# ZERO-Conflict: A Grouping-based Approach for Automatic Generation of IPSec/VPN Security Policies*

Kuong-Ho Chen, Yuan-Siao Liu, Tzong-Jye Liu, and Chyi-Ren Dow

Department of Computer Science, Feng Chia University,
No. 100 Wenhwa Rd., Seatwen, Taichung, Taiwan 40724, R.O.C.
cyne@pluto.iecs.fcu.edu.tw, {m9301324, tjliu, crdow}@fcu.edu.tw

**Abstract.** IPSec/VPN management is a complicated challenge, since IPSec functions correctly only if its security policies satisfy all administrated requirements. Computer-generated security policies tend to conflict with each other, which would causes network congestion or creates security vulnerability. Thus conflict resolving has become an issue. In this paper, a method to automatically generate policies is proposed. Instead of performing complicated conflict-checking procedures as most existing works do, the proposed *Zero-Conflict* algorithm is able to predict and avoid conflict in advance by using *requirement groups* and *cut points* techniques. Since policies are established without the need to perform backward conflict check, thus yielding a significantly less time-complexity, which is *O(nlogn)*. Experimental results show that it maintains a satisfactorily minimal numbers of generated tunnels.

**Keywords:** IP security, network management, policy conflict, security policy, security requirement.

## 1 Introduction

Network management in large distributed networks, in particular IPSec/VPN management [8, 13], is a complicated challenge. IPSec functions will be executed correctly only if policies are correctly specified and configured, but due to the growing number of secure Internet applications today, IPSec policy [1, 7] deployment has become rather complex in large distributed networks, and manual configuration is rather tedious, ineffective, and often erroneous. On the other hand, a policy-based management system treats network requirements as goals to be achieved, automatically translates them into low-level machine-understandable policies, and systematically applies them to right network devices. Since IPSec is basically a typical policy-enabled networking service, policy-based network management [3, 11] is a good solution in handling complicated IPSec policy, and various solutions for IPSec/VPN policy management have been proposed in researches such as [2, 4, 5, 6, 12, 13].

* This Research is supported in part by the National Science Council under the grants No. NSC NSC94-2213-E035-025, NSC95-2221-E035-071.

A class of high level policy is defined in [5], which is called *security requirement*. Conceptually, security requirements (high level policy) are like goals, while implemental IPSec policies (low level policy) are like specific plans to achieve these objectives. IPSec policies are considered correct only if these policies as a whole are able to satisfy all specified security requirements. However, security requirement and IPSec policy may not directly map to each other since one security requirement might be satisfied by several sets of implemental policies. Moreover, it is possible that there are conflicts between requirements and policies. If such conflicts exist on one of the gateways/routers, packets could be dropped, network could be down, and security could be breached. Conflicts occur when given requirements conflict with each other, or when a set of policies binding together is unable to support given requirements. An exemplary scenario of the latter case is given in Table 1 and Fig. 1. (In this paper, the terms *policy* and *tunnel* will be used interchangeably.)

**Table 1.** Two Requirements

| | Requirement |
|---|---|
| $Req_1$ | All traffics from A to D must be applied encryption |
| $Req_2$ | All traffics from C to E must be applied authentication |

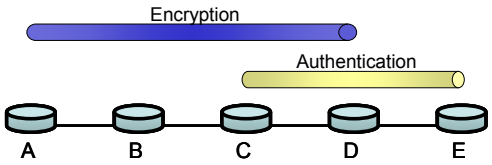



**Fig. 1.** Overlapping Tunnels

In this scenario, there are two requirements for all traffics from A to E: the coverage of $Req_1$ is from A to D and the coverage of $Req_2$ is from C to E. According to these two requirements, two tunnels were built: one from A to D with encryption and another from C to E with authentication. With these tunnels, all packets are encapsulated in A, encapsulated again in C, and then sent to E. E decapsulates these packets and finds that their destinations are D, thus send them there. Finally, D decapsulates them and sends the packets to their final destination E. However, while the original requirement was to authenticate the traffic from C to E, the traffic is actually sent without protection from D to E due to tunnel overlapping.

