A Scalable Search Index for Binary Files

Wesley Jin*, Charles Hines†, Cory Cohen‡ and Priya Narasimhan‡
*wesleyj@andrew.cmu.edu, CMU ‡cfc@cert.org, CERT †hines@cert.org, CERT §priya@cs.cmu.edu, CMU

***Abstract*—The ability to locate specific byte-sequences in large collections of binary files is important in many applications, especially malware analysis. However, it can be a time consuming process. Researchers and analysts, such as those at CERT, often have to search terabytes of data for characteristic patterns and signatures, which can take upwards of days to complete. Although many search systems, designed specifically to expedite text and metadata queries, exist, these tools are unsuitable for searching files containing arbitrary bytes. By using probabilistic techniques to pre-filter likely search candidates, we present a scalable architecture for searching and indexing terabyte-size collections of binary files. Our implementation performs searches in minutes that would required days to complete using iterative techniques. It also reduces storage costs by balancing the amount of data indexed with the total time required to conduct and verify a query.**

I. Introduction

Organizations, like CERT [13], maintain collections of millions of malware, with numerous variants from different families. Analysts often face a scenario in which they possess a copy of a particular variant, and wish to search the collection for other family members. Specifically, they undergo the following recursive process: 1) A unique pattern (i.e. a particular command and control string, a custom encryption routine, etc.) is identified in a sample. 2) Variants with this pattern are found using iterative search tools [13]. 3) Additional knowledge is gained (i.e., new functionality, modifications to existing code, etc.), from which a new query is produced. The problem is that the size of the collection has grown to such an extent that iterative search can take several days to complete, making ad-hoc searching impractical.

Significant research effort has been dedicated to search indexes designed to expedite text and specialized data queries [3] [4] [5] [6]. More generic indexes that permit queries of files' binary contents present a number of unique challenges, however:

- Unlike text, where the number of possible terms (i.e. words associated with documents) is limited to sequences of printable characters, arbitrary byte patterns can take on significantly more combinations. Our experiments show that this difference can result in much larger indexes.
- In text documents, whitespace provides a natural delimiter for terms. Also, it is often only necessary to index a few keywords per document; common words such as 'the' can be ommitted. In arbitrary binary files, there are no natural delimiters and no obvious ways of deciding which byte-sequences to exclude. We found that inserting all of these terms into the index, in a reasonable amount of time, is a non-trivial task.

In this paper, we present a system that is built around an N-gram inverted index, a data structure that maps byte sequences of length N to files [5]. Although the use of N-gram indexes for *exact* search is nothing new, our system is novel because we use the structure to pre-filter a set of files that is likely to contain the query, and then search this smaller set iteratively. Using the index in this way allows us to reduce the amount of data stored, but still return results in an acceptable amount of time for day-to-day analysis tasks. Specifically, our system does not store file offsets.

Traditional indexes contain offsets that record the precise locations of terms within files. Files containing a search string can be found by breaking the search string into its constituent terms and identifying documents with those terms in sequential order. Unfortunately, because of the large number of terms that must be stored for binary files, the cost of recording all those positions is too high. Our system discards offsets and instead returns a candidate set of files, each containing all of the specified N-gram terms (but not necessarily in sequential order). Thus, at the price of having to perform iterative search over a reduced set for verification, our index occupies significantly less space than traditional search indexes. Despite this two step process, our experiments demonstrate that we can still achieve very reasonable overall search times.

The contributions of this paper are:

- We present an algorithm that was able to construct an N-gram index for ≈9TB of binary files in a reasonable amount of time and disk space.
- We demonstrate the importance of batching (likely) similar binary files to the overall size of the index. In particular, we contrast index sizes for batches of nearly identical files versus batches containing randomly selected files.
- We demonstrate that this index structure can reduce query times from days to minutes compared to queries performed using iterative techniques over all 26 million+ binary files.

The rest of this paper is organized as follows: Section II discusses related work. Section III formalizes the problem. Section IV scopes this paper. Section V discusses our approach. Section VI analyzes the performance of our system. Section VII discusses our experimental results. Section VIII discusses future work. Section IX concludes the paper.

II. Background

Our system constructs an N-gram index, a type of inverted index. In this section, we review the relevant concepts on inverted indexes and N-grams as traditonally proposed (containing term offsets.) For a more thorough discussion, we direct the reader to Manning's work [5]. We also review relevant algorithms for index compression. For additional details, please see [7] [8] [9].

A. N-grams Index

1) N-grams Extraction: N-grams are byte sequences of length N. Kim et al. [3] introduces a technique, called 1-sliding or shingling, for extracting all of the N-grams from a buffer with a length >= N. The algorithm slides a window of size N bytes through a buffer, one byte at a time. The contents of the window at every iteration form an N-gram. For example, applying shingling to the string ABCDEFG with N=4 produces: ABCD, BCDE, CDEF and DEFG.

2) N-grams List and Postings List: An inverted index is a data structure that maps file content (i.e., individual words or byte-sequences) to lists of files in which that data can be found. The traditional N-grams index is a variant of the inverted index which maps N-grams to files. It is composed of two parts: an N-grams list and postings list. The first is a catalogue of all unique sequences extracted from the files. The second contains entries, for each N-gram sequence, of the form:

<FileID1, offset11, offset12, ..., frequency1>,
<FileID2, offset21, offset22, ..., frequency2>,
...

where `FileIDX` is a unique identifier of a file that contains this particular N-gram, and `offsetXY` is a position, within file X, where the N-gram can be found. `frequency` is the number of times that the term appears within the file. The N-grams list allows the postings entry for a particular N-gram to be found quickly. It is commonly comprised of a list of pointers/offsets into the postings list for each N-gram.

