependent and packets will be selected by a policy of the first match. For example, only packets from A to B are allowed can be specified as (src=A, dst=B → allow) (src=* dst=* → deny). The packets will be mapped against policies one by one orderly until find the first applicable one.

Now we can further elaborate the attributes of ipsec_action. IPSec has two security protocols, namely, Encapsulating Security Payload (ESP) and Authentication Header (AH). ESP is for encryption with authentication while AH is for authentication only. There are two modes in ipsec_action: transport and

tunnel mode. In transport mode, only payload is secured (encrypted or authenticated) and the header part is left in clear. In tunnel mode, the whole IP packet is secured and a new IP header is added (encapsulation). In the new header, tunnel entry and exit points are new source and destination addresses and the security protocol is the new protocol. The packet will then be encapsulated at tunnel entry point and sent to tunnel exit point where the packets are decapsulated and recovered to previous state. Attributes for the security enforcement are conveyed in extension headers with fields like Security Parameter Index (SPI), Sequence Number (for anti-replay purpose) etc. The policy condition part can specify fields in original header as well as new encapsulated outer header or new ESP or AH extension header.

2.2 Policy Problem Analysis

An erroneous policy could lead to communication blockade or serious security breach. Some problems might be caused by careless human error while some others might arise from interactions that can not be easily detected even with careful and experienced administrators. We will illustrate three scenarios of policy problems below.

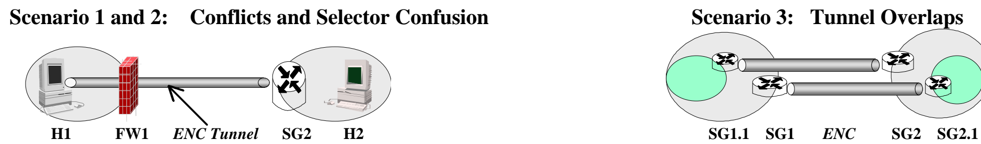


Figure 2: Examples of Policy Problems

In the scenario 1, an encryption tunnel is built between H1 and SG2 to protect their sensitive communication. In IPSec policy, policies can be specified to deny encrypted packet by denying all packets with ESP protocol like (src=* dst=* prot=esp → deny). Therefore, if Firewall FW1 has policy to deny all encrypted traffic because of the need to examine the content of the traffic, all packets will be dropped in the middle of the transmission.

In the scenario 2, H1 has policy to encrypt all packets from H1 to SG2. FW1 is a firewall to perform access control or selective security enforcement. We assume FW1 has simple access control policy as (src=H1, dst=H2 → allow) and (others → deny)[1]. However, because of the encryption tunnel that changes destination to be SG2 in outer header, the FW1 will mistakenly drop all traffic from H1 to H2 that should be allowed. Similarly, it can also mistakenly allow some traffic to pass due to confusion with selectors.

In the scenario 3, there are financial department 1.1 in site 1 and financial department 2.1 in site 2. We assume each department has its security gateways and has authority to make security policies to protect their property. In this example, department 1.1 decides that all traffic from 1.1 to site 2 must be encrypted through a tunnel from SG1.1 to SG2. At the same time, the administrator for site 1 decides that all traffic from site1 to department 2.1 must be encrypted through a tunnel from SG1 to SG2.1. Therefore, the traffic from SG1.1 to SG2.1 shall be governed by two policies and should be protected from SG1.1 all the way to SG2.1. However, the policies might not work as desired. First, if the administrator does not adjust the selector for the upper tunnel in specifying policies in SG1, then the traffic might skip lower tunnel completely such that it lacks the required protection from SG2 to SG2.1. On the other hand, if it is indeed adjusted to include the tunneled traffic, then the traffic will go through two tunnels. With this configuration, traffic is encapsulated with a new header by SG1.1 and then encapsulated with another new header by SG1 to send to SG2.1. When SG2.1 decapsulates and finds out the destination is SG2, SG2.1 will send traffic back to SG2. Finally SG2 will decapsulate and send traffic to its real destination. Although it is originally intended to encrypt traffic from SG2 to SG2.1, the traffic is eventually sent in clear from SG2 to SG2.1 because of tunnel interaction.

The above results are obviously undesirable and incorrect. The interaction of policies can lead to outcome that deviates from the original intention. What makes correct policy specification very difficult is two folds: First, tunnel operations cause complications in selector choices. Second, Lack of high-level view of overall objectives although each individual policy may appear to satisfy its individual goal. Therefore, it is important to specify policy at a higher level that maps to low level policies efficiently and unambiguously.

---

[1] This is for example only. Real access control policies will be much more complex than this example.

Based on requirement draft [6] in IPSP working group of IETF, IPSec policy management has the following requirements. First, we need to clearly define policy at a higher level. We should be able to understand what the policy does. The semantics must capture the relationship between IPSec SAs and higher-level security policies are clearly well defined. Second, we should have confidence that the policy does what it claims to and its implementation is correct. Third, it must scale to support complex policy administration schemes. Administrator must have ability to control and change policies for several different devices remotely at the same time. Fourth, in larger networks with complex hierarchy, different entities must be delegated with authorities to decide their own policies. Fifth, the mechanisms used must not require any protocol modification in any of the IPSec standards (ESP, AH, IKE). The mechanisms must be independent of the SA negotiation protocol.

