BigGrep: Fast Binary Sequence Searching

Charles Hines
mc-contact@cert.org
CERT® Coordination Center, Software Engineering Institute
Publication CERTCC-2012-37
DM-0001480

October 26, 2012

1. Executive Summary

BigGrep is a set of tools written by the CERT/CC Malicious Code group designed to produce indexes of binary data and search these indices in a manner that attempts to balance a few key factors:

- Minimize disk usage for indexes
- Minimize time spent repeatedly searching a large quantity of data

This report describes the impetus for and implementation of BigGrep (Section 2) and gives general information on usage (Section 3), including what it is good for and what it is not good for (to help you decide if it might be a good fit for potential applications for your area). It then concludes with a detailed example demonstrating our successful real world usage of BigGrep for large scale malware analysis and R&D activities (Section 4).

An appendix is also included describing additional history, early implementation experiments, etc.

2. Impetus & Implementation Notes

The CERT/CC Malicious Code group has a significantly sized[1] stockpile of malware that we call the Artifact Catalog (often abbreviated AC or Catalog). Through the course of our analysis and research activities we often need to search through the Catalog to find files with common characteristics (such as embedded strings or the binary values of assembly instructions that are part of a function).

2.1. Searching the "old way"

Traditionally searching the Catalog was done using parallel[2] invocations of a tool called YARA[3], or prior to that using an in house developed binary grep tool called 'bgrep' (took ASCII encoded byte values and searched for them via the Boyer-Moore-Horspool[4] algorithm) or simply GNU grep using escaped binary characters in the search term.

However, as the collection grew this method became more and more infeasible as the searches would literally take several days to complete. To add to the pain, we began to do more malware family analysis, which meant an increased desire to do more (and faster) searches.

2.2. N-gram[5] inverted indexes

To briefly explain the general concept of an n-gram inverted index, let us assume that a file 'A' contains the string "foobar"[6]. This string breaks down to the following 3-grams (determined by sliding a window of length 3 over the contents one byte at a time): "foo", "oob", "oba", "bar". To create a traditional inverted index of these, we would record for each 3-gram the file it occurred in (in this case 'A') and the position in the file.

To search an index for a particular string you would convert the search term to 3-grams in the same way, and then inspect the index entry for each 3-gram occurrence and for sequential order in each file occurrence to determine all of the exact matches of the search term.

[1] Approximately 38 million MD5 unique samples in 14TB of disk space in 10/2012

[2] Via GNU Parallel: http://www.gnu.org/software/parallel/

[3] http://code.google.com/p/yara-project/

[4] http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore%E2%80%93Horspool_algorithm

[5] http://www.vldb2005.org/program/paper/wed/p325-kim.pdf

[6] This example uses a readable string for simplicity, but our implementation was designed for and works on arbitrary byte values

2.3. Additional constraints

In addition to wanting to be able to quickly search the Catalog, we were also faced with another issue: disk space. Seems like an easy problem to solve, in that you can just buy more disks, but other factors such as rack space, power, and cooling also need to be considered.

So to best work within our current hardware setup, we decided we needed to be sure that our indexes were kept as small as possible in addition to being fast to search, two conditions that are often mutually exclusive.

2.4. Initial design concepts

To help satisfy the index size constraints, we decided to eliminate one of the key features in a typical inverted index, in that we would only record the presence of a particular n-gram instance in a file, instead of recording the positions of every instance of that n-gram in every file indexed.

This would result in a probabilistic search that would return all the candidate files that could possibly contain the exact search term(s) - i.e. they contain all of the n-grams of the search terms. Then a secondary verification step would need to be done to ensure that they were present in the exact order in the candidate files to get the final search results list.

This technique would allow us to save (considerably) on index space but theoretically still search in a reasonable timeframe by only having to search (verify) these candidate files instead of the entire Catalog. The overlapping nature of the n-gram generation means that for most real world search terms, that the number of false positives should be "acceptable[7]".

To further save space, we decided to assign sequentially increasing identifiers (4 byte integer) to the files in an index, instead of our normal method of identifying files by either MD5 (16 bytes) or SHA-256 (32 bytes) hash values. Since we only have on the order of 40 million files so far and aren't likely to hit the ~4 billion[8] file mark anytime soon[9], we decided that the n-gram to file mappings would be better served by using a 32 bit integer that mapped to the file name or hash value externally to save on disk space. Integer list compression techniques were also discussed (described in the next section).

We also began with the assumption that 4-grams would likely be a better choice than 3-grams for a number of reasons, including:

[7] A subjective term that will vary by individual and search terms

[8] The maximum value of an unsigned 4 byte integer is 4294967295

[9] If the catalog doubles every year, we'd still have ~7 more years until hitting that 4 billion mark.

- Faster indexing and searching code due to being able to easily use 32 bit instructions in the implementation[10].
- Larger n-gram size equates to fewer false positives.

