Assisted Firewall Policy Repair Using Examples and History

Robert Marmorstein and Phil Kearns – The College of William & Mary

ABSTRACT

Firewall policies can be extremely complex and difficult to maintain, especially on networks with more than a few hundred machines. The difficulty of configuring a firewall properly often leads to serious errors in the firewall configuration or discourage system administrators from implementing restrictive policies.

In previous research, we developed a technique for modeling firewall policies using Multiway Decision Diagrams and performing logical queries against a decision diagram model. Using the query logic, the system administrator can detect errors in the policy and gain a deeper understanding of the behavior of the firewall. The technique is extremely efficient and can process policies with thousands of rules in just a few seconds. While queries are a significant improvement over manual inspection of the policy for detecting that errors exist, they provide only limited assistance in repairing a broken policy. In this paper we present two extensions to our work, examples and history, which enable the administrator to more easily repair a policy which contains errors.

An example is a representative packet which illustrates that the firewall complies with or (more importantly) deviates from its expected behavior. History records the specific rules involved in the deviation. Examples and history provide guidance in finding and fixing faults in a firewall rule set. These contributions can be also be used with the equivalence class analysis to reduce the burden of designing a complicated set of assertions.

Introduction

The administrator who maintains a restrictive firewall policy on a large network must spend a considerable amount of time and effort updating and testing the filtering rules. Requests for new services, changes in the physical topology of the network, and the emergence of new security threats require continual modification of the policy. As the policy changes and grows, it can be difficult for the administrator to avoid introducing errors into the rule set. Repairing these errors is often very challenging. Firewall policy errors are subtle and difficult to detect. Even when the existence of an error is obvious, discovering the source of the problem and correcting it can be tedious and expensive.

In previous work, we introduced techniques for quickly and easily validating a firewall policy using logical queries against a Multiway-Decision Diagram model of the firewall policy. The MDD approach is very efficient (complex queries involving rule sets with hundreds of rules usually take only a few seconds) and allows for very flexible identification of errors. Like most existing approaches, the MDD query technique addresses the issue of testing the firewall for errors, but leaves the problem of repairing the policy entirely up to the administrator. Because tracing through dozens or perhaps hundreds of correct rules to find the two or three critical inconsistencies can take hours or even days, this is a significant burden for administrators of a large network.

Chain Forward (Default Drop)					
#	Target	Source	Destination	Interface	Flags
1	DROP	192.168.1.0/24	anywhere	!eth2	
2	DROP	192.168.3.0/22	192.168.2.0/24	any	
3	ACCEPT	anywhere	192.168.2.4	any	dpt:tcp 80
4	DROP	anywhere	192.168.2.0/24	any	
5	ACCEPT	192.168.1.0/24	anywhere	any	

Figure 1: A rule set which incorrectly blocks access from the 192.168.1.0/24 subnet. Rule 1 of the policy ensures that traffic from the 192.168.1.0/24 subnet arrives on the correct interface. Rule 2 blocks traffic from the insecure wireless network to the server subnet. Rule 3 grants HTTP access to the web server to appropriate hosts. Rule 4 prevents external access to other servers. Rule 5 allows hosts on the trusted subnet to transmit packets that have not been blocked by some previous rule.

In this work, we present two novel techniques that enable "directed repair" of the firewall policy. Using these techniques, the system administrator not only can identify the existence of an error in the policy, but can trace it backs to its root causes without an expensive manual inspection of the rule set. The first major contribution is a technique for providing example packets that illustrate that the firewall violates a set of security requirements. The second contribution creates a history map which identifies the particular firewall rules which cause the firewall to deviate from its desired behavior.

While these techniques do not fully automate the process of repairing the firewall, they do provide the system administrator with information that makes repair much easier than a simple verification of the policy. We have implemented both techniques in ITVal [6], a firewall analysis and repair tool for iptables firewalls. Although we use the Linux iptables firewall for the examples in this paper, it is possible to adapt these techniques to work with other platforms such as PIX and Checkpoint firewalls. There are also several fairly effective scripts for converting ipfw and ipchains firewall to iptables syntax [11] which can be used to adapt such policies to a format compatible with ITVal.

The remainder of this paper is structured as follows. The next sectoin describes the difficulties which a system administrator encounters in repairing a firewall. Then we discuss partially automated repair of the policy. The next two sections detail our techniques for generating examples and history, respectively. Sections on implementation using MDDs and a description how this work can be combined with our previous work on equivalence class analysis of a firewall policy follow. Finally, we discuss related work and make a few concluding remarks.

Firewall Policy Errors

The techniques discussed in our previous work allow a system administrator to perform basic logical queries against a firewall policy using a simple specification language. For instance, to ask which hosts can access a web server, host 192.168.2.4, the administrator can use the query `QUERY SADDY TO 192.168.2.4 AND FORWARD ACCEPTED;` which will list the source addresses of any host that can access the web server without being blocked by the firewall. Inspecting this list of addresses may allow the user to detect an error in the configuration of the firewall. If the address of a malicious host appears in the list, for instance, it is clear that there is a problem with the firewall policy.

Queries allow the system administrator to identify many serious errors in the configuration of a firewall, but provide only a limited amount of information about each error. For instance, if the system administrator uses the query "Which hosts can connect to the mail server?" to determine whether the firewall blocks external hosts, the analysis engine will list those hosts that have unwanted access to the server, but will not provide any additional information that can be used to understand why the firewall failed to prevent access. It may be that the error only occurs when connections are made on a particular network interface or by a particular network protocol. Access to this information could greatly assist the system administrator in repairing the policy, but traditional testing tools do not provide these helpful clues. Another way to think about this is to say that a query engine discovers an error (the ultimate consequence of a problem in the policy), but not the fault (the mistake in the rules that causes the error).