3) Querying: Querying the index for a string involves four steps:

1) decomposing a search string into its constituent N-grams via shingling
2) retrieving the postings entries for each N-gram
3) finding the common FileIDs between all of the entries
4) verifying that the file offsets for each N-gram are sequentially ordered.

Example: Consider an index (N=4) containing three files:

File 1:	AAADEADBBB
File 2:	ADEADBEEFC
File 3:	DEADBEECBEEF

and the query string DEADBEEF, which breaks down to the following 4-grams: DEAD, EADB, ADBE, DBEE, and BEEF.

File 1 contains only two of the five 4-grams in DEADBEEF. File 2 contains all of the N-grams in the search term. Also, each N-gram begins at a position that is one greater than the previous. Thus, File 2 contains the search string. File 3 also contains all of the N-grams. However, BEEF is located at offset 8, which does not follow DBEE at offset 3, so it is not a match.

Note that in our index, which does not store file offsets, we refer to files like 3 (those that contain all of the N-grams of a search string, but not ordered consecutively) as false positives. In our system, we omit step 4 of the querying process. Instead, we eliminate false positives with an iterative search over the set of files returned by performing steps 1-3.

B. Compression Techniques

A significant amount of research has been done on index compression [7] [8] [9]. These techniques aim to reduce the amount of space taken by file identifiers within the postings entries. Here, we review two algorithms that are relevant to our system.

1) Variable Byte Encoding: Variable byte (aka VarByte) encoding is a byte-aligned technique designed to compress single integers into a variable number of bytes that is (hopefully) smaller than the default size of the integer. It is simple, quickly decoded, and suited to situations where a small number of small integers need to be compressed [8].

The key idea of the scheme is to utilize a single bit of each byte (usually either the the least significant bit, LSB, or the most significant bit, MSB) as a flag to indicate the presence of another byte with more bits for the encoded value. By using this scheme, the numbers 0-127 can be encoded in one byte, 128-16,383 in two, 16,384-2,097,151 in three, and so on.

For example, consider the 32-bit number 345 (in little-endian binary: 01011001 00000001 00000000 00000000) where it is plain to see that the leading zeros are "wasted." Using variable byte compression (using the LSB as a flag), this number is compressed into just two bytes (10110011 00000100). Notice that the LSB of the first byte is set to 1 to indicate the presence of another byte following. The second byte has an LSB of 0, signaling the end of the number. Bits are shifted and concatenated (minus the indicator bit) to recreate the original number.

2) PForDelta Encoding: PFOR (Patched Frame Of Reference) encoding is a method used to compress batches of integers [9]. It achieves a high-compression ratio, a fast decode time, and is well suited to applications where many integers need to be compressed.

The scheme works as follows: First, a value ρ is determined such that 2^ρ is greater than the majority (i.e. 90%) of the integers in a batch of size $\mathbf{N}$. The choice of $\mathbf{N}$ is typically constrained to a multiple of 8 to ensure byte level alignment of the compressed values. These numbers are then packed into a block that is the $\mathbf{N}*\rho$ bits wide. Numbers that do not fit into ρ bits are called "exceptions." Following the compressed block is a list of exception indexes and then the actual exception values themselves.

For example, consider the following list of integers:

67, 23, 65, 123456, 24, 3, 76451, 12

For space consideration, we assume that the block size is eight and the maximum number of exceptions is two. All of the numbers except for 123456 (index=3) and 76451 (index=6) fit into seven bits. Therefore, the compressed block would be 8*7 bits (7 bytes) wide. All numbers that fit into seven bits would be packed into their respective locations. Following the compressed block would be the exception index values 3 and 6, and following those would be the exception values 123456 and 76451.

Assuming 32 bit integers, and that the exception offsets and exception values are stored using variable byte encoding, this would reduce the input integer list from 32 bytes down to 15

$(7 + 1 + 1 + 3 + 3)$ bytes, plus a little overhead required to encode the value of ρ and actual number of exceptions for this block, which for this example both can easily be stored in a single leading packed byte, for a grand total of 16 bytes.

PForDelta is the name given to using the PFOR technique to compress lists that are based on deltas from a given value instead of actual values. This is done to reduce the values that need to be stored, and hence to potentially reduce the value of ρ needed for a given block.

To illustrate, we return to the example above. The numbers would first be sorted, like so:

3, 12, 23, 24, 65, 67, 76451, 123456

You then proceed by subtracting each value in the list from the one before it. The first value can be assumed to be "delta from zero" or from some other value external to this list, like perhaps the last value of the previous list that was compressed. So our delta encoded list for this example is now:

3 *(3-0)*, 9 *(12-3)*, 11 *(23-12)*, 1 *(24-23)*, 41 *(65-24)*, 2 *(67-65)*, 76384 *(76451-67)*, 47005 *(123456-76451)*

Note that these values can fit in a ρ of six with (again) two exceptions and a leading packed byte containing the value of ρ and the actual number of exceptions for this block (2), this time taking up only 15 bytes $(1 + 6 + 1 + 1 + 3 + 3)$.

An important observation is that the size of the compressed block is dependent on the relative differences of the numbers being compressed. In particular, long sequences of very small deltas result in exceptionally good compression. Therefore, it is advantageous to assign files frequently found in the same postings lists to IDs with small deltas from one another.

III. Problem Statement

Let F be a collection of binary files, where each file f can be decomposed into a sequence of overlapping N-grams, $f = (N_0, N_1, ..., N_x)$, as extracted by 1-sliding.

