MIS Review Vol. 21, Nos. 1/2, September(2015)/March (2016), pp. 15-34
DOI: 10.6131/MISR.2015.2101.02

Metamorphic Malware Detection Using Function Call Graph Analysis

Prasad Deshpande, Mark Stamp
San Jose State University, San Jose, CA

ABSTRACT: *Previous work has shown that well-designed metamorphicmalware can evade many commonly-used malware detection techniques, including signature scanning. In this paper, we consider a previously developed score which is based on function call graph analysis. We test this score on challenging classes of metamorphic malware and we show that the resulting detection rates yield an improvement over other comparable techniques. These results indicate that the function call graph score is among the stronger malware scores developed to date.*

KEYWORDS: *Malware, Function Call Graph, Metamorphic Software*

1. Introduction

Malware is a software that is designed to perform malicious activity (Panda Security, 2011). Examples of such malicious activity range from crashing a system to collecting and infiltrating sensitive data. There are many different categories of malware, including virus, worm, trojan horse, logic bomb, back door, and spyware (Aycock, 2006). In this paper, we use the term virus generically to refer to any type of malware.

According to Symantec (2011), the number of unique malware variants increased from about 286 million to more than 403 million between 2010 and 2011. Also, in 2011, Symantec claimed to have blocked more than 5.5 billion attacks (Symantec, 2011). These numbers give some indication of the scope and prevalence of the malware threat--a massive threat that shows no sign of abating anytime soon.

Code obfuscation is used to obscure the characteristics of code (Xu et al., 2013). Virus writers have developed a variety of code obfuscation techniques, many of which are designed to evade signature detection. Arguably, the most potent such technique is metamorphism, that is, code morphing that changes the internal structure with each infection, while maintaining the essentials of its original function (Shang et al., 2010). Metamorphic generators are readily available, so that even a novice attacker can easily take advantage of this powerful technique. Examples of notable metamorphic generators

include

- NGVCK (Next Generation Virus Creation Kit) (Snakebyte, 2000)
- MPCGEN (Mass Code Generator) (Tips Trik Dan Berbagi Informasi, n.d.)
- G2 (Second Generation Virus Generator) (VX Heavens, n.d.)
- VCL32 (Virus Creation Lab for Win32) (Attaluri et al., 2009)
- MetaPHOR (The Mental Driller, 2002)
- NRLG (NuKE's Random Life Generator) (Symantec, n.d.)
- NEG (NoMercy Excel Generator) (Symantec, n.d.)

Function call graphs have been previously applied to the malware detection problem. For example, Bilar (2007) propose and analyze a mechanism to generate call graphs for malware detection. The paper Shang et al. (2010) proposes an algorithm to determine similarity between function call graphs, while Karnik et al. (2007) uses a cosine similarity metric to measure the overall similarity between code samples, based on call graphs. In Christodorescu et al. (2007), a data mining algorithm is used to construct call graphs via dynamic analysis.

In this paper, we apply a call graph-based score to several challenging classes of metamorphic malware. We compare the results obtained using this call graph approach to previous results obtained using hidden Markov model (HMM) analysis (Wong & Stamp, 2006). These HMM results have previously served as a benchmark for comparing the effectiveness of a wide variety of detection techniques (Attaluri et al., 2009; Kazi & Stamp, 2013; Lin & Stamp, 2011; Runwal et al., 2012; Shanmugam et al., 2013; Sridhara & Stamp, 2012; Tamboli et al., 2014). We show that call graph analysis can yield improved results over many of these previous techniques in these particularly challenging cases.

This paper is the first to test call graph based scoring on such challenging classes of malware. Our results indicate that function call graphs are a powerful technique for scoring malware, and such scores are relatively immune to many common obfuscation techniques.

This paper is organized as follows. Section 2 provides background information on malware and detection techniques, including a discussion of Hidden Markov Models. We also briefly discuss various metamorphic techniques. In Section 3 we discuss call graph analysis and its application to malware detection and, of course, we emphasize the specific implementation that we have chosen. Section 4 contains our experimental results. Finally, Section 5 has our conclusion and suggestions for possible future work.

2. Background

In this section, we first discuss metamorphic malware and various code morphing techniques. Then we briefly discuss Hidden Markov Models (HMMs) and their use in malware detection. HMMs will serve as a benchmark for comparing the call graph scores analyzed in this paper.

2.1 Metamorphic techniques

A metamorphic generator can produce a large number of different generations of code, where the functionality remains the same, but the internal structure differs. Such code obfuscation can alter instructions as well as program data and control flow (Borello & Mé, 2008; You & Kim, 2010). These techniques can be used to evade signature detection, as well as to evade statistical analysis. Next, we briefly consider some code morphing techniques.

2.1.1 Register swap

Register swapping is one of the easiest metamorphic techniques to implement, but it is also one of the least effective. RegSwap, which was arguably the first metamorphic viruses, used this technique exclusively (Szor, 2005). Table 1 shows code fragment from different generation of W95/RegSwap virus.