Those assumptions were true, however the more sparse qualities of the 4-gram space vs 3-gram space led to much larger indexes than expected, so in the end we decided to use 3-grams internally (see Appendix for more details). The BigGrep code is able to generate either 3 or 4-gram based indices (and it uses 32 bit instructions to do this internally either way, for speed), so you can choose for yourself which is more appropriate.

2.4.1. Custom index format

Discussions on our custom index file format also included talks of various compression schemes, possibility of using bitfields instead of integer file ids, etc. In the end we decided that our most likely benefits were going to come from storing our (translated integer) file id lists as delta lists (values stored as deltas from the previous value) encoded using a Patched Frame of Reference[11] (PFOR) encoding, with some Variable Byte[12] (VarByte) encoding in places.

We also quickly realized that we were going to need "hints" on where to quickly jump to various points in our index files, since the data would be variably sized. We decided that a decent approach would be an initial table indicating where every 256 n-grams began in the index file. This would allow us to quickly skip to within 255 entries of the n-gram we were looking for, further aided by having the first value of each entry contain the size of the compressed data for that entry so we would know how to quickly skip over each n-gram of data that we weren't searching for.

The 3rd component as mentioned above was the (integer) file id to file name mapping. Our files are typically stored in a directory hierarchy named by their MD5 or SHA-256 hash values so mapping to a file name also gives us mapping to one of those alternate identifiers at no extra cost. Additionally, small bits of metadata can be optionally stored along with the file names in this mapping, such as another hash value, file size, alternate known names, etc.

2.5. Implementation

The current implementation is briefly described here. Refer to the Appendix for additional information on early experiments and implementations if you are interested.

[10] Our work is conducted primarily on x86_64 platforms, so instructions that operate on 32 bit and 64 bit values are preferred

[11] http://oai.cwi.nl/oai/asset/15564/15564B.pdf

[12] http://nlp.stanford.edu/IR-book/html/htmledition/variable-byte-codes-1.html

2.5.1.The indexer

The indexing code (bgindex) is written in POSIX style C++, designed for speed as we didn't want to take too long in generating indexes for our existing malware Catalog. The code is relatively simple, and has only a few dependencies beyond the base C++ STL, namely POCO[13] and Boost[14].

The indexing code has multithreaded components for the shingling (n-gram generation) and the compression/write portions in an effort to interleave I/O and CPU usage effectively. The files are memory mapped for efficiency of access as well during shingling.

The index generation is all done in RAM, so care must be taken with the amount of input. The general rule of thumb is to not exceed ¼ of the available RAM with the total size of a batch of input files for a particular index (as the number of possible unique n-grams extracted from a file is roughly equivalent to the size of the file, and both 3 and 4 grams are treated as 32 bit integers internally).

This means you will likely be building multiple indexes, depending on the size of your dataset. This is not a drawback, however, as multiple batches/indexes turns out to have other benefits, such as being able to do both index generation and queries in parallel (potentially amongst multiple machines) easily.

A Loser Tree[15] implementation of a multiway merge sort feeds lists of file ids to the Compression and Write threads for PFOR and VarByte compression an insertion into the index file proper.

2.5.1.1. The index file

A single index file is created for each batch of files passed to the indexer, named "<index_name>.bgi". The index file internally consists of a header (fixed size), a hints list (fixed size based on value of 'n'), the n-gram data itself (variably sized), and finally the file id map and metadata (variably sized, ASCII text list).

The code can handle an 'n' of either 3 or 4 (quite likely small tweaks would be needed to handle outside that range). As mentioned earlier we initially assumed that 4 would be the better choice, as far as false positives are concerned (and early experiments confirmed that). However, the 3-gram based indexes are significantly smaller, on average about 40% of input size (the above mentioned ~14TB of indexed files takes approx. 5.9TB of index) versus the ~2.4x the input size on average that we saw with our initial 4-gram indexes. So the tradeoffs are higher false positive rates with 3-grams (1 order of magnitude higher than 4-grams, typically).

[13] http://pocoproject.org/

[14] http://www.boost.org/

[15] Essentially a Tournament Tree where the lower number is propagated up. See Donald E Knuth, "The Art of Computer Programming", Vol3, Sections 5.3.3 and 5.4.1

Experimental usage with real world searches showed us that the increased false positives of the 3-grams was still acceptable, especially in light of our disk space shortage issues at the time and the substantial savings with 3-grams.

2.5.2.The search code

The search code is a mixture of Python (bgsearch.py) & C++ (bgparse). Search is performed using the bgsearch.py Python script, which does some up front processing on the search terms (such as converting ASCII encoded byte values to actual bytes, or ASCII strings to bytes, or ASCII strings to bytes representing UTF-16 encoded ASCII), then it in turn makes several parallel invocations of the bgparse C++ code on the index files.