Let Q be a binary query that can be decomposed into a sequence of overlapping N-grams, $Q = (q_0, q_1, q_2...q_L)$:

Identify the subset of F, C, in which each file $c = (m_0, m_1, ..., m_y)$ contains an identical sequence of N-grams as those found in Q:

$$\{C \subset F, c \in C, \exists a \mid m_{a+i} = q_i \wedge 0 \leq a \leq y - L \wedge 0 \leq i \leq L\}$$

IV. Goals and Non-goals

Goals:

1) Construct an N-gram index from a terabyte sized collection of binary files, on a single machine, in a reasonable amount of time.
2) Minimize the space occupied by the index.
3) Reduce query times from days to minutes or seconds.
4) Minimize the time required to add new files, after an initial index has been constructed.

Non-goals:

1) We are only interested in exact, binary string matching. Regular expressions are out of scope for this work.
2) Although our algorithms could be extended to a cluster of machines, we are focused on producing and querying an index on a single machine only.

V. Approach

In this section we provide a high-level description of our system. To catalog the experiences in leading up to our final approach, we discuss several unsuccessful attempts to implement this system using general purpose database/key-value stores and tools from computational biology. We conclude the section with the algorithm of our final implementation.

Our system builds an offset-free inverted index using all of the unique N-grams extracted from a file with 1-sliding. Files are assigned unique identifiers (see Document Ordering). A postings entry is created for each unique N-gram, containing a compressed list of file IDs whose files contain that N-gram.

Querying for a search string involves decomposing the string into its constituent N-grams, retrieving each of their postings entries, and finding files that are common across all entries. As our index does not store offsets, each of these files is then checked to see whether it actually contains the search string.

Initially, we tried to construct the index with open-source packages: PostgreSQL [11], Tokyo Cabinet [14], MongoDB [15] and Redis [12]. We also tried a tool from computational biology, Tallymer [18], designed to build large k-mer[1] indexes. However, in each case, performance problems encountered while storing postings entries prevented us from completing the index.

A. Initial Attempts

1) PostgreSQL: In our first attempt, we created a (deliberately naive) table with two columns: `n-gram` and `fileIDlist`. `fileIDlist` was a variable length field that contained the sequence of document identifiers. We tried to fetch and update the entries corresponding to N-grams, extracted using 1-sliding. Unfortunately, the process of fetching and updating database rows proved to be too slow (even when common N-grams from multiple files were batched). An attempt to create postings entries for a small set of 3,465 malware files (≈5GB of data), was aborted after a week of processing.

In our second PostgreSQL attempt, we tried to eliminate the bottleneck caused by fetching and updating. We created a table (again, deliberately naive) with two columns: `n-gram` and `fileID`. The idea was to create a unique row for each file/N-gram pair. Using a bulk-loading utility [16], we inserted the N-grams into the table in batches. Although we were able to construct the index for more than 3000 files in under a day, the size of the table ballooned to an unacceptable size.

2) Tokyo Cabinet and MongoDB: Tokyo Cabinet [14] and MongoDB [15] are two high-speed, NoSQL style open-source databases (a B-Tree based key value store and a document store respectively). Both possess attractive features, such as the ability to rapidly update lists. Unfortunately, the quantity of

[1]The bioinformatic community uses the term k-mer instead of N-gram.

N-grams that had to be updated or inserted was apparently too large for either configuration. In the case of Tokyo Cabinet, insertions proceeded at a steady pace at the beginning, but as the database became more populated, the rate of insertion slowed steadily and significantly. In both cases, for the same set of 3,465 files, the index was less than half complete after two days of processing.

3) Redis: Redis [12] is another open-source key-value store with built-in support for strings, hashes, lists and sets. Unfortunately, the system also requires that the entire key-space be kept in memory. Since the size of the key space depends on the total number of possible N-grams, and we had chosen 4-grams to work with, that meant a key-space of 4,294,967,296 keys, each of which is 4 bytes in size for a minimum of 16GB for the keys alone, not counting data structure overhead. The machines that we tested on should have had enough RAM to handle this, however there were still issues with running out of RAM during our experiments despite the fact that the entire key space was not covered during those experiments.

4) File-Based Approach: Aside from the open-source packages, we also tried to implement the index simply using a set of files to store postings entries. For example, file 000000 would store the postings entries for all N-grams 0x00000000-0x000000FF. Unfortunately, extracting the N-grams and creating these files proved to be too slow once again.

5) Tallymer: Tallymer [18] is a bio-informatics project that uses Enhanced Suffix Arrays (ESA) to index large genome-sequence sets, which can then be used to build a k-mer index. Index creation is a two step process: an ESA is created from the raw data and a k-mer index is built on top of this ESA.

We converted approximately 19,000 binary files (2.2GB of raw data) to a representation understandable by the tool (4.4GB converted).[2] We then tried generating the ESA. After six hours of processing, the tool's progress bar remained at 0%, with an estimated completion time measured in decades. The process appears to have been CPU bound, as disk usage was minimal. We terminated the job and re-tried with a smaller input set of about 800MB. We eventually stopped this job as well, after several hours of no progress.

An interesting note is that we ran Tallymer on the hg19 [19] dataset which contains approximately 3GB of data in the standard DNA alphabet, and it finished producing an ESA in less than 45 minutes. Although large in size, a key difference between the genomic data and the binary data is the alphabet size. In hg19, five symbols are used to represent genomic data. Each binary character in our test set, a nybble of information, can be one of sixteen symbols. As noted by [17], this alphabet size difference can have a dramatic impact on the construction time of a suffix tree. We also suspect that the binary data has a larger diversity of value patterns than genomic data, further affecting the performance of the algorithms used in building the ESA.

[2]The FASTA [22] file format is an ASCII representation of the binary data, using a custom alphabet consisting of the hex symbols 0-9 and A-F. We would have preferred to use a whole byte as the alphabet range, but all of the bioinformatic tools we looked at required ASCII representations of the alphabet, limiting our choice to the nybble.

