Grammar Compression of Call Traces in Dynamic Malware Analysis

Takahiro Okumura1,a) Yoshihiro Oyama2,b)

Received: June 14, 2016, Accepted: November 1, 2016

Abstract: A significant number of logs are generated in dynamic malware analysis. Consequently, a method for effectively compressing these logs is required to reduce the amount of memory and storage consumed to store such logs. In this study, we evaluated the efficacy of grammar compression methods in compressing call traces in malware analysis logs. We hypothesized that grammar compression can be useful in compressing call traces because its algorithm can naturally express the dynamic control flows of program execution. We measured the compression ratio of three grammar compression methods (SEQUITUR, Re-Pair, and Byte Pair Encoding (BPE)) and three well-known compressors (gzip, bzip2, and xz). In experiments conducted in which API call sequences collected from thousands of Windows malware were compressed, the Re-Pair grammar compression method was found to outperform both gzip and bzip2.

Keywords: grammar compression, data compression, malware analysis, Windows API, API call sequences

1. Introduction

Dynamic analysis is essential for understanding the behavior of modern malware, which are becoming resistant to static analysis by code obfuscation and packing. In dynamic analysis, the runtime behavior of malware is recorded and examined. Nowadays, a significant amount of malware is being continuously detected and analyzed. For example, Kaspersky Lab reported that it detected approximately 310,000 new malicious files daily in 2015 [5]. When dynamic analysis is applied to these rapidly proliferating malware, a substantial number of analysis logs are generated. These logs require significant amounts of storage and memory.

In this study, we are concerned with Windows platforms and logs of Windows API call sequences. In general, dynamic analysis logs contain various types of data such as sets of files and registries accessed by malware and process trees observed during malware execution. Among them, API call sequences are well-known to be an extremely important clue to understanding and detecting malware behavior [1], [6], [8], [15]. However, logs of API call sequences tend to grow particularly large because most Windows APIs represent basic and small operations, and hence are invoked more frequently than file accesses and network communication.

We consider that the use of compression can significantly reduce the amount of storage and memory consumed to store such logs. Further, API call sequences are particularly suitable for compression for a number of reasons. First, call sequences in general have low information entropy because of limited variations in program behavior and execution of iterations. Second, some malware repeatedly execute the same operation in attack attempts such as network scanning and file encryption. Call sequences invoked in repeated operations are likely to contain many occurrences of common subsequence patterns. Finally, call sequences collected from different malware samples can also contain many common subsequences because multiple variants of a single malware will behave similarly, and recently a considerable number of the malware that have spread globally are actually variants of other malware.

We hypothesized that high compression ratio can be obtained with grammar compression [7], which is a lossless compression method that transforms an input string into context-free grammar generation rules. Grammar compression is known to be useful in compressing data that contain repeated common patterns such as gene sequences. Our observation is that grammar compression has a high potential for effectively compressing call sequences, which have generative and hierarchical structures owing to the execution of nested loops and function calls. Another observation leading us to consider grammar compression is that it enables the application of various operations such as pattern matching without decompressing the data. Pattern matching performance is critical in malware analysis and the ability to avoid decompressing all of the data is crucial. However, to the best of our knowledge, no work has evaluated grammar compression in the compact representation of malware behavior logs application field.

In this study, we evaluated the efficacy of grammar compression against call sequences of Windows APIs included in logs of dynamic malware analysis. Specifically, we conducted experiments involving the compression of API call sequences collected from thousands of malware samples and compared the compris-
2. Grammar Compression

We evaluated three grammar compression methods: SEQUITUR, Re-Pair, and Byte Pair Encoding (BPE). All three methods produce a straight line program (SLP), which is a set of generation rules that derive the given input string only. The generation of the smallest SLP is NP-hard and none of the methods can always bring about the optimum solution.