Table 1 Two Generations of RegSwap

pop edx	pop edx
mov edi, 0004h	mov ebx, 0004h
mov esi, ebp	mov edx, ebp
mov eax, 000Ch	mov edi, 000Ch
add edx, 0088h	add eax, 0088h
mov ebx, [edx]	mov esi, [eax]
mov [esi+eax*4+00001118], ebx	mov [edx+edi*4+00001118], esi

Source: Szor, 2005

2.1.2 Transposition

Subroutine permutation is another elementary code morphing technique. If there are n subroutines, then it is trivial to generate $n!$ different metamorphic copies by simply permuting the order of the subroutines. BadBoy and W32/Ghost are two viruses that employ subroutine permutation (Szor, 2005). BadBoy has 8 subroutines, so it can generate 8! = 40320 different variants.

More generally, if two instructions (or groups of instructions) are independent of each other then their order can be changed. Even more general transposition can be used, provided jump instructions are inserted to preserve the order of code execution.

2.1.3 Dead code insertion

Dead code insertion can be a highly effective morphing strategy. Dead code may or may not be executed; if such code is executed, care must be taken so that it has no effect on the functioning of the program. Examples of dead code insertions are given Table 2 Note that none of the instructions in Table 2 change the value of the register.

Table 2 Example of Dead Code

Instruction	Description
add Reg,0	Add value 0 to register
mov Reg,Reg	Transfer register value to itself
or Reg, 0	Logical OR operation of register with 0
nop	No operation

Source: Szor and Ferrie (2001)

Dead code insertion is useful for evading signature detection and can also aid in evading statistical-based detection. Dead code insertion is used, for example, in Win95/Zperm (Szor, 2005) and also in the experimental metamorphic worm MWOR, which is analyzed in Sridhara and Stamp (2012).

2.1.4 Instruction substitution

An instruction or group of instructions can be substituted for another equivalent instruction or group of instructions. For example, the instruction xor eax, eax can be replaced by sub eax, eax. Instruction substitution can be highly effective, but is relatively difficult to implement. Instruction substitution is used extensively in W32/MetaPHOR (Szor, 2005) and also to some extent in the MWOR worm Sridhara and Stamp (2012).

2.1.5 Formal grammar mutation

A code morphing engine can be viewed as nondeterministic automata, where transitions are possible from every symbol to every other symbol (Zbitskiy, 2009). Here, the set of symbols consists of the set of possible instructions. By formalizing mutation techniques in this way, we can apply formal grammar rules and create malicious copies with large variation; see Zbitskiy (2009) for an example.

2.1.6 Host code mutation

Some viruses mutate the code of the host along with their own code (Konstantinou & Wolthusen, 2008). Win95/Bistro is an example of malware that uses this concept of host code mutation Szor (2000).

2.1.7 Code integration

Win95/Zmist implements a "code integration" technique. Specifically, Zmist decompiles a portable executable (PE) file, inserts itself into the code of the file, regenerates the code and data references, and recompiles the executable (Szor & Ferrie, 2001).

2.2 Hidden Markov model based detection

Hidden Markov Model (HMM) analysis has proven useful in a wide array of fields, ranging from speech recognition (Rabiner, 1989) to software piracy detection (Kazi & Stamp, 2013). Previous research has shown that HMMs can be a highly effective tool for detecting metamorphic malware (Attaluri et al., 2009; Lin & Stamp, 2011; Wong & Stamp, 2006). Since HMMs have been widely studied, we use an HMM-based score as the benchmark for comparison with the call graph score considered in this paper.

An HMM includes a "hidden" Markov process, and a sequence of observations that are probabilistically related to this hidden process. We can train an HMM for a given sequence of observations. Then we can score a sequence against this trained model to determine how closely it matches the training data. The relevant notation commonly used in HMMs appears in Table 3.

A generic HMM is illustrated in Figure 1, where X_t and $\mathcal{O}_t$ represent the (hidden) state sequence and the observation sequence, respectively. The underlying Markov process is driven by the A matrix. The observations $\mathcal{O}_t$ are related to the current state of the Markov

Table 3 HMM Notation

Symbol	Description
T	length of the observed sequence
N	number of (hidden) states in the model
M	number of distinct observation symbols
$\mathcal{O}$	observation sequence ($\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{\mathrm{T}}-1$)
A	$N \times N$ state transition probability matrix
B	$N \times M$ observation probability matrix
π	$1 \times N$ initial state distribution matrix

Source: Stamp (2015).

process by probability distributions contained in the B matrix. The matrices A, B, and π are row stochastic, that is, the elements of each row satisfy the conditions of a probability distribution.

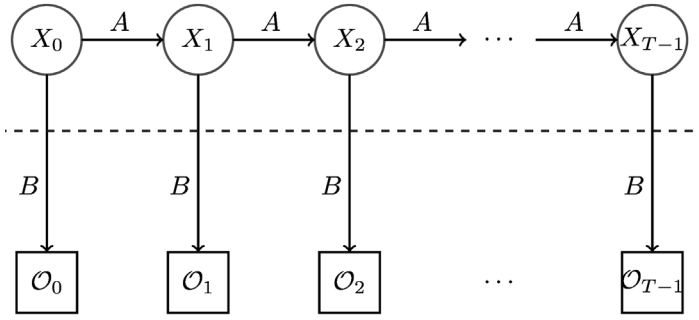


Figure 1 Generic Hidden Markov Model (Stamp, 2015)

For the metamorphic malware detection problem considered in Wong and Stamp (2006), opcodes are extracted from several members of a given metamorphic family. These opcode sequences are concatenated to form a sequence $\mathcal{O}$, and an HMM is trained on $\mathcal{O}$. To score a given file, its opcode sequence is extracted and scored against the trained HMM. The results in Wong and Stamp (2006) indicate that this technique is highly effective at detecting hacker-produced metamorphic code.