The bgparse code is responsible for knowing the internal structure of the index files and how to effectively and efficiently parse them (hence the name). It shares some common code with the indexer for this purpose.

The result of the bgsearch.py run will be the list of candidate files that could possibly contain the exact search term(s) desired.

2.5.3.The verification code

An additional program for verification of search results is included, called bgverify. It too is written in C++ (for speed), and it uses Boyer-Moore-Horspool[16] fast search techniques for doing the exact verification that the original search terms exist in the results. The best way to use it is by passing a flag to the bgsearch.py code to have it invoke it for you (also in parallel invocations for speed).

Alternately, you may use YARA or some other method of verifying the candidate search results which may have other benefits (such as looking at specific places in the files before declaring a match). We provide bgverify as a simplistic verification method which in many instances will likely suffice.

2.5.4.Miscellaneous scripts

Some helper/utility scripts are available as well, primarily coded in Python. One in particular is ‘extract-or-replace-fileids.py’, which can be used to make a copy of the file id map data at the end of an index, which is ASCII text and easily manipulated or edited (for example, to add additional metadata to entries or change paths to files if needed), and then replace the file id map data in the index.

[16]It evolved from the in house ‘bgrep’ tool mentioned elsewhere in the paper.

3. Usage

Usage of the BigGrep code is simple, assuming you are comfortable with using UNIX/Linux command line tools.

3.1. Creating Indexes

Let us start with examining the help output of bgindex:

```
$ ./bgindex -h

Usage:

bgindex [OPTS]

  bgindex reads a list of files on stdin to process, produces an N-gram
 inverted index (3 files)

  OPTS can be:

    -n, --ngram # Define N for the N-gram (3 or 4, 3 is default)
    -b, --blocksize #    PFOR encoding blocksize (multiple of 8, default 16)
    -e, --exceptions #   PFOR encoding max exceptions per block (default 2)
    -m, --minimum #      PFOR encoding minimum number of entries to consider
 PFOR (default 4)
    -p, --prefix STR     A prefix for the index file(s) (directory and/or
 partial filename)
    -s, --stats FILE     A file to save some statistics to (not implemented
 yet)
    -S, --sthreads #     Number of threads to use for shingling (default 4)
    -C, --cthreads #     Number of threads to use for compression (default 8)
    -v, --verbose show some additional info while working
    -d, --debug   show more diagnostic information
    -h, --help    show this help
```

So, in order to create index files using bgindex, you first need to create a list of the input files to be placed in this particular index (along with optional metadata, if desired). The format of these files is a simple CSV format, where each line looks like:

```
Path_to_input_file,key=val,key2=val2,key3=val3
```

Everything after the path for the file to be indexed is optional and arbitrary key value pairs (as many as desired). We use it to provide some extra info back with every query, such as the MD5 sum for each file, the size of the file, a generic file type identifier (like “EXE” or “PDF”), etc.

Here’s an example snippet:

```
/data/sakima-
  files/ef/efcb/efcb09cf1f463231c43f1e7fc1e2d153b52ba741daf680fe32580b99a0d24a9
  b,md5=75c4cb03a427ac2c5a2cea1c08543bf1,size=661737,known=AutoIT,type=EXE
/data/sakima-
  files/f2/f299/f299fe28f89872c8e3044b1a083bc5d1c2e36efce95d42fff14947cc5fe76d8
  e,md5=23713bd691b389696c3664ceade02f26,size=5517997,known=Besverit:UPX,type=E
  XE
```

Save that list off to a file, say called 'batch_0001.txt' and assuming the default values of 'n', block size, number of exceptions, etc are acceptable, you just need to pick a name for the index file (in this example "index_0001") and have bgindex read the batch file from standard input:

```
bgindex –p ../indexes/index_0001 < batch_0001.txt
```

This will produce the file index_0001.bgi (3-gram based by default). You can specify a path as part of the index name prefix if desired (as shown).

3.1.1.Optimizing index creation

While a random collection of binary files should produce a decently compressed 3-gram index, if you attempt some ordering techniques for your batches such that suspected similar files are in the same indexes and close to one another in the order, then there is a good chance that you'll be able to drastically reduce the sizes of the resulting index files.

We have had a lot of success with a relatively simple ordering scheme. We take the list of files that we need to create indexes out of, and we sort by the following (in the order presented):

- Known "similar" files (same malware "family", for instance)
- File type categories (EXE, PDF, DOC, etc)
- File size

You can get more advanced, for instance you could use composite section hashes and common sets of section hashes, common sets of section names, etc. But we found that more complex ordering resulted in a lot more work for very little additional gain.

We've already mentioned input batch size to index creation being limited to approximately ¼ of the RAM on the box you are working on, which means that you'll likely end up with multiple indexes (for example, in our usage we process the resulting ordered list through a program that splits up the batches to a given size). This allows for parallelization of index creation over multiple machines, and also benefits queries by allowing those to be easily parallelized on a single host, and could also be extended out to multiple machines.