Sometimes, helpful information can be obtained using additional queries or by refining the query to provide more information. Often, however, the process of developing a sufficiently detailed set of queries requires almost as much effort as manual repair of the policy.

This means that query tools are usually limited to detecting whether an error exists and have only limited utility in guiding repair of the policy. To repair the policy by hand, the system administrator must carefully consider each filtering rule to determine whether it is relevant to the error and, if so, whether it is correct. Since most of the rules will usually be either irrelevant or valid, manual repair is a very inefficient and time consuming process, especially when an error has many potential causes.

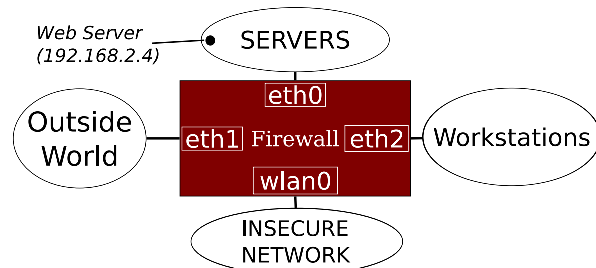

Figure 2: A typical firewall, which protects hosts on two subnets against intrusions from a third, untrusted network and the outside world. One of the protected subnets contains a web server, host 192.168.2.4, to which remote connections are allowed.

Figure 1 shows how difficult it can be to trace a firewall error to its source. This rule set protects workstations on the subnet 192.168.1.0/24 and servers on the 192.168.2.0/24 subnet against attacks from the outside world and an insecure wireless network on the 192.168.3.0/24 subnet. An illustration of the network is given in Figure 2.

The system administrator wants to allow access to the web server, host 192.168.2.4, from any system in the outside world except those on the unsecured wireless network. All other external traffic to the web server should be blocked.

It is fairly easy to determine that the rule set fails to enforce these requirements. If the system administrator opens up a web browser and tries to connect to the web server from a host on the trusted network, the firewall will refuse to allow the connection. Discovering the cause of this error is more challenging, since nearly every rule of the policy plays some role in the filtering decision. An error in rule 4, which drops traffic to the protected subnet, could be the source of the error. A typo in rule 3, which overrides rule 4 to allow web traffic to enter the network might be another the cause. Rule 1, an anti-spoofing rule which blocks traffic from the ‘‘wrong’’ interface, might also be to blame.

As it turns out, the fault that produces the observed error is in rule 2. An incorrect subnet mask in rule 2 causes the firewall to block traffic from the protected network as well as the untrusted net. Manual analysis of the policy requires a careful and tedious inspection of every rule in the policy to identify this fault. For the five rule policy shown here, this inspection might not take too long. However, a policy with more than a few dozen rules would be much more difficult to analyze. Partially automating the repair process in a way that narrows down the potential sources of the error to just one or two rules could save the administrator a significant amount of effort.

Partially Automated Firewall Repair

Unfortunately, it is impossible to fully automate repair of a generic firewall policy because incorrect behavior on one network may be expected behavior on another. For instance, on one network it may be desirable to allow SMTP traffic to reach certain hosts, such as the mail servers. On another network, however, a policy that permits SMTP traffic may spam-bots to compromise important systems. Without input from the user, a repair algorithm cannot distinguish between these two cases.

While a fully automatic strategy for firewall repair is impossible, partial automation is possible. Gouda, Liu, et al. have done significant work on repair of structural errors in the firewall policy [4]. Their technique uses transformation of decision diagrams to produce an improved rule set in which problems such as shadowed or duplicate rules have been eliminated. This strategy does not require any assistance from the user. Unfortunately, these techniques do not address repair of logical errors such as typos or out-of-order rules.

Another approach is to allow the user to make the final decision about how to repair the policy, but automate the process of finding the faults responsible for the error. By providing the system administrator with sufficient information about the possible causes of the error, we can guide her toward a few possible solutions, from which she can choose the one best suited to her network. This ‘‘directed repair’’ of the policy alleviates much of the tedious work required to find faults and fix the policy.

Directed Repair

In previous work [7, 8, 9], we explored ways to detect errors in a firewall configuration using logical queries and an equivalence class decomposition of the network. In this paper, we describe two extensions of this work that enable directed repair of the firewall policy. One technique generates relevant counterexamples from which the system administrator can obtain detailed information about security failures in the policy. The second technique provides an extensive ‘‘history analysis’’ that identifies potential sources of the error and lists rules which should be considered for modification. The history analysis can also be used with the equivalence technique described in [9], which addresses the need for extensive preparation of logical queries. We implement both of these techniques as extensions to ITVal [6], an open source firewall testing tool developed as part of our previous work.

FROM <address_range>
matches all packets with source address in address_range.

TO <address_range>
matches all packets with destination address in address_range .

ON <port_range>
matches all packets with source port in port_range.

FOR <port_range>
matches all packets with destination port in port_range

IN <s>
matches all packets associated with connections in state s

WITH <flag_set>
matches all packets with the TCP flags in flag_set enabled

ACCEPTED <chain>
matches all packets accepted by built-in chain chain

DROPPED <chain>
matches all packets rejected by built-in chain chain

INFACE <iface>
matches all packets received by network interface iface

OUTFACE <iface>
matches all packets transmitted on network interface iface

Figure 3: ITVal primitives.