These results have been confirmed and further analyzed in a substantial body of subsequent research, including Attaluri et al. (2009), Kazi and Stamp (2013), Lin and Stamp (2011), Runwal et al. (2012), Shanmugam et al. (2013), Sridhara and Stamp (2012) and Tamboli et al. (2014). Consequently, we use HMM scoring as a benchmark to measure the effectiveness of the call graph technique considered here.

3. Call Graph Analysis

In this section, we first discuss previous malware detection work based on using call graph analysis. Then we discuss function call graphs in general, and explain in detail the scoring algorithm used in this research.

3.1 Previous work

Malware writers have developed a variety of techniques for evading signature detection (Aycock, 2006). In contrast to signature detection, functional call graph analysis relies on higher-level structure, that should be more difficult to obfuscate. The purpose

of this research is to determine the effectiveness of call graph-based techniques when confronted with advanced metamorphic malware. Such malware easily defeats signature scanning and, if properly constructed, can also evade statistical-based detection (Sridhara & Stamp, 2012).

A function call graph is created from the disassembled code of an executable as follows. Each function is represented by a vertex, with directed edges represent the caller-callee relationships between functions (Xu et al., 2013). In addition, edge "weights" can be considered, which can be based on opcode analysis and graph coloring techniques. Once such graphs have been constructed, determining the similarity between programs reduces to determining the similarity between their function call graphs.

We implemented the technique given in (Xu et al., 2013). We apply this technique to several advanced metamorphic generators. Our test results are given in Section 4. But first we discuss the process used to construct function call graphs and to measure the similarity of such graphs.

3.2 Function Call Graph Construction

An graph can be represented as $G = (V, E)$, where V is the set of vertices and E is the set of edges. For a function call graph, the vertices represent functions while the edges represent caller-callee relationships between functions (Xu et al., 2013). The functions in V are classified as one of two types, namely, local functions or external functions. Local functions are contained within the executable, while external functions are system or library functions. After disassembling an exe, functions begin with sub_xxxxxx and end with sub_xxxxxx where "xxxxxx" represents the name of the function. For local functions, the name of the called function is also found within the executable, while external functions names are not.

Given an executable, we first disassemble it using IDA Pro (Hex-Ray, n.d.). From the resulting assembly code, we search for functions and extract relevant information for each. Once the relevant information has been extracted for all functions, the function call graph is constructed. Figure 2 shows part of the function call graph for the virus Win32. Bolzano. As can be seen in Figure 2, the graph consists of local functions (those with names of the form sub_xxxxxx) and external functions, such as GetVersion. Note that local functions can call external functions, but an external function cannot call a local function.

As in Shang et al. (2010), we use a breadth first search (BFS) to determine caller-callee relationships between functions. In a BFS, we start from a root node and process successive levels. For our experiments, the entry point function serves as the root node and the the algorithm used is a straightforward BFS; for additional details, see Shang et al. (2010) or Deshpande (2013).

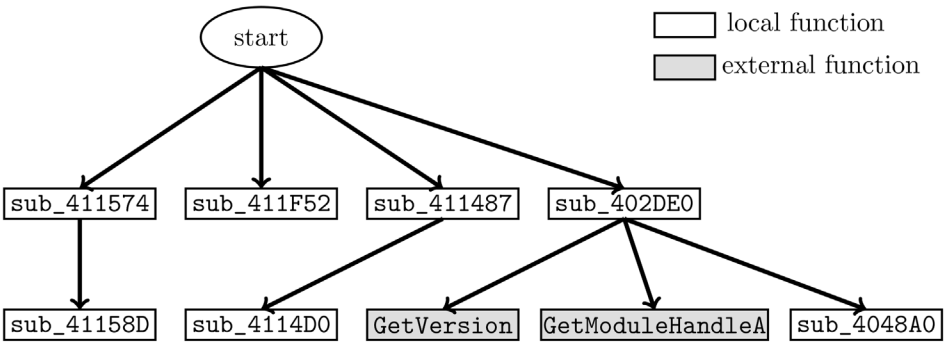


Figure 2 Function Call Graph in Win32.Bolzano (Xu et al., 2013).

3.3 Function call graph similarity

Once function call graphs have been constructed, we then determine the simi-larity of the corresponding programs by measuring the similarity of their graphs. External functions are matched using their symbolic names, since these will be the same across different programs. However, the symbolic names used for local functions need not be the same across metamorphic code variants. Consequently, we must analyze local functions to determine their similarity across different programs. We use three different techniques to match local functions.

3.3.1 Matching external functions

External functions have the same name across all executables and make no further calls within the call graph (Carrera & Erdelyi, 2004). Hence, in terms of the call graph, external functions have in-degree 1 and out-degree 0, and these functions can be matched based simply on their symbolic names. For example, the GetVersion function in one function call graph can be matched with same function in any other call graph.

Given function call graphs G_1 and G_2, we extract the external functions from each. As noted above, these functions are easily determined. Then both sets of external functions are compared, and for any common symbolic names, the corresponding vertex is saved to a common external vertex set, which will be used for scoring. Details of the scoring process are given in Section 3.3, below.

