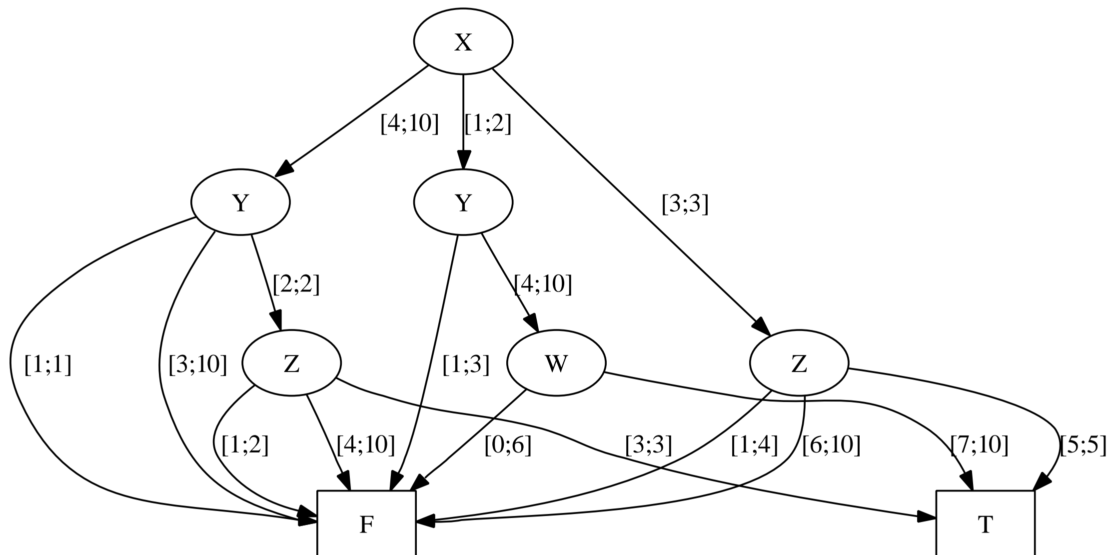
Optimizing the Interval Decision Diagram Implementation in the CF Firewall

Master Thesis



Aalborg University
Department of Computer Science

Aalborg University

Department of Computer Science

TITLE: Optimizing the Interval Decision Diagram Implementation in the CF Firewall

PROJECT PERIOD:
Febuary 1st, 2005 - November 30th, 2005

PROJECT GROUP:
d603a

GROUP MEMBERS:
Jesper Sloth Christensen, *huzzler@cs.aau.dk*

PROJECT SUPERVISOR:
Emmanuel Fleury, *fleury@cs.aau.dk*
Gerd Behrmann, *behrmann@cs.aau.dk*

NUMBER OF COPIES: 5

REPORT: NUMBER OF PAGES: 67

APPENDIX: NUMBER OF PAGES: 6

TOTAL: NUMBER OF PAGES: 92

SYNOPSIS:

The purpose of this project is to reduce the time it takes to build a decision diagram from a set of firewall rules.
We improve the implementation of the CF firewall and are able to reduce the worst-case runtime for the functions that perform logical operations on decision diagrams from exponential to polynomial. We also improve on other functions and for large rulesets we cut the time it takes to build the decision diagram from hours to seconds.
We also look at the order of variables in the decision diagram and examine what effect changing the order would have. We present an algorithm for finding a better order but discover that the order this gives does not reduce the size of the decision diagram. However, we also see that even if changing the order does not affect the size of the decision diagram, it can still influence the time it takes to build the decision diagram.

Summary

To secure a personal computer or a local network from attacks from someone on the internet a firewall is often used. A firewall sits between the computer or the network to be protected and the rest of the internet. The job of the firewall is to scan incoming and outgoing network traffic and filter out any unwanted network traffic. The firewall administrator indicates what traffic to filter out by using a list of rules. If a network packet matches a rule, the policy associated with the rule is carried out on the packet. The firewall for linux works by matching a packet against the first rule in the list and moving on to the next until a match is found. This can be very costly if the matching rule is at the bottom of the list.

Compact Filter (CF) represents the rules as a decision diagram instead. This makes it possible to match a packet against the ruleset much faster. However, building the decision diagram is very time consuming for large rulesets. We therefore look at ways to improve CF so that the decision diagrams can be build faster.

The implementation of the functions that work on decision diagrams in CF is very ineffective. An iterator is used in many places when it is necessary to traverse the entire decision diagram. However, the implementation of this is very ineffective.Furthemore we implement an operator cache which is used to avoid performing the same logical operation on the same input more than once. We change the implementation of the functions used to perform logical operations on decision diagrams and are able to reduce the worst-case runtime of the functions from exponential to polynomial.

We go on to investigate other ways of reducing the build time for the decision diagrams that are directly related to the code. We look at the order of the variables in the decision and also at something called complement nodes which allows two complement IDDs (IDDs where the TRUE and FALSE terminal is interchanged) to be represented by the same IDD. In a decision diagram each node represents a check on a variable and we show that the order in which these checks are performed is of great importance to the size of the decision diagram. We propose a method to find an order of the variables that will give smaller decision diagrams than the order currently used in CF. Test results show that although the method does not improve on the size of the graph, changing the order can still reduce the

time it takes to build the decision diagram.

Finally we give some suggestions for further work. It is important to implement global IDDs in the future to fully utilize the improvements that we have made. When support of more header fields is added to CF it is necessary to have another look at the order of variables.

Preface

This report is the documentation of a master thesis project (DAT6) at Aalborg University, Dept. of Computer Science. The goal of the project is to reduce the time it takes to build a filter for the CF firewall. This is done through code optimization and changes in data structure holding the filter. All source code can be found at the following internet address:

http://www.cs.aau.dk/ huzzler/dat6.html

Jesper Sloth Christensen

Table of Contents

1 Introduction **1**
1.1 Problem description 1
1.2 Project goal 3

2 Decision Diagrams **5**
2.1 Representing firewall rules as predicate logic 5
2.2 Reduced IDDs 7
2.3 Boolean operators on IDDs 8
2.4 Representing filter rules with IDDs 12
2.5 MTIDD 12
2.6 Representing an IDD in CF 13

3 Performance Analysis Tools **15**
3.1 Gcov 15
3.2 Gprof 16

4 Implementation **19**
4.1 A new IDD constructor 19
4.2 The Hashvalue 20
4.3 The operator cache 21
4.4 The Iterator Problem 22
4.5 Cloning an IDD 25
4.6 The Logical Operators 27
4.6.1 NOT 27
4.6.2 AND and OR 32
4.7 iddDeepFree 35

4.8 Benchmarks 37
4.8.1 Building the filters 37
4.8.2 Test setup 38
4.9 Summary 40

5 Performance Analysis **43**
5.1 Testing the old implementation 43
5.2 Testing the new implementation 45

6 Other solutions **49**
6.1 Ordering of variables 49
6.2 Complement nodes 51
6.3 Summary 53

7 Finding a better order for CF **55**
7.1 The current order 55
7.2 The algorithm 56
7.3 Test results 58
7.4 Summary 64

8 Conclusion **65**

9 Further work **67**

A The original source code **69**
A.1 iddDeepClone() 69
A.2 iddNot() 70
A.3 iddAnd() 73
A.4 iddDeepFree() 75

Chapter 1

Introduction

This chapter will give a basic introduction to the background of the project and present the motivation for the project. Furthermore, the overall goals of the project are presented.

1.1 Problem description

Today, making your personal or company network secure has become more important than ever as the internet has become more and more widespread. This has made access to information, entertainment, etc. much easier but it has also made it easier for a malicious person to attack whomever he or she wishes. One way to guard against such attacks is using a firewall.

A firewall sits between one network (such as the internet or a local area network) and a local area network or a single computer. It may be located on a separate machine or one of the machines in the network. The primary purpose of the firewall is to detect and discard unwanted network traffic. The traffic may either be inbound or outbound.

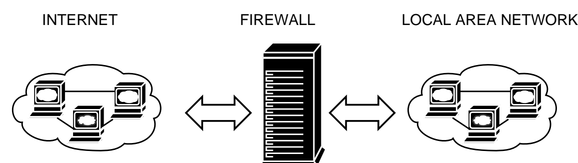


Figure 1.1: The posibble position of a firewall

The firewall works by comparing each packet in the network traffic with a set of rules defined by the administrator of the firewall. In linux, netfilter is used to build the firewall from a set of rules. The rules are written as a list [NF]. A rule for netfilter will look something like this:

```
iptables -A INPUT -s 192.168.0.0/24 -i eth0 -d 192.168.0.10 -p tcp --dport 80 -j ACCEPT
```

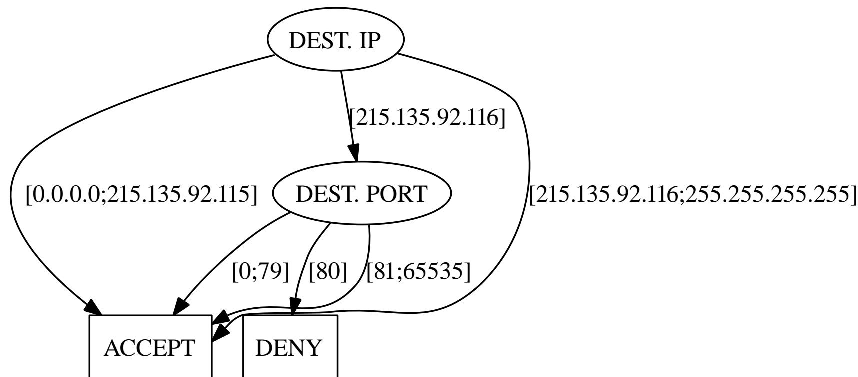


Figure 1.2: A simple illustration of the decision diagrams used in CF

If the packet does not match the first rule, it is compared with the next and so on until a matching rule is found or until the entire list has been traversed in which case a default policy is applied. In a worst case scenario the matching rule is located at the end of the list meaning that the firewall spends a lot of time traversing the list. The rules in netfilter may, however, be split into several lists. Thus a match or partial match in one list may cause netfilter to begin traversing another list. Thereby the proces of matching a network packet with a rule may be split into several smaller checks.

To make mathching network traffic against the ruleset faster, a new type of firewall has been proposed, Compact filter (CF), which represents the rules in an entirely different way [CF]. Insted of representing the rules in a sequential matter as Netfilter does, CF represents the rules using a type of directed acyclic graph (DAG) called Multi Terminal Interval Decision Diagrams (MTIDD). This makes it possible to match a network packet against the entire ruleset by the use of a number of checks on individual packet header fields. Each node in the graph represents a check on one of the fields in a filter rule, such as IP address or port number. Depending on the outcome of this check one of the outgoing edges is chosen until a terminal node is reached which says what to do with the packet. A simple illustration of this is shown in figure 1.2. A more elaborate explanation can be found in chapter 2

CF has ben tested against Netfilter and HiPAC (another proposed replacement for Netfilter - see http://www.hipac.org) and has proved to very efficient compared with these two firewalls once it is up and running. However, building the MTIDD from the rules can be very time consuming for large rulesets. Small filters with up to around 100 rules takes less than one second to build, but with 1,000 rules it takes 37 seconds, 5,000 rules takes around 5 minutes and 50,000 rules takes almost 1.5 hour according to [CF] (in chapter 4 we present a number of filters which take even longer

to build which further underlines the problem with long build times for filters).

When the number of rules goes up so does the compile time and this can be a problem if frequent or urgent updates of the ruleset are required. This could be if e.g. a firewall administrator has a list of banned servers that computers on the local network are not allowed to visit and which needs to be updated or if an external host needs to be denied access to the local network like in the case of a malicious attack. Although this should not require rebuilding the MTIDD from scratch, CF does not currently support adding a new rule to an existing filter.

1.2 Project goal

The overall goal of this project is to try to reduce the amount of time spent on compiling the filter.

We will look at the implementation of LIBIDD, the library in CF used to build and manipulate decision diagrams. Specifically we will examine the worst-case runtime of the primary functions to see if this can be improved and thereby reducing the time it takes to build a filter.

We will also look at other papers on the subject of decision diagrams in order to find methods that can be applied here and which may also contribute to reducing the compile time of the filters. In particular we will look at the importance of the ordering of variables. This is a subject which has claimed a lot of attention in the litterature of decision diagrams. We will look at ways to discover a better order for a decision diagram and investigate how these findings affect CF and the compile time of filters.

In chapter 2 we take a closer look at interval decision diagrams and describe what they are and how they are used in CF. In chapter 3 we present the performance analysis tools used. In chapter 4 we identify several problem areas in LIBIDD and present a new implementation for several functions. We also present test results for the current and new implementation which shows a huge reduction in the time it takes to build a decision diagram. In chapter 5 we run some performance analysis on the current and new implementation of LIBIDD. In chapter 6 we present other ways of improving LIBIDD by looking at the order of variables in the decision diagram and the use of complement nodes. In chapter 7 we present an algorithm for finding a better order for CF. We find that the algorithm does not give an order which produces a smaller graph. However, we also see that although an order does not produce a smaller graph it may still help in reducing the time it takes to build a decision diagram. In chapter 8 we conclude on the report and in chapter 9 we present ideas for further work.

Chapter 2

Decision Diagrams

This chapter introduces Interval Decision Diagrams (IDD) and Multi Terminal Interval Decision Diagrams (MTIDD) and how they can be used in packet filtering. First we show how a rule in a packet filter can be represented using predicate logic. Then we show how to represent the predicate logic formula using IDDs. We also how basic logical operators are applied to IDDs. Finally we present the MTIDD which is used to combine several IDDs into one decision diagram.

2.1 Representing firewall rules as predicate logic

The first step in showing that IDDs can be used to represent rules in a packet filter, is to show how these rules may be expressed using predicate logic. A more elaborate explanation of this is given in [CF04], here we give a brief summary.

First we give a formal definition of a rule in a firewall filte. A rule (r) is made up by a set of header fields (η) and a policy (π). Thus we have:

- H, the superset of all possible sets of header fields, $\eta \in H$
- $\Pi = permit, deny$, the set of policies, $\pi \in \Pi$
- A rule $r = (\eta, \pi)$

A rule that drops packets with http traffic to a specific host would be written as:

(2.1) $r = ((IPDADDR \in \text{212.95.114.156}) \wedge (TCPDEST = 80), deny)$

where $IPDADDR$ is the destination ip address and $TCPDEST$ is the destination port number. We can then define a filter φ as a set of rules over $H \times \Pi$.

(2.2) $\varphi = ((\eta_1, \pi_{k_1}), (\eta_2, \pi_{k_2}), ..., (\eta_n, \pi_{k_n}))$

$(\eta_i)_{i \leq n}$ is said to be a partition of H iff

- $\bigcup_{i \leq \eta_i} = H$
- $\eta_i \bigcap \eta_j = \emptyset, \forall i, j \leq n$ with $i \neq j$

In order to avoid confusion in ambigous filters, we define an ordered filter where the order prioritizes among overlapping rules. We have that

- ψ is an ordered filter iff $\psi = (\eta_i, \pi_{k_i})_{i \leq n}$ with $\eta_i \subset H, \pi_{k_i} \in \Pi$ for all $i \leq n$, where n is the number of rules, and we have an implicit order $\succ$ such that $(\eta_i, \pi_i) \succ (\eta_j, \pi_j) \iff i > j$

An IDD is a DAG (directed acyclic graph) where each node represents an evaluation of a bounded integer variable. Each outgoing edge corresponds to an interval within the domain of the variable. The result of evaluating the variable determines which outgoing edge to follow. An edge either points to another node or a terminal which is either true or false. An IDD is defined as follows:

- x is an integer variable defined on the domain $D_x \subseteq N$
- t is an IDD node iff
  - $t \in True, False$ or
  - $t = ((x \in I_0 \wedge t_0) \vee (x \in I_1 \wedge t_1) \vee ... \vee (x \in I_k \wedge t_k))$, where $I_i, i \leq k$ is a partition of D_x and $t_i, i \leq k$ is a set of IDD nodes.

A node is called a root node if there is no predecessor and a set of nodes is called an IDD if there is only one root node and there are no cycles. $var(t)$ is the function which returns the value of the variable of the node:

(2.3) $$var(t) = \begin{cases} x, & \text{if } t = x \rightarrow (I_0, t_0), (I_1, t_1), \ldots, (I_k, t_k) \\ t, & \text{if } t \in \{True, False\} \end{cases}$$

We call $I = ((t_i)_{i \leq n}), \succ)$ an ordered IDD iff $\succ$ is an order on the variables such that for all $t \in (t_i)_{i \leq n}$ we have $x \succ var(t'_i)$ for all $i \leq k$ and where $t = x \rightarrow (I_0, t'_0), (I_1, t'_1), ..., (I_k, t'_k)$. Lets consider a logical formula with four variables, each with the domain [1;10], where x is first in the order, then comes y,z and w.

(2.4)
$$(1 \leq x \leq 2 \wedge 4 \leq y \leq 10 \wedge 7 \leq w \leq 10) \vee (x = 3 \wedge z = 5) \vee (4 \leq x \leq 10 \wedge y = 2 \wedge z = 3)$$

This gives us the following nodes:

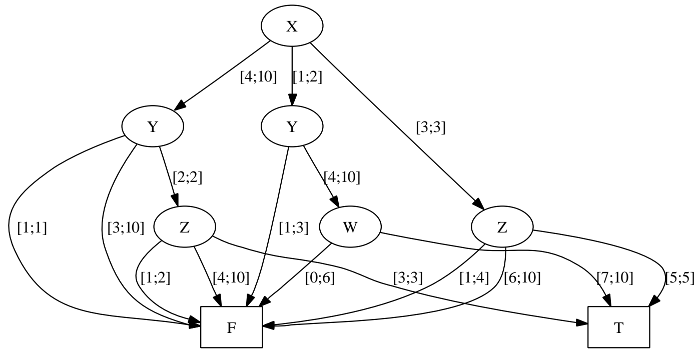


Figure 2.1: An example of an IDD

- $t_0 = x \rightarrow ([1;2], t_{00})([3;3], t_{000})([4;10], t_{01})$
- $t_{00} = y \rightarrow ([1;3], F)([4;10], t_{0000})$
- $t_{01} = y \rightarrow ([1;1], F)([2;2], t_{001})([3;10], F)$
- $t_{000} = z \rightarrow ([1;4], F)([5;5], T)([6;10], F)$
- $t_{001} = z \rightarrow ([1;2], F)([3;3], T)([4;10], F)$
- $t_{0000} = w \rightarrow ([0;6], F)([7;10], T)$

A graphical representation of the graph can be seen in figure 2.1.