2.3 Related Work

A working group IP security policy (IPSP) [5] is formed in IETF to address complex IPSec security policy problems. No complete solution has been developed to meet all the objectives specified in the requirement draft yet. A Security Policy Specification Language (SPSL) [2] was proposed in the working group to standardize policy specification, which currently only address one level (low-level policy) specification. Similarly, other proposed drafts such as policy information base [3] and data model [4] have been focused on low-level policies. A protocol called Security Policy Protocol (SPP) [7] was proposed to systematically resolve IPSec policies with Policy Servers. It defines protocol message exchange format and process. However, without consideration of potential conflicts and interactions as analyzed in section 2.2, the resolution algorithm may not provide assurance in policy correctness.

There are many firewall and IPSec VPN vendors. (See [9] for a long list of VPN vendors.) Many of the firewall and VPN router product offerings include policy configuration and management tools with varying degree of sophistication, such as Cisco's security policy manager [10], Indus River's Policy Vision4 [11], Xedia's VPN manager [12] etc. Typically, with a state-of-art VPN management tool, the policies can be centrally specified (and stored in LDAP) and automatically distributed to appropriate devices. Although the centralized management tools greatly eased administrators, it is still far from hassle-free without a clear high-level policy language. Furthermore, none of the policy management products seemed to focus on the potential conflicts. The problem is that some people defining some policies may not know other people defining other policies in large organization with complex hierarchy, hence the conflicts are inevitable. Without a rigorous way to verify correctness of policy specifications, large-scale VPN deployment is going to be troublesome due to possible unexpected security breaches.

The research effort closest to ours is probably firewall management toolkit [13] and filtering postures [14], in the sense of defining higher level policies centrally and distributing the policies to enforce. While they only focus on access control policies, we focus on interacted IPSec policies, i.e. VPN tunnel policies as well as access policies. Currently most firewalls are equipped with VPN capabilities such that the VPN policies and access control policies are processed together.

The needs of separating high-level requirements and low-level policies were addressed in [15, 16]. Our work applied the concepts to a specific policy service by defining IPSec security requirements at a high level. Some recent work [17,18] 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.

3. Security Requirements and Their Satisfaction

Clearly, there is need to define a security policy at a higher level with well-defined semantics. For simplicity, in the context of this paper, the higher level policy is called the security requirement[2]. 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 at a high level efficiently and unambiguously.

Because of size limit, we will present security requirements only in an informal way here. Interested readers can find rigorous and complete definitions on security requirements and their satisfaction in [21]. In security requirement, people can clearly specify their intention of security treatment on certain traffic

---

[2] We focus on two-level specification and transformation in this research, which does not exclude policy specification in more levels in policy hierarchy.

without concerning specific low-level parameters. Therefore, the attributes of flow identities specified in requirements are those of original flows. There are four main security requirements for IPSec policies.

- **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.*[3] → *deny | allow*
- **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 of the traffic to cover all links and nodes within the area. Optionally, users can authorize certain nodes in the area to access content since some nodes on the path may need to examine content. For example, an authorized firewall is allowed to access plain text content and examine content for intrusion detection purpose. Various algorithms can be specified in low-level policy while strength is specified in requirement level to indicate strength level of protection. We expect that appropriate algorithms can be selected in low level policies to be with sufficient strength[4]. 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.* → *protect (sec_function, strength, from, to, trusted_nodes)* The requirement is satisfied only if the traffic is with sufficient security protection on every link and node in protection area from *"from*" to *"to"*, except that the trusted nodes can be left uncovered by the function.
- **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. However, one node is not able to view the content of traffic if an encryption tunnel is built across it. As exemplified in Section 2.2, a policy can be specified to deny all encrypted traffic like (src=* dst=* prot=esp → deny). Similarly, some nodes might need to modify content for special processing but can not if authentication tunnels are built across them. We allow CAR to be explicitly specified to express the need for specific nodes to access content of certain traffic. CAR can be expressed as denying certain security function to prevent the nodes from accessing certain traffic as follows: *flow id.* → *deny_sec (sec_function, access_nodes).* The requirement is satisfied only if the traffic is not secured with the function *"sec_function"* on any node specified in *"access_nodes"*.
- **Security Association Requirement (SAR)** Security Associations (SA) [1] need to be formed to perform encryption/authentication function. There might be needs to specify that some nodes desire or not desire to set up SA of certain security function with some other nodes because of public key availability, capability match/mismatch etc. A simple way to specify a SAR could be: *flow id.* → *deny_SA (SA_peer1, SA_peer2, sec_function).* The requirement is satisfied only if none of nodes specified in *"SA_peer1"* forms SA with any of nodes specified in *"SA_peer2"* with function *"sec_function"*.

