Firewall Policy Reconstruction by Active Probing: An Attacker's View

Taghrid Samak, Adel El-Atawy, Ehab Al-Shaer,
School of Computer Science, Telecommunication, and Information Systems
DePaul University
Chicago, Illinois 60604
{ taghrid,aelatawy,ehab} @cs.depaul.edu

Hong Li
Information Technology Research
Intel Corporation
Folsom, CA 95630
hong.c.li@intel.com

***Abstract*— Having a firewall policy that is correct and complete is crucial to the safety of the computer network. An adversary will benefit a lot from knowing the policy or its semantics. In this paper we show how an attacker can reconstruct a firewall's policy by probing the firewall by sending tailored packets into a network and forming an idea of what the policy looks like. We present two approaches of compiling this information into a policy that can be arbitrary close to the original one used in the deployed firewall. The first approach is based on region growing from single firewall response to sample packets. The other approach uses split-and-merge in order to divide the space of the firewall's rules and analyzes each independently. Both techniques merge the results obtained into a more compact version of the policies reconstructed.**

I. Introduction

Identifying the anatomy of the victim's defences is always an asset to any attacker. In computer networks, the attacker is always keen to know every possible detail about his target network or corporate. The information can be the topology of the network, the IP's used for servers and workstations inside the network, and the defense mechanisms available. Firewalls are currently considered one of the cornerstones of any network defense mechanism. Firewalls' main purpose is to filter out packets coming to (or going out of) the network from (to) external sources based on a given policy that represents the need and strategy of the network's owners and administrators.

The firewall policy consists of a list of rules, with each rule representing a set of conditions. If an incoming packet matches all of these conditions, then a certain action is taken: allow the packet to pass or drop this packet immediately. A packet can match the conditions of more than one rule, in such a case, the first rule will have priority and its action (*i.e.*, , allow/reject) will be applied to the packet. So, logically, the firewall checks these rules sequentially, one by one, till a rule is matched by the packet. However, to optimize the filtering operation, firewalls rarely perform the rule matching sequentially, and many techniques are used to provide the same policy semantics with a much faster matching approach.

In this paper, we study the possibility that an attacker can actively probe the firewall in order to deduce the policy used to protect his target network. By designing a technique that can perform such task, we show that this vulnerability is a real threat and it is not valid for a network administrator to assume that the attacker does not have a copy of his policy (or at least a semantically equivalent one). Using the shortcomings in the policy itself, the attacker can target the protected network via erroneously opened protocols/ports and address ranges. Moreover, we claim that a more dangerous attack can be launched if the attacker starts to heavily send packets that will only be matched by the default rule of the firewall. In this case, the motive of the attacker is not targeting a server/host within the network, but the firewall itself. This is due to the fact that packets that are rejected by the default rule only are the most expensive packets with respect to filtering time. Thus, such attack will be a form of a denial of service (DoS) attack against the firewall itself.

To our knowledge, very little work was directed to attacking the firewall actively in order to obtain the policy it implements [3]. Most of the available work addresses the problem on a host-by-host basis, and without using any knowledge about the nature of firewall ACL rules, making such techniques less likely to be scalable or automated. However, some of firewall analysis tools are concerned with policy discovery using specific queries to help administrators understand and modify configurations [10]. Those methods work well given extra information about network topology and configuration. Our main focus instead is to discover the whole policy without knowing anything about the internal network structure. Other firewall analysis tools focus on the performance of firewall in terms of implementation and filtering delays [9] and [6]. Some work has been done where the analysis was performed to test firewalls for the vulnerability to traffic-specific attacks, as IP spoofing attacks which was addressed in [11]. In [8], performance metrics for vulnerabilities resulting from firewall operations are presented and analyzed.

We present two simple algorithms that can each obtain good portion of the policy with accuracy and resolution that can be enhanced by providing the attacker more time to probe the firewall. The first algorithm uses region-growing like technique, where the attacker starts with a random sample point then searches in all directions to identify the whole rule. The second algorithm uses the split-and-merge technique to partition the whole space (*i.e.*, space consists of different packet header values) into partitions and investigate each

This research was supported in part by Intel IT. Any opinions, findings, conclusions or recommendations stated in this material are those of the authors and do not necessarily reflect the views of the funding sources.

