Pattern Matching Using VFind And The CVDL Language

The Leader In Total Security Solutions For
Linux, UNIX and Mac OS X

- Anti-Virus
- Baseline Control
- Crypto Cracking
- Pattern Analysis
- Computer Forensics
- Operational Monitoring
- Self-Healing and Repair
- Command Line or Web Based

Providing Security Tools Since 1988

Pattern Matching Using VFind

Richard J. Perry† and Peter V. Radatti‡
†Associate Professor, Electrical and Computer Engineering
Villanova University, Villanova, PA 19085
richard.perry@villanova.edu, (610)-519-4969, FAX: (610)-519-4436

‡President, CyberSoft Operating Corporation
1508 Butler Pike, Conshohocken, PA, 19428
radatti@cyber.com, (610)-825-4748, FAX: (610)-825-6785

June 23, 2005

Chapter 1

Introduction

1.1 Pattern Analysis for Computer Security

By Peter Radatti

This introduction (adapted from [1]) is somewhat philosophical in nature but sometimes that is exactly what is needed to convey meaning. It is my belief that after you have read this you will have a different view of pattern analysis. Pattern analysis is not the "end-all" of computer security but it is a very big hammer. Sometimes (often), when you have a big hammer, other methods just don't matter. What works, works!

Most of computer security is about being able to tell the good guys from the bad. This is called pattern analysis. To a certain extent all of us do pattern analysis. When you look at an orange and recognize it as such your brain just did pattern analysis. You brain interpreted what your eyes saw, compared it to known objects, rejected things that were similar but not "it" and arrived at an answer. In fact, you may have been wrong. It may not have been an orange but a tangerine. The ability to tell the difference between oranges and tangerines requires additional parameters that you might not have.

If you think clearly about pattern analysis you will find that it is an overwhelming problem that is usually done wrong. Two examples where pattern analysis was done wrong that changed the course of human history were Pearl Harbor in Hawaii and the World Trade Center in New York City. In both cases, the information necessary to predict and prevent these events existed prior to the event but was lost in the noise. That is because in real life, pattern analysis is a very difficult problem. The volume of data itself makes it difficult and frequently you do not know what patterns to look for until it is too late. Fortunately, this work is about digital pattern analysis where the problems of volume and hindsight are still large but within the realm of the possible.

There are two aspects to volume making problems complex. The first is that a whole lot of stuff needs to be processed. The second is that given large volumes of data, patterns you are looking for will naturally occur as a misidentification. This is actually related to the well known postulate that an infinite number of monkeys typing on an infinite number of typewriters will eventually reproduce the entire works of Shakespeare. It is, however, different because when you are dealing with an infinite number, everything is possible. In computer security everything is finite even if it seems infinite from a practical point of view. An example of this is that many digital photos are 640 by 480 pixels which yield a total of 307,200 pixels. Using 256 colors, you get the number 256^{307200}. This number represents every single photo that has ever been taken or will ever be taken at that size. It contains every picture of every face, building, animal, aliens on distant planets, in fact anything that can be contained in a photo of that size. This number is finite but still so large that it becomes meaningless.

When a pattern exists but is not found, it is called a miss. Pearl Harbor and the World Trade Center were misses. When a pattern is correctly found it is called a hit. An example is when a virus scanner finds a virus. When a pattern is found but it is not correct it is called a false hit. There are several types of false hits, including a special type called a “ghost hit”. An example of a false hit is when a virus scanner detects a virus that is not there. An example of a ghost is when a virus scanner detects a virus that was disinfected. The virus is actually there but moot. There are excellent reasons to want to know when ghosts are present but it is most useful to know they are ghosts.

In 1988 I wrote a program called “ring toss”. I named it that based on the backyard game called quates where you toss a ring onto a stake in the ground. It is basically the game horseshoes but with rubber rings. The object of the program is to find all of the letters in a complex sentence in positional order within a single file. I had an entire server to run the program against. What I found was that many files triggered the ring toss program as a hit. When I examined the files I found that what I was detecting was data below the noise level. In reality, there was no hit. My pattern was complex and long and the amount of data to search was finite. There was no significant data in the detected files but the way I was looking for my pattern gave me the false hits. This brings us to the point that the way you look for a pattern matters.

The problem of volume exists in computer security. The volume of data is much smaller than in the real world but still very large. The issue is that we are looking for patterns that are small. Many computer viruses are only about 60 bytes long! Lexical patterns are rarely more than 300 bytes long. The ratio of the pattern size to the volume of data to be searched is critical to the difference between a hit and a false hit. To avoid false hits, the pattern size and complexity needs to increase when the volume of data to be searched increases. When you are looking for a virus, you are limited by the maximum size of the virus so additional data such as target file identification and pattern location become meaningful.

I call pattern complexity “data richness”. For example, the pattern zero, zero repeated a few hundred times is not data rich. There is no statical significance to the pattern. In fact, this exact pattern will false hit on almost all executables since many executables set aside variable space using the byte value zero.

What I have just said is that pattern analysis is a game of statistics. The pattern to be searched needs to have a significant length and needs to have significant data richness IN RELATION to the target files that need to be searched. In addition, the pattern must be searched for in a way that is appropriate for the problem at hand. Do any of this wrong and you get a false hit.

Another view is that pattern analysis is puzzle solving. There are many different types of parameters that can add data richness beyond the actual pattern. For example, if you know that the specific pattern you are interested in can only validly exist in a file of a specific type, you can increase the data richness of the pattern master by restricting its use to files where the pattern can exist. This also helps to reduce false hits because if a matching pattern does exist in a file of the wrong type it will not be flagged. Positional information can also be of value when enhancing data richness. For example, if a pattern only exists at a specific location in a file, restricting the search for that pattern to where it can exist positionally in the file enhances data richness.

The next part of understanding pattern analysis is that it really exists as two separate problems. The first is to create the master search pattern, "The thing you are looking for." This is not always obvious or easy. The second is to find the master pattern in target files. This is actually a different set of problems which require different skills. An example is that the analyst who detects a virus for the first time and writes the master pattern can be a different person from the person who wrote the program that executes the master pattern. They are almost always different from the people who run the program in the field.

The good news for the mathematically challenged is that you do not need to know statistics to do this well. In fact, certain people with mental problems do this well. In the movie, "A Beautiful Mind" about Dr. John Forbes Nash, Jr. you can see scenes where Dr. Nash sees patterns that don't exist. The problem as told in the movie was that sometimes with schizophrenia it can be hard to tell the difference between a hit and a false hit. (The movie was not a true representation of the problem Dr. Nash and others suffer but is a good example to make this point.) Another type of mental condition that seems to make people good at pattern analysis is Attention Deficit Hyperactivity Disorder, ADHD. Some people with ADHD trained in pattern analysis can see the patterns they are looking for, whereas people without ADHD cannot. In one case, I found an ADHD person with a high IQ who rarely has false hits. They are a natural talent at creating pattern masters.

So far, I have only discussed simple pattern masters where there is a near one to one correspondence between the pattern master and the search pattern. There are additional types of pattern masters which can match valid patterns where the exact pattern is unknown in advance. A simple example of that would be if you were searching for the pattern, "hello" and you knew that some people might spell the word "hello" with the number zero in the letter–O place or the numbers one in the letter–L place. Of course they may use any or every combination of the above. A pattern master for a pattern of that type could be described verbally as:

Search for the letter 'h' while ignoring case.
Search for the letter 'e'
Search for the letter 'l' or the number '1'
Search for the letter 'l' or the number '1'
Search for the letter 'o' or the number '0'

As you can see, this is a very simple little pattern master yet it can detect 16 actual patterns, all of which are valid. Add the ability to ignore white space between letters or search for a prefix and postfix of white space and the data richness is still increased while the number of actual patterns that can be detected increases substantially. I call techniques that deal with this problem as "pattern looseness". Again, while pattern looseness opens the number of patterns you can find with a single master, data richness dictates that you cannot include false hits!

An area where pattern looseness is extremely useful is when you are trying to detect a member of a family of data. Most commonly, this is to detect a new member of a family of computer viruses such as the Bagle virus. It is, of course, extremely valuable to be able to have a pattern master that detects all or most of the members of a family. Imagine when a new strain of the Bagle virus is released and even before it becomes known, it is already detected by a family pattern master that included enough looseness to detect it! Pattern looseness also has applications in the biological sciences and in lexical analysis such as preserving secrets.

Another area of study in pattern analysis is data enhancement through transitional formatting. I believe that I implemented the first transitional pattern analysis engine in VFind in 1999 when we noted that pattern masters of Java byte code were not reliable because of the way the Java system itself worked. The solution was to create a transitional engine that prepared the data for pattern analysis. In the case of the Java byte code, we created a fast, light disassembly system that processed the binary byte code into reliable and easily searched disassembly code. This is backward from a straight pattern search because instead of modifying the pattern master you modify the data being searched. Transitional formatting of data to enhance pattern analysis is a very powerful and effective valid tool that can reveal data when no other tool can. Always remember to look at all sides of the problem!

Another aspect of transitional pattern analysis is the meta–data creation that is the transitional phase of the problem. It is necessary to arrive at a transitional function that will process the raw data into meta–data that by its nature lends itself to pattern analysis. This is not as easy as it seems and sometimes the resulting meta–data can not be searched by a simple pattern master but by an enriched pattern master that includes information about the resulting meta–data. An example of this is when a CPU emulator is invoked to emulate a binary. For the purposes of pattern analysis, this is most often used to detect attack programs such as viruses. The results of emulation may be several useful patterns. The first may be a hexadecimal string of values that have been decrypted or decoded by the emulated code. The second may be a string of events that triggered flags in the emulator. For example, if a program generates a list of target files and then modifies those target files, the setting of the emulator flags are useful information in themselves. In fact, you might generate separate pattern masters for a problem of this type. The first is the decoded string of values. That can be a straight–forward pattern master that is used solely against values provided to it by the emulator (Notice that since the pattern master is only applied against results from the emulator, that data richness is enhanced). The second is a pattern master that detects dangerous results of an emulated program. In both cases, the results of the emulation must be interpreted. That interpretation is also "pattern analysis".

Another method of pattern analysis is straight statistical analysis such as the Bayes method. In Bayes analysis, a database is created. The target is statically compared to the database to see how closely it compares to the database. The higher the statistical probability that the target under examination matches the contents of the Bayes database the better the chances are that the target belongs to the class of targets in the database. The drawback to this is that everything depends upon a properly created database. If the database contains flaws, then the analysis will be flawed. This is why an unsolicited bulk e-mail (UBE) filter will have problems detecting some e-mail as UBE while detecting desired e-mail as such. One of the ways that CyberSoft solved the problem of the comparative database in the Bayes method was to not directly rely upon human population of the database. This solution was implemented in our SafeInternetEmail program and found to be very successful (additional reading can be found on both cybersoft.com [2] and safeinternetemail.com [3]). We create pattern masters with both enough data richness and data looseness to detect a good number of UBE messages. The result is still not good enough to significantly reduce UBE messages from a reader's perspective but it is good enough to populate the Bayes database. The pattern masters and the Bayes method can then be applied against our target file synergistically. Notice that this is another form of transitional data. We created a pattern master which detected patterns which were then used as a method of detection of other patterns. The universe of pattern analysis is only limited by your imagination.

Just as a matter of clarification, I have only discussed our implementation of Bayes pattern analysis to determine if something is UBE or not. While this is the most common use of Bayes, it is not limited to that. It can be used to determine if a target conforms to any type of database. This is why VFind allows you to create multiple Bayes databases and CVDL programs. You could create a database to determine if a target has a high probability of being a terrorist communication, criminal communication or is in a specific foreign language. For example, if you create a database that contains classical literature written in Latin then you would have a way of determining if a target is a classical literature written in Latin. If you dumped a large amount of unspecialized communications written in French, then you would have a way of detecting French. Put in a large collection of love letters and you can detect love letters. The secret is that you can statically determine if something conforms to the Bayes database. What you put in the database is unlimited. I believe that you could even put in the hexadecimal strings that compose virus bodies and statically determine if a target was a virus. There is no practical limit to this technology.

The reader should note that so far I have discussed the use of pattern analysis for the purpose of hostile software detection or UBE (Antivirus professionals are very fussy about the use of the term "computer virus" but the general public uses the term to mean any hostile software program). Pattern analysis is not limited to these areas. You can apply pattern analysis to anything. One of the more interesting uses for pattern analysis is searching stock market results real time looking for trends that can result in a gain for the operator. Pattern analysis can also be used to determine if something is a forgery. In fact, if very large databases of pattern analysis programs were developed, they could be used to analyze and determine even very complex problems such as medical diagnosis or structural analysis. In this view, pattern analysis is a way of capturing expert knowledge in a very specific and highly usable way.

One of the esoteric aspects of pattern analysis is that the tools you use define the pattern masters you can use. Many end users create their own tools to solve single use pattern analysis problems. More sophisticated users may use a language like Perl or Regex but there are significant limitations to these languages, the most important limitation being that there are practical upper limits to what these languages can perform, especially in the number of simultaneous pattern masters that can be solved for. If you only have a small number of pattern masters or a small amount of data to be targeted, then any size hammer will work. On the other hand, if your pattern master is complex, is dependent upon other pattern masters, requires a large number of simultaneous searches, uses Boolean logic or you just have a ton of data to search, then only a really big hammer will solve that problem. I like to think that the pattern analysis system that I designed and have been developing since 1990 is the largest general purpose pattern analysis hammer in the world. It started out life as a UNIX virus scanner but very quickly became a pattern analysis system for use with lexical analysis. It has continued to be enhanced over the years and is now capable of doing pattern analysis in all fields. That tool is called VFind and is part of the VFind Security ToolKit family of products. You can find out more about VFind at cybersoft.com [2].

There is one more aspect of pattern analysis that is often overlooked by users. That aspect is as important as the creation of the pattern master and often goes hand in hand with it. That aspect is what to do with the knowledge created by the pattern analysis. In the event that the knowledge is that the pattern fits a known virus, delete the virus. If it is UBE, do not forward it. If the knowledge is more complex, knowing something does or does not match a pattern is of no value without a plan for action. In the example given earlier, what would have happened if the airport screeners had been able to recognize the terrorists who boarded the planes that crashed into the World Trade Center but did not have a plan or authority to deal with them? My guess is that they would have allowed them to board and the event would still have happened. If, however, they had a plan to detain the terrorists, then the event would not have happened. Sometimes even a simple plan is effective and almost always, any plan is more effective than no plan.

1.2 Overview

This introductory chapter concludes with a note on the examples and some exercises. Accompanying material includes *examples.zip* containing all of the example files, and a CVDL (CyberSoft Virus Description Language) reference card.

Chapter 2 lists the VFind command line options and briefly discusses the CyberSoft VDLs and use of UAD with VFind. Chapter 3 introduces enough basic CVDL to understand Chapter 4 on the VFind scan engines. Chapter 5 is devoted to the CBayes scan engine, and Chapters 6 to 12 form the major portion of this reference on CVDL. Chapter 13 concludes with discussions and examples of performance testing.

1.3 A Note on the Examples

Interactive examples were performed on a Sun/Solaris system using csh with prompt '%'. In csh, '|&' can be used to pipe both standard output and standard error into another program, so many of the examples run VFind like this to limit visible output to just detections:

```
% vfind ... | & grep VIRUS
^           ^^
prompt    pipe stdout and stderr
```

The equivalent command using a Bourne shell is:

```
$ vfind ... 2>&1 | grep VIRUS
^           ^^^^
prompt    merge stderr with stdout
```

1.4 Exercises

1. Check the CyberSoft web site for updates on the VSTK ToolKit and virus definitions.
2. Search on the web for information about the most recent computer virus outbreaks, and create a 10-minute presentation about current threats.

Chapter 2

Using VFind

This chapter lists the current VFind command–line options, and briefly describes the CyberSoft VDLs and use of UAD with VFind.

2.1 VFind Command–Line Options

The following is the current list of VFind command-line options:

Usage: vfind [options] [file ...]
where options are one or more of:

--lock=<value>:
Set lock file for VDL file access. Exclusive locking is done by Lvfind and Lvfind-mt, shared locking by lvfind and lvfind-mt. This feature is scheduled to go away in a future version of VFind.

--lock-retries=<value>:
How many times to retry acquiring the VDL file lock. This feature is scheduled to go away in a future version of VFind.

--lock-sleeptime=<value>:
Number of seconds to delay the reading of the VDL file after acquiring the lock. This feature is scheduled to go away in a future version of VFind.

--exit-on-error|-e:
Exit immediately after encountering any kind of error or warning condition. Normally, VFind prints a warning message and attempts to continue processing after encountering a non-fatal error.

--exit-on-vdl-error|-ev:
Exit immediately if there's any kind of error loading the VDLs. Normally, VFind prints a warning message and attempts to continue processing after encountering a non-fatal error, such as a syntax error in a VDL description.

--vdl-list=<value>:
Read the VDL library list from the specified file instead of from the $VSTK_HOME/data/vfind/vdl.list. Must be the first command-line option if used because it must be processed before other options like --libon= which require the VDL library list file to have already been read. Note that the VDL files specified in the VDL
library list must be in the $VSTK_HOME/data/vfind/ directory.

--copyright|-c:
Display copyright information and then exit. All other options will be ignored.

--help|-h:
Display usage message and then exit. All other options will be ignored.

--version|-v:
Display version information and then exit. All other options will be ignored.

--dup-check|-d:
Check for duplicate VDL names and definitions, and other potential problems. With this option enabled, duplicates will be reported as parser errors. Also, any VDL segment which starts with an offset range or .* operator will be reported as a parser error. An offset at the beginning of a VDL segment is allowed by the CVDL syntax, but does not make sense to use, and may cause the VDL to run very slow.

--emu=<value>:
Set options for polymorphic virus emulation. This option is still under development and its usage will be documented further in a future release.

--emu-help:
List options for polymorphic virus emulation. This option is still under development and its usage will be documented further in a future release.

--emu-config=<value>:
Specify emulation configuration file. This option is still under development and its usage will be documented further in a future release.

--ignore-eof|-i:
Do not close the input stream after encountering EOF, but continue to read file names or SmartScan input.

--jadevdl=<value>:
Load additional virus signatures from file. File contains VDL models for hostile java applets and applications.

--libon=<value>:
Turn on the specified library. VFind will list the available libraries on startup. Some libraries are turned off by default.

--liboff=<value>:
Turn off the specified library. VFind will list the available libraries on startup. Most libraries are turned on by default.

--noload=<value>:
Disable loading the specified VDL. This may be useful if the scanned data contains a lot of false positives for some particular virus. The name of the virus is as it appears in the VDL file, e.g. --noload="W32/Sircam.a".

--noloads=<value>:
Disable loading the VDL listed in the specified file. Each line of the file should contain the name of a virus as specified for the --noload option. Leading and trailing white space is ignored, as are empty lines, and comment lines beginning with '#'.

--notell=<value>:
Turn off reporting the specified virus. This may be useful if the scanned data contains a lot of false positives for some particular virus. The name of the virus is as it is reported by VFind, e.g. --notell="CVDL W32/Sircam.a".

--notells=<value>:
Turn off reporting the viruses listed in the specified file. Each line of the file should contain the name of a virus as specified for the --notell option. Leading and trailing white space is ignored, as are empty lines, and comment lines beginning with '#'.

--pid:
Print process id to stderr.

--savepid=<value>: Save process id to file.

--quiet=<value>:
Suppress some of VFind's verbosity:
0: the default output.
1: suppress the "enter filename" prompt.
2: suppress the "checking file" output.
3: suppress all per-file output, including virus detections.

