

Unity in diversity: Phylogenetic-inspired techniques for reverse engineering and detection of malware families

Wei Ming Khoo
University of Cambridge
wmk26@cam.ac.uk

Pietro Lió
University of Cambridge
pl219@cam.ac.uk

Abstract—We developed a framework for abstracting, aligning and analysing malware execution traces and performed a preliminary exploration of state of the art phylogenetic methods, whose strengths lie in pattern recognition and visualisation, to derive the statistical relationships within two contemporary malware families. We made use of phylogenetic trees and networks, motifs, logos, composition biases, and tree topology comparison methods with the objective of identifying common functionality and studying sources of variation in related samples. Networks were more useful for visualising short `nop`-equivalent code metamorphism than trees; tree topology comparison was suited for studying variations in multiple sets of homologous procedures. We found logos could be used for code normalisation, which resulted in 33% to 62% reduction in the number of instructions. A motif search showed that API sequences related to the management of memory, I/O, libraries and threading do not change significantly amongst malware variants; composition bias provided an efficient way to distinguish between families. Using context-sensitive procedure analysis, we found that 100% of a set of memory management procedures used by the FakeAV-DO and “Skyhoo” malware families were uniquely identifiable. We discuss how phylogenetic techniques can aid the reverse engineering and detection of malware families and describe some related challenges.

Keywords-malware analysis; phylogenetics;

I. INTRODUCTION

Today’s malware is written to be persistent. Financial incentives are the dominant motivation for writing and spreading malware, and making sure that the malware remains as long as possible on the victims’ machines. As a result of this, malware exist in families, often numbering in the thousands, in order to constantly evade anti-virus products and operating systems defences. The task of analysing all these variants is resource-intensive and automating the process of reversing and classifying samples is inevitable as the current malware trend continues.

Like human viruses, computer-borne malware has been co-evolving with the operating systems that they target as well as the external environment. Almost all of today’s malware do not exist unobfuscated or unencrypted. Malware samples usually contain a packer which implement code compression and code entry point obfuscation. Other forms of obfuscation frequently employed include virtual machine detection, anti-debugging routines, and code metamorphism.

However, malware is seldom written from scratch. Because new malware variants are usually inspired by previous ones, at some level they show a convergence of functionality. Analysis of the Internet worm Conficker [16] showed that there was a 35% overlap in functional prototypes based on hashes of subroutines and basic blocks, as well as similar domain generation algorithms between variants A and B.

In biology, phylogenetic methods help to extract statistical relationship information from multiple human viral sequences. Phylogeny differs from taxonomy in that taxonomy is a grouping based on shared characteristics, whereas phylogeny assumes a common ancestor and seeks to derive evolutionary relationships between the ancestor and evolved species. Sequences or structures that are conserved between distantly related samples are likely to be important or essential for the functioning of the virus. Conversely, regions that differ are equally interesting for studying evolutionary relationships between samples.

The key questions that we want to address are

- 1) What portions of malware families are most conserved?
- 2) Conversely, what portions of malware families are most diverse?
- 3) Since malware exists in families, can we leverage on malware variants for the task of reverse engineering and detection?

We make the following contributions.

- 1) We developed a tool, *Chronicler*, to perform full execution capture (Section II). *Chronicler* logs all instructions executed, memory modifications, register modifications so that post-hoc program analysis can be done using scripts.
- 2) We describe a framework to abstract and align instruction sequences in order to perform analysis at two levels—the API level and the procedure level (Sections III and IV).
- 3) We demonstrate how phylogenetic trees and networks, tree topology comparisons, motif searches, sequence biases can be used for reverse engineering and detecting two malware families. At the API level, we observe that system calls related to memory manage-

ment, I/O, dynamic library management and threading are most conserved amongst malware variants (Section V-D). At the instruction level, we show how logos can be used for identification of metamorphic code transformations (Section V-C).

- 4) Instead of classifying the whole malware binary, we propose context-sensitive procedure analysis (Section VI). We tested this method on a set of malicious and benign procedures and were able to uniquely identify 100% of the malicious ones.

II. FULL EXECUTION CAPTURE

A tool, Chronicer, was written to track and log the execution using the PIN binary instrumentation framework [12]. The advantage of using dynamic binary instrumentation is it allows full control of the execution and developing analysis tools is easy. However, such frameworks can have a high performance overhead compared to native execution.

Because of the performance overhead, full execution capture is performed, that is, all memory read and writes, register read and writes, disassembly, instruction pointer address and debugging symbols are logged. In order to reduce the size of the resulting logs, they were compressed using the zlib compression library [11]. High compression ratios of about 0.5% were achieved and log sizes were about 10 megabytes per million instructions. Post-hoc analysis of the execution trace is subsequently done using scripts.

