Firewall Policy Diagram: Structures for Firewall Behavior Comprehension

Patrick G. Clark and Arvin Agah
(Corresponding author: Patrick G. Clark)

Department of Electrical Engineering and Computer Science
University of Kansas, Lawrence, KS 66045 USA
(Email: patrick.g.clark@gmail.com and agah@ku.edu)
(Received Apr. 9, 2014; revised and accepted Dec. 24, 2014)

Abstract

Communication security and regulatory compliance have made the firewall a vital element for networked computers. They provide the protections between parties that only wish to communicate over an explicit set of channels, expressed through protocols, traveling over a network. These explicit set of channels are described and implemented in a firewall using a set of rules. The firewall implements the will of the organization through an ordered list of these rules, collectively referred to as a policy. In small test environments and networks, firewall policies may be easy to comprehend and understand; however, in real world organizations these devices and policies must be capable of handling large amounts of traffic traversing hundreds or thousands of rules in a particular policy. Added to that complexity is the tendency of a policy to grow substantially more complex over time and the result is often unintended mistakes in comprehending what is allowed, possibly leading to security breaches. Therefore, it is imperative that an organization is able to unerringly and deterministically reason about network traffic, while being presented with hundreds or thousands of rules. This work seeks to address this problem using a data structure, the Firewall Policy Diagram, in an effort to advance the state of large network behavior comprehension.

Keywords: Firewall policy, firewall policy diagram (FPD), human comprehension, policy analysis

1 Introduction

Computer networking has arguably been one of the most important advancements in modern computing. Allowing disparate applications to trade information, conduct business, exchange financial transactions, and even the routine act of sending an email are some of the most common things we do with computers today. Even with the advancement of ever faster computer chips, the trend continues to connect devices at an astounding rate. In addition, there is also a thriving mobile device market, thus increasing the amount of traffic flowing between systems. An important aspect of this interconnected system is security. Without security, the convenience and speed of networked transactions would present more risk than the majority of applications could handle. In order to mitigate that risk and provide a much more secure communication channel, the firewall device was designed and deployed. It is one of the most widely used and important networking tools and exists in virtually every organization. In fact, over the past two decades the landscape of network security has come to rely heavily on that single type of device. The primary purpose of a firewall is to act as the first line of defense against malicious and unauthorized traffic, keeping the information that the organization does not want out, while allowing approved access to flow.

1.1 Firewall Basics

Firewalls allow two entities to connect their networks together through existing infrastructure and protocols, while securing the private networks behind them [16, 20]. The typical placement of a firewall is at the entry point into a network so that all traffic must pass through the firewall to enter the network. The traffic that passes through the firewall is typically based on existing packet-based protocols, and a packet can be thought of as a tuple with a set number of fields [16]. Examples of these fields are the source/destination IP address, port number, and the protocol field. A firewall will inspect each packet that travels through it and decide if it should allow that traffic to pass based on a sequence of rules. This sequence of rules is made up of individual rules that follow the general form:

$$\langle predicate \rangle \rightarrow \langle decision \rangle$$

The *predicate* defines a boolean expression over the fields in a packet tuple that are evaluated and the physical network interface from which the packet arrives. For example, source IP is 10.2.0.1 and destination IP address

is 192.168.1.1 on eth0 (a common label for a linux interface). Then the *decision* portion of a rule is what happens if the predicate matches to a true evaluation. A *decision* is typically accept or deny with the possibility of additional actions, such as an instruction to log the action [16]. However, for the purpose of this research we are only concerned with the accept or deny decision.

A firewall policy is made up of an ordered list of these rules such that as a packet is processed by the firewall, it attempts to match the packet to the predicate one rule at a time, from beginning of the rule list to the end. Matching the packet means that the firewall evaluates a packet based on the fields in the rule tuple, a packet matches the rule if it matches all the fields identified in the predicate of the rule [14]. The predicate does not necessarily need to contain a value for all possible fields and can sometimes contain the "any" variable in a field to indicate to the rule processing software that this is a "do not care" condition of the predicate and any value for that variable will match. It must completely match all the fields for the firewall to take the appropriate action. These rules are processed in order until the firewall finds a match, at that time it will take the appropriate action identified by the decision [14].

1.2 Motivation

The cost of a security breach has the potential to negatively impact business and cause large financial losses. This has been studied in the risk management area and falls under the avoidance topic [19]. The firewall is an important avoidance tool, however, over the past decade firewall rule-bases have grown in size at a remarkably fast pace. In a study finished in 2001, it was discovered that the typical organization will have 200 firewalls under the control of its network consisting of an average of about 150 rules per device [22]. In addition, these rule sets have been shown to grow to thousands of rules controlling routing between as many as 13 distinct networks [22]. More recent statistics gathered further support that the growth has only accelerated. In a study finished in 2009 the authors determined that the growth in complexity has out paced the growth in the organization's ability to synthesize and comprehend the changes [8]. The average number of rules has substantially increased from 150 in 2001 to 793, with a largest rule set found comprised of 17,000 rules [8]. The later study did not discuss the number of firewalls deployed, but in the unlikely case that firewall deployment growth stopped and the number of firewalls at an average organization stayed at 200 [22], then approximately 160,000 rules (200×793) would be under active management. In addition, the study also discovered that the average rule turnover (change) rate for an organization is 9.9% of the rules per month [8]. This means that an organization's firewall administration team has to accurately manage about 160,000 rules where 16,000 of those are changing on a monthly basis [8]. Therefore, the ability to accurately and confidently understand firewall policies and know what changes have occurred is more difficult than ever, and continues to increase in complexity.