1-4244-0773-7/06/$20.00 ©2006 IEEE

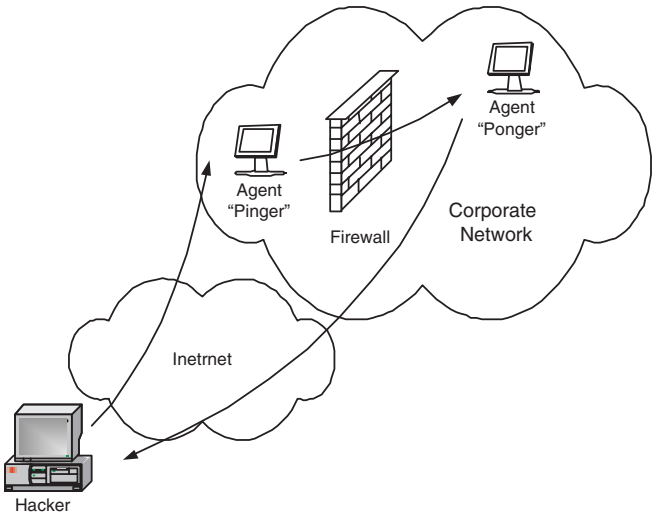


Fig. 1. Attacker System Structure

partition.

The remaining parts of the paper are organized as follows. Section II describes the structure of the system an attacker can use. In section II-A, the problem of locating and selecting the attacker's insiders agents is explained along with some proposed ideas and solutions. Sections III and IV explain the two techniques for reconstructing the policy based on the observations obtained by the agents. Section V discusses general guidelines on appropriate selection of test packets. Evaluation of the system is presented in section VI followed by the conclusion and future work in section VII.

II. Structure

A general design of the attacker plan is showed in Figure 1. He uses two very simple agents; the first sends what the attacker requests into the firewall "Pinger" and the other receives these packets and return them back (most probably encapsulated) to the attacker for analysis "Ponger". Obviously, the "Ponger" works as a sniffer in the inside of the attacked network.

A. Agent Location and Selection

The reason behind having the "Pinger" agent is that it facilitates the injection of packets into a firewall from a close location to the firewall without the actual/physical presence of the attacker's machine. Also, it has a better shot in sending packets that otherwise would have been blocked by other routers and firewalls upstream (*i.e.*, works as a tunneling gateway).

The benefits of having the "Ponger" are similar to the ones mentioned for the "Pinger". However, in this case, it is more necessary to have an agent inside the network as it is much harder to embed the main system of the hacker directly behind the firewall from the inside. These agents can be in the form of harmless worms, that can stay stealth till they are required by the attacker to ping (or pong) his probe packets through (or from) the firewall.

If the agents were embedded by a worm like transmission, it is highly probable that the attacker will have the luxury to choose among different instances of his agents. In our experiment, we assume that the attacker will choose his two agents to have the shortest distance between them. In other words, a quick experiment will be performed to measure the transmission delay between different pairs of his agents such that each pair has a representative on each side of a firewall, and then the attacker will choose the pair closest to each other (and consequently to the firewall).

B. Basic Operations

The only primitive operation that the attacker uses is sending a packet and waiting to see wether it is going to be allowed by the firewall (*i.e.*, seen by the "Ponger" agent, and - optionally - measuring the latency) or it is going to be filtered by the firewall (*i.e.*, not received at the other side by the "Ponger").

Every packet sent is a point in the traffic space, which is the space composed of all possible values of the dimensions being investigated (*i.e.*, the space composed of different values for source/destination IP address and port). Every packet (*i.e.*, point) has one of two values (or signs), either allowed or denied (positive or negative). Thus a single test packet sent will evaluate the filtering policy at a single point in order to identify the hyper rectangles covered by each one of the policy rules. We will use the notion of points and signs to simplify the discussion and analysis.

Running an exhaustive sampling of all possible packets in order to reconstruct the policy will have a 100% accuracy, but will simply be impossible because of the huge space needed (*i.e.*, $2^8 * 2^{32} * 2^{32} * 2^{16} * 2^{16} = 2^{104}$). Therefore, an intelligent sampling is essential to exploit the way by which we specify firewall rules. Almost all firewalls restrict rule syntax to be a conjunctive of conditions on packet header fields, and each field is checked against a range of consecutive values. For example, a very short firewall policy might look like:

```
R1: deny icmp any:any 150.160.170.*:any
R2: deny tcp any:any 150.160.170.*:1024-65535
R3: allow tcp any:any 150.160.170.*:80
R4: deny any any:any any:any
```

As we can see, on each dimension the condition is stated as a single value or a range of consecutive values (including the "any"). Thus, the overall space that falls within a rule is a hyper rectangle, and our task is to identify its boundaries.