To use these techniques, the user specifies the desired behavior of the firewall using logical assertions. The syntax for assertions is derived from the query language explained in [8]. The right and left conditions of the assertion are built from a set of simple primitives such as those in Figure 3, which can be combined using the logical operators AND, OR, and NOT to create complex conditions describing sets of packets whose treatment by the firewall requires analysis. For example, we can describe all accepted SSH

packets from subnet 192.168.1.0/24 on interface eth0 using the condition

```
FOR TCP 22 AND
FROM 192.168.1.* AND
INFACE eth0 AND
   (ACCEPTED FORWARD OR
    ACCEPTED INPUT);
```

The user can construct two types of assertions from these conditions. Equality assertions have the form:

```
ASSERT <A> IS <B>
```

where A and B are conditions. Containment assertions have the form

```
ASSERT <A> SUBSET OF <B>
```

Equality assertions specify that those packets which match condition A are exactly those that match condition B. Containment assertions specify that the set of packets that satisfy condition A is (non-strictly) contained in the set of packets that satisfy condition B. Using these assertions, the user can describe important high-level security invariants which the policy should always satisfy.

For instance, the containment assertion

```
ASSERT FROM 192.168.3.*
   SUBSET OF DROPPED FORWARD;
```

specifies that any packet from subnet 192.168.3.0/24 is dropped. The equality assertion

```
ASSERT FROM 192.168.2.*
   IS (FOR TCP 80
     AND ACCEPTED FORWARD);
```

can be used to check that only HTTP packets are allowed to enter the network from the 192.168.2.0/24 subnet and that no other web connections are allowed by the firewall. We call the set of packets that match a condition its *match set* and the set of packets that cause an assertion to fail the assertion's *fail set*. Assertions provide many advantages over simple queries. While queries allow the user to obtain a significant amount of information about the policy, a query does not provide the analysis engine with any description of the expected behavior of the firewall. Therefore, using assertions enables the engine to provide more useful and relevant output.

Counterexamples and Witnesses

One useful advantage of assertion analysis is that it allows generation of relevant counterexamples. These counterexamples provide a context for the error which can often help the administrator discover why a failure has occurred.

The example policy in Figure 4 isolates an untrusted research network 192.168.2.0/24 from the outside world. SSH traffic from the untrusted network to hosts on subnet 192.168.1.0/24 is permitted, but all other traffic from the network is denied. The 192.168.1.0/24 subnet contains several world-accessible web servers to which the policy grants access. However, the rule set blocks connections from 63.118.7.16, a malicious host. Trusted hosts are allowed to make connections to the web servers and an external server, host 131.106.3.253, but cannot make any other connections.

To test whether the untrusted hosts are sufficiently restricted by the firewall, the administrator uses the assertion

```
ASSERT (FROM 192.168.2.*
      AND NOT FOR TCP 22)
      SUBSET OF DROPPED FORWARD;
```

which specifies that only SSH traffic is accepted from hosts on the untrusted network. Due to an error in the ordering of rules 2 and 4, the assertion will fail. This subtle error could be very difficult to detect in a lengthier policy in which the rules were much further apart. Using ITVal, the administrator can easily discover that the assertion fails. Knowing that the assertion does not hold is an important first step, but does not give much information about the cause of the error. To give the user more information about the source of the error, we generate a counterexample – a packet that demonstrates the falsity of the assertion. Figure 5 shows the generation of one possible counterexample. The user specifies that an example should be generated by inserting the keyword EXAMPLE at the beginning of the assertion.

Examination of the counterexample gives the system administrator important information about the assertion failure. One significant clue is that the example packet arrived on interface eth1. Since only rule 2 mentions eth1, this fact draws the administrator's immediate attention to the rule ordering error, which can now be corrected by moving rule 2 to the correct location in the policy.

Sometimes it is desirable to obtain an example even when an assertion succeeds. We call such an

Chain FORWARD (Default DROP)					
#	**Target**	**Source**	**Destination**	**Interface**	**Flags**
1	ACCEPT	anywhere	192.168.1.0/24	eth0	dpt:tcp 22
2	ACCEPT	anywhere	131.106.3.253	eth1	
3	DROP	63.118.7.16	anywhere	eth0	
4	DROP	192.168.2.0/24	anywhere	any	
5	ACCEPT	anywhere	anywhere	any	dpt:tcp 80

Figure 4: An incorrect forwarding chain which allows non-SSH traffic from hosts on the 192.168.2.0/24 network.

example a ''witness,'' since it illustrates the assertion. Witnesses are less powerful than counterexamples in that the existence of a counterexample demonstrates conclusively that an assertion is false while the existence of a witness only demonstrates that it is possible to satisfy the assertion. Nevertheless, witnesses can be very useful for convincing yourself (or others) that an assertion really holds. They can also be useful for debugging certain kinds of problems that can best be tested with an assertion that you expect to fail.

```
ASSERT EXAMPLE (FROM 192.168.2.*
    AND NOT FOR TCP 22)
    SUBSET OF DROPPED FORWARD;
Assertion failed.
Counterexample:
TCP packet from 192.168.2.1:6362[eth1]
        to 131.106.3.253:25[eth1]
        in state NEW with flags[    ].
```

Figure 5: Counterexample for the example assertion.

Suppose that you wanted to ensure that the firewall policy does not block all SMTP connections. To test whether your firewall correctly implements this policy, you might use the assertion:

```
ASSERT EXAMPLE FOR TCP 25
    SUBSET OF DROPPED FORWARD;
```

If the policy is correct, the assertion should fail, since the assertion implies that all SMTP packets are dropped. In this situation, a witness can often provide useful information that allows you to discover why the assertion succeeds. Since using assertions in this manner is very counter-intuitive, we provide the user with NOT SUBSET OF and IS NOT operators that can be used in place of the SUBSET OF and IS keywords. This allows the user to avoid using ''backward assertions'' like the one above. Using these operators, the user can test whether all SMTP packets are dropped as follows:

```
ASSERT EXAMPLE FOR TCP 25
    NOT SUBSET OF ACCEPTED DROPPED;
```

This new assertion will hold in exactly the cases in which the old assertion would fail and vice versa, but is much more intuitive to use.