3.2. Searching Indexes

Searching the indexes using bgsearch.py is simple. Again, let us start with the help output:

```
$ ./bgsearch.py -h

Usage: bgsearch.py [options] term term term...

Options:

  -h, --help            show this help message and exit
  -a ASCII, --ascii=ASCII
                        ascii string search term
  -b BINARY, --binary=BINARY
                        binary hexadecimal string search term
  -u UNICODE, --unicode=UNICODE
                        unicode string search term
  -d DIRECTORY, --directory=DIRECTORY
                        directory to look for .bgi files in
  -r, --recursive       recurse down the directories looking for .bgi files
  -M, --no-metadata     do not show metadata associated with each result, if
                        available
  -v, --verify          invoke bgverify on candidate answers
  -l LIMIT, --limit=LIMIT
                        do not verify above this number of candidates (default
                        15000)
  -n NUMPROCS, --numprocs=NUMPROCS
                        number of simultaneous .bgi files to search (default
                        12)
  -V, --verbose         verbose output
  -D, --debug           diagnostic output
```

Note that you can specify multiple search terms, and if you don't specify the -a, -b, or -u parameters, then the code attempts to guess whether or not you are looking for binary or ASCII data based on the characters in the query term. So a typical usage example (searching for a binary sequence) might look like:

```
$ ./bgsearch.py -d ../indexes
  75E483FB5A7E07B8CCCCCCCCEB1183FB507E0433C0EB0883FB467E0583C8FFFFD0E8
```

This would produce (on stdout) a list of candidate files (those that could possibly contain the sequence), as well as a running tally of how many candidates it has found thus far and the percentage of the index files that it has searched through to find them (this output is sent to stderr, so you can see it even if redirecting the output of the command to a file).

Note that to stop the search before it is finished you may hit ctrl-c or ctrl-\ (which will respectively send a SIGTERM or SIGQUIT to the underlying processes).

3.2.1. Verifying searches

If you wish to verify that the results contain the exact search term(s) you are looking for, you have a couple of choices. For simple verification, pass the '-v' option to the bgsearch.py command line (although you probably don't want to do that at first, just in case the query returns a large number of candidates). To perform more complex verification (for instance, maybe one of the search terms needs to be located some number of "random bytes" before the other, or located at the entry point), you are better off writing a YARA signature, and running that on the list of candidate files produced by the bgsearch.py command.

You can combine those two options as well, to further reduce the number of candidate files that are passed on to the second verification.

3.2.2. What works well

Binary searches of length 4 or more that aren't really common patterns will produce the best results (fewest false positives). And generally, the longer the search terms, the better (up to the point where it is too specific and you may miss minor variants, of course). Note that longer search terms will increase your search time, but that shouldn't be noticeable for most useful searches.

3.2.3. What doesn't work well

ASCII strings that are too short or made of all "common" words are likely to have a large degree of false positives. For instance, the word "program" appears in the DOS stub of nearly all EXE files, so if your search term contains that word, it essentially does nothing to trim down the list of candidate files (except for the n-gram overlap at the ends, possibly).

Unicode (UTF-16) encoded ASCII strings (of any length) will also quite likely result in a larger number of false positives. This is primarily due to the fact that the encoding contains a '0' as every other character, eliminating much of the benefit of the n-gram shingling (overlap) in reducing the candidate result set.

Note that in many instances both of those above situations (for our dataset) produce enough false positives that verification could take several hours (which while better than taking 5-6 days to search the Artifact Catalog the "old way", we often view as unacceptable performance now).

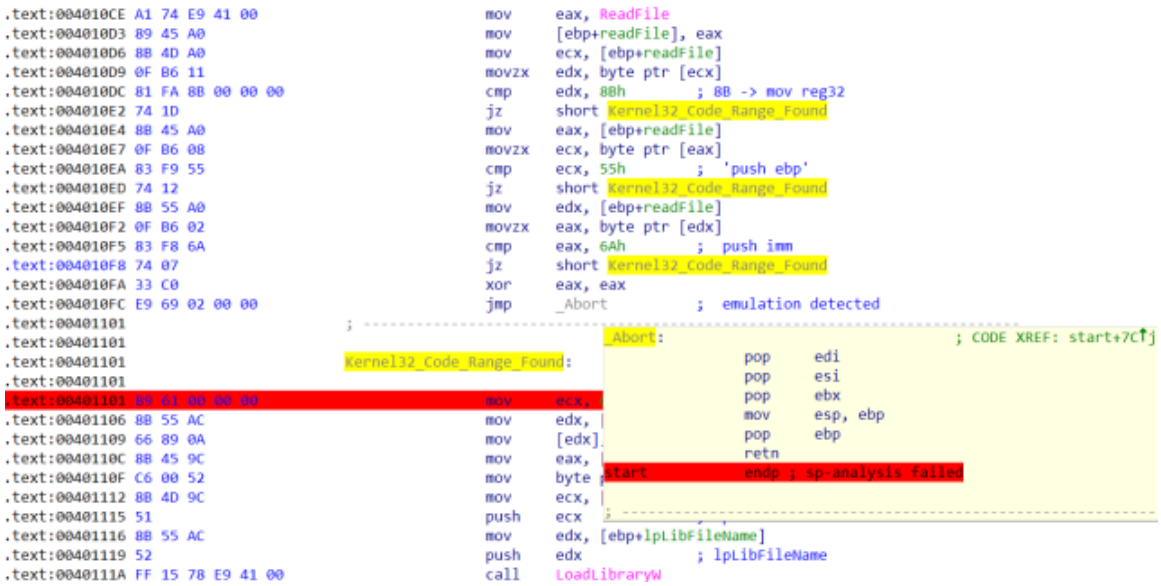
If you find yourself needing to do a lot of string searches like this, you might be better off with putting just the string data in a dedicated string search tool such as Apache Lucene[17] and only using BigGrep for actual binary searches.