III. Method I: Region Growing Approach

In this method, we start assuming the default "deny all" rule. By sampling from the space, we wait till a packet passes through the firewall indicating a "permit" rule, and this packet will be called the rule "kernel". We then perform an exponential search in each of the d dimensions (*e.g.*, $d = 5$ for the case of a firewall that uses the protocol, source and destination address and port fields to specify rules), to find a change in the sign that indicate the end of the rule in that dimension. Following the exponential search comes a binary search to pinpoint the exact boundary of the rule's space (See

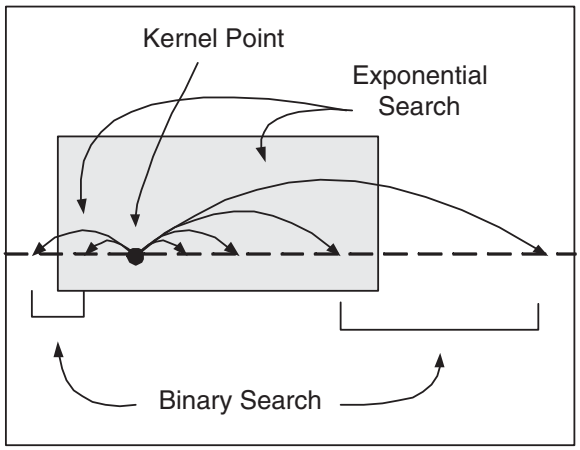


Fig. 2. Searching for rule boundaries starting from a kernel

Algorithm 1 Extract_Policy(maxCost)

1: $sign = -1$ {default is deny}
2: $P \Leftarrow$ Extract_Recurse(global bounds, -1, maxCost)
3: Apply Policy Simplification Techniques on P
4: **return** P

Algorithm 2 ExtractRecurse(rule bounds, sign, maxCost)

1: $P \Leftarrow \Lambda$
2: sampleCount $\leftarrow 0$
3: **repeat**
4: Increment sampleCount
5: $< sample > \Leftarrow$ SelectSample()
6: Inject ($< sample >$) {Send packet to Pinger}
7: sampleSign = WaitResult() {Wait for reply of Ponger}
8: **until** $(sampleSign * sign < 0)\, OR\, (sampleCount > maxCost)$
9: **if** sampleCount $>$ maxCost **then** {No rule found}
10: **return** P
11: **end if**
12: **for all** $i \in$ Dimensions **do**
13: new_rectangle.$min_i \leftarrow$ Search on dimension i, decreasing
14: Adjust sampleCount
15: new_rectangle.$max_i \leftarrow$ Search on dimension i, increasing
16: Adjust sampleCount
17: **end for**
18: $P \Leftarrow$ Rule(newRectangle, action=-sign)
19: **if** sampleCount $<$ maxCost **then**
20: max_cost $\leftarrow (maxCost - sampleCount)/3^d$
21: $P \Leftarrow P \cup$ ExtractRecurse(newRectangle, -sign, maxCost)
22: $P \Leftarrow P \cup$ ExtractRecurse for all other subspaces
23: **end if**
24: **return** P

Figure 2). These two steps takes only $O(\lg n)$, where $\lg n$ is the number of bits in each one of the fields (*i.e.*, IP=32, port=16, protocol=8 bits).

Once the boundaries of the rule are identified, we start sampling inside the rule searching for any rules that are exceptions to this rule (*i.e.*, searching for a kernel). This is the same operation performed previously as we searched for this rule as an exception of the "default deny" rule, and so on.

In Algorithm 1, we simply start with the network-wide limits as the default sampling space, which represents the "default deny" rule. The value of -1 assigned to the sign parameter indicates the deny action as mentioned before. A parameter should be provided by the attacker which is the $maxCost$, that indicates the maximum number of packets the algorithm should use. This limit is chosen depending on the time he has to discover the policy and the maximum number of packets that can be used without triggering any IDS system or get the Administrator attention. After calling the recursive function, the policy will be simplified by any of the already available rule simplification procedures as the ones mentioned in [5].