The advantage of full execution capture is several analyses can subsequently be performed in parallel. One disadvantage is that the resulting logs are large. Because of this, the file size was limited to 500 Mb per sample, equivalent to about 50 million instructions.

III. PROGRAM ABSTRACTION

Program abstraction is a process by which a program is defined with a representation of its semantics, while hiding its implementation details. Examples include n-grams [6], instruction mnemonics [18], API calls [1], and control flow graphs [5]. As an initial study, we made use of instruction mnemonics and API calls because they can be easily mapped to an alphabet and fed to existing phylogenetic tools.

For example, given an execution trace of instructions,

```
push ebp
mov ebp, esp
mov eax, dword ptr [ebp-0x4]
jmp +0x14
```

it is abstracted as a sequence of mnemonics, i.e.

```
push, mov, mov, jmp
```

ignoring the operands. Each mnemonic is then mapped to a unique alphabet-pair, e.g. `mov` = MO, `push` = PH, `jmp` = JM. The resulting sequence is thus PHMOMOJM. We did not distinguish between addressing modes, e.g. `mov eax,`

`ebx` versus `mov eax, [ebx]` and for instructions with prefixes such as `repeat`, the prefix was used. A similar mapping scheme was used for API calls. Each API was mapped to a unique alphabet-pair; arguments and return values were ignored.

IV. SEQUENCE ALIGNMENT

Sequence alignment is a process of arranging two or more sequences placed one below each other, for their full length (global alignment) or for short subsequences with highest similarity (local alignment). A scoring system rewards with a positive score those positions at which the sequences agree and with a negative score (a penalty) those positions where there is a disagreement (“mutation”) and insertion of a blank (“gap”). Regions of similarity are called homologous regions. Regions that cannot be well aligned, i.e. their scores are similar to that of random sequences, are simply ignored according to this scheme [10]. A commonly used scoring system makes use of substitution matrices which assigns scores based on the evolutionary probability of the mutation. A substitution matrix is a 20 by 20 matrix, where each entry (i, j) is the probability of protein i being substituted with protein j . The point accepted mutation (PAM) substitution matrices were introduced by Dayhoff [2] and were based on observed mutations in 71 families of closely related proteins. Since we are modeling instructions and not proteins, the matrix can be determined empirically or by simply using an identity matrix.

In the context of malware analysis, sequence alignment can be used for tasks such as code normalisation via logos (Section V-C), studying malware evolution via trees and networks (Section V-A) and classification (Section VI), just to name a few.

Sequences can be aligned by leveraging on structural similarities, e.g. aligning along basic blocks. The main disadvantage of doing so is the number of aligned sequences can potentially be large. Secondly, malware writers may not use the standard call-ret calling convention.

Another method of sequence alignment is to align based on functional similarities, by looking at the program semantics, or indirectly by looking at the context, for example determining whether it is running in user space or in kernel space. In the latter, the main idea is that kernel space code, which consists of APIs, should be functionally similar across different malware samples. Furthermore, sequences preceding and following from the same API call should behave in a similar fashion.

The alignment strategy adopted is as follows. The execution trace is first divided into contiguous sequences which can be kernel sequences or user sequences. The boundaries of each sequence are code transitions either from user space to kernel space or from kernel to user space. Subsequently, we conduct analysis at two levels, the first is at the API-level. To perform API-level analysis, each kernel sequence

is mapped to an alphabet-pair and concatenated to form more succinct API sequences. Multiple API sequences are aligned using the CLUSTAL method [7] in which an identity substitution matrix is used in scoring, and the neighbour joining method is used in clustering.

The second level of analysis is performed at the malware procedure level. Procedures are continuous instruction sequences that are executed before and after API calls. For the purposes of analysis, this sequence is treated as a single procedure, which may be made up of more than one function. Procedure-level alignment is similarly performed using the CLUSTAL method. Code that is re-executed, e.g. in a for- or while-loop, appear as repeated sequences. To deal with the difficulty of aligning loops in several different sequences, the instruction pointer address is logged and only the first instance of the instruction is recorded.

A. Case study: Compiler options and optimisation

As an initial experiment, we investigated the use of sequence alignment to compare code generated by different compiler options and optimisation.

As an example, we used the well-known triangle program (Figure 1), which takes the three lengths of the sides of a triangle, and returns what kind of triangle it is. The program was compiled in Microsoft Visual Studio using three different compiler settings—with debugging symbols (`dbg`), default settings (`def`) and optimised for speed (`spd`). The objective of this experiment was to compare the effect of different substitution matrices on the alignment. The goal was to align the structural similarities, represented by the seven `cmp` instructions, and functional similarities, represented by the three `imul` instructions. From a reverse engineering perspective, the `cmp` instructions are important as they reveal the control flow of this program execution; the `imul` instructions are important as they hint at the fact that the program is computing the square of the three arguments. The more these instructions were aligned, the better the overall alignment.