3.3.2 Local function similarity based on external functions

The first method that we use to find matching local functions consists of match-ing common external function calls. All local functions in the graphs G_1 and G_2 are compared and we simply tabulate matches in the external functions called. If the number of such matches is two or greater, the corresponding local functions are considered to match by this criteria, so they are saved to a common local set.

3.3.3 Local function similarity based on opcode sequences

Local functions that do not match based on external functions are compared based on their opcode sequences. There are many different opcode-based similarity techniques (Attaluri et al., 2009; Runwal et al., 2012; Shanmugam et al., 2013; Wong & Stamp., 2006). So that our results will be consistent with previous work on call graph similarity, here we use the opcode similarity technique in Xu et al. (2013), which we now describe in detail.

Each vertex in the call graph is "colored" depending on the instructions used. Functions are considered to match provided their "colors" match. In case of matches, the score is computed using cosine similarity.

To make this score more robust against morphing techniques such as code substitution, we classify all X86 instructions into one of 15 categories, according to their function (Xu et al., 2013). These categories are listed in Table 4.

A 15-bit color variable is associated with each vertex corresponding to a local function in the graph. If an opcode of type *Ci* appears in the function, bit *i* of the color variable is set -- if no such opcode exists in the function, color bit *i* is 0. There is also a corresponding vector that holds the count of the number of instructions in each class. For example, the first column in Table 5 contains a function from Win32.DarkMoon. The

Table 4 x86 Instruction Classification

Class	Type	Description
C_1	Data	data transfer such as mov
C_2	Stack	stack operation
C_3	Port	in and out
C_4	Lea	destination address transmit
C_5	Flag	flag transmit
C_6	Arithmetic	shift, rotate, etc.
C_7	Logic	bitbyte operation
C_8	String	string operation
C_9	Jump	unconditional transfer
C_{10}	Branch	conditional transfer
C_{11}	Loop	loop control
C_{12}	Halt	stop instruction
C_{13}	Bit	bit test and bit scan
C_{14}	Processor	processor control
C_{15}	Float	floating point operation

Source: Xu et al. (2013).

second column in Table 5 lists the opcode, while the third column is the category of the opcode, as found in Table 4.

The color variable and vector of counts from Table 5 appear in Table 6. These are used for computing a score based on cosine similarity, which we now discuss.

Table 5 Local Function from Win32.DarkMoon

Assembly code	Opcode	Category
sub 4059DC proc near	—	—
push ebx	push	C_2
mov ebx,eax	mov	C_1
cmp ds:byte 4146C1, 0	cmp	C_6
jz short loc 405A04	jz	C_{10}
push 0	push	C_2
call SwapMouseButton	call	C_9
mov ds:byte 4146C1, 0	mov	C_1
mov eax,ebx	mov	C_1
mov edx,offset dword 405A28	mov	C_1
call sub 403DEC	call	C_9
pop ebx	pop	C_2
retn	retn	C_9
push 0FFFFFFFFh	push	C_2
call SwapMouseButton	call	C_9
mov ds:byte 4146C1, 1	mov	C_1
mov eax,ebx	mov	C_1
mov edx,offset dword 405A28	mov	C_1
call sub 403DEC	call	C_9
pop ebx	pop	C_2
retn	retn	C_9
sub 4059DC endp	—	—

Source: Xu et al. (2013).

Table 6 Color Vector of Win32.DarkMoon

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}
color	1	1	0	0	0	1	0	0	1	1	0	0	0	0	0
count	7	5	0	0	0	1	0	0	6	1	0	0	0	0	0

Source: Xu et al. (2013).

Let $X = (x_1, x_2, \ldots, x_{15})$ and $Y = (y_1, y_2, \ldots, y_{15})$ be the vectors of counts from two functions. Then the cosine similarity between these two vectors is computed as

$$\text{sim}(X, Y) = \frac{X \cdot Y}{||X||\,||Y||} = \frac{x_1 y_1 + x_2 y_2 + \cdots + x_{15} y_{15}}{\sqrt{x_1^2 + x_2^2 + \cdots + x_{15}^2}\,\sqrt{y_1^2 + y_2^2 + \cdots + y_{15}^2}} \tag{1}$$

If the color vectors match exactly, and the cosine similarity between the corresponding count vectors is greater than a predetermined threshold, then the functions are considered a match. In our experiments, we use the same parameters for scoring as in Xu et al. (2013).

3.3.4 Local Function Similarity Based on Matched Neighbors

If two functions match, then it is more likely that functions corresponding to neighboring vertices in the function call graphs should match. For example, suppose that in Figure 3, vertex *A* has been matched with vertex *B*. Then there is a higher likelihood that one or more of vertices *U*, *V* will match with one or more of the vertices *X*, *Y*, *Z*.

Because of this higher likelihood of a match, we alter the score computation for such neighboring vertices. Successors and predecessors of previously matched functions are scored using a slightly relaxed version of the color-based score discussed in the previous section. The difference here is that there is no requirement that the color vectors match, that is, we compute the cosine similarity score in (1), regardless of the color vectors. In addition, we use a slightly different threshold for the similarity score.