In Algorithm 2, we start with initializing the policy discovered by this recursive branch as an empty policy and we set the counter for samples used to be equal to zero. By subsequent selection of packets from the space under investigation; we search for the kernel of a rule (some of the guidelines for such sample selection will be discussed in V). Once a kernel is found, the search for the boundaries over each dimension is performed using exponential search to enclose the region edge and a binary search to pinpoint the exact value. Following this step, we have to check if we still have packets to use (*i.e.*, making sure we did not pass the limits in order to stay

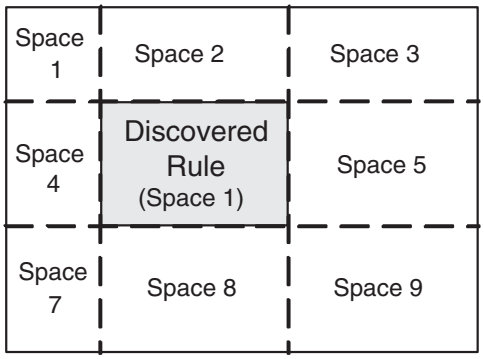


Fig. 3. Partitioning the space to subspaces for further investigation

stealth). Once the rectangle of the newly identified rule is obtained, we partition the original space (as given by the caller function) as shown in Figure 3. Each subspace (including the rule subspace) will be recursively analyzed with an equivalent allowance of the maximum cost remaining (maximum number of packets to use).

IV. Method II: Split-and-Merge Approach

In this method, we use the split-and-merge algorithm that is used for image segmentation [4]. The main objective is to partition the space into n non-overlapping regions based on a partitioning criterion. In our case the regions will represent individual policy rules, and the criterion will be the action of the rule and its shape. To formalize our problem we consider the whole space as the initial region R which will be the default "deny" rule that covers all the space. We wish to partition R into n sub-regions (sub-rules) $R_1, R_2, \ldots, R_n$ such that:

1) $\bigcup_{i=1}^{n} R_i = R$
2) R_i is a connected region
3) R_i has a rectangular shape (policy rule restriction)
4) $R_i \bigcap R_j = \Phi$ for all i and j where $i \neq j$
5) $P(R_i) = TRUE$ for $i = 1, 2, \ldots, n$
6) $P(R_i \bigcup R_j) = FALSE$ for $i \neq j$

where $P(R_i)$ is a logical predicate applied to region R_i. In the case of rule identification, the predicate value will be whether all points (packets belonging to this sub-space) in this region have the same action (*i.e.*, receive the same treatment from the firewall: permit/deny). The third condition is specific to our application, so each detected rule will have a rectangular shape.

The following is a simple description for the split-and-merge algorithm:

1) Split into four disjoint quadrant any region R_i for which $P(R_i) = FALSE$.
2) Merge any adjacent regions R_i and R_j for which $P(R_i \bigcup R_j) = TRUE$.
3) Stop when no further merging or splitting is possible.

This algorithm restricts the splitting to equal quadrant. The splitting can also be performed in a more general way depending on the space of the problem. For testing the predicate $P(\ldots)$ sampling is preformed to select which points in space to consider for evaluating the predicate. First the whole space is divided into four quadrants then recursively each quadrant will be investigated and split. The predicate is evaluated on the lower level, then the merging is performed on the way up to the whole space. The sampled points for our algorithm will be the packets sent by the attacker. The limits on the maximum total number of samples used will be used as the recursion stopping criterion (as described in section III). In Section V, we describe some sampling guidelines, which will consider also the biasing towards lower and/or popular values in the space.

So far, our discussion assumes that the data are in two dimensional space. For the general case of d dimensions (multiple packet header fields), slight change is required to the algorithm. The rule will have a d-dimensional hyper-rectangular shape. The splitting also will be performed considering the higher dimensionality. Instead of splitting each region into four quadrants, the region will be partitioned into 2^d partitions.

Algorithm 3 explain the details of split-and-merge approach for policy reconstruction.

