Technical Note

Grammar Compression of Call Traces in Dynamic Malware Analysis

Takahiro Okumura[1,a)] Yoshihiro Oyama[2,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 reduces 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 a 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 compres-

[1] The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[2] University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[a)] o1210033@mail.uec.jp
[b)] oyama@cc.tsukuba.ac.jp

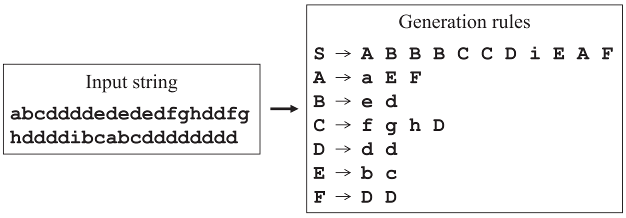


Fig. 1 Example of transformation by SEQUITUR.

sion ratio of various grammar compression methods to other well-known methods. In this study, we focused on compressing sequences of called API names only and did not treat other information such as call arguments and return values.

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 adds a generation 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—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 I 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.

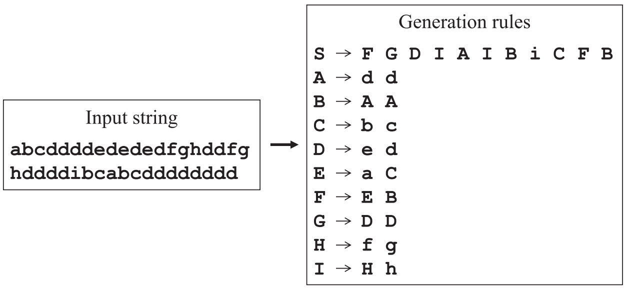


Fig. 2 Example of transformation by Re-Pair.

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.

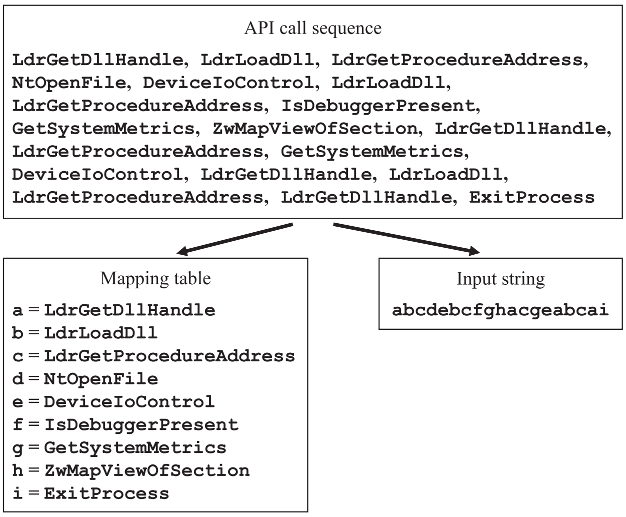
We extracted the call-sequence parts from the logs of each malware, and then extracted the API-name section from each call sequence. Subsequently, we concatenated them into one long sequence of API names with a delimiter character inserted between the sequences to be concatenated. Because the call sequences 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.

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

Table 2 Results of compression

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



```
LdrGetDllHandle, LdrLoadDll, LdrGetProcedureAddress,
NtOpenFile, DeviceIoControl, LdrLoadDll,
LdrGetProcedureAddress, IsDebuggerPresent,
GetSystemMetrics, ZwMapViewOfSection, LdrGetDllHandle,
LdrGetProcedureAddress, GetSystemMetrics,
DeviceIoControl, LdrGetDllHandle, LdrLoadDll,
LdrGetProcedureAddress, LdrGetDllHandle, ExitProcess
```

```
a = LdrGetDllHandle
b = LdrLoadDll
c = LdrGetProcedureAddress
d = NtOpenFile
e = DeviceIoControl
f = IsDebuggerPresent
g = GetSystemMetrics
h = ZwMapViewOfSection
i = ExitProcess
```

```
abcdebcfghacgeabcai
```

Fig. 3 Example of transformation from a sequence of API names into a mapping table and an input string of characters.

character was recorded in a *mapping table*. Because the 2013 to 2015 datasets each contains less than 256 API names, we represented each character with one byte. For the 2016 dataset however, we represented each character with two bytes because it contains more than 256 API names. We also conducted an experiment in which we represented each character in the 2013 to 2015 datasets with two or four bytes, but found that the choice of representation had negligible impact on the compression ratio.