The mnemonic sequences were extracted from the three execution traces from the start to the end of the `classify` subroutine. Figure 2 shows the three sequences before and after alignment for a right-angled triangle with sides 3, 4 and 5. When an identity matrix was used, the seven `cmp` instructions were aligned but only one out of three `imul` instructions were. However, when a PAM matrix was used, only four `cmp` instructions and one `imul` instructions were aligned. Thus the identity matrix out-performed the PAM matrix, which was to be expected as the PAM matrix was modeled after protein substitution probabilities. The identity matrix was used for the rest of our experiments.

V. PHYLOGENY

Reconstruction of molecular phylogenetic relationships is typically done by building a mathematical model describing

```

1 int classify(int a, int b, int c){
2     int kind = UNKNOWN_TRIANGLE;
3     if(a+b<=c || b+c<=a || c+a<=b)
4         return INVALID_TRIANGLE;
5     if(a*a+b*b==c*c || b*b+c*c==a*a || c*c+a*a==b*b)
6         kind |= RIGHT_TRIANGLE;
7     else if (a*a+b*b>c*c || b*b+c*c>a*a || c*c+a*a>b*b)
8         kind |= ACUTE_TRIANGLE;
9     else
10        kind |= OBTUSE_TRIANGLE;
11
12    if(a==b || b==c || c==a)
13        if(a==b && b==c)
14            kind |= EQUILATERAL_TRIANGLE;
15        else
16            kind |= ISOSCELES_TRIANGLE;
17    else
18        kind |= SCALENE_TRIANGLE;
19    return kind;
20 }

```

Figure 1. The triangle program

the evolution of the virus of interest in order to estimate the distance between each sequence pair. A model can be built empirically, using properties calculated through comparisons of observed virus strings, or parametrically, using logic properties of the instructions. An example of the former is to use the expected number of substitutions per position that have occurred on the evolutionary lineages between them and their most recent common ancestor (if any). The main disadvantage of this model is that sequences have to be aligned beforehand.

A. Trees and networks

Distances may be represented as branch lengths in a phylogenetic tree; the extant sequences form the tips of the tree, whereas the ancestral sequences form the internal nodes and are generally not known. Phylogenetic networks are a generalisation of trees for modeling uncertainty about which bifurcation in the tree comes first, thus allowing for different evolutionary paths to be possible. Networks are also useful for modeling horizontal gene transfer, that is, the passage of strings from one child to another instead of from parent to child.

To demonstrate the usefulness of networks over trees, we made use of two datasets. The first was a set of sequences from a procedure “ f_1 ” found in the FakeAV-DO malware family. The FakeAV-DO malware family is a trojan that displays fake alerts that coax users into buying rogue antivirus products. Sequences of “ f_1 ” contain multiple redundant code sequences mechanically inserted by a metamorphic engine described in more detail in Section V-C. To simulate a different model of evolution, a second dataset was generated comprising a set of execution traces from the triangle program, by running it with different arguments.

While some relationships in the trees (Figures 3a and 3c) are obviously closer, e.g. t_{344} and t_{345} , differences in the two datasets are much clearer by looking at the corresponding networks (Figures 3b and 3d). The branches of the

```

(a)
dbg PHMGVSPHPPHLEMGGRPMGMGADCMHLMGADCMHLMGADCMHYMGIMMGIMADMGIMCMHZMGGRMGHMMGCMHZMGCMMHZMGCMMHYMGGRMGMGPPPPPMGPPRE
def PHMGPHMGMGADCMHLMGADCMHLMGADCMHYMGIMMGIMADMGIMCMHZMGGRMGHMMGCMHZMGCMMHZMGCMMHYMGGRMGMGPPRE-----MGPPRE
spd PHMGPHMGPHMGLECMHLLCECMHLLCECMHLMGMGIMIMMGPHIMLECMHZMGCMPHZCMHZCMHZPPPPGRPPRE-----

(b)
dbg PHMGVSPHPPHLEMGGRPMGMGADCMHLMGADCMHLMGADCMHYMGIMMGIMADMGIMCMHZMGGRMGHMMGCMHZMGCMMHZMGCMMHYMGGRMGMGPPPPPMGPPRE
def PHMG--PH-----MGMGADCMHLMGADCMHLMGADCMHYMGIMMGIMADMGIMCMHZMGGRMGHMMGCMHZMGCMMHZMGCMMHYMGGRMGMG-----MGPPRE
spd PHMG--PHMG-----HMGLECMHLL--ECMHL--ECMHLMG--MGLMIMMGPHIMLECMHZMG-----CMPPHZCMHZ--CMHZ-----PPPPGRPPRE

(c)
dbg PHMGVSPHPPHLEMGGRPMGMGADCMHLMGADCMHLMGADCMHYMGIMMGIMADMGIMCMHZMGGRMGHMMGCMHZMGCMMHZMGCMMHYMGGRMGMGPPPPPMGPPRE
def -----PHMGPHMGMGADCMHLMGADCMHLMGADCMHYMGIMMGIMADMGIMCMHZMGGRMGHMMGCMHZMGCMMHZMGCMMHYMGGRMGMGPPRE-----
spd -----PHMGPHMGPHMGLECMHLLCECMHLLCECMHLMGMGIMIMMGPHIMLECMH-----ZMGMPPHZCMHZCMHZPPPPGRPPRE-----