Rule History

Witnesses and counterexamples provide the system administrator with very detailed information about the causes of an assertion failure. Nevertheless, it can be difficult to trace an error to the fault that causes it even when a good counterexample is available. We can obtain more precise information about the particular rules that create an error by constructing a ''history map'' during generation of the rule set MDD. The history map matches each packet to the set of rules that potentially accept or drop it.

Using the history map, we can associate packets in an assertion's fail set with a small number of filtering rules – usually much smaller than the number of rules in the entire policy. This permits the administrator to narrow his inspection of the policy to just a few critical areas. Since the set of rules to examine includes every rule that matches a packet in the assertion's fail set, it is possible that we may list some correct rules as well as the incorrect ones. In many cases, however, the history map will enable the system administrator to ignore most of rules that are not related to the problem.

Implementation

To evaluate an assertion, we represent each condition using a Multiway Decision Diagram (MDD), a data structure which is suitable for representing and manipulating large sets of packets. An MDD is a directed acyclic graph in which the nodes are organized into levels and all arcs from a node at a given level point to nodes at the level below.

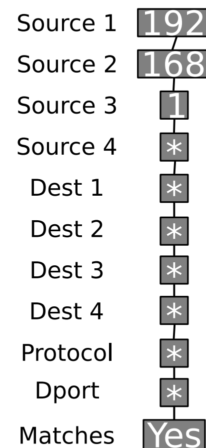

Figure 6: MDD representing **FROM 192.168.1.***.

Each level of the MDD corresponds to an attribute such as protocol, connection state, or destination port. For instance, level K, the top level of the graph, represents the first source octet of a packet. The bottom level of the MDD is a special terminal level which indicates whether or not a packet belongs to the match set. For space reasons, our figures show only some of the levels of each MDD. We also use an asterisk as a wildcard character to represent ''all arcs not explicitly listed in this node'' when many of the arcs leading from a node point to the same child.

To construct an MDD representation of a primitive such as FROM <address> or FOR <port>, we use nodes with all arcs pointing to the same child to mask out all but the relevant levels of the MDD. To represent FROM 192.168.1.*, we start at the top of the MDD and work down, inserting a node with just one arc labeled ''192'' at the top level (since the top level corresponds to the first octet of the source address). This arc connects to a node at the next level down which has a single arc labeled ''168'' which points to a node with an arc labeled ''1'' in the next level. The ''2'' arc connects to a node representing the fourth source octet

of the condition. All the remaining nodes in the graph are labeled with the wildcard character, since the other criteria are not relevant to the condition. This process is illustrated by Figure 6, which shows a simple condition MDD.

To test whether a particular packet is in the match set of a condition, we simply descend the MDD from its root to a terminal node using properties of the packet to guide the descent. If we reach the "matches" node, the packet is in the set. Otherwise, it is not. In Figure 6, a packet from 192.168.1.1 to 64.130.15.7 on TCP port 25 matches the condition, but a packet from 192.168.4.1 does not, since there is no arc for "4" leaving the node for source octet three.

The primitives ACCEPTED <chain> and DROPPED <chain> require special treatment. To generate MDDs for these conditions, we construct an MDD representation of each firewall chain. We then remove all the paths except those which point to the correct terminal node (either ACCEPTED or DROPPED) using a projection operation.

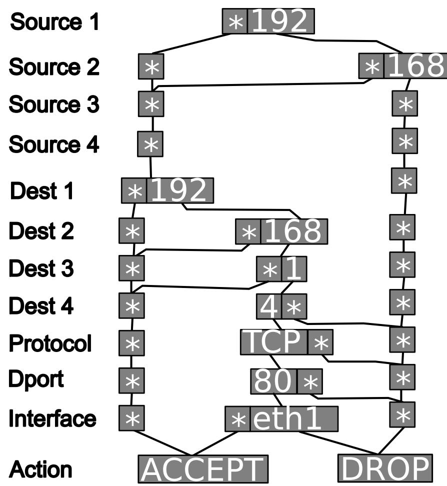

Figure 7: MDD for the FORWARD chain.

Figure 7 shows part of the MDD for the chain in Figure 1. To create an MDD representing ACCEPTED FORWARD, we copy those paths of the chain MDD which lead to the ACCEPTED node into the condition MDD and ignore paths leading to the DROPPED node. The resulting MDD is given in Figure 8.

Complex conditions containing the AND, OR, and NOT operators can be represented by using MDD intersection and union operators to combine the primitive MDDs. An example MDD for a more complex condition is given in Figure 9. Union and intersection can be performed very efficiently using MDDs. Using operation caches, we can obtain a guarantee that each pair of nodes in the graph is visited only once during these operations. Since the number of nodes in the graph is usually much smaller than the number of packets represented by each condition, we can complete these operations very rapidly. The complement operator is also very efficient. It requires a single descent of the MDD, which is linear with respect to the size of the graph.