2.2 Reduced IDDs

A very important aspect when implementing a library for decision diagrams is to ensure that the graphs we build are reduced at all times. We say that an IDD is reduced if it adheres to the following conditions:

- There may not be two identical nodes in the IDD
- Two adjacent edges may not point to the same node
- A node must have at least two children

Figure 2.2 shows a graph which is not reduced and figure 2.3 shows the same graph in reduced form.

A graph should be kept reduced at all times. This ensures that we perform no unnecessary operations and that the graph takes up as little space as possible.

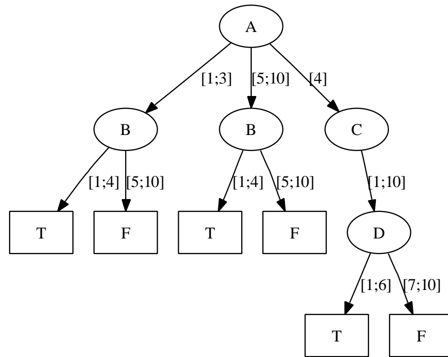


Figure 2.2: An unreduced graph

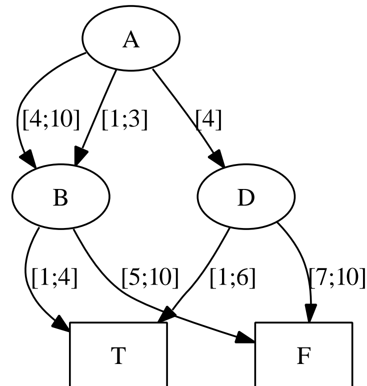


Figure 2.3: A reduced graph

2.3 Boolean operators on IDDs

We can use the same basic boolean operators on IDDs as we can with predicate logic formulas such as AND, OR, NOT etc. If we want to combine two IDDs, A and B, we use the AND operator which works in the following way on IDDs:

```
If we have performed AND on A and B before
  return result from that operation
If A and B are terminals
  If both are TRUE
    return TRUE
  Else
    return FALSE
If A is a terminal and B is a node
  If A is FALSE
    return FALSE
  Else
    For each child of B
      If the child is equal to the adjacent child
        merge them
      If only one child remains
        return that child
      Else
        return B
If A is node and B is a terminal
  If B is false
    return FALSE
  Else
    For each child of A
```

```
      If the child is equal to the adjacent child
        merge them
      If only one child remains
        return that child
      Else
        return A
If A and B are nodes
  If A comes before B in the order
    For each child of A
      If the child is equal to the adjacent child
        merge them
      If only one child remains
        return that child
      Else
        AND B with each of A's children
  If B comes before A in the order
    For each child of B
      If the child is equal to the adjacent child
        merge them
      If only one child remains
        return that child
      Else
        AND A with each of B's children
  If A and B are equal in the order
    Build a new node and combine the edges of A and B to form
      the edges of the new node
    For each child of the new node
      If the child is equal to the adjacent child
        merge them
      If only one child remains
        return that child
      Else
        For each edge AND the IDD pointed to by that interval by A
          and B respectively
```

The check in the beginning of AND ensures that we do not perform any operation on the same input more than once. This also means that the worst case runtime of AND is $O(n * m)$ where n is the number of nodes in A and m is the number of nodes in B.

Let's elaborate on the last part a bit. If A and B share the same position in the order, meaning that they represent the same variable, we take the intervals assigned to the edges of A and B and use these to construct a new set of intervals. Let's say that A has two edges $[0; 3]$ which points to node C and $[4; 10]$ which points to node D, and B has two edges $[0; 5]$ which

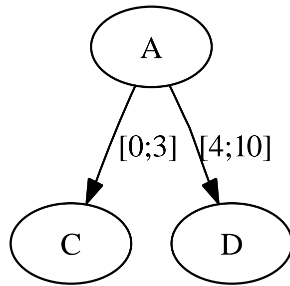


Figure 2.4: IDD A

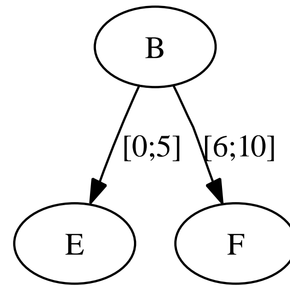


Figure 2.5: IDD B

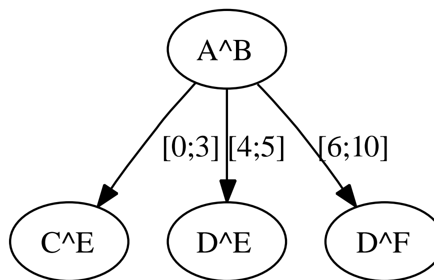


Figure 2.6: The IDD of A AND B

points to node E and $[6;10]$ which points to node F, see figures 2.3 and 2.3. Performing AND on A and B would result in a new node with the edges $[0;3]$ which points to C AND E, $[4;5]$ which points to D AND E and $[6;10]$ which points to D AND F, see figure 2.3.

OR looks very similar. In the case that A and B share the same position in the order, the approach is the same as with AND except that the IDDs are OR'ed instead. The worst-case runtime is also the same.

```
If A and B are terminals
  If just one is TRUE then return TRUE
    else return FALSE
If A is a terminal and B is a node
  If A is TRUE
    return TRUE
  Else
    For each child of B
      If the child is equal to the adjacent child
        merge them
      If only one child remains
        return that child
      Else
        return B
```

```
If A is node and B is a terminal
  If B is TRUE
    return TRUE
  Else
    For each child of A
      If the child is equal to the adjacent child
        merge them
      If only one child remains
        return that child
      Else
        return A
If A and B are nodes
  If A comes before B in the order
    For each child of A
      If the child is equal to the adjacent child
        merge them
      If only one child remains
        return that child
      Else
        OR B with each of A's children
  If B comes before A in the order
    For each child of B
      If the child is equal to the adjacent child
        merge them
      If only one child remains
        return that child
      Else
        OR A with each of B's children
  If A and B are equal in the order
    Build a new node and combine the edges of A and B to form
      the edges of the new node
    For each child of the new node
      If the child is equal to the adjacent child
        merge them
      If only one child remains
        return that child
      Else
        For each edge OR the IDD pointed to by that interval by A
          and B respectively
```

NOT is simply a matter of finding the terminals in the IDD and negating their value. The worst case of is therefore $O(n)$ where n is the number of nodes in `A`.

```
If A is a node
```

```
  Perform NOT on each of A's children
If A is a terminal
  If A is TRUE then return FALSE
    Else return TRUE
```

2.4 Representing filter rules with IDDs

By using the header fields of IP packets as the variables, the IDD can be used to represent the rules in a packet filter. For example, the IDD representing the two following rules can be seen in figure 2.4. The rules are written in the syntax used in CF. `INPUT` means that the rule should be added to the filter attached to the INPUT hook, `deny` is the policy that should be applied to a packet that matches the rule, `host x.x.x.x` denotes an IP address, the first being the source and the second being the destination.

```
INPUT deny host 51.252.160.40 host 73.165.243.220
INPUT deny host 99.225.183.39 host 211.69.141.119
```

In order to keep it simple we only check source and destination IPs and we use the standard 4x8 bit notation instead of a single 32 bit value which is how it is represented in CF. It is also possible to split the variable check on the IP adress into 4 checks, one for each of the 8 bit segments. While this may cause the IDD to grow in size because it would take four checks to check an IP address, the resulting four nodes will have far smaller ranges and may also result in more sharing.

The IDD is built using the following algorithm:

```
For each rule in ruleset
  Build an IDD for each variable in the rule
  AND the IDDs together
  OR the rule with the filter IDD
```

With IDDs we have two different terminals (permit and deny), which are represented using the *true* and *false* terminals in the IDD. If we wanted to perform more actions on a packet other than simply permitting or denying it (e.g. logging), this is not possible with an IDD. For this purpose, we introduce the MTIDD (Multi Terminal Interval Decision Diagram).

2.5 MTIDD

The MTIDD and the definition of an MTIDD node is in many ways similar to the IDD. The main difference is that instead of two boolean terminals, we have a set of terminals. Thus the definition of an MTIDD node looks as follows:

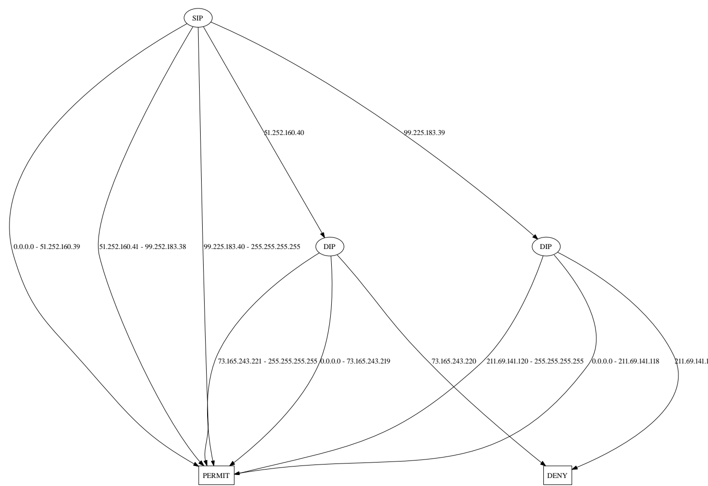


Figure 2.7: An example of an IDD representing a packet filter

- x is an integer variable defined on the domain $D_x \subseteq N$
- t is an MTIDD node iff
  - $t \in T$, where T is a set of terminals, or
  - $t = x \rightarrow (I_0, t_0), (I_1, t_1), ..., (I_k, t_k)$, where $I_i, i \leq k$ is a partition of D_x and $t_i, i \leq k$ is a set of MTIDD nodes.

It is important here to notice that although an MTIDD can be seen as an extension of an IDD it is not a boolean formula, it is actually a combination of several IDDs. That is, evaluating an MTIDD can be seen as evaluating several IDDs at the same time and performing the actions specified in the terminals. So you may have to identical IDDs where one leads to a TRUE terminal which should be interpreted as PERMIT and another leads to a TRUE terminal which should be interpreted as LOG. In an MTIDD these are combined into a PERMIT+LOG terminal

2.6 Representing an IDD in CF

We briefly present the way an IDD is currently represented in CF. This will also help in understanding the following chapters better. The representation of an MTIDD is almost the same except for the terminal which contains a list of actions instead of two boolean values.

The representation of a terminal in an IDD is very straightforward. It is simply an *enumeration constant* which can take on the values of FALSE and TRUE.

```
typedef enum { FALSE, TRUE} IddTerminal_t;
```

The node struct contains three elements. `name` is the name of the variable represented by the node. It is an integer which gives the position of the variable in the order associated with the decision diagram. `partition` contains the partitions of the node and `partition_size` is the number of partitions the node has. A partition entry has a lower bound, an upper bound and a pointer to an idd. When the variable of a node is evaluated the direction to go in is determined by which partion the result matches.

```
typedef struct {
  int name;
  IddPartitionEntry_t* partition;
  int partition_size;
} IddNode_t;
```

The idd struct has a `type` which is an enumerator that tells whether this is a node or a terminal, a `value` which holds the node/terminal struct, a `hashvalue` and an `id` used for assigning unique IDs

```
typedef struct Idd {
  enum { NODE, TERMINAL } type;
  union { IddNode_t node; IddTerminal_t terminal; } value;
  uint32_t hashvalue;
  uint32_t id;
} Idd_t;
```

Before moving on we briefly present the names we use when talking about the firewall. *CF* is the overall name for the entire firewall project, *cfconf* is the tool used for building decision diagrams from rulesets and *LIBIDD* is the part of the code responsible for building IDDs.

Now that we have presented the data structure on which CF relies we go on to introducing the performance analysis tools that we will use later on.

Chapter 3

Performance Analysis Tools

This chapter presents the performance analysis tools used in this procjet, *gcov* and *gprof*. These tools are used for coverage measurement and profiling respectfully. The first deals with which parts of the code are executed and which are not, the second describes how much time a particular part of the code uses.

3.1 Gcov

Coverage measurement can be be divided into three major groups. The first is *function coverage* where we measure which functions were executed, the second is *statement coverage* where we look at which individual lines of code were executed and finally there is *branch coverage* where we look at which condition in branch statement is satisfied. We are primarily interested in statement coverage to see which lines in a highly time consuming function are executed most.

Gcov does not require much preparation to use. All you have to do is to compile the code with two special GCC options: `-fprofile-arcs -ftest-coverage`. A *.da* file will then be generated for each source file when the program is run. It is now time to run gcov with the source files as argument. This will produce a file with the name *sourcefile.c.gcov*. The contents of this file will look something like table 3.1

Each line of code is preceeded by one of the following symbols:

- `-` means that the line contains no source code.
- `42` means that the line was executed 42 times.
- `####` means that the line was never executed.

Although we cannot directly see which lines the CPU spends the most time on, we can use the results to see the relationship between the number

```
           ...
     195:  526:  iddIteratorPostorderInit(&iddite, idd);
     195:  527:  while ((cur_idd = iddIteratorPostorderNext(&iddite)) != NULL) {
    2386:  528:    match = 0;
   33783:  529:    for (i = 0; i < parray_used; i++) {
   31779:  530:      if (parray[i] == cur_idd) {
     382:  531: match = 1;
     382:  532: break;
       -:  533:      }
       -:  534:    }
    1809:  535:    if (!match) {
     195:  536:      parray[parray_used] = cur_idd;
    2004:  537:      parray_used++;
       -:  538:    }
       -:  539:
    2386:  540:    if (parray_used >= IDD_PARRAY_SIZE) {
   #####:  541:      printf("iddDeepFree out of memory");
   #####:  542:      exit(0);
       -:  543:    }
       -:  544:  }
           ...
```

Table 3.1: Example of the output generated by gcov

of times each line is executed. Thus when we know which part of the codes requires a lot of CPU time, we can use coverage measurement to figure out which lines or statements we need to do something about in order to reduce run time. In order to find out how much time is spent on individual parts of the code we use the profiling tool *gprof*, which is described in 3.2.

3.2 Gprof

Gprof requires us to set the option `-pg` when compiling the code. This enables us to see how much time is spent on each function in the program. When the program is run a file called `gmon.out` is generated. After running the program you run the gprof command with the executable file and the gmon.out file as input.

This is assuming that we are in the directory where the executable is located and that we ran the program when this directory was the current working directory. This is important as the gmon.out file is always written to the current working directory, so we have to be aware of directory changes when we for instance initiate multiple tests using a shell script. The output from running gprof can be devided into two parts, *the flat profile* and *the call graph*. An example of the flat profile is shown in table 3.2.

The first column (% time)is the percentage of the total time spent on this particular function, the second column (cumulative seconds) is the number of seconds spent on the function listed so far, the third column (self seconds) is the number of seconds spent on the respective function, the fourth column (calls) is how many times the function was called, the fifth column (self s/call) is the average time spent on the function per call and the sixth

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 42.20      5.22     5.22    18988     0.00     0.00  iddDeepFree
 14.92      7.07     1.85   317114     0.00     0.00  iddHashGetHashValue
  6.39      7.86     0.79  2204551     0.00     0.00  iddEqual
  5.09      8.49     0.63  5417644     0.00     0.00  iddIteratorPreorderNext
  4.12      9.00     0.51      997     0.00     0.00  iddHashInit
  3.88      9.48     0.48  1458647     0.00     0.00  iddComparePartitionBounds
  2.99      9.85     0.37      994     0.00     0.00  iddHashFree
  2.26     10.13     0.28  4454926     0.00     0.00  nodeStackInit
  1.78     10.35     0.22   142716     0.00     0.00  iddHashFind
  1.70     10.56     0.21  4496659     0.00     0.00  nodeStackPush
  ...
  ...
```

Table 3.2: An example of the gprof flat profile

column (total s/call) is the average time spent on the function *and* its descendants per call. The entries are sorted based on first *self seconds*, then *calls* and finally the function name.

At the beginning of the flat profile there is a line that reads "Each sample counts as 0.01 seconds". This tells us that a 100 Hz sampling rate was used. What this means is that gprof takes a sample 100 times each second and if a function is running at the time when the sample is taken then 0.01 second is added to the runtime of that function. This means that there will be some uncertainty associated with the results we get, especially for tests run for a very short time. Here the results will be sensitive to when the samples are taken.

An example of the call graph is shown in table 3.3.

The entries in the call graph are seperated by a line of dashes. For each entry there is one line with an index number. This is the function being profiled. The lines above are the functions that call this function and the lines below are the functions that this function calls, *time* is the percentage of the total time spent in this function and its children. The following columns have different meaning depending on whether it is a parent of the function, the profiled function itself or a child of the function.

For a parent *self* is the time propagated from the function into this parent, *children* is time that was propagated from the function's children into this parent, *called* is the number of times this parent called the function and the total number of times the function was called (recursive calls not included).

For the profiled function itself *self* is the number of seconds spent in this function, *children* is the total amount of time propagated into this function by its children, *called* is the number of times the function was called by another function (recursive calls are indicated by a '+' and the number of recursive calls).