Thus said, in spite of policy generation, an IPSec policy management system will also need to tack tunnel overlapping and identify possible policy conflicts. Researches so far in automatic IPSec policy generation [2, 4, 6, 12] focus their efforts on conflict resolving: in these algorithms, each newly generated security policy is compared with existing policies to check for conflict. Once found, operations are called for conflict resolve. The process of requirement comparison, however, is rather time-consuming, since any change or addition in security requirements will require the entire execution of conflict-check (and possibly conflict-resolving) procedure.

If tunnels were constructed in a way such that conflicts are predicted and avoided in advance, establishment of policies without need for time-consuming backward conflict check would be made possible, thus yielding a faster result. With this in mind,

this paper proposed a Zero-Conflict algorithm, which is an efficient conflict-avoiding method for automatic generation of IPSec/VPN security policy. In this paper, a 3-phased automatic policy construction procedure is proposed, which seeks to lower the complexity of conflict dealing by dividing requirements into groups, and establish *bus tunnels* and *branch tunnels* inside each group. In comparison with several existing methods [2, 4, 12], which mostly came with the efforts of $O(n^2)$, this approach requires the effort of only $O(n\log n)$, where $n$ is the number of requirements. Simulation results also show that the proposed Zero-Conflict approach maintains an appropriately minimal number of established tunnels.

The rest of this paper is organized as follows. Related works are addressed in Section 2. Analysis of policy conflict problem is described in Section 3. Automatic policy generation algorithms are described in Section 4. The complexity analysis and simulation are in Section 5 and Section 6. Finally, conclusions are made in Section 7.

## 2 Related Works

In this section, research backgrounds and literatures related to our work are described in Section 1.1, including the categories and the definitions of security requirements. Various approaches for automatic IPSec/VPN policy generation are then described and discussed in Section 2.2, including bundle approach [4], direct approach [4], Order-Split approach [12], and Conflict-Free approach [2].

### 2.1 Security Requirements

In [5], two levels of security policies are defined: the *requirement* level and the *implementation* level. The needs to distinguish high-level security requirements and low-level policies were addressed in [9, 10]. A security policy set is correct if and only if it satisfies all the requirements. A requirement $R$ is a rule of the following form: If condition $C$ then action $A$:

$$R \equiv C \rightarrow A \quad (1)$$

There are four cases of requirements defined in [5]:

- Access Control Requirement (ACR):
  *flow id* → *deny* | *allow*
- Security Coverage Requirement (SCR):
  *flow id* → *enforce (sec-function, strength, from, to, [trusted-nodes])*
- Content Access Requirement (CAR):
  *flow id, [sec-function, access-nodes]* → *deny* | *allow*
- Security Association Requirement (SAR):
  *flow id, [SA-peer1, SA-peer2]* → *deny* | *allow*

*flow id* is used to identify a traffic flow, and is composed of 5 to 6 sub-selectors including *src-addr*, *dst-addr*, *src-port*, *dst-port*, *protocol*, and optional *user-id*. A requirement is satisfied if and only if all packets selected by the condition part execute the action part of the requirement. As were mentioned in [2, 12], other

requirements such as SAR or CAR can be validated after SCR results are produced. ACR policies also can be determined after tunnel configurations are done. Therefore the algorithm in this paper will seek to focus on the handling SCR requirements only.

### 2.2 Previous Works

**Bundle approach** [4] is the first algorithm for automatic policy generation. In this approach, the problem is divided into two phases. From given requirements, the entire traffic is first divided into several disjointed traffic flows, which are called bundles. Sets of security policies are then built from each bundle. Although correct and solution-guaranteed, this approach is not efficient since redundant tunnels for the same area could be built from different bundles.

**Direct approach** which was also proposed in [4], tunnels are built from each requirement directly, and all the while with the system making sure new tunnels do not overlap with any existing ones. If overlapping occurs, the new tunnel is divided into two connecting tunnels. In comparison with bundle approach, this approach produces fewer tunnels and has better efficiency. It does not, however, yield solutions for every case.

**Ordered-Split algorithm** [12] is based on traditional task-scheduling schemes for automatic policy generation. Original requirements are converted into tie-free requirement sets; a minimal sized Canonical Solution for the new requirements are then acquired. The condition for a Canonical Solution is that no two tunnels share the same start as well end, while the condition for a tie-free requirement is that no two requirements share the same *from* and *to*. According to [12], this algorithm generates fewer tunnels than Bundle/Direct approach, and is free of tunnel-redundancy problem. Its time-complexity is $O(n^2)$.