4. Determining Policies to Satisfy Security Requirements

Given a set of requirements, the problem is to find a set of policies to satisfy all of the requirements, or return a "failure" message if there is no such a set of policies. The problem could be very complex. We will subsequently present three different approaches to solve the problem and analyze their strengths and weaknesses. We expect the policy set generated by each approach to be **correct**, i.e. the policy set indeed satisfies all requirements. In addition, The algorithm to determine policies is desired to be **complete**, i.e. the algorithm can find a solution if there is one. Furthermore, we want the algorithm to be **efficient**. It is very important to have algorithm that can scale to large network size. Because of size limit, we omitted a lot of detail on algorithms and proofs. Interested readers can refer to [21] for more detail.

We first explain the mapping from requirements to policies. Low-level policies are specific implementations of high level requirements. One requirement might be satisfied by different policies. For example, a SCR specifying the protection of traffic from H1 to H2 as represented by (src=H1, dst=H2 → protect (sec_func[ENC], strength[strong], from[H1], to[H2], trusted_nodes[Ra, Rb])), can be satisfied by one tunnel or chain of tunnels that connect to each other. The trusted connecting nodes can de-apply certain security function and apply it again for the next tunnel. Chained tunnel is sometimes preferable to one non-stop tunnel for CAR or SAR satisfaction. Obviously no any CAR or SAR is violated if there is no tunnel. Both CAR and SAR are only restricting where and how to build tunnels. Furthermore, access control

---

[3] Flow is typically identified by 5-tuple (source address, destination address, source port, destination port, protocol).

[4] We can safely assume the mapping from strength to algorithms is deterministic.

policies can be easily determined after the tunnel configurations are determined, and thus we omit access control requirement in policy generation algorithms. Therefore, central part of policy determination is to determine tunnel policies that satisfy all SCRs without violating CARs and SARs.

For example, there are a set of three requirements: ***Three_Reqs*** = {Req1 (src = 1.* dst = 2.* → ENC Weak 1.* 2.* {all}), Req2 (src = 1.1.*, dst = 2.* → AUTH Strong 1.1.* 2.* {all}), Req3 (src = 1.*, dst = 2.1.* → ENC Strong 1.* 2.1.* {all})}, in which {all} means all enroute security gateways are trusted. In figure 4 1), the middle bar illustrates weak encryption protection area for Req1. The top bar illustrates protection area for Req2 and the bottom bar illustrates the protection area for Req3. Figure 4 2) shows the relationship among traffic selectors of the three requirements (traffic filters F1, F2 and F3). The question is what are policies to satisfy the three requirements. We will use this simple example throughout the paper to illustrate different approaches.

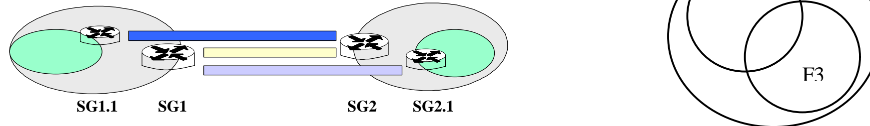


Figure 4: 1) Three_Reqs Example: Protection Areas 2) Three_Reqs Traffic Filters

4.1 Bundle Approach: Policies to Satisfy Requirements for Every Bundle of Flows

We can separate entire traffic into several disjoint traffic flow sets, we call bundles, each of which is subject to a unique set of security requirements. For example, in Three_Reqs example, there are four bundles that are governed by different set of requirement actions: (src = 1.1.*, dst = 2.1.*) subject to {Req1.action, Req2.action, Req3.action}, (src = 1.1.*, dst = 2.* – 2.1.*) subject to {Req1.action, Req3.action}, (src = 1.* – 1.1.*, dst = 2.1.*) subject to {Req2.action, Req3.action} and (src = 1.* – 1.1.*, dst = 2.* – 2.1.*) subject to {Req3.action}. In bundle approach, we will generate policies to satisfy all requirements for each bundle. For one particular bundle, the condition part of policies contains bundle selectors and action part contains appropriate security actions to meet all requirements for the bundle.

Using bundle approach, the problem is resolved in two steps. First, from given requirements, we will group entire traffic flows into a number of disjoint bundles and find out the subset of the requirements that are applied to each bundle. Second, for each bundle, given a set of requirement actions for the bundle, we will generate action part of policies for the bundle, and use bundle filters as selector part of the policies. We will first focus on the second step: how to generate correct policy actions given a set of requirement actions for one bundle.

4.1.1 Policy Actions To Satisfy A Set Of Security Requirement Actions

We design a polynomial algorithm to solve this problem. The basic idea is that one tunnel or chained tunnels across protection area are needed to fulfill security coverage requirement of the area, as illustrated in the following.