1.3 Key Contributions

This work presents a novel set of data structures, together called a Firewall Policy Diagram (FPD). These data structures seek to solve the problem of large network behavior comprehension as it relates to firewall policies in several key areas:

- De-correlation of the firewall policy from the source rule set to gain a holistic view of behavior. This will remove any overlapping rules that will typically exist in a firewall policy [23] and the resulting data structure will model the actual ACCEPT and DENY space.
- Provide the ability to perform arbitrary mathematical set based operations like *and*, *or*, and *not*. These operations will assist in reasoning about firewall policy changes over time. They will also provide the foundation for many other firewall operations, such as understanding the functional differences between two policies. In addition, these operations will also provide the base for querying an arbitrary policy.
- Provide the foundation data structure for the implementation of a method to query the policy.
- Once a policy has been decomposed into an FPD, allow the reconstitution of that policy into a human comprehensible form, like an equivalent policy rule set.
- Finally, the experiments will show that these operations can be executed in seconds of computation time even on large policies (up to 10,000 rules).

The remainder of the paper is organized as follows: We begin with an overview of the FPD data structure and a description of how it is constructed, operated on, and translated into a set of de-correllated rules. We will then present the results of performance related experiments in terms of creation, SET operations, and reconstitution of firewall policies of various sizes. In the final two sections, related work and conclusions about FPDs will be covered.

2 Firewall Policy Diagram

A Firewall Policy Diagram is a set of data structures and algorithms used to model a firewall policy into an entity allowing efficient mathematical SET operations. The entity also has the ability to reconstitute the policy into a set of human comprehensible rules.

Table 1 demonstrates an access list that might be defined for a particular organization [14]. As shown, firewall policy traditionally consists of a list of rules. These rules are comprised of a subset of the fields in the Internet Protocol version 4 (IPv4) packet as defined by the

Open Systems Interconnection (OSI) model [14]. In this research effort only certain fields will be analyzed and modelled. The source address, destination address, protocol, and destination port will be used. The decision was made for two reasons, the source port is not often used in most firewall products and the flag is primarily used at layer three for setup and tear down of TCP connections [14]. However, if source port is necessary, it is a straightforward process to extend a FPD to include an additional 16 variables.

The internal storage mechanism of an FPD uses Reduced Ordered Binary Decision Diagrams (ROBDD or BDD) [5, 6, 18]. These data structures were introduced as an efficient way to capture hierarchical binary data and related works have described their use in firewall policy validation [11, 13, 23]. In addition to those that support the use of the BDD compressed data structure, there are research efforts that argue against its use in favor of other combination data structures [17]. However, in our work, the BDD provides the efficient storage and the necessary operations that allow our algorithms to reason about policy changes over time and differences between policies. Also, using network address translation (NAT) methods presented by [13], multi-firewall behavior over time can be modelled using BDDs, a missing research component in other policy comprehension work to date.

In a similar manner to the FIREMAN system [23], policies and rules are modelled as variable sets represented as BDDs. Using a BDD is an efficient way to represent a Boolean expression, like $(a \vee b) \wedge c$. Extending this concept to firewall policies, the variables in the expression become the bits of the associated IPv4 field. In this research, 32 bits representing the source address, 32 bits representing the destination address, 8 bits representing the protocol, and 16 bits representing the destination port. This means that for a particular ACCEPT space, there are 88 variables and 2^{88} potential combination of variable values.

2.1 Creation and Decomposition

When an FPD is initialized and created, the process begins by iterating over the policy rule set one rule at a time. Each rule is decomposed into its constituent parts relevant to our research: source, destination, protocol and destination port. At this point, each bit of each field is converted into an input vector of an appropriate size for the field. For example, the source field is 32 bits and therefore the vector is of size 32. Once the input vectors are constructed, the four input vectors are appended and added as a constraint in the underlying BDD [23].

2.2 Operations

Based on SET mathematics, if a data structure can accurately implement the *union*, *intersection*, and *complement* operation, other operations can be derived. For the purposes of this research and experimentation there are two primary operations that are being studied, namely, Difference and Symmetric Difference.

DIFFERENCE is used in the situation where there exists a base policy P and the desire is to understand what has changed in a later version of the policy, P'.

$$\begin{aligned} \triangle &= P' - P \\ &= P' \wedge \neg P \end{aligned}$$

SYMMETRIC DIFFERENCE is used in the situation where there exists two policies $P1$ and $P2$ and the desire is to know what is not shared between the two policies.

$$\begin{aligned} \triangle &= (P1 - P2) \vee (P2 - P1) \\ &= (\neg P2 \wedge P1) \vee (\neg P1 \wedge P2) \end{aligned}$$

Using these two operations with basic SET functions, we are able to reason about policy discrepancies and understand how one policy is related to another arbitrary policy. In addition, these operations can be chained together to produce a FPD' such that changes to a policy over time or as a set of access flows through multiple policies, the data structure can then be used to extract the resulting allowed (or denied) rules in a human comprehensible form.