We also observed similar results in ad-hoc attempts to pass the same binary data through other similar bioinfomatic tools.

B. Offset-free n-grams Index

After struggling with the various open-source database systems, we decided to implement our own inverted index from scratch. This decision was motivated in part by the observation that a custom solution could be better tuned to our specific needs that the general-puporse systems that we had been evaluating. Specifically, we sought an architecture that minimized disk seeks, which seemed to be the primary performance problem in the other systems.

1) N-grams List and Postings List: The N-grams list is designed to allow for nearly constant time lookup of postings entries. It consists of a sequence of 64-bit offsets into the postings list. The offset for a particular N-gram is located by taking its N-1 most significant bytes, shifting them right by one byte, and multiplying this quantity by 8. For example, the N-grams 0xDEADBEEF and 0xDEADBE11 share a common offset, located at 0xDEADBE * 8 in the N-grams list. Beginning at this location in the postings list, a linear search (of at most 256 entries) will determine if the N-gram is present in any of the files. See Fig. V-B1 for an approximate graphical representation.

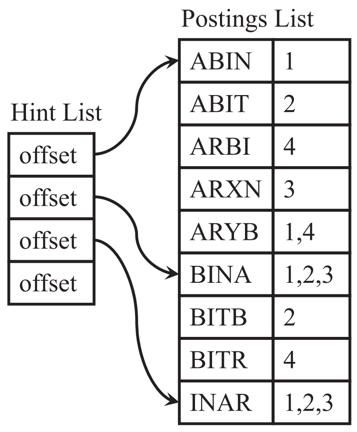

Fig. 1. N-grams and Postings List

The postings list consists of data in the following form:
`<n-gram, 4 bytes>` The N-gram shared by all of the files in this list
`<compression type + compressed data size, 4 bytes>` The most significant bit is used as a flag identifying the compression type (variable byte = 0, PForDelta = 1), the remaining bits are the size of the compressed data that follows.
`<compressed data>` The file IDs in a delta encoded list, compressed as indicated by the compression type bit.

When PForDelta is used, each block of the compressed data is structured as follows:
`<`ρ`, 1 nybble>` - 90% of IDs are less than 2^{ρ}
`<# of exceptions, 1 nybble>` - The number of IDs that

are greater than 2^ρ
`<first ID>` - The smallest file ID, used to recover IDs from deltas, in variable byte encoding
`<compressed block>` - ρ*block size bits wide
`<exception indexes>` - list of exception identifiers
`<exception values>` - list of variable byte compressed exception values

PForDelta compression is used when the number of IDs is greater than or equal to the PForDelta block size, and variable byte compression is used for smaller lists or lists that cannot meet the requirements for our PFOR implementation (ρ less than 16 and a maximum number of exceptions based on block size but also constrained to be less than 16, so the two values can be encoded as two nybbles of a single byte).

2) Index Construction: Separate indexes (N-gram/postings list pairs) are created for batches of files and can be merged afterwards if desired. Since the index is constructed in memory and flushed to disk, the maximum batch size is determined by the size of physical memory.

Algorithm 1 is composed of three logical parts. First, the unique N-grams from all of the files in a batch are extracted using 1-sliding. Second, common N-grams across all of the files must be merged into a single unified list. For this purpose, we use a Loser Tree [21], which is an efficient data structure merging multiple sorted lists. Third, file IDs for each N-gram are compressed to form new postings entries. Once a sufficient number of entries have been buffered, they are flushed to disk. The N-grams list is updated accordingly to point to these newly written entries.

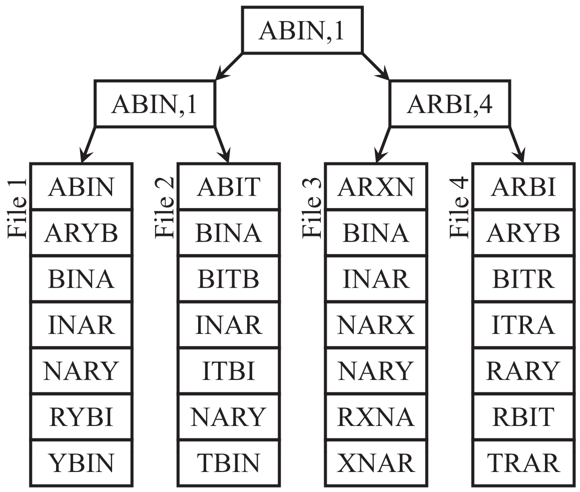

Fig. 2. N-gram collection with Loser Tree 1

Fig. V-B2 and Fig. V-B2 illustrate the collection of N-grams for four example files. Each leaf node is a sorted list of unique N-grams for each file. At each iteration of Algorithm 1, the minimum N-gram and file ID pair is found by taking the node at the top of the tree, and appending that file to the list of matching files.

Two key features of Algorithm 1 are periodic disk writes

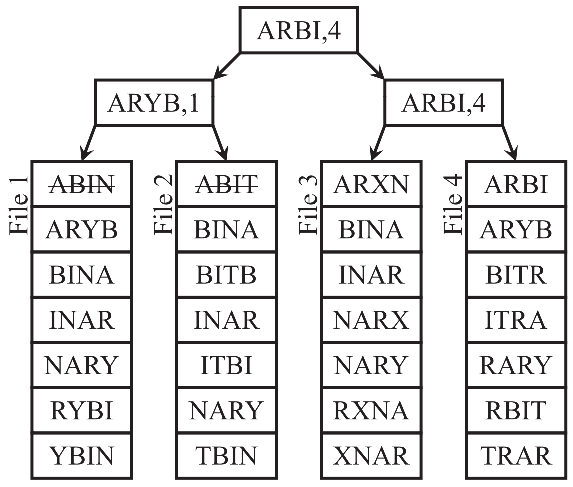

Fig. 3. N-gram collection with Loser Tree 2

and the use of the Loser Tree. Once files have been read and their N-grams extracted (O($M * N + N * log(N)$) where M is the number of files and N is average file size), `FetchSmallestUnprocessedNGram` has a logarithmic runtime. Periodic disk writes help reduce the seek delays that occur with frequent disk I/O.