**Conflict-Free approach** [2] focuses on the handling with the intersection relationship between tunnels. In this approach all tunnel are made as long as possible since if two security policy sets have the same number of tunnels, the set which has longer average tunnel length will be preferred since longer tunnel decreases the number of times a traffic has to be encapsulated/decapsulated. The time-complexity, of this algorithm is $O(n^2)$.

## 3 Analysis of Overlapping Relationship Possibilities

A policy conflict is caused when two or more tunnels have certain overlapping relationships. To be more specific, when packets in one tunnel are passing through a node in the network, they will be pulled into other tunnels due to the policies of the same node, which is likely to cause a policy conflict. To better understand the nature of policy conflict, shown in Fig. 2 are the six possibilities of overlapping relationships between two tunnels, whose analysis could be used to find the possible cause of conflicts.

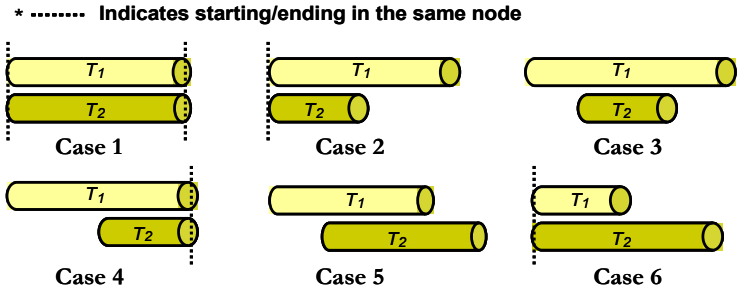



**Fig. 2.** Overlapping Relationship Possibilities for Two-tunnel Scenario

According to [2], conflicts could possibly appear in case 5 and case 6 only. In case 5, packets are encapsulated at the start of $T_1$. When traveling through the network to the middle node, which is the start of $T_2$, they will be encapsulated again, then be directly sent to the most right node, which is the end of $T_2$, and be unwrapped and sent back to the end of $T_1$. After arriving at the end of $T_1$, the traffic will leave $T_1$ and be sent to the most right node. Since there is a sent back occurrence, conflicts are likely to be caused. In case 6, $T_1$ and $T_2$ start at the same node, and packets will be encapsulated twice here. When being unwrapped first time at the end of $T_2$, these packets will be sent back to the end of $T_1$, which might cause a policy conflict. Note that case 2 and case 6 differ only in the order of input tunnels. While processing, algorithms in [2, 4, 6, 12] will perform extra order switching in order to convert case 6 into case 2, thus avoiding conflict, which generates extra overhead. In contrast with these, we hope to construct a scheme that is free of this problem of ordering, whereas policies are processed in the order as they were inputted.

To sum it up, a conflict exists when *send back* occurs in two overlapping tunnels, therefore only case 5 and case 6 can possibly cause conflicts. Knowing this in advance, our algorithm, different from those aforementioned, seeks to avoid the occurrences of these two situations at all, rather than dealing with them headfirst.

## 4 Zero-Conflict Algorithm

Taking advantages from analysis above, *Zero-Conflict* algorithm was designed with the concepts of *requirement group* and *cut point* in mind. In this section these two major concepts are described, the mechanism of the Zero-Conflict algorithm itself explained, and an example is also given for better understanding.

### 4.1 Requirement Group

In a two-tunnel scenario, if these two tunnels do not overlap with each other, no conflict will occur. A requirement group is a set of requirements that do not overlap with the requirements belonging to other group. In other words, a group is composed of at least one or more overlapping requirements so that the conflicts will appear only in their own respective groups.

### 4.2 Cut-Point

Once requirement groups are finalized, conflicts inside each group are to be resolved. Common methods for resolving overlapping is to divide the requirement in question into two non-conflicting ones.