2.3 Human Comprehension

Using the resulting policy represented by $\triangle$, the procedure to reconstitute that policy into a human consumable set of rules involves transforming the BDD. The first step in the algorithm is to only explore the space necessary for the operation, typically the ACCEPT space. Therefore, starting at the root node the graph is traversed to the accept node and avoids having to explore the potentially large opposite space. During this traversal, the BDD is transformed into human comprehensible rules. To accomplish this sort of rule extraction without having to completely decompress the data structure, two additional data structures and algorithms are used.

The first data structure is a Ternary Tree, which, as the name suggests, consists of three child nodes for every parent. The purpose behind this tree is to take the fully compressed BDD and decompress portions of it without the full cost of the worst case of 2^{88} leaf nodes being represented. There is a low, high, and combination child node, such that the low represents a 0, the high represents a 1, and the combination represents both 0 and 1. The idea behind using a Ternary Tree was inspired by [5, 6] and other systems allowing the processing of a "do not care" variable, such as ternary content-addressable memory (TCAM). Therefore, for a particular sequence of ordered variables in a BDD, solutions that occupy the same space (i.e., ACCEPT) have variables of three potential values: 0, 1, or both. The representation of the potential values is *0*, *1*, and *X*, respectively. The definition of the *0* and *1* are the same as a binary tree, and the variable represented in the node assumes that value. The new edge

Table 1: Example access control list for a firewall interface

rule	**action**	**src address**	**dest address**	**protocol**	**dest port**
1	allow	172.16.0.0/16	10.2.0.0/16	TCP	80
2	allow	10.2.0.0/16	172.16.0.0/16	TCP	> 1023
3	allow	172.16.0.0/16	10.2.0.0/16	UDP	53
4	allow	10.2.0.0/16	172.16.0.0/16	UDP	> 1023
5	deny	all	all	all	all

value of X represents that it is both 0 and 1 for that particular variable at that particular node in the tree. The result is compression in the size of the tree by removing the need for a left and right sub tree.

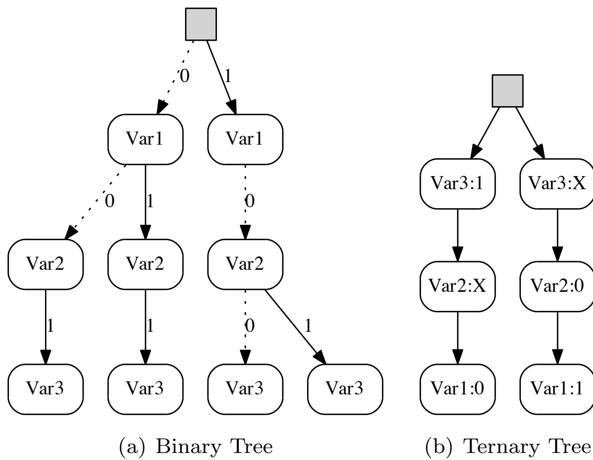

Figure 1: Identical binary numbers in a (a) binary tree and (b) Ternary tree

Figure 1 illustrates the concept and what the model would resemble when a binary tree is represented as a Ternary Tree. The tree still maintains the hierarchical nature of the data, but in a more compressed format. There are two important differences as a binary tree transforms to a Ternary tree. The first is the representation of the values of a particular variable (identified by bit location) is stored with the node. The second difference is the order of the variables in the Ternary Tree, as they are representative of how a tree formed from the algorithm shown in Figure 2 resulting in the least significant bit (LSB) variable at the root. These differences allow the tree to be pruned in reverse such that the process begins at the root and generates intervals as it traverses to a leaf. The binary numbers represented in these identical trees are 1, 3, 4, and 5.

The Ternary Tree provides an intermediary between the BDD and the pruned rules by allowing ranges in data to be represented and subsequently combined as the tree is pruned. This is information that cannot be easily ascertained from a canonical BDD representation. The algorithm to transform the ROBDD into a Ternary tree is shown in Figure 2.

An additional concern is that a true tree structure has the potential to have a higher storage cost because of replicated data nodes in child trees when the pattern could be shared where appropriate. This has been addressed in the Ternary Tree by allowing it to share nodes where the underlying pattern is shared. The resulting rule set deals with any collisions that may occur when pruning by copying intervals. For example, if the pattern starting at a certain variable is common and shared by parent variables, then the Ternary Tree will share those nodes by an edge pointing to those nodes. This accomplishes the goal of reducing space requirements without sacrificing the expressiveness of the data structure.