```
granularity: each sample hit covers 2 byte(s) for 0.08% of 12.37 seconds

index % time    self  children    called     name
                0.01   12.13       1/1           main [2]
[1]     98.1    0.01   12.13       1         yyparse [1]
                0.01    5.96    1000/1006        cacl_filter_stack_popping [3]
                0.00    4.44     917/917         cacl_fun_tcp_permit_rule [8]
                1.10    0.06    4000/18988       iddDeepFree [5]
                0.00    0.40      83/83          cacl_fun_udp_permit_rule [25]
                0.05    0.00   11001/11001       yylex [43]
                0.00    0.04    2000/2000        cacl_fun_ip_idd [46]
                0.00    0.04    2000/2000        cacl_fun_port_eq_idd [47]
                0.00    0.04       3/3           cacl_filter_add_permitall_rule [49]
                0.01    0.00    4000/8000        iddOrderFind [62]
                0.00    0.00    2000/2000        cacl_fun_port_value_check [78]
                0.00    0.00    1000/1006        cacl_filter_addrule [79]
-----------------------------------------------
                                                 <spontaneous>
[2]     98.1    0.00   12.14                 main [2]
                0.01   12.13       1/1           yyparse [1]
                0.00    0.00       1/1           cfconfParseOpts [103]
                0.00    0.00       1/1           setinput [105]
                0.00    0.00       1/1           parsefilter [104]
-----------------------------------------------
...
```

Table 3.3: An example of the gprof call graph

For a child of the function *self* is the time that was propagated directly from the child into the function, *children* is the time that was propagated from the child's children to the function and *called* is the number of times the profiled function called the child and the total number of times the child was called (recursive calls not included).

We are now ready to look at ways of improving the implementation of LIBIDD. This is done in the next chapter.

Chapter 4

Implementation

This chapter deals with the ways the current implementation can be improved to reduce the runtime of *cfconf*. In chapter 2 we presented the IDD, the operators NOT, AND and OR and the importance of keeping an IDD reduced. Unfortunately the current implementation of LIBIDD does not ensure that an IDD is kept reduced and therefore the worst-case runtime of the logical operators in LIBIDD is higher than it needs to be. In this chapter we present the problems with the current implementation and come up with solutions for fixing these problems

First we present a new IDD constructor which ensures that we do not build a new IDD if we know of an identical IDD which we have previously built. We then present a new way of computing the hashvalue of an IDD. Currently the IDD is turned into a string representation which is then hashed. Then we present an operator cache which stores information about previous uses of NOT, AND and OR. Using this will help avoid performing the same operation more than once. We then go on to presenting the iterator used in LIBIDD to find all the nodes and terminals in an IDD and why this poses a problem. After that we present the logicial operators and how we can change the worst case runtime from exponential to polynomial by using the operator cache. We also present `iddDeepFree` which is used to free all the memory associated with an IDD. The worst case runtime for `iddDeepFree` is also reduced. We then present benchmark tests of the original and the new implementation to show which improvements we have achieved. Finally we give a summary of the chapter.

4.1 A new IDD constructor

LIBIDD relies on a function for allocating space for a new IDD (`iddAlloc()` and another function for initiating the IDD with specified values (`iddNodeInit` and `iddTerminalInit`). When using these functions to build a new IDD there is no check to see if the IDD already exists.

The new IDD constructor, named `iddMakeNode()` ensures that no new IDD is built which is equivalent to an already existing IDD. The key to the new IDD constructor is a hashtable which is given to it as an argument from the calling function. This hashtable must contain all IDDs that the calling function has constructed previously to the call to `iddMakeNode()`.

It is up to the calling function to allocate and initiate a hashtable which is then passed to `iddMakeNode()` each time it is called together with the name, partition and the partition size we wish returned. First the function goes through the partition entries to see if the size of the partition can be reduced. This is possible if two adjacent partition entries reference the same child. This is not expensive to do if we know that all IDDs referenced in the partition are made with `iddMakeNode()`. If they are we know that for two IDDs to be identical they have to be the same IDD and therefore the check is simply a matter of comparing pointer references. If the partition is reduced to a single entry we return the IDD in that entry (ie. the only child left in the node) as the result.

`iddMakeNode()` then looks through the hashtable to see if a match can be found. If so, the matching IDD is returned, otherwise a new IDD is allocated and initiated.

A function for making terminals, called `iddMakeTerminal` also exists. This is a little simpler in that it only takes a hastable and a terminal value (TRUE or FALSE) as arguments. It then looks through the hashtable to see if it can find a terminal with the given terminal value. If it does this terminal is returned, if not a new terminal is allocated and initiated and then returned.

By using `iddMakeNode()` we ensure that there are no identical nodes in the IDD because it looks for a match in the hashtable prior to creating a new node, that two adjacent edges do not point to the same child and that a node has at least two children because if we encounter a node that does not the child is returned as the result thus discarding the node in question.

4.2 The Hashvalue

In LIBIDD the hashvalue for an IDD is calculated in two steps. First a string representation of the IDD is produced and then this is turned into a hashvalue using Horner's algorithm, see table 4.1.

This is a very cumbersome way of getting the hashvalue because we first have to compute a string representation of the IDD. Therefore we implement a new function for getting a hashvalue for an IDD.

If the IDD is a terminal the hash function simply returns a constant. There is no point in calculating a hash value for terminals as it will always be the same. If the IDD is a node we compute a new hashvalue based on the name of the node, the partitions and the hashvalues of the children. The

```
uint32_t iddHashGetHashValue(char *idd_string_rep) {
  uint32_t h = 0;
  int pos; /* position in idd_string_rep */
  for (pos = 0; idd_string_rep[pos] != '\0'; pos++) {
    h = (64*h + idd_string_rep[pos]) % MAXHASH;
  }
  return h;
}
```

Table 4.1: The current function for obtaining a hashvalue in LIBIDD

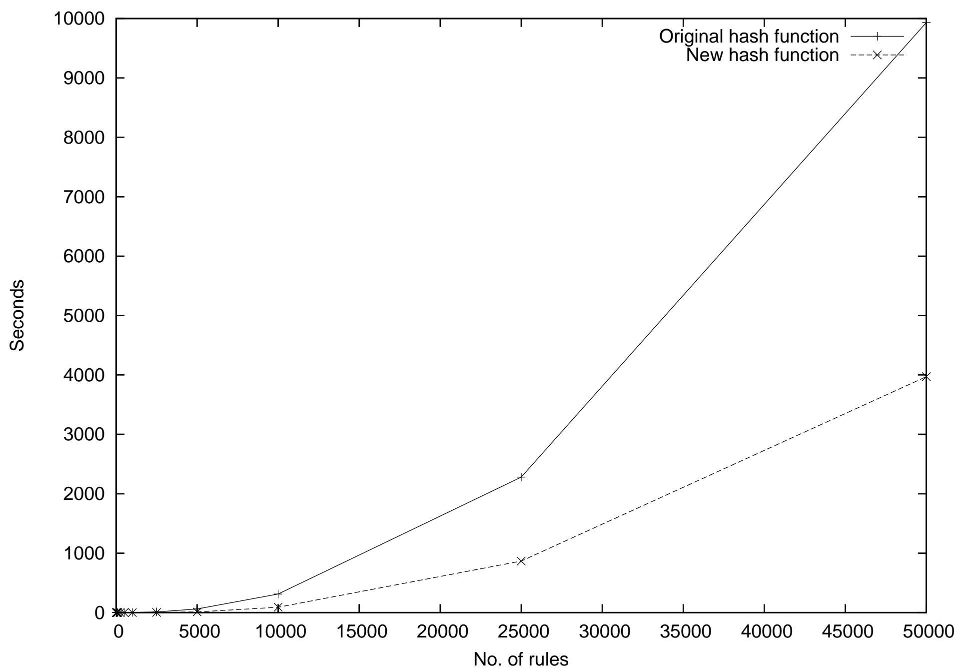


Figure 4.1: Comparing runtimes using the old and new hash function

hashfunction we use is taken from *http://burtleburtle.net/bob/c/lookup2.c*.

In figure 4.1 we compare the runtime for cfconf using the old and new hash function and see that there is a significant reduction in runtime.

4.3 The operator cache

The operator cache is used to store information about operations that we have already performed. Whenever we are about to perform an operation on one or two IDDs we look in the operator cache (using the hashvalue of the operands) to see if we have already performed that operation before. If so we simply retrieve the result instead of performing the operation again. If no match is found in the cache we perform the operation and add the

result together with the operands to the operator cache afterwards.

Using the operator cache ensures that we do not perform the same operation on the same input more than once. When we know this we also know that the worst-case runtime for the operation is $O(n)$ for unary operations and $O(n * m)$ for binary operations.

In table 4.2 we see the struct for a cache entry. Each entry contains pointers to the two operands and the result. It also contains an integer called `tag`. This is used to distinguish operations performed by different functions or even instances of the same functions from each other. Each time one of the functions that uses the cache is called, it increments a global counter (`opcounter`) and when we look for a match in the cache, the `tag` must match the current value of the global counter. There is a potential problem here if the counter wraps around but this would imply an extremely large number of operations which we do not reach. The operator cache is a global array of cache entries and it is allocated and initiated when the program starts.

```
typedef struct
{
   uint32_t tag;
   Idd_t *arg1;
   Idd_t *arg2;
   Idd_t *result;
} cacheentry_t;
```

Table 4.2: The operator cache

There are three functions available that can be applied to the operator cache. An init function (table 4.3) which allocates and initializes the cache with a given size, a destroy function (table 4.4) which frees the array and all the entries in it (but not the IDDs referenced in it) and finally a lookup function (table 4.5) which looks for an entry which contains the two specied IDDs. If no match is found an empty entry is returned so that the calling function may fill out the values from the operation. If more than one operation should be placed at the same position in the cache, the previous entry is overwritten. The reason for overwriting existing entries is that we expect collisions to be very rare, so it is better to risk overwriting an entry than having to maintain a collision list.

4.4 The Iterator Problem

Several functions in the current implementation such as `iddNot()` and `iddDeepFree` use an iterator to go through an IDD and discover all the nodes and terminals. However, the way the iterator works is not efficient and depending on the level of sharing we risk running into the same node or terminal several

```
int iddCacheInit(int s)
{
   iddCacheDestroy();
   size = s;
   entries = calloc(sizeof(cacheentry_t), s);
   return entries ? 1 : 0;
}
```

Table 4.3: The operator cache init function

```
void iddCacheDestroy()
{
   free(entries);
   size = 0;
   entries = NULL;
}
```

Table 4.4: The operator cache destroy function

times. In fact, by using the iterator we risk performing the same operation an exponential number of times instead of a polynomial number of times.

LIBIDD has two iterators, a preorder and a postorder iterator, but they work in pretty much the same way. The iterator works on a struct which, among other things contains a pointer to the root of the IDD we wish to index and a nodestack which is, as the name suggests, a stack containing IDD nodes. Each time the iterator is called a new node is returned. The iterator works by taking a node (the first time this is the root node) and adding that to the nodestack. Then the node pointed to by the first partition entry of the node at the top of the stack is added to the stack. This continues until we reach a terminal which is returned as the result. The next time we look at the second partion entry in the node at the top of the stack and the node which it points to. This continues until there are no more partion entries left at which point the node is popped from the stack and returned as the result.

An example of the proces is shown in figures 4.2 - 4.8. Nodes in the stack are marked as grey and the IDD returned as the results is marked as black (remember that the term IDD covers both node and terminal as well as a set of IDDs, see chapter 2)

- First A is pushed onto the stack and we continue to the IDD pointed to by the first partition of A, which is B. B is then pushed onto the stack and again we continue to the IDD pointed to by the first partition. This is the terminal T, which is returned as the result. (Fig. 4.2)

- On the next run we look at the second partition of B which points to the terminal F. This is returned as the result. (Fig. 4.3)

```
cacheentry_t *iddCacheLookup(unsigned int hash)
{
  return &entries[hash % size];
}
```

Table 4.5: The operator cache lookup function

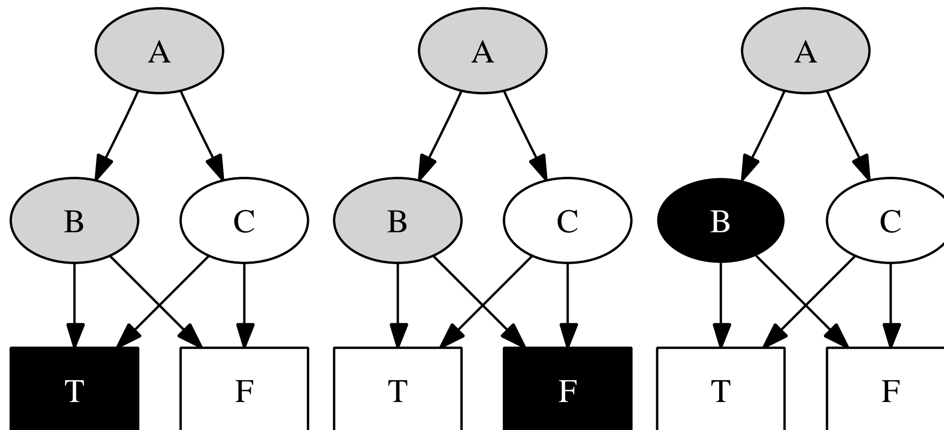


Figure 4.2: Step 1 **Figure 4.3:** Step 2 **Figure 4.4:** Step 3

- The next time there are no more partions in B, which is then returned as the result and popped from the stack. (Fig. 4.4)
- We are now back at A and we look at the second partition which points to C. C's first partition points to the terminal T, which is returned as the result - here we run into an IDD which we have seen before. (Fig. 4.5)
- We now look at the second partition of C, which points to F. Again we return an IDD that we have seen before. (Fig. 4.6)
- C has no more partitions and it is returned as the result and popped from the stack. (Fig. 4.7)
- A has no more partions and is returned as the result. The next time the iterator is called it will return NULL because there are no more nodes in the nodestack. (Fig. 4.8)

When using an iterator to go through an IDD we will run into some of the nodes more than once if the IDD has any kind of sharing. The more sharing the more we will run into nodes we have already seen before and because reduced IDDs only have one TRUE terminal and one FALSE terminal we will run into those many times. This means that the calling function must check if the node or terminal it gets from the iterator is one that it has

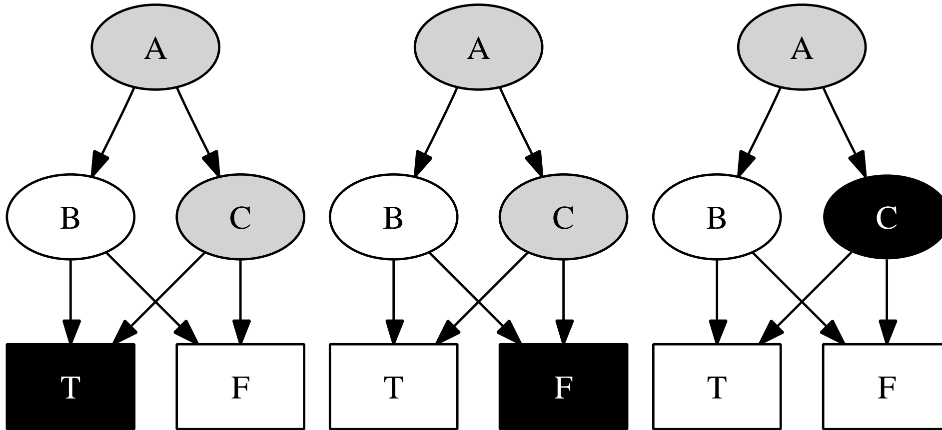


Figure 4.5: Step 4 **Figure 4.6:** Step 5 **Figure 4.7:** Step 6

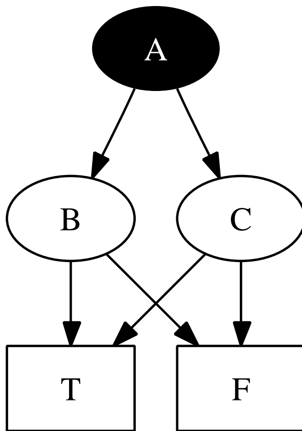


Figure 4.8: Step 7

already received. In the worst-case scenario the running time of the iterator increases exponentially with the number of nodes which gives functions that use it, such as `iddNot()` an exponential runtime.

4.5 Cloning an IDD

Several places in LIBIDD an IDD is cloned and the clone is returned instead of the IDD in question. This is due to the lack of global IDDs and because the operands for the logical operators are sometimes freed after the operation is performed it is necessary to return a clone (using `iddDeepClone()`) instead of the actual IDD which may be freed after the operation is peformed. The function copies the entire IDD rooted at the IDD given to it as argument. The current implementation is seen in appendix A.1.

Not only does the current implementation not use a hashtable which

means that the same IDD idd may be cloned several times, it also uses the iterator to go through the IDD and find all the nodes and terminals. The iterator has the flaw that it cannot see if a node has already been returned (by traversing another path through the graph) so we risk copying the same node or terminal several times. This will also eliminate any sharing that might exist in the IDD thus making the IDD unreduced. Therefore we have implemented a new version of the cloning function. The new implementation consists of an outer function which may be called by the logical operators in LIBIDD and an inner helper function which goes through the IDD using a number of recursive calls. The outer function is seen below.

```
Idd_t *iddOperatorDeepClone(Idd_t *idd, IddHash_t *hashtable) {
  Idd_t *result;

  result = iddOperatorDeepCloneHelper(hashtable, idd);

  return result;
}
```

In the helper function we check if the IDD is a node or a terminal. If it is a terminal we simply return a matching terminal using `iddMakeTerminal()`. If it is a node we look in the cache to see if we have already cloned this node before. If so this is returned. Notice that we use the value `opcounter+1` instead of simply `opcounter`. This is because we cannot increment `opcounter` when we call `iddOperatorDeepClone()` because when we return from the call the value of `opcounter` should be same as before the call, otherwise the calling function cannot continue with using the cache. Therefore we use the value `opcounter+1` here and increment `opcounter` twice for each call to the logical operators. If the node has not been cloned before a new node is made using `iddMakeNode()` and the result is added to the cache. The helper function is seen below.