[17] http://lucene.apache.org/

4. An example of "real world" usage

Here is an example of just how one might use an iterative process of fast searches and examining candidate results to quickly find what you are looking for in the Artifact Catalog.

On July 3rd, 2012, Symantec's blog had a post[18] describing a particular anti-emulation technique seen in recent Zbot samples, where the code was examining at the implementation of the Windows ReadFile API in memory for certain opcode bytes to be present. For convenience, the picture of the code they provide in that blog entry is reproduced here:



Looking at that code, the majority of the checking code is address independent (ignoring the address of ReadFile it just has some minor relative conditional jumps). So if we skip that first instruction that saves the address of ReadFile into eax and use the rest of the bytes until hitting 'jmp _Abort' we have a potentially "good" search string:

```
8945A0884DA00FB61181FA8B000000741D8845A00FB60883F95574128B55A00FB60283F86A7407
  33C0
```

So, let's search for that (without verification initially):

```
./bgsearch.py -d ../indexes
  8945A0884DA00FB61181FA8B000000741D8845A00FB60883F95574128B55A00FB60283F86A740
  733C0
```

[18] http://www.symantec.com/connect/blogs/relentless-zbot-and-anti-emulations

This search returned 289 candidate files that didn't show up in any of our "knowns[19]" and that were mostly PE files, although there were a few marked as "Data" (which means that the 'file' command returned a string that didn't fall into any of the known executable formats, or compressed, or document types, but had the word "data" or "octet" in it).

So that sounds like a good starting point. Let us try it again with verification (same command, just add '-v' to the options). Unfortunately that came back with 0 confirmations. Why might that be? Several possible reasons jump to mind immediately:

- We could actually not have any of these samples in our Catalog.
- That code could actually be hidden behind layers of packing (the article doesn't really say).
- Different compiled versions could use different registers, different offset for stack variable storage, variants of the conditional branch types, or doing the comparisons in a different order (any of which can result in changing the byte values of the opcodes/operands and/or the control flow, changing the order of the bytes).

So, given those potential reasons, what could we change in our query that might help? Well, out of those reasons we can really only address part of the last one easily…so we could break it up between the conditional jumps and just look for the various checks, giving us multiple search terms:

```
8945A0884DA00FB61181FA8B000000 8845A00FB60883F955 8B55A00FB60283F86A 33C0E9
```

That could introduce additional false positives (both from a "has all the n-grams but not contiguous" and in the "those 4 snippets are all in there but not connected to each other at all" sense). But let's try it anyways…the search gives us 354 candidates, so the next step is to verify them…and unfortunately this still produces no verified results.

Is there anything else we can try? Well, we could try searching for just the 3 comparison instructions:

```
81FA8B000000 83F955 83F86A
```

Now this is quite likely going to get us into the realm of high false positives, but we'll try it anyways. It returned 11,349 candidates. Which is fairly high, but we can still verify that amount relatively quickly. So let's do that and save the output off to a file since the list may still be fairly long:

```
./bgsearch.py -d ../indexes -v 81FA8B000000 83F955 83F86A >
 /tmp/zbot_anti_emu_search_cmps_only.txt
```

[19] Our name for our internal list of malware families we have examined.

This time we actually get some verified results, 5136 of them. This of course means that those 3 search terms are all present, but it doesn't mean that they are near enough to one another to be related in the same way as in the original code. So we are going to need to do some additional verification…and there are enough that "visual inspection" probably isn't the right answer here.