Input: ROBDD Bdd
Output: A fully formed Ternary Tree T
1: **procedure** TRANSLATE(Bdd)
▷ Bdd Variables labels start with 0
2: $T \leftarrow newTernaryTree$
3: **for all** $R \in RootNodes(Bdd)$ **do**
4: $N \leftarrow createTernaryRoot(T)$
5: WALKEDGE($R.low, N$)
6: WALKEDGE($R.high, N$)
7: **end for**
8: **end procedure**
9: **procedure** WALKEDGE(bN, tN)
10: **for** $bN.parent.var$ to $bN.var - 1$ **do**
11: $tN \leftarrow tN.middle \leftarrow newTernaryNode(X)$
12: **end for**
13: **if** $bN.var = Bdd.One$ **then return**
14: **end if**
15: **if** $bN.low \neq Bdd.Zero$ **then**
16: $tN.left \leftarrow newTernaryNode(0)$
17: WALKEDGE($bN.low, tN.left$)
18: **end if**
19: **if** $bN.high \neq Bdd.Zero$ **then**
20: $tN.right \leftarrow newTernaryNode(1)$
21: WALKEDGE($bN.high, tN.right$)
22: **end if**
23: **end procedure**

Figure 2: Algorithm for translation of a ROBDD to Ternary tree

```
for i = Min; i <= Max; i += 2^(Conversion Factor)
do
    Value ← i
end for
```

Figure 3: Algorithm for expressing all interval values

2.4 Pruning the Ternary Tree

The Ternary tree acts as an intermediate data structure where the primary purpose is to allow a second algorithm to collapse the tree into a set of human comprehensible data structures. The second algorithm is the pruning procedure that starts at the place-holder root of the Ternary tree and then traverses the tree to the leaves, pruning and generating intervals. The resulting data structure captures an interval and a count of bits in a conversion factor. The interval has a starting and ending number, with the conversion factor identifying how the interval sequence progresses. As an example, the source IP address Ternary Tree that represents an odd number of IP addresses is considered. In this example, the interval would be the starting and ending values of the range with a conversion factor of 1. All individual values in an interval can be expressed using the procedure shown in Figure 3.

As the tree is walked to the heuristically defined leaves of the tree, the interval is transformed by bit shifting and sometimes cloning of the interval to represent a transformation from a hierarchical data set into a rule. Additional details of the algorithm are identified in Figure 4. As the algorithm progresses, a number of these intervals are captured and get merged when appropriate. The internal representation of the ranges makes use of a red-black tree algorithm [9] to maintain a balanced structure while the intervals are assembled into a human readable form.

Notably, the entire rule is represented on the originating BDD; and therefore heuristics are used to help separate the data and make the rules more human comprehensible. This means that for a particular policy model, the data structure represents the concatenation of the source IP, destination IP, destination port, and protocol. The algorithms will separate the data structure into three separate Ternary Tree boundaries during the processing of the algorithm, namely, a source IP boundary, destination IP boundary, and service boundary. The process of extracting human comprehensible rules from an FPD runs at $O(V + E)$ where V and E are the number of vertices and edges in the Ternary trees, respectively.

The upper bound on the number of copies an interval must endure is 16. This is because a copy must occur when a variable transitions from 0 or 1 to X, and for a 32 variable number that can only occur a maximum of 16 times. This is also an upper bound on the number of intervals that could potentially exist at 2^{16}.

The final useful function achieved by using the interval data structure with conversion factor is a simple way to determine the number of addresses, ports or services in a particular rule. For an interval in a rule, it results in the

```
Input: Ternary Tree
Output: List of Rules that composed the SPACE
 1: procedure PRUNERULES(T)
 2:     for all L ∈ Children(T.root) do
 3:         Interval ← create(from = 0, to = 0)
 4:         Depth ← 0
 5:         PARSENODE(L, Interval, 0)
 6:     end for
 7: end procedure
 8: procedure PARSENODE(N, Interval, lnVal)
 9:     if N.value = X and Interval.factor empty and
    lnVal ≠ X then
10:         Interval.factor ← Depth
11:     end if
12:     if N.value = X then
13:         Interval.to ← to | 1 << depth
14:     end if
15:     if N.value = 1 then
16:         Interval.from ← from | 1 << depth
17:         Interval.to ← to | 1 << depth
18:     end if
19:     if N is boundary then
20:         Rules ← Interval
21:         Interval ← create(from = 0, to = 0)
22:         Depth ← 0
23:         PARSENODE(N.parent, Interval, 0)
24:     else if N not root then
25:         Depth = Depth + 1
26:         if lnVal ≠ X and N.value = X then
27:             Interval′ ← clone(Interval)
28:             PARSENODE(N.left, Interval′, N.value)
29:             PARSENODE(N.right, Interval′, N.value)
30:             PARSENODE(N.middle, Interval′, N.value)
31:         end if
32:         PARSENODE(N.parent, Interval, N.value)
33:     end if
34: end procedure
```

Figure 4: Algorithm for rule extraction

equation:

$$\left[\frac{(Interval\ Maximum - Interval\ Minimum)}{2^{\ Conversion\ Factor}}\right] + 1$$

The result of the pruning is a list of de-correlated rules with three intervals, the source IP range, destination IP range, and service range. An interesting result of the way that we have chosen to order the BDD variables is that the source IP to destination IP variable transition drives the number of distinct rules present in the final reconstituted rule set. This means that when that boundary in the variables is crossed while traversing the Ternary Tree, a new rule is generated and the remaining destination IP and service ranges are collected in that rule. Additional research could be done using the BDD variable reordering capability to find the most concise rule set representing a particular policy.