--rcf=<value>:
Read additional command line arguments from rc file.

--speed=<value>:
Control VFind speed priority:
0: shortest startup time, but slowest scan.
1: medium startup time, medium scan speed.
2: longest startup time, fastest scan speed.

--stdin:
Scan raw data on standard input.

--smart-scan-types|-sst:
Display file types and any VDLs skipped due to file type restrictions.

--no-smart-scan-types|-nosst:
Disable VDL file type restrictions; apply all VDLs to all file types

--threads=<value>:
Ignored when single-threaded.

--unbufl-u:
Use unbuffered reads for SmartScan input use of this option may cause a performance penalty, use it only when required.

--uad=<value>:
Run UAD as a sub-process with the specified command-line opts passed to UAD in addition to -ssw and -s. Thus, UAD will read file names from standard input and write SmartScan output to VFind.
Examples:

```
vfind --uad=
vfind --uad="-M -H3"
```

are equivalent to:

```
uad -s -ssw | vfind -ssr
uad -s -ssw -M -H3 | vfind –ssr
```

The --uad= option is mainly useful for multithreaded vfind-mt which will run a separate UAD process for each thread. In this case, there is no equivalent command using pipes.

Example:
vfind-mt --threads=4 --uad=
will run 4 instances of UAD, with each instance connected to a separate thread, and the file names from standard input will be distributed among the threads running in parallel.

--smartscan-readl-ssr:
Read a SmartScan stream on standard input.

--vexit:
Return a known value on exit.

--vdl=<value>:
Read additional virus descriptions from the specified file.

--vdl0=<value>:
Read additional speed=0 virus descriptions from the specified file.

--vdlc=<value>:
Read additional case insensitive virus descriptions from the specified file.

--vdle=<value>:
Read additional decrypted polymorphic virus descriptions from the specified file. This option is still under development and its usage will be documented in a future release.

--vdlE=<value>:
Read additional Entry point virus descriptions from the specified file. This option is still under development and its usage will be documented in a future release.

--vdlm=<value>:
Read additional meta virus descriptions from the specified file.

--cbayes=<value>:
Read additional cbayes data from the specified file. This option is still under development and its usage will be documented in a future release.

--md5=<value>:
Read additional MD5 signatures from the specified file. This option is still under development and its usage will be documented in a future release.

--vlist:
Print to stdout a list of all viruses VFind scans for.

--#=<value>:
Stop scanning a file after finding so many viruses.

2.2 CyberSoft VDLs

The VFind Security ToolKit, including programs and data files, is installed in a $VSTK HOME directory, usually /opt/vstk/. In VSTK HOME, the subdirectory *data/vfind/* contains the master VDL list file *vdl.list*, and VDL files, described in Chapter 4. The CyberSoft VDL files are written in plain text using the CVDL language, and may be examined as an aid in learning to write your own patterns.

For most of the examples presented here, the VFind *–liboff='*'* option is used, which skips the CyberSoft VDLs in order to focus on user–written VDLs.

2.3 Using UAD with VFind

In practice, for scanning arbitrary files, CyberSoft's *UAD* tool is used in a pipeline with VFind to unpack and/or uncompress archives and other composite files, such as zip or .tar.gz filesor e-mail with encoded attachments. UAD transfers data, file names, and file types to VFind using CyberSoft's SmartScan protocol [4] specified by the uad *-ssw* (smartscan write) and vfind *–ssr* (smartscan read) command–line options, for example:

% uad -ssw somefiles... | vfind -ssr other_vfind_options...

For simplicity, the examples here use standalone VFind, without UAD, but for most kinds of production use of the VSTK ToolKit it is recommended to use both UAD and VFind with SmartScan.

2.4 Exercises

1. Install VFind on your system, or use an existing installation. Use VFind to scan all files on the system, or at least all of those files that are readable for you. Record the number of VDLs reported, number of files scanned, real and CPU run–time used, and peak RAM usage.
2. Examine the VFind CyberSoft VDLs, and create an example file which will match one of the VDLs.

Chapter 3

Basic CVDL

CVDL is the CyberSoft Virus Description Language. It was first defined in January 1992 to provide flexibility in scanning for viruses and general purpose scanning. Since that time, CVDL has undergone many enhancements and continues to actively evolve to meet new scanning needs.

CVDL is used to define patterns for scanning for most of the engines within VFind, i.e. the vdl, vdlc, vdle, vdlE, vdlm, and jadevdl engines. This chapter introduces just enough basic CVDL so the reader will be able to understand the examples in the next chapter on VFind Scan Engines. The rest of CVDL is described in Chapters 6 to 12.

3.1 Pattern Size Limit

Before using CVDL to create patterns, it is important to understand how data is scanned by VFind. First of all, note that there is no limit on the size of the input data being scanned. To accommodate limitless data, VFind uses an input data sliding scan window of size 1 MB (1024*1024 = 1048576 bytes), with an overlap of 64 KB (64*1024 = 65536 bytes) between windows. So after scanning the first MB of input data, the next window of data scanned consists of the last 64 KB from the first window followed by an additional 983040 bytes (1048576 – 65536) of new data.

Patterns matching 65537 bytes or less of data will always work, but patterns matching 65538 or more bytes of data will not work if the matching data overlaps a scan window edge. For example, if 65538 bytes of data are to be matched and the first 65537 of those bytes occur at the end of one scan window, it will not match in that window since it is missing the last byte of the data; the next window will start with the last 65537 bytes of the data and will also not match since it is missing the first byte of the data.

3.2 VDL Format

A VDL is a named set of CVDL patterns. VDLs start with a colon character, followed by the VDL name, comma, definition, and terminating # character:

```
: name , definition # ; comment
```

The VDL *name* can consist of any characters, including spaces, but cannot contain a comma, since the comma is used to separate the *name* from the *definition*. The VDL can extend over multiple lines, and white space is ignored except in the *name* and literal strings.

The VDL *definition* consists of a sequence of *patterns* and *operators* which can span multiple lines. The *definition* is terminated by the # character. The semicolon (;) character is used to specify a comment which extends to the end of the line.

The simplest type of *patterns* are literal strings, e.g. “abc”, and the simplest type of *operators* are concatenation and offsets. These are described in the following sections.

3.3 Strings

A CVDL string pattern is a sequence of characters and *escape sequences* enclosed in “double quotes”.

An *escape sequence* is \c or a *hex escape sequence*. If character c is n, r, or t this represents new line, carriage–return, or tab, respectively; otherwise, it represents character c itself. A *hex escape sequence* is \x or \X followed by two hex digits.

Prefixing a string with a tilde (~) specifies case–insensitive matching, e.g. ~“abc” will match “abc”, “ABC”, “AbC”, etc. Inside of a string, lines ending with a backslash (“\”) specify continuation on the next line.

VDL string examples:

```
:v1, "abc\"efg" # ; matches: abc"efg
:v2, "abc\x22efg" # ; same as v1
:v3, "abc\x22\
efg" # ; same as v1
```

3.4 Concatenation and Offsets

VDL pattern elements are concatenated into larger patterns by using a comma, for example:
"abc", "def"

is the same as:
"abcdef"

In a VDL definition the comma operator means *followed by*.
An offset range for concatenation can be specified using the @ operator, for example:
"abc", @0-10, "def"

will match ”abc” followed by ”def” at an offset of anywhere from 0 to 10 bytes from the end of ”abc”. So it will match ”abcdef”, ”abcXdef”, ..., ”abcXXXXXXXXXXdef”, where X represents any byte.

In general, an offset range is specified as @*start–end*, where *start* or *end* may be omitted to use the defaults of 0 and 32767. So @–10 is the same as @0–10, and @20000– is the same as @20000–32767. If only one value is specified with no hyphen, i.e. @*val*, that is the same as @*val–val*, which is a fixed offset instead of a range.

Note that an offset range only makes sense if preceded by a pattern to offset from. It does not make sense to specify an offset range at the beginning of a VDL segment because there is nothing to offset from. The beginning of a VDL is automatically checked at all offsets from the beginning of the scanned data. The VFind *-d, –dup-check* option (see Section 2.1), which checks for duplicate VDL names and definitions, also checks for offsets at the beginning of a VDL segment, and reports those as parser errors.

Offset ranges are useful for matching data which contains fixed patterns separated by variable offsets. For example, here are "unsubscribe" sentences from four different spam e-mail messages:

```
% cd examples/vdl/spam
% foreach i (1 2 3 4)
? echo s$i
? cat s$i
? end
s1
If you would like to unsubscribe from receiving further
s2
You can unsubscribe anytime if you want.
s3
Or, you may unsubscribe via postal mail by printing
s4
If you wish to unsubscribe from future mailings
%
```

If we decide that detecting the word "you", case–insensitive, followed closely by "unsubscribe" may be a good way to match such sentences, we could do so using the following VDL:

```
% cat s.vdl
:SPAM unsub, ~"you", @-20, "unsubscribe" #
% vfind --liboff='*' --vdl=s.vdl s{1,2,3,4} |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "s1"
##==>>>> VIRUS ID: CVDL SPAM unsub
##==>>>> VIRUS END OFFSET: 32, matched: to unsubscribe
##==>>>> VIRUS POSSIBLE IN FILE: "s2"
##==>>>> VIRUS ID: CVDL SPAM unsub
##==>>>> VIRUS END OFFSET: 19, matched: can unsubscribe
##==>>>> VIRUS POSSIBLE IN FILE: "s3"
##==>>>> VIRUS ID: CVDL SPAM unsub
##==>>>> VIRUS END OFFSET: 23, matched: may unsubscribe
##==>>>> VIRUS POSSIBLE IN FILE: "s4"
##==>>>> VIRUS ID: CVDL SPAM unsub
##==>>>> VIRUS END OFFSET: 26, matched: to unsubscribe
%
```

Offset ranges can be used to match executable code fragments irrespective of variable data values. For example, the disassembly shown in Section 4.3.2 includes the following code:

```
emu: 00004f23 bf474f mov DI 0x4f47 ; DI=0x0 ; \xbfGO
emu: 00004f26 b86b1b mov AX 0x1b6b ; AX=0x0 ; \xb8k\x1b
emu: 00004f29 90 nop ; ; \x90
emu: 00004f2a fc cld ; ; \xfc
emu: 00004f2b b9b504 mov CX 0x4b5 ; CX=0x0 ; \xb9\xb5\x04
emu: 00004f2e 90 nop ; ; \x90
emu: 00004f2f 2bda sub BX DX ; BX=0xffff DX=0x0 ; +\xda
emu: 00004f31 2bd8 sub BX AX ; BX=0xffff AX=0x1b6b ; +\xd8
emu: 00004f33 33d9 xor BX CX ; BX=0xe494 CX=0x4b5 ; 3\xd9
emu: 00004f35 310d xor DS:[DI] CX ; DS=0x0 DI=0x4f47 CX=0x4b5 ; 1\x0d
emu: 00004f37 3105 xor DS:[DI] AX ; DS=0x0 DI=0x4f47 AX=0x1b6b ; 1\x05
emu: 00004f39 f8 clc ; ; \xf8
emu: 00004f3a 43 inc BX ; BX=0xe021 ; C
```

This is part of a polymorphic virus decryption engine, and each time the virus replicates this engine code is copied with different initial values for the DI and AX registers. If we simply take the byte values from column three of the disassembly:
bf474fb86b1b90fcb9b504902bda2bd833d9310d3105f843

and create a VDL:

```
:vpoly,"\xbf\x47\x4f\xb8\x6b\x1b\x90\xfc\xb9\xb5\x04\x90\x2b\
\xda\x2b\xd8\x33\xd9\x31\x0d\x31\x05\xf8\x43"#
```

That will match this particular sample, but will not match other copies of the virus which use different initial DI and AX values. But using offsets, we can modify the VDL as follows so that it will match irrespective of the initial DI and AX values:

```
:vpoly,"\xbf",@2,\xb8",@2,\x90\xfc\xb9\xb5\x04\x90\x2b\
\xda\x2b\xd8\x33\xd9\x31\x0d\x31\x05\xf8\x43"#
```

3.5 Entropy and Serial Correlation

To avoid false hits on clean data, VDL patterns should be taken from “data rich” sections of the samples to be matched. “Data rich” can be defined as patterns which are unlikely to appear randomly even in compressed data. These are generally patterns which have high entropy and low serial correlation.

Entropy is a measure of information content, with units of bits-per-byte as used here, so it ranges from a minimum of 0 to a maximum of 8. Entropy is computed from the probabilities of occurrence of each possible byte value 0...255:

$$e = \sum_{i=0}^{255} pi \log_2 (1/pi) \qquad (3.1)$$

Where the probabilities p_i are estimated using frequency of occurrence of each byte.

It is possible for a highly correlated data sequence to have high entropy, even though the data is not very randomly distributed. Thus, in conjunction with the entropy calculation, the serial correlation coefficient should also be checked.

The serial correlation coefficient is a measure of the amount that sequential data is related. A correlation coefficient always lies between –1 and +1. When it is zero or very small, it indicates that the data values are (relatively speaking) independent of each other. But when the correlation coefficient is close to ±1 it indicates linear dependence between data values (Knuth [5] page 70).

For n data samples $U_0, \ldots, U_{n-1}$ the serial correlation coefficient is defined as:

$$s = \frac{n\,(U_0U_1 + U_1U_2 + \ldots + U_{n-2}U_{n-1} + U_{n-1}U_0\,) - S_1^2}{nS_2 - S_1^2} \qquad (3.2)$$

where S_1 and S_2 are defined as:

$$S_1 = U_0 + U_1 + \cdots + U_{n-1}$$
$$S_2 + U_0^2 + U_1^2 + \cdots + U_{n-1}^2$$

The sample entropy and serial correlation coefficient should be computed over a sliding window through the sample data, and the window with maximum ($e - 8|s|$) selected to produce a VDL string pattern. The factor of 8 simply scales the absolute serial correlation coefficient value to be in the same range as entropy to balance the maximization.

Three simple example files were created for testing entropy and serial correlation calculations:

```
% cd examples/vdl
% cat mkdata.sh
#! /bin/sh
dd ibs=1 count=100 < /dev/zero > zeros.dat
dd ibs=1 count=100 < /dev/random > random.dat
gzip < /var/log/syslog | dd ibs=1 skip=100 count=100 > gzip.dat
% perl es.pl < zeros.dat
e = 0, |s| = 1, e-8|s| = -8
% perl es.pl < random.dat
e = 6.4, |s| = 0.0478, e-8|s| = 6.02
% perl es.pl < gzip.dat
e = 6.22, |s| = 0.176, e-8|s| = 4.81
%
```

The example file *zeros.dat* contains all 0 bytes, so it has very low entropy and high correlation (the *es.pl* Perl script used to compute e, $|s|$, and ($e - 8|s|$) is shown below). In contrast, the random data has very high entropy and low correlation, and the gzip/compressed data has high entropy but more correlation than the random data.

Now suppose we need to create a VDL pattern to match the file *sleep.exe* from the *examples/vdlE/* directory. Evaluating the entropy and correlation over a window of 100 bytes at a few different file offsets yields:

```
% sh
$ for i in 0 1000 2000 10000
> do
> x=‘dd if=../vdlE/sleep.exe ibs=1 skip=$i count=100 2>/dev/null | perl es.pl‘
> echo "offset=$i $x"
> done
offset=0 e = 4.96, |s| = 0.064, e-8|s| = 4.45
offset=1000 e = 5.16, |s| = 0.233, e-8|s| = 3.3
offset=2000 e = 4.43, |s| = 0.139, e-8|s| = 3.32
offset=10000 e = 5.09, |s| = 0.172, e-8|s| = 3.72
$
```

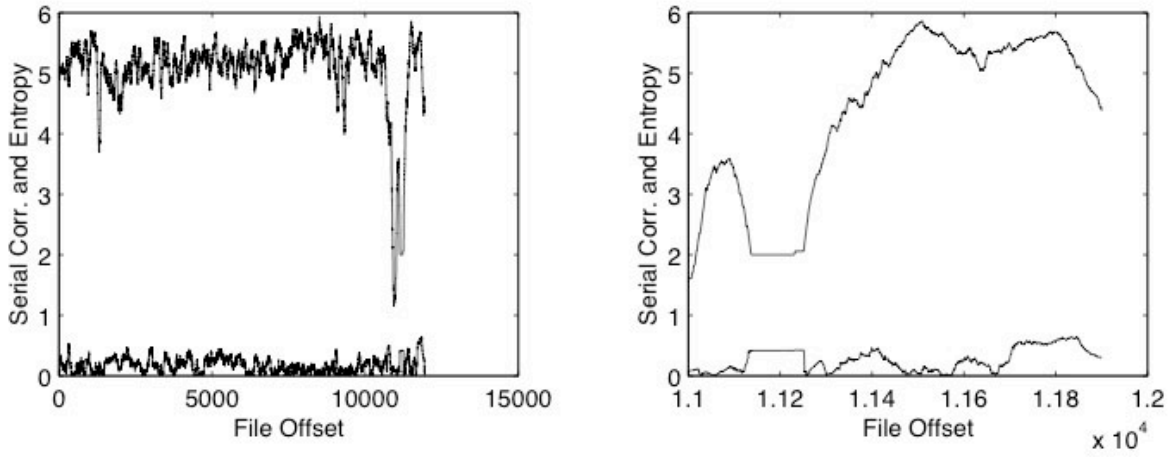

Figure 3.1: Serial Correlation and Entropy vs. File Offset – 100 byte sliding window

So we may decide to use the offset=0 data, since that had the largest (e - 8$|s|$) value; however, a more detailed analysis, illustrated in Figure 3.1, would consider all possible file offsets. The left side of the figure plots the serial correlation and entropy using a window of 100 bytes sliding over the whole file. The right side is zoomed in starting at file offset 11000. Further analysis reveals that the maximum value of (e - 8$|s|$) occurs at file offset 11506, where the values are:

```
% dd if=../vdlE/sleep.exe ibs=1 skip=11506 count=100 | perl es.pl
e = 5.86, |s| = 0.0166, e-8|s| = 5.73
```

We can extract that file data in hex:

```
% dd if=../vdlE/sleep.exe ibs=1 skip=11506 count=100 | od -tx1
0000000 c0 bf ff ff 2d 00 10 73 0a f7 d8 b1 04 d3 e0 2b
0000020 f8 33 c0 8e c0 fd 8a 5c fd 8a 3c 8b 6c fe 8a d7
0000040 8a c3 8b cd 83 ee 03 8b de f6 c2 01 75 03 2b d9
0000060 43 4b 8b 6f fe 8a 67 fd 8a 3f 86 dc f6 c2 01 74
0000100 07 4e e3 06 f3 aa eb 02 f3 a4 84 d2 78 38 b1 04
0000120 8b c7 f7 d0 d3 e8 81 cf f0 ff 8c c2 2b d0 73 08
0000140 f7 da d3 e2
```

and create the corresponding VDL pattern:

```
:sleep.exe,
"\xc0\xbf\xff\xff\x2d\x00\x10\x73\x0a\xf7\xd8\xb1\x04\xd3\xe0\x2b\xf8\x33\
\xc0\x8e\xc0\xfd\x8a\x5c\xfd\x8a\x3c\x8b\x6c\xfe\x8a\xd7\x8a\xc3\x8b\xcd\
\x83\xee\x03\x8b\xde\xf6\xc2\x01\x75\x03\x2b\xd9\x43\x4b\x8b\x6f\xfe\x8a\
\x67\xfd\x8a\x3f\x86\xdc\xf6\xc2\x01\x74\x07\x4e\xe3\x06\xf3\xaa\xeb\x02\
\xf3\xa4\x84\xd2\x78\x38\xb1\x04\x8b\xc7\xf7\xd0\xd3\xe8\x81\xcf\xf0\xff\
\x8c\xc2\x2b\xd0\x73\x08\xf7\xda\xd3\xe2"#
```

Following is the perl script used to compute entropy and serial correlation:

```
% cat es.pl
undef $/; @b = split("",<STDIN>); $len = 1+$#b;
# compute entropy $e
for( $i = 0; $i < 256; ++$i) { $x[$i] = 0; }
for( $i = 0; $i < $len; ++$i) { $c = ord($b[$i]); ++$x[$c]; }
$e = 0; $l2 = log(2);
for( $i = 0; $i < 256; ++$i) { if( $x[$i] > 0) {
$p = $x[$i]/$len; $e += $p * log(1/$p)/$l2; } }

# compute serial correlation coefficient $s
$d = ord($b[0]);
$s1 = ord($b[$len-1]); $s2 = $s1*$s1; $s12 = $s1*$d;
for( $i = 1; $i < $len; ++$i) { $c = $d; $d = ord($b[$i]);
$s1 += $c; $s2 += $c*$c; $s12 += $c*$d; }
$s1 *= $s1; $d = $len * $s2 - $s1;
if( $d == 0.0) { $s = 1.0; } else { $s = ($len * $s12 - $s1)/$d; }
printf "e = %.3g, |s| = %.3g, e-8|s| = %.3g\n", $e, abs($s), $e-8*abs($s);
```

3.6 Exercises

1. Write a program to compute e, $/s/$, and (e-8$/s/$) over an entire file using a sliding window of adjustable size, find the file offset to maximize (e-8$/s/$), and output the corresponding VDL.
2. Create a VDL to match some executable file on your system, then check all executable files for false hits.