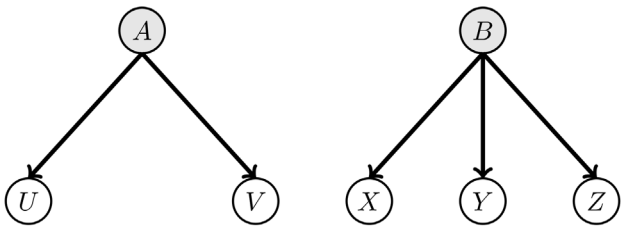


Figure 3 Successors Functions of Matched *A* and *B*

3.3.5 Similarity Score

Given two function call graphs G_1 and G_2, we determine all common vertices using the function matching algorithms outlined in Sections 3.3 through 3.3, above. Once we have found all common vertices, we determine the common edges. Suppose vertices *A* and *B* from G_1 have been matched to vertices *C* and *D* from G_2, respec-tively. If there is

an edge between A and B in G_1 and an edge between C and D in G_2, then G_1 and G_2 are said to have a common edge. Let common edge (G_1, G_2) be the set of such common edges. Then the similarity between two function call graphs is calculated as Xu et al. (2013).

$$\text{sim}(G_1, G_2) = 100 \cdot \frac{2\,|\text{common\_edge}(G_1, G_2)|}{|E(G_1)| + |E(G_2)|} \tag{2}$$

where $E(Gi)$ is the edge set of the graph Gi.

4. Experiments

In this section, we analyze the performance of the similarity scoring algorithm discussed in Section 3. We test the technique on two families of metamorphic malware, namely, the Next Generation Virus Generation Kit (NGVCK) Snake-byte (2000) and the experimental MWOR worms developed and analyzed in Sridhara and Stamp (2012). The NGVCK viruses have previously been shown to be highly metamorphic, but detectable using statistical-based techniques (Runwal et al., 2012; Shanmugam et al., 2013; Toderici & Stamp, 2013; Wong & Stamp, 2006). The MWOR worms were designed to be highly metamorphic and to evade statistical-based detection -- and they do successfully evade such detection (Sridhara & Stamp, 2012). Both of these metamorphic families have been used in studies of several other malware scoring techniques (Attaluri et al., 2009; Baysa et al., 2013; Lin & Stamp, 2011; Runwal et al., 2012; Shanmugam et al., 2013; Toderici & Stamp, 2013; Wong & Stamp, 2006), and hence they provide a basis for judging the effectiveness of the call graph similarity score considered here.

4.1 Test Data

Our test data consists of 50 NGVCK virus files and a total of 120 MWOR files. The MWOR worms have an adjustable “padding ratio” parameter that specifies the fraction of dead code to worm code. For example, a padding ratio of 2.0 means that each worm has twice as much dead code as actual functioning worm code. The dead code is selected from benign files and, at higher ratios; it serves to effectively defeat statistical-based detection techniques (Sridhara & Stamp, 2012). We consider distinct sets of MWOR worms with padding ratios of 0.5, 1.0, 1.5, 2.0, 2.5, and 3.0. For the NGVCK viruses we use 50 Cygwin utility files as representative examples of benign files. Since MWOR is a Linux worm, we use a set of 20 Linux library files for the representative benign set for the MWOR experiments. These data sets are consistent with those used in previous related research (Baysa et al., 2013; Runwal et al., 2012; Shanmugam et al., 2013; Wong & Stamp, 2006). In all experiments, we score all pairs of malware samples with each other, and we score all pairs consisting of one malware sample and one benign file.

4.2 Evaluation Criteria

To evaluate our results, we use Receiver Operating Characteristic (ROC) curves. To construct an ROC curve, we plot the fraction of true positives versus the fraction of false positives as the threshold varies through the range of scores. The area under the curve (AUC) provides a single measure that enables us to directly compare experimental results. An AUC of 1.0 indicates ideal separation, that is, we can set a threshold for which no false positives of false negatives occur. At the other extreme, an AUC of 0.5 indicates that the binary classifier performs no better than flipping a coin.

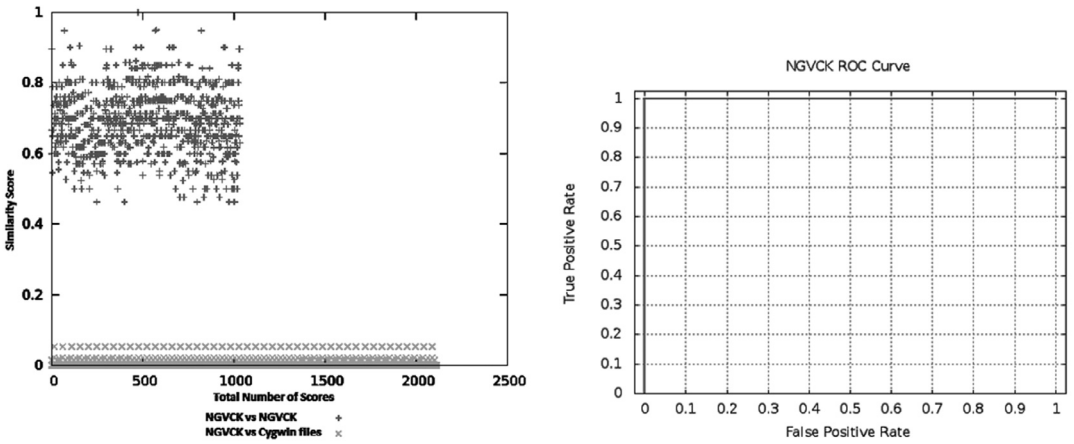
4.3 Test Results

4.3.1 NGVCK

First, we tested the call graph based scoring technique on NGVCK viruses. A scatterplot of the resulting scores is given in Figure 4 (a), and the corresponding ROC curve appears in Figure 4 (b). In this case, the AUC is clearly 1.0, as we have ideal detection.