```

Figure 2. Sequence alignment (dbg: with debugging symbols, def: default settings, spd: optimised for speed). (a) Before alignment. (b) After alignment using an identity substitution matrix. (c) After alignment using a PAM substitution matrix.

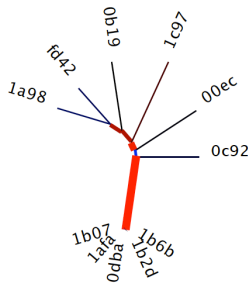


Figure 4. A neighbour joining tree of FakeAV-DO set of procedures F_1 .

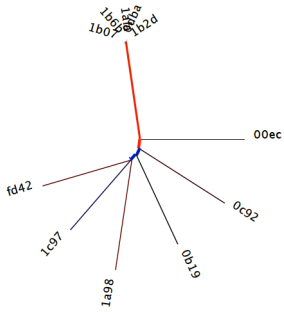


Figure 5. Neighbor joining tree of FakeAV-DO set of procedures F_2 from the same samples as Figure 4.

FakeAV-DO network are widely spread out implying that the evolution is more varied, and that the differences between samples are significant and not simple. Moreover, there are more “ancestor” nodes in the FakeAV-DO network compared to the triangle network, eight versus three, implying that the evolutionary path is more complex and uncertain. We note that the generation of trees and networks depends on the sequence alignment, and in particular the substitution matrix used. Depending on the matrix used, different alignments, and in turn different trees and networks, will result.

B. Comparing tree topologies

Trees constructed from different phylogenetic methods for the same sequences can be compared so as to high-

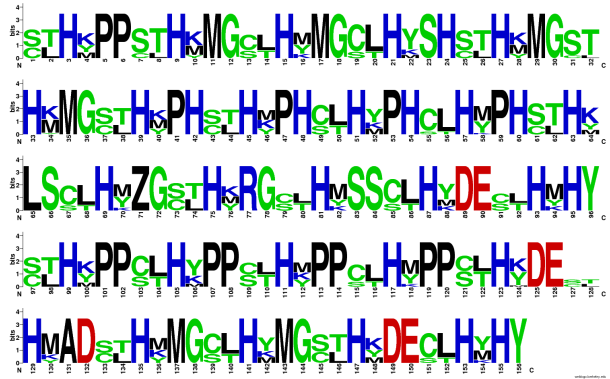


Figure 6. A sequence logo for the FakeAV-DO function “ f_1 ”. Positions with large characters indicate invariant parts of the function, while positions with small characters vary due to code metamorphism.

light the parts of the trees that differ, both in terms of topology and branch length [15]. This method is useful for detection changes, e.g. code rearrangements or replacements, in different sets of procedures. Two sets of homologous procedures, $F_1 = \{f_{1,1}, f_{1,2}, \dots, f_{1,10}\}$ and $F_2 = \{f_{2,1}, f_{2,2}, \dots, f_{2,10}\}$, were extracted from ten FakeAV-DO execution traces. A comparison of the tree topology of F_1 and F_2 (Figures 4 and 5) shows a topological score of 65.4%. The thickness provides an estimate of the differences between the two topologies: Thick lines highlight statistically relevant branch length differences. The group including 1b07, 1afa etc. seem closer in F_2 than in F_1 . Importantly, this method was able to correctly distinguish between two groups of samples despite the high amount of code metamorphism present.

C. Sequence logos

A sequence logo [19] is a graphical method for identifying statistically significant patterns in a set of aligned sequences. The characters of the sequence are stacked on top of each other for each position in the sequence. The height of each letter is made proportional to its frequency, and letters are sorted so that the most common one is on top.