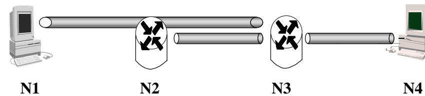


Figure 5: Tunnel Examples

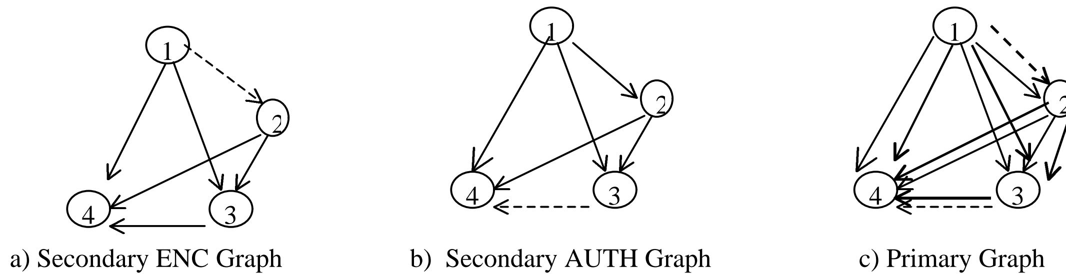
In the figure 5, assume the upper tunnels are for authentication and lower tunnels are for encryption. The packets are encapsulated at N1, then encapsulated again at N2, and send to N3, where the packets are decapsulated and encapsulated again to sent to N4, and so on. With this configuration, the upper chain of tunnels provides authentication coverage from N1 to N4 while the lower chain of tunnels provides encryption coverage from N1 to N4. Because of restriction posed by SARs, CARs or tunnel overlaps, one non-stop tunnel across protection area is not always possible. If we call the inner most tunnels to carry packets **primary tunnels**, and the corresponding SA **primary SA**, then the primary tunnels need to be chained together across an area to provide coverage for the area. On top of the primary tunnels, the tunnels to provide the other function are called secondary tunnels. Therefore, to satisfy all SCRs, at least we need to find allowed SA pairs that concatenate together to cover all required areas. If we depict allowed SAs as

edges, then we are to find a SA path. If there is no such a SA path to carry packet across the protection area for a set of requirements, the requirement set is certainly unsatisfiable.

Based on SCR satisfaction definition, one SCR is satisfied only when coverage requirement on every link and node within protection area is satisfied. First, combining three SCRs, we can obtain coverage requirements for each link and node. We use arrays sec_link and sec_node to store protection requirement for each link or node. For example, sec_link[1] is the coverage requirement for link 1-2. Then for the Three_Reqs example, we have the following protection requirements: sec_link[1]=(auth, strong), sec_link[2]=(enc, strong) and (auth, strong), sec_link[3] = (enc, strong); sec_node[1]=none, sec_node[2]=none, sec_node[3]=none, sec_node[4]=none.

We will do a CAR conflict check first. If one node is required to access content (packets can not be encrypted) based on a CAR and is distrusted for encryption (packets must be encrypted) based on a SCR, then there is a conflict and the requirements are unsatisfiable with any set of policies.

Next we start to construct graphs. To find eligible primary SAs, we need three graphs: ENC graph, AUTH graph and primary graph, in which ENC and AUTH graphs are needed in determining secondary SA paths. In the initial graphs, the edges are all allowed SAs. Dashed lines represent zero SA links, on which no security coverage is required for the particular security function. In primary graph, lighter links are primary AUTH edges and darker links are primary ENC links.



a) Secondary ENC Graph b) Secondary AUTH Graph c) Primary Graph

Figure 6: Building SA Digraphs: 1) Initial Graphs

For CAR node, all edges crossing the node have to be deleted from all three graphs to prevent it from being either primary SA or secondary SA. For distrusted node, all edges stopping and starting at the node should be deleted from primary graph. Furthermore, the edges stopping and starting at the node should also be deleted from AUTH or ENC graph to eliminate the possibility for those edges to be secondary SAs.

Next we will examine all edges in primary graphs one by one to see if they are eligible primary edges. One edge is eligible primary edge only if there is SA path in secondary function graph over the link span. For example, edge 1-3 ENC in primary graph is eligible because we can find path 1-3 or 1-2-3 in AUTH graph. In this example, all edges are eligible because each of them can have secondary SAs to provide necessary security coverage for the other security function. Finally, among eligible SAs, we find a shortest-path through 1-4. A tunnel configuration to satisfy all requirements is shown in the figure 5, in which the upper ones are AUTH tunnels and lower ones are ENC tunnels. There is no tunnel of AUTH function over link 3-4 because we've chosen the zero SA edge over the link in AUTH graph. The overall algorithm is as follows. To better understand the algorithm, readers can refer to [21] for more explanation and illustrations.

Algorithm Policy_Action_Generation (Req_actions)
1. Link_Node_Coverage(SCRs) // Calculate link node coverage
2. CAR_Conflict_Check (CARs, sec_node)
3. Initialize_Graphs (SARs) // Initial three graphs with all SAs allowed by SARs
4. CAR_Preprocessing (CARs) // Delete edges crossing CAR nodes
5. Distrusted_Nodes_Preprocessing (sec_node) // Delete edges connecting at distrust nodes
6. Finding_Eligible_Primary_SAs (Graphs) // Check secondary SA path on every primary edge
7. path = Dijkstra_Single_Source_Shortest_Path(Primary_graph, N_1) // Shortest path from N1
8. if (path[$Node_N$] = infinity)
9. return("No path found! Requirements are unsatisfiable.")
End Of Algorithm

4.1.2 Bundling and Ordering

With bundle approach, we can resolve policies in two steps. In first step, we group traffic into disjoint bundles and find requirement list for each bundle. Then in second step, we can use Policy_Action_Generation algorithm to determine action part of the policies. Having developed the policy action generation algorithm, we focus on the first step in this subsection.