Some of the logs in the FFRI Datasets do not contain any call sequences, and some contain unfinished call sequences. We did not use such logs in our experiment.

We evaluated the abovementioned grammar compression methods and three other well-known compressors: gzip 1.6 (GNU), bzip2 1.0.6, and xz (XZ Utils) 5.1.0alpha. Gzip and xz are based on LZ77-like dictionary coder algorithms. Bzip2 is based on block-sort and move-to-front algorithms. We implemented SEQUITUR, Re-Pair, and BPE using the source code available at Refs. [16], [14], and [2], respectively. We used Ubuntu 14.04 LTS (64 bit) running on Intel Core i7-3770 as our platform.

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 compression 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

[1] Bayer, U., Moser, A., Kruegel, C. and Kirda, E.: Dynamic analysis of malicious code, *Journal of Computer Virology*, Vol.2, No.1, pp.67–77 (2006).

[2] bpe, available from ⟨https://github.com/vteromero/byte-pair-encoding⟩.

[3] Gage, P.: A New Algorithm for Data Compression, *The C Users Journal*, Vol.12, No.2, pp.23–38 (1994).

[4] Kamizono, M., Akiyama, M., Kasama, T., Murakami, J., Hatada, M. and Terada, M.: Datasets for Anti-Malware Research – MWS Datasets 2015 (in Japanese), IPSJ SIG Technical Report, Vol.2015-CSEC-70 (2015).

[5] Kaspersky Lab: Kaspersky Lab's New Malware Count Falls by 15,000 a Day in 2015, as Cybercriminals Look to Save Money, available from ⟨http://www.kaspersky.com/about/news/virus/2015/Kaspersky-Labs-New-Malware-Count-Falls-by-15000-a-Day-in-2015-as-Cybercriminals-Look-to-Save-Money⟩ (2015).

[6] Kawaguchi, N. and Omote, K.: Malware Function Classification using APIs in Initial Behavior, *Proc. 10th Asia Joint Conference on Information Security*, pp.138–144 (2015).

[7] Kieffer, J.C. and Yang, E.-H.: Grammar-based Codes: A New Class of Universal Lossless Source Codes, *IEEE Trans. Inf. Theory*, Vol.46, No.3, pp.737–754 (2000).

[8] Kirat, D. and Vigna, G.: MalGene: Automatic Extraction of Malware Analysis Evasion Signature, *Proc. 22nd ACM Conference on Computer and Communications Security*, pp.769–780 (2015).

[9] Larsson, N.J. and Moffat, A.: Offline Dictionary-Based Compression, *Proc. Data Compression Conference*, pp.296–305 (1999).

[10] Larus, J.R.: Whole Program Paths, *Proc. ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pp.259–269 (1999).

[11] Li, S.-H., Yen, D.C. and Chuang, Y.-P.: A Real-Time Audit Mechanism Based on the Compression Technique, *ACM Trans. Management Inf. Syst.*, Vol.7, No.2 (2016).

[12] Maruyama, S., Tanaka, Y., Sakamoto, H. and Takeda, M.: Context-Sensitive Grammar Transform: Compression and Pattern Matching, *IEICE Trans. Inf. Syst.*, Vol.E93-D, No.2, pp.219–226 (2010).

[13] Nevill-Manning, C.G. and Witten, I.H.: Compression and Explanation using Hierarchical Grammars, *The Computer Journal*, Vol.40, No.2/3, pp.103–116 (1997).

[14] re-pair, available from ⟨https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/re-pair/repair110811.tar.gz⟩.

[15] Rieck, K., Holz, T., Willems, C., Düssel, P. and Laskov, P.: Learning and Classification of Malware Behavior, *Proc. 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp.108–125 (2008).

[16] sequitur, available from ⟨https://github.com/craignm/sequitur⟩.

[17] Takeda, M., Shibata, Y., Matsumoto, T., Kida, T., Shinohara, A., Fukamachi, S., Shinohara, T. and Arikawa, S.: Speeding Up String Pattern Matching by Text Compression: The Dawn of a New Era, *IPSJ Journal*, Vol.42, No.3, pp.370–384 (2001).

[18] Walkinshaw, N., Afshan, S. and McMinn, P.: Using Compression Algorithms to Support the Comprehension of Program Traces, *Proc. 8th International Workshop on Dynamic Analysis*, pp.8–13 (2010).

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.