This is where YARA could come in handy to verify using some constrained wildcards perhaps to make sure the 3 search terms were found within a "short" distance of one another and potentially in different orders. Here's a YARA signature for something along those lines:

```
rule zbot_antiemu_cmps
{
  strings:
    $s1 = { 81FA8B000000 [1-10] 83F955 [1-10] 83F86A }
    $s2 = { 81FA8B000000 [1-10] 83F86A [1-10] 83F955 }
    $s3 = { 83F955 [1-10] 81FA8B000000 [1-10] 83F86A }
    $s4 = { 83F86A [1-10] 81FA8B000000 [1-10] 83F955 }
    $s6 = { 83F86A [1-10] 83F955 [1-10] 81FA8B000000 }
    $s7 = { 83F955 [1-10] 83F86A [1-10] 81FA8B000000 }

  condition:
    any of them
}
```

So now we run our 5136 candidates through this check using YARA, and this matches 88 of them. We are now looking much more likely to be the code in question.

The time has come for some visual inspection. So, bringing one of the samples up in IDA and searching for the 6 bytes of the first comparison opcode in our search reveals that this is indeed the code we were looking for:

```
.text:004010B7 A1 88 E9 41 00                       mov     eax, ReadFile
.text:004010BC 89 45 A4                             mov     [ebp+readFileAddr], eax
.text:004010BF 8B 4D A4                             mov     ecx, [ebp+readFileAddr]
.text:004010C2 0F B6 11                             movzx   edx, byte ptr [ecx]
.text:004010C5 81 FA 8B 00 00 00                    cmp     edx, 8Bh
.text:004010CB 74 1D                                jz      short CorrectOpcodeFound
.text:004010CB
.text:004010CD 8B 45 A4                             mov     eax, [ebp+readFileAddr]
.text:004010D0 0F B6 08                             movzx   ecx, byte ptr [eax]
.text:004010D3 83 F9 55                             cmp     ecx, 55h
.text:004010D6 74 12                                jz      short CorrectOpcodeFound
.text:004010D6
.text:004010D8 8B 55 A4                             mov     edx, [ebp+readFileAddr]
.text:004010DB 0F B6 02                             movzx   eax, byte ptr [edx]
.text:004010DE 83 F8 6A                             cmp     eax, 6Ah
.text:004010E1 74 07                                jz      short CorrectOpcodeFound
.text:004010E1
.text:004010E3 33 C0                                xor     eax, eax
.text:004010E5 E9 64 02 00 00                       jmp     _Abort          ; emulation detected
.text:004010E5
.text:004010EA                          ; ---------------------------------------------------------------
.text:004010EA
.text:004010EA                          CorrectOpcodeFound:                 ; CODE XREF: start+5B↑j
.text:004010EA                                                              ; start+66↑j ...
.text:004010EA B9 61 00 00 00                       mov     ecx, 61h
.text:004010EF 8B 55 B0                             mov     edx, [ebp+lpLibFileName]
.text:004010F2 66 89 0A                             mov     [edx], cx
.text:004010F5 8B 45 A0                             mov     eax, [ebp+lpProcName]
.text:004010F8 C6 00 52                             mov     byte ptr [eax], 52h
.text:004010FB 8B 4D A0                             mov     ecx, [ebp+lpProcName]
```

Inspection reveals that the primary reason we didn't find it with our initial BigGrep search was due to a different value for the offset of the stack variable used to store the address of ReadFile. This changed the operands (highlighted in yellow) of several of the mov operations (recall that this was one of our initial "why might this not have worked" statements). If we had started with that assumption instead, we could have gone with this search:

```
0FB61181FA8B000000741D 0FB60883F9557412 0FB60283F86A740733C0
```

That would have hit 459 candidates with 34 verified. And this updated YARA signature verifies that all 34 of them are indeed the code we were looking for:

```
rule zbot_readfile_anti_emu
{
  strings:
    $s1 = { A1 [4] 89 45 ?? 8B 4D ?? 0F B6 11 81 FA 8B 00 00 00 74 1D 8B 45 ??
 0F B6 08 83 F9 55 74 12 8B 55 ?? 0F B6 02 83 F8 6A 74 07 33 C0 }
  condition:
    $s1
}
```

Note that this same YARA signature run against the 5136 files we found with the "cmp only" BigGrep search verifies 34 of them as well.