Chapter 4

VFind Scan Engines

VFind scans data using several different engines which implement pattern matching techniques appropriate for different kinds of data. This chapter describes each engine in detail with examples, except for the *cbayes* engine which is described separately in Chapter 5. It is important to understand the VFind engines because VDLs and other patterns you may want to match must be designed for use with a specific engine.

The engines are:
vdl - The most general and basic scan engine. This engine can be used to scan any kind of data.

vdlc - A case–insensitive scan engine which should be used only for scanning text.

vdle - An emulator–based engine for Microsoft executables. For encrypted polymorphic viruses, this engine can decrypt using emulation, then scan the decrypted code.

vdlE - An Entry–point engine for Microsoft executables, which skips file headers and data segments and scans starting at the beginning of the executable code.

vdlm - A *meta* engine which detects patterns in the results of the other engines rather than in the data.

jadevdl - A Java disassembler engine which scans the results of Java class file disassemblies rather than the original data.

md5/cit - An MD5–hash–based engine for efficient whole–file matches using hash databases.

cbayes - A scan engine based on Bayesian probability classification.

The engines are configured with pattern files specified in *VSTK HOME/data/vfind/vdl.list* which contains entries specifying pattern file name, engine, enable/disable, and description. Except for *CIT* engine pattern files, which reside in the *VSTK HOME/data/cit/* directory, all of the pattern files reside in the *VSTK HOME/data/vfind/* directory.

Example *vdl.list* file:

exe	vdl	enable	MicroSoft Executable
ole	vdl	enable	MicroSoft OLE
ctext	vdlc	disable	Case--insensitive Text
dpoly	vdle	disable	Decrypted Polymorphs
Entry	vdlE	enable	Entry--Point Scalpel
jade	jadevdl	enable	Java Classes
meta	vdlm	enable	Meta VDLs
notell	vdl	enable	notell VDLs
danger	cit	enable	CIT Danger Files
spam	cbayes	enable	SPAM

Each engine also has a command–line option to specify additional user–defined pattern files.

–vdl=file Read additional virus descriptions from the specified file.

–vdlc=file Read additional case insensitive virus descriptions from the specified file.

–vdle=file Read additional decrypted polymorphic virus descriptions from the specified file.

–vdlE=file Read additional Entry point virus descriptions from the specified file.

–vdlm=file Read additional meta virus descriptions from the specified file.

–jadevdl=file Load additional virus signatures from file. File contains VDL models for hostile java applets and applications.

–md5=file Read additional MD5 signatures from the specified file.

–cbayes=file Read additional cbayes data from the specified file.

4.1 vdl Engine

For general purpose scanning for binary patterns in arbitrary file types, the *vdl* engine should be used. Some attention should be given to constructing patterns in parts or using hex notation to avoid false hits on the pattern file itself.

For example, if the pattern to be detected is “abcdefg”, we could create VDL *ag1* in file ag.vdl:
:ag1, "abcdefg" #

however, VFind would report a match when scanning the ag.vdl file itself. To avoid such matches, we could either split the pattern into two parts as in VDL *ag2*, or write part of it in hex or decimal as in *ag3*, and *ag4* :

```
% cd examples/vdl
% cat x.txt
abc
abcd
----

ZZabcdefghij
% cat ag.vdl
:ag2, "abc", "defg" #
:ag3, "abc\x64efg" #
:ag4, "abc", 100, "efg" #
% vfind --liboff='*' --vdl=ag.vdl x.txt ag.vdl |& egrep 'Checking file|VIRUS'
##==> Checking file: "x.txt"
##==>>>> VIRUS POSSIBLE IN FILE: "x.txt"
##==>>>> VIRUS ID: CVDL ag4
##==>>>> VIRUS END OFFSET: 23, matched: \x0a----\x0aZZabcdefg
##==>>>> VIRUS POSSIBLE IN FILE: "x.txt"
##==>>>> VIRUS ID: CVDL ag3
##==>>>> VIRUS END OFFSET: 23, matched: \x0a----\x0aZZabcdefg
##==>>>> VIRUS POSSIBLE IN FILE: "x.txt"
##==>>>> VIRUS ID: CVDL ag2
##==>>>> VIRUS END OFFSET: 23, matched: \x0a----\x0aZZabcdefg
##==> Checking file: "ag.vdl"
```

Another example is the pattern provided by EICAR [6] for use in testing anti-virus scanners:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}
$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

The pattern should be specified all on one line with no blanks or new line characters, but it is split over two lines above to avoid detection if this report is scanned. The EICAR printable string is actually a legitimate MS/DOS .COM program, and when run it prints the message *EICAR STANDARD-ANTIVIRUS-TEST-FILE!*

Detection of the EICAR test pattern includes additional constraints which we will consider later in Section 6.7. Here we use a shell script to create a VDL which will match the pattern anywhere in a file:

```
% cat eicar.sh
#! /bin/sh
# create eicar hex vdl from eicar.com
echo ':EICAR,"\c'
perl -e 'undef $/; foreach $b (split("",<STDIN>)){ printf("\\x%02x",ord($b));}'
echo '"#'
% ./eicar.sh < eicar.com > eicar.vdl
% fold eicar.vdl
:EICAR,"\x58\x35\x4f\x21\x50\x25\x40\x41\x50\x5b\x34\x5c\x50\x5a\x58\x35\x34\x28
\x50\x5e\x29\x37\x43\x43\x29\x37\x7d\x24\x45\x49\x43\x41\x52\x2d\x53\x54\x41\x4e
\x44\x41\x52\x44\x2d\x41\x4e\x54\x49\x56\x49\x52\x55\x53\x2d\x54\x45\x53\x54\x2d
\x46\x49\x4c\x45\x21\x24\x48\x2b\x48\x2a"#
% vfind --liboff='*' --vdl=eicar.vdl eicar.com |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "eicar.com"
##==>>>> VIRUS ID: CVDL EICAR
##==>>>> VIRUS END OFFSET: 68, matched: TEST-FILE!$H+H*
%
```

For a pure binary example, suppose we have used an analysis of entropy and/or serial correlation (see Section 3.5) or other means to identify that offset lines 9 and 10 of an octal dump of the file *binary.dat* would produce a good VDL. We can create the VDL semi–automatically using a small shell script:

```
% cat binary.sh
#! /bin/sh
# create vdl from octal dump input
echo ':vbin,"\c'
cut -c8- | sed 's/ /\\x/g' | tr -d '\n'
echo '"#'
% od -tx1 binary.dat | head -10 | tail -2
0000200 3c 7e 81 8d c2 8a bb 3e de ae 97 fd 6f a5 3c 20
0000220 c0 58 c7 35 22 c5 19 f4 6c 60 91 b2 77 44 6c a2
% od -tx1 binary.dat | head -10 | tail -2 | ./binary.sh > binary.vdl
% fold binary.vdl
:vbin,"\x3c\x7e\x81\x8d\xc2\x8a\xbb\x3e\xde\xae\x97\xfd\x6f\xa5\x3c\x20\xc0\x58\
xc7\x35\x22\xc5\x19\xf4\x6c\x60\x91\xb2\x77\x44\x6c\xa2"#
% vfind --liboff='*' --vdl=binary.vdl binary.dat |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "binary.dat"
##==>>>> VIRUS ID: CVDL vbin
##==>>>> VIRUS END OFFSET: 160, matched: X\xc75"\xc5\x19\xf4l‘\x91\xb2wDl\xa2
%
```

4.2 vdlc Engine

As will be discussed in Section 13.2, VFind uses a parallel search engine design, so scanning runtime is mostly independent of the number of VDL rules. However, the general *vdl* parallel search engine does not handle case–insensitive (Section 3.3) or wildcard white space (Section 7.5) patterns, so VDLs containing such patterns generally run slow. The *vdlc* engine was created specifically to provide fast/parallel run–time for these patterns.

Note that use of the *vdlc* engine only affects the scanning speed and does not affect the accuracy of pattern detection. VDL patterns are always checked completely and serially if triggered by a parallel search engine, to avoid false detections. This means, for example, the following VDL:

```
:mixy, "Hello" and ~"goodbye" #
```

will only match text which contains “Hello” (case–sensitive), and “goodbye” (case–insensitive, so “GoodBye” and “GOODbYe” also match) regardless of whether the *vdl* or *vdlc* engine is used, as can be seen in the sample run below: file t2.txt is matched but t1.txt is not; however, scanning will be faster for this VDL if used with the *vdlc* engine.

```
% cd examples/vdlc
% cat mixy.vdl
:mixy, "Hello" and ~"goodbye" #
% cat t1.txt
abc heLLO GoodBYE
% cat t2.txt
abc Hello GoodBYE
% vfind --liboff='*' --vdl=mixy.vdl t1.txt t2.txt |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "t2.txt"
##==>>>> VIRUS ID: CVDL mixy
##==>>>> VIRUS END OFFSET: 17, matched: c Hello GoodBYE
% vfind --liboff='*' --vdlc=mixy.vdl t1.txt t2.txt |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "t2.txt"
##==>>>> VIRUS ID: CVDL mixy
##==>>>> VIRUS END OFFSET: 17, matched: c Hello GoodBYE
%
```

4.3 vdle Engine

The *vlde* scan engine is an emulator–based engine for Microsoft executables. For encrypted polymorphic viruses, this engine can decrypt using emulation, then apply VDLs to scan the decrypted code.

When the emulator activates on an executable, one of the following informative messages is displayed, depending on the executable file type:
##==> Emulating .COM: filename
##==> Emulating EXE: filename
##==> Emulating Win32 PE: filename

Additional results may then be displayed depending on the emulator options selected. In particular, the emulator scans the executable for pairs of obfuscated instructions which are typically found in certain polymorphic viruses, and may report virus *GenPoly.1* (for Generic Polymorph Type 1) if enough pairs are found.

4.3.1 Emulation Options

In addition to the *–vdle=file* option to read VDLs, the following are VFind command–line options specific to the emulator engine:
–emu=value Set options for polymorphic virus emulation.
–emu-help List options for polymorphic virus emulation.
–emu-config=file Specify emulation configuration file.

The *–emu-help* option lists the bit–map of values available for the *–emu=value* option:

emu mode bits:

0000 0000 0000 0001	show disassembly
0000 0000 0000 0010	enable JMP instructions
0000 0000 0000 0100	enable JCC instructions
0000 0000 0000 1000	enable LOOP instructions
0000 0000 0001 0000	enable CALL instructions
0000 0000 0010 0000	enable MEM modification
0000 0000 0100 0000	write decrypted, Code, and/or Entry
0000 0000 1000 0000	disable .COM emulation
0000 0001 0000 0000	enable unknown type emulation
0000 0010 0000 0000	enable bad instruction pair counter
0000 0100 0000 0000	enable bad instruction pair reporting

For example, with option *–emu=11* the emulator will display a disassembly of any MicroSoft executables being scanned, and will execute JMP instructions during emulation. To disable the emulator engine, use *–emu=0*.

The *–emu=value* option is a quick way to set a subset of possible emulator options on the command–line. The complete set of options may be controlled by specifying an emulation configuration file using the *–emu-config=file* option. Following is a sample configuration showing default values for all options:

```
# bad_count_max - maximum number of bad/invalid instructions
# to tolerate before stopping emulation.
#
bad_count_max 10
#
# do_call - enable (1) or disable (0) emulation of CALL instructions.
#
do_call 0
#
# dont_do_com - enable (0) or disable (1) emulation of .COM files.
#
dont_do_com 0
#
# do_unknown - enable (1) or disable (0) emulation (as .COM) of unknown file types.
#
do_unknown 0
#
# do_jcc - enable (1) or disable (0) emulation of conditional
# jump instructions.
#
do_jcc 0
#
# do_jmp - enable (1) or disable (0) emulation of unconditional
# jump instructions.
#
do_jmp 1
#
# do_loop - enable (1) or disable (0) emulation of LOOP instructions.
#
do_loop 0
#
# do_mem - enable (1) or disable (0) emulation of instructions which
# modify contents of memory.
#
do_mem 0
#
# do_write - enable (1) or disable (0) writing of a decrypted
# polymorphic code segment to a file named "decrypted",
# start of code segment to a file named "Code",
# and/or Entry-point scalpel code segment to a file named "Entry".
#
```

```
do_write 0
#
# op_count_max - maximum number of instructions to emulate.
#
op_count_max 2000
#
# pair_count_enable - enable (1) or disable (0) detection of obfuscated
# decryption engine instructions and instruction pairs (i.e. GenPoly.1).
#
pair_count_enable 1
#
# pair_count_max - maximum number of obfuscated instruction pairs to
# tolerate before declaring a hit for GenPoly.1.
#
pair_count_max 2
#
# pair_count_range - maximum number of instructions to emulate when
# looking for GenPoly.1.
#
pair_count_range 50
#
# pair_count_report - enable (1) or disable (0) reporting of individual
# obfuscated instruction pairs.
#
pair_count_report 0
#
# show_disassem - enable (1) or disable (0) disassembly listing during
# emulation.
#
show_disassem 0
#
# Entry_jmp_limit - maximum number of instructions emulated during which
# the presence of a jmp instruction will set the Entry-Point engine scan
# starting point to the destination of the jmp.
#
Entry_jmp_limit 10
#
# Entry_len_limit - maximum number of bytes to scan for the Entry-Point
# engine.
#
Entry_len_limit 2000
#
# Code_len_limit - maximum number of bytes to write if do_write is enabled
# and code segment is written.
#
Code_len_limit 2000
```

4.3.2 Emulation Example

This example first demonstrates running the emulator engine on a virus sample which performs a decryption loop. Note that emulation starts at the entry point, 0x100 for .COM, and the initial "jmp" instruction is followed. Current emulated register values are displayed in the comments after the opcode and operands:

```
% cd examples/vdle
% cat config1
do_call 1
do_jcc 1
do_jmp 1
do_loop 1
do_mem 1
do_write 1
op_count_max 30000
show_disassem 1
% vfind --liboff='*' --emu-config=config1 vpoly.com
...
##==> Emulating .COM: "vpoly.com"
emu: JMP enabled.
emu: JCC enabled.
emu: CALL enabled.
emu: LOOP enabled.
emu: MEM enabled.
emu: pair count enabled.
emu: 00000100 e91d4e jmp 0x4e1d ; 0x4f20 ; \xe9\x1dN
emu:
emu: 00004f20 f8 clc ; ; \xf8
emu: 00004f21 4b dec BX ; BX=0x0 ; K
emu: 00004f22 46 inc SI ; SI=0x0 ; F
emu: 00004f23 bf474f mov DI 0x4f47 ; DI=0x0 ; \xbfGO
emu: 00004f26 b86b1b mov AX 0x1b6b ; AX=0x0 ; \xb8k\x1b
emu: 00004f29 90 nop ; ; \x90
emu: 00004f2a fc cld ; ; \xfc
emu: 00004f2b b9b504 mov CX 0x4b5 ; CX=0x0 ; \xb9\xb5\x04
emu: 00004f2e 90 nop ; ; \x90
emu: 00004f2f 2bda sub BX DX ; BX=0xffff DX=0x0 ; +\xda
emu: 00004f31 2bd8 sub BX AX ; BX=0xffff AX=0x1b6b ; +\xd8
emu: 00004f33 33d9 xor BX CX ; BX=0xe494 CX=0x4b5 ; 3\xd9
emu: 00004f35 310d xor DS:[DI] CX ; DS=0x0 DI=0x4f47 CX=0x4b5 ; 1\x0d
emu: 00004f37 3105 xor DS:[DI] AX ; DS=0x0 DI=0x4f47 AX=0x1b6b ; 1\x05
emu: 00004f39 f8 clc ; ; \xf8
emu: 00004f3a 43 inc BX ; BX=0xe021 ; C
emu: 00004f3b 90 nop ; ; \x90
emu: 00004f3c 40 inc AX ; AX=0x1b6b ; @
```

```
emu: 00004f3d 42 inc DX ; DX=0x0 ; B
emu: 00004f3e 4b dec BX ; BX=0xe022 ; K
emu: 00004f3f 46 inc SI ; SI=0x1 ; F
emu: 00004f40 47 inc DI ; DI=0x4f47 ; G
emu: 00004f41 e2eb loop 0xeb ; 0x4f2e ; \xe2\xeb
emu:
emu: 00004f2e 90 nop ; ; \x90
...
##==>> emu found suspicious code: "vpoly.com"
emu: writing decrypted, len = 1160
emu: writing Entry code, len = 1200
```

Note that emulation follows the “loop” instruction jump back to 0x4f2e. The decrypted section of virus code is now available in file *decrypted* and we use the first 32 bytes of that to create a VDL:

```
% od -tx1 decrypted | head -2
0000000 b4 2a cd 21 81 fa 01 04 75 11 b8 15 05 b5 00 ba
0000020 00 00 8e c2 bb 00 00 cd 13 cd 20 e9 a4 00 48 69
% od -tx1 decrypted | head -2 | ../vdl/binary.sh > vbin.vdl
% fold vbin.vdl
:vbin,"\xb4\x2a\xcd\x21\x81\xfa\x01\x04\x75\x11\xb8\x15\x05\xb5\x00\xba\x00\x00\
x8e\xc2\xbb\x00\x00\xcd\x13\xcd\x20\xe9\xa4\x00\x48\x69"#
```

Note below that if the vbin.vdl VDL is used with the general *vdl* engine, it does not match. However using it with the *vdle* engine it does match, since the VDL is run on the virus code fragment after decrypting:

```
% vfind --liboff='*' --vdl=vbin.vdl vpoly.com |& grep VIRUS
% cat config2
do_call 1
do_jcc 1
do_jmp 1
do_loop 1
do_mem 1
op_count_max 30000
% vfind --liboff='*' --emu-config=config2 --vdle=vbin.vdl vpoly.com |&
egrep -i 'VIRUS|emu'
##==>> Loading emu configuration from: config2
##==> Emulating .COM: "vpoly.com"
##==>> emu found suspicious code: "vpoly.com"
##==>>>> VIRUS POSSIBLE IN FILE: "vpoly.com"
##==>>>> VIRUS ID: CVDL vbin
##==>>>> VIRUS END OFFSET: 32, matched: \x00\x8e\xc2\xbb\x00\x00\xcd\x13\xcd
\xe9\xa4\x00Hi
##==>>>> Number of possible virus infections: 1
%
```

4.4 vdlE Engine

The *vdlE* scan engine was designed to match code near the Entry Point of Microsoft executables. It skips file headers and data segments and scans a limited portion of the input starting at the beginning of execution. This makes the engine fast, with reduced chance of false hits. Emulator options *Entry jmp limit* and *Entry len limit*, described in Section 4.3.1, can be used to control the behavior of the *vdlE* engine.