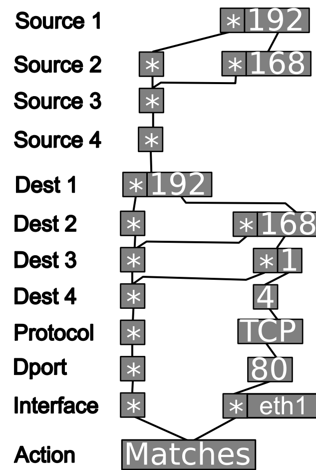

Figure 8: Packets accepted by the FORWARD chain.

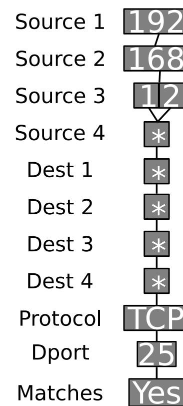

Figure 9: MDD representing FOR TCP 25 AND (FROM 192.168.1.* OR FROM 192.168.2.*).

To determine whether a containment assertion holds, we examine the set of packets that match condition *A*, but do not match condition *B*. If the assertion fails, this set will be non-empty, as illustrated by Figure 10.

The pseudocode in Figure 11 describes this process in detail. First, we construct MDDs representing the packets that match condition *A* and condition *B*, respectively, in steps 1 and 2. In step 3, we use an MDD complement operation to find the set of packets which do not match condition *B*, the right-hand side of the assertion. We intersect the MDD returned by the complement operation with the MDD representing condition *A*, the left-hand side of the assertion, in step 4. This creates an MDD representing the fail set of the assertion. In steps 5 through 8, we test whether the set is empty and return an appropriate value.

To test an equality assertion, we use the algorithm given in Figure 13, which is similar to the

algorithm for testing a containment assertion. Steps 1 through 7 create an MDD representing the fail set. If the fail set is non-empty, we have the situation illustrated by Figure 12 and the assertion fails. If it is empty, the assertion holds.

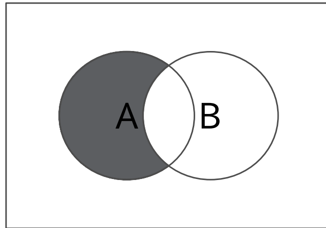

Figure 10: Fail set for the SUBSET OF operator.

```
bool testSubsetAssertion(cond A, cond B):
[1] mddA = condition_to_MDD(A);
[2] mddB = condition_to_MDD(B);
[3] notB = MDD_complement(mddB);
[4] result = MDD_intersect(mddA, notB);
[5] if notEmpty(result) then:
[6]    return ASSERTION_FAILED;
[7] else:
[8]    return ASSERTION_HELD;
```

Figure 11: Checking a containment assertion.

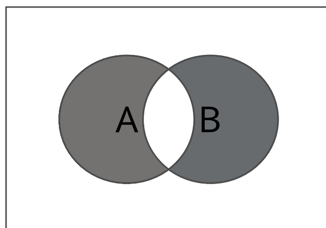

Figure 12: Fail set for the IS operator.

These techniques allow us to determine whether or not a firewall policy satisfies a set of assertions. These operations form a basis for performing more advanced calculations, such as example generation and history analysis.

Implementing Examples

To generate the counterexample for an assertion, we change the algorithms in Figure 11 and Figure 13 to return an arbitrary element from the fail set. This is done by replacing the last four lines of each algorithm with those in Figure 14.

The function *choose_element*(*X*) picks an arbitrary element from the set represented by MDD *X*. If the assertion does not fail, we choose an element from the fail set as the counterexample. If the assertion fails, we choose an element from the match set of the left-hand condition as a witness, since the elements of that set must match both conditions. To select an element, the *choose_element* function walks the MDD from the root node to the bottom of the graph, arbitrarily selecting arcs at each level (in practice, we select the first non-zero arc of each node) and storing each selected attribute in a "packet" structure which can be printed at the end of the traversal.

```
bool TestISAssertion(cond A, cond B):
[1]  mddA = condition_to_MDD(A);
[2]  notB = MDD_complement(mddB);
[3]  resultA = MDD_intersect(mddA, notB);
[4]  mddB = condition_to_MDD(B);
[5]  notA = MDD_complement(mddA);
[6]  resultB = MDD_intersect(notA, mddB);
[7]  result = MDD_union(resultA, resultB);
[8]  if notEmpty(result) then:
[9]    return ASSERTION_FAILED;
[10] else:
[11]  return ASSERTION_HELD;
```

Figure 13: Checking an equality assertion.

```
bool TestSubsetAssertion(cond A, cond B):
[5] if notEmpty(result) then:
[6]    return choose_element(result);
[7] else:
[8]    return choose_element(mddA);
```

Figure 14: Generating an example.

Implementing History

In order to build the history map, we construct a "history MDD" for each built-in chain of the firewall. The history MDD is similar to the MDD for a rule set or an assertion, but has two extra levels at the bottom of the graph. The top levels of the MDD correspond to the levels of a rule set MDD. The extra levels at the bottom represent a chain identifier and an index for each rule. We reserve index 0 for the default policy of a chain and index the remaining rules sequentially starting from 1. Construction of the history MDD is done concurrently with construction of the MDD representation of each firewall chain. As we insert rules into the rule set MDD for the chain, we copy those rules into the history MDD, adding nodes to identify the chain and rule to the bottom levels. If we encounter a rule which matches packets already matched by some other rule, we use MDD union to store a mapping to both rules.