```
Idd_t *iddOperatorDeepCloneHelper(IddHash_t *hashtable, Idd_t *idd) {
  Idd_t *result;
  int i;
  IddPartitionEntry_t *partition;
  int partition_size;
  cacheentry_t *entry;

  if (idd->type == TERMINAL) {
    if (idd->value.terminal == TRUE) {
      result = iddMakeTerminal(hashtable, TRUE);
      return result;
    } else {
      result = iddMakeTerminal(hashtable, FALSE);
      return result;
    }
```

```
  } else if (idd->type == NODE) {
    entry = iddCacheLookup(HASH1(idd->hashvalue));
    if (entry->tag == opcounter+1 && entry->arg1 == idd) {
      printf("iddOperatorDeepCloneHelper: match found in operator cache\n");
      return entry->result;
    }

    partition = iddPartitionAlloc(idd->value.node.partition_size);
    partition_size = idd->value.node.partition_size;

    for (i = 0; i < idd->value.node.partition_size; i++) {
      //printf("iddOperatorDeepCloneHelper: i = %d\n", i);
      partition[i].lower_bound = idd->value.node.partition[i].lower_bound;
      partition[i].upper_bound = idd->value.node.partition[i].upper_bound;
      partition[i].idd = iddOperatorDeepCloneHelper(hashtable,
                         idd->value.node.partition[i].idd);
    }

    result = iddMakeNode(hashtable, idd->value.node.name,
                         partition, partition_size);

    entry->tag = opcounter+1;
    entry->arg1 = idd;
    entry->result = result;

    return result;
  }

  return result;
}
```

4.6 The Logical Operators

The functions responsible for the logical operations in LIBIDD have a worstcase running time that increases exponentially with respect to the number of nodes in the input IDDs. This chapter describes why this is, how this can be changed into a worstcase running time which only increases polynomically, and what changes we have made to the implementation to achieve this.

4.6.1 NOT

The objective of the NOT function is to return an IDD which is identical to the input IDD except that the TRUE and FALSE terminals are interchanged. The current implementation of NOT in LIBIDD also does this and the IDD it returns is also reduced, however the way in which the result is computed

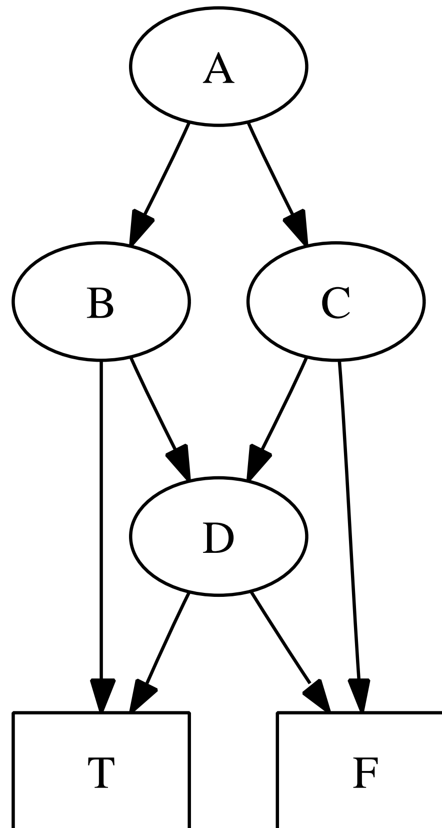


Figure 4.9: IDD used for iddNot() example

is somewhat cumbersome. We come up with a way which is simpler and more straightforward.

The Current Implementation

In the current implementation the function relies on an iterator to go through the input IDD which means that we risk doing the same computation several times. The more sharing in the IDD, the more times we will be performing a computation that we have already done. The source code for `iddNot()` is seen in appendix A.2.

We now give an example of what happens when `iddNot` is run. The example is based in the IDD in figure 4.9.

First the iterator returns A to `iddNot`. The partition of A is then copied into `cur_p`. Nothing more happens on the first pass through the outer while-loop.

On the **second** pass through the outer while-loop the node B is returned by the iterator. The `cur_p` that we set on the first pass (that of the A node) is now pushed onto the partition stack. Nothing more happens on the second

pass through the outer while-loop.

On the **third** pass a TRUE terminal is returned by the iterator. A FALSE terminal is created and because no match is found in the hashtable, the new IDD is inserted into the hashtable. The value `cur_p_pos` associated with B's partion on the partition stack is incremented by one.

On the **fourth** pass the D node is returned by the iterator. The previous `cur_p` (that of B) is pushed onto the partition stack.

On the **fifth** run we run into the TRUE terminal again and should return a FALSE terminal. A new FALSE terminal is still created but because a matching terminal is found in the hashtable it is returned instead and the new terminal is freed. The `cur_p_pos` associated with D's partion is also incremented by one.

On the **sixth** run a FALSE terminal is returned by the iterator. This causes a TRUE terminal to be created which is then inserted into the hashtable. Again the `cur_p_pos` associated with D's partion is incremented by one, and because `cur_p_pos` is now equal to `cur_p_size` the inner while-loop is run. This builds a new IDD with the values located in `cur_name`, `cur_p` and `cur_p_pos`. This IDD is identical to D except that the terminals are interchanged. The partition on top of the stack (that of B) is then popped making the partition of A the new partition at the top of the stack. The IDD referenced in the last position of B's position is now set to that of the new node. Finally `cur_p_pos` for B is incremented once again. This triggers a second pass through the inner while-loop. Here the top of the partition stack is popped again making the partition stack empty. The IDD reference in the first entry of A's partition is set to that of B and the `cur_p_pos` of A is incremented by one.

On the **seventh** run through the outer while-loop C is returned by the iterator. Because `cur_p` still holds the partition of A this is repushed onto to the stack and `cur_p` is set to C's partition.

On the **eighth** pass through the outer while-loop something interesting happens which illustrates the trouble with the current implementation. The iterator returns D once again, the partition of C is pushed onto the stack and `cur_p` is set to D's partition.

On the **ninth** pass we encounter the TRUE terminal again just as we did in the fifth pass and the IDD referenced by the first partition in `cur_p` (that of D) is set to the FALSE terminal.

On the **tenth** pass we reach the FALSE terminal again which results in the IDD reference in the last position of `cur_p` being set to the TRUE terminal. There are now no more partitions in `cur_p` and a new node is initiated with the values now in `cur_name`, `cur_p` and `cur_p_pos`. This is the same IDD that we build in the sixth pass and so we can find it in the hashtable. The IDD reference in the first partition of C is set to this IDD. It is clear that the eight, ninth and tenth passes through the outer while-loop does exactly the same as the fourth, fifth and sixth passes.

On the **eleventh** pass through the outer while-loop we run into the FALSE terminal again. The TRUE terminal is found in the hashtable and is set as the IDD reference in the last position of C. We then go through the inner while-loop two times to build the IDDs for C and A's partitions respectively. After that the final result is returned to the function that called `iddNot()`.

As can be seen from the above example the current implementation of `iddNot()` performs a lot of needless allocations and initializations of IDDs. Furthermore there is the extra overhead of creating and using the iterator and the partition stack. We will get rid of this overhead by presenting a new implementation, this is done next.

The New Implementation

The new implementation is divided into two functions, a main function also called `iddNot()` which is responsible for allocating and freeing the hashtable used for this call to`iddNot()` and a helper function which goes through the IDD to find the terminals through a series of recursive calls.

```
Idd_t *iddNot(Idd_t *idd) {
  IddHash_t hashtable;
  Idd_t *result;

  opcounter++;

  iddHashInit(&hashtable, IDDHASHTBLSIZE);
  result = iddNotHelper(&hashtable, idd);
  iddHashFree(&hashtable);

  return result;
}
```

The way `iddNotHelper()` works is very simple. First we see if the IDD is a terminal. If so we just return the opposite terminal using `iddMakeTerminal()` (remember that `iddMakeTerminal()` looks in the hashtable to see if the terminal already exists, so we do not risk building the same terminal twice). If the IDD is a node we perform a lookup in the operator cache to see if we have already performed NOT on this IDD before. If we have `iddNotHelper()` simply returns the result listed in the operator cache, hence the runtime is bounded by the size of the input IDD. This check will help avoid performing NOT on the same IDD more than once as we did in steps eight through ten in the example of the previous implementation that we gave in section 4.6.1.

If no match is found in the operator cache we copy the partition and call `iddNotHelper()` on the IDD referenced in each partition entry. When we finally exit the for-loop (when we have been through the entire IDD) we

build a new node using the copied partition (or get a matching IDD from the hashtable) with `iddMakeNode()` and an entry in the operator cache is made. `iddNotHelper()` then returns the result to `iddNot()`.

```
Idd_t *iddNotHelper(IddHash_t *hashtable, Idd_t *idd) {
  Idd_t *result = NULL;
  int i;
  IddPartitionEntry_t *partition;
  int partition_size;
  cacheentry_t *entry;

  /* If idd is a terminal just return the opposite */
  if (idd->type == TERMINAL) {
    if (idd->value.terminal == TRUE) {
      result = iddMakeTerminal(hashtable, FALSE);
    } else {
      result = iddMakeTerminal(hashtable, TRUE);
    }
    printf("iddNotHelper: DANGER! We shouldn't get here.\n");
  } else if (idd->type == NODE) {
    entry = iddCacheLookup(HASH1(idd->hashvalue));
    if (entry->tag == opcounter && entry->arg1 == idd) {
      return entry->result;
    }

    partition = iddPartitionAlloc(idd->value.node.partition_size);
    partition_size = idd->value.node.partition_size;

    if(partition_size < 1) {
      printf("iddNotHelper: partition_size less than 1, exiting\n");
      exit(0);
    }

    for (i = 0; i < idd->value.node.partition_size; i++) {
      //printf("iddNotHelper: i = %d\n", i);
      partition[i].lower_bound = idd->value.node.partition[i].lower_bound;
      partition[i].upper_bound = idd->value.node.partition[i].upper_bound;
      partition[i].idd = iddNotHelper(hashtable,
                                idd->value.node.partition[i].idd);
    }

    result = iddMakeNode(hashtable,
                     idd->value.node.name,
                     partition,
                     partition_size);

    entry->tag = opcounter;
    entry->arg1 = idd;
    entry->result = result;
```

```
  }

  return result;
}
```

By implementating `iddNot()` this way we have eliminated the need for the partition stack and the iterator. By eliminating the iterator and using the operator cache, the worst case runtime is reduced from exponential to polynomial. We have also removed a lot of unnecessary allocations and initializations. Furthermore the code is now a lot easier to read. The lookup in the operator cache does, however, present an extra overhead if there is no sharing in the IDD.

We will now go on to looking at the implementation of AND and OR in LIBIDD.

4.6.2 AND and OR

The implementation of AND and OR is very similar so we will only describe the current and new implementation of AND. Appendix A.3 shows the current implementation of `iddAnd()`.

If both IDDs are terminals we return a TRUE terminal if both are true, otherwisewe return false. If one IDD is a terminal and the other is a node, we return a copy of the IDD if the terminal is TRUE, otherwise we return a FALSE terminal. If both IDDs are nodes the course of action depends on the order of the variables they represent. If they represent different variables, the node with the variable last in the order is AND'ed with each of the children of the other node. If the nodes represent the same variable, the partitions are merged and for each new partition entry we AND the IDDs that this partition entry would refer to in each of the two IDDs.

The current implementation of `iddAnd()` does not require too many changes. As with `iddNot()` we use an outer function (`iddAnd()`) and a helper function (`iddAndHelper()`). As before the purpose of the outer function is to allocate, initiate and free the hashtable and the operator cache. This approach also makes it possible to leave the rest of code intact (i.e. functions that call iddAnd are not affected). The outer function is seen below.

```
Idd_t *iddAnd(Idd_t *a_idd, Idd_t *b_idd) {
  Idd_t *result;
  IddHash_t hashtable;

  iddHashInit(&hashtable, IDDHASHTBLSIZE);
  opcounter++;
  result = iddAndHelper(&hashtable, a_idd, b_idd);
  opcounter++;
  iddHashFree(&hashtable);
```

```
  return result;
}
```

Unless both IDDs are nodes we do not perform a lookup in the operator cache. If both are terminals we simply return a terminal using `iddMakeTerminal()`. If one is a node and the other a terminal, either a terminal is returned or a copy of the IDD. Since the copy is now produced with `iddOperatorDeepClone()` which also caches its results there is no need to store the result in the operator cache as well.

If both IDDs are nodes we check if we have already performed this operation before. If we have not we proceed as in the original implementation with the exception that nodes are now build with `iddMakeNode()` to ensure that we do not build the same IDD twice. Also each time we compute a result it is added to the operator cache. This ensures that the function runs in polynomial time. The helper function is seen below.

```
Idd_t *iddAndHelper(IddHash_t *hashtable, Idd_t *a_idd, Idd_t *b_idd) {
  int err;
  IddPartitionEntry_t *partition;
  int partition_size;
  Idd_t *result = NULL;
  IddPartitionEntryTwo_t *merge_res;
  int merge_res_size, merge_res_used;
  cacheentry_t *entry;

  if (a_idd->type == TERMINAL && b_idd->type == TERMINAL) {
    if (a_idd->value.terminal == TRUE && b_idd->value.terminal == TRUE) {
      result = iddMakeTerminal(hashtable, TRUE);
    } else {
      result = iddMakeTerminal(hashtable, FALSE);
    }
  } else {
    if (a_idd->type == TERMINAL) {
      if (a_idd->value.terminal == FALSE) {
        result = iddMakeTerminal(hashtable, FALSE);
      } else {
        /* if a is true then return a copy of b_idd */
        result = iddOperatorDeepClone(b_idd, hashtable);
      }
    } else if (b_idd->type == TERMINAL) {
      if (b_idd->value.terminal == FALSE) {
        result = iddMakeTerminal(hashtable, FALSE);
      } else {
        /* if b is true then return copy of a */
        result = iddOperatorDeepClone(a_idd, hashtable);
      }
    } else {
```

```
entry = iddCacheLookup(HASH2(a_idd->hashvalue, b_idd->hashvalue));

if (entry->tag == opcounter &&
    entry->arg1 == a_idd &&
    entry->arg2 == b_idd) {
  return entry->result;
}

if (a_idd->value.node.name > b_idd->value.node.name) {
  //printf("iddAndHelper: a > b\n");
  /* do the apply left thing */
  partition_size = b_idd->value.node.partition_size;
  partition = iddPartitionAlloc(partition_size);
  err = iddAndApplyLeft(hashtable, partition, partition_size,
                        b_idd->value.node.partition,
                        b_idd->value.node.partition_size,
                        a_idd);
  if (err == 0) {
    printf("iddAndHelper received: %d from iddAndApplyLeft\n", err);
  }

  result = iddMakeNode(hashtable, b_idd->value.node.name,
                       partition, partition_size);
} else if (a_idd->value.node.name < b_idd->value.node.name) {
  //printf("iddAndHelper: a < b\n");
  /* do the apply right b thing */
  partition_size = a_idd->value.node.partition_size;
  partition = iddPartitionAlloc(partition_size);
  err = iddAndApplyRight(hashtable, partition, partition_size,
                         a_idd->value.node.partition,
                         a_idd->value.node.partition_size,
                         b_idd);
  if (err == 0)
    printf("iddAndHelper received: %d from iddAndApplyRight\n", err);

  result = iddMakeNode(hashtable, a_idd->value.node.name,
                       partition, partition_size);
} else {
  //printf("iddAndHelper: a == b\n");
  /* do the merge thing */
  merge_res_size = a_idd->value.node.partition_size +
    b_idd->value.node.partition_size;
  merge_res = iddPartitionTwoAlloc(merge_res_size);

  merge_res_used = iddMergeTwo(a_idd->value.node.partition,
                               a_idd->value.node.partition_size,
                               b_idd->value.node.partition,
                               b_idd->value.node.partition_size,
                               merge_res, merge_res_size);
```

```
        if (merge_res_used == 0)
          printf("iddAndHelper received error %d form iddMergeTwo",
                 merge_res_used);

        partition_size = merge_res_used;
        partition = iddPartitionAlloc(partition_size);
        err = iddAndMerge(hashtable, merge_res, merge_res_used,
                          partition, partition_size);

        if (err == 0)
          printf("iddAndHelper received error %d from iddAndMerge", err);

        iddPartitionTwoFree(merge_res); /* won't need this anymore */

        result = iddMakeNode(hashtable,
                             a_idd->value.node.name,
                             partition,
                             partition_size);
      }
      entry->tag = opcounter;
      entry->arg1 = a_idd;
      entry->arg2 = b_idd;
      entry->result = result;
    }
  }

  return result;
}
```

The implementation is now consistent with the description we gave in chapter 2.3 and the worst-case runtime is $O(n * m)$ where n is the number of nodes in `a_idd` and m is the number of nodes in `b_idd`. The key here is the use of an operator cache which ensures that we do not perform the same operation twice.

4.7 iddDeepFree

The function `iddDeepFree()` is a good example of how ineffective a function can get by the use of an iterator. The purpose of the function is quite simple. Given an IDD as input the function finds all the nodes and terminals in the IDD and frees the memory associated with them. The problem is in the way this is currently implemented, see appendix A.4. The function uses the iterator to get the nodes and terminals one by one and then store them in an array. Of course we would have a problem if we tried to free the same node or terminal twice so before a node or terminal is stored in the array, the function runs through it and compares each each entry with

```
/* Registers all nodes and terminals in the idd */ void
iddRegister(Idd_t *idd) {
  int i;

  if (idd->marked) {
    return;
  }

  idd->marked = 1;
  iddarray[array_used] = idd;
  array_used++;

  if (array_used >= IDD_ARRAY_SIZE) {
    printf("iddRegister out of memory");
    exit(0);
  }

  if (idd->type == NODE) {
    for (i = 0; i < idd->value.node.partition_size; i++) {
      iddRegister(idd->value.node.partition[i].idd);
    }
  }
}
```

Table 4.6: The implementation of iddRegister()

the IDD returned from the iterator. If it is already in the array it is simply discarded, otherwise it is added. Because of the use of an iterator and the check of the array the worst-case runtime of `iddDeepFree()` is exponential and even best-case is $O(n^2)$ because of the search through the array.

To avoid using an iterator to get hold of all the nodes and terminals in an IDD we implement a function called `iddRegister()`. The job of this function is to register all the nodes and terminals in the IDD in a global array. Unlike the iterator, `iddRegister()` stops going down a path if it encounters a node or terminal it has seen before. In order to do this we add a flag to the IDD struct that indicates if we have encountered it before. When `iddRegister()` is called it first checks if we have visited the given IDD before. If not the flag `marked` is set and the node or terminal in question is added to the global array. If it is a node `iddRegister()` is then called on each of the children.

With the use of `iddRegister()` the implementation of `iddDeepFree()` becomes very simple. It is simple a matter of calling `iddRegister()` on the root and then freeing the IDDs in the global array. The worst-case runtime of `iddDeepFree` is now linear, $O(n)$, with respect to the number of nodes n in the IDD.