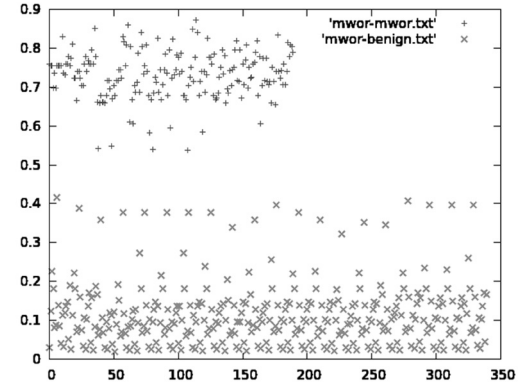
4.3.2 MWOR

Next, we tested the call graph score on MWOR worms, using a wide range of padding ratios. Recall that the MWOR padding ratio is the fraction of dead code to functional worm code. Figure 5 shows the similarity scores for MWOR worms, where the padding ratio ranges from 0.5 to 3.0. These results show that for padding ratios of 2.0 or less, we obtain ideal classification in each case. However, for padding ratios of 2.5 and above, there will be some misclassifications, regardless of the threshold.

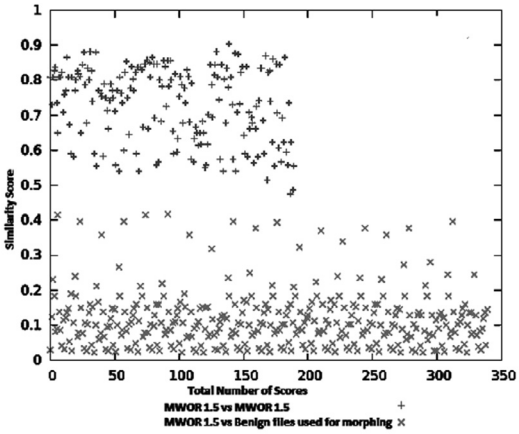


(a) Similarity scores (b) ROC curve

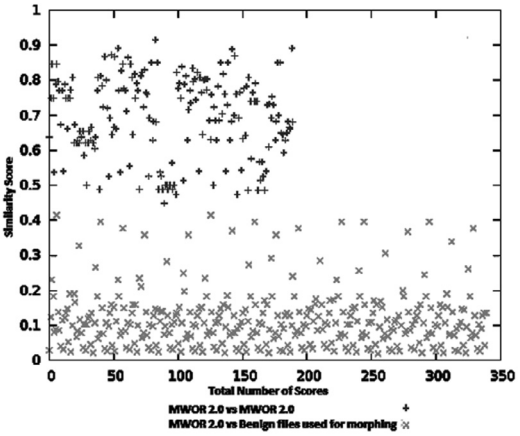
Figure 4 Call Graph Similarity for NGVCK Virus Family



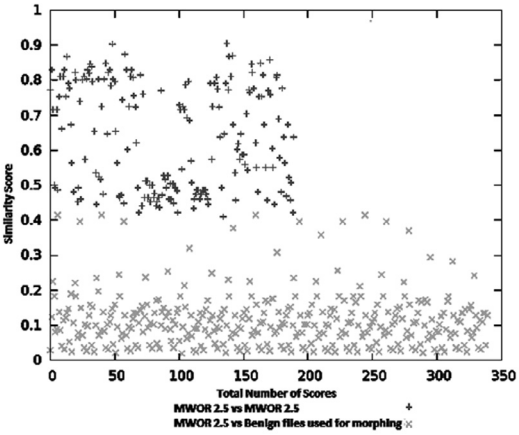
(a) Padding ratio 1.0



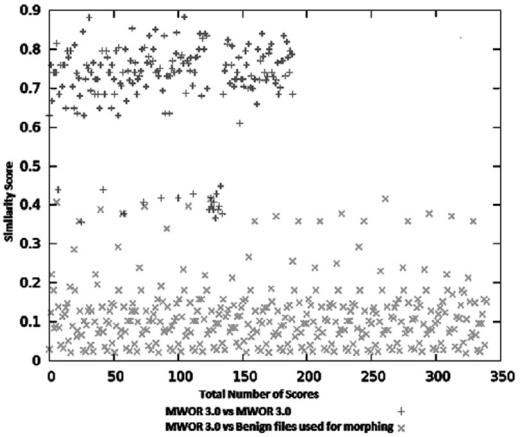
(b) Padding ratio 1.5



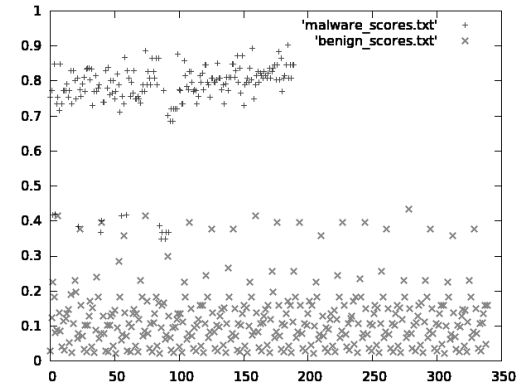
(c) Padding ratio 2.0



(d) Padding ratio 2.5



(e) Padding ratio 3.0



(f) Padding ratio 4.0

Figure 5 MWOR Similarity Scores

ROC curves corresponding to the scores in Figure 5 were constructed, and the AUC for each computed. The first column of Table 7 contains the AUC statistic for each of the resulting ROC curves.