Algorithm 3 Split-and-Merge(ruleBounds, sign, maxCost)

```
1: P ⇐ Rule(ruleBounds, action=sign)
2: sampleCount ← 0
3: Partitions ⇐ PartitionSpace(ruleBounds)
4: UsefulPartitions ⇐ Λ
5: repeat
6:    Increment sampleCount
7:    currentPartition = Partitions.current
8:    < sample > ⇐ SelectSample()
9:    Inject (< sample >)                    {Send packet to Pinger}
10:   sampleSign = WaitResult()              {Wait for reply of Ponger}
11:   if sampleSign*sign¡0 then
12:      Partitions.delete (currentPartition)
13:      UsefulPartitions.add (currentPartition)
14:      if Partitions.IsEmpty then
15:         break
16:      end if
17:   end if
18: until (NOT UsefulPartitions.IsEmpty)AND(sampleCount > α×maxCost)
19: if UsefulPartitions.IsEmpty then {No rule found}
20:    return P
21: end if
22: N ← UsefulPartitions.Size
23: for all p ∈ UsefulPartitions do
24:    P ⇐ P∪ Split-and-Merge(p.bounds, -sign, maxCost/N)
25: end for
26: Merge(P) {Merge based on same sign}
27: return P
```

V. Sampling Guidelines

Network Administrators have some tendencies and patterns in writing their firewall policies. However, there is no established study about these behaviors except that policies are always not perfect. Some of the policies might lack required rules for preventing well known vulnerabilities from being exploited inside their networks [2], [7], [10]. Other problems arise from the policy structure, and rule-rule interaction [1]. In this work, however, we are focused on obtaining the firewall policy even if it does not contain any of the above mentioned problems; emphasizing on the fact that even perfect policies are not truly perfect if they are known to the attacker. Some of the administrators patterns and common practices will come in handy when selecting the ideal sampling strategy.

From observing many firewall policies as well as our experience in the field, we believe that the following are common practices that can be exploited to state our sampling guidelines:

- IP addresses are allocated for servers at first, and using the lowest IP values for the network management servers (*e.g.*, Gateways, DNS servers, etc). Following this small range comes the business/functionality servers (*e.g.*, web servers, DB servers, etc). Client IPs are either statically allocated from a medium to high range, or dynamically allocated in more of a high range.
- Ports used on all servers follow some common practices and well known values (aside from the standard port constants).
- Source IP addresses are rarely - if ever - used in firewall policies.
- The protocol field and the destination port, almost always come in pairs. In other words, mentioning a port value in the rule (*i.e.*, not an "any" value) means the protocol field will be specified as well.

Although this list is very far from complete, it can help us design better-than-random strategies to select our test packets. The following is the strategy used in the $SelectSample()$ function:

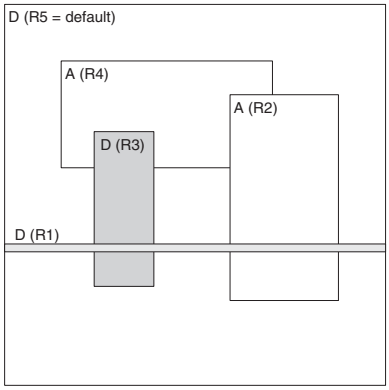


Fig. 4. A Test Policy of 5 Rules

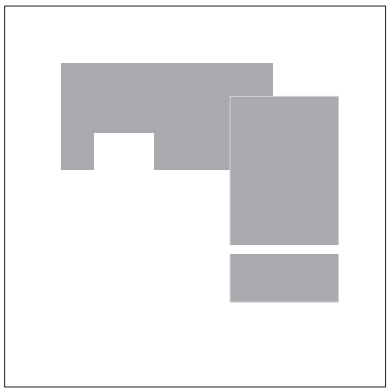


Fig. 5. Accept space of the test policy (Deny space due to rules or default action are indistinguishable))

- A percentage is allocated to popular values. For example, standard port values, and 0, 1 and 255 values for bytes of the IP addresses. Of course these values are only selected when they fall within the area under consideration.
- The rest is distributed over the space, given a slight bias towards smaller values. For the Split-and-Merge algorithm, this also affects the splitting boundaries (*i.e.*, non-equal quadrants).
- Generating some of the samples to be in streaks, having consecutive values in one or more the dimensions. For example, generating consecutive packets with address and port equals (10.11.12.13:5430), and (10.11.12.14:5431). These lines in space help captures these rules that cover a simple line (as seen in the example policy in Figure 4).
- With a lower ratio, some packets should be assigned to cover parts of the boundary of the space investigated.