```
void iddDeepFree(Idd_t *idd) {
  int i;

  array_used = 0;
  iddRegister(idd);

  /* free list of unique idds */
  for (i = 0; i < array_used; i++) {
    if (iddarray[i]->type == NODE) {
      iddPartitionFree(iddarray[i]->value.node.partition);
    }
    iddFree(iddarray[i]);
  }
}
```

Table 4.7: The new implementation of iddDeepFree()

4.8 Benchmarks

We now run some tests to see how the new implementation has affected the runtime of *cfconf*.

4.8.1 Building the filters

The filters are built using a simple algorithm. We use a rule generator that goes through a network trace one packet at a time and builds a rule that denies all packets from the sender to the receiver stated in the packet. The rule also includes the protocol and port numbers used in the packet headers. If a packet is encountered that would produce a rule already in the filter it is skipped. This proces continues until the desired number of rules is reached. For this test we used the option to begin the rule building process at different points in the network trace. Furthermore the IP adresses are mangled before they are used to construct the rules. This means the filters will not begin with the same rules.

The various filters used for the test were all built using the same big network trace. The network trace was produced using `tcpdump`. We used this tool to listen on incoming and outgoing traffic on a single machine to produce the trace. The machine was positioned in a group room on AAU and the traffic was produced by ordinary network applications such as a web browser, an IM client, an SSH login on another server and so on. Information about the packet headers was collected and saved. No information about the packet contents was saved as it is not needed. We continued the recording of network traffic until we reached 100.000 packets. We used the standard settings meaning that we saved the first 96 bytes from each packet - enough for the IP and TCP headers.

We made 12 different filters with 10, 20, 50, 100, 250, 500, 1,000, 2,500, 5,000, 10,000, 25,000 and 50,000 rules respectively.

The filter sizes range from very few rules to an extremely high number of rules. 10 rules is a very short filter which doesn't allow a firewall administrator to be very specific about what traffic he wants to permit and deny. Therefore it is safe to use 10 rules as the smallest number of rules in a firewall filter. On the other hand, 50,000 rules is an extremely large filter which allows for very specific rules. It is very unlikely that a filter will ever get this large, so the range of likely filter sizes is well covered. The reason for using these fabricated filters is that it has not been possible to find real-life filters of an acceptable size. The filters we have found are example filters of 10-20 rules which are not useful for testing as they are to small to give any measurable diffence in the test results before and after an optimization.

4.8.2 Test setup

This section describes how the actual tests were done and what results we got. The tests were run on an AMD XP 1800+ with 512 MB PC133 SDRAM. Initially we had some problems with the original implementation which resulted in test results for 25,000 rules and 50,000 being very similar. We discovered that this is due to the fact that one of the functions in libidd, `iddDeepFree`, runs out of memory before the 25,000 rules can be processed. From running gprof we learn that iddDeepFree runs exactly 311,069 times for 25,000 and 50,000 rules. It turns out that this is because when an array used in `iddDeepFree` reaches a predefined size, the programs halts and and error message is printed. The array is used to keep track of the nodes in the IDD, so apparently the filters we use generate a large number of nodes. The maximum size of the array is currently set to 250,000. The purpose of this limit is to ensure that `iddDeepFree` halts before it runs out of memory. However, in our case this is much too soon. Therefore we have reset the maximum size to 4,000,000. This value ensures that the program does not halt before having processed 50000 rules.

An example filter is shown in table 4.8. The program is run with the `-s` option which only simulates creating the filter. The filter is not sent to the kernel when using this option. This is not necessary as our focus is on the userspace part of CF. We have, however, modified CF slightly so that the final MTIDD is still built which is normally not the case when using the simulate option. This MTIDD is what cfconf passes on to the kernel module for CF to use. We did this to create as realistic a test setup as possible. Building the MTIDD is still a part of the userspace part of CF and should therefore be included when doing the tests. After each test the results will be moved as the following test results would otherwise be added to those already present.

Figures 4.10 and 4.11 show the running time for *cfconf* for the original

```
INPUT permit tcp host 51.252.160.40 eq 3128 host 73.165.243.220 eq 2306
INPUT permit tcp host 99.225.183.39 eq 3128 host 211.69.141.119 eq 2306
INPUT permit tcp host 154.46.120.186 eq 2306 host 191.168.185.214 eq 3128
INPUT permit tcp host 3.211.14.207 eq 2306 host 189.94.235.147 eq 3128
INPUT permit udp host 101.112.91.5 eq 1985 host 51.230.197.202 eq 1985
INPUT permit tcp host 116.205.168.122 eq 3128 host 199.101.76.19 eq 2306
INPUT permit tcp host 215.159.5.255 eq 3128 host 138.59.72.157 eq 2306
INPUT permit tcp host 135.109.129.51 eq 2306 host 150.191.4.168 eq 3128
INPUT permit tcp host 134.25.106.61 eq 2306 host 169.179.25.44 eq 3128
INPUT permit tcp host 148.214.128.253 eq 3128 host 107.157.118.171 eq 2306
```

Table 4.8: A filter with 10 rules

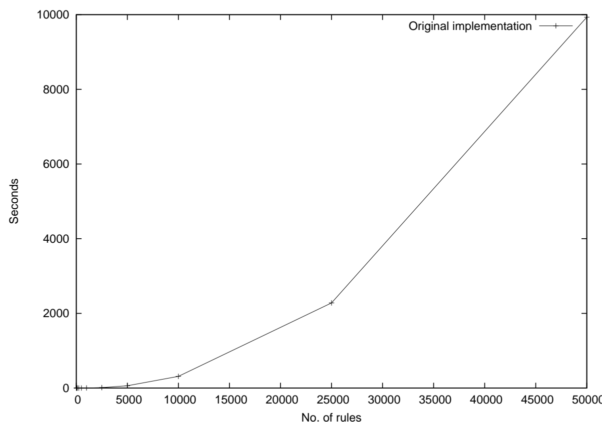


Figure 4.10: The runtime of cfconf with the original implementation

and new implementation. Here we see a huge decrease in runtime. For 50,000 rules it has dropped from just under 10,000 seconds to about 18 seconds.

If we look at the matches found in the operator cache, we see that `iddAnd()`, `iddOr()` and `iddNot()` only find results that are terminals. This is, however, not so strange when we consider the way rules are built and the ruleset that we have. All the rules have the same format and the same variables which means that when we add a new rule to the filter the root node of both IDDs is the same. When we call `iddOr()` the IDDs will be merged at the root node (or the following node if the partitions of both root nodes are identical). Because LIBIDD does not currently support global IDDs the results stored in the operator cache are removed when `iddOr()` exits. The same goes for the IDD hashtable. Therefore the information we acquire about IDDs and operations cannot be used in subsequent calls to `iddOr()`. The same goes for `iddAnd()`. To make the operator cache more useful it will require the implementation of global IDDs, but this was not possible in this project due to lack of time.

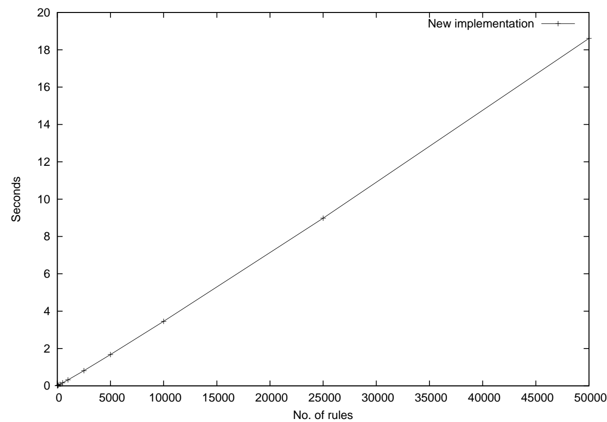


Figure 4.11: The runtime of cfconf with the new implementation

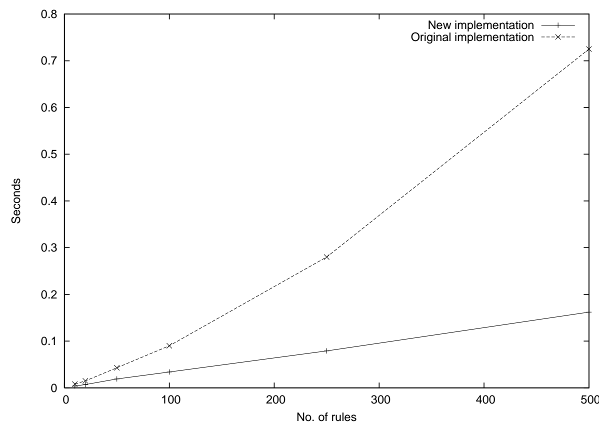


Figure 4.12: Comparing runtimes up to 500 rules

4.9 Summary

In this chapter we have identified the main problems of the current implementation of LIBIDD. First of all IDDs are needlessly allocated and initiated when an identical existing IDD exists. This problem is removed by introducing a new IDD constructor which first checks if an identical IDD exists. If so this is returned instead of making a new one. Secondly we replaced the existing function for getting hashvalues for IDDs. This was based on turning the IDD into a string representation first and then hashing it. It also required the hashtable size to be a prime number. The new function simplifies the hashing proces. We also introduced an operator cache for holding information about previous logical operations on IDDs. However, the effectiveness of this is reduced by the fact that LIBIDD does not currently use global IDDs. We have also changed the implementation of the logical operators NOT, AND and OR and by using the new IDD constructor and

operator cache we are able to reduce the worst case run time from exponential to polynomical. We have also improved the runtime of the function `iddDeepFree()` by getting rid of the iterator. The benchmarks in section 4.8 show how the growth in runtime for *cfconf* has gone from exponential to polynomical. While we have improved the implementation of LIBIDD we have also seen that in order for it to be really efficient the implementation of LIBIDD must be altered to incorporate global IDDs. However, this was not possible due to lack of time.

Chapter 5

Performance Analysis

Here we test *cfconf* using the performance analysis tools described in chapter 3.

5.1 Testing the old implementation

Tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6 shows the 10 most time consuming functions for 500, 1000, 5000, 10000, 25000 and 50000 rules respectively.

The functions `iddDeepFree()`, `iddEqual()`, `iddHashFind`, `iddComparePartitionBounds`, `iddIteratorPreorderInit()` and `iddIteratorPreorderNext` take up a lot of the total time spent.

It is interesting to notice that of these functions, `iddDeepFree()` has been altered and `iddEqual()`, `iddComparePartitionBounds`, `iddIteratorPreorderInit()` and `iddIteratorPreorderNext` are no longer used.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 41.69      2.91     2.91     6520     0.00     0.00  iddDeepFree
 12.11      3.76     0.85   126798     0.00     0.00  iddHashGetHashValue
  7.88      4.31     0.55  1313417     0.00     0.00  iddEqual
  4.87      4.65     0.34  3110302     0.00     0.00  iddIteratorPreorderNext
  4.15      4.94     0.29  2635244     0.00     0.00  iddIteratorPreorderInit
  4.01      5.22     0.28      511     0.00     0.00  iddHashInit
  3.44      5.46     0.24  2659302     0.00     0.00  nodeStackPush
  2.72      5.65     0.19      502     0.00     0.00  iddHashFree
  1.58      5.76     0.11    70589     0.00     0.00  iddHashFind
  1.50      5.86     0.11   589328     0.00     0.00  iddComparePartitionBounds
```

Table 5.1: The gprof flat profile for 500 rules

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
47.61	11.33	11.33	13020	0.00	0.00	iddDeepFree
7.98	13.23	1.90	5033894	0.00	0.00	iddEqual
6.05	14.67	1.44	11115408	0.00	0.00	iddIteratorPreorderNext
5.71	16.03	1.36	267258	0.00	0.00	iddHashGetHashValue
4.16	17.02	0.99	10085610	0.00	0.00	iddIteratorPreorderInit
3.05	17.75	0.73	2280397	0.00	0.00	iddComparePartitionBounds
2.35	18.31	0.56	10136850	0.00	0.00	nodeStackPush
2.23	18.84	0.53	151957	0.00	0.00	iddHashFind
1.85	19.28	0.44	1002	0.00	0.00	iddHashFree
1.81	19.71	0.43	10098630	0.00	0.00	nodeStackInit

Table 5.2: The gprof flat profile for 1,000 rules

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
38.92	256.52	256.52	95018	0.00	0.00	iddDeepFree
17.29	370.47	113.95	103309060	0.00	0.00	iddComparePartitionBounds
11.44	445.89	75.42	1051662	0.00	0.00	iddHashFind
9.99	511.74	65.85	135007858	0.00	0.00	iddEqual
6.07	551.72	39.98	277223558	0.00	0.00	iddIteratorPreorderNext
4.43	580.93	29.21	270182040	0.00	0.00	iddIteratorPreorderInit
2.10	594.78	13.85	270461277	0.00	0.00	nodeStackPush
1.78	606.49	11.70	1988216	0.00	0.00	iddHashGetHashValue
1.75	618.04	11.55	270277058	0.00	0.00	nodeStackInit
1.64	628.88	10.84	270182040	0.00	0.00	iddIteratorPreorderFree

Table 5.3: The gprof flat profile for 5,000 rules

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ks/call	total Ks/call	name
41.27	1487.82	1487.82	129940	0.00	0.00	iddDeepFree
13.65	1979.94	492.12	1917025	0.00	0.00	iddHashFind
10.76	2367.81	387.88	256658775	0.00	0.00	iddComparePartitionBounds
9.86	2723.33	355.52	546313708	0.00	0.00	iddEqual
4.89	2899.60	176.27	1106035125	0.00	0.00	iddIteratorPreorderNext
3.41	3022.68	123.08	1092835783	0.00	0.00	iddIteratorPreorderInit
2.50	3112.96	90.28	66929508	0.00	0.00	mtiddEqual
2.43	3200.60	87.65	109939	0.00	0.00	mtiddHashFind
1.45	3253.01	52.40	1093441934	0.00	0.00	nodeStackPush
1.42	3304.22	51.21	25229135	0.00	0.00	mtiddComparePartitionBounds

Table 5.4: The gprof flat profile for 10,000 rules

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ks/call	total Ks/call	name
24.88	2482.84	2482.84	324924	0.00	0.00	iddDeepFree
10.85	3565.89	1083.05	2080999663	0.00	0.00	iddIteratorPreorderNext
9.46	4509.53	943.64	3167286725	0.00	0.00	iddEqual
7.46	5253.81	744.28	5151859	0.00	0.00	iddHashFind
7.33	5985.03	731.21	2040161922	0.00	0.00	iddIteratorPreorderInit
6.02	6586.01	600.99	422285683	0.00	0.00	mtiddEqual
5.93	7177.35	591.33	274921	0.00	0.00	mtiddHashFind
5.26	7702.45	525.10	1494524557	0.00	0.00	iddComparePartitionBounds
3.28	8029.65	327.20	160119507	0.00	0.00	mtiddComparePartitionBounds
3.16	8345.08	315.44	847706150	0.00	0.00	mtiddIteratorPreorderNext

Table 5.5: The gprof flat profile for 25,000 rules

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ms/call	total Ms/call	name
18.36	4187.34	4187.34	4235926685	0.00	0.00	iddEqual
11.35	6776.65	2589.31	16843429	0.00	0.00	iddIteratorPreorderNext
10.07	9072.31	2295.65	4178047795	0.00	0.00	iddIteratorPreorderInit
7.78	10846.46	1774.15	1738313869	0.00	0.00	iddComparePartitionBounds
5.84	12178.62	1332.16	649924	0.00	0.00	iddDeepFree
5.40	13410.80	1232.18	10842705	0.00	0.00	iddHashFind
5.27	14613.27	1202.47	4210664578	0.00	0.00	nodeStackPush
5.13	15783.52	1170.25	1737137879	0.00	0.00	mtiddEqual
4.24	16750.01	966.49	4178047795	0.00	0.00	iddIteratorPreorderFree
4.04	17670.72	920.71	4178697719	0.00	0.00	nodeStackInit

Table 5.6: The gprof flat profile for 50,000 rules

5.2 Testing the new implementation

Tables 5.7, 5.8, 5.9, 5.10, 5.11 and 5.12 shows the 10 most time consuming functions for 500, 1000, 5000, 10000, 25000 and 50000 rules respectively.