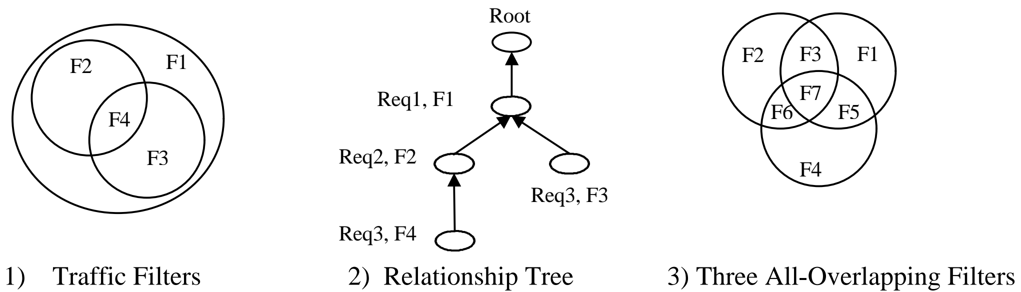
We need to calculate filter intersection and difference in order to separate traffic into bundles. For example, in figure 4 of Three_Reqs example, the intersection of F1 ∩ F2 ∩ F3 is subject to all Req1.action, Req2.action and Req3.action, F2 – F3 is subject to Req1.action and Req2.action while F3 – F2 is subject to Req1.action and Req2.action etc. A straightforward way to calculate intersections among K filters is to compare each filter with existing filters to determine intersections one by one, which takes $1+2+\ldots+K=O(K^2)$. What we do for intersection calculation is to do string matching to find out those filters with common prefix in all fields. Performance can be greatly improved by using a trie-based algorithm. The detail description of a trie-based algorithm to calculate filter intersection can be found in [20].

It appears to be not too difficult to separate traffic flows into bundles and find requirement list for each bundle. However, there is performance concern as follows. To group traffic flows into bundles, we need a lot of filter difference calculation. The difference calculation could be very time consuming and space consuming. For example, (1.* - 1.1*) might result in filters of 1.2.* 1.3.* 1.4.* and so on. It is undesirable to make policy set unnecessarily large. In addition, the selectors of existing policies have to be modified every time when a new requirement is added in. For instance, in Three_Reqs example, assume initially there is only Req1, and a tunnel is built for traffic 1.* to 2.*. When the new requirement Req2 comes up, the old tunnel has to be torn down in order to build new tunnel for traffic (1.*-1.1.*, 2.*) because SPI is already set up for old selector.

We will develop an algorithm to achieve the same result as generating policies for each disjoint bundle without need of difference calculation. To achieve best efficiency, the approach we take is to add new filters and policies while keep old policies as much as possible. For example, a tunnel is already built for traffic (1.* 2.*) and we have a new requirement for traffic (1.1.* 2.*). We will not change the existing tunnel but we add new policies on top of the existing policy to be with filter (1.1.* 2.*). Therefore, the new filter will be checked first and only traffic of (1.*-1.1.*, 2.*) can be selected by the bottom policies, which is exactly the same result as if we build policies for disjoint bundles (1.* 2.*) and (1.*-1.1.*, 2.*).

There are three main issues. First is how to calculate bundle filters in order to generate selector part of policies. Second is how to get requirement list for every bundle before we can use Policy_Action_Generation algorithm to generate action part of policies. Third is how to ensure policies are inserted at right place to guarantee the correctness of policies. As exemplified, we utilize the order of the policies to simplify filter calculation, and thus the order becomes critical.

To facilitate requirements and order tracking, we developed relationship tree mechanisms. For instance, in Three_Reqs example, there are three filters F1 = (1.* 2.*) F2 = (1.1.* 2.*) and F3 = (1.* 2.1.*). We will illustrate how to construct relationship tree as follows.



1) Traffic Filters 2) Relationship Tree 3) Three All-Overlapping Filters

Figure 8: Example of Constructing Relationship Tree

In the example, we first process filter F1 with Req1. When Req2 with filter F2 comes up, new node F2 is a child of F1 since F2 is contained with F1. When Req3 with F3 comes up, we will calculate overlap and generate filter F4 with Req3 and insert as child of F2 since F4 is contained by F2. Then new node F3 will be inserted as child of F1 since it is contained by F1.