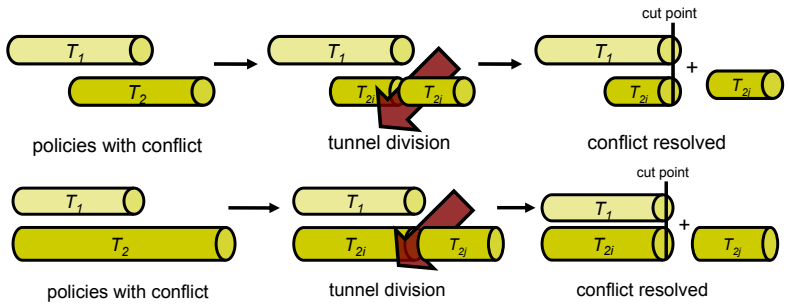



**Fig. 3.** Conflict Resolving with Tunnel Division

The center of the problem is to find where to "cut", thus the *cut point*. An observation was made: If the original requirement list is first sorted by *from* values in ascending order and tunnels are established accordingly, when conflicting cases in Fig. 3 appear, the tunnel $T_2$ will be divided at the end of the tunnel $T_1$ (thus $T_2.cutpoint=T_1.to$). Thus $T_2$ will be replaced by $T_{2i}=(T_2.from,\ T_1.to)$ and $T_{2j}=(T_1.to,\ T_2.to)$. Any subsequent tunnels, if in conflict with $T_1$, will be divided at the same cut point, thus $T_1.to$.

Thus if every *to* in the requirement list is treated as a cut point, and all tunnels are to be divided according to these cut points, conflicts can be avoided (In this way, a tunnel covering *n* cut points will be divided *n* times). Basing on this assumption, two facts can be derived: a) a cut point is the end of one tunnel and the start of another, but a start of one tunnel is not necessarily a cut point. b) Between every two neighboring cut points in a group, there must be at least one tunnel. According to a) and b), those tunnels whose establishments are guaranteed can be determined in early stage of the algorithm. These are called *bus tunnels*, which will be established after the acquisition of the *to* of a security requirement *in advance*, and serve as backbones shared by all requirements in the same group. Branching tunnels will later be built from these buses, covering remaining areas, which are henceforth called *branch tunnels*. It could be observed that the *from* and *to* of a bus tunnel are both cut points. For a branch tunnel, its *from* must not be a cut point, while its *to* must be one. Branch tunnels, in conjunction with bus tunnels, satisfy the overall covering demands of the requirement list. Therefore any given requirement can satisfied by connecting its *from* with a closest bus tunnel using a branch tunnel.

### 4.3 Zero-Conflict Algorithm

Taking advantages from analysis above, an algorithm for automatic policy generation which avoids the two conflict cases can thus be designed, which is called "*Zero-*

*Conflict* algorithm". The pseudo code of Zero-Conflict is shown in Fig. 4. Sub-functions are explained as follows:

**Zero-Conflict Algorithm**

```
01  Zero-Conflict_Algorithm (Reqs)
02  {
03      remove_length_1_requirement (Reqs, lenegth1_Req_list);
04      sort_by_from_value (Reqs);
05      assign_group_number_to_each_requirement (Reqs);
06      gather_cut_points_for_each_requirement_group (Reqs, Cut-Point_list);
07
08      build_bus_tunnel (Cut-Point_list, Policy_List);
09      build_branch_tunnel (Reqs, Cut-Point_list, Policy_List);
10      build_length_1_tunnel (length1_Req_list, Policy_List);
11
12      remove_redundant_tunnel (Policy_List);
13
14      return Policy_List;
15  }
```

**Fig. 4.** Pseudo code for Zero-Conflict Algorithm

**remove_length_1_requirment (Reqs, length1_Req_list).** Requirements with their *from* and *to* as neighbors (hop count is 1, thus one-hop requirement) does not conflict with other requirements, but increases the number of cut points unnecessarily, thus has to be moved to backup space *length1_Req_list* and be processed in later stage.

**sort_by_from_value (Reqs).** The original requirement list is then sorted by *from* in ascending order. Note that most subsequent operations are done directly on the sorted requirements list, thus lowering their time-complexity to *O(n)*.