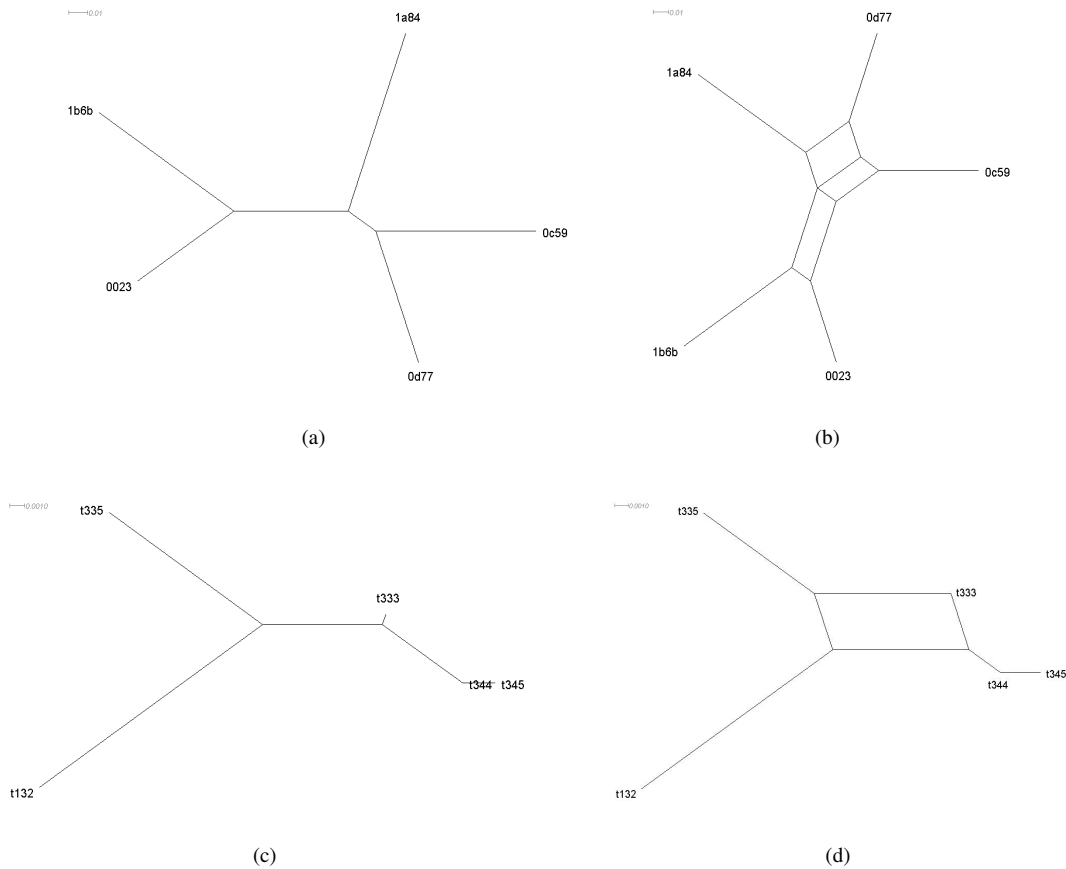


Figure 3. An unrooted phylogenetic (a) tree and (b) network of the FakeAV-DO function “ f_1 ”. An unrooted phylogenetic (c) tree and (d) network of different executions of the triangle program.

One use of a sequence logo is code normalisation. For example, the sequence logo constructed for f_1 (Figure 6) shows that positions 5, 6, 11, 12 and so on do not vary across all sequences, while there was a series of substitutions that were prominent, particularly between *ST* and *CL* at positions 1, 2, 7, 8 and so on. These sequences correspond to the instructions sequences *jmp, stc; jnb* and *clc; jnb*. We observe that the *jnbc* conditional jump is always taken when preceded by *stc*, and similarly *jnbc* always branches when preceded by *clc*. Therefore, code sequences *jmp, stc; jnb* and *clc; jnb* are semantically equivalent and their presence is likely to be the result of a metamorphic engine. Code normalisation was performed by converting all *stc; jnb* and *clc; jnb* sequences to simply *jmp* instructions since they are semantically equivalent. After normalisation, all 17 sequences were identical. This method of code normalisation reduced the number of instructions by about 62% in FakeAV-DO and by about 33% in Skyhoo.

D. Searching for motifs

A motif is a sequence pattern that occurs repeatedly in a group of related sequences. The purpose of a motif search is

to identify consensus sequences which can be subsequently used for alignment. A common method is Gibbs sampling, which draws samples from a joint distribution based on the full conditional distributions of all possible sequences. The advantage of this approach is that sequences do not have to be aligned beforehand.