In the Emulation Example of Section 4.3.2, note at the end of the VFind output that besides writing a "decrypted" file, it also wrote an "Entry" file. Continuing that example, we could create a VDL from the beginning of the Entry code:

```
% od -tx1 Entry | head -2
0000000 f8 4b 46 bf 47 4f b8 6b 1b 90 fc b9 b5 04 90 2b
0000020 da 2b d8 33 d9 31 0d 31 05 f8 43 90 40 42 4b 46
%
```

Note that the extracted Entry code "f8 4b 46 ..." starts after the emulator has followed the initial jmp instruction.

Another example, using an innocuous "sleep.exe" executable:

```
% cd examples/vdlE
% vfind --liboff='*' --emu=11 sleep.exe
...
##==> Emulating EXE: "sleep.exe"
emu: JMP enabled.
emu: 00000012 8cc8 mov AX CS ; AX=0x0 CS=0x0 ; \x8c\xc8
emu: 00000014 8ed8 mov DS AX ; DS=0x0 AX=0x0 ; \x8e\xd8
emu: 00000016 8cc3 mov BX ES ; BX=0x0 ES=0x0 ; \x8c\xc3
emu: 00000018 81c31000 add BX 0x10 ; BX=0x0 ; \x81\xc3\x10\x00
emu: 0000001c 891e1000 mov DS:0x10 BX ; DS=0x0 BX=0x10 ; \x89\x1e\x10\x00
emu: 00000020 8bc3 mov AX BX ; AX=0x0 BX=0x10 ; \x8b\xc3
emu: 00000022 03060600 add AX DS:0x6 ; AX=0x10 DS=0x0 ; \x03\x06\x06\x00
emu: 00000026 8ec0 mov ES AX ; ES=0x0 AX=0x10 ; \x8e\xc0
emu: 00000028 8b0e0200 mov CX DS:0x2 ; CX=0x0 DS=0x0 ; \x8b\x0e\x02\x00
emu: 0000002c 8bf1 mov SI CX ; SI=0x0 CX=0x110 ; \x8b\xf1
emu: 0000002e 4e dec SI ; SI=0x110 ; N
emu: 0000002f 8bfe mov DI SI ; DI=0x0 SI=0x10f ; \x8b\xfe
emu: 00000031 fd std ; ; \xfd
emu: 00000032 f3a4 rep movsb DS:[SI] ES:[DI] ; DS=0x0 SI=0x10f ES=0x10
DI=0x10f ; \xf3\xa4
emu: 00000034 50 push AX ; AX=0x10 ; P
emu: 00000035 b83a00 mov AX 0x3a ; AX=0x10 ; \xb8:\x00
emu: 00000038 50 push AX ; AX=0x3a ; P
emu: 00000039 cb retf ; ; \xcb
...
%
```

We can extract the third column (data bytes in hex) of the emulator output into a file, then create a VDL:

```
% cat j
8cc88ed88cc381c31000891e10008bc3030606008ec08b0e02008bf14e8bfefdf3a450b83a0050cb
% sed -e 's/../\\x&/g' -e 's/^/:sleep,"/' -e 's/$/"#/' < j > j.vdl
% fold j.vdl
:sleep,"\x8c\xc8\x8e\xd8\x8c\xc3\x81\xc3\x10\x00\x89\x1e\x10\x00\x8b\xc3\x03\x06
\x06\x00\x8e\xc0\x8b\x0e\x02\x00\x8b\xf1\x4e\x8b\xfe\xfd\xf3\xa4\x50\xb8\x3a\x00
\x50\xcb"#
```

Note below that using the VDL with the general *vdl* engine, it matches the pattern at end offset 11482, whereas the *vdlE* engine matches at end offset 40. This illustrates the efficiency of the *vdlE* engine; since it started scanning at the entry point, it skipped the first 11,442 bytes of the input data:

```
% vfind --liboff='*' --vdl=j.vdl sleep.exe |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "sleep.exe"
##==>>>> VIRUS ID: CVDL sleep
##==>>>> VIRUS END OFFSET: 11482, matched:
\x00\x8b\xf1N\x8b\xfe\xfd\xf3\xa4P\xb8:\x00P\xcb
% vfind --liboff='*' --vdlE=j.vdl sleep.exe |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "sleep.exe"
##==>>>> VIRUS ID: CVDL sleep
##==>>>> VIRUS END OFFSET: 40, matched:
\x00\x8b\xf1N\x8b\xfe\xfd\xf3\xa4P\xb8:\x00P\xcb
%
```

4.5 vdlm Engine

The *vdlm* (m for *meta*) engine detects patterns in the results of the other engines rather than in the data. For each input file scanned, all "VIRUS ID" results are saved in a temporary buffer. When the end of the input file is reached, and all other engines are finished scanning the file, then the *vdlm* engine scans the buffer of results.

Take the first example from Section 4.1; based on the hits reported, the temporary buffer scanned by the *vdlm* engine would contain the following three lines:

```
CVDL ag4
CVDL ag3
CVDL ag2
```

Using the high–level CVDL “AND” operator from Chapter 8, a meta vdl could be written to match if both “ag2” and “ag3” are found:

```
% cd examples/vdlm
:ag2+ag3, "CVDL ag2" AND "CVDL ag3" #
% cat ag23.vdl
:ag2+ag3, "CVDL ag2" AND "CVDL ag3" #
% vfind --liboff='*' --vdl=../vdl/ag.vdl --vdlm=ag23.vdl ../vdl/x.txt |& grep VIRUS
...
##==>>>> VIRUS ID: CVDL ag2+ag3
##==>>>> VIRUS END OFFSET: 17, matched: DL ag4\x0aCVDL ag3
%
```

In conjunction with notell VDLs described in Section 11.4, meta VDLs can be useful for detection with UAD/SmartScan input that decomposes archives into separate components. Since each component is scanned separately, it is not possible to create a regular VDL which would match across components. However, notell VDLs can be created to match individual components, and then a meta VDL can be written to match on the notell hits. For more information and examples see Chapter 11 and Section 11.5 in particular.

4.6 jadevdl Engine

The *jadevdl* engine is a Java disassembler which scans the results of Java class file disassemblies rather than the original data. The engine is based on the VSTK *JDIS* tool, which can be used for analysis of class files to design VDLs.

Note that Java disassembly does not produce Java source code, it produces the instructions that comprise the Java byte codes, for each of the methods in a class. These are documented in the Java Virtual Machine Specification [7].

The following example shows a Java applet which may be hostile since it attempts to read a local file (SYSTEM.DAT):

```
% cd examples/jadevdl
% cat applet.java
import java.io.*;
import java.applet.*;
public class applet extends Applet {
        public void start() {
           try { BufferedReader fin = new BufferedReader(
                    new InputStreamReader( new FileInputStream( "SYSTEM.DAT")));
             String line;
             while( (line = fin.readLine()) != null) {
               // ... process line ...
             }
             fin.close();
           } catch( Exception e) { }
        }
}
%
```

Using *jdis* on the applet class file, we can analyze the disassembly:

```
% jdis -f applet.class

public class applet extends java.applet.Applet

Method public void <init>()
        0 aload_0
        1 invokespecial #8 <Method: (void) java.applet.Applet.<init>()>
        4 return

Method public void start()
        0 new #4 <Class: java.io.BufferedReader>
        3 dup
        4 new #6 <Class: java.io.InputStreamReader>
        7 dup
        8 new #5 <Class: java.io.FileInputStream>
        11 dup
        12 ldc #1 <String: "SYSTEM.DAT">
        14 invokespecial #11 <Method: (void) java.io.FileInputStream.<init>(java.lang.String)>
        17 invokespecial #9 <Method: (void)
        java.io.InputStreamReader.<init>(java.io.InputStream)>
        20 invokespecial #10 <Method: (void) java.io.BufferedReader.<init>(java.io.Reader)>
        23 astore_1
        24 goto 27
        27 aload_1
        28 invokevirtual #13 <Method: (java.lang.String) java.io.BufferedReader.readLine()>
```

```
31 dup
32 astore_2
33 ifnonnull 27
36 aload_1
37 invokevirtual #12 <Method: (void) java.io.BufferedReader.close()>
40 goto 44
43 pop
44 return
}
```

Based on the above, we create a VDL to match ”SYSTEM.DAT” followed within 80 bytes by ”FileInputStream”. Note below that using the vdl in the general *vdl* scan engine it does not match, however using it in the *jadevdl* engine it does match the class file:

```
% cat system.vdl
:system, "\"SYSTEM.DAT\"", @-80, "FileInputStream" #
% vfind --liboff='*' --vdl=system.vdl applet.java applet.class |& grep VIRUS
% vfind --liboff='*' --jadevdl=system.vdl applet.java applet.class |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "applet.class"
##==>>>> VIRUS ID: CVDL system
##==>>>> VIRUS END OFFSET: 468, matched: FileInputStream
%
```

4.7 MD5/CIT Engine

The *md5* engine is an MD5–hash–based [8] engine for efficient whole–file matches using hash databases. The engine is related to the VSTK *CIT* program (Cryptographic Integrity Tool), which can be used to create MD5 databases. The “dangerfile” distributed with cit is an example of an md5 database.

In general, for use with the *md5* engine, a database should contain one line per hash consisting of 32 hex digits (the file hash or “signature”) followed by a blank or tab, followed by the file name. Empty lines and lines starting with # are ignored, and any trailing blanks, tabs, carriage-returns, and new lines are ignored.

The following example shows a script used to create an md5 database from a set of files using either CIT or OpenSSL [9]:

```
% cd examples/md5
% cat mkdb.sh
#! /bin/sh
# read file names from stdin and create md5 db
while read pathname
do
        # use cit or openssl
        h=‘cit "$pathname"‘
        #h=‘openssl md5 "$pathname" | awk ’{ print $2 }’‘
        fname=‘basename "$pathname"‘
        echo "$h $fname"
done
% find ../cbayes/z -name ’*.txt’ -print | ./mkdb.sh > z.db
% cat z.db
dab3b1bc468c2a147f705a016b88842f z1.txt
373cee1606a140f8a436e0dbb0d7698e z2.txt
e0dccb5ed73f5e1f1373f7c752109eee z3.txt
07bf0b1b489b6086469cce20a60ae9ca z4.txt
4b6daf7ef35a5a704aa389e5c89e43cb z5.txt
5725d4a05911e0fb73a617503d368c4e z6.txt
39e8b6d60503540c9414e9d687a3038f z7.txt
dd9c712d9be1623e001823d6cd2ed35b z8.txt
62b8650e7f7eac92e53217c3b6b4e6e6 z9.txt
```

Now we use VFind to scan a collection of files, including those used above to create the MD5 db, and verify the hits:

```
% find ../cbayes -type f -print | vfind --liboff=’*’ --md5=z.db |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "../cbayes/z/z1.txt"
##==>>>> VIRUS ID: MD5 z1.txt
##==>>>> VIRUS POSSIBLE IN FILE: "../cbayes/z/z2.txt"
##==>>>> VIRUS ID: MD5 z2.txt
##==>>>> VIRUS POSSIBLE IN FILE: "../cbayes/z/z3.txt"
##==>>>> VIRUS ID: MD5 z3.txt
##==>>>> VIRUS POSSIBLE IN FILE: "../cbayes/z/z4.txt"
##==>>>> VIRUS ID: MD5 z4.txt
##==>>>> VIRUS POSSIBLE IN FILE: "../cbayes/z/z5.txt"
##==>>>> VIRUS ID: MD5 z5.txt
##==>>>> VIRUS POSSIBLE IN FILE: "../cbayes/z/z6.txt"
##==>>>> VIRUS ID: MD5 z6.txt
##==>>>> VIRUS POSSIBLE IN FILE: "../cbayes/z/z7.txt"
##==>>>> VIRUS ID: MD5 z7.txt
##==>>>> VIRUS POSSIBLE IN FILE: "../cbayes/z/z8.txt"
##==>>>> VIRUS ID: MD5 z8.txt
##==>>>> VIRUS POSSIBLE IN FILE: "../cbayes/z/z9.txt"
##==>>>> VIRUS ID: MD5 z9.txt
%
```

4.7.1 Hash Theory and Collisions

Cryptographic hash functions like MD5 are designed to take an arbitrary amount of input m and produce a fixed–length number (128 bits for MD5) H using a one–way transformation, $H = h(m)$. One–way means that the inverse function $h^{-1}(H)$ does not exist, and it is not possible to reproduce the original input m from the hash. A good hash function should produce random looking results, uncorrelated for similar inputs. If it is likely that different inputs may produce the same hash value, i.e. a *collision*, that would lead to false hits in scanning.

An important property of a good hash function, for use with signature–based scanning, is that it should be very unlikely to find an input m_2 which hashes to a given $h(m_1)$. Let n represent the number of bits in a hash, with $N = 2^n$ the total number of possible hash values. Given a random input m_2, the probability that it produces the same hash as m_1 is $1/N$, i.e. 2^{-128} for MD5, which is extremely unlikely.

We may also consider how many random inputs must be examined in order for the probability of a collision with a specific hash to reach 1/2. For k inputs we have approximately $1/2 = k(1/N)$, and solving for k yields:

$$k = N/2 \; . \qquad (4.1)$$

For MD5, that is 2^{127} inputs, a very large number. For a more rigorous derivation of this and other results, see Stallings [10] page 341.

But in the overall scheme of things, we are examining input files and deciding for one reason or another that some are “bad”, so we may create MD5 signatures for those, and some files are “good”, so we hope those do not have a collision with any of our signatures. It is in this broader context that perhaps the most important property of a good hash function arises, that is: it is unlikely to find two different inputs which produce the same hash value.

Using an argument from Kaufman, et. al. [11] page 103, with k inputs, there are $k(k - 1)/2$ distinct pairs of inputs. For each pair, there is a probability of $1/N$ of a collision, so to reach a probability of 1/2 we have approximately $1/2 = (k(k - 1)/2)(1/N)$, and solving for k yields:

$$k = \sqrt{N} \; . \qquad (4.2)$$

For MD5, that is 2^{64} inputs, a fairly large number, considering that the odds of winning the top prize in a lottery and being killed by lightning in the same day are about 1 in 2^{55} (see Schneier [12] page 18).

4.8 Exercises

1. Why is (4.1) only approximate? Perform a more rigorous derivation. Hint: solve for P(no collisions) = 1/2.
2. Why is (4.2) only approximate (besides treating (k - 1) as k)? Perform a more rigorous derivation.

Chapter 5

CBayes S*can Engine*

The VFind *cbayes* (CyberSoft Bayes) scan engine uses probabilistic techniques to classify data as being in one of two opposite groups. For each set of opposite groups to be detected, the classification uses a database created from samples of both groups.

The groups are generically referred to as “bad” and “good”, but the actual classification reported is based on the cbayes database file name, e.g. a hit using a database named *spam.dat* would be reported as:
##==>>>> VIRUS ID: CBAYES spam 1.000000000

5.1 Bayesian Classification Theory

Given some input *data*, the problem is to decide if the data should be classified as *bad* or *good*. The decision is based on a probability P computed using Bayes rule:

$$P = P(bad|data) = P(data|bad)P(bad)/P(data)\ . \tag{5.1}$$

The apriori $P(bad)$ may be set to 0.5 if it is unknown. $P(data)$ is basically just a scale factor which will cancel out when we compute the ratio $P/(1 - P)$ below.

The data is decomposed into a set of n tokens, which for text data may be words, groups of words, or other features.

$$data = \{t_1, t_2, \ldots, t_n\}\ . \tag{5.2}$$

Then we have:

$$P(data|bad) = P(t_1, t_2, \ldots, t_n|bad) \tag{5.3}$$

$$\approx P(t_1|bad)P(t_2|bad)\cdots P(t_n|bad)\ , \tag{5.4}$$

where the approximation assumes that the tokens are independent; this is called *naive* Bayesian Classification. For text data, using tokens derived from consecutive groups of words helps to make the approximation better.
Now consider $P(good|data)$, using Bayes rule:

$$1 - P = P(good|data) = P(data|good)P(good)/P(data)\ , \tag{5.5}$$

and $P(data|good)$ may be approximated as:

$$P(data|good) \approx P(t_1|good)P(t_2|good)\cdots P(t_n|good)\ . \tag{5.6}$$

By computing the ratio $P/(1 - P)$ we can eliminate the $P(data)$ scale factor:

$$\frac{P}{1-P} = \frac{P(t_1|bad)\ P(t_2|bad)\ \ldots\ P(t_n|bad)P(bad)}{P(t_1|good)\ P(t_2|good)\ \ldots\ P(t_n|good)P(good)} \tag{5.7}$$

5.1.1 Token Probabilities

For each token t_i, $i = 1, \ldots, n$, the probabilities $P(t_i|bad)$ and $P(t_i|good)$ are estimated from counts of occurrences of *ti* in bad and good training data:

$$b_i = count(t_i, bad)/count(bad) \tag{5.8}$$
$$g_i = count(t_i, good)/count(good)\ , \tag{5.9}$$

where $count(bad)$ is the number of bad data sets in the training data, and $count(t_i, bad)$ is the number of occurrences of t_i in the bad training data. The count ratios can turn out larger than 1, but will be normalized below. b_i and/or g_i may be 0, so to avoid multiplying or dividing by zero we compute the ratio r_i:

$$r_i = 1/2,\ if\ b_i = g_i \tag{5.10}$$
$$= b_i/(b_i + g_i),\ if\ b_i \neq g_i\ ,$$

and define:

$$p_i = \max(P_{min}, \min(P_{max}, r_i)) \tag{5.11}$$
$$q_i = 1 - p_i \tag{5.12}$$

Since r_i may be 0 or 1, the max,min restriction ensures that $0 < P_{min} \leq p_i \leq P_{max} < 1.0$ so that neither p_i nor q_i are zero. Typical values used for P_{min} and P_{max} depend on the number of messages in the training data.

The final ratio of probabilities for t_i is:

$$\frac{P\ (t_i|bad)}{P\ (t_i|good)} = \frac{p_i}{q_i}\ . \tag{5.13}$$

5.1.2 Overall Probability

To avoid underflow to 0 and overflow to 1 when computing products of probabilities, we compute the log ratio:

$$X = \log(P/(1 - P)) \tag{5.14}$$
$$= \log(p_1) - \log(q_1) + \cdots + \log(p_n) - \log(q_n) + \log(P(bad)) - \log(P(good))\ ,$$

from which the overall probability of bad can be computed as:

$$P = \frac{1}{(1+e^{-x})} \tag{5.15}$$

If $P > 0.5$ then the data is more likely to be bad than good, but to help reduce false hits a higher threshold can be used.