2.5 Heuristics Applied to Policies

Knowledge about the data set being modelled is important to the conversion and pruning algorithms. The heuristics provide the knowledge about where the hierarchical data sets begin and allow the summary of those data as the tree is pruned.

In this work the size of each of the fields drives the separation of the trees such that the 32nd, 64th, and 88th variables divide the hierarchy of a solution into the source IP address, destination IP address, and service (a combination of protocol and port). Notably, these algorithms can be applied to other hierarchical data sets in different domains. They may be especially useful when dealing with large solution spaces and multiple hierarchies being combined to provide the composition of a desired "space".

2.6 Generalized Example

This section will step through an example of how the algorithms of the FPD function on a smaller solution space in an effort to both show how the boundary heuristics may be applied to other domains, as well as a better description of how the algorithms function. A example is considered where we represent the solution space as 2^8, with a fictitious rule being made up of two fields of 4 binary variables, A and B, which subsequently form an 8 binary variable solution space. As the stored ROBDD is generated, A will represent the decimal values 2 through 12 and B will represent the decimal values 3 through 13. Figure 5 visualizes the canonical ROBDD data structure representing A and B with the LSB at the root, and referred to as variable zero. As the ROBDD is traversed from root to leaf, the tree is transformed into the graph as seen in Figure 6. The variable transition values *high* and *low* are shifted to the variable values in the individual nodes. The new root node with label *1* is created and the tree is rooted at that node. In addition, the solution space removes the need for the edges leading to the *zero* value as we do not care about those numbers when extracting the stored data. The change from one tree to another is what is described in the algorithm shown in Figure 2.

As indicated in earlier explanations of the Ternary tree, it acts as an intermediary data structure as human comprehensible intervals are extracted. Therefore, the next step of process is to traverse the Ternary Tree from root to leaf in an effort to generate a set of *rules* with the described intervals that make up those rules. We have already identified our single boundary heuristic in this generalized example as variable 4 of our 8 variables (using a zero index). Therefore as the algorithm shown in Figure 4, begins with that knowledge by traversing the root to leaf in an effort to create intervals.

The initial result is two groupings or *rules* being generated. This can be visually confirmed in the Ternary tree with the existence of two nodes for variable 4. Subsequently, in each rule there are two sets of intervals, those representing the variables 0 through 3 and those representing the variables 4 through 7. Rule 1, segment 1 (variables 0-3):

$$2\ to\ 10\ every\ 2^3$$
$$3\ to\ 11\ every\ 2^3$$
$$4\ to\ 12\ every\ 2^3$$
$$5, 6, 7, 8, 9$$

Rule 1, segment 2 (variables 4-7):

$$4\ to\ 12\ every\ 2^3$$
$$6, 8, 10$$

Thus, the intervals may be merged into segment 1 (variables 0 through 3) being represented by the decimal numbers 2 through 12. Subsequently the segment 2 interval may be merged into 4 *to* 12 *every* 2, so *even* numbers 4 through 12. Based on a similar interval merging procedure for Rule 2, segment 1 (variables 0-3) becomes:

$$2\ to\ 10\ every\ 2^3$$
$$3\ to\ 11\ every\ 2^3$$
$$4\ to\ 12\ every\ 2^3$$
$$5, 6, 7, 8, 9$$

Rule 2, segment 2 (variables 4-7):

$$3\ to\ 11\ every\ 2^3$$
$$5\ to\ 13\ every\ 2^3$$
$$7, 9$$

The intervals may then be merged into segment 1 being represented by the decimal numbers 2 through 12. Then segment 2 may be merged into 3 *to* 13 *every* 2, so *odd* numbers 3 through 13.

The final merge may then be reviewed between two rules such that when a segment of a rule matches another segment, as it does with segment 1, it may become one rule with the segment that is not matching (segment 2),

being combined and reviewed for merges. Therefore, after rule 1 merges with rule 2, and since the segment 2 intervals represents overlapping odd and even ranges, i will become a contiguous decimal interval from 3 to 13. This reconstitutes our original rule that created the BDD: A = 2-12, B = 3-13; which is fit for human comprehension.