Now that raises another question…what about the 88 files that verified from the cmp only search? Why does the above YARA signature only match 34 of them? Well, let's take a look at one that didn't verify with the above, and look for the 6 bytes in the first cmp operation and see what the code looks like:

```
.text:10001272 A1 80 D0 03 10                          mov     eax, ds:ReadFile
.text:10001277 89 45 F4                                mov     [ebp+var_C], eax
.text:1000127A 89 2D 50 2C 04 10                       mov     dword_10042C50, ebp
.text:10001280 60                                      pusha
.text:10001281 C7 45 B0 30 00 00 00                    mov     [ebp+var_50], 30h
.text:10001288 C7 45 B4 03 00 00 00                    mov     [ebp+var_4C], 3
.text:1000128F C7 45 B8 F0 1D 00 10                    mov     [ebp+var_48], offset sub_10001DF0
.text:10001296 C7 45 BC 00 00 00 00                    mov     [ebp+var_44], 0
.text:1000129D C7 45 BC 00 00 00 00                    mov     [ebp+var_44], 0
.text:100012A4 C7 45 C0 00 00 00 00                    mov     [ebp+var_40], 0
.text:100012AB C7 45 BC 00 00 00 00                    mov     [ebp+var_44], 0
.text:100012B2 C7 45 BC 00 00 00 00                    mov     [ebp+var_44], 0
.text:100012B9 C7 45 C0 00 00 00 00                    mov     [ebp+var_40], 0
.text:100012C0 C7 45 C0 00 00 00 00                    mov     [ebp+var_40], 0
.text:100012C7 C7 45 BC 00 00 00 00                    mov     [ebp+var_44], 0
.text:100012CE C7 45 C0 00 00 00 00                    mov     [ebp+var_40], 0
.text:100012D5 C7 45 B8 F0 1D 00 10                    mov     [ebp+var_48], offset sub_10001DF0
.text:100012DC C7 45 BC 00 00 00 00                    mov     [ebp+var_44], 0
.text:100012E3 C7 45 C0 00 00 00 00                    mov     [ebp+var_40], 0
.text:100012EA C7 45 C4 01 00 00 00                    mov     [ebp+var_3C], 1
.text:100012F1 8B 4D F4                                mov     ecx, [ebp+var_C]
.text:100012F4 0F B6 11                                movzx   edx, byte ptr [ecx]
.text:100012F7 81 FA 8B 00 00 00                       cmp     edx, 8Bh
.text:100012FD 74 1B                                   jz      short loc_1000131A
.text:100012FD
.text:100012FF 8B 45 F4                                mov     eax, [ebp+var_C]
.text:10001302 0F B6 08                                movzx   ecx, byte ptr [eax]
.text:10001305 83 F9 55                                cmp     ecx, 55h
.text:10001308 74 10                                   jz      short loc_1000131A
.text:10001308
.text:1000130A 8B 55 F4                                mov     edx, [ebp+var_C]
.text:1000130D 0F B6 02                                movzx   eax, byte ptr [edx]
.text:10001310 83 F8 6A                                cmp     eax, 6Ah
.text:10001313 74 05                                   jz      short loc_1000131A
.text:10001313
.text:10001315 E8 26 FF FF FF                          call    DllEntryPoint
.text:10001315
.text:1000131A
.text:1000131A                         loc_1000131A:                           ; CODE XREF: DllEntryPoint+BD↑j
.text:1000131A                                                                 ; DllEntryPoint+C8↑j ...
.text:1000131A 8B 0D BC D1 03 10                       mov     ecx, ds:LoadIconA
.text:10001320 89 8D 68 FF FF FF                       mov     [ebp+var_98], ecx
.text:10001326 C7 85 60 FF FF FF 18 F0+                mov     [ebp+var_A0], offset aShstem1hrrhntc ; "ShSTEM\\
.text:10001330 C7 85 5C FF FF FF 50 F0+                mov     [ebp+var_A4], offset aControlBackupr ; "Control\
.text:1000133A 8B 15 84 D0 03 10                       mov     edx, ds:CreateFileA
```

This is interesting, we appear to have found another variant of the same technique, and this one is inside a DLL, not an EXE file. And if you look closely you can see that there are additional (unrelated) instructions interleaved with what we were searching for.

There are two other interesting items to note there. First, the check failure condition is different, as that code is in the DllEntryPoint function, which at a quick glance appears to call itself recursively instead of the 'jmp _Abort' we saw in the original code. Second is that another item mentioned in the blog posting is seen here as well, the deliberate misspelling of certain strings. The word "SYSTEM" in the string referenced near the end has the 'Y' replaced with an 'h', to hide the more interesting (and likely to be searched for) string value.

So, depending on our goals for doing these searches, this could lead us down another path of exploration, leading to more iterations of BigGrep searches, more YARA verifications, and more visual inspections.

This sort of iterative process was not feasible before producing BigGrep. The above insights took a relatively short period of time to generate (part of an afternoon) instead of the weeks that it might have taken with our earlier ways of searching.

5. Conclusion and Future Work

For our internal uses, BigGrep has been very successful and useful, allowing us to perform a cycle of fast ad-hoc searches in the course of activities such as malware family analysis. While it has its drawbacks when it comes to searching for strings, that wasn't why it was created, and for the most part an acceptable limitation with potential workarounds.