5.2 Text Tokenization and Hashing

Bayesian classification may be performed on any type of data, but here we concentrate on text data and the tokenization and hashing utilities currently available in the CyberSoft CBayes toolkit. Operation of these utilities is described below: chash, chash–merge, chash–combine, chash2–merge, cbayes.

The CBayes utilities were designed to facilitate an adaptive sliding window approach to Bayes training and scanning. For detecting e-mail spam for example, collections of spam and clean hash files may be archived on a daily basis using *chash*. Then a window of some number of past days spam and clean archives would be merged into cumulative hash files using *chash–merge*. The cumulative spam and clean hash files would then be combined into one file using *chash–combine*. Combined hash files may themselves be merged with other combined files, e.g. from archives on multiple e-mail servers, using *chash2–merge*. The combined hash file is then converted by *cbayes* into a scan database for use with VFind.

To create a CBayes scan database, start with a set of bad and good sample training files, and split each file into words (i.e. tokens). The set of delimiters separating tokens used by emphCHash are defined by this C string:
" \t\r\n\f\v\"#()*+,/:;<=>?@[\\]^‘{|}~"

The tokens are then hashed to produce fixed–size (i.e. 64–bit) data. Tokens are hashed individually as well as optionally in groups of sequential tokens. Token group level 3 is recommended and is the default for chash. Level 3 means that all individual words, as well as word pairs (pairs of sequential words), and word triplets (groups of three sequential words) will be tokenized and hashed. The hashed data is then stored in bad and good database files together with the counts of frequency of occurrence. The format of these individual bad and good hash files is called CHASH1. Sets of bad or good CHASH1 data may be merged into one file using *chash–merge*.

chash–combine is then used to combine one bad and one good CHASH1 file into a CHASH2 file representing all of the hashes and counts. *chash2–merge* can then be used to merge multiple CHASH2 filesif desired. Finally, the log differences $\log(p_i) - \log(q_i)$ are pre-computed from the CHASH2 data and stored as a CBAYES1 database using *cbayes*.

During scanning, tokens and groups of tokens from the data being scanned are hashed and looked up in the cbayes database to obtain $\log(p_i) - \log(q_i)$ which are accumulated to produce the overall probability.

5.3 Examples

z/*.txt are the “bad” files, containing data that we will use for Bayes training and eventually try to detect in new files. e/*.txt are the “good” training files, containing data opposite in some sense to the data in the “bad” files.

In these examples, the “bad” z/*.txt filesare actually just encyclopedic entries about zebras, and the “good” e/*.txt filesare entries about elephants.

```
% cd examples/cbayes
% ls -R
.:
dat e other z
./e:
e1.txt e2.txt e3.txt e4.txt e5.txt e6.txt e7.txt
./z:
z1.txt z2.txt z3.txt z4.txt z5.txt z6.txt z7.txt z8.txt z9.txt
# examine some of the z/ files
#
% foreach i (1 2 3)
? echo z$i.txt
? head -1 z/z$i.txt
? end
z1.txt
Zebras are members of the horse family native to central and southern
z2.txt
The Plains Zebra (Equus quagga, formerly Equus burchelli) is the most
z3.txt
The Mountain Zebra (Equus zebra) of southwest Africa tends to have a sleek
# examine some of the e/ files
#
% foreach i ( 1 2 3 4 )
? echo e$i.txt
? head -1 e/e$i.txt
? end
e1.txt
Elephantidae (the elephants) is the only extant family in the order
e2.txt
An elephant’s most obvious characteristic is the trunk, a much elongated
e3.txt
Elephants have three premolars and three molars in each quadrant. They erupt
e4.txt
Skin diseases often occur, from which they try to protect themselves by
# create separate hash file for each z/ input text file
#
% foreach i (1 2 3 4 5 6 7 8 9)
? chash z/z$i.txt > dat/z$i.dat
```

```
? end
# note CHASH1 format
#
% head -6 dat/z1.dat
CHASH1
n 293
nfiles 1
DATA
00311bc8cf7920ed 1
0035dc18f0bbd4d7 1
# alternatively, create one hash file from all z/ input files
#
% chash z/z*.txt > dat/z-all.dat
#
# note CHASH1 format
#
% head -6 dat/z-all.dat
CHASH1
n 2219
nfiles 9
DATA
000fcfcc3008220c 1
0011ca19bf83690c 2
# check: chash-merge result is equivalent
#
% chash-merge dat/z?.dat | diff - dat/z-all.dat
---
# create separate hash file for each e/ input text file
#
% foreach i (1 2 3 4 5 6 7)
? chash e/e$i.txt > dat/e$i.dat
? end
# create merged hash file
#
% chash-merge dat/e?.dat > dat/e-all.dat
# note CHASH1 format
#
% head -6 dat/e-all.dat
CHASH1
n 2091
nfiles 7
DATA
0007bbfb33ad6b5a 2
0008267ef6acabe5 1
---
```

```
# combine z and e hash files into a CHASH2 database
#
% chash-combine dat/z-all.dat dat/e-all.dat > dat/ze-all.dat
# note CHASH2 format
#
% head -7 dat/ze-all.dat
CHASH2
n 4172
badfiles 9
goodfiles 7
DATA
0007bbfb33ad6b5a 0 2
0008267ef6acabe5 0 1
# create Bayes database for use in scanning
#
% cbayes dat/ze-all.dat > dat/ze-cbayes1.dat
# note CBAYES1 format
#
% head -14 dat/ze-cbayes1.dat
CBAYES1
n 4163
badfiles 9
goodfiles 7
group 3
notell 0
Ethresh 0
Hthresh 1
Pbad 0.5
Pdetect 0.99
Pmin 0.01
DATA
0007bbfb33ad6b5a -4.5951199e+00
0008267ef6acabe5 -4.5951199e+00
---
# check: chash2-merge can merge CHASH2 databases
#
# first create two partial CHASH2 databases
#
% mkdir tmp
% chash-merge dat/z{1,2,3,4,5}.dat > tmp/z1-5.dat
% chash-merge dat/z{6,7,8,9}.dat > tmp/z6-9.dat
% chash-merge dat/e{1,2,3,4}.dat > tmp/e1-4.dat
% chash-merge dat/e{5,6,7}.dat > tmp/e5-7.dat
% chash-combine tmp/z1-5.dat tmp/e1-4.dat > tmp/ze1.dat
% chash-combine tmp/z6-9.dat tmp/e5-7.dat > tmp/ze2.dat
# now merge and compare with dat/ze-all.dat
```

```
#
% chash2-merge tmp/ze1.dat tmp/ze2.dat | diff - dat/ze-all.dat
% rm -r tmp
---
# use VFind and ze-cbayes1.dat, scan original training files
#
% vfind --liboff='*' --cbayes=dat/ze-cbayes1.dat z/*.txt e/*.txt |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "z/z1.txt"
##==>>>> VIRUS ID: CBAYES ze-cbayes1 1.000000000
##==>>>> VIRUS POSSIBLE IN FILE: "z/z2.txt"
##==>>>> VIRUS ID: CBAYES ze-cbayes1 1.000000000
##==>>>> VIRUS POSSIBLE IN FILE: "z/z3.txt"
##==>>>> VIRUS ID: CBAYES ze-cbayes1 1.000000000
##==>>>> VIRUS POSSIBLE IN FILE: "z/z4.txt"
##==>>>> VIRUS ID: CBAYES ze-cbayes1 1.000000000
##==>>>> VIRUS POSSIBLE IN FILE: "z/z5.txt"
##==>>>> VIRUS ID: CBAYES ze-cbayes1 1.000000000
##==>>>> VIRUS POSSIBLE IN FILE: "z/z6.txt"
##==>>>> VIRUS ID: CBAYES ze-cbayes1 1.000000000
##==>>>> VIRUS POSSIBLE IN FILE: "z/z7.txt"
##==>>>> VIRUS ID: CBAYES ze-cbayes1 1.000000000
##==>>>> VIRUS POSSIBLE IN FILE: "z/z8.txt"
##==>>>> VIRUS ID: CBAYES ze-cbayes1 1.000000000
##==>>>> VIRUS POSSIBLE IN FILE: "z/z9.txt"
##==>>>> VIRUS ID: CBAYES ze-cbayes1 1.000000000
%
# create some new samples, s1 with text similar to zebras
# s2 with text about elephants
#
% fortune -i -m horse > other/s1.txt
% fortune -i -m elephant > other/s2.txt
# scan: note s1.txt is detected
#
% vfind --liboff='*' --cbayes=dat/ze-cbayes1.dat other/* |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "other/s1.txt"
##==>>>> VIRUS ID: CBAYES ze-cbayes1 1.000000000
%
```

5.4 Exercises

1. Assume that the first lines from the three z/ and four e/ files shown in Section 5.3 are Bayes training data. Using $P_{min} = 0.01$ and $P_{max} = 0.99$, calculate p_i for each of the following four single word tokens: Zebra, the, most, is. Using $P(bad) = 0.5$, calculate the overall probability that data containing those four words is bad.
2. For each of the dat/z?.dat and dat/e?.dat files, create a Bayes database which does not include that file, then use it to scan the associated omitted text file to determine if it is similar to the others. For example, use a Bayes database built from dat/z1–8.dat and dat/e1–7.dat to scan z/z9.txt. There are 16 text files, so 16 separate Bayes databases and scan tests must be performed. *Hint: write a shell script to automate the testing.*
3. Select some of your own spam and clean e-mail samples to create a Bayes database. Test the database on the original samples and on new mail. *Hint:* uad -M *is useful for extracting components from e-mail.*

Chapter 6

Low–Level CVDL

This chapter covers the CVDL operators used at the lowest level of scanning, i.e. at the byte level of data, and sequences of bytes.

6.1 Low–level Or

The low–level or operator ‘|’ specifies the occurrence of alternative patterns at a position relative to the preceding pattern in the scanned data. For example:
"x", ("a" | "b"), "y"

matches ”x”, followed by ”a” or ”b”, followed by ”y”.

‘|’ has higher precedence than concatenation (the comma operator, Section 3.4), so the parentheses in the above example are not needed. Multiple ‘|’ patterns may be specified together, for example:
"x", "a" | "b" | "ZZZ" | "", "y"

matches ”x”, followed by ”a” or ”b” or ”ZZZ”, followed by ”y”, or ”x” directly followed by ”y” (matching the empty pattern ”” in the or).

Do not confuse the low–level ‘|’ operator with the high–level OR operator discussed later in Section 8.1. The high–level OR operator specifies the occurrence of alternative patterns at *any* positions in the scanned data.

6.2 Byte Expressions

Bytes, byte ranges, and compliments of bytes and byte ranges can be specified using characters and decimal or hex integers. Characters are written using ’single quotes’. Hex integers may be written as 0xdd or ’\xdd’ where dd represents two hexadecimal digits (from the set 0..9, a..f, A..F). Byte ranges are written in the form *start–end*, and complement is specified using a circumflex (^) prefix.
Examples:
65
0x41
’\x41’
’A’ any of these matches a byte whose value is 65 (decimal)
’a’-’f’ matches a byte whose value is in the range 0x61-0x7a
^0-10 matches a byte whose value is not in the range 0-10

6.3 Set Expressions

Sets of string or byte patterns are specified by enclosing the patterns in *{* braces *}*. For example:
{ "abc", 0-9 } matches any one of the bytes ’a’, ’b’, ’c’, 0, 1, ..., 9
Sets can be complimented using ^:
^{ "abc", 0-9 } matches a byte which is not in the set ’a’, ’b’, ’c’, 0, 1, ..., 9
Sets can also contain other sets or complimented sets.

6.4 Fuzzy Expressions

Fuzzy expressions are a convenient way of specifying ranges for bytes and strings. Fuzziness is specified using integers which represent plus and minus offsets. Examples:

```
FUZZY 2 100
FUZZY +-2 100          are the same as: 98-102
fuzzy -2 +2 100

FUZZY -2 +3 "cow"      is the same as: 'a'-'f', 'm'-'r', 'u'-'z'
```

Recognized case–insensitive spellings for the fuzzy operator are FUZZY and FUZZ.

6.5 Repetition Expressions

Multiple occurrences of bytes, byte ranges, sets, strings, or case-insensitive strings can be specified by using [*number*] after the expression. For example:
15[20]
0xF[0x14] either form matches 20 occurrences of the byte value 15
0-10[40] matches a sequence of 40 bytes whose values are in the range 0..10
"X"[3] matches "XXX"
The repetition expression can also specify a range, for example:
"X"[0-3] matches from 0 to 3 occurrences of "X"

6.6 Indefinite Offsets

The '.*' operator is used to specify an indefinite offset, up to the next new line ("\n") character. This can be used when scanning text data to match patterns which must be restricted to a single line of text. Example:
~"\nSubject:", .*, ~"herbal", .*, ~"viagra"

will match "\nSubject: 100% Pure Herbal Potent Viagra On Sale!"

As with the '@' operator (see Section 3.4), it does not make sense to use '.*' at the beginning of a VDL segment, and such use will be reported as a parser error using the vfind *-d,–dup-check* option (see Section 2.1).

6.7 Absolute Offsets

The ‘ABS’ operator is used to specify an absolute offset from the beginning of the scanned data to match a pattern. The ‘EOD’ operator is used to match exactly at end-of-data. Example VDLs:

```
; match Bourne shell script file header
;
:a1, ABS 0, "#!", WS0, "/bin/sh" #
; match "abc" followed by "def" within the next 20 bytes,
; and "01234" at absolute position 14
;
:a2, "abc", @0-20, "def" AND ABS 14, "01234" #
; match the 8-byte Microsoft signature header which appears
; at the very beginning of most Microsoft application files,
; and "\xFE\xCA" anywhere in the scanned data
;
:MS/VBA, ABS 0, "\xD0\xCF\x11\xE0\xA1\xB1\x1A\xE1" AND "\xFE\xCA" #
; match "zzz" only if it appears exactly at the end of the scanned data
;
:a3, "zzz", EOD #
; match a scanned data file which is exactly 4 bytes long and contains "abc\n"
;
:a4, ABS 0, "abc\n", EOD #
```

Section 4.1 showed an example VDL to match the EICAR [6] test pattern anywhere in a file. But a more precise definition of the EICAR test says: *the file starts with the following 68 characters, and is exactly 68 bytes long*. Using absolute offsets, the following VDL precisely matches that definition:

```
:EICAR TEST -- Fake Virus -- IGNORE, ABS 0, "\x58\x35\x4f\x21\x50\x25\x40\x41\
\x50\x5b\x34\x5c\x50\x5a\x58\x35\x34\x28\x50\x5e\x29\x37\x43\x43\x29\x37\x7d\
\x24\x45\x49\x43\x41\x52\x2d\x53\x54\x41\x4e\x44\x41\x52\x44\x2d\x41\x4e\x54\
\x49\x56\x49\x52\x55\x53\x2d\x54\x45\x53\x54\x2d\x46\x49\x4c\x45\x21\x24\x48\
\x2b\x48\x2a", EOD #
```

But a further refinement of the EICAR specification states: *The first 68 characters is the known string. It may be optionally appended by any combination of white space characters with the total file length not exceeding 128 characters. The only white space characters allowed are the space character, tab, LF, CR, CTRL–Z.* Using a combination of absolute offsets, a set, and repetition expression, the following VDL precisely matches that definition:

```
:EICAR TEST -- Fake Virus -- IGNORE, ABS 0, "\x58\x35\x4f\x21\x50\x25\x40\x41\
\x50\x5b\x34\x5c\x50\x5a\x58\x35\x34\x28\x50\x5e\x29\x37\x43\x43\x29\x37\x7d\
\x24\x45\x49\x43\x41\x52\x2d\x53\x54\x41\x4e\x44\x41\x52\x44\x2d\x41\x4e\x54\
\x49\x56\x49\x52\x55\x53\x2d\x54\x45\x53\x54\x2d\x46\x49\x4c\x45\x21\x24\x48\
\x2b\x48\x2a", {" \t\r\n\x1a"}[0-60], EOD #
```

6.8 Offset Groups

Offset ranges, indefinite offsets, and absolute offsets can be applied to groups of patterns, resulting in simpler and more readable VDLs. Examples:
The VDLs:
:o1, "t1", @-10, ("t2"|"t3") #
:o2, "t1", .*, ("t2"|"t3") #
:o3, ABS 0, ("t2"|"t3") #

are equivalent to:
:o1, "t1", (@-10,"t2"|@-10,"t3") #
:o2, "t1", (.*,"t2"|.*,"t3") #
:o3, ABS 0,"t2"|ABS 0,"t3" #

6.9 Exercises

Select some of your own spam and clean e-mail samples and write VDLs to match the spam samples. Run the VDLs on new mail to split it into spam and clean samples, then use those samples to create a Bayes database.

Chapter 7

CVDL and Text

Although VFind treats all data as binary, this just means that no data is skipped when scanning, not even line terminators that may appear in text files. A text file is simply a special case of a binary file, one that tends to not have 8–bit or control characters, with human readable data organized by lines and separated carriage–return and/or new line characters. For scanning text data, CVDL provides a set of specialized operators described in this chapter.

CVDL also provides a file type restriction directive, described in Section 11.1, that can be used to restrict the scope of VDLs to be applied only to text files, which can reduce scanning time and the potential for false hits.

7.1 White space

There are two special operators, ‘W0’ and ‘W1’, for offsets consisting of white space characters as defined by the C library isspace() function, i.e. characters ”\t” (horizontal tab), ”\n” (new line), ”\v” (vertical tab), ”\f” (form feed), ”\r” (carriage return), and ” ” (space).
W0 matches zero or more white space characters
W1 matches one or more white space characters

Examples:
"From:", W0, ~"<spam@spam.com>"
~"Bulletproof", W1, ~"Web", W1, ~"Hosting"

7.2 White space (shell)

There are two special operators, ‘WS0’ and ‘WS1’, for offsets consisting of characters which are treated as white space by the UNIX shells, i.e. characters ” ” (blank), ”\t” (horizontal tab), and ”\\\n” (backslash new line):
WS0 matches zero or more shell white space characters
WS1 matches one or more shell white space characters

Examples:
"/bin/rm", WS1, "-rf", WS1, "/"
"cat", WS0, ">>", WS0, "/etc/passwd"

7.3 White space and Punctuation

There are two special operators, ‘WP0’ and ‘WP1’, for offsets consisting of white space and punctuation characters as defined by the C library isspace() and ispunct() functions. The punctuation characters are:
! " # $ % & ’ () * + , - . / : ; < = > ? @ [\ ] ^ _ ‘ { | } ~
WP0 matches zero or more white space or punctuation characters
WP1 matches one or more white space or punctuation characters
Example:
~"bomb", WP1
matches ”bomb ” and ”bomb.” but does not match ”bombastic” or ”bombproof”.

7.4 Skipping White space and Punctuation

The '~~' operator was designed for matching phone numbers, and is an extension of the case–insensitive string matching '~' operator. '~~' performs case–insensitive string matching while skipping any white space and punctuation characters. White space and punctuation characters are those defined by the C library isspace() and ispunct() functions. Examples:
~~"123-456-7890"

matches "(123)-456-7890" and "123.456.7890" and
"1 2 3 - 4 5 6 - 7 8 9 0", etc.

~~"800 FREE CAR"
matches "(800) - F r e e C a r !!!", etc.