### 2.1 SEQUITUR

SEQUITUR [13] transforms an input string into context-free grammar generation rules through an online algorithm in which characters are scanned one by one from the beginning of the string to the end. It creates generation rules so that the following conditions are satisfied:

- Digram uniqueness: No character pair must occur more than once in the resulting generation rules.
- Rule utility: All of the resulting generation rules must be used more than once to recover the original input.

Every time SEQUITUR reads a character, it checks whether the last character pair in the scanned part has previously occurred. If it has, SEQUITUR replaces the pairs with a new character and generates a rule to generate the pair from the character. After the scan of the entire input string, SEQUITUR repeatedly finds a rule that is used only once and applies “inlining” to it—i.e., it replaces the source character of the rule in the compressed string with the output characters of the rule, and removes the rule from the resulting set of rules.

Figure 1 shows an example of transformation by SEQUITUR. Symbol S represents the start symbol and the symbols from A to F represent intermediate context-free grammar symbols.

### 2.2 Re-Pair

Re-Pair [9] is based on an offline algorithm that scans an entire input string and then starts to transform it. Re-Pair finds the character pair that occurs most frequently in the currently transformed string. It then replaces the pair with a new character and adds a rule that generates the pair from the new character. It repeats the operation until no character pair in the string occurs more than once. Then, it outputs the current string and the current generation rules as the final result.

Figure 2 shows an example of transformation by Re-Pair. Symbol S represents the start symbol and the symbols from A to F represent intermediate context-free grammar symbols. The rule for the start symbol may have more than two output characters while any other rule has exactly two output characters.

### 2.3 Byte Pair Encoding (BPE)

The BPE [3] grammar compression method is a variant of Re-Pair. BPE fundamentally uses Re-Pair’s method except that the sum of the number of characters in an input string and the number of newly introduced characters is limited to 256. When the sum reaches 256, BPE abandons the replacement of pairs and outputs the current string and the current generation rules as the final result. Although compressed strings generated by BPE are often longer than those generated by Re-Pair, limitations on the number of characters enable BPE to represent all characters compactly with one byte.

3. Experimental Evaluation

3.1 Method

We measured the compression ratio of API call sequences using both grammar and other compression methods.

We used FFRI Datasets [4], which are datasets of dynamic analysis logs of real Windows malware collected by FFRI Inc. All of the currently available datasets were used: FFRI Datasets 2013, 2014, 2015, and 2016. These datasets contain rich information including sequences of API calls invoked in malware execution, as well as information about network communication, file accesses, and registry accesses. FFRI created the datasets by executing malware in virtual Windows environments managed by Cuckoo Sandbox. The operating systems for datasets of 2014 and 2015 are Windows 7 and Windows 8.1 (x64), respectively. The 2016 dataset contains logs collected on Windows 8.1 (x64) and Windows 10 (x64). We used the Windows 8.1 logs. The version of Windows for the 2013 dataset has not been published. Each malware was executed for at most 120 seconds.

We extracted the call-sequence parts from the logs of each malware, and then extracted the API-name section from each call sequence. Subsequently, we concatenated them into one long sequence of API names with a delimiter character inserted between the sequences to be concatenated. Because the call sequences of all malware samples in a dataset were concatenated into one, compression operations affected common call sequences of different malware. Then, we transformed the sequence into an input character string suitable for compression. Figure 3 shows an example of the transformation. Each API name was transformed into a character, and mapping between the API name and the
Table 1  Statistics information.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of call sequences used</th>
<th>Total size of call sequences</th>
<th>Number of API names</th>
<th>Average length of call sequences</th>
<th>Size of input string</th>
<th>Size of mapping table</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016</td>
<td>8,243</td>
<td>569.10 MB</td>
<td>294</td>
<td>4,599</td>
<td>75.85 MB</td>
<td>4.83 KB</td>
</tr>
<tr>
<td>2015</td>
<td>2,970</td>
<td>55.10 MB</td>
<td>141</td>
<td>1,191</td>
<td>3.54 MB</td>
<td>2.37 KB</td>
</tr>
<tr>
<td>2014</td>
<td>2,999</td>
<td>576.16 MB</td>
<td>145</td>
<td>11,663</td>
<td>34.99 MB</td>
<td>2.42 KB</td>
</tr>
<tr>
<td>2013</td>
<td>2,612</td>
<td>28.11 MB</td>
<td>120</td>
<td>652</td>
<td>1.71 MB</td>
<td>2.11 KB</td>
</tr>
</tbody>
</table>