4.3.3 Comparison with Previous Work

Next, we compare the results obtained using the call graph score to an HMM-based score. As previously mentioned, this HMM score has served as a benchmark for several previous studies on malware detection and hence provides a useful measure of the success of the call graph score, relative to previous work.

For the MWOR worms, a direct comparison (in terms of the AUC statistic) is provided in Table 7, where the HMM results are taken from Sridhara and Stamp (2012) and the "simple substitution" results are from Shanmugam et al. (2013) (which itself improved on the HMM score for the MWOR family). From these results, we see that the call graph technique is superior to both of these other techniques for padding ratios of 1.5 or greater. In Figure 6, we have plotted the results from Table 7 in the form of a bar graph, which clearly shows the robustness of the call graph score with respect to common morphing techniques.

5. Conclusion and future work

We implemented a function call graph technique and applied it to the malware detection problem. Opcode analysis and graph coloring techniques are employed to compute this score.

We tested this similarity score on two challenging metamorphic malware families. The results show that the function call graph score outperforms a straightforward HMM-

Table 7 MWOR AUC Comparison

Padding ratio	Area under the ROC curve (AUC)		
	call graph	HMM	simple substitution
0.5	1.0000	1.0000	1.0000
1.0	1.0000	0.9900	1.0000
1.5	1.0000	0.9625	0.9980
2.0	1.0000	0.9725	0.9985
2.5	0.9999	0.8325	0.9859
3.0	0.9989	0.8575	0.9725
4.0	0.9979	0.8225	0.9565

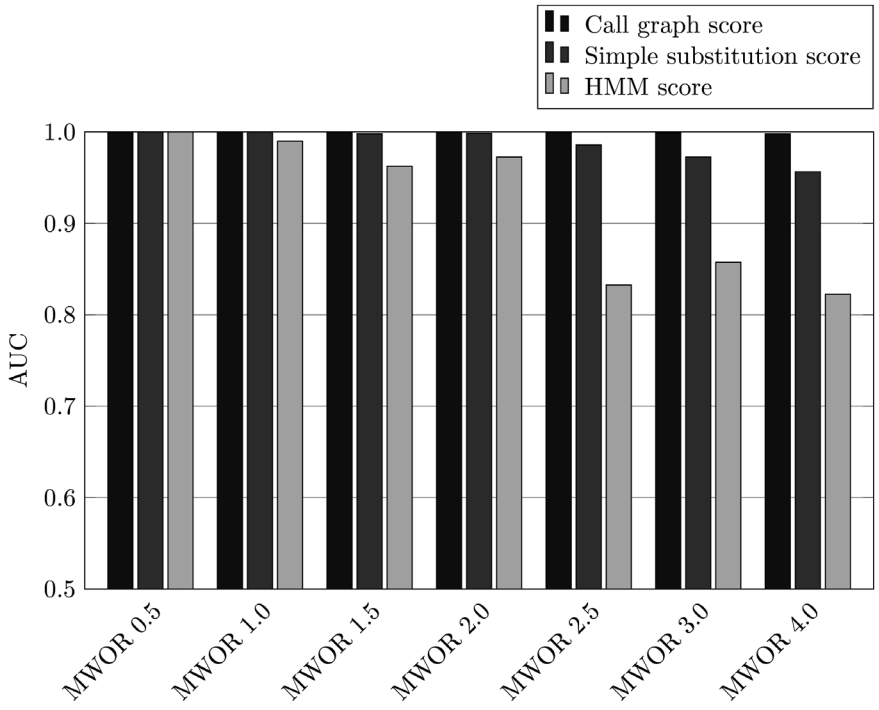


Figure 6 AUC for Function Call Graph, Simple Substitution, and HMM Scores

based score and a "simple substitution" score. This is impressive, since the HMM score has served as a benchmark in several previous studies, and it has proven difficult to significantly improve on the HMM results.

Future work could focus on possible improvements to call graph score technique considered in this paper. Specifically, the step where we match similar functions is worth reconsidering. Possible alternatives to the graph coloring approach used here include any of a variety of additional statistical techniques, such as HMM analysis (Wong & Stamp, 2006), chi-squared statistics (Toderici & Stamp, 2013), and the "simple substitution" distance in Shanmugam et al. (2013). In addition, structural techniques such as the entropy-based score in Baysa et al. (2013) and Sorokin (2011) or the compression-based score in Lee et al. (2015) could prove more robust than scores that rely directly on opcode-based analysis.

References

Attaluri, S., McGhee, S. and Stamp, M. (2009), 'Profile hidden Markov models and metamorphic virus detection', *Journal in Computer Virology*, Vol. 5, No. 2, pp. 151-169.

Aycock, J.D. (2006), *Computer Viruses and Malware*, Springer-Verlag, New York, NY.

Baysa, D., Low, R.M. and Stamp, M. (2013), 'Structural entropy and metamorphic malware', *Journal of Computer Virology and Hacking Techniques*, Vol. 9, No. 4, pp. 79-192.

Bilar, D. (2007), 'On callgraphs and generative mechanisms', *Journal in Computer Virology*, Vol. 3, No. 4, pp. 285-297.