3) Document Ordering: A key observation in [10] and [7] is that document ordering can have a significant impact on compression when using PForDelta. Specifically, a greater compression ratio can be achieved by assigning IDs, that are close to one another (i.e., share a small difference), to documents that share many N-grams. Thus their IDs will be present in many of the same postings entries, and the computed deltas between these identifiers will be small. Using code comparison techniques developed at CERT, binaries suspected of being from the same malware family were given IDs close to one another.

4) Querying: Given a search string, querying on a single index (an N-grams and postings list pair) is performed according to Algorithm 2. Querying across all indexes is performed iteratively across all pairs[3].

VI. Analysis

This section analyzes the performance of our system in terms of false positives, time required to query the index, and index size. We begin with a qualitative discussion of the choices that influence these variables, and follow with a more detailed analysis of each metric.

A. Design Decisions

The most significant tradeoff in our system is between total search time and index size, controlled by N. The total search time is the sum of the index query time and the time required to verify the results. The former is proportional to the number of N-grams that must be retrieved from the index per

[3]This step could easily be parallelized or split across multiple machines.

Algorithm 1: Index Builder

```
Input: Set of files files
Output: N-grams List nlist and Postings List plist
prev = nil;                                  // previous ngram
curr = nil;                                  // current ngram
pbuf = nil;                                  // postings list buffer
start = 0;                                   // start offset of current entry
end = 0;                                     // end offset of current entry
written = 0;                                 // bytes written to postings list
nbuf = nil;                                  // ngram list buffer
index = 0;                                   // next ngram to write to index
foreach $f$ in files do
    $buffer_f$ = ReadFully($f$);
    $ngrams_f$ = ExtractNGrams($buffer_f$);
    Sort($ngrams_f$);
loser = BuildLoserTree($ngrams_{1...n}$);
while HasUnprocessedNGrams(loser) do
    (curr, fileid) = FetchSmallestNGram(loser);
    if $curr \neq prev$ then                 // starting new ngram?
        if $pbuf \neq nil$ then              // output pending?
            CompressBuffer(pbuf, start, end);
            start = end;
        if $curr \geq index$ then            // new ngrams entry?
            while Count(nbuf) $\leq$ (curr » 8) do
                Append(nbuf, written + start);
            index = curr & 0xFFFFFF00;
        if IsNearlyFull(pbuf) then           // for performance
            written += WritePostingsList(plist, pbuf);
            start = 0;                       // reset buffer offsets
            end = 0;
        InitializePostingsEntry(pbuf, curr, start);
        end += 1;
    AppendFileID(pbuf, fileid);              // new postings entry
    end += 1;
    prev = curr;
WriteNgramsList(nlist, nbuf);
```

Algorithm 2: Query

```
Input: Search String string,
       N-grams List nlist, Postings List plist
Output: List of matching file IDs matches
matches = nil ;   // file IDs matching all ngrams in string
sngrams = ExtractNGrams(string);  // all ngrams in string
foreach sngram in sngrams do
    noffset = (sngram » 8) * 8;        // offset in ngrams list
    Seek(nlist, noffset);
    poffset = ReadOffset(nlist);       // offset in postings list
    Seek(plist, poffset);
    files = nil;               // file IDs matching current ngram
    cngram = ReadPostingEntry(plist); // at current offset
    while $cngram \neq sngram$ do        // at most 256 times
        if (cngram » 8) > (sngram » 8) then
            break ;                    // no files match this ngram
        cngram = ReadPostingEntry(plist);
        files = DecompressEntry(plist);
    if $files \neq nil$ then
        if matches = nil then matches = files;
        else matches = matches $\cap$ files;
    else break;                        // no files match string
matches = IterativeVerification(matches);
```

query, which is inversely proportional to the size of N. The verification time is inversely proportional to the number of false positives. Intuitively, as N approaches the query length (i.e., the length of the search string), the number of false positives falls to zero. Thus, from a search time perspective, it is advantageous to select a value of N that is equal to the minimum query length[4].

The index size is controlled by the number of unique N-grams and the size of their postings lists. As the size of N decreases, the number of unique N-grams decreases and their respective postings list sizes increase. Furthermore, as the distribution of file IDs becomes more dense, compression techniques, designed to compress large blocks of integers like PForDelta, may be used to further reduce the size of the index. Thus, from a disk perspective, it is advantageous to select the smallest possible value of N.

Traditional indexes that contain offset information face similar choices when selecting an N-gram size. Storing offsets implies that the disk requirement of these systems will always be greater than one without offsets. However, offset information also means that the verification time of these systems will always be zero. Thus, traditional systems will always have superior temporal performance, while our architecture will always require less storage.

[4]At CERT, analysts often need to search for strings as short as five or six bytes long

B. False Positives

Since our index lacks offset information, an important question is how many false positives occur for a given query that must be eliminated by iterative verifcation. Our analysis of this probabiltiy is as follows:

Let Q be a search string, containing L overlapping N-grams q_1,q_2,...,q_L where L = |Q| - N + 1

Let F be a file, containing X overlapping N-grams f_1,f_2,f_3,...,f_X where $X = |F| - N + 1 >= L$

Let P be the probability density function over the set of N-grams (i.e., $P[q_y]$ = probability of the N-gram q_y occurring in any file).