Unlike the results for the old implementation the results for the new implementation are more similar. For each filter `iddHashInit()` and `iddHashFree()` are by far the most time consuming functions. If at some point global IDDs are implemented in LIBIDD we will not have all these calls to these two functions but on the other the hashtable will contain more IDDs and there will be more collisions. It would be interesting, at some point in the future, to see what effects implementing global IDDs will have on the runtime of *cfconf* and also what functions will then be the most time consuming.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 51.33      1.16     1.16     3018    0.00     0.00  iddHashInit
 37.61      2.01     0.85     3018    0.00     0.00  iddHashFree
  2.21      2.06     0.05     7589    0.00     0.00  iddOperatorDeepCloneHelper
  1.77      2.10     0.04    53291    0.00     0.00  iddHashFindTerminal
  0.89      2.12     0.02    23949    0.00     0.00  iddhash2
  0.89      2.14     0.02    20949    0.00     0.00  iddHashFindNode
  0.89      2.16     0.02     6018    0.00     0.00  iddRegister
  0.89      2.18     0.02     5501    0.00     0.00  yylex
  0.44      2.19     0.01    25131    0.00     0.00  iddHashInsert
  0.44      2.20     0.01     3926    0.00     0.00  mtiddIddMergeRec
```

Table 5.7: The gprof flat profile for 500 rules

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 54.94      2.56     2.56     6018    0.00     0.00  iddHashInit
 34.12      4.15     1.59     6018    0.00     0.00  iddHashFree
  1.50      4.22     0.07   111242    0.00     0.00  iddHashFindTerminal
  1.07      4.27     0.05    16233    0.00     0.00  iddOperatorDeepCloneHelper
  0.86      4.31     0.04    12018    0.00     0.00  iddRegister
  0.64      4.34     0.03    49419    0.00     0.00  iddGetHashValue
  0.64      4.37     0.03    11001    0.00     0.00  yylex
  0.64      4.40     0.03     6065    0.00     0.00  iddMergeTwo
  0.64      4.43     0.03    48706    0.00     0.00  iddCacheLookup
  0.43      4.45     0.02   111242    0.00     0.00  iddGetHashValueTerminal
```

Table 5.8: The gprof flat profile for 1,000 rules

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 51.32     11.91    11.91    30016    0.00     0.00  iddHashInit
 36.45     20.37     8.46    30016    0.00     0.00  iddHashFree
  1.55     20.73     0.36    91733    0.00     0.00  iddOperatorDeepCloneHelper
  0.90     20.94     0.21   596963    0.00     0.00  iddHashFindTerminal
  0.90     21.15     0.21   176799    0.00     0.00  iddOrHelper
  0.86     21.35     0.20   259915    0.00     0.00  iddhash2
  0.86     21.55     0.20    60016    0.00     0.00  iddRegister
  0.82     21.74     0.19    55001    0.00     0.00  yylex
  0.62     21.89     0.15   259915    0.00     0.00  iddGetHashValue
  0.60     22.03     0.14   229915    0.00     0.00  iddHashFindNode
```

Table 5.9: The gprof flat profile for 5,000 rules

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 50.87     24.22    24.22    59977     0.00     0.00  iddHashInit
 36.28     41.49    17.27    59977     0.00     0.00  iddHashFree
  2.21     42.54     1.05   193266     0.00     0.00  iddOperatorDeepCloneHelper
  1.09     43.06     0.52  1234039     0.00     0.00  iddHashFindTerminal
  0.79     43.44     0.38   119938     0.00     0.00  iddRegister
  0.78     43.81     0.37   472533     0.00     0.00  iddHashFindNode
  0.69     44.14     0.33   532494     0.00     0.00  iddhash2
  0.61     44.43     0.29   373546     0.00     0.00  iddOrHelper
  0.55     44.69     0.26   541378     0.00     0.00  iddCacheLookup
  0.51     44.93     0.25   109949     0.00     0.00  yylex
```

Table 5.10: The gprof flat profile for 10,000 rules

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 50.15     60.76    60.76   149970     0.00     0.00  iddHashInit
 35.07    103.26    42.49   149970     0.00     0.00  iddHashFree
  2.73    106.57     3.31   509650     0.00     0.00  iddOperatorDeepCloneHelper
  1.22    108.05     1.48  3212111     0.00     0.00  iddHashFindTerminal
  1.20    109.50     1.45  1222633     0.00     0.00  iddHashFindNode
  0.99    110.69     1.20   299922     0.00     0.00  iddRegister
  0.83    111.69     1.00  1372585     0.00     0.00  iddhash2
  0.73    112.58     0.89   772629     0.00     0.00  iddAndHelper
  0.69    113.42     0.84   274937     0.00     0.00  yylex
  0.69    114.25     0.83   986022     0.00     0.00  iddOrHelper
```

Table 5.11: The gprof flat profile for 25,000 rules

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 49.51    122.31   122.31   299970     0.00     0.00  iddHashInit
 34.34    207.13    84.82   299970     0.00     0.00  iddHashFree
  3.08    214.75     7.62  1034082     0.00     0.00  iddOperatorDeepCloneHelper
  1.45    218.32     3.57  2482426     0.00     0.00  iddHashFindNode
  1.19    221.27     2.95  6532990     0.00     0.00  iddHashFindTerminal
  1.00    223.74     2.47   599922     0.00     0.00  iddRegister
  0.96    226.10     2.36  2782378     0.00     0.00  iddhash2
  0.75    227.95     1.86  1543451     0.00     0.00  iddAndHelper
  0.69    229.65     1.70  2001985     0.00     0.00  iddOrHelper
  0.66    231.28     1.63  2782378     0.00     0.00  iddGetHashValue
```

Table 5.12: The gprof flat profile for 50,000 rules

Chapter 6

Other solutions

This chapter discusses possible ways of improving CF by altering the data representation of the IDDs or the way they are manipulated. So far we have been looking at optimizing the code in order to reduce the time it takes to compile the filters, but there may also be something to gain by changing the representation of the IDDs or the algorithms used to manipulate them. Many papers exist that deal with different ways of representing decision diagrams and manipulating them. We will here take a look at some of the ideas presented in these papers. We particularly look at the ordering of variables and in chapter 7 we benchmark several different orders to see what effect chosing a different order will have.

6.1 Ordering of variables

In CF it is up to the user to chose a suitable ordering and once that is done it remains fixed. If the order is poorly chosen it will not only influence runtime but also require the decision diagram to be build again from scratch if the user decides to use a new order.

The ordering of variables in a decision diagram has a significant influence on the size of the graph [OGM] and therefore also on the time it takes to traverse the graph with a given input. As it will also take more time and require more memory to build a filter the bigger the graph is, this is also of interest to us.

Finding an optimal variable ordering is NP-complete [OGM], so a heuristic approach is usually applied. We will here present some of these approaches.

Richard Rudell has proposed a solution [Rud] where a minimization algorithm is run periodically to reduce the size of the decision diagram. The solution was developed for ordered binary decision diagrams (OBDD), but can also be applied to IDDs as we shall show here.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	initial
x_1	x_3	x_2	x_4	x_5	x_6	x_7	swap (x_2,x_3)
x_1	x_3	x_4	x_2	x_5	x_6	x_7	swap (x_2,x_4)
x_1	x_4	x_3	x_2	x_5	x_6	x_7	swap (x_3,x_4)
x_1	x_4	x_2	x_3	x_5	x_6	x_7	swap (x_3,x_2)
x_1	x_2	x_4	x_3	x_5	x_6	x_7	swap (x_4,x_2)
x_1	x_2	x_3	x_4	x_5	x_6	x_7	swap (x_4,x_3)
x_1	x_3	x_2	x_4	x_5	x_6	x_7	swap (x_2,x_3)
x_1	x_3	x_4	x_2	x_5	x_6	x_7	swap (x_2,x_4)

Table 6.1: Window Permutation Algorithm example

Rudell proposes two algorithms which both rely on swapping the variables in two adjacent layers in a decision diagram. If $F = (x_i, F_1, F_0)$ (meaning that if x_i is true then F_1 is chosen and F_0 otherwise) is a node at level i in the OBDD, then F can be replaced by the tuple $(x_{i+1}, (x_i, F_{11}, F_{01}), (x_i, F_{01}, F_{00}))$. Here F_{11} is the node reached by evaluating x_i and x_{i+1} to 1 or TRUE. In an IDD the corresponding tuple would in most cases be somewhat longer, but the principle is the same. That is, a variable could be evaluated to a value that lies within one of many intervals thus producing a tuple with many more elements. The variables are reordered based on one of two algorithms:

Window Permutation Algorithm This algorithms works by selecting a layer and then trying all possible permutations of the k adjacent layers. After all possibilities have been examined, the variables are swapped again in order to place them at the level that yields the smallest decision diagram. This requires an initial $k! - 1$ swaps followd by up to $k(k-1)/2$ swaps to return the variables to the optimal level. Table 6.1 (borrowed from Rudells paper [Rud]) demonstrates the proces. First 5 swaps are made to test all permutations of x_2, x_3 and x_4. Then a further 3 swaps are made to return the variables to the best position found. This is the worst-case scenario.

Rudell also sugests using marking to indicate that the current permutation of the variables is optimal. The mark is reset when a new permutation is found for one of the preceding k-1 levels. When all the levels are marked, there can be no further optimization using this algorithm.

Sifting Algorithm This algorithm works on one variable at a time instead of several as in the Window Permutation Algorithm. All other variables are presumed fixed. The variable in question is repeatedly swapped with its successor until it reaches the bottom of the decision diagram. It is then swapped with its predecessor until it reaches the top of the diagram. The optimal position is remembered and the variable is once again swapped

x_1	x_2	x_3	x_4	x_5	x_6	x_7	initial
x_1	x_2	x_3	x_5	x_4	x_6	x_7	swap (x_4,x_5)
x_1	x_2	x_3	x_5	x_6	x_4	x_7	swap (x_4,x_6)
x_1	x_2	x_3	x_5	x_6	x_7	x_4	swap (x_4,x_7)
x_1	x_2	x_3	x_4	x_5	x_4	x_7	swap (x_7,x_4)
x_1	x_2	x_3	x_5	x_4	x_6	x_7	swap (x_6,x_4)
x_1	x_2	x_3	x_4	x_5	x_6	x_7	swap (x_5,x_4)
x_1	x_2	x_4	x_3	x_5	x_6	x_7	swap (x_3,x_4)
x_1	x_4	x_2	x_3	x_5	x_6	x_7	swap (x_2,x_4)
x_4	x_1	x_2	x_3	x_5	x_6	x_7	swap (x_1,x_4)
x_1	x_4	x_2	x_3	x_5	x_6	x_7	swap (x_4,x_1)
x_1	x_2	x_4	x_3	x_5	x_6	x_7	swap (x_4,x_2)
x_1	x_2	x_3	x_4	x_5	x_6	x_7	swap (x_4,x_3)
x_1	x_2	x_3	x_5	x_4	x_6	x_7	swap (x_4,x_5)
x_1	x_2	x_3	x_5	x_6	x_4	x_7	swap (x_4,x_6)
x_1	x_2	x_3	x_5	x_6	x_7	x_4	swap (x_4,x_7)

Table 6.2: Sifting Algorithm example

with is successor until it reaches this position. The advantage of this algorithm is that a variable may be moved a long distance. It does, however, require a lot of swaps in the worst case scenario (if the optimal position for a variable is at the bottom of the graph). An example of the worst-case scenario is shown in table 6.1. Here all possible position are explored using 9 swaps and the variable is returned to the optimal position using another 6 swaps.

Both algorithms do not try to find a complete overall order in one go. Instead they work on one variable at a time in order to reduce the time spent on each run of the algorithm.

6.2 Complement nodes

Brace et al. [KSBB90] proposes the use of complement edges to reduce the size of BDDs. This makes it possible to represent two BDDs, that are similar except that their terminal nodes (True and False) are interchanged. By setting a complement bit on a given edge you indicate that the associated formula is to be interpreted as the complement of itself. To maintain consistency the true-edge of a node must always be a regular edge. Brace et al. use the low bit of node pointers as the complement bit thus avoiding added memory. A number of experiments were performed which showed a decrease in size of 7 % for the BDD and a reduction in the time it took to generate the BDDs by almost 50%.

The results presented by Brace et al. appear very promising, at least

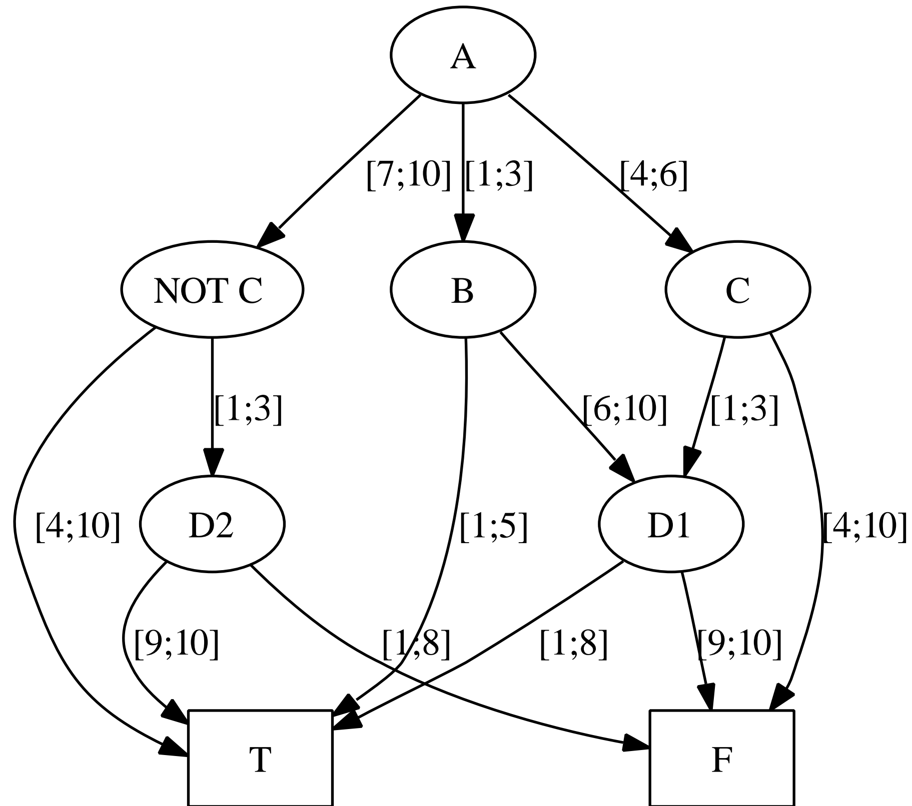


Figure 6.1: A simple IDD that does not utilize complement nodes

enough to make it worth while to see if their methods can be of use to us. Although developed for BDDs it is alo possible to apply the approach to IDDs as they have been shown to be equivalent. The question is how feasible it will be. Will it require to many modifications to the representation of the IDDs or can it be done fairly simple.

In LIBIDD complement nodes could be implemented by using the low bit of the IDD reference in partition entries in nodes. Alternatively an extra bit could be added to the IDD struct although this would then, as stated, result in additional memory use.

If complement nodes are implemented this would make it unnecessary to copy the entire IDD when we perform NOT on it. Instead we set the complement bit of the IDD reference in the partition entry leading to it. Figure 6.1 shows a simple IDD where the last partition of the node A refers to $\neg C$. Figure 6.2 shows the same IDD but with the use of complement nodes. The ′ indicates that an edge is a complement edge.

Depending on how much NOT is used, the implementation of complement nodes can help reduce the size of the IDD and also the time it takes to build it because we can reduce the number of times we have to con-

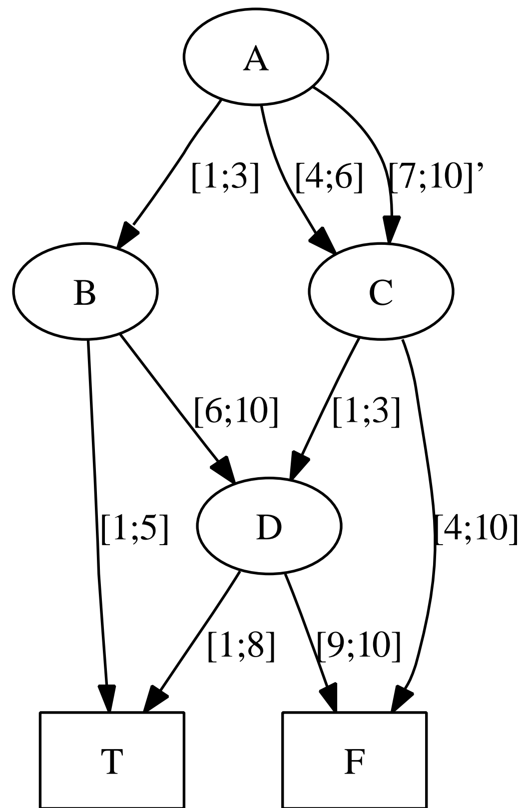


Figure 6.2: A simple IDD that incorporates complement nodes

struct a new node. The use of complement nodes will require that LIBIDD supports global IDDs because we cannot risk that the IDD referenced by a complement edge is freed at some point because the original IDD is no longer needed.

6.3 Summary

We have looked at different ways of changing the data structure of the decision diagram in order to reduce the time it takes to build it. We have primarily looked at the order of the variables in the decision diagram.

We have also looked at the possibility of implementing complement nodes which can help reduce the size of the IDD and also the time it takes to build it, because it will no longer be necessary to copy an entire IDD when performing not. Instead a complement bit will be set on the edge leading to the IDD in question which means that the IDD should be seen as the complement of the one actually present (i.e. the TRUE and FALSE terminals are interchanged).

We will no go on to examine the effects of changing the order of vari-

ables in CD. We will use a variant of one of the dynamic algorithms, the sifting algorithm, as this provides a simple way of determining an order for the decision diagram. Also our focus is not on finding the very best order but on discovering the effects of changing the order in CF.

Chapter 7

Finding a better order for CF

In this chapter we look at finding a better order for CF than the existing one using a variant of the sifting algorithm described in chapter 6. First we show how changing the order of variable can reduce the size of a graph. Then we present the algorithm we will use to find the new order. We then find the best position in the order for each variable and finally we test what effect changing the order has on the runtime.

7.1 The current order

In the current implementation of LIBIDD the order is fixed. The order is:

1. IP_PROTOCOL The number of the transport layer protocol used. Currently only 6 protocols are supported - TCP (6), UDP (17), ICMP (1), IGMP (2), IP in IP (4), PIM (?).
2. IP_FRAGMENT_DATALEN This is the *fragment offset* and *datalength* fields which have been combined.
3. IP_SADDR The source IP address.
4. IP_DADDR The destination IP address.
5. TCP_DEST The destination TCP port.
6. TCP_SOURCE The source TCP port.
7. UDP_DEST The destination UDP port.
8. UDP_SOURCE The source UDP port.