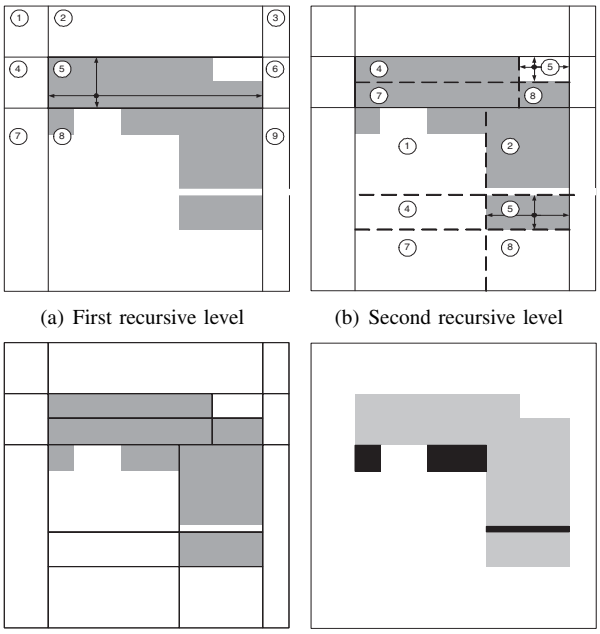
VI. Test Case Demonstration

We are going to use the policy in Figure 4 as a test case to show the behavior of the proposed methods. This policy consists of five rules including the default deny rule. The rules in this policy use only two dimensions (source and destination address).

A. Region Growing Method

The technique starts with sampling the space till the first sample with opposite sign (*i.e.*, accept instead of deny) is observed. This sample is shown as the black dots in Figure 6. These dots become a kernel; for region growing till the boundaries of the hit rule are identified. The final rules obtained are shown in Figure 6(c), and the error from the original policy is shown in Figure 6(d) marked in black color.

The overall error was 3.26% relative to the whole space (*i.e.*, error region by total space), and the error in identifying the accept space was 12.11% (*i.e.*, error region by accept space).



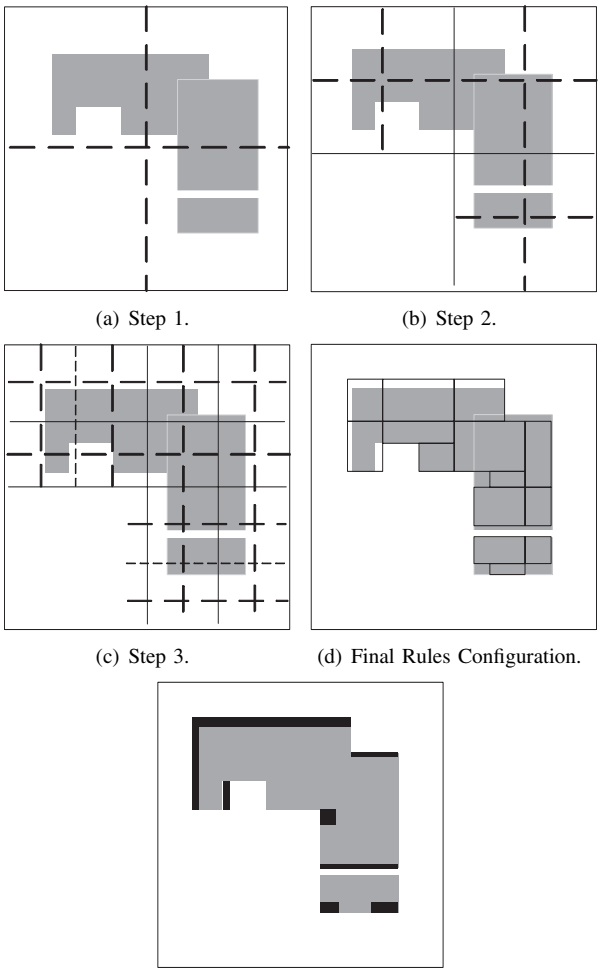
(a) First recursive level (b) Second recursive level

(c) Final Rules Configuration (d) Error in Identified Rules

Fig. 6. Running steps for the region growing technique.