**assign_group_number_to_each_requirement (Reqs).** To the sorted requirement list, a variable *max_end_node* is used to record the end node of current group, which is also the *to* of the first requirement. A single *n*-loop operation is executed to determine the group of each requirement. If a requirement belongs to current group, its *from* must be less or equal to *max_end_node*, else it belongs to the next requirement group. In the latter case, a new group is created, and *max_end_node* is set to the *to* of first requirement in this group. Note that if *to* is greater then *max_end_node*, then *max_end_node* is set to the *to.*

**gather_cut_points_for_each_requirement_group (Reqs, Cut-Point_list).** The end nodes from each requirements are collected in non-repeated fashion as cut points, and then sorted with their group numbers as primary key and *to* as second key, thus generating the cut point list.

**build_bus_tunnel (Cut-Point_list, Policy_List).** For every two neighboring cut points in each requirement group, bus tunnels are established between them.

**build_branch_tunnel (Reqs, Cut-Point_list, Policy_List).** Once bus tunnels are established, a single *n*-loop operation is executed to establish branch tunnels. Since

the *to* of a branch tunnel is a cut point, which itself is the *from* of a certain bus tunnel. Thus for each requirement, a branch tunnel is established between its *from* and the nearest cut point, thus linking itself with the backbone of the requirement group. Note that for each *from*, only one branch tunnel will be established, since there may be multiple requirements with identical *from*.

**build_length_1_tunnel (length1_Req_list, Policy_List).** Finally, the removed one-hop requirements are established. (After this function is completed, established tunnels are already able to satisfy all requirements.)

**remove_redundant_tunnels (Policy_List).** This function removes redundant tunnels. If the area covered by several tunnels is also the covered area by a single tunnel, the latter tunnel is considered redundant, and will be removed.

Generated tunnels are first sorted by *from* in descending order and by hop count (hop count = *to-from)* in ascending order. This is due to that all tunnels excluding one-hop end in a cut point. Since hop count is the distance between *to* and *from*, therefore sorting by *from* equals to sorting by the distance between the start of each tunnel and its nearest cut point. In this way, shorter tunnels will be pushed toward the top. Since a redundant tunnel can only be replaced by several shorter tunnels (thus tunnels with less hop count) that interconnect together, therefore a variable $M_a$ is used to record the area covered by current set of connecting tunnels.

An *n*-loop operation is then executed for removal. Since there could be multiple tunnels connecting together, a variable $M_a$ is used to record the area covered by current set of connecting tunnels. The initial value of $M_a$ is set to the area covered by the first tunnel, thus $M_a=(T_o.from, T_0.to)$.

Each tunnel $T_i$ is first compared with $M_a$. If identical, $T_i$ will be erased. If not, the operation proceeds to see whether $T_i$ is connected with the area covered by $M_a$. If $T_i.to = M_a.form$, then $T_i$ is added into the set (thus $M_a.from = T_i.from$). Else, $M_a$ is set to ($T_i.from, T_i.to$), and then the loop is carried onto the next tunnel.

### 4.4 An Example of Zero-Conflict Method

For better understanding, an example is given in Table 2, where 8 requirements are input.

First of all, one-hop requirements are to be removed. In this case, $Req_3$ is removed, and the remaining requirements are sorted by *from*, thus generating Table 3.

To the finding of group numbers, there are two groups in this case. The first group, $G_0=\{Req_2, Req_4, Req_5, Req_6, Req_8\}$, with *max_end_node* = 6. While processing $Req_1$ it can be noted that $Req_1.from$ is greater than current *max_end_node*, thus forming a new group, $G_1=\{Req_1, Req_7\}$, with *max_end_node = 10*.

Subsequently, cut points in each requirement groups are to be decided. In this case, they are *{5, 6}* for $G_o$, and *{9, 10}* for $G_1$.

Thus onto the construction of bus tunnels. In this case, $T_1=(5,6)$ is established for $G_0$, while $T_2=(9,10)$ is established for $G_1$.

For branch tunnel construction, the nearest cut point for requirements in $G_0$ is 5. And since $Req_5$ and $Req_8$ share the same *from*, only one branch tunnel will be

generated for these two requirements. Thus 4 branch tunnels : $T_3$= *(1,5),* $T_4$= *(2,5),* $T_5$= *(3,5),* $T_6$= *(4,5)* are established for $G_0$, and 2 for $G_1$ : $T_7$= *(7,9),* $T_8$= *(8,9)*.