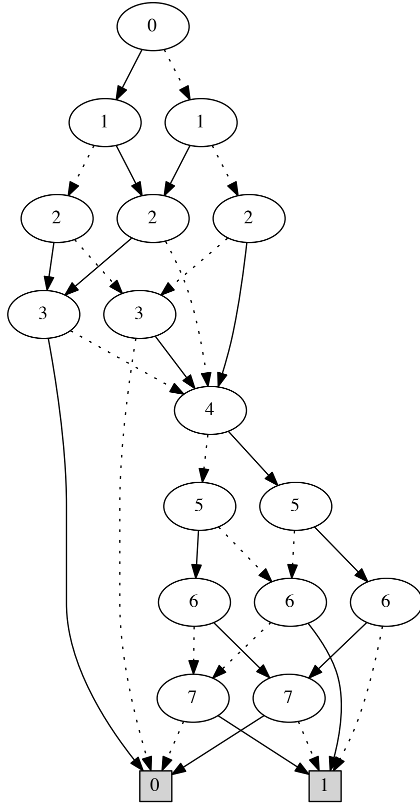

Figure 5: ROBDD generalized example

This algorithm is in contrast to a more naive, brute force approach to processing all known "solutions" to the BDD. In the example presented here, there are 64 8-bit numbers that will need to be parsed back into the known intervals that created the BDD from the start. While that number may seem small, the example and solution space are small by design. The important difference is between the number of solutions and the number of nodes in the Ternary tree. In addition, the brute force method removes the hierarchical relationships between the nodes, i.e. the heuristic value of node 4, further complicating reconstituting related intervals.

2.7 De-correlation

When a list of rules in a policy is decomposed into an FPD, a reconstituted policy that covers the same equivalent ACCEPT or DENY space will result in a rule set with none of the resulting rules overlapping in any area. The primary reason for this behavior is that as a policy is decomposed one rule at a time, all inconsistent or overlapping rules are removed and just the space is represented.

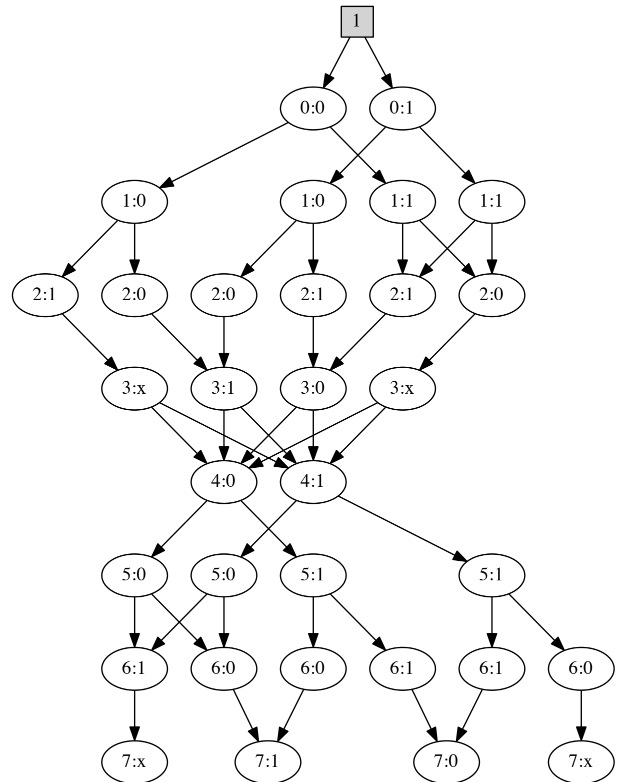

Figure 6: ROBDD as Ternary tree

The de-correlation property is useful in a number of scenarios:

- A policy no longer has a need to be processed as an ordered set of rules, since the FPD removes any overlapping rules. As a result, if the FPD is built by the rules in the policy from last to first, the resulting system can match an incoming packet to all rules simultaneously.
- A policy may be substantially smaller and take much less time to process once it has been de-correlated. This behavior is the effect of the procedure that converts rules into the FPD where it has also merged adjacent rules and removed any redundancies. In addition, the matching operation of a rule can be performed in constant time. This is because matching a rule involves walking the data structure from root to result, which is a constant 88 elements.

3 Experiments

The experiments designed for our work seek to review and address problems seen in the outside industry, i.e., large and difficult to understand firewall policies. We will first measure the time required to construct an FPD from rule sets of various sizes. We will then measure the time to extract a set of human comprehensible rules from an FPD. Finally, we will review the results of the DIFFERENCE and SYMMETRIC DIFFERENCE operations between two policies

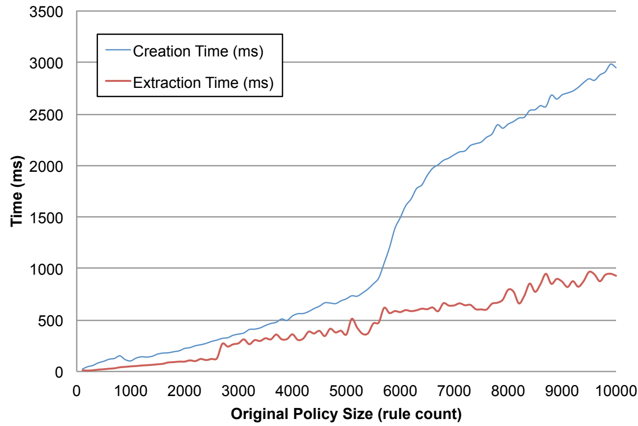

Figure 7: FPD generation and extraction of rules for policy sizes 100 to 10,000 rules