This gives a total of 8! = 40320 different permutations. There are also some permutations that would not make a difference. For instance, using the above order but with with UDP_DEST, UDP_SOURCE, TCP_DEST,

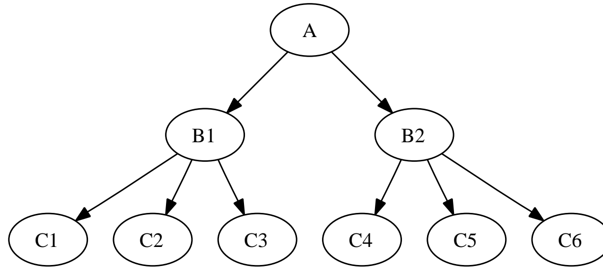


Figure 7.1: An example of a small graph

TCP_SOURCE as the last four would give the same result, as a rule would naturally never include both tcp and udp ports, and they would therefore never be found on the same path through the graph. This also raises the question of why CF does not simply use the same to variables for port numbers instead of having two for UDP and two for TCP. This could be changed in the future. There are, however, still a large number of possible permutations and we will now look at how we can determine the best suitable order.

7.2 The algorithm

The primary goal of the project is to reduce the compile time for the filters. Therefore we will come up with a method to find the a better order for CF.

If we are to be able to measure any difference then we have to use filters with a large enough number of rules in it, but this will also make it a very time-consuming experiment. Therefore we will look at how we can reduce the number of variable permutations that we have to test.

The range of the variables are very different. IP_PROTOCOL has a range from 0 to 255 (8 bits); TCP_DEST, TCP_SOURCE, UDP_DEST and UDP_SOURCE have a range from 0 to 65,535 (16 bit); IP_SADDR and IP_DADDR have a range from 0 to 4,294,967,296 (32 bit). Although not all values for a variable are in use for a single graph, there is no doubt that in this case the number of values used increases as the range gets bigger. The placement of nodes with many edges has an influence on the size of the graph. If all nodes representing the same variable are said to have the same number of outgoing edges, then the size of the graph (number of nodes) can be computed using $size = e_0 + e_0 e_1 + e_0 e_1 e_2 + ...$, where e_i is the number of outgoing edges from a node in level i (assuming that there is no sharing). An example of this can be seen on figures 7.1 and 7.2. A nodes have 2 edges, B nodes have 3 edges and C nodes have 4 edges.

As can be seen from the graphs and deducted from simple calculation, the lower in the graph nodes with many edges are, the smaller the graph will be, again assuming no sharing. In our case it is a very fair assumption

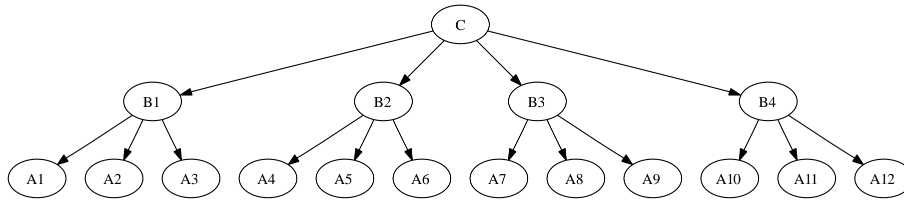


Figure 7.2: An example of a big graph

```
for each variable in order
  for each possible position in order
    compute and remember size of graph
  move the variable to the position which gave the smallest graph
    in case of a tie select the highest position in the order
  go to the next variable
```

Table 7.1: The algorithm we use to rearrange variables

that the nodes with the biggest range has the highest number of edges, and therefore should be placed at the bottom of the graph. However, depending on the level of sharing this might not be the case in LIBIDD. We will test this assumption using a variant of the *sifting algorithm* to test which placement of each variable gives the best result ie. the graph with fewest nodes and edges. This involves testing all the possible positions of the variable. We simply remember the best position and update the order before testing the next variable. We can do this because we do not have to worry about minimizing the number of swaps pr. run. In case of a tie, we will use the position which is nearest the top of the graph. The algorithm we use is as follows:

In section 7.3 we test the best position in the order for each variable using this algorithm. The position in the order we come up with after testing each variable is the best position for that variable assuming that the position for the other variables remain fixed. By doing this for every variable we are able to get closer to an optimal order without having to check every possible permutation of the variables.

By optimal order we mean an order which will produce a graph with the smallest possible number of nodes. By reducing the number of nodes and edges we can reduce the number of logical operations we need to perform and thereby the time it takes to build the graph.

As we have shown, we will get the smallest graph (assuming no sharing) if nodes with the highest number of outgoing edges are located at the lowest level in the graph, nodes with the second highest number of outgoing edges are located at the second lowest level and so on. It goes without saying that variables with the largest domain can result in nodes with the highest number of outgoing edges, although this doesn't have to be the

IP_PROTOCOL		
Position	Nodes	Edges
1	1038	5112
2	1038	5113
3	2036	8271
4	1054	5325
5	1054	5316
6	1044	5295
7	1044	5295
8	1038	5247

Table 7.2: Testing the best position for IP_PROTOCOL

New order
IP_PROTOCOL
IP_FRAGMENT_DATALEN
IP_SADDR
IP_DADDR
TCP_DEST
TCP_SOURCE
UDP_DEST
UDP_SOURCE

Table 7.3: The order after testing IP_PROTOCOL

case. In our case we have domains of three different sizes, 8, 16, and 32 bits. We would expect a variable like IP_PROTOCOL to be positioned at the top of the order as there are (currently) only six different values that this variable can have. We would also expect the variables IP_SADDR and IP_DADDR to be positioned at the bottom of the order as they will have a high number of outgoing edges because of the many different IP addresses used in the filters. The variables representing port numbers (TCP_DEST, TCP_SOURCE, UDP_DEST, UDP_SOURCE) would then be positioned in between.

7.3 Test results

The order before any optimization is done is as shown at the beginning of this chapter. We now test the size of the graph for each variable at each position in the order according to the algorithm described in section 7.2. That is, after testing a variable we move it to the position which gave the smallest graph. The new optimized order is shown next to the test results. We test the variables in the order they were in in the original order. The filter used for these test has a 1000 rules, all with a DENY policy.

Tables 7.2 and 7.4 show that moving either IP_PROTOCOL or IP_FRAGMENT_DATALEN doesn't yield any improvement, it just increases the size of the graph. The best position for these two variables is the one they occupy in the original order.

Moving IP_SADDR to a new position didn't reduce the size of the graph (see table 7.6), but according to our algorithm a variable should be moved to the highest possible postion in case of a tie. Therefore IP_SADDR is moved to position 8.

The results in table 7.8 show that again no improvement is reached but IP_DADDR is moved to position 3 according to the algorithm.

IP_FRAGMENT_DATALEN		
Position	Nodes	Edges
1	1038	5113
2	1038	5112
3	2036	8106
4	1054	5160
5	1051	5151
6	1044	5130
7	1044	5130
8	1038	5112

Table 7.4: Testing the best position for IP_FRAGMENT_DATALEN

New order
IP_PROTOCOL
IP_FRAGMENT_DATALEN
IP_SADDR
IP_DADDR
TCP_DEST
TCP_SOURCE
UDP_DEST
UDP_SOURCE

Table 7.5: The order after testing IP_FRAGMENT_DATALEN

IP_SADDR		
Position	Nodes	Edges
1	3034	11100
2	2036	8106
3	1038	5112
4	1038	5112
5	1944	7830
6	2853	10557
7	2929	10785
8	3005	11013

Table 7.6: Testing the best position for IP_SADDR

New order
IP_PROTOCOL
IP_FRAGMENT_DATALEN
IP_SADDR
IP_DADDR
TCP_DEST
TCP_SOURCE
UDP_DEST
UDP_SOURCE

Table 7.7: The order after testing IP_SADDR

IP_DADDR		
Position	Nodes	Edges
1	3034	11100
2	2036	8106
3	1038	5112
4	1038	5112
5	1944	7830
6	2853	10557
7	2929	10785
8	3005	11013

Table 7.8: Testing the best position for IP_DADDR

New order
IP_PROTOCOL
IP_FRAGMENT_DATALEN
IP_DADDR
IP_SADDR
TCP_DEST
TCP_SOURCE
UDP_DEST
UDP_SOURCE

Table 7.9: The order after testing IP_DADDR

TCP_DEST		
Position	Nodes	Edges
1	1053	5173
2	1044	5130
3	1036	5106
4	1944	7830
5	1038	5112
6	1039	5115
7	1039	5115
8	1039	5115

Table 7.10: Testing the best position for TCP_DEST

New order
IP_PROTOCOL
IP_FRAGMENT_DATALEN
TCP_DEST
IP_DADDR
IP_SADDR
TCP_SOURCE
UDP_DEST
UDP_SOURCE

Table 7.11: The order after testing TCP_DEST

TCP_SOURCE		
Position	Nodes	Edges
1	1053	5171
2	1045	5133
3	1038	5112
4	1039	5115
5	1945	7833
6	1036	5106
7	1036	5106
8	1036	5106

Table 7.12: Testing the best position for TCP_SOURCE

New order
IP_PROTOCOL
IP_FRAGMENT_DATALEN
TCP_DEST
IP_DADDR
IP_SADDR
TCP_SOURCE
UDP_DEST
UDP_SOURCE

Table 7.13: The order after testing TCP_SOURCE

Moving TCP_DEST doesn't change much, however moving it to position 3 does give a minute improvement. The graphs is reduced by two nodes and six edges.

Moving TCP_SOURCE doesn't give any improvements (as seen in table 7.12, and it remains at position 6.

UDP_DEST is moved to position 3 as the gives a reduction of a single edge.

Thus after having testing the optimal position for each variable (assuming that the other variables remain fixed), we have arrived at a new order, which looks like this:

1. IP_PROTOCOL
2. IP_FRAGMENT_DATALEN
3. UDP_SOURCE
4. UDP_DEST

UDP_DEST		
Position	Nodes	Edges
1	1049	5156
2	1042	5123
3	1036	5105
4	1036	5105
5	1112	5334
6	1036	5106
7	1036	5106
8	1036	5106

Table 7.14: Testing the best position for UDP_DEST

New order
IP_PROTOCOL
IP_FRAGMENT_DATALEN
UDP_DEST
TCP_DEST
IP_DADDR
IP_SADDR
TCP_SOURCE
UDP_SOURCE

Table 7.15: The order after testing UDP_DEST

UDP_SOURCE		
Position	Nodes	Edges
1	1049	5156
2	1042	5123
3	1036	5105
4	1036	5105
5	1036	5105
6	1112	5333
7	1036	5105
8	1036	5105

Table 7.16: Testing the best position for UDP_SOURCE

New order
IP_PROTOCOL
IP_FRAGMENT_DATALEN
UDP_SOURCE
UDP_DEST
TCP_DEST
IP_DADDR
IP_SADDR
TCP_SOURCE

Table 7.17: The order after testing UDP_SOURCE

5. TCP_DEST
6. IP_DADDR
7. IP_SADDR
8. TCP_SOURCE

Let us just summarize on the keyresults. In particular let us look at the size of the graph for the original order, the best order and the worst order. The results are shown in table 7.18. Here we can see how the ordering of the variables influences the size of the graph. We have not reduced the size of the graph by much but we can see, that there is a major difference between the worst order and best order found with our algorithm.

We also tried running the algorithm where we chose the lowest possible position in case of a tie. This gave the same size graph but with a somewhat different order:

1. IP_PROTOCOL

	Nodes	Edges
Original order	1038	5112
Best order	1036	5105
Worst order	3034	11100

Table 7.18: Summary of the test results

2. TCP_DEST
3. UDP_DEST
4. IP_SADDR
5. IP_DADDR
6. TCP_SOURCE
7. UDP_SOURCE
8. IP_FRAGMENT_DATALEN

In the beginning of the chapter we stated that we expected the variables with the largest domains to be at the bottom of the graph. If that had been the case we would have got an order that looks like this:

1. IP_PROTOCOL
2. IP_FRAGMENT_DATALEN
3. TCP_DEST
4. TCP_SOURCE
5. UDP_DEST
6. UDP_SOURCE
7. IP_SADDR
8. IP_DADDR

Testing the size of the graph that this order gives us shows that we would get a graph with 1038 nodes and 5113 edges. So apparently there is not much to gain from changing the order of the variables. But when we measure the time it takes to run cfconf with the three different orders we have just presented, we get some interesting results. The results are shown in figure 7.3 where *Order 1* is the order we found with the algorithm, *Order 2* is the order we would have got if we chose the lowest position in case of

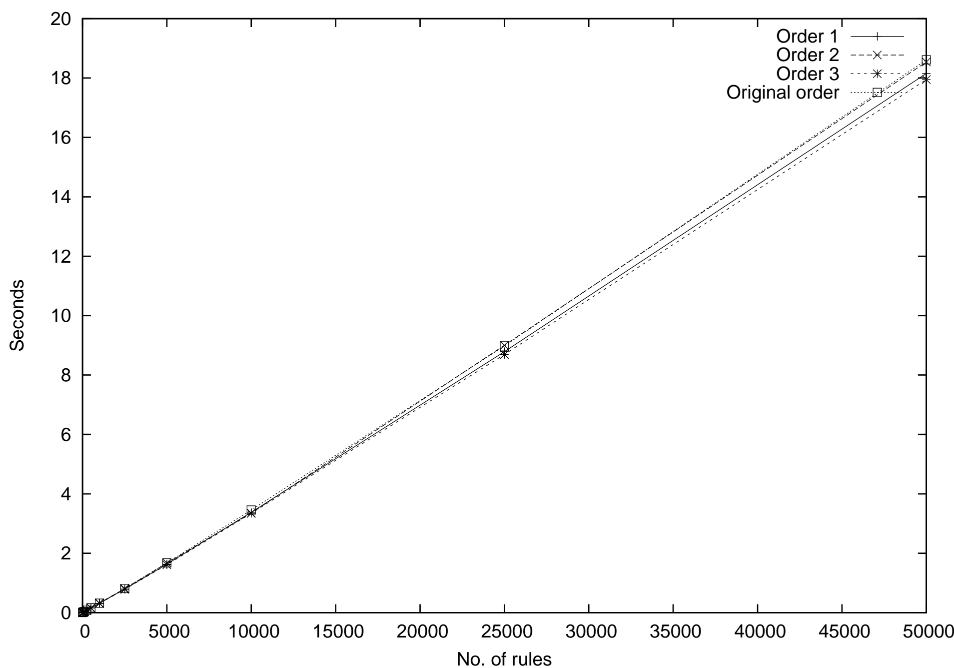


Figure 7.3: Comparing the time for the orders plus the original order

a tie, *Order 3* is the order we originally expected to see and *Original order* is ofcourse the order orginally used in CF.

Order 2 and the original order gave almost identical results and so the line for these two orders look as one. This is the top most line in the graph. Order 1 which is the order found with the algorithm gave a somewhat better result than the original order but the best results are achieved with Order 3. For all orders we have to use filters with many rules to see any noticable changes. But why do we get the results we do. The difference in the size of the graph we get is extremely small but you should consider that the graph is built using many logical operations and a different order can result in some of these operations running slightly faster. Order 3 which resulted in the lowest runtime does not give the smallest graph but this does not matter since the difference in the size of the graph is so small. However, it does raise the point that even though an order does not result in the smallest graph it can result in the lowest runtime. This presents a problem when the goal is to reduce the runtime because how do you determine that order without having to test all possible orders? This is a matter for future work.

7.4 Summary

We have presented an algorithm for finding a better variable order which is based on the *sifting algorithm* described in chapter 3. We have presented the order that we expected would give the best results. We have applied the algorithm to the current order in LIBIDD and come up with a different order. We then tested this order along with the order we expected to give the best results and the order we would get if we had used a slightly different version of the algorithm. The tests showed that size of the graph we get with the different orders is virtually the same. The runtime for *cfconf* does not change much either with a different order, but there is a sligt reduction when using the order we originally expected to see. This indicates that perhaps we should look for an order which gives the shortest runtime instead of one which gives the smallest graph since, apparently, this does not need to be the same.

Chapter 8

Conclusion

In this paper we have presented Compact Filter (CF) which is an alternative firewall for linux. CF represents the rules as a graph instead of one or more lists of rules. However, building the graph takes a long time when the ruleset grows. This project has focused on reducing the time it takes to build the graph.

We make use of the performance analysis tools *gcov* and *gprof* which are described in chapter 3. By adding a few flags when compiling the code, we are able to get information about which functions are very time consuming. We use this knowledge to make suggestions for an optimization of the code.

We focus on the logical operators plus their helper functions and the function `iddDeepFree()`. `iddDeepFree()` used an iterator to discover all the nodes and terminals in an IDD but unfortunatly this does not distinguish between new nodes and ones it has already visited. Therefore it is necessary to check for this whenever a new node is returned by the iterator. We are able to eliminate the use of the iterator and the very time-consuming check of the node returned by the iterator by adding a `marked` bit to the IDD struct. This is used to ensure that we do not run into nodes we have already seen. We also add an operator cache which the logical operators use to save information about operations already performed. Thereby we can avoid performing the same operations twice. Before each logical operation we look in the operator cache to see if we have performed it before. If so we reuse the result instead of performing the operation again. By using the operator cache we have reduced the runtime for the logical operators from exponential to polynomial. Finally the function which computes hashvalues for IDDs has been replaced so it no longer relies on a string representation of the IDD.

We have examined papers on improving the data structure of a decision diagram. An interesting subject is the order of the variables in the decision diagram. Many papers focus on the need to select the order carefully and many algorithms exist to help in selecting the order. We have showed that

the order has a significant influence on the size of the diagram. We have used a variant of the *sifting algorithm* to find the best position for each variable in the order. Test showed that the order found with the algorithm yielded a graph which has virtually the same size as a graph based on the original order in CF. However we did notice that although changing the order does not produce a smaller graph, it can have an effect on the overall runtime.