Now the one-hop requirements removed earlier can be put back and established, thus $T_9$= *(2,3)*.

Onto the redundancy check. Sorted by *from* in descending order, the check will start from the farest tunnel, which is $T_2$, thus $M_a$=*(9, 10)*.

**Table 2.** An Example of Eight Requirements

| Requirement | |
|---|---|
| $Req_1$ | SCR(E, 7, 9) |
| $Req_2$ | SCR(E, 1, 5) |
| $Req_3$ | SCR (A, 2, 3) |
| $Req_4$ | SCR (E, 2, 6) |
| $Req_5$ | SCR (E, 3, 5) |
| $Req_6$ | SCR (E, 4, 6) |
| $Req_7$ | SCR (E, 8, 10) |
| $Req_8$ | SCR (A, 3, 6) |

**Table 3.** Sorted Requirement List

| Sorted Requirement | |
|---|---|
| $Req_2$ | SCR (E, 1, 5) |
| $Req_4$ | SCR (E, 2, 6) |
| $Req_5$ | SCR (E, 3, 5) |
| $Req_8$ | SCR (A, 3, 6) |
| $Req_6$ | SCR (E, 4, 6) |
| $Req_1$ | SCR (E, 7, 9) |
| $Req_7$ | SCR (E, 8, 10) |

$T_8$ is then compared with $M_a$, and it is found that $M_a.from = T_8.to$, indicating that $T_8$ are connected to current $M_a$, therefore $T_8$ is merged with $M_a$, thus $M_a$ = *($T_8$.from, $M_a$.to)*.

$T_7$ is then compared with $M_a$, and it is found that $M_a.from \neq T_7.to$, indicating that $T_7$ are neither redundant nor connected with $M_a$, therefore $M_a$ = *($T_7$.from, $T_7$.to)*.

While checking $T_1$, since $T_1.to > M_a.from$, indicating that $T_1$ is not connected with $M_a$, thus $M_a$ is set to *($T_1$.from, $T_1$.to)*.

Onto the checking of $T_6$, $M_a.from = T_6.to$, indicating $T_6$ is connected with $M_6$, therefore $T_6$ is merged into $M_a$, thus $M_a$=*($T_6$.from, $M_a$.to)*.

$T_5.to < M_a.from$, indicating that $T_5$ is not connected with $M_a$, thus $M_a$ = *(T5.from, T5.to)*.

Onto the checking of $T_9$, $M_a.from=T_9.to$, thus $M_a$ = *($T_9$.from, $M_a$.to)*.

Onto the checking of $T_4$, it is found that both $T_4.from$ and $T_4.to$ equals to those of $M_a$, thus $T_4$ is considered redundant, and is removed.

Finally, $T_3$ passed the checking with $M_a$, thus the redundant check is completed. The final result is shown in Fig. 5.

Note that the goal of this approach focuses on rapidly establishment of non-conflict tunnels. Once a tunnel is established, its attributes could be determined right away. Since this algorithm shares the same goal with both Order-Split and Conflict-Free approach, and since these two are proven so far to be out-perform other approaches, thus here Zero-Conflict is compared with them. Shown in Table 4 are the numbers of resulting tunnels of Order-Split, Conflict-Free, as well as Zero-Conflict, generated from requirements in Table 2. It can be observed that Zero-Conflict yields same results for this case.

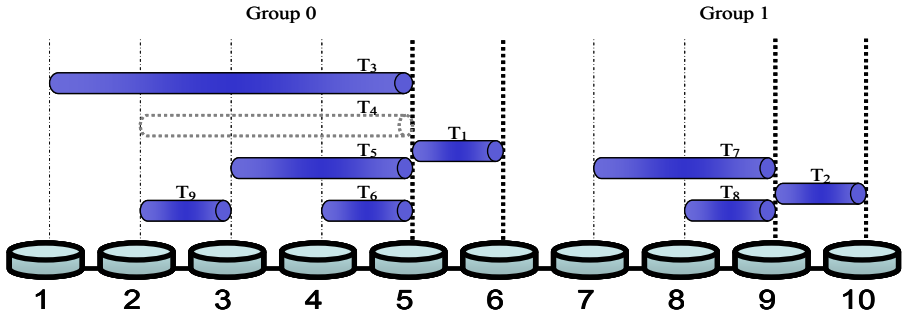



**Fig. 5.** The Solution for the Example of **Table 2** by Using Zero-Conflict Algorithm

**Table 4.** The Compare of Three Algorithms

| Approach | Total Number of Tunnels |
|---|---|
| Ordered-Split Approach | 8 |
| Conflict-Free Algorithm | 8 |
| Zero-Conflict Algorithm | 8 |