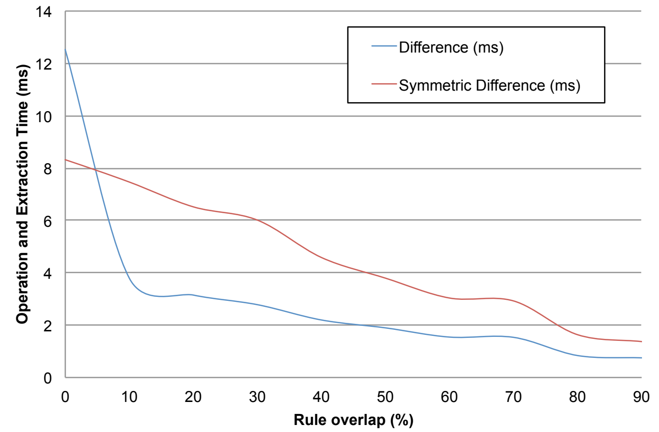

Figure 8: FPD difference and symmetric difference operations for a 200 rule policy

that share from 0% to 90% of the same space. The FPD data structure and algorithms are implemented in the Java™ 6 programming language and the tests are run on a Mac Book Pro laptop computer. For ROBDD software, the Buddy and JavaBDD libraries were used [15, 21].

For the data sets used to test creation and extraction of rules, policies of sizes 100 to 10,000 with 100 rule increments were created. When testing the DIFFERENCE and SYMMETRIC DIFFERENCE operations, we created two randomly generated 200 rule policies that share from 0% to 90% of the same space. Security reasons prevented us from being able to gain access to actual industry rule sets, as the majority of companies with firewall policies of those sizes are hesitant to allow outside parties access.

Figure 7 charts the performance of the FPD data structure for creation of a policy from a set of rules through to extraction of the policy into an equivalent set of rules. For creation of the FPD from a set of rules, the performance is consistently below one second for policies up to 5,500 rules. The creation times then begins to take more than a second until finally approximately 3 seconds to create a policy that originated from 10,000 rules. The extraction stays consistently less than a second to produce an equivalent set of rules and stays less than 500 ms for the majority of the data sets.

Figure 8 charts the performance of a DIFFERENCE operation and SYMMETRIC DIFFERENCE operation between two 200 rule policies. The experiment involved executing the appropriate operation; and then extracting the human comprehensible rules from the FPD. Notably, over 90% of the computation time is used producing human comprehensible rules. The actual DIFFERENCE and SYMMETRIC DIFFERENCE operations are averaging 5 milliseconds in all operations. SYMMETRIC DIFFERENCE operation exhibited slower processing performance due to the size of the resulting space represented by that operation. This means for that particular data set, the worst case representation of a space was very close to the maximum size of the potential *space*. DIFFERENCE operation yielded much smaller resulting data set sizes and reflected that fact in the computation times.

4 Related Work

Other work has been done modeling firewall policies, however, most of the models reflect their intended use and not all are capable of the sort of operations described in this work. Much of the research has focused on rule processing and validation of those rules where the goal is to identify redundant, shadowed, and inconsistent rules [1, 2, 3, 4, 7, 10, 16, 23]. In general the focus in those efforts is on algorithms for finding policy anomalies both from a single policy model to a multi-policy model. A portion of the related research introduced the use of BDDs for the models and became the foundation for some of the algorithms in our work [5, 6, 11, 18, 23].

In [12] the authors present a rule de-correlation algorithm with efforts to extract rules from existing firewall policies. The methods appear to be strictly for de-correlating rules with an exponential running time correlated to the size of the original policy and no overall policy comprehension.

One of the earliest works on policy comprehension built a query engine on top of a formal verification system named Voss [11]. Hazelhurst implemented a simple functional language that is capable of modeling a firewall policy in a formal verification system. While the underlying Voss hardware verification system used ordered binary decision diagrams to model a rule set, it is unclear the method in which the results were extracted from the canonical BDD form. The algorithm complexity was cited to be linear with respect to the number of rules for creation of the policy and constant time with respect to the number of variables in the BDD for any SET operations on the of the policy [11]. This is consistent with our work on processing FPDs. However, that analysis does not discuss the processing time related to extracting human comprehensible data from a binary decision diagram. This is

a large segment of processing and could involve millions of individual BDD solutions. Our analysis indicates that discussing the complexity as a function of the resulting BDD and the number of distinct solutions is a more accurate representation of the running time.

The research most similar to ours focuses on a data structure called a Firewall Decision Diagram [10, 16, 17]. The goal of the data structure is similar to ours, but the authors chose a different internal representation of a policy, specifically a combination of prefix and interval trees with additional data structures. It is capable of some DIFFERENCE operations, and is portrayed as a tool in which to achieve more accurate firewall design and change impact analysis. They specifically cite reasons for not using BDDs as the core of the data structure, although our work seeks to overcome the limitations referenced. In addition, it is not clear that they are capable of handling more complicated situations, such as network address translation, as a *space* is modelled through a real network.