7.5 Wildcard White space

The '~W' operator was designed for case–insensitive matching for sequences of words separated by white space, where white space characters are those defined by the C library isspace() function. The " " (blank) character acts as a white space wildcard in the VDL, and will match one or more white space characters in the data being scanned. Examples:
~W" this is a test"
matches " this \r\nis a test"
matches "\nthis \r\nis a test"
~W"\nthis is a test"
matches "\nthis \r\nis a test"
does not match " this \r\nis a test"

7.6 Digits

The \d+ operator matches one or more digits, and is named after the similar Perl operator. This can be used, for example, to detect obfuscated URLs:
"http://", \d+, "/"
matches URLs like http://3626287830/

"http://0", \d+, ".", "0", \d+, ".", "0", \d+, ".", "0", \d+, "/"
matches URLs like http://00000325.0000030.00000341.00000116/

can also be written using a macro:
$define zerod "0", \d+
"http://", $zerod, ".", $zerod, ".", $zerod, ".", $zerod, "/"

CVDL macros are discussed in detail in Chapter 10.

7.7 Only Digits

The '~#' operator matches only digits, skipping all other characters, over a default maximum range of 30 bytes of scanned input data. The maximum range of scanned input data can be specified by placing a number between '~#' and the digit string.

Examples:
~#"code 1234 sub-code 567"
matches the digits 1234567 in sequence, regardless of any intervening non-digit characters, over any 30 byte range of scanned input data, e.g. it will match "1abc2efg34---5 6 7"

~#60"code 1234 sub-code 567"
As above, but over a maximum range of 60 bytes of input data.

7.8 Floating–point Comparison

The operator pair '%f >' can be used to extract a floating–point value from scanned text data and compare it to a constant. It will match if the value is greater than the constant. For example:
"Fuz1=", %f > 0.5
will match if the data following the string "Fuz1=" is a floating–point value greater than 0.5, e.g. it will match "Fuz1=0.76".

'%f' should only be used in conjunction with a file–type restriction or preceding match so that the presence of floating–point data is assured. As the first pattern in a VDL it is only tried once, at the beginning of the scanned data buffer.

The '%f >' operator pair was first added to CVDL on an experimental basis for use in checking Bayesian spam probabilities, and has subsequently been found to be useful for checking other types of numerical text data. Eventually, floating–point comparison operators besides > (i.e. <, ==, <=, >=, !=) may also be added to CVDL.

7.9 Exercises

Write a VDL to detect the following data, including the two new line characters:

Mary had a little lamb
Its fleece was white as snow.

a) in MS/Win text;
b) UNIX text;
c) platform independent.

Chapter 8

High–Level CVDL

Originally, CVDL had only one high–level logic operator, ‘&’, meaning AND. The current logical operators are ‘AND’, ‘OR’, ‘XOR’, and ‘NOT’, which allow construction of arbitrarily complex logic patterns. Also, a file size operator is available, and is used with comparison operators to produce logical values.

The precedence of the logic operators is, in order of decreasing precedence: NOT, AND, XOR, OR. Parentheses can be used to group operations into a desired execution order irrespective of operator precedence.

8.1 AND and OR

AND and OR are used to logically combine pattern matching results. Let t1...t4 represent any CVDL pattern, and consider the following VDLs:
:v12, t1 AND t2 #
:v34, t3 OR t4 #

v12 will match only if t1 matches anywhere in the scanned data and t2 also matches anywhere in the same scanned data. v34 will match if either t3 or t4 (or both) match.

Both of these are independent of the positions of the matches in the data; for example, t4 could match a pattern that precedes the pattern matched by t3. In contrast, the low–level or operator ‘|’ discussed in Section 6.1 specifies the occurrence of alternative patterns at a position relative to the preceding pattern in the scanned data.

Some examples illustrating the precedence of the operators:
:ex1a, "pets" AND "cat" OR "dog" #
:ex1b, ("pets" AND "cat") OR "dog" # ; same as ex1b
:ex2a, "pets" AND ("cat" OR "dog") #
:ex2b, ("pets" AND "cat") OR ("pets" AND "dog") # ; same logically as ex2a

VDLs ex1a and ex1b are equivalent due to the higher precedence of AND as compared to OR, and will match data that contains either: the word ”pets” anywhere and the word ”cat” anywhere; or the word ”dog” anywhere. If the intention was to match data containing the word ”pets” anywhere, and either the word ”dog” or the word ”cat” anywhere, then parentheses can be used as in VDL ex2a and ex2b.

8.2 XOR and NOT

NOT by it self seems to be a strange logical operator for pattern matching, since it means that we have a match if some pattern is not present; however, consider a situation where all files or e-mail must contain a certain notice, and we want to detect the lack of that notice. An example pattern is:
:missing, NOT "Copyright 2005, CyberSoft Operating Corporation" #

XOR is the exclusive–OR operator, which by definition can be expressed using AND and OR:

a XOR b == (a AND NOT b) OR (b AND NOT A)

For example, VDL ex3:
; guns or bullets, but not both
;
:ex3, ("guns" AND NOT "bullets") OR ("bullets" AND NOT "guns") #
can be expressed more efficiently using XOR:
:ex3, "guns" XOR "bullets" # ; guns or bullets, but not both

8.3 File Size Operator

The CVDL file size operator returns the size in bytes of the file or data being scanned, if known. It can be written as ‘SIZE’, ‘Size’, or ‘size’, and is used with the comparison operators: < > != == <= >=

Examples:
:s0, size > 100 #
:s1, size == 5 AND "abc" #
:s2, size < 5 AND "abc" #
:s3, 5 > size AND "abc" #
:s4, ((10 <= size AND size <= 20) OR size > 1000) AND "abc" #

VDL s0 will match if the data size is greater than 100 bytes regardless of the contents of the data. For the other examples the data must contain the string ”abc” in order to match.

VDL s1 will only match if the data size is exactly 5 bytes. VDLs s2 and s3 are identical and only match if the data size is less than 5 bytes. VDL s4 will match if the data size is between 10 and 20 bytes inclusive, or greater than 1000 bytes.

Note that if the data size is not available, the size operator will not match, regardless of the comparison operator used. Currently, the file or data size is only guaranteed to be available for files of size less than 1 MB (1048576 bytes) and during processing of the last 1 MB portion of files larger than 1 MB.

8.4 File Name Operator

The CVDL file name operator compares the quantified name of the file being scanned with an arbitrary low–level CVDL pattern, and returns true if the pattern is matched by the file name.

The quantified file name is the name reported by vfind's "Checking file:" report, including the double quotes. In a quotified file name, any literal double quotes will be displayed as \",and newline characters as \n. When processing smartscan output from uad, the file name will include both the top–level and component–level file names. For example:

```
% cd examples/hlcvdl
% vfind --liboff='*' abc |& grep 'Checking file:'
##==> Checking file: "abc"
% uad -ssw abc.tar | vfind --liboff='*' -ssr |& grep 'Checking file:'
##==> Checking file: "abc.tar" -> "ZZabcYY"
##==> Checking file: "abc.tar" -> "abc"
```

The three quotified file names reported above are:
"abc"
"abc.tar" -> "ZZabcYY"
"abc.tar" -> "abc"

including the double quotes.
The file name operator consists of the word 'NAME' (case–insensitive), followed by ~=, followed by the pattern to match. For example:

```
% cat name.vdl
:contains-abc, name ~= "abc" #
:exactly-abc, name ~= ABS 0, "\"abc\"", EOD #
:exactly-abc-regex, name ~= ~R"^\"abc\"$" #
:component-abc, name ~= "-> \"abc\"", EOD #
```

The first VDL will match any file name containing the three–letter substring "abc". The second VDL uses 'ABS' and 'EOD' (see Section 6.7) to match only if the reported file name is exactly equal to the five–letter string "abc", including the double quotes, and the third VDL uses a regular expression pattern (see Chapter 9) to perform the same match. The last VDL matches on a SmartScan component file name of "abc". Example runs:

```
% vfind --liboff='*' --vdl=name.vdl abc ZZabcYY |& grep 'VIRUS'
##==>>>> VIRUS POSSIBLE IN FILE: "abc"
##==>>>> VIRUS ID: CVDL exactly-abc-regex
##==>>>> VIRUS POSSIBLE IN FILE: "abc"
##==>>>> VIRUS ID: CVDL exactly-abc
##==>>>> VIRUS POSSIBLE IN FILE: "abc"
##==>>>> VIRUS ID: CVDL contains-abc
##==>>>> VIRUS POSSIBLE IN FILE: "ZZabcYY"
##==>>>> VIRUS ID: CVDL contains-abc
% uad -ssw abc.tar | vfind -ssr --liboff='*' --vdl=name.vdl |& grep 'VIRUS'
##==>>>> VIRUS POSSIBLE IN FILE: "abc.tar" -> "ZZabcYY"
##==>>>> VIRUS ID: CVDL contains-abc
##==>>>> VIRUS POSSIBLE IN FILE: "abc.tar" -> "abc"
##==>>>> VIRUS ID: CVDL component-abc
##==>>>> VIRUS POSSIBLE IN FILE: "abc.tar" -> "abc"
##==>>>> VIRUS ID: CVDL contains-abc
```

The two file names in this example, "abc" and "ZZabcYY", both contain the "abc" substring, so the contains-abc VDL always matches. Note that the exactly-abc VDLs match "abc" when scanned directly by VFind, but they do not match when file "abc" is a component extracted from a tar archive (which does match the component-abc VDL).

As an application of the file name operator for detecting side–effects of a virus infection, it has been noted that the W32.Mytob.FT@mm virus adds a registry value containing the string "skybotx.exe" to SYSTEM.DAT. This string on its own or appearing in any file other than SYSTEM.DAT would not be suspicious. So the following VDL matches only if the string appears in SYSTEM.DAT:

```
% cat system.vdl
:W32.Mytob.FT@mm SYSTEM,
name ~= { '"', '/', '\\' }, ~"system.dat\"" AND "skybotx.exe" #
```

The file name operator above will match a quotified case–insensitive system.dat file name (e.g. "SYSTEM.DAT") or path name (e.g. "\WINDOWS\SYSTEM.DAT" or "/pc/export/system.dat").

8.5 Exercises

Select a set of system text files (e.g. man pages) and look through some of them for common words and phrases. Then create a set of VDLs using combinations of AND and OR operators, and scan all of the selected files using those VDLs.

Chapter 9

CVDL and Regular Expressions

Regular expressions (regex) are a powerful pattern matching mechanism originally designed as part of the UNIX operating system, and currently standardized by the Open Group [13]. Regex operates on character strings, i.e. sequences of non–zero bytes terminated by a zero byte, and is generally line–oriented, so it is suitable only for pattern matching in text.

VFind and CVDL were designed for fast pattern matching on binary or text data. Some of the low–level CVDL pattern matching operators are similar to regex, but CVDL is not restricted to text. As CVDL evolved to meet the needs of specialized scanning, particularly for use in detecting e-mail spam by SafeInternetEmail [3], more operators similar to regex were added. Eventually, a direct interface to the complete capabilities of regex was added.

Note that although VFind and CVDL operate on arbitrary binary data, regex only works on text. Before invoking a regex during scanning, VFind temporarily appends a zero byte to the buffer being scanned so that the regex doesn't run off of the end.

This chapter describes the use of regex with CVDL, including summaries of basic and extended regular expressions. For more information about regex, recommended references are the O'Reilly book by Friedl [14], the Open Group [13], and the UNIX man pages for regcomp(3C), regexec(3C), and regex(5).

9.1 Regex Operator

CVDL regular expression matching is specified using the '~R' operator followed by zero or more single–letter regex option settings, followed by a string containing the regex pattern. Option settings correspond to the flags passed to regcomp(3C) to compile the pattern:

X	REGEXTENDED	Use extended regular expressions.
I	REGICASE	Ignore case in match.
N	REGNEWLINE	If not set, then a new line character will be treated as an ordinary character.
B	REGNOTBOL	The first character of the data is not the beginning of a line; therefore, the circumflex character (^), when used as a special character, will not match the beginning of the data.
E	REGNOTEOL	The last character of the data is not the end of a line; therefore, the dollar sign ($), when used as a special character,will not match the end of the data.

If no regex options are specified, the result is a case–sensitive basic regular expression with new line treated as an ordinary character. Note that a regular expression as the first pattern in a VDL is only applied once, at the beginning of the scanned data buffer.

A useful feature of basic regex, which is not available in extended regex, is the ability to tag subexpressions and then use back–references to the tags. For example:

```
% cd examples/regex
% cat bob.msg
From: somebody@somewhere.com
To: cyberbob@cyber.com
Subject: welcome to spam heaven CyberBob

hi
% cat r1.vdl
:r1, ~"\nTo:", W0, ~RI"\\([^@\n]*\\)@.*\\1" #

% vfind --liboff='*' --vdl=r1.vdl bob.msg |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "bob.msg"
##==>>>> VIRUS ID: CVDL r1
##==>>>> VIRUS END OFFSET: 92, matched: heaven CyberBob
%
```

The first part of VDL r1 matches a new line followed by "To:" (case–insensitive) followed by zero or more white space characters. The regex part produces the case–insensitive pattern "\([^@\n]*\)@.*\1" which is intended to match the userid portion of an e-mail address occurring again anywhere after the initial occurrence. It works as follows: after matching "To:", the regex sub-expression [^@\n] matches any sequence of characters up to the first @ or \n, and that match is tagged since it is enclosed by \(and \). That match would be the userid portion of the mail address. Then the rest of the pattern @.*\1 matches @ followed by zero or more of any characters followed by an occurrence of the previously tagged sub-expression, reference by \1 in the regex.

Note that a regular expression is specified as a CVDL string, so it is subject to interpretation of escape sequences (see Section 3.3). This is why the example above used two backslashes to produce one. Also, \n in the regex string produced a literal new line in the pattern.

9.2 Trigger Data

As discussed in detail in Chapter 13, many CVDL operators, including regex, are not able to be implemented in the VFind parallel search engine. To improve run–time speed for regex, the regex string may optionally be followed by a parenthesized expression containing trigger data for the fast/parallel search. The only CVDL elements accepted in the trigger data are strings, concatenation, low–level or, parentheses, and AND (see Section 12.4).

For example:

```
:r2, ~RX"abcd.*(efgh|xxxx).*zzzz" ( "abcd", "efgh" | "xxxx", "zzzz" ) #
```

VDL r2 specifies an extended regular expression, including trigger data. The regex will not be triggered to run unless the parallel search engine first encounters the trigger data.

9.3 Summary of Basic Regular Expressions

Characters in a basic regular expression are classified as either *ordinary* or *special*. The special characters are . [\ * ^ $ and all other characters are ordinary. A special character preceded by a backslash becomes an ordinary character that matches the special character itself. The ordinary characters ( ) { } 1 2 3 4 5 6 7 8 9 become special when preceded by a backslash.

A bracket expression is a set of characters enclosed in [square brackets]. It matches any single character from the set, unless the first character in the set is a circumflex (^), which negates the set so it matches any single character not in the set. A range of characters can be specified using a hyphen, for example '[0-9]' is the same as '[0123456789]'. In a bracket expression, the special characters . * [\ lose their special meaning and become ordinary characters.

In a bracket expression the following character class expressions are supported: [:alnum:], [:alpha:], [:blank:], [:cntrl:], [:digit:], [:graph:], [:lower:], [:print:], [:punct:], [:space:], [:upper:], [:xdigit:]. So, for example, '[0-9]' may be alternatively written as '[[:digit:]]'.

A period (.), when used outside a bracket expression, matches any single character. A character followed by an asterisk (*) matches zero or more consecutive occurrences of the character. Thus '.*' matches zero or more of any characters, i.e. it matches anything or nothing. For example, "a.*b" would match "ab", "aZb", "aZZb", "aXYZXYZb", etc.

The circumflex (^) and dollar sign ($) match the beginning and end of data respectively, unless the REG NEWLINE option is set, then they match the beginning and end of a line respectively.

A subexpression can be defined by enclosing it between the character pairs \(and \). The back–reference expression \n, where n is a digit 1..9, matches the same string of characters as was matched by the n'th subexpression.

A pattern matching a single character, a subexpression, or a back–reference can optionally be followed by an interval expression of the form \{m\}, \{m,\} or \{m,n\} to match repeated consecutive occurrences of the pattern. m specifies the exact or minimum number of occurrences and n specifies the maximum number of occurrences. \{m\} matches exactly m occurrences, \{m,\} matches at least m occurrences, and \{m,n\} matches any number of occurrences between m and n, inclusive.

9.4 Summary of Extended Regular Expressions

The rules specified for basic regular expressions also apply to extended regular expressions with the following exceptions: the characters | + ? are special; the { } characters, when used as the duplication operator, are not preceded by backslashes, \{ and \} simply match the characters *{* and *}*, respectively; the back reference operator is not supported; anchoring using ^ and $ is supported in subexpressions.

Two expressions separated by a vertical–line (|) match a string that is matched by either expression. In other words, | is the extended regular expression 'or' operator. An expression followed by a plus–sign (+) matches one or more consecutive occurrences of the pattern, and an expression followed by a question–mark (?) matches zero or one consecutive occurrences.

9.5 Exercises

1. Try to construct trigger data for the regex in VDL r1 from Section 9.1. Modify the problem and produce a regex from which you can construct good trigger data *Hint: instead of matching the userid anywhere after the initial occurrence, match it in a Subject: header.*
2. For VDL r2 in Section 9.2, what is the difference between patterns matched by the regex vs. patterns matched by the trigger data? Can an equivalent VDL be written without using the CVDL regex operator?

Chapter 10

CVDL Macros

CVDL includes the ability to define macros, which expand to their definition when invoked, and simplify the repeated use of patterns. For patterns which are used frequently in VDLs, there is also a storage advantage provided by macros, discussed in Section 10.4.

10.1 Defining VDL Macros

A VDL macro is specified using $define as the first word on a line, and the entire macro definition must be contained all on one line. The syntax is:
$define name value

where the line contains: optional leading white space, $define, white space, name, white space, value.

The macro name must start with a letter from the set *{* 'a'-'z', 'A'-'Z', ' ' *}* followed by 0 or more letters or digits. Note that macro names are case-sensitive. The value continues to the end of the line, and trailing white space is trimmed.

10.2 Using VDL Macros

VDL macros are invoked by specifying their name after a $ character. Macros are lexical tokens,\ which means that they can not be confused with other tokens, e.g. strings. Thus:
"abc", $mac, ...

invokes the macro named *mac*, but:
"abc$mac", ...

does not invoke any macro, and is simply a literal string.

VDL macros can be nested to unlimited depth, so macros can refer to other macros in their definition. Macros cannot be used in a VDL rule before being defined, but they can be used in other VDL macro $define's before being defined.

10.3 VDL Macro Examples

Some examples using AND and OR:
$define pf1 $pets AND $food
$define pets "dog" OR "cat"
$define food "fish" OR "pie"
$define pf2 ($pets) AND ($food)

:v1, $pf1 AND "ate" #
:v2, "ate" AND $pf2 #

Note that pf1 resolves to: "dog" OR "cat" AND "fish" OR "pie"
which is the same as: "dog" OR ("cat" AND "fish") OR "pie"
but pf2 resolves to: ("dog" OR "cat") AND ("fish" OR "pie")

As with C/C++ #define macros, parentheses may be used in the VDL macro definition or invocation to ensure that the intended result is obtained.