## 5 Time Complexity Analysis

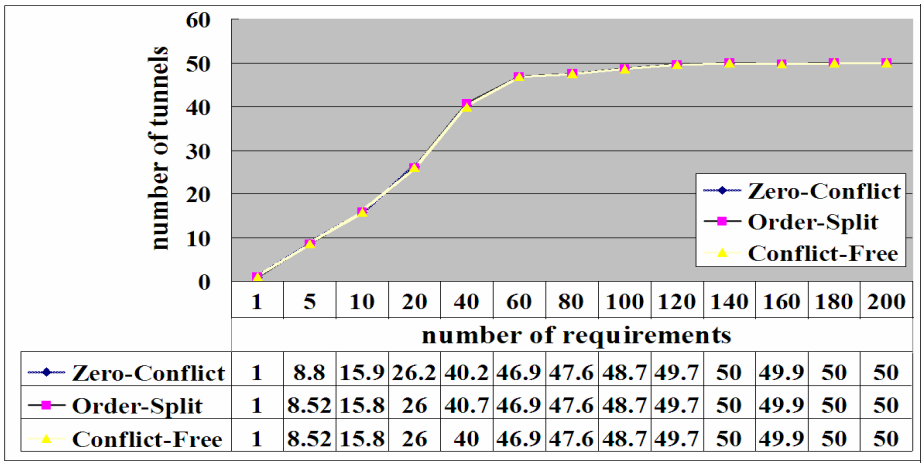
The proposed Zero-Conflict Algorithm generates cut points right after security requirements are acquired. Checking for conflicts are unnecessary, since possible cases are successfully avoided. Removing of 1-hop requirements, grouping, and the three phases of tunnel building, are all *O(n)* operations. However several steps in the algorithm employed sorting operation, such as the sorting of requirement list, and gathering of the cut point list, which raised the over-all time-complexity to *O(nlogn)*.

In the final redundancy removal, the generated tunnels are sorted. It should be noted that in this approach, $n$ input requirements will generate at most $2n$ tunnels. Assuming there are $x$ 1-hop requirements in these $n$ requirements, then there will be at most $n$-$x$ bus tunnels, $n$-$x$ branch tunnels, and $x$ 1-hop tunnels. Therefore the maximal number of generated tunnels is $2n$-$x$. Since $x \leqq n$, it is thus proven that $n$ requirements generated at most $2n$ tunnels, making redundancy removing itself a

*O(nlogn)* operation. Thus the time-complexity of Zero-Conflict is bounded in *O(nlogn)*, which, in comparison with Order-Split and Conflict-Free, is significantly more efficient, as well as scalable.

## 6 Simulation Results

To show that Zero-Conflict, in addition of being fast, generates no more tunnels then existing approaches, a simulation was conducted. The simulator for Zero-Conflict algorithm was implemented under Windows platform. The simulation program takes a requirement file as input, and outputs a file containing generated tunnels. The Order-Split and the Conflict-Free approaches were also implemented. The performances of these algorithms were tested with inputs ranging from 1-200 requirements, each randomly generated 1000 times. The results of average amount of tunnels are shown in Fig. 6. The X-axis represents the number of the requirements input, while the Y-axis represents the number of tunnels generated. It could be seen that the result of Zero-Conflict is close Order-Split and Conflict-Free. Noted that under the assumption that the end nodes of all requirements are cut points, a tunnel covering *n* cut points will be divided into *n+1* connecting tunnels, which would raise the number of resulting tunnels. However, the simulation results show that the average number of tunnels generated by the proposed Zero-Conflict approach meets (or in some cases, beats) the results of most known approaches.