We used Gibbs sampling to identify motifs in samples from FakeAV-DO and “Skyhoo”, a small but active botnet. The Skyhoo binaries were downloaded over the course of 5 months starting from November 2010. Figure 7 shows the motifs found in the API sequences of FakeAV-DO and Skyhoo samples. These motifs show the patterns formed by four main groups of APIs. The first group was concerned with memory allocation, for example *GlobalAlloc*, *GlobalFree*, *GlobalProtect*; the second group comprised I/O APIs such as *OutputDebugString*, *OpenSCManager*, *OpenService*, *CreateFile*; the third is made up of functions concerned with dynamic libraries such as *LoadLibrary*; the last group comprised threading API such as *GetProcessAddress*. The fact that these patterns are commonly found suggests that they

(a)
 ILLILILILHYHYQRTVHYTVNSILILILILNSILNSILNSILNSILNSILNSILNSILNS

(b)
 WCIGQPTMDMTMDMTMDMTMDMVFIIWHQCAAII
 IGIGITPEDKIMKIKEIKTVTVTVTVTVTVTVTVTV

AI kernel32.dll:CloseHandle
 CA ntdll.dll:NtClose
 DK kernel32.dll:GlobalLock
 DM ADVAPI32.DLL:OpenServiceA
 HQ kernel32.dll:DuplicateHandle
 HY kernel32.dll:LocalAlloc
 IG kernel32.dll:GetModuleHandleA
 IK kernel32.dll:GlobalFree
 IL kernel32.dll:GetProcAddress
 IM kernel32.dll:GetModuleFileNameA
 IT kernel32.dll:VirtualQuery
 IW kernel32.dll:GetCurrentProcess
 KE kernel32.dll:GlobalUnlock
 KI kernel32.dll:GlobalHandle
 NS kernel32.dll:LoadLibraryA
 PE kernel32.dll:GlobalAlloc
 QP ADVAPI32.DLL:OpenSCManagerA
 QR kernel32.dll:LocalFree
 TM kernel32.dll:CreateFileA
 TV kernel32.dll:VirtualProtect
 VF ADVAPI32.DLL:CloseServiceHandle
 WC kernel32.dll:OutputDebugStringA

Figure 7. Common motifs among (a) FakeAV-DO and (b) Skyhoo API sequences.

are not significantly altered from generation to generation and are unique to the malware family.

E. Sequence base composition bias

Comparison of base composition, for example GC-content versus AT-content, in different species can be used to infer their phylogenetic relationships. The same can be true for symmetric or opposite instructions such as `call` and `ret`, or `push` and `pop`. Malware may not follow the standard call-ret procedure, but may instead use them as obfuscation routines. One example is the following code snippet found in Skyhoo.

```
call 0x46cae6
xchg dword ptr [esp], eax
pop eax
```

The stack pointer `esp` is not modified by the `call` because the `pop` restores it; `eax` gets “pushed” unto the stack by `xchg` but gets restored by the `pop`. These three instructions are semantically equivalent to

```
jmp 0x46cae6; nop; nop
```

Let q be the number of instruction Q in a window of n instructions and w the number of W , a symmetric or opposite instruction to Q . The bias can be calculated by the formula

$$bias = \frac{q - w}{q + w}$$

Figure 8 shows the bias in `call` and `ret` instructions for FakeAV-DO, Skyhoo and the Windows Notepad application

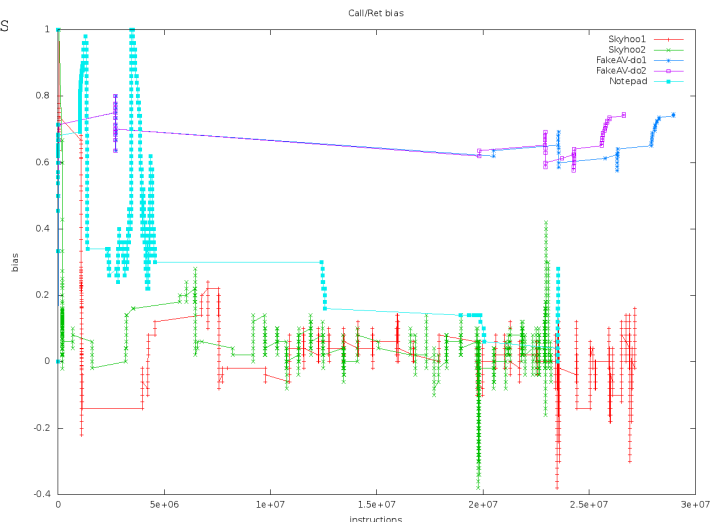


Figure 8. Call/Ret bias in FakeAV-DO, Skyhoo and notepad

for window size $n = 100$. The higher the bias, the higher the proportion of `call` to `ret` instructions. The bias is 1 when there are only `call` instructions and no `ret` instructions, and 0 when there is an equal number of each. The pattern of the bias is more distinctive in the two FakeAV-DO samples than for the Skyhoo samples. Nevertheless when compared to Notepad, we can see noticeable differences in the bias for the three groups of programs.