With the relationship tree, we solve the three issues as follows. We will generate a set of policies for each node in the relationship tree with selectors to be the node filter. The requirement list of each node is the concatenation of requirement identity associated with the node itself and all its ancestors. Policy order is correct as long as the policies for one particular node are inserted right on top of the policies for its

closest containing node. Therefore, with the above example relationship tree, we will generate four policy set {policy_set1, policy_set2, policy_set3, policy_set4} to satisfy requirement set {{Req1}, {Req1, Req2}, {Req1, Req3}, {Req1, Req2, Req3}}. The order of the policy sets will be {policy_set4, policy_set2, policy_set3, policy_set1} and the filters of the policy sets are {F4, F2, F3, F1}. The results will be exactly the same as if we calculate policies for disjoint bundles {F4, F2-F4, F3-F4, F1-F2-F3}.

Algorithm Bundle_Approach (Reqs)
1. *For every Reqi in Reqs*
2. *new_req_ID = i*
3. *overlaps_set = GetOverlappingFiltersInTrie(Reqi.filter)*
4. *filter_set = Reqi.filter* **È** *overlaps_set*
5. *for every* $filter_j$ *in filter_set*
6. *if (*$filter_j$ *is Reqi.filter)* // *original filter*
7. *new_filter =* $filter_j$
8. *closest_containing_filter = GetClosestContainingFilterInTrie(new_filter)*
9. *else* // *overlapping filter*
10. *new_filter = Intersection (*$filter_j$*,* Req_i*.filter)*
11. *closest_containing_filter =* $filter_i$
12. *InsertTrie (new_filter)*
13. *InsertRelaTree (new_filter, new_req_ID, closest_containing_filter -> rela_ptr)*
14. *req_list = GetReqList (new_filter -> rela_ptr)*
15. *new_policies.actions = Policy_Action_Generation (new_filter, req_list)*
16. *new_policies.selector = new_filter*
17. *InstallPolicy(new_filter, new_policies)*
18. *UpdateContainedPolicies (new_filter -> rela_ptr)*

End Of Algorithm

With relationship tree to keep track of requirements and policy order, the performance is much improved. We present our performance test result in section 4.4.

For the Three_Reqs example, there are four bundles {(1.1.*, 2.1.*), (1.*-1.1.*, 2.1.*), (1.1.*, 2.*-2.1.*), (1.*-1.1.*, 2.*-2.1.*)} which subject to requirements {(Req1, Req2, Req3), (Req1, Req3), (Req1, Req2), (Req1)} respectively. If in addition to three SCRs, SG1 and SG2 are CAR nodes for both encryption and authentication, then the resultant policies using bundle approach are those shown in the figure 9. In the figure 9, there are four groups of tunnels for four bundles. The group of four pink tunnels selects traffic (1.1.*, 2.1.*) and satisfies Req1, Req2 and Req3. The group of blue tunnels select traffic (1.1.*, 2-2.1.*) and satisfied Req1 and Req2. The group of purple tunnels selects traffic (1.*-1.1.*, 2.1.*) and satisfies Req1 and Req3. The yellow tunnel is for all other traffic and satisfies Req1.

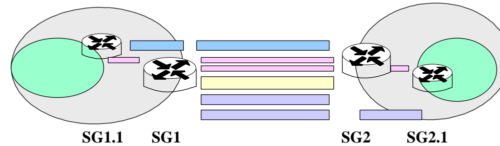


Figure 9: Solutions For Three_Reqs Example Using Bundle Approach

4.2 Direct Approach: Building Chained Tunnels For Each SCR

Although the bundle approach is complete and correct, it is still less ideal in its efficiency and scalability. Specifically, we have to calculate bundle filters and then generate policies for each bundle. We can see from figure 9 that tunnels have to be built separately for different bundles. In addition, a new requirement can trigger a lot of SA reestablishment. In this section, we develop algorithm to generate policies to correspond to each SCR without separating traffic into bundles, which is more efficient and requires less update. The reason is that in most cases tunnels can directly work together to provide necessary protection. Only when there are overlaps between interacted tunnels (i.e. selectors have non-nil intersection) can tunnels together cause requirement violation. We will develop algorithm to generate non-interacted-overlapping tunnel policies for each SCR.

For instance, in Three_Reqs example, we do not bother to separate traffic into bundles. We simply build tunnels for each SCR and make sure the new tunnels do not overlap with any of existing tunnels. The

following figure showed the solutions using direct approach. We have three SCRs. For Req1, we first build a middle tunnel from border of SG1 to border of SG2. Then for Req2, we will build the top tunnel from SG1.1 to SG2 that does not overlap with the existing middle tunnel. Last for Req3, since one nonstop tunnel will be overlapping with the top tunnel, we build two connecting bottom tunnels that do not overlap with any existing tunnels. Each tunnel selects all traffic of the corresponding SCR. The four tunnels together can satisfy all three requirements.

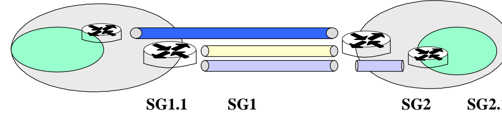


Figure 10: Solutions For Three_Reqs Example Using Direct Approach

Direct approach has two advantages over bundle approach, i.e. efficient policy generation and efficient requirement update. In bundle approach, we generate policies for each bundle. There might be much more bundles than SCRs. It will be more efficient to generate policies to correspond to SCR rather than bundles. Furthermore, the policies generated using direct approach will be less than those using bundle approach, which ultimately makes policy management easier. For Three_Reqs example, in direct approach solution, the middle tunnel will govern all traffic from SG1 to SG2, while we have to build Req3 into each bundle using bundle approach.