Suppose we want to test the assertion

Chain Forward (Default DROP)					
#	Target	Source	Dest	Flags	
1	DROP	192.168.2.0/22	anywhere		
2	ACCEPT	anywhere	192.168.3.0/24		
3	ACCEPT	anywhere	anywhere	dpt:tcp 25	

Figure 15: Example rule set which protects a network 192.168.4.0/24.

```
ASSERT HISTORY NOT FROM 192.168.2.*
     AND TO 192.168.4.*
     AND FOR TCP 22
   SUBSET OF ACCEPTED FORWARD;
```

against the policy given in Figure 15. The assertion verifies that SSH traffic is allowed to a protected network 192.168.4.0/24 unless it originates on an untrusted network 192.168.1.0/24, from which all traffic is blocked by the firewall. The assertion will fail due to a typo in the subnet mask of the source address in rule 1. As a consequence of this fault, an SMTP packet from 192.168.3.1 to 192.168.4.2 that should be accepted will be dropped.

An example history MDD for the rule set is given in Figure 16. To save space, only some levels of the MDD are represented in the figure.

We can use the history MDD to find the rules that match this packet by descending the graph. To make this descent easier to follow, we have highlighted the path representing the example packet in Figure 16. Starting from the root node, we follow the arc labeled 192, since the source address of our example packet begins with 192. We next follow the arc labeled 168.

Because there is a typo in the subnet mask of rule 1, the node we are now examining has arcs for 0, 1, and 3 that point to the same child node as the arc for 2. We follow the arc labeled 3 to a node with all arcs pointing to the same child. We continue past the child node. The destination address of our example packet begins with 192, so we follow the arc labeled 192. We then follow the arc labeled 168 and the arc labeled with a wildcard, which represents all values other than 3.

This brings us to another node with all arcs pointing to the same child, which we continue past. We now take the arc labeled TCP, then the arc labeled 25. This takes us to a node at the chain level. The only arc leaving this node is labeled 1, so we know that the only rules that affect this packet are in the FORWARD chain. Following the arc takes us to a node representing rules 1, 3, and the default policy (represented by the label 0). Rule 2 is not listed since it matches only packets sent to subnet 192.168.3.0/24.

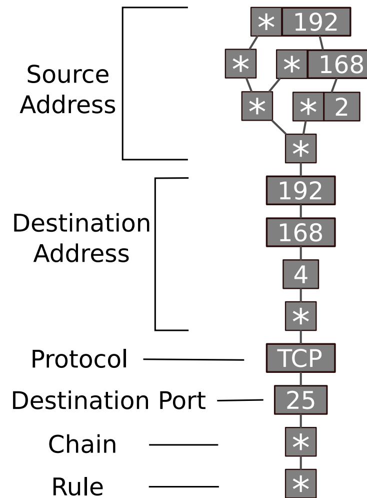

Figure 17: History MDD for an assertion.

This traversal gives the history map for a single packet. To find the rules that match those packets that violate the assertion, we intersect the history MDD for a chain with an MDD representing the fail set of the assertion. The fail set MDD can be computed as for counterexample generation, except that the result must be extended to include levels for the rule index and chain identifier. This can be done by padding the fail set MDD with wildcard nodes at the bottom two levels. This is illustrated by Figure 17, which gives an extended fail set MDD for the assertion.

The top four levels of the MDD correspond to the source addresses of packets that match the assertion. In this case, the assertion matches all packets except those from the 192.168.2.0/24 subnet. The next four levels represent the destination addresses of packets that match the assertion. In the example, only packets to subnet 192.168.4.0/24 match. The next levels represent protocol and destination port. The bottom levels represent the chain identifier and rule index of

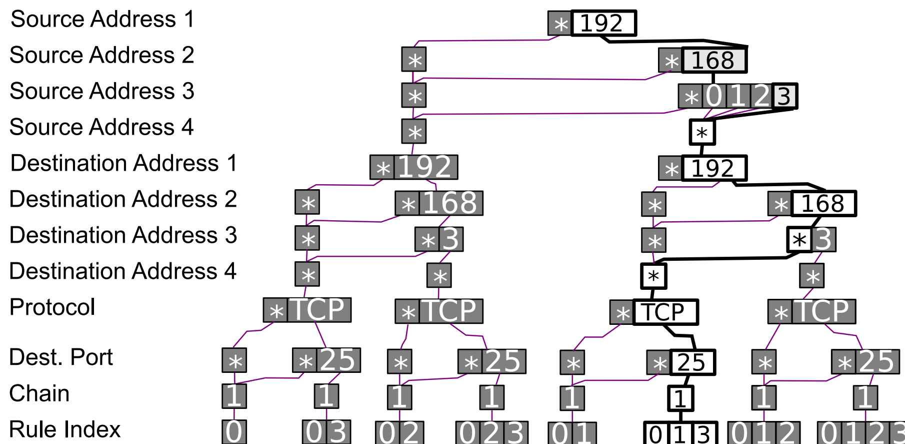

Figure 16: History MDD for a firewall chain.

the packet. Since we have not yet determined which rules are related to the assertion, these are represented as wildcard nodes.

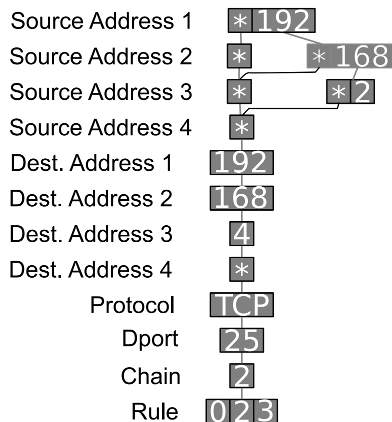

Figure 18: Result MDD after intersection.