Table 2  Results of compression

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size of input string</th>
<th>gzip</th>
<th>bzip2</th>
<th>xz</th>
<th>SEQUITUR</th>
<th>Re-Pair</th>
<th>BPE</th>
<th>Compressed string (Re-Pair)</th>
<th>Generation rules (Re-Pair)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016</td>
<td>75.85 MB</td>
<td>3.23 MB</td>
<td>1.85 MB</td>
<td>1.11 MB</td>
<td>1.70 MB</td>
<td>1.10 MB</td>
<td>—</td>
<td>338,678</td>
<td>133,294</td>
</tr>
<tr>
<td></td>
<td>(100%)</td>
<td>(4.25%)</td>
<td>(2.44%)</td>
<td>(1.47%)</td>
<td>(2.24%)</td>
<td>(1.45%)</td>
<td>(—)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2015</td>
<td>3.54 MB</td>
<td>0.27 MB</td>
<td>0.13 MB</td>
<td>0.10 MB</td>
<td>0.21 MB</td>
<td>0.13 MB</td>
<td>1.21 MB</td>
<td>38,506</td>
<td>23,913</td>
</tr>
<tr>
<td></td>
<td>(100%)</td>
<td>(7.73%)</td>
<td>(3.74%)</td>
<td>(2.95%)</td>
<td>(6.04%)</td>
<td>(3.57%)</td>
<td>(34.2%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>34.99 MB</td>
<td>2.01 MB</td>
<td>1.07 MB</td>
<td>0.55 MB</td>
<td>1.08 MB</td>
<td>0.62 MB</td>
<td>10.41 MB</td>
<td>176,578</td>
<td>95,028</td>
</tr>
<tr>
<td></td>
<td>(100%)</td>
<td>(5.76%)</td>
<td>(3.06%)</td>
<td>(1.58%)</td>
<td>(3.10%)</td>
<td>(1.77%)</td>
<td>(29.7%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>1.71 MB</td>
<td>0.14 MB</td>
<td>0.08 MB</td>
<td>0.07 MB</td>
<td>0.13 MB</td>
<td>0.08 MB</td>
<td>0.61 MB</td>
<td>24,202</td>
<td>16,509</td>
</tr>
<tr>
<td></td>
<td>(100%)</td>
<td>(8.28%)</td>
<td>(4.53%)</td>
<td>(4.11%)</td>
<td>(7.55%)</td>
<td>(4.61%)</td>
<td>(35.9%)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.2 Results

Table 1 shows call sequence statistics information. In the table, “Number of call sequences used” indicates the number of malware samples used to generate the sequences. “Total size of call sequences” indicates the size of the file that contains the sequences of original API names, where both API names and call sequences are delimited by a one-byte character. “Number of API names” indicates the number of unique API names and does not indicate the total number of API calls. “Average length of call sequences” indicates the average number of calls contained in a call sequence of one malware sample. “Size of input string” and “Size of mapping table” indicate their sizes.

Table 2 shows the experimental results obtained. The columns labelled with the compression methods indicate the sizes of the resulting output files generated by the corresponding programs. The numbers in parentheses indicate the compression ratio. We were unable to apply BPE to the 2016 dataset because its characters were represented with two bytes. The last two columns indicate the length of the resulting string and the number of generation rules for Re-Pair. We included the last two columns to better understand the grammar compression statistics and chose Re-Pair because it performed the best.