Let Pc be the conditional probability density function over the set of N-grams (i.e., $Pc[q_y|q_i \cap q_j]$ = probability of the N-gram q_y occurring in files known to contain the N-grams q_i and q_j).

Let a false positive be defined as some file containing all of the N-grams in Q, in non-consecutive/non-overlapping order. Thus the probability of all N-grams in Q occurring in F is:

$$A = X!/(X-L)! * \prod_{q_i \in Q} P[q_i]$$

And the probability of all N-grams in Q occurring in overlapping order is:

$$TP = (X-L+1) * \prod_{q_i \in Q} Pc[q_i | q_{i-1} \cap q_{i-2} ... q_1]$$

The probability of a false positive is therfore $FP = A - TP$.

In practice, the probability distributions P and Pc will depend on the nature of the files. For example, for batches of files containing packed malware, the N-gram distributions are nearly uniform and P is approximately equal to Pc. Our experimental results show that the probability distribution in our target dataset causes the false positives to fall exponentially as the length of the search string grows.

C. Index Query Time

In this section, we give an estimate of the worst-case query performance of our system.

Let Q be a search string, containing L overlapping N-grams.

Let P be the PForDelta block size and E be the maximum number of exceptions in any given block.

Let F be the average number of files in a single N-gram and postings list pair (index).

Let B be the total number of indexes.

The lookup time for a single N-gram is constant, $O(1)$. The postings entry offset is located by seeking to the appropriate location in the N-gram list as described previously. An iteration over at most 256 postings entries will determine if the N-gram is present in any of the files of the batch.

Assuming $F > P$ and that every posting entry must use PForDelta, the maximum number of PForDelta blocks in a single posting entry is F/P.

Decoding a single PForDelta block requires unpacking P deltas and substituting E exceptions: $O(P + E)$

Decoding a PForDelta block requires decoding each individual block and summing together the deltas to recover the file identifiers:

$$O(F/P * (P + E) + F) \approx O(F(1 + EF/P))$$

The worst-case query time for a single index is:

$$O(LF(1 + E/P))$$

The worst case query time across all indexes is:

$$O(BLF(1 + E/P))$$

As E is a tunable parameter and typically much smaller than P, the first term dominates the query time. As one might expect, a query containing more N-grams will take more time to complete than a shorter one. Indexes containing more file IDs per postings entry on average will take longer to decompress, causing query times to grow accordingly. Experimentally, the constant portions of the query times were found to be quite small, making queries against large numbers of reasonbly sized files very fast.

D. Index Size

The parameters affecting the size of a single batch are: total number of unique N-grams found in all files (the total number of postings entries T), the percentage of files requiring PForDelta compression C, average number of blocks in a PForDelta-compressed posting entry A, average number of IDs in a Variable-Byte compressed posting entry B, and the average number of exceptions per PForDelta block E.

The first parameter is dependent on the nature of the input files. For example, a batch containing packed malware will most likely have a larger number of unique N-grams than one that contains un-packed programs. The other parameters are related to file similarity within the batch. As the number of common N-grams increases, the number of PForDelta compressed postings entries will increase. Accordingly, the number of blocks within those entries will also increase. The number of exceptions is related to file ID assignment. A greater number of exceptions will occur if files containing common N-grams are assigned IDs with large differences.

Let Y be the average size of a single PForDelta block. Recall that a single PForDelta compressed posting entry may contain multiple PForDelta blocks.

Let Z be the average size of a single delta, compressed using variable byte encoding.

The number of posting entries compressed with PForDelta is TC. The average size[5] of a PForDelta posting entry is: $A(Y + EZ)$. The average size of a variable byte posting entry is: BZ. The size of the N-gram list is a constant 128MB $(8 * 2^{24})$.

Thus the overall average size of the postings list is:

$$TCA(Y + EZ) + (T(1 - C)BZ)$$

VII. Performance Evaluation

We conducted a number of experiments to evaluate the performance of our system and test its suitability for malware analysis.

A. Experimental Setup

Our experiments demonstrate that it is possible to create a search index for terabytes of binary files, efficiently in both time and size, on a single machine. The results show the importance of assigning IDs based upon file similarity. They also show the relationships between query length, number of false positives and query time. We also conducted experiments designed to mimic queries conducted by actual analysts.

We use the *output/input ratio*[6] to highlight the effects of compression:

$$Output/Input = \frac{TotalSizeofpostingsList}{TotalSizeofInputFiles}$$

[5]For simplicity, we neglect the overhead from the block header that includes information about the N-gram, block size and compression type.

[6]The size of N-grams list is constant. Thus, we only include the size of the postings list.

We give the index construction and query times in terms of both CPU and wall-clock time.

We generated N-grams/posting list pairs for the entire malware collection, which as of the end of 2011 consisted of 24,792,223 files comprising 8,736GB. The binaries are a mixture of packed and unpacked files, some grouped into common malware families. To illustrate the impact of document ordering, we measured the index size for batches containing files known to be similar and different (i.e., as determined by automated and manual techniques). The index was created on a system running Red Hat Enterprise Linux 5, kernel version 2.6.18-274.12.1, 16 Intel Xeon CPUs, and 48GB of RAM. Queries were conducted on the same machine.

1) Index Generation: Table I shows the creation time and output/input ratio for six groups of files: Batches 14-19 are members of the Allaple (polymorphic) family, 20-22 Scraze family, 24-26 Trymedia family, 55-56 malicious PDF files, 185-195 malware files classified as being similar, and 531-537 (un-categorized) files received in November 2011.