Intersecting the extended fail set MDD with the rule set MDD gives us the history map MDD in Figure 16. ITVal converts this graph into the human readable output given in Figure 19. From this output, it is very easy to see that the fault lies in either rule 1 or rule 3. Rule 2 is ignored, since it only matches packets that do not cause the assertion to fail. In a longer policy, other extraneous rules would also be ignored.

```
ASSERT HISTORY NOT FROM 192.168.2.*
     AND TO 192.168.4.* AND FOR TCP 25
   SUBSET OF ACCEPTED FORWARD;
#Assertion failed.
Critical Rules:
  Firewall 0 Chain 1 Default Policy.
  Firewall 0 Chain 1 Rule 1:
 DROP   all -- * * 192.168.2.0/22
             0.0.0.0/0
  Firewall 0 Chain 1 Rule 3:
 ACCEPT tcp -- * * 0.0.0.0/0
             0.0.0.0/0 tcp dpt:25
```

Figure 19: Human readable history map.

In this case the fault lies in rule 1. An examination of that rule quickly reveals the typo. Our algorithm also lists rule 3 and the default policy of the FORWARD chain. Rule 3 is the rule that should have accepted the dropped packets and therefore may help the administrator understand the error. The default policy can be ignored, since it always matches every packet seen by the firewall. Using this enumeration of the matching rules, the system administrator can concentrate on the rules directly relevant to the problem.

History and Equivalence Classes

It is often much easier to use assertions than to perform a manual inspection of the policy. For one thing, the rules in a policy interact with each other in ways that can be confusing to the user. One rule in the policy might mask another rule or cause the rule to be applied only in certain, unusual, circumstances. Because each assertion is independent of the others, writing and understanding a list of assertions is often easier than manually correcting the rule set. More importantly, it is possible to construct a partial or high-level specification of the policy using assertions. This partial specification can ignore many of the details of the policy, which allows it to be simpler than the rule set to which it is applied.

Nevertheless, debugging the firewall using assertions has certain limitations. There is a tradeoff between the completeness of a specification and how easy the specification can be constructed. Deriving assertions that are both useful and effective can be a very challenging task.

Another limitation of the assertion approach is that certain kinds of faults cannot easily be identified using history maps for an assertion.

The policy in Figure 20 is supposed to protect a secure subnet 192. 168. 2. 0/24 from intrusions on an untrusted network 192. 168. 1. 0/24. An assertion checking that legitimate SSH traffic can reach the protected network is also given in the figure. A typo in rule 2 causes the assertion to fail. Unfortunately, the history map for the assertion will show only the default policy. None of the other rules in the policy match any packets in the fail set. In particular, rule 2, which contains the fault, does not match any packets from the 192.168. 2.0/24 subnet and, therefore, is not listed.

One way to address this problem is to create an assertion that checks whether packets from 192.186. 2.0/24 are accepted. The history map for such an assertion would immediately identify the typo in rule 2. The problem with this is that the system administrator has no way of knowing such an assertion is needed. It is not practical to create assertions for all of the possible typos in a policy, since doing so would require at least as much work as manual inspection of the policy.

A better way to address the problem is to extend the technique described in [9] to provide history information that can be used to discover faults in the policy. In that work, we described a technique for separating the computers on a network into related classes of hosts based on information taken directly from the firewall policy. Hosts in each class are equivalent in the sense that the firewall will accept or drop a packet from (or to) a host in the class only if it will accept or drop an otherwise identical packet from (or to) any other host in the class. For instance, if the firewall drops all SSH packets from host *A*, it will also drop all SSH packets from host *B* if they are in the same class. Each class of hosts on the network is represented as a five level "class MDD" which can be manipulated efficiently using MDD intersection and union operators.

Figure 21 lists the three classes shown in Figure 20. Class 2 corresponds to the untrusted subnet 192.

168.1.0/24. Class 3 is an anomalous class of hosts caused by the typo in rule 2. The existence of this class is an immediate clue to the system administrator that the firewall policy contains a serious error. Class 1 corresponds to all other hosts on the network.

```
QUERY HISTORY CLASSES;
There are 3 total host classes:
Class 1:
   <Everything not in the other classes>
Class 2:
     192.168.1.[0-255]
Class 3:
     192.186.2.[0-255]
```

Figure 21: Equivalence class decomposition of a policy.

Partitioning the hosts on a network into equivalence classes allows us to generate a "policy map" that shows functional groupings of the hosts on a network. The only input necessary to create a policy map is the firewall policy itself. When the policy contains a fault, it will often be manifested in the policy map as a missing class or by the presence of an unexpected class of hosts. The equivalence class technique can detect many kinds of faults that are difficult to identify using assertions. These faults include typos, overly broad rules, shadowed rules, outdated rules, and even missing rules. Unfortunately, while the policy map assists the system administrator in detecting these problems, it provides him with little information that can be used to identify the rules that must be changed to repair the issue.

We can enhance the policy map by annotating each class of hosts with a list of rules that match packets to and from a host in the class. To do this, we extend each class MDD with wildcard nodes. The resulting graph is similar in structure to the condition MDDs used to analyze assertions, but has wildcards at every level except the source address levels. This MDD matches the set of all packets whose source address matches a host in the class. We then repeat the procedure to produce an MDD with wildcards everywhere except the destination address levels. We can now intersect with the history MDDs for each chain to determine which rules match these packets. This intersection generates a result MDD which can be translated into a human-readable history map.

An MDD representing all packets with source address from class 3 is given in Figure 22. The top four levels of the MDD correspond to source addresses on subnet 192.186.2.0/24. The remaining levels contain wildcard nodes.