Direct approach is also more efficient in regard to requirement update. In bundle approach, a new requirement may trigger series of policy change due to requirement list change for contained bundles. In direct approach, new tunnels for new requirement will be selecting all traffic of the requirement, and thus the new requirement will be automatically applied to all contained traffic without need to change existing policies. To make sure the new tunnel applies to all traffic in requirement selector, the only additional work to do is to adjust selectors to include the encapsulated traffic. For example, if the middle tunnel's original selector is (src = 1.* dst = 2.*), to include the traffic encapsulated by the top tunnel, the new selectors could be (src = 1.*, dst = 2.*) and (src = 1.1.*, dst = 2.* prot = ESP).

The focus of the direct approach is then how to build chained tunnels for each SCR that satisfy the SCR and relevant CARs and SARs, and do not overlap with any existing tunnels. Again, because of size limit, interested readers can find detail algorithms in [21].

4.3 Combined Approach: Combining Direct Approach With Bundle Approach

Although the bundle approach is complete and correct, it is less ideal in its efficiency and scalability. Direct approach is very efficient but not complete. There is no solution using direct approach if it can not find non-overlapping solution for one SCR, while there might be solution using bundle approach. By combining the two approaches, we can achieve both efficiency and completeness. In the combined approach, we will first test if there is solution using direct approach. Only when direct approach can not find a non-overlapping solution, we will use bundle approach to find a solution. In combined approach, we use direct approach as much as possible for maximal efficiency and use bundle approach to deal with problems that can not be solved by direct approach to achieve completeness.

4.4 Implementation

Having developed the algorithms, we implemented them in C on Linux platform. The software takes a requirement file as input, and outputs a policy file containing automatically generated policies. We test performance of the algorithms using randomly generated requirements, i.e. a random sub-domain in source domain requires secure communication with a random sub-domain in destination domain with random protocol and strength. Domains in our research are hierarchically organized, i.e. each domain contains several sub-domains, which may contain smaller sub-domains. Security gateways locate at borders of sub-domains to protect the sub-domains' communications.

We implemented bundle and direct approach. The performance is shown as the following.

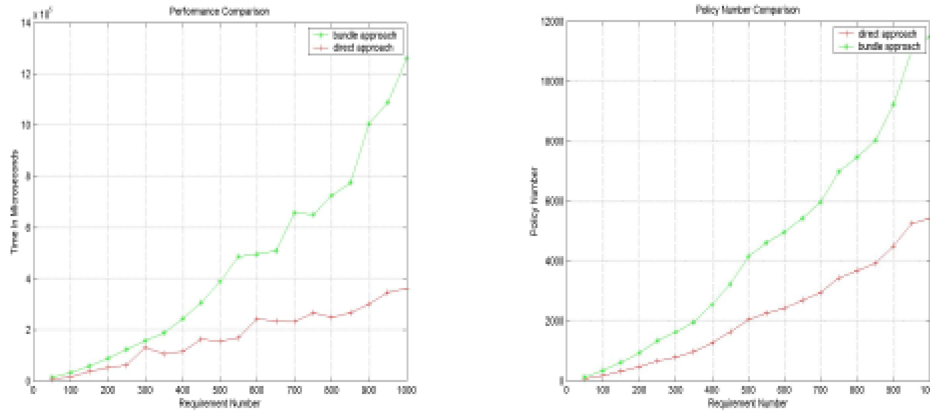


Figure 12: Experimental Results

In the experiment, we randomly generate 50, 100, 150, …, until 1000 requirements and use bundle/direct approach to process the requirement file respectively. The time it takes to generate all policies for different number of requirements are shown in the figure 12 1). Please note that the time we recorded here is only policy generation cost without including SA establishment cost. From the result, we can see that bundle approach takes longer generation time than direct approach does. The second figure shows policy numbers generated using two approaches. As we analyzed before, the bundle approach will generate more policies than those generated by direct approach.

5 Conclusion and Future Work

IPSec/VPN security policies are widely used in firewalls or security gateways to protect information property. The security treatment (e.g. deny, allow or encrypt etc.) of all inbound or outbound traffic will be determined by the security policies such that it is critical for policies to be specified and configured correctly. IPSec policies are manually configured to individual security gateway in current practice, which could be very inefficient and error-prone. Even each individual policy appears to be correct, the policies together may interact to cause security breach. In this research, we focus on one important question: **How to ensure correctness of policies?**

We analyzed potential problems in IPSec policy specification and found correct IPSec policy specification difficult to achieve due to three reasons. First, Encapsulation in IPSec makes it hard to specify correct selectors. Second, even every policy is correct by its own, policies together might interact (e.g. overlapping tunnels) and cause undesired security violation. Third, there is vague relationship between objective and specific policies to meet the objective, and there is lack of overall view in policy specification. Because of the selector changes for encapsulation and tunnel interactions, the above problems cannot be resolved in one level. To solve the problems, we first clearly defined security policies in two levels: requirement level security policy and implementation level security policy. Requirement level policies reflect security objective and are implementation independent. Therefore, security requirements become criteria in evaluating policy correctness, i.e. low-level policies are correct if and only if they satisfy all security requirements.