The effects of file ordering are particularly evident in the case of Trymedia, whose compressed indexes have an output/input ratio of 0.13. In the case of Allaple, a polymorphic malware family, the output/input ratio is significantly higher because of obfuscation. The lowest observed output/input ratio was 0.01. The highest was 7.x.

TABLE I
INDEX CREATION TIME AND SIZE

Batch Number	CPU Time (sec)	Input Size (GB)	Posting Size (GB)	Within Pct.	Cross Pct.	Out/In Ratio
14	7547.32	8.44	45.04	88	43	5.34
15	7651.17	8.44	46.00	91	43	5.45
16	7559.95	8.44	45.35	91	42	5.37
20	3187.45	9.13	16.98	56	31	1.86
21	3187.45	9.13	17.00	56	31	1.86
22	3185.82	9.13	16.99	56	31	1.86
24	4175.58	7.99	1.03	95	0	0.13
25	4235.68	7.99	1.03	95	0	0.13
26	4241.36	7.99	1.17	95	0	0.15
55	2361.43	5.42	17.54	45	60	3.24
56	1639.07	5.40	18.90	52	62	3.50
185	5952.58	9.94	40.80	98	32	4.10
186	5752.22	9.94	39.56	99	31	3.98
187	5451.61	9.94	40.92	97	33	4.12
188	5157.10	9.94	41.02	98	33	4.13
189	4860.35	9.93	42.02	97	35	4.23
190	3078.25	6.33	16.90	97	18	2.67
191	2429.53	6.33	24.22	99	31	3.83
192	3375.16	11.00	22.38	56	30	2.03
193	3192.75	11.00	19.97	42	32	1.82
194	4248.56	11.00	28.45	66	29	2.59
195	4228.38	11.00	28.29	64	31	2.57
531	1574.10	3.94	16.54	93	41	4.20
532	3328.84	11.00	7.99	54	12	0.73
533	1500.70	6.87	5.48	36	16	0.80
534	2852.18	9.38	15.20	59	22	1.62
535	3065.81	9.91	27.72	58	42	2.80
536	4123.89	10.41	21.00	81	20	2.02
537	3719.06	10.93	18.07	67	20	1.65

2) Query Time and False Positives: Using substrings of a 256-byte binary sequence (the standard DES S-box, sometimes used by malware for encryption), we issued queries of increasing length. We measured the number of false positives and query time for each substring. Fig. VII-A2 and Fig. VII-A2 show the results of the experiment. The search string was known to be present in 81,727 files in the collection. As suggested by the false positive analysis, the number of false positives falls exponentially as the length of the query increases. The query time increases linearly as more 4-grams are fetched and their postings entries are decoded.

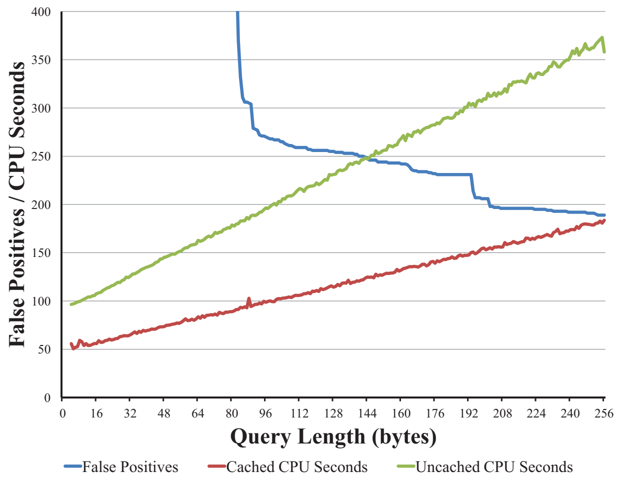

Fig. 4. Query Performance vs. False Positives Linear Scale

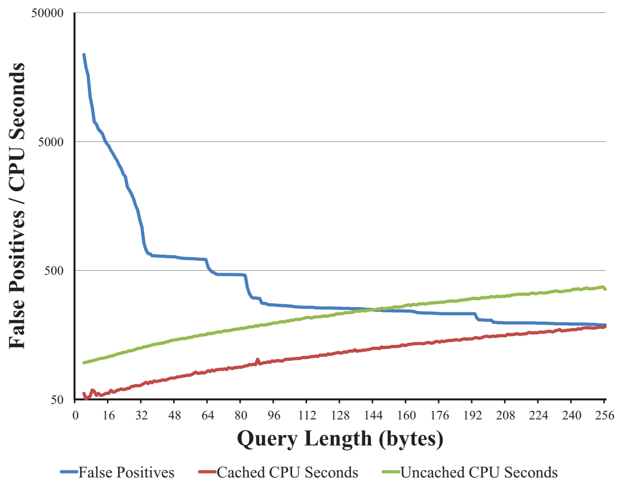

Fig. 5. Query Performance vs. False Positives Log Scale

3) Malware Analysis Scenario: In our second query experiment, we simulated a typical malware analysis scenario. We visited an antivirus vendor website, and considered the technical description of a common malware family named "Stuxnet" [20]. The vendor's analysis identified several potentially distinctive strings, which we used to make queries against our database. In this case, the strings were portions of modified filenames and registry keys. The six selected queries took between 13 and 20 seconds each to complete their search of 26 million malware samples. The query strings were between 13 and 19 characters long and were composed of ASCII text. The queries collectively returned 56 unique files, in 3 primary clusters as shown in Table II. There were no false positives in any of the queries. At this point in a real-life scenario, a malware analyst interested in investigating the family described by the antivirus report now has a collection of files to begin analysis. But more importantly, he has already begun to identify inconsistencies in the existing analysis, by recognizing that the criteria selected does not produce consistent results.