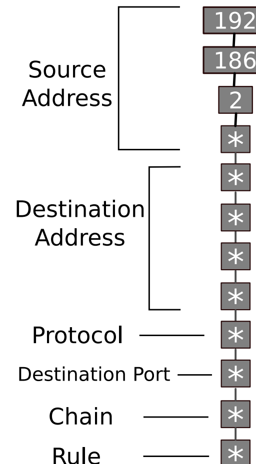

Figure 22: History MDD for class three.

```
Class 3:
Firewall 0 Chain 1 Default Policy.
Firewall 0 Chain 1 Rule 2:
ACCEPT all -- * * 0.0.0.0/0 192.186.2.0/24
         tcp dpt:22
```

Figure 23: History Map for class three.

A portion of the history map for the equivalence classes of the policy in Figure 20 is given in Figure 23. The existence of an anomalous class containing hosts from the 192. 186. 2. 0/24 subnet immediately alerts the system administrator to a serious error. A quick glance at the history map for class 3 reveals that only two rules are of interest: the default policy and rule 3. The system administrator now takes a careful look at rule 2 and discovers the fault, which enables her to repair the policy.

Existing Work

There are many good techniques for finding errors in a firewall policy. Tools such as nmap [3], Nessus [5], and ftester [1] allow the system administrator to actively test a firewall for specific well-known vulnerabilities. Unfortunately, these tools are

Chain Forward (Default DROP)				
#	Target	Source	Destination	Flags
1	DROP	192.168.1.0/24	anywhere	
2	ACCEPT	anywhere	192.186.2.0/24	tcp dpt:22

```
ASSERT HISTORY TO 192.168.2.* AND
      FOR TCP 22 AND
      NOT FROM 192.168.1.*
   SUBSET OF ACCEPTED FORWARD;
```

Figure 20: A fault that history mapping misses.

little help in identifying the faults which cause an error. For instance, nmap may indicate that a critical network port is unavailable for a variety of reasons: if the host is down, the firewall prohibits access to that port, the TCP wrappers on the server are incorrectly configured, or a routing error makes the host unreachable. Distinguishing between these potential causes is extremely difficult. Once the error has been narrowed down to the firewall, these tools do not provide any information about the policy itself that aid the user in determining why the error has occurred.

More rigorous testing can be done using passive testing tools, such as the Lumeta firewall analysis engine [10, 12], that perform an exhaustive off-line analysis of the policy. These tools simplify the task of determining whether the firewall policy contains errors by allowing the user to specify a set of logical queries that provide information about the policy. Some work has also been done on using expert systems to test the firewall policy [2]. The Lumeta engine provides support for History Mapping and limited example generation. However, the Lumeta engine is a proprietary closed-source product, which does not support iptables. These passive analysis tools also do not provide class-based analysis and therefore require the user to invest a significant amount of time designing appropriate queries or specifications against which the policy must be tested.

Conclusion

Using examples and history mapping, a system administrator can easily identify the two or three critical rules in a rule set that lead to a serious firewall error. Detecting these faults greatly reduces the amount of time an administrator must spend in careful examination of the policy and makes it much easier to manage and maintain a large, restrictive firewall policy. Using counterexamples and witnesses, the system administrator also gains valuable knowledge about the circumstances under which an error occurs. Using rule history with equivalence classes allows the system administrator to quickly and easily detect both errors and faults in the policy without constructing a large number of complicated assertions. The only required input is the policy itself. This greatly simplifies the process of maintaining, debugging, and repairing a restrictive firewall policy on a large network.

The techniques for generating a history map and relevant counterexamples have been implemented in our tool, ITVal, which can be downloaded from http://itval.sourceforge.net . The web site also provides several example specification files which can be downloaded and customized for use on a variety of networks.

Bibliography

[1] Barisani, Andrea, "Testing Firewalls and IDS With ftester," *Insight, Newsletter of the Internet Security Conference*, Vol. 5, 2001, http://www.tisc2001.com/newsletters/56.html .

[2] Eronen, Pasi and Jukka Zitting, "An Expert System for Analyzing Firewall Rules," *Proceedings of the 6th Nordic Workshop on Secure IT Systems*, 2001.

[3] Fyodor, "The Art of Port Scanning," *Phrack*, Vol. 7, Num. 51, September, 1997.

[4] Gouda, Mohamed G. and Alex X. Liu, "Firewall Design: Consistency, Completeness, and Compactness," *Proceedings of the International Conference on Distributed Computing Systems*, IEEE Computer Society, March, 2004.

[5] Lampe, John, *Nessus 3.0 Advanced User Guide*, October, 2005, http://www.nessus.org .

[6] Marmorstein, Robert, *ITVal Project Website*, 2005, http://itval.sourceforge.net .

[7] Marmorstein, Robert and Phil Kearns, "An Open Source Solution for Testing NAT'd and Nested iptables Firewalls," *19th Large Installation Systems Administration Conference (LISA '05)*, pp. 103-112, December, 2005.

[8] Marmorstein, Robert and Phil Kearns, "A Tool for Automated iptables Firewall Analysis," *FREENIX Track, 2005 USENIX Annual Technical Conference*, pp. 71-82, April, 2005.

[9] Marmorstein, Robert and Phil Kearns, "Firewall Analysis With Policy-based Host Classification," *20th Large Installation Systems Administration Conference (LISA '06)*, pp. 41-51, December, 2006.

[10] Mayer, Alain, Avishai Wool, and Elisha Ziskind, "Fang: A Firewall Analysis Engine," *Proceedings of the IEEE Symposium on Security and Privacy*, May, 2000.

[11] Stearns, Bill, http://www.stearns.org/ .

[12] Wool, Avishai, "Architecting the Lumeta Firewall Analyzer," *Proceedings of the 10th USENIX Security Symposium*, August, 2001.