Re-Pair achieved the best compression ratio among the various grammar compression methods, and xz achieved the best compression ratio among the other compression methods. Comparing Re-Pair and xz, xz performed better on the 2013–2015 datasets and Re-Pair performed better on the 2016 dataset. It should be noted that SEQUITUR and Re-Pair achieved higher compression ratios than that of gzip. Re-Pair was superior to even bzip2 and xz in several cases. Even when Re-Pair’s compression ratio was lower than that of xz, the difference was quite small (in particular, the difference was 0.2% on the 2014 dataset). The result demonstrates that grammar compression can achieve compression ratios that are as high as, and even higher than, those achieved by widely-used compression methods.

We surmise that the reason why Re-Pair’s compression ratio is sometimes lower than that of xz is the inefficient encoding executed by the program. Grammar compression programs finally encode generation rules into a compressed file. The high com-
pression ability of xz is partially due to the use of an efficient encoding algorithm called range coder. The optimization of encoding operations in the grammar compression programs can improve their compression ability.

Although details are omitted, we briefly report on the performance of compressed pattern matching in which a character string was searched for in the compressed data as-is (i.e., a set of generation rules that represent an input string). We used BPE as the grammar compression method and the KMP automaton for a pattern matching algorithm according to the description in Ref. [17]. The strings used had 5–10 characters containing no regular expression. The result showed that the amount of time elapsed for pattern matching was dominantly correlated with the data lengths and the compressed representation did not significantly degrade the performance. Compared with ordinary pattern matching that scans the original input string, compressed pattern matching achieved a speedup whose degree was close to the reciprocal of the compression ratio.

4. Related Work

Larus [10] proposed a method of generating whole program paths, which are a compressed expression of whole control-flow information recorded in program execution. Larus used SEQUITUR to generate entire program paths. Larus’ work is similar to our work in that both works study the efficacy of grammar compression methods to compress program traces. However, our work differs from Larus’ work in that it provides insights about the compression ratio of API call sequences in dynamic malware analysis.

Walkinshaw et al. [18] used SEQUITUR to recognize repeated patterns in program traces and to visualize them. As in our experiment, they generated an input string by transforming each element of API call sequences into one character. Whereas they adopted grammar compression to support user comprehension of dynamic application behavior, we adopted it to compress logs of dynamic malware behavior.

Li et al. [11] proposed an LZW-based technique for compressing system logs, including antivirus firewall logs. They did not evaluate grammar compression and the format of their logs is unclear.

Many techniques for compressed pattern matching have been proposed [12], [17]. However, the targets of these works are English texts and gene sequences, as opposed to API call sequences.

There has been much work in which API call sequences in the FFRI Datasets have been used to evaluate systems of malware detection or classification (e.g., Ref. [6]). Our work is complementary to such work because it focuses on efficient compression of malware analysis logs.

5. Conclusion and Future Work

In this study, we evaluated the efficacy of grammar compression in compressing sequences of Windows API calls generated in dynamic malware analysis. Our conclusion is that in several cases, grammar compression methods achieved a better compression ratio than other well-known compressors. Further, from another aspect, we consider grammar compression as an attractive option for managing analysis logs because it enables fast execution of security operations such as pattern matching against compressed data as-is.

There are several directions for future work. First, it is necessary to develop an extended method to compress all parts of API calls, including arguments and return values. A technique for effectively encoding and compressing these additional pieces of information is required. Second, it is also necessary to conduct further evaluation using other types of input data such as logs of benign applications and Linux programs.

Acknowledgments This work was supported by JSPS KAKENHI Grant Number 26330080.

References

Takahiro Okumura received a B.E. degree from the University of Electro-Communications in 2016.

Yoshihiro Oyama received a Ph.D. degree in information science from the University of Tokyo, Japan, in 2001. He was an associate professor at the University of Electro-Communications, Japan, from 2006 to 2016. He is currently an associate professor at University of Tsukuba, Japan, since 2016.