TABLE II
STUXNET QUERIES

Search String	**# Matched Files**	**False Positives**	**Query Time (seconds)**
\drivers\mrxcls.sys	30	0	17.38
\drivers\mrxnet.sys	30	0	19.35
\inf\mdmeric3.PNF	23	0	15.35
\inf\mdmcpq3.PNF	23	0	13.33
\∼WTR4132.tmp	7	0	16.09
\∼WTR4141.tmp	7	0	17.38

4) Queries with Large Numbers of Matches: To test a scenario in which a poorly selected search term returns a large number of results, we searched for the string "This program cannot be run in DOS mode". This string is present in the DOS stub of most Windows executables, which is the primary file type in the malware collection. As expected, the query matched approximately 16.4 million files in the collection of 26 million samples. The query used 2,987 seconds of CPU time and completed in 6 minutes and 32 seconds of wall clock time. We also repeated the test using the binary string `0E1FBA0E00B409CD21B8014CCD21`, which corresponds to the 16-bit DOS mode code which displays the message mentioned earlier. This query took under two minutes, and returned approximately 16.5 million files. The overlap between the two queries was just under 16.3 million files, suggesting approximately 200,000 false positives.[7]

VIII. FUTURE WORK

Although our initial implementation was single-threaded, we are currently exploring parallelization for both index generation and querying. Indexes could easily be divided amongst multiple machines. Even on a single machine, preliminary experiments indicate that a significant reduction in search time can be achieved with multiple cores. Similarly, we expect multi-threading to expedite the index generation process. Preliminary tests show that parallelization can reduce the construction time for an index that took over eleven minutes to build to under two. Finally, we are also exploring other compression techniques such as Adaptive Frame of Reference [AFOR] implementation, [23].

IX. CONCLUSION

In this paper, we propose the use of an inverted N-gram index lacking term offset data as a viable way of expediting exact string matching for large collections of binary files. The novelty of our approach comes from eliminating offset information, and using the index as a way of reducing the number of files that need to be searched iteratively.

Our initial experiments with open-source tools demonstrated the negative impact that the large number of N-grams and large alphabet size had on index construction times and disk space consumption. Attention must alo be paid to minimizing disk seeks and memory usage if multiple terabytes of binary data is to be indexed efficiently. We found that general purpose database tools did not meet the requirements of our problem.

Our analysis suggests that query performance and index size are closely tied to the nature of the input files. Assigning close IDs to files that share many N-grams results in a smaller index due to improved compression efficiency. Furthermore, although the number of false positives is dependent on the exact N-grams of the search string, the number of false positives falls exponentially with query length for most queries[8].

ACKNOWLEDGMENT

This material is based upon work funded and supported by Department of Homeland Security under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense

REFERENCES

[1] http://code.google.com/p/yara-project/
[2] Email from Cory Cohen, 4/26/2012
[3] Min-Soo Kim, Kyu-Young Whang, Jae-Gil Lee, Min-Jae Lee, "n-Gram/2L:A Space and Time Efficient Two-Level n-Gram Inverted Index Structure" In *Proceedings of the International Conference On Very Large Data Bases* 2005, p. 325-336 USA.
[4] Ogawa, Y. Matsuda, T., "An Efficient Document Retrieval Method Using n-gram Indexing," In *System and Computers in Japan*, 2002, Vol. 33, pp. 54-64, USA.
[5] Christopher Manning, Prhabhakar Raghavan, Hinrich Schutze, "Introduction to Information Retrieval," 3rd Edition, Cambridge University Press 2009
[6] Clinton P. Mah and Raymond J. D'Amore. "Complete statistical indexing of text by overlapping word fragments." SIGIR Forum 17, 3 (January 1983), p. 6-16.
[7] Silvestri, Fabrizio. Sorting Out the Document Identifier Assignment Problem." Advances in Information Retrieval. In *Advances in Information Retrieval* 2007. pp. 101-112.
[8] Jiangong Zhang, Xiaohui Long, Torsten Suel, "Performance of Compressed Inverted List Caching in Search Engines." WWW 2008. Beijing China.
[9] Zukowski, M.; Heman, S.; Nes, N.; Boncz, P. "Super-Scalar RAM-CPU Cache Compression." In *Proceedings of the 22nd International Conference on Data Engineering* 2006. Netherlands.
[10] Hao Yan, Shuai Ding, and Torsten Suel. "Inverted index compression and query processing with optimized document ordering." *In Proceedings of the 18th international conference on World wide web (WWW '09).* ACM, New York, NY, USA, 401-410
[11] http://www.postgresql.org
[12] http://www.redis.io
[13] http://www.cert.org
[14] http://fallabs.com/tokyocabinet/
[15] http://www.mongodb.org
[16] http://pgbulkload.projects.postgresql.org/

[7] Actual Windows executables should contain both strings since the code in the second query displays the message.

[8] Assuming that the user has not intentionally constructed a string containing N-grams that are prevalent in most files, but not in the correct overlapping sequence.

[17] Aggarwal, Alok and Rangan, C. and Shibuya, Tetsuo. “Constructing the Suffix Tree of a Tree with a Large Alphabet.” *In Lectures Notes in Computer Science*. Volume 1741. pp. 225-236. 1999
[18] http://www.genometools.org
[19] http://genomewiki.ucsc.edu/index.php/Hg19_Genome_size_statistics
[20] http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf
[21] Donald E Knuth, “The Art of Computer Programming”, Vol3, Section 5.3.3, Minimum-Comparison Selection, 2nd edition
[22] http://blast.ncbi.nlm.nih.gov/blastcgihelp.shtml
[23] http://www.deri.ie/fileadmin/documents/deri-tr-afor.pdf