VI. CONTEXT-SENSITIVE PROCEDURE ANALYSIS

The objective of this set of experiments is to investigate similarities between procedures used by FakeAV-DO and Skyhoo and those used in Notepad, 7zip, WinSCP and the triangle program. While the term “context” used in program analysis typically refers to a set of function pointers on the call stack, we use a different definition. In an execution trace of alternating kernel APIs and user procedures, we define the context of a user procedure to be APIs preceding or succeeding it. These procedures are expected to be similar within the same group of programs. Procedures preceding or succeeding the kernel32 `GetProcAddress` API were chosen as this API was the most commonly used. We tested 19 FakeAV-DO, 24 Skyhoo binaries, plus 7zip, WinSCP, Notepad and the 3 triangle programs. Code was normalised using methods described in Sections V-C and V-E. In total, experiments were performed using six datasets:

- 1) Sequences preceding the `GetProcAddress` API
- 2) Sequences preceding the `GetProcAddress` API, normalised
- 3) Sequences succeeding the `GetProcAddress` API
- 4) Sequences succeeding the `GetProcAddress` API, normalised
- 5) Sequences between two consecutive `GetProcAddress` APIs

Expt.	Num. of sequences			TP0	TP1	TP2	Total
	Benign	FakeAV	Skyhoo				
1	20	94	42	86.1%	99.3%	90.8%	95.0%
2	20	94	42	87.3%	79.0%	89.8%	83.1%
3	24	94	43	85.0%	100%	84.6%	93.8%
4	24	94	43	82.1%	100%	85.2%	93.8%
5	10	67	12	83.9%	100%	100%	98.3%
6	10	67	12	81.6%	100%	100%	97.9%

Table I
CONTEXT-SENSITIVE PROCEDURE DETECTION RATES. TP: TRUE POSITIVE RATE OF THE THREE CLASSES, BENIGN, FAKEAV-DO AND SKYHOO RESPECTIVELY.

6) Sequences between two consecutive GetProcAddress APIs, normalised

We extracted 156 sequences for experiments 1 and 2, 166 for experiments 3 and 4 and 89 for the last two. Half of the sequences were used for training and the remainder for testing. The sequences were aligned using the CLUSTAL algorithm using the identity substitution matrix for alignment with a gap penalty of 0.02 and the neighbour joining clustering algorithm. Test sequences are pairwise aligned with training sequences and classified based on the nearest neighbour classifier. Successful matches were matches where the correct class (benign, FakeAV-DO or Skyhoo) was assigned.

A. Results

Table I shows the results of the 6 experiments. Matches were highest when both contexts were taken into consideration (experiments 5 and 6). Code normalisation did not affect the results much, suggesting that context-sensitive analysis can cope with short `nop`-equivalent code metamorphism.

The high false positive rate for benign samples was because the number of available benign samples were small and, unlike the malware samples, belonged to a variety of programs and thus were under-represented. While this could be fixed with a more comprehensive evaluation, context-sensitive procedure analysis is perhaps more useful for exploratory analysis, providing clues to the malware’s capabilities. For example, one mismatch was between a procedure in WinSCP and a Skyhoo one, implying that the Skyhoo procedure was more similar to WinSCP than it was to, say, 7zip. This may be seen as a false positive, but from the perspective of understanding Skyhoo better this is a useful data point.

VII. DISCUSSION

Our current study was limited to a small number of samples. In future, we intend to pursue this line of inquiry with a larger sample size, as well as to test the robustness of the alignment and clustering methods on other well-known metamorphic engines.

Secondly, we intend to choose a better mnemonic-alphabet mapping as the current simple scheme results in the

same letter being used frequently, which affects alignment in many cases as it is based on single alphabets and not alphabet-pairs. Further work is also needed to consider how the program abstraction can be improved. One possibility is adding control and data flow information, but doing so efficiently. We also hope to investigate the use of substitution matrices other than the identity matrix. As the matrix size is quite large (400 by 400), it is necessary to develop efficient techniques to generate them.

Another limitation is that the VMWare environment can be detected by malware through the use of “red pill” routines [17]. Red pills exploit imperfections in the virtual environment to detect it and are currently being used in the wild. A possible solution is to use hardware-assisted virtualisation as proposed by Dinaburg et al. [3].

Finally, Chronicer currently cannot handle multiple threads and only the parent thread was analysed. Support for multithreaded programs is left as future work.

VIII. RELATED WORK

Goldberg et al. [6] was perhaps the first to study malware phylogenies using suffix trees to construct “phyloDAGs”, or directed acyclic graphs using occurrence counts of 20-gram byte sequences. Erdélyi and Carrera [5] used phylogenetic trees to show the relationships between 6 distinct groups of malware using function call graphs. Karim et al. [8] used “n-perms” in addition to n-grams on opcode sequences to classify malware and to make comparisons with existing classification schemes. This comparison was visualised using phylogenetic trees. Ma et al. [13] studied the diversity of shellcode by looking at exedit distances of instruction byte sequences obtained from code emulation. Phylogenetic trees were constructed to show that shellcode was close-related to the type of vulnerability being exploited. Wehner [20] used phylogenetic trees to show how families of internet worms were related by looking at their normalized compression distance. Our contribution is in expanding the use of phylogenetic tools that, to our knowledge, have not been used in the context of malware analysis before.