For detecting spam, macros have been used to define alternative representations of letters which are frequently used for obfuscation, for example:

```
% cd examples/macros
% cat letters.vdl
; some spam letters
;
$define letterP {"Pp\xDE\xFE"}
$define letterI {"Ii1;!l:Ll\xa1\xa6\xcc\xcd\xce\xcf\xec\xed\xee\xef"}
$define letterL {"Ll1!l:;\xef\xa1\xa6"}
$define letterS {"Ss$2Zz\xa7\x8a\x9a"}
% cat spam.vdl
; pills
;
$define pillz $letterP,@-3,$letterI,@-3,$letterL,@-3,$letterL,@-3,$letterS
:vpillz, ~"discount", .*, $pillz #
% cat x.dat
DISCOUNT p-i-1-1-z
% vfind --liboff='*' --vdl=letters.vdl --vdl=spam.vdl x.dat |& grep VIRUS
##==>>>> VIRUS POSSIBLE IN FILE: "x.dat"
##==>>>> VIRUS ID: CVDL vpillz
##==>>>> VIRUS END OFFSET: 18, matched: COUNT p-i-1-1-z
%
```

For detecting viruses, macros have been used to specify variable machine code segments to match polymorphic samples. The following examples are from the VFind VDLs version vfind13-2005-05-25-09-36-46 macros.vdl and 166+.vdl files:

```
$define junk (0xB8-0xBF,0x00-0xFF[2])|(0xEB,0x04-0x0F)|
(0x81,0xC0-0xC7|0xD0-0xDF|0xE8-0xF7,0x00-0xFF[2])
$define crypt (0x80,0xB4-0xB7,0x00-0xFF[3])
$define incofs (0x43|0x45|0x46|0x47)
$define inccnt (0x40|0x41|0x42|0x43|0x45|0x46|0x47)
$define cmpcnt (0x81,0xF8|0xF9|0xFA|0xFB|0xFD|0xFE|0xFF,0x00-0xFF[2])
$define jzend (0x74,0x00-0x7F) ; jumps only forward
$define loop (0xE9,0x00-0xFF,0x80-0xFF) ; jumps only backward
```

:Fono.poly,<".COM","EXE"> $junk,@-0x100,$crypt,$junk,@-0x100,$incofs,
$junk,@-0x100,$inccnt,$junk,@-0x100,$cmpcnt,$jzend,$junk,@-0x100,$loop#

Besides heavy use of macros, the Fono.poly VDL also specifies a file type restriction (see Section 11.1) so that it will only scan Microsoft executable files.

10.4 VDL Macro Storage Advantage

H2: VDL Macro Storage Advantage

The data associated with the definition of a macro is stored only once, regardless of how many times the macro is used. In contrast, identical data which is repeated in VDL definitions is stored separately for each occurrence.

For example, if you use the string ”http://” 100 times in VDL definitions, that string will be stored separately 100 times, and use a total of 100*8 = 800 bytes storage (internally the string includes a terminating 0 byte at the end, so uses 8 bytes). But if you define a macro:
$define http "http://"

and use $http in the VDL definitions instead of the literal ”http://” string, then the string is stored only once, regardless of how many times the macro is used. There is an internal 16-byte overhead associated with macro data storage, so for this example the total is 16+8 = 24 bytes, compared to 800 bytes without using a macro.

10.5 Exercises

Contrive an example using 1000 VDLs all containing a particular string. Then create a compatible VDL file with the string replaced by a macro. Compare performance, both run–time CPU and RAM usage, with and without macros.

Chapter 11

CVDL Meta Operators

The CVDL meta operators control various features of scanning. They are specified either as directives which apply to all VDLs in a CVDL file, or applied for a single VDL.

The meta operators are written inside < angle brackets > and their general syntax is:

```
< "filetype", "filetype", ... >          ;list of file types to scan
< ! "filetype", "filetype", ... >        ;list of file types to not scan
< "version=nnnnnn" >                     ;report version number nnnnnn
< "start=value", "limit=value" >         ; restrict scan range
< "notell" >                             ; turn off reporting hits
< "sticky" >                             ; stay on if hit
< "clear" >                              ; clear sticky hit
```

When applied to a single VDL, the meta operators are specified directly after the comma following the VDL name, before the VDL definition. When specified as part of a VDL, the directives apply only for that particular VDL. When specified outside of a VDL, the directives have VDL file scope and apply only for VDL rules which appear following the directive in the same VDL file.

11.1 File Type Restriction Directives

File type restriction directives are written in the form:

```
< "filetype", "filetype", ... >          ; list of file types to scan
< ! "filetype", "filetype", ... >        ; list of file types to not scan
< >                                      ; resets to scan everything.
```

specifying a list of one or more file types to scan or to not scan. An empty directive resets to scan everything regardless of file type.

To utilize file type restrictions most effectively, UAD should be used with the SmartScan *–ssw* option, piped into VFind with the *-ssr* option. If run without UAD/SmartScan input, VFind does recognize the following file types on its own: ”EXE”, ”OLE”, ”Java class file”, ”text”, ”text (8-bit)”, ”.COM”, otherwise ”unknown”.

Note that the file type ”unknown” is considered a valid type, since it means that the type is not one of the many types recognized which may be placed in a file type restriction directive, so something indeed is known about the type. In this case, all file type restrictions will be applied. For example, a VDL specifying a file type restriction of <”text”> will not scan a file type of ”unknown”.

Examples:
:v1,"..."# ; applied for all file types
<"text"> ; only "text" file types for the following vdls
:v2,"..."# ; only "text"
:v3,"..."# ; only "text"
<!"HTML"> ; no "HTML" file types for the following vdls
:v4,"..."# ; only non-"HTML"
:v5,<"JPEG","GIF">"..."# ; only "JPEG" and "GIF" file types
:v6,"..."# ; only non-"HTML"
<> ; all file types for the following vdls
:v7,"..."# ; applied for all file types.

Matching for file type restrictions is case–sensitive, and only requires that the VDL–restricted type be a substring of the file type.

11.2 VDL Version Reporting

Versions for VDL files and rules can be reported using an extension to the file type restriction syntax. If you specify a string starting with *version=* in a file type restriction directive, whatever follows the '=' character in that string will be printed as an informative message about the version of the VDL file or rule.

The following example specifies a version for the VDL file and a version for VDL rule 'b':
% cd examples/meta
% cat v.vdl

<"text","version=1.2.3">

:a, "abc"#
:b, <"version=9.9"> "bbb"#

% vfind --vdl=v.vdl hi
...
##==>> Loading VDL code from: v.vdl
##==>> All SmartScan file types disabled.
##==>> SmartScan file type '*text*' enabled.
##==>> VDL file 'v.vdl' Version: 1.2.3
##==> VDL model for 'a' loaded.
##==> VDL 'b' Version: 9.9
##==> VDL model for 'b' loaded.
##==> Checking file: "hi"
...

11.3 VDL start/limit specification

The start/limit specification is not a CVDL operator; instead, it fits in with the VDL <"meta-data">, like the file-type restrictions and *version=* specification.

For example:
:x, <"start=10","limit=20"> "abc" #

The VDL above will only scan data starting at offset 10 with a limit of 20 bytes scanned. In general, if start is not specified it will start at offset 0, and if limit is not specified there is no limit. Thus you can use start and limit together, or just one of them. Where appropriate, by restricting the offset and size of data scanned by a VDL using the start/limit specification, VFind may run faster.

The start/limit specification can be mixed in with file type restrictions and the version specification.

For example:
% cat y1.vdl
:y1, <"text", "start=100", "limit=1000", "version=1.2.3"> "abc", @-100, "efg" #
% vfind --vdl=y1.vdl
...
##==>> Loading VDL code from: y1.vdl
##==> VDL 'y1' Version: 1.2.3
##==>> SmartScan file types enabled for VDL model 'y1': '*text*'
##==> VDL model for 'y1' loaded.
...
% cat y2.vdl
:y2, <!"GIF", "JPEG", "start=100"> "frog", @-100, "fish" #
% vfind --vdl=y2.vdl
...
##==>> Loading VDL code from: y2.vdl
##==>> SmartScan file types disabled for VDL model 'y2': '*JPEG*' '*GIF*'
##==> VDL model for 'y2' loaded.

11.4 VDL notell specification

The notell specification is not a CVDL operator; instead, it fits in with the VDL <"meta-data">, like the file-type restrictions and *version=* specification. It turns of reporting of individual viruses the same as the VFind *–notell=* and *–notells=* command-line options.

For example:
```
% cat n.vdl
<"notell">
:x,"a"#
:y,<!"notell">"b"#
:z,"c"#
% cat n
cab
% vfind --vdl=n.vdl -p n
...
##==> VDL model for ‘x’ loaded (notell).
##==> VDL model for ‘y’ loaded.
##==> VDL model for ‘z’ loaded (notell).
...
##==> Checking file: "n"
##==>>>> VIRUS POSSIBLE IN FILE: "n"
##==>>>> VIRUS ID: CVDL y
```

```
##==>>>> VIRUS END OFFSET: 3, matched: cab
##==> Number of possible virus infections found in file "n": 1
```

The first notell specification turns off reporting for all subsequent VDLs in the n.vdl file; however, for VDL y only, the notell is overridden by a !”notell” specification, so VDL y will be reported. All three VDLs x, y, and z hit on file n, but only y is reported.

The notell specification can be useful in conjunction with meta VDLs which are discussed in the next section.

11.5 Meta VDLs

Meta VDLs match on the names of other VDL hits, not on the data being scanned. They were described previously in Section 4.5. Meta VDLs reside in a separate search engine and are loaded from a file using the *–vdlm=* option.

Continuing the notell example from the previous section, if we create a meta VDL to match on detection of VDLs x and z then it will match even though x and z are not reported:
```
% cat meta.vdl
:x and z, "CVDL x" and "CVDL z" #
% vfind --vdl=n.vdl --vdlm=meta.vdl -p n
```

```
...
##==> Checking file: "n"
##==>>>> VIRUS POSSIBLE IN FILE: "n"
##==>>>> VIRUS ID: CVDL y
##==>>>> VIRUS END OFFSET: 3, matched: cab
##==>>>> VIRUS POSSIBLE IN FILE: "n"
##==>>>> VIRUS ID: CVDL x and z
##==>>>> VIRUS END OFFSET: 6, matched: CVDL z
##==> Number of possible virus infections found in file "n": 2
```

In conjunction with notell VDLs, meta VDLs can be useful for detection with UAD/SmartScan input that decomposes archives into separate components. Since each component is scanned separately, it is not possible to create a regular VDL which would match across components. Notell VDLs can be created to match individual components, and then a meta VDL can be written to match on the notell hits.

For example:

```
% cat notell.vdl
<"notell">
:password, <"text"> ~"password" #
:image, <"text"> ~".gif", WP1 OR ~".jpeg", WP1 OR ~".bmp", WP1 #
:encrypted zip, <"encrypted zip"> 0-255 #
% cat meta2.vdl
:password and image and encrypted zip,
"CVDL password" AND "CVDL image" AND "CVDL encrypted zip" #
% uad -ssw -sst -M -z e.msg | vfind -ssr --vdl=notell.vdl --vdlm=meta2.vdl -p
...
##==> Checking file: "e.msg" -> "npxormprwo.gif"
##==> Checking file: "e.msg" -> ""
##==> Checking file: "e.msg" -> "Your_complaint.zip"
##==> Checking file: "e.msg" -> "cnxkvl.exe"
##==> Checking file: "e.msg" -> "gchuttf.sys"
##==> Checking file: "e.msg" -> ""
##==>>>> VIRUS POSSIBLE IN FILE: "e.msg" -> ""
##==>>>> VIRUS ID: CVDL password and image and encrypted zip
##==>>>> VIRUS END OFFSET: 54, matched: L encrypted zip
##==> Number of possible virus infections found in file "e.msg": 1
```

The password VDL matches the word ”password” case–insensitively in any text component. The image VDL matches any of three typical image file extensions. The encrypted zip VDL demonstrates matching by file type; it will match the first byte in any component (since 0-255 is the range of all possible byte values) but is restricted to files of type ”encrypted zip”.

Hits on the notell VDLs will not be reported, but the VDL names will be accumulated for scanning by the meta VDL engine. The VDL in meta2.vdl simply requires all three notell VDLs to be present. The e.msg file shown scanned above is an actual e-mail virus sample where the password for the encrypted zip attachment is contained in the GIF image component. Besides the attachments, the text body of e.msg contained the following, which was matched by the password and image notell VDLs:

For security purposes the attached file is password protected.
Password -- <img src="cid:npxormprwo.gif">

11.6 VDL sticky and clear specifications

VDL hits are normally reset to zero after processing a file, but there may be situations where one would want a VDL hit to be persistent. For example, if the components of the e.msg file from the previous example were extracted into separate files and then scanned, the meta VDL would not hit.

This case can be handled using the ”sticky” specification, which makes VDL hits *stick* around for all subsequent scanned files, for use with meta VDLs. Sticky VDL hits can be cleared if necessary using a VDL with the ”clear” specification.

Redoing the previous example, sticky.vdl is the same as notell.vdl except it has the ”sticky” specification; meta2 clear.vdl is the same as meta2.vdl except it has the ”clear” specification. The mail message components have been extracted into separate files in subdirectory e/, which also contains an unrelated file hi.txt:

```
% cat sticky.vdl
<"notell", "sticky">
:password, <"text"> ~"password" #
:image, <"text"> ~".gif", WP1 OR ~ ".jpeg", WP1 OR ".bmp", WP1 #
:encrypted zip, <"encrypted zip"> 0-255 #
% cat meta2_clear.vdl
<"clear">
:password and image and encrypted zip,
"CVDL password" AND "CVDL image" AND "CVDL encrypted zip" #
% ls e/
Your_complaint.zip body.html hi.txt
% uad -ssw -sst -z e/* | vfind -ssr --vdl=sticky.vdl --vdlm=meta2_clear.vdl –p
```

```
...
##==> Checking file: "e/Your_complaint.zip"
##==> Checking file: "e/Your_complaint.zip" -> "cnxkvl.exe"
##==> Checking file: "e/Your_complaint.zip" -> "gchuttf.sys"
##==> Number of possible virus infections found in file "e/Your_complaint.zip": 0
##==> Checking file: "e/body.html"
##==>>>> VIRUS POSSIBLE IN FILE: "e/body.html"
##==>>>> VIRUS ID: CVDL password and image and encrypted zip
##==>>>> VIRUS END OFFSET: 18, matched: L encrypted zip
##==> Number of possible virus infections found in file "e/body.html": 1
##==> Checking file: "e/hi.txt"
##==> Number of possible virus infections found in file "e/hi.txt": 0
```

Note that if the meta VDL did not contain the ”clear” specification it would hit on hi.txt and all subsequent files due to the persistent sticky VDL hits.

11.7 Exercises

Select one spam message that you have received which has a combination of features suitable to be matched by notell and meta VDLs. Write the VDLs and test them on that message and a collection of other clean and spam messages you have received.

Chapter 12

CVDL Syntax Summary

CVDL syntax is described as a set of recursive parsing rules which are applied to “tokens” produced by lexical preprocessing. Preprocessing skips comments and non–literal white space, and expands macros.

We start with some definitions:
hex escape sequence \x or \X followed by two hex digits.
escape sequence \c or hex escape sequence. If character c is n, r, or t this represents new line, carriage–return, or tab, respectively; otherwise it represents character c itself.
INTEGER A decimal or hex integer (e.g. 1234, 0xa9BE, 0Xff) or a single character or escape sequence in single ’quotes’, (e.g. ’a’, ’\n’, ’\x9f’) or an escape sequence with no quotes.
BYTE An INTEGER in the range 0 to 255 inclusive.
DOUBLE A non–negative floating–point decimal value (e.g. 1.234)
STRING Zero or more characters or escape sequences in double ”quotes”.
VDLNAME Following a colon (:), any characters except comma (,) up to a terminating comma forms the VDL name.

12.1 Top–Level CVDL

The top level of CVDL is parsing of a file of VDLs:

```
file:
        empty
        file vdl
        file < meta_list >
        file < >
vdl:
        : VDLNAME , top_or #
        : VDLNAME , < meta_list > top_or #
        : VDLNAME , < > top_or #
meta_list:
        STRING
        ! STRING
        meta_list , STRING
```

The first rule says that a file of VDLs is either empty or is a file followed by a vdl or meta operator. When a VDL file is opened for processing, we initially have nothing (i.e. the file is opened but we did not yet read anything), so the first line of the first rule is always matched immediately.

If the file is not completely empty, the next thing we expect to find is either a VDL or a meta operator such as a file type restriction list (see Chapter 11). Meta operators start with either a string or a negated (!) string, optionally followed by more strings separated by commas. VDLs start with colon (:), followed by the VDL name, comma, optional meta operator, then ‘top or’ (see below), then ‘#’.

Once the first VDL or meta list is read, that matches the file rule, which then continues to apply recursively. For example, if a file contained three VDLs, the file rule above would recursively match over processing time 0, . . ., 3, as follows:

$file_{(0)} = empty$
$file_{(1)} = file_{(0)}\ vdl = empty\ vdl_1$
$file_{(2)} = file_{(1)}\ vdl = empty\ vdl_1\ vdl_2$
$file_{(3)} = file_{(2)}\ vdl = empty\ vdl_1\ vdl_2\ vdl_3$

he syntax description continues with the top–level (i.e. high–level) CVDL operators: OR, XOR, AND, NOT, SIZE, and NAME (see Chapter 8):

```
top_or:
        top_xor
        top_or OR top_xor

top_xor:
        top_and
        top_xor XOR top_and

top_and:
        top_not
        top_and AND top_not

top_not:
        cat
        size
        name
        NOT top_not
        ( top_or )

size
        SIZE RELOP INTEGER
        INTEGER RELOP SIZE

RELOP:
        <       >       !=      ==      <=      >=

name:
        NAME  "~= cat
```

The list of operators above falls through to 'cat' which is described below. RELOP is the list of relational operators used in conjunction with the SIZE operator. Note that all CVDL operators are recognized case–insensitively, so 'AND' may be written as 'and', etc.

12.2 Low–Level CVDL

The basic low–level CVDL operators are described in Chapter 6, and the text and regex operators are described in Chapters 7 and 9:

cat:
    or
    cat , or

or:
    code
    or | code

code:
    data
    (cat)
    @ range_expr , data
    @ range_expr , (cat)
    .* , data
    .* , (cat)
    ABS INTEGER , data
    ABS INTEGER , (cat)
    %f > DOUBLE
    %f > - DOUBLE
    ~R... STRING
    ~R... STRING (trig_and)

range_expr:
    INTEGER
    INTEGER -
    - INTEGER
    INTEGER - INTEGER

The list of operators above depends on ‘data’ and ‘trig and’ which are described below. As an example of parsing using the description above, ’a’, ’b’ | ’c’, ’d’ would result in the parse tree ((’a’ CAT (’b’ | ’c’)) CAT ’d’) where ‘CAT’ represents concatenation (i.e. the comma operator).

Note that ‘range expr’, ‘.*’, and ‘ABS’ may be applied to ‘data’ or ‘(cat)’ corresponding to the discussion of offset groups in Section 6.8.