5 Internet Protocol Version 6

Internet Protocol Version 6 (IPv6) is the latest iteration in the Internet protocol routing stack. It is an improvement to the current protocol IPv4, with the primary difference related to our research being the unique address space being expanded from 32 bits to 128 bits. This increase will allow many more devices to communicate and connect on the Internet. However, this also represents a challenge for firewalls and firewall administration teams. The growth of the addressable space means that these teams will need better and faster tools in which to comprehend how the firewalls under their control are configured and what sort of changes are happening to the security policies over time. This trend in the market space also supports the need for structures such as FPD for allowing the comprehension of these policies. While the experiments in this dissertation are on 32 bit addresses, expanding the FPD to use 128 bit addresses is a straightforward process and is planned for future experimentation. There is no reason that the algorithms would not function, although the size of the search space would become at least 2^{192} larger. This expansion is a result of the source and destination IP address space growing from 2^{32} to 2^{128}, therefore the number of bits needing to be represented grows $2^{96} \times 2^{96} = 2^{192}$.

6 Conclusions

In this paper we presented the Firewall Policy Diagram, an efficient and accurate data structure that serves as a basis for reasoning about firewall policy behavior and change. There are four primary contributions in this work:

- Data structures to efficiently model firewall policies that can be used to reason about them over time and modification.
- A data structure to act as the basis for the implementation of a means in which to query the policy.
- A set of algorithms in which to extract understandable rules from the FPD.
- Experimental evidence that these algorithms can perform the appropriate operations in seconds even on large firewall policy rule sets.

References

[1] E. S. Al-Shaer and H. H. Hamed. "Design and implementation of firewall policy advisor tools,". technical report, School of Computer Science, Telecommunications and Information Systems, DePaul University, Chicago, USA, August 2002.

[2] E. S. Al-Shaer and H. H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 4, pp. 2605–2616, March 2004.

[3] E. S. Al-Shaer and H. H. Hamed, "Modeling and management of firewall policies," *IEEE Transactions on Network and Service Management*, vol. 1, pp. 2–10, April 2004.

[4] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: a novel firewall management toolkit," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 17–31, 1999.

[5] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, August 1986.

[6] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, pp. 293–318, September 1992.

[7] C. Chao, "A flexible and feasible anomaly diagnosis system for internet firewall rules," in *Proceedings of the 13th Asia-Pacific Network Operations and Management Symposium*, pp. 1–8, September 2011.

[8] M. J. Chapple, J. D'Arcy, and A. Striegel, "An analysis of firewall rulebase (mis)management practices," *ISSA Journal*, pp. 12–18, February 2009.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2009.

[10] M. G. Gouda and A. X. Liu, "Firewall design: Consistency, completeness, and compactness," in *Proceedings of the 24th International Conference on Distributed Computing Systems*, pp. 320–327, 2004.

[11] S. Hazelhurst, A. Attar, and R. Sinnappan, "Algorithms for improving the dependability of firewall and filter rule lists," in *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pp. 576–585, 2000.

[12] E. Horowitz and L.C. Lamb, "A hierarchical model for firewall policy extraction," in *Proceedings of the 2009 International Conference on Advanced Information Networking and Applications*, pp. 691–698, may 2009.

[13] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer, "Modeling modern network attacks and countermeasures using attack graphs," in *Proceedings of the 2009 Computer Security Applications Conference*, pp. 117–126, December 2009.

[14] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*. Addison Wesley, 4th edition, 2007.

[15] J. Lind-Neilsen. "Buddy version 2.4. http://sourceforge.net/projects/buddy,", 2004.

[16] A. X. Liu and M. G. Gouda, "Complete redundancy detection in firewalls," in *Proceedings of the 19th Annual IFIP Conference on Data and Applications Security*, pp. 196–209, 2005.

[17] A. X. Liu and M. G. Gouda, "Diverse firewall design," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, pp. 1237–1251, September 2008.

[18] C. E. Shannon, "A symbolic analysis of relay and switching circuits," *Transactions of the American Institute of Electrical Engineers*, vol. 57, pp. 713–723, December 1938.

[19] N. Sklavos and P. Souras, "Economic models and approaches in information security for computer networks," *International Journal of Network Security*, vol. 2, no. 1, pp. 14–20, 2006.

[20] M. Uma and G. Padmavathi, "A survey on various cyber attacks and their classification," *International Journal of Network Security*, vol. 15, no. 5, pp. 390–396, 2013.

[21] J. Whaley. "Javabdd version 1.0b2. http://javabdd.sourceforge.net,", 2007.

[22] A. Wool, "A quantitative study of firewall configuration errors," *Computer*, vol. 37, no. 6, pp. 62–67, 2004.

[23] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra, "Fireman: A toolkit for firewall modeling and analysis," *IEEE Symposium on Security and Privacy*, pp. 199–213, 2006.

Patrick G. Clark is a Research Scientist for an information security analytics firm. He earned a Ph.D. degree in Computer Science with distinction from the University of Kansas in 2013 and his research interests include: rough-set theory, applied pattern recognition in medicine, IP network behavior models and big data algorithm analysis. He is a member of the ACM and of Upsilon Pi Epsilon, with over 30 publications in peer reviewed journals, conference proceedings and book chapters.

Arvin Agah is the Associate Dean for Research and Graduate Programs for the School of Engineering at the University of Kansas. He is the author of over 140 peer-reviewed articles and has earned multiple awards for research and graduate education. His research interests include algorithm analysis, artificial intelligence, applied AI in medicine and mobile robotics.