Possible future efforts/improvements that could be made to BigGrep include:

- Index file merging
- Investigate better 4 gram compression techniques (may require a new or extended index file format)
    - If that works, could investigate longer n-gram sequences as well
- Web interface (an early prototype already exists)
- Helper code to assist in distributing over multiple machines
- Tie in Lucene to handle the string data separately
- Updating existing index files incrementally (would likely require a new index file format)

Possible other uses of n-gram techniques in general may include:

- n-gram histogram comparisons for gross similarity approximations
- use of n-gram subsets for other similarity comparisons

Appendix A. Background and History

Here we shall discuss some more background on the BigGrep project, for those who are interested.

A.1. Initial prototypes

We created some initial (quick and dirty) prototypes to check on how various simplistic/naïve implementations built on some already available systems/libraries might work with regards to speed and disk space, to get a general feel about how various implementation strategies might work out. Some of these were:

- A very simplistic PostgreSQL approach where 32 bit n-grams mapped to 32 bit file ids in a single table
- A Tokyo Cabinet[20] (key-value store) implementation.
- A Redis[21] (a "richer" key-value store) implementation.
- A MongoDB[22] (document store) implementation.

All of those experiments were "failures" in both dimensions (space and speed), and that quickly led us to the realization that our most likely path to success lay in devising a custom file format to store the index in.

A.2. First "real" implementation.

The first "real" implementation consisted of some simple C code for the indexer (Python based ones were WAY too slow), and Python code for search (along with an early version of the C++ based verification).

The indexing code created 3 files for each index: index.dat (which contained the "hints" table to help quickly navigate the next file contents), ngrams.dat (the n-grams and file ids that they mapped to, PFOR and VarByte encoded), and fileids.dat (the mapping of the integer file ids to the file paths/names they belonged to). It too used the Loser tree approach to multiway merge.

A.2.1. Discoveries, Limitations and Issues

As discussed above, this is where we discovered that RAM effectively limits the input batch size on the indexer, and that ordering could be pretty important for good compression.

[20] http://fallabs.com/tokyocabinet/
[21] http://www.redis.io/
[22] http://www.mongodb.org/

We also discovered that having multiple files with fixed names for indexes means that a set of directories is needed to accommodate multiple indexes, with only one index per directory. This proved to be a bit more cumbersome than it needed to be.

False positive rates were pretty low on the 4-gram index searches however. In fact, for a lot of initial searches we could usually safely assume that the majority of the results would in fact verify. One glaring exception being how poorly Unicode encoded ASCII string searches performed.

We also discovered that our 4-gram indexes were turning out MUCH bigger than we had initially expected (avg: ~2.4 times input size). Our lack of disk space was a significant issue at the time, so that coupled with the fact that the indexer would utilize all of the RAM but only one core of our multi-core machines lead to the 2nd implementation.

A.3. Second "real" implementation

Indexer (as discussed in the main part of this paper) changed to C++, with multithreaded portions to help utilize hardware more effectively and decrease overall time (theoretically). It also produced a single index file, and could produce either 3-gram or 4-gram indexes.

A.3.1. New discoveries and optimizations

This is where we discovered that the 3-gram indexes were significantly smaller. Part of the reason is that the 3-gram space is more fully saturated over typical input batches than the 4-gram space, and the PFOR encoding works on blocks of data, and can be more effective with these blocks filled (we would fall back to the less efficient VarByte encoding if a particular set of file ids wasn't long enough for a "full" PFOR block, which we had defaulted to 128 values).

That realization allowed us to experiment with the block sizes used, and we discovered that decreasing the block sizes could really help our compression rates (to a certain point). Allowing for the minimum number to qualify for PFOR encoding to be less than a full block helped as well. These are all configurable parameters now.

We also added minor enhancements to PFOR based on a paper about AFOR[23] (Adaptive Frame Of Reference), specifically we have a status byte that uses one nybble for the value of 'b' for the block, and the other nybble for the number of exceptions in that block, but by declaring the special value FF to say "a delta list of all 1s follows, but is not stored" we could reduce an entire PFOR encoded block to 1 byte (plus possible exceptions), which provides an additional improvement in well ordered batches.

We also removed the n-gram value from the index. Previously we had the n-gram value (3-4 bytes) encoded in the list, so we wouldn't have any entries for missing n-grams

[23] http://www.deri.ie/fileadmin/documents/deri-tr-afor.pdf

for that batch. However, there were very few batches where that reduced the size of the index over making the assumption that all n-grams were present and simply having a 0 size (VarByte encoded, so a single byte) of the file id list data indicate a missing n-gram. So that helped reduce the size of our indexes as well (much more so in 4-gram indexes than 3-gram ones).

A.3.2. Failed optimizations

Multithreaded programing can be tricky. An experiment in doing a multithreaded continuous merge was done, and it could be very fast for test datasets, but in practice consumed way too much RAM to be useful for any significant index batch sizes. All attempts to minimize the RAM usage resulted in much slower operation than the initial implementation and/or still caused RAM explosion because the continuous merge was slowed down to the point where it couldn't keep up with the multithreaded shingling.

This is an area that might be useful to explore again in the future though.