12.3 CVDL Data

CVDL ‘data’ is defined in terms of several data types including byte expressions, string data, set data, etc. as follows:

data:

    byte_expr
    string_data
    pseudo_data
    set_data
    ^ set_data
    W0
    W1
    WP0
    WP1
    WS0
    WS1
    EOD
    \d+
    repetition_expr
    fuzzy BYTE
    fuzzy STRING

fuzzy:

    FUZZY INTEGER
    FUZZY +- INTEGER
    FUZZY -+ INTEGER
    FUZZY - INTEGER
    FUZZY + INTEGER
    FUZZY - INTEGER + INTEGER
    FUZZY + INTEGER - INTEGER

repetition_expr:

    byte_expr [range_expr]
    string_data [range_expr]
    set_data [range_expr]
    ^ set_data [range_expr]

byte_expr:

    BYTE
    ^ BYTE
    byte_range
    ^ byte_range

byte_range:

    - BYTE
    BYTE -
    BYTE - BYTE

```
set_data:
        { set_list }
set_list:
        set_item
        set_list , set_item
set_item:
        STRING
        byte_expr
        set_data
        ^ set_data
string_data:
        STRING
        ~ STRING
pseudo_data:
        ~~ STRING
        ~W STRING
        ~# STRING
        ~# INTEGER STRING
```

12.4 Regex Trigger Data

As discussed in Section 9.2, regular expressions may include trigger data to improve run–time speed. The syntax for regex trigger data is:

```
trig_and:
        trig_cat
        trig_and AND trig_cat
trig_cat:
        trig_or
        trig_cat , trig_or
trig_or:
        STRING
        ( trig_cat )
        trig_or | STRING
        trig_or | ( trig_cat )
```

12.5 Exercises

Based on some pattern matching operation which is not currently included in CVDL, design a new CVDL operator, specify how it would fit in with the current syntax, and discuss how it could be implemented efficiently.

Chapter 13

Performance and Testing

The three main aspects of performance and testing are: accuracy, speed, and memory usage. Testing for accuracy is straightforward; it requires scanning many clean and infected files to check for false positives (hits on clean files) and false negatives (misses on infected files). If false hits occur, the pattern or VDL must be adjusted, using techniques such as analysis of entropy and serial correlation (Section 3.5), use of high–level AND (Section 8.1) to combine pattern features, use of file–type specific engines (Chapter 4), or use of meta VDLs (Chapter 11).

This chapter covers issues related to speed and memory usage in performance and testing. Sample runs are performed on a relatively slow Sun 333 MHz Ultra 5 system running Solaris 7, using VFind-15.4.5 compiled with the cbayes engine included, and CyberSoft VDLs version 2005-06-06-15-53-58 (13509 VDLs). Although absolute speed results depend on the system processor, most results presented here will reflect relative speed improvements which are directly applicable to any system.

Testing should be done on an unloaded system, not an active server, and timing results should generally show that real time used is approximately equal to the sum of user and system CPU time. For comparison and generalization of results, at least three runs should be performed for each test, and the results averaged. The results for each run should be consistent, i.e. approximately the same. In certain cases, generally involving scanning a large file or large number of files, the operating system will use RAM for I/O caching, and the first run of a test may take much longer than successive runs. In this case one can perform four runs and just use results from the last three.

13.1 Testing Basics

Here we present some basic techniques for performance testing, and examine VFind startup and run–time speed. VFind, like most application software, takes some time to start up and initialize internal data structures before it actually scans any files. We can measure the startup time by not specifying any input files on the command–line, and redirecting input from the null device:

```
% cd examples/perf
% timex vfind < /dev/null |& egrep '^(real|user|sys)'
real        2.86
user        2.18
sys         0.60
% timex vfind < /dev/null |& egrep '^(real|user|sys)'
real        2.86
user        2.16
sys         0.65
% timex vfind < /dev/null |& egrep '^(real|user|sys)'
real        2.85
user        2.09
sys         0.71
```

To calculate the average startup CPU time used, add together all of the user and sys times from above and divide by 3; the result is 2.80, which is consistent with the real times shown.

To check RAM usage, run VFind interactively, suspend it at the file name prompt, then run 'top':

```
% vfind
...
Enter the name of the file to be checked: ^Z
% top
```

PID	USERNAME	THR	PRI	NICE	SIZE	RES	STATE	TIME	CPU	COMMAND
20197	perry	1	50	0	50M	50M	stop	0:02	6.84%	vfind

So 'top' shows that 50 MB of RAM is used.

As an aid for subsequent examples, the '3.sh' script shown below will be used to run VFind three times and display the run times. Using 3.sh to recheck the above example, we obtain just about the same average result of about 2.80 seconds CPU time:

```
% cat 3.sh
#! /bin/sh

(timex "$@" >/dev/null; timex "$@" >/dev/null; timex "$@" >/dev/null) 2>&1 |
nawk ' # note timex can report hours:minutes:seconds, e.g. 1:15:23.20
BEGIN { real = user = sys = r = u = s = n = 0; }
function tconv( t) { m=split(t,a,":"); x=1; y=0;
while( m > 0) { y += x*a[m--]; x *= 60; } return y; }
$1 == "real" { r = tconv($2); real += r; ++n; }
$1 == "user" { u = tconv($2); user += u; }
$1 == "sys" { s = tconv($2); sys += s;
printf( "real = %.2f, user = %.2f, sys = %.2f\n", r, u, s); }
END { printf( "avg real = %.2f, cpu = %.2f\n", real/n, (user + sys)/n);
}'
% ./3.sh vfind < /dev/null
real = 2.85, user = 2.04, sys = 0.74
real = 2.90, user = 2.12, sys = 0.67
real = 2.86, user = 2.19, sys = 0.61
avg real = 2.87, cpu = 2.79
```

The initial startup time shown above, before scanning any files, is actually only part of the total startup time and represents VFind reading VDL and other engine configuration files. A further step of initialization, to compile the patterns for fast/parallel search (see Section 13.2), is required before scanning the first file. vfind-mt, the multithreaded version of VFind, does perform this further step right at the beginning, as we can see by comparing the following run to that above:

```
% vfind-mt < /dev/null |& grep Compil
##==> Compiling scan engines...
% ./3.sh vfind-mt < /dev/null
real = 14.39, user = 13.61, sys = 0.63
real = 14.28, user = 13.60, sys = 0.55
real = 14.19, user = 13.47, sys = 0.60
avg real = 14.29, cpu = 14.15
```

The non–multithreaded version of VFind defers the final step of initialization until just before scanning the first file, so we have to scan at least one file to see that:

```
% cat hi
hi
% ./3.sh vfind hi
real = 14.37, user = 13.38, sys = 0.74
real = 14.16, user = 13.20, sys = 0.75
real = 14.23, user = 13.30, sys = 0.65
avg real = 14.25, cpu = 14.01
% ./3.sh vfind hi hi
real = 14.27, user = 13.46, sys = 0.68
real = 14.12, user = 13.27, sys = 0.71
real = 13.96, user = 13.25, sys = 0.64
avg real = 14.12, cpu = 14.00
```

The total startup CPU time of about 14 seconds is consistent with the vfind-mt run above. Note that the second run above scanned the ‘hi’ file twice in about the same time as scanning it once. So for such a small file, the VFind cpu–time is practically all startup overhead.

For scanning a number of files, the run–time will be much less if VFind is invoked just once to scan all of the files, vs. invoking VFind separately for each file. For example, file ‘rand.dat’ contains1,000,000 bytes of random data and is scanned once, then ten times:

```
% ./3.sh vfind rand.dat
real = 15.56, user = 14.65, sys = 0.76
real = 15.50, user = 14.73, sys = 0.66
real = 15.61, user = 14.63, sys = 0.83
avg real = 15.56, cpu = 15.42
% ./3.sh vfind ‘perl -e ’print "rand.dat " x 10;’‘
real = 28.96, user = 27.91, sys = 0.82
real = 29.07, user = 27.94, sys = 0.83
real = 29.19, user = 27.99, sys = 0.71
avg real = 29.07, cpu = 28.73
```

If VFind was invoked once for each of ten scans of rand.dat, the total CPU time would be 10*15.42 = 154.2 seconds. But as shown above, invoking VFind once to scan rand.dat ten times uses only 28.73 seconds. Considering the 14 second startup overhead, only 14.73 of the 28.73 seconds were used for actual scanning, yielding 1.473 seconds per scan of rand.dat.

13.2 Fast/Parallel Search

VFind uses a fast parallel search engine design, so scanning run–time is mostly independent of the number of VDL rules. A parallel search for all patterns is always performed first, and if that produces no matches then the patterns definitely do not appear in the data; however, if the parallel search does match a pattern, that pattern must be rechecked individually, in a slower serial search mode, since the parallel search does not handle options such as offsets and regular expressions.

Only a partial set of CVDL pattern types can be handled by the fast/parallel search. It does not handle case–insensitivity and wildcard white space, but those features are handled by the *vdlc* engine as discussed in Sections 4.2 and 13.3. All other pattern strings consisting of at least 4 characters, including offset operators, but not including sets, fuzzy, repetition, white space, or regex operators, are handled.

By using patterns which can be included in the parallel search, run–time can be lowered significantly. The following contrived example demonstrates this using “slow.vdl” and “fast.vdl” files containing 1000 copies of a VDL. The slow VDL file uses patterns containing less than 4 characters, which will not be included in the parallel search. The fast VDL file uses patterns containing 6 characters, which will be included in the parallel search:

```
% head -2 slow.vdl
:abc-efg, "abc", "efg" #
:abc-efg, "abc", "efg" #
% wc -l slow.vdl
1000 slow.vdl
% head -2 fast.vdl
:abc-efg, "abcefg" #
:abc-efg, "abcefg" #
% wc -l fast.vdl
1000 fast.vdl
```

Using “slow.vdl” the run–time is about 30 seconds, whereas using “fast.vdl” the run–time is less than 1 second:

```
% ./3.sh vfind --liboff='*' --vdl=slow.vdl --quiet=1 rand.dat
real = 29.58, user = 29.47, sys = 0.04
real = 30.04, user = 29.72, sys = 0.05
real = 29.85, user = 29.56, sys = 0.05
avg real = 29.82, cpu = 29.63
% ./3.sh vfind --liboff='*' --vdl=fast.vdl --quiet=1 rand.dat
real = 0.18, user = 0.15, sys = 0.00
real = 0.17, user = 0.11, sys = 0.04
real = 0.17, user = 0.13, sys = 0.03
avg real = 0.17, cpu = 0.15
```

In this example, using the slow VDLs, rand.dat was actually scanned 1000 times, once for each VDL. Using the fast VDLs, rand.dat was only scanned once.

Note that there are two basic ways to test an individual VDL: running the VDL on a large number of files, or running a large number of copies of the VDL on a small number of files. The tests shown here use the second method, with 1000 copies of the VDL run on one data file. In production, one should not use multiple copies of the same VDL; that is only done for testing to make the run–time large enough to be measured accurately. For production VDLs, the VFind *-d,–dup-check* option can be used to check for duplicate VDL names and definitions (see Section 2.1).

As mentioned at the beginning of this section, when the parallel search engine detects a match, the VDL must be run in a slower serial mode, rescanning the data, to check for an exact match. If the data really does not match the pattern, then rescanning represents a run–time penalty which we would like to avoid. To avoid excessive rescanning, it is important that the parallel search engine includes strings which will rarely not hit when rescanned.

The parallel search knows the difference between AND and concatenation, and will not trigger a rescan if concatenated matches occur in the wrong order. For example, take the last 32 bytes of rand.dat:

```
% od -tx1 rand.dat | tail -3
3641040 36 12 7c 73 dc 37 5d 86 9a bc 20 32 7b 4c 50 ec
3641060 12 e9 a9 ad 56 fa d2 98 25 a0 30 3c 8b 4b 39 f1
3641100
```

and consider VDLs which specify those bytes with the last 16 bytes first:

```
% cat s2.vdl
:vand,
"\x12\xe9\xa9\xad\x56\xfa\xd2\x98\x25\xa0\x30\x3c\x8b\x4b\x39\xf1"
AND
"\x36\x12\x7c\x73\xdc\x37\x5d\x86\x9a\xbc\x20\x32\x7b\x4c\x50\xec"
AND
size == 1234 #
% cat f2.vdl
:vcat,
"\x12\xe9\xa9\xad\x56\xfa\xd2\x98\x25\xa0\x30\x3c\x8b\x4b\x39\xf1",
"\x36\x12\x7c\x73\xdc\x37\x5d\x86\x9a\xbc\x20\x32\x7b\x4c\x50\xec"
AND
size == 1234 #
```

Each VDL has a “size” operator included simply to force it to not match. VDL “vand” in file s2.vdl uses AND for the two 16–byte patterns; that part of the VDL will match in the parallel search, causing the data to be rescanned. But VDL “vcat” in file f2.vdl uses concatenation (,) of the patterns, and since they are in the wrong order, they will not match in the parallel search, and the data will not be rescanned.

The performance of the VDLs above is tested using file “slow2.vdl” containing 1000 copies of VDL vand, and file “fast2.vdl” containing 1000 copies of VDL vcat:

```
% ./3.sh vfind --liboff='*' --vdl=slow2.vdl --quiet=1 rand.dat
real = 59.14, user = 58.91, sys = 0.06
```



```
real = 59.11, user = 58.92, sys = 0.05
real = 59.11, user = 58.89, sys = 0.07
avg real = 59.12, cpu = 58.97
% ./3.sh vfind --liboff='*' --vdl=fast2.vdl --quiet=1 rand.dat
real = 0.21, user = 0.13, sys = 0.06
real = 0.20, user = 0.18, sys = 0.02
real = 0.20, user = 0.15, sys = 0.05
avg real = 0.20, cpu = 0.20
```

Using “fast2.vdl” the run–time is almost zero, whereas using “slow2.vdl” the run–time is close to 1 minute, since the parallel search triggers 1000 rescans of the data.

In cases where triggering of rescans by the parallel search is unavoidable, one should at least try to make the rescan as fast as possible. One way to do that in patterns which use the AND or OR operators is to put the fastest operations first. CVDL AND and OR operators use short–circuit evaluation, similar to the logical && and || operators in the C programming language. That is, if the left side of an AND is false, the right side is not evaluated. And if the left side of an OR is true, the right side is not evaluated.

For example, VDL vand from “s2.vdl” above uses a size operator as the last part of the VDL. The size operator is fast since it just checks the file size and does not scan any data. Thus, the VDL should run faster in a rescan if the size operator is specified first. VDL “vand-b” in file s2b.vdl is the same as VDL vand except for listing the size operator first instead of last:

```
% cat s2b.vdl
:vand-b,
size == 1234
AND
"\x12\xe9\xa9\xad\x56\xfa\xd2\x98\x25\xa0\x30\x3c\x8b\x4b\x39\xf1"
AND
"\x36\x12\x7c\x73\xdc\x37\x5d\x86\x9a\xbc\x20\x32\x7b\x4c\x50\xec"#
```

File “slow2b.vdl” contains 1000 copies of VDL vand-b, and its performance is about the same as “fast2.vdl”:

```
% ./3.sh vfind --liboff='*' --vdl=slow2b.vdl --quiet=1 rand.dat
real = 0.21, user = 0.14, sys = 0.05
real = 0.20, user = 0.15, sys = 0.04
real = 0.20, user = 0.13, sys = 0.07
avg real = 0.20, cpu = 0.19
```

13.3 Case–Insensitive Fast/Parallel Search

As discussed in Section 4.2, although the *vdl* parallel search engine does not handle case–insensitivity and wildcard white space, a separate *vdlc* parallel search engine is provided to perform fast scanning for VDLs consisting mostly of those types of patterns.

To illustrate the performance gain using the *vdlc* engine, a set of case–insensitive word pattern VDLs were first created from the online /usr/dict/words dictionary:

```
% cat mkwords.sh
#!/bin/sh
#
# make case-insensitive word VDLs
# run < /usr/dict/words > words.vdl
grep '^.....*' | # at least 12 letters
sed -e "s/.*/:&,~\"&\"#/"
% ./mkwords.sh < /usr/dict/words > words.vdl
% wc -l words.vdl
1064 words.vdl
% head -3 words.vdl
:abovementioned,~"abovementioned"#
:absentminded,~"absentminded"#
:accelerometer,~"accelerometer"#
% tail -3 words.vdl
:wholehearted,~"wholehearted"#
:Williamsburg,~"Williamsburg"#
:Winnipesaukee,~"Winnipesaukee"#
```

Now compare performance between the *vdl* and *vdlc* engines using these VDLs:
% ./3.sh vfind --liboff='*' --vdl=words.vdl --quiet=1 rand.dat
real = 55.75, user = 55.57, sys = 0.04
real = 54.73, user = 54.56, sys = 0.05
real = 54.71, user = 54.52, sys = 0.06
avg real = 55.06, cpu = 54.93
% ./3.sh vfind --liboff='*' --vdlc=words.vdl --quiet=1 rand.dat
real = 0.33, user = 0.26, sys = 0.06
real = 0.32, user = 0.28, sys = 0.04
real = 0.32, user = 0.23, sys = 0.08
avg real = 0.32, cpu = 0.32

Using *–vdl=* to load the VDLs, none can be handled by the parallel search engine, so all are run in the slower serial mode, and the run–time is almost 1 minute. But using *–vdlc=* to load the VDLs, all are handled by the case–insensitive parallel search, so none are run in the slower serial mode, and the run–time is less than 1 second.

13.4 Exercises

1. Collect a set of 100 clean files of various types for use in performance testing. Include some Files of each of the following types: MS/EXE, OLE, text, and unknown (e.g. random or encrypted data). Put the file names in a '100.list' file. Run VFind with input redirected from 100.list and the *–sst,–smart-scan-types* option to check that it is correctly detecting the file types and not producing false hits on the collection.
2. Extract a set of 100 VDLs from the current CyberSoft VDL distribution exe, ole, scanall, other, and text VDL files. Put the set into a file '100.vdl', being careful to set the same file–type restrictions for each type of VDL in each section. Run VFind using only your set of VDLs on your collection of clean files to check for any problems.
3. Concatenate 100.list with itself to produce file '200.list'; concatenate that with itself to produce '400.list'; etc. up to '3200.list'. Similarly, use concatenation starting with 100.vdl to produce VDL files'200.vdl', '400.vdl', etc. up to '3200.vdl'. These files will be used in subsequent exercises for performance testing.
4. For each of the VDL files from the previous exercise, measure the performance of VFind for each of the set of list files. Produce tables and a plot of the results.
5. Assume that VFind scan time can be modeled as:
   i. $t = t_p N_p + t_f N_f$ (13.1)
   where t_p and t_f are CPU time per pattern (VDL) and per file scanned respectively, N_p is the number of patterns, and N_f is the number of files. Using results from the previous exercise, determine linear least–square–error estimates of t_p and t_f.

Bibliography

[1] P. Radatti, “Pattern analysis for computer security.” http://cybersoft.com/, whitepapers.

[2] CyberSoft Operating Corporation. http://cybersoft.com/.

[3] SafeInternetEmail. http://safeinternetemail.com/.

[4] CyberSoft, “Smartscan.” http://cybersoft.com/, whitepapers, programming examples.

[5] D. E. Knuth, *The Art of Computer Programming, Second Edition, Volume 2, Seminumerical Algorithms*. Addison–Wesley, 1981.

[6] European Institute for Computer Antivirus Research. http://www.eicar.com/.

[7] Java. http://java.sun.com/.

[8] RFC 1321 - The MD5 Message-Digest Algorithm. ftp://ftp.rfc-editor.org/in-notes/rfc1321.txt.

[9] OpenSSL Project. http://www.openssl.org/.

[10] W. Stallings, *Cryptography and Network Security, Principles and Practice, Third Edition*. Prentice Hall, 2003.

[11] C. Kaufman, R. Perlman, and M. Speciner, *Network Security, Private Communication in a Public World*. Prentice Hall, 1995.

[12] B. Schneier, *Applied Cryptography, Second Edition*. Wiley, 1996.

[13] http://www.opengroup.org/, ed., *The Open Group Base Specifications Issue 6*. IEEE Std 1003.1, 2004.

[14] J. E. F. Friedl, *Mastering Regular Expressions, Second Edition*. O’Reilly, 2002.