All in all we have showed that it is possible to reduce the time it takes to build the decision diagram from a ruleset and quite significantly in fact. For a ruleset of 50,000 rules the time it takes to build the decision diagram is reduced from about 9933 seconds (2h45m55s) to about 18 seconds. Also we have gained knowledge that can be useful in other programs that builds decision diagrams.

Chapter 9

Further work

Here we lok at some of the further work that needs to be done or could be done on cfconf in the future.

We have made several improvements to LIBIDD which has helped in making the library much more efficient but in order to fully utilize the improvements it is necessary to implement global IDDs which will make it possible to reuse results from previous calls to the logical operators and not just from within a single call. Furthermore it will also make it unnecessary to return a clone of an IDD when performing AND or OR on a node and a terminal. The node itself may be returned as there is no risk of the operands being deleted afterwards.

CF currently supports 8 fields (actually it is 9 but FRAGMENT and DATALEN are combined into one check in CF) in the IP, TCP and UDP headers. If/when this is extended to include more fields it will be necessary to look at the order again. It may also be necessary to make a tool which automatically selects the order based on which yields the smallest graph in terms of the number of nodes it contains.

We have looked at the ordering of variables as a way of reduzing the size of the graph representing the filter and thereby also the time it takes to build the filter. This is, however, just one way of optimizing the graph. Others may exist, including complement nodes that we described in chapter 6, that need to explored. Although the runtime of cfconf has already been reduced significantly there may still be an advantage to gain after the filter has been sent to the kernel ie. when the CF firewall is up and running. Furthermore, we have not investigated what effects changing the order has had on the time it takes to match a packet against the filter. This could also be interesting to look at.

Appendix A

The original source code

This chapter presents the original source code for some of the functions in LIBIDD.

A.1 iddDeepClone()

```
Idd_t *iddDeepClone(Idd_t *idd_in_root) {
  IddIteratorPreorder_t iddite;
  Idd_t *res_root = NULL; /* the root node of the result */
  Idd_t *new = NULL; /* a new idd node waiting to be assinged */
  IddPartitionEntry_t *cur_p = NULL; /* the current partition  */
  int cur_name;
  int cur_p_size;
  int cur_p_pos; /* p for partition */
  Idd_t *idd = idd_in_root;
  IddPartitionStack_t pstack;

  iddPartitionStackInit(&pstack);
  iddIteratorPreorderInit(&iddite, idd_in_root);

  idd = iddIteratorPreorderNext(&iddite); /* return the root */

  while (idd != NULL) {
    switch (idd->type) {
    case TERMINAL:
      new = iddAlloc();
      iddTerminalInit(new, idd->value.terminal);
      /* assign res to some partition entry */
      if (cur_p != NULL) {
cur_p[cur_p_pos].idd = new;
cur_p_pos++;
      } else {
res_root = new; /* idd_in_root is a terminal */
      }
```

```
      break;
    case NODE:
      if (cur_p != NULL)
iddPartitionStackPush(&pstack, cur_name, cur_p, cur_p_size, cur_p_pos);
      cur_name = idd->value.node.name;
      cur_p = iddPartitionCopy(idd->value.node.partition,
       idd->value.node.partition_size);
      cur_p_pos = 0;
      cur_p_size = idd->value.node.partition_size;
      break;
    default:
      printf("iddDeepClone: unitialized idd detected!\n");
    }
    /* if partition is full then instantiate node and backtrack */
    /* note: by ensuring that res_root==NULL we know that cur_p has been set */
    while ((cur_p_pos == cur_p_size) && (res_root == NULL)) {
      new = iddAlloc();

      iddNodeInit(new, cur_name, cur_p, cur_p_size);
      /* pop from the stack and set pos and size again */
      if (!iddPartitionStackIsEmpty(&pstack)) {
iddPartitionStackPop(&pstack, &cur_name, &cur_p,
     &cur_p_size, &cur_p_pos);
cur_p[cur_p_pos].idd = new;
cur_p_pos++;
      } else { /* if stack is empty then we are done */
res_root = new;
      }
    }
    idd = iddIteratorPreorderNext(&iddite);
  }

  iddIteratorPreorderFree(&iddite);

  if (res_root == NULL) {
    printf("iddDeepClone Failed\n");
  }
  return res_root;
}
```

A.2 iddNot()

```
Idd_t *iddNot(Idd_t *idd) {
  IddHash_t iddhash;
  Idd_t *tmp_idd = NULL;
  Idd_t *res_root = NULL;

  IddIteratorPreorder_t iddite;
```

```
  Idd_t *cur_idd = NULL; /* idd returned from iterator */

  IddPartitionStack_t pstack; /* store partitions that we have not finished */

  Idd_t *new_idd;  /* the idd we are building */
  IddPartitionEntry_t *cur_p = NULL; /* partition we work on currently*/
  int cur_p_size = 0; /* size of the current partition */
  int cur_p_pos = 0; /* current pos in the new idd */
  int cur_name;

  int i;

  iddPartitionStackInit(&pstack);
  iddHashInit(&iddhash, IDDHASHTBLSIZE);
  iddIteratorPreorderInit(&iddite, idd);


  while ((cur_idd = iddIteratorPreorderNext(&iddite)) != NULL) {
    switch (cur_idd->type) {
    case TERMINAL:
      new_idd = iddAlloc();
      if (cur_idd->value.terminal == FALSE)
iddTerminalInit(new_idd, TRUE);
      else
iddTerminalInit(new_idd, FALSE);

      tmp_idd = iddHashFind(&iddhash, new_idd);
      if (tmp_idd == NULL) {
iddHashInsert(&iddhash, new_idd);
      } else {
iddFree(new_idd);
new_idd = tmp_idd;
      }

      if (cur_p != NULL) {
cur_p[cur_p_pos].idd = new_idd;
cur_p_pos++; /* later we check if new_p is full */
      } else {
res_root = new_idd;
      }
      break;
    case NODE:
      if (cur_p != NULL) {
iddPartitionStackPush(&pstack, cur_name, cur_p,
      cur_p_size, cur_p_pos);
      }
      cur_name = cur_idd->value.node.name;
      cur_p_size = cur_idd->value.node.partition_size;
      cur_p_pos = 0;
```

```
      cur_p = iddPartitionAlloc(cur_p_size);
      for (i = 0; i < cur_p_size; i++) { /* copy bounds */
cur_p[i].lower_bound =
  cur_idd->value.node.partition[i].lower_bound;
cur_p[i].upper_bound =
  cur_idd->value.node.partition[i].upper_bound;
cur_p[i].idd = NULL;
      }
      break;
    default:
      printf("iddOptimize FATEL error: uinitialized idd detected!\n");
    }

    while ((cur_p_pos == cur_p_size) && (res_root == NULL)) {
      new_idd = iddAlloc();
      iddNodeInit(new_idd, cur_name, cur_p, cur_p_size);

      /* pop from stack to set cur_* again */
      if (!iddPartitionStackIsEmpty(&pstack)) {
iddPartitionStackPop(&pstack, &cur_name, &cur_p,
     &cur_p_size, &cur_p_pos);
/* optimize a little */
if (new_idd->value.node.partition_size == 1) {
  cur_p[cur_p_pos].idd = new_idd->value.node.partition[0].idd;
  iddPartitionFree(new_idd->value.node.partition);
  iddFree(new_idd);
} else {
  /* check if similar idd exists */
  tmp_idd = iddHashFind(&iddhash, new_idd);
  if (tmp_idd == NULL) {
    iddHashInsert(&iddhash, new_idd);
  } else {
    /* free partition and idd */
    iddPartitionFree(new_idd->value.node.partition);
    iddFree(new_idd); /* well we dont need this no more */
    new_idd = tmp_idd;
  }
  cur_p[cur_p_pos].idd = new_idd;
}
cur_p_pos++;
      } else { /* if stack is empty then we are done */
if (new_idd->value.node.partition_size > 1) {
  res_root = new_idd;
} else {
  res_root = new_idd->value.node.partition[0].idd;
}
      }
    } /* while cur_p_pos */
  } /* while preordernext*/
```

```
  iddIteratorPreorderFree(&iddite);

  if (res_root == NULL) {
    printf("iddNot failed\n");
  }
  return res_root;
}
```

A.3 iddAnd()

```
Idd_t *iddAnd(Idd_t *a_idd, Idd_t *b_idd) {
  int err;
  IddPartitionEntry_t *partition;
  int partition_size;
  Idd_t *result;
  IddPartitionEntryTwo_t *merge_res;
  int merge_res_size, merge_res_used;

  if (a_idd->type == TERMINAL) {
    if (b_idd->type == TERMINAL) {
      /* and the two terminals and return the result */
      result = iddAlloc();
      err = iddTerminalInit(result, iddTermAnd(a_idd->value.terminal,
       b_idd->value.terminal));
      if (err == 0)
printf("iddAnd received: %d from iddTerminalInit\n", err);
    } else {
      /* if a is false then return false term */
      if (a_idd->value.terminal == FALSE) {
result = iddAlloc();
err = iddTerminalInit(result, FALSE);
if (err == 0)
  printf("iddAnd received: %d from iddTerminalInit\n", err);
      } else { /* if a is true then return a copy of b_idd */
result = iddDeepClone(b_idd);
      }
    }
  } else {
    if (b_idd->type == TERMINAL) {
      /* if b is false then return false */
      if (b_idd->value.terminal == FALSE) {
printf("iddAnd: b_idd->value.terminal == FALSE\n");
result = iddAlloc();
err = iddTerminalInit(result, FALSE);
if (err == 0)
  printf("iddAnd received: %d from iddTerminalInit\n", err);
      } else {
```

```
/* if b is true then return copy of a */
result = iddDeepClone(a_idd);
      }
    } else {
      if (a_idd->value.node.name > b_idd->value.node.name) {
/* do the apply left thing */
partition_size = b_idd->value.node.partition_size;
partition = iddPartitionAlloc(partition_size);
err = iddAndApplyLeft(partition, partition_size,
      b_idd->value.node.partition,
      b_idd->value.node.partition_size,
      a_idd);
if (err == 0) {
  printf("iddAnd received: %d from iddAndApplyLeft\n", err);
}
result = iddAlloc();
err = iddNodeInit(result, b_idd->value.node.name,
  partition, partition_size);
if (err == 0)
  printf("iddAnd received: %d from iddNodeInit\n", err);
      } else if (a_idd->value.node.name < b_idd->value.node.name) {
/* do the apply right b thing */
partition_size = a_idd->value.node.partition_size;
partition = iddPartitionAlloc(partition_size);
err = iddAndApplyRight(partition, partition_size,
       a_idd->value.node.partition,
       a_idd->value.node.partition_size,
       b_idd);
if (err == 0)
  printf("iddAnd received: %d from iddAndApplyRight\n", err);
result = iddAlloc();
err = iddNodeInit(result, a_idd->value.node.name,
  partition, partition_size);
if (err == 0)
  printf("iddAnd received: %d from iddNodeInit\n", err);
      } else {
/* do the merge thing */
merge_res_size = a_idd->value.node.partition_size +
  b_idd->value.node.partition_size;
merge_res = iddPartitionTwoAlloc(merge_res_size);
merge_res_used = iddMergeTwo(a_idd->value.node.partition,
     a_idd->value.node.partition_size,
     b_idd->value.node.partition,
     b_idd->value.node.partition_size,
     merge_res, merge_res_size);
if (merge_res_used == 0)
  printf("iddAnd received error %d form iddMergeTwo", merge_res_used);
partition_size = merge_res_used;
partition = iddPartitionAlloc(partition_size);
```

```
err = iddAndMerge(merge_res, merge_res_used,
  partition, partition_size);
if (err == 0)
  printf("iddAnd received error %d from iddAndMerge", err);
iddPartitionTwoFree(merge_res); /* won't need this anymore */
result = iddAlloc();
err = iddNodeInit(result, a_idd->value.node.name,
  partition, partition_size);
if (err == 0)
  printf("iddAnd received: %d from iddNodeInit\n", err);
      }
    }
  }
  return result;
}
```

A.4 iddDeepFree()

```
void iddDeepFree(Idd_t *idd) {
  IddIteratorPostorder_t iddite;
  Idd_t *cur_idd = NULL;
  int i;

  int parray_used = 0;
  int match = 0;

  if (idd == NULL) {
    return;
  }

  iddIteratorPostorderInit(&iddite, idd);
  while ((cur_idd = iddIteratorPostorderNext(&iddite)) != NULL) {
    match = 0;
    for (i = 0; i < parray_used; i++) {
      if (parray[i] == cur_idd) {
        match = 1;
        break;
      }
    }
    if (!match) {
      parray[parray_used] = cur_idd;
      parray_used++;
    }

    if (parray_used >= IDD_PARRAY_SIZE) {
      printf("iddDeepFree out of memory");
      exit(0);
    }
```

```
  }

  iddIteratorPostorderFree(&iddite);

  /* free list of unique idd's */
  for (i = 0; i < parray_used; i++) {
    if (parray[i]->type == NODE)
      iddPartitionFree(parray[i]->value.node.partition);
    iddFree(parray[i]);
  }
}
```

List of Figures

1.1 The posibble position of a firewall 1
1.2 A simple illustration of the decision diagrams used in CF . . 2

2.1 An example of an IDD 7
2.2 An unreduced graph 8
2.3 A reduced graph 8
2.4 IDD A 10
2.5 IDD B 10
2.6 The IDD of A AND B 10
2.7 An example of an IDD representing a packet filter 13

4.1 Comparing runtimes using the old and new hash function . 21
4.2 Step 1 24
4.3 Step 2 24
4.4 Step 3 24
4.5 Step 4 25
4.6 Step 5 25
4.7 Step 6 25
4.8 Step 7 25
4.9 IDD used for iddNot() example 28
4.10 The runtime of cfconf with the original implementation . . . 39
4.11 The runtime of cfconf with the new implementation 40
4.12 Comparing runtimes up to 500 rules 40

6.1 A simple IDD that does not utilize complement nodes 52
6.2 A simple IDD that incorporates complement nodes 53

7.1 An example of a small graph 56
7.2 An example of a big graph 57
7.3 Comparing the time for the orders plus the original order . . 63

List of Tables

3.1 Example of the output generated by gcov 16
3.2 An example of the gprof flat profile 17
3.3 An example of the gprof call graph 18

4.1 The current function for obtaining a hashvalue in LIBIDD . . 21
4.2 The operator cache 22
4.3 The operator cache init function 23
4.4 The operator cache destroy function 23
4.5 The operator cache lookup function 24
4.6 The implementation of iddRegister() 36
4.7 The new implementation of iddDeepFree() 37
4.8 A filter with 10 rules 39

5.1 The gprof flat profile for 500 rules 43
5.2 The gprof flat profile for 1,000 rules 44
5.3 The gprof flat profile for 5,000 rules 44
5.4 The gprof flat profile for 10,000 rules 44
5.5 The gprof flat profile for 25,000 rules 45
5.6 The gprof flat profile for 50,000 rules 45
5.7 The gprof flat profile for 500 rules 46
5.8 The gprof flat profile for 1,000 rules 46
5.9 The gprof flat profile for 5,000 rules 46
5.10 The gprof flat profile for 10,000 rules 47
5.11 The gprof flat profile for 25,000 rules 47
5.12 The gprof flat profile for 50,000 rules 47

6.1 Window Permutation Algorithm example 50
6.2 Sifting Algorithm example 51

7.1 The algorithm we use to rearrange variables 57
7.2 Testing the best position for IP_PROTOCOL 58
7.3 The order after testing IP_PROTOCOL 58
7.4 Testing the best position for IP_FRAGMENT_DATALEN . . 59
7.5 The order after testing IP_FRAGMENT_DATALEN 59
7.6 Testing the best position for IP_SADDR 59
7.7 The order after testing IP_SADDR 59
7.8 Testing the best position for IP_DADDR 59
7.9 The order after testing IP_DADDR 59
7.10 Testing the best position for TCP_DEST 60
7.11 The order after testing TCP_DEST 60
7.12 Testing the best position for TCP_SOURCE 60
7.13 The order after testing TCP_SOURCE 60
7.14 Testing the best position for UDP_DEST 61
7.15 The order after testing UDP_DEST 61
7.16 Testing the best position for UDP_SOURCE 61
7.17 The order after testing UDP_SOURCE 61
7.18 Summary of the test results 62

Bibliography

[And] Henrik Reif Andersen. An introduction to binary decision diagrams.

[Bry] Randal E. Bryant. Graph-based algorithms for boolean function manipulation.

[BUD] The buddy project.

[CF] Mikkel Christiansen and Emmanuel Fleury. Compact filter.

[CF04] Mikkel Christiansen and Emmanual Fleury. An mtidd based firewall - using decision diagrams for packet filtering. 2004.

[ctt] Gcov manual.

[Fre04] Zech Frey. Coverage measurement and profiling. 2004.

[GPR] Gprof manual.

[KMBM] Rohit Kapur Kenneth M. Butler, Don E. Ross and M. Ray Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams.

[KSBB90] Richard L. Rudell Karl S. Brace and Randal E. Bryant. Efficient implementation of a bdd package. 1990.

[MFK] H. Fujisawa M. Fujita and N. Kawato. Evaluation and improvements of boolean comparison methond based on binary decision diagrams.

[NF] The netfilter/iptables project.

[OGM] Shlomi Livne Orna Grumberg and Shaul Markovitch. Learning to order bdd variables in verification.

[PCP93] I. Hajj P. Chung and J. Patel. Efficient variable ordering heuristics for shared robdd. 1993.

[Rud] Richard Rudell. Dymanic variable ordering for ordered binary decision diagrams.

[SMSV] R. Brayton S. Malik, A. Wang and A. Sangiovanni-Vinventelli. Logic verification using binary decision diagrams in a logic synthesis environment.

[SMY90] N. Ishura S. Minato and S. Yajima. Shared binary decision diagrams with attributed edges for efficient boolean function manipulation. 1990.