Given the threat of malicious software, there has been significant effort invested in understanding malware functionality using static analysis, dynamic analysis or both ([22], [4], [16], [9]). Commonly used techniques are program slicing [21] and dynamic taint analysis [14]. Our objective is similar in spirit in that we also aim to extract functionality from malware, but taking a phylogenetic-inspired approach. We believe that phylogenetic methods can be applied to both static and dynamic analysis and complement existing methods for malware functionality extraction.

IX. CONCLUSIONS AND FUTURE WORK

We defined a framework for using phylogenetics methods to perform malware reverse engineering and detection. Although our results are still in preliminary stages, we have

shown that these techniques can be used to identify unique patterns in contemporary malware families. Moreover, we have shown that these patterns may be identifiable despite code metamorphism. In future, we intend to improve and adapt these techniques, as well as to conduct a more comprehensive study. Ultimately, malware does not exist in a vacuum and studying the unified diversity of malware has the potential to leverage on complexities of the evolving environment, something that malware writers cannot control or predict, to develop new analysis and detection techniques. Lastly, Chronicler is available at

<http://www.cl.cam.ac.uk/~wmk26/chronicler>

X. ACKNOWLEDGEMENTS

We would like to thank Sophos Plc. and Richard Clayton for kindly providing the malware samples, and also to our anonymous reviewers and our shepherd, Marco Cova, for providing extremely detailed constructive criticism.

REFERENCES

- [1] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRÜGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *Proceedings of the 16th annual network and distributed system security symposium (NDSS'09)* (2009).
- [2] DAYHOFF, M. O., AND SCHWARTZ, R. M. Chapter 22: A model of evolutionary change in proteins. In *in Atlas of Protein Sequence and Structure* (1978).
- [3] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on computer and communications security (CCS'08)* (2008).
- [4] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference* (2007).
- [5] ERDÉLYI, G., AND CARRERA, E. Digital genome mapping: advanced binary malware analysis. In *Proceedings of the 15th Virus Bulletin International Conference* (2004), pp. 187–197.
- [6] GOLDBERG, L., GOLDBERG, P., PHILLIPS, C., AND SORKIN, G. Constructing computer virus phylogenies. *J. Algorithms* 26 (1998), 188–208.
- [7] HIGGINS, D. G., BLEASBY, A. J., AND FUCHS, R. Clustal v: improved software for multiple sequence alignment. *Computer Applications in the Biosciences* 8, 2 (1992), 189–191.
- [8] KARIM, M. E., WALLENSTEIN, A., LAKHOTIA, A., AND PARIDA, L. Malware phylogeny generation using permutations of code. *Journal in Computer Virology* 1 (2005), 13–23.
- [9] KOLBITSCH, C., HOLZ, T., KRUEGEL, C., AND KIRDA, E. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *2010 IEEE Symposium on Security and Privacy* (2010).
- [10] LIO, P., AND BISHOP, M. Modeling sequence evolution. *Methods in molecular biology (Clifton, N.J.)* (2008), 255–285.
- [11] LOUP GAILLY, J., AND ADLER, M. The zlib compression library. <http://zlib.net>.
- [12] LUK, C., COHN, R., PATIL, R., KLAUSER, H., LOWNEY, A., WALLACE, G., REDDI, S. V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005).
- [13] MA, J., DUNAGAN, J., WANG, H. J., SAVAGE, S., AND VOELKER, G. Finding diversity in remote code injection exploits. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (IMC'06)* (2006), pp. 53–64.
- [14] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS'05)* (2005).
- [15] NYE, T. M., LI, P., AND GILKS, W. R. A novel algorithm and web-based tool for comparing two alternative phylogenetic trees. *Bioinformatics* 22, 1 (2005), 117–119.
- [16] PORRAS, P., SAIDI, H., AND YEGNESWARAN, V. A foray into conficker's logic and rendezvous points. In *Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET'09)* (2009).
- [17] RUTKOWSKA, J. Red pill or how to detect VMM using (almost one) cpu instruction. <http://invisiblethings.org/papers/redpill.html>, 2004.
- [18] SANTAMARTA, R. Generic detection and classification of polymorphic malware using neural pattern recognition. <http://www.reversemode.com>, 2006.
- [19] SCHNEIDER, T. D., AND STEPHENS, R. M. Sequence logos: A new way to display consensus sequences. *Nucleic Acids Res.* 18 (1990), 6097–6100.
- [20] WEHNER, S. Analyzing worms and network traffic using compression. *J. Comput. Secur.* 15 (August 2007), 303–320.
- [21] WEISER, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.
- [22] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panaroma: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on computer and communications security (CCS'07)* (2007).