| number of requirements | 1 | 5 | 10 | 20 | 40 | 60 | 80 | 100 | 120 | 140 | 160 | 180 | 200 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Zero-Conflict | 1 | 8.8 | 15.9 | 26.2 | 40.2 | 46.9 | 47.6 | 48.7 | 49.7 | 50 | 49.9 | 50 | 50 |
| Order-Split | 1 | 8.52 | 15.8 | 26 | 40.7 | 46.9 | 47.6 | 48.7 | 49.7 | 50 | 49.9 | 50 | 50 |
| Conflict-Free | 1 | 8.52 | 15.8 | 26 | 40 | 46.9 | 47.6 | 48.7 | 49.7 | 50 | 49.9 | 50 | 50 |

**Fig. 6.** The Average Number of Tunnels in the Network of 50 Routers

## 7 Conclusion

This paper proposed a Zero-Conflict algorithm, an automatic policy construction algorithm which is able to predict and avoid conflict in advance by using *requirement*

*groups* and *cut points* techniques. Moreover, the worse case of time-complexity of this approach is only *O(nlogn)*, which as far as we know, beats most known approaches, whose worse cases of time-complexity are at least $O(n^2)$. Thus it is shown that by avoiding possible cases for conflicts, this approach yields both satisfying efficiency as well as effectiveness so that the resource for network management and the performance of the entire network is further improved.

In addition, most preceding algorithms are suitable for *central* processing, whereas security requirements are dealt with only after *all* of them are collected. The proposed concept of cut point prediction is more suitable for distributed processing. Future works can be made on utilizing this concept on constructing distributed processing algorithms.

## References


1. M. Blaze, A. Keromytis, M. Richardson, and L. Sanchez, "IP Security Policy (IPSP) Requirements," RFC 3586, IPSP Working Group, August 2003.
2. C. L. Chang, Y. P. Chiu, and C. L. Lei, "Automatic Generation of Conflict-Free IPSec Policies," International Conference on Formal Techniques for Networked and Distributed Systems, pp. 233-246, October 2005.
3. J. Conover, "Policy-Based Network Management," Network Computing, Vol. 10, No. 24, pp. 44-50, November 1999.
4. Z. Fu and S. F. Wu, "Automatic Generation of IPSec/VPN Security Policies in an Intra-Domain Environment," 12th International Workshop on Distributed Systems: Operations & Management (DSOM 2001), pp. 279-290, 2001.
5. Z. Fu, S. F. Wu, H. Huang, K. Loh, F. Gong, I. Baldine, and C. Xu, "IPSec/VPN Security Policy: Correctness, Conflict Detection, and Resolution," IEEE Policy 2001 Workshop, pp, 39-56, 2001.
6. H. Hamed, E. Al-Shaer, and W. Marrero, "Modeling and verification of IPSec and VPN security policies," 13th IEEE International Conference on Network Protocols (ICNP 2005), Vol. 0, pp. 259-278, November 2005.
7. S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," RFC 2401, Internet Society, Network Working Group, November 1998.
8. M. Li, "Policy-based IPSec management, " Network, IEEE, Vol. 17, No. 6, pp. 36-43, November 2003.
9. J. D. Moffett, "Requirements and Policies," Position paper for Workshop on Policies in Distributed Systems, HP- Laboratories, November 1999.
10. J. D. Moffett and M. S. Sloman, "Policy Hierarchies for Distributed Systems Management," IEEE Journal on Selected Areas in Communication, Vol. 11, No. 9, pp. 1404-1414, December 1993.
11. M. Sloman, "Policy Driven Management for Distributed Systems," Journal of Network and Systems Management, Vol. 2, No. 4, pp. 333-360, December 1994.
12. Y. Yang, C. U. Martel, and S. F. Wu, "On Building the Minimal Number of Tunnels - An Ordered-Split approach to manage IPSec/VPN policies," 9th IEEE/IFIP Network Operations and Management Symposium (NOMS 2004), Vol.1, pp. 277-290, April 2004.
13. Y. Yang, Z. Fu, and S. F. Wu, "BANDS: An Inter-Domain Internet Security Policy Management System for IPSec/VPN," 8th IFIP/IEEE International Symposium on Integrated Network Management 2003, pp. 231- 244, March 2003.