B. Split-and-Merge Method

For the policy defined in figure 4, we are only interested in the rules having "accept" action. Figure 5 shows an equivalent policy with respect to the accepted traffic. The split-and-merge algorithm will generate a set of rules (rectangles) that cover the shaded parts of figure 5. The steps of running the algorithm are demonstrated in figure 7. Each step shows the current divided subspaces. The heavy lines show the current splitting borders, where the light ones indicate previous steps. The undivided area means that the predicate was true at this region (same action for the rule). The final rules configuration detected by the algorithm is shown in figure 7(d). The bordered rectangles represent each individual rule. In this case the algorithm has stopped at the third level of splitting. The set of rules detected by this method is larger than the original policy set, but is equivalent in terms of rule actions and dimensions. Errors from the original policy configuration is shown in figure 7(e), where the black areas represent the error regions.



(a) Step 1. (b) Step 2.

(c) Step 3. (d) Final Rules Configuration.

(e) Error in Identified Rules.

Fig. 7. Running steps for split-and-merge algorithm.

The overall error was 4.4% relative to the whole space (*i.e.*, error region by total space), and the error in identifying the accept space was 16% (*i.e.*, error region by accept space).

VII. Conclusion and Future Work

In this paper, we presented two approaches that can be used by attackers or intruders to reveal firewall policies and network topology of a target network. Using active probing from outside and inside the target network, the attacker can build up an image of the global firewall's policies in the network. By planting agents inside the network in places that are selected carefully to increase measurements accuracy, the attacker is able to reconstruct the policy with reasonable success. Applying the technique to simple sample scenarios, the error was as low as 5% of the firewall policy address space.

Future Work

- *Agent Selection:* Choosing agent locations or selecting from available agents based on their location is not a trivial problem. Considering the blind spots (*i.e.*, packets with specific ranges of IPs that will never reach the "Ponger" due to packet routing), the agent selection problem will get much more complicated.
- Adaptive agent selection based on the part of the space that is currently being sampled. This can enhance the visibility as we can escape from the blind spots problem by selecting another agent that can observe these packets. However, this needs some information about the routing and topology of the victim network.
- *Enhancing the sampling strategy:* By choosing the packet samples used in a more clever way, we can exploit more of the administrator/network patterns. Using better tables for well known ports and IPs can enhance the overall accuracy.
- *Enhancing the technique using packet latency to tune the rule merging step:* Packets that suffer different time latencies from the firewall are more probable to belong to different rules. However, a calibration phase is needed to make sure the "Pinger"/"Ponger" pair are close enough and do not suffer from high cross traffic activity that might lower the accuracy of the measurements.

Many ideas are also being investigated to tighten the gap between the reconstructed and the original policies. However, the next step essentially is to investigate and develop counter techniques to make security policies/devices more obsecure and hard to discover for better protective networks.

References

[1] Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM*, 2004.

[2] CERT/CC. Cert/cc, overview incident and vulnerability trends. April 2006.

[3] David Goldsmith and Michael Schiffman. Firewalking: A traceroute-like analysis of ip packet responses to determine gateway access control lists, http://www.packetfactory.net/firewalk/firewalk-final.html, October 1998.

[4] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*, chapter 10, pages 615–617. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2^{nd} edition.

[5] Mohamed G. Gouda and Alex X. Liu. Firewall design: Consistency, completeness, and compactness. In *ICDCS*, pages 320–327, 2004.

[6] B. Hickman, D. Newman, S. Tadjudin, and T. Martin. Benchmarking methodology for firewall performance, 2003.

[7] Ken Cutler John Wack and Jamie Pole. Guidelines on firewalls and firewall policy. Technical report, January 2002. Special Publication 800-41.

[8] Seny Kamara, Sonia Fahmy, Eugene Schultz, Florian Kerschbaum, and Michael Frantzen. Analysis of vulnerabilities in internet firewalls. *Computers and Security*, 22(3):214232, 2003.

[9] Michael R. Lyu and Lorrien K.Y. Lau. Firewall security: Policies, testing and performance evaluation. *compsac*, 00:116, 2000.

[10] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A firewall analysis engine. *sp*, 00:0177, 2000.

[11] Voravud Santiraveewan and Yongyuth Permpoontanalarp. A graph-based methodology for analyzing ip spoofing attack. *aina*, 02:227, 2004.