We developed algorithms to automatically generate correct low level policies to meet all requirements. Therefore, people can just specify the desired requirements for protection then correct low level policies will be automatically generated and delivered to appropriate devices to enforce, which will greatly improve policy management. The input of the algorithm is a set of requirements and the output of the algorithm is a set of policies that satisfies all the requirements or return "failure" message if there is no such a set of policies. We developed three different approaches. The first is bundle approach in which we generate policies for a set of flows that are subject to a unique set of requirements (we call it a bundle of flows). The approach is correct and complete but not very efficient. In the second approach, we build non-overlapping policies for each SCR respectively, and then the resultant policies can satisfy all requirements. This approach is correct and very efficient but not complete. Then in the third approach, we combine the bundle and direct approach to achieve correctness, completeness and efficiency. The experiment results demonstrated the performance of the developed algorithms.

In the future, the research can be extended from several aspects. First of all, we focused on centralized policy management in this research. The research can be extended for distributed policy management in which distributed policy servers can communicate and make joint decision on correct policies. The constraint and optimization of policy generation for distributed architecture need further study. Furthermore, we developed algorithm to automatically generate correct policies to satisfy all given requirements. If no policies can satisfy all requirements, then we will generate a failure message. In this case, the conflict may reside at requirement level. The requirement conflict resolution techniques will demand further research. Last, we've specified a higher-level security policy that is implementation independent. More levels of security policy may be specified until the whole hierarchy is clearly established.

6. References

[1] S. Kent, R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, Internet Society, Network Working Group, Nov. 1998
[2] M. Condell, C. Lynn, J. Zao, "Security Policy Specification Language", <draft-ietf-ipsp-spsl-00.txt>, Internet Draft, March, 2000
[3] J. Jason, "IPsec Configuration Policy Model", Internet Draft, <draft-ietf-ipsp-config-policy-model-00.txt>, March, 2000
[4] R. Pereira, P. Bhattacharya, "IPSec Policy Data Model", Internet Draft, <draft-ietf-ipsec-policy-model_00.txt>, Feb. 1998
[5] See http://www.ietf.org/ipsp_charter.html
[6] M. Blaze, A. Keromytis, M. Richardson, L. Sanchez, "IPSP Requirements", <draft-ietf-ipsp-requirements_00.txt>, Internet Draft, July, 2000
[7] L.A. Sanchez, M.N. Condell, "Security Policy Protocol", <draft-ietf-ipsp-spp-00.txt>, Internet Draft, July, 2000
[8] A. Durand, P. Fasano, I. Guardini, D. Lento, "Ipv6 Tunnel Broker", <draft-ietf-ngtrans-broker-00.txt>, Internet Draft, Sept. 2000
[9] Alphabetical VPN Vendor List, http://www.timberlinetechnologies.com/products/vpn.html
[10] http://www.cisco.com
[11] http://www.indusriver.com
[12] http://www.xedia.com
[13] Y. Bartal, A. Mayer, K. Nissim, A. Wool. "Firmato: A novel firewall management toolkit". In Proc. 20th IEEE Symp. On Security and Privacy, pp. 17-31, Oakland, CA, May 1999
[14] J. D. Guttman, "Filtering Postures: Local enforcement for global policies". In Proc. IEEE Symp. on Security and Privacy, Oakland, CA, 1997
[15] 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
[16] J. D. Moffett, "Requirements and Policies", Position paper for Policy Workshop 1999
[17] E.C. Lupu and M. Sloman. "Conflict Analysis for Management Policies". Proc. 5th IFIP/IEEE International Symposium on Integrated Network Management, pp. 430-443, 1997
[18] E.C. Lupu and M. Sloman. "Conflicts in Policy-Based Distributed Systems Management". IEEE Transaction on Software Engineering. Vol. 25, No. 6, pp. 852-869, Nov./Dec. 1999
[19] V. Srinivasan, G. Varghese, S. Suri and M. Waldvogel, "Fast and Scalable Layer Four Switching," Proceedings of the ACM SIGCOMM'98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 1998, pp. 191-202
[20] A. Hari, S. Suri, G. Parulkar, "Detecting and Resolving Packet Filter Conflicts", Infocom 2000, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Proceedings, IEEE, pp. 1203-1212, Vol.3.
[21] Z. Fu, Technical Report, Automatic Generation of Security Policies, http://shang.csc.ncsu.edu/papers/secpolicy.pdf
[22] Z. Fu, S. F. Wu, H. Huang, K. Loh, F. Gong, "IPSec/VPN Security Policy: Correctness, Conflict Detection and Resolution", IEEE Policy 2001 Workshop, Jan. 2001.