Borello, J. and Mé, L. (2008), 'Code obfuscation techniques for metamorphic viruses', *Journal in Computer Virology*, Vol. 4, No. 3, pp. 211-220.

Carrera, E. and Erdelyi, G. (2004), 'Digital genome mapping-advanced binary malware analysis', *Proceeding Virus Bulletin Conference*, City, State, pp. 187-197.

Christodorescu, M., Jha, S. and Kruegel, C. (2007), 'Mining specifications of malicious behavior', *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, pp. 5-14.

Deshpande, P. (2013), 'Metamorphic detection using function call graph analysis', Unpublished master thesis, San Jose State University, San Jose, CA.

Hex-Ray. (n.d.), 'IDA: about', available at: http://www.hex-rays.com/products/ida/index.shtml (accessed on February 14 2017).

Karnik, A., Goswami, S. and Guha, R. (2007), 'Detecting obfuscated viruses using cosine similarity analysis', *Proceedings of the First Asia International Conference on Modelling Simulation*, Phuket, Thailand, pp. 165-170.

Kazi, S. and Stamp, M. (2013), 'Hidden Markov models for software piracy detection', *Information Security Journal: A Global Perspective*, Vol. 22, No. 3, pp. 140-149.

Konstantinou, E. and Wolthusen, S. (2008), 'Metamorphic virus: analysis and detection', Technical Report RHUL-MA-2008-02, Department of Mathematics, Royal Holloway, University of London , Egham, UK.

Lee, J., Austin, T.H. and Stamp, M. (2015), 'Compression-based analysis of metamor-phic malware', *International Journal of Security and Networks*, Vol. 10, No. 2, pp. 124-136.

Lin, D. and Stamp M. (2011), 'Hunting for undetectable metamorphic viruses', *Journal in Computer Virology*, Vol. 7, No. 3, pp. 201-214.

Panda Security (2011) 'Virus, worms, trojans and backdoors: other harmful relatives of viruses', available at: http://www.pandasecurity.com/homeusers-cms3/security-info/about-malware/generalconcepts/concept-2.html (accessed on February 14 2017).

Rabiner, L.R. (1989), 'A tutorial on hidden Markov models and selected applications in speech recognition', *Proceeding of the IEEE*, Vol. 77, No. 2, pp. 257-286.

Runwal, N., Low, R. and Stamp, M. (2012), 'Opcode graph similarity and metamor-phic detection', *Journal in Computer Virology*, Vol. 8, No. 1-2, pp. 37-52.

Shang, S., Zheng, N., Xu, J., Xu, M. and Zhang, H. (2010), 'Detecting malware variants via function-call graph similarity', *5th International Conference Malicious and Unwanted Software*, Nancy, France, pp. 113-120.

Shanmugam, G., Low, R. and Stamp, M. (2013), 'Simple substitution distance and metamorphic detection', *Journal of Computer Virology and Hacking Techniques*, Vol. 9, No. 3, pp. 159-170.

Snakebyte (2000) 'Next generation virus construction kit (NGVCK)', available at: http://vxheaven.org/vx.php?id=tn02 (accessed on 14 February 2017).

Sorokin, I. (2011), 'Comparing files using structural entropy', *Journal in Computer Virology*, Vol. 7, NO. 4, pp. 259-265.

Sridhara, S. and Stamp, M. (2012), 'Metamorphic worm that carries its own morphing engine', *Journal of Computer Virology and Hacking Techniques*, Vol. 9, No. 2, pp. 49-58.

Stamp, M. (2015), 'A revealing introduction to hidden Markov models', available at: http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf (accessed on 14 February 2017).

Symantec. (2011) 'Internet security threat report, Vol. 17', available at: http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_2011_21239364.en-us.pdf (accessed on 14 February 2017).

Symantec. (n.d.), `Virus construction kit', available at: http://computervirus.uw.hu/ch07lev1sec7.html (accessed on 14 February 2017).

Szor, P. (2000), 'The new 32-bit medusa', available at: https://www.virusbulletin.com/uploads/pdf/magazine/2000/200012.pdf (accessed on 14 February 2017).

Szor, P. (2005), 'Advanced code evolution techniques and computer virus generator kits', available at: http://www.informit.com/articles/article.aspx?p=366890&seqNum=6 (accessed on 14 February 2017)

Szor, P. and Ferrie, P. (2001), 'Hunting for metamorphic', available at: https://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf (accessed on 14 February 2017).

Tamboli, T., Austin, T. and Stamp, M. (2014), 'Metamorphic code generation from LLVM IR bytecode', *Journal of Computer Virology and Hacking Techniques*, Vol. 10, No. 3, pp. 177-187.

The Mental Driller. (2002) 'Metamorphism in practice or "How I made MetaPHOR and what I've learnt", available at: http://biblio.l0t3k.net/magazine/en/29a/ (accessed on 14 February 2017).

Tips Trik Dan Berbagi Informasi. (n.d.), 'Virus creation tools: VX heavens', available at: http://oktridarmadi.blogspot.com/2009/09/virus-creation-tools-vx-heavens.html (accessed on 14 February 2017).

Toderici, A.H. and Stamp, M. (2013), 'Chi-squared distance and metamorphic virus detection', *Journal of Computer Virology and Hacking Techniques*, Vol. 9, No. 1, pp. 1-14.

VX Heavens (n.d.), 'Access macro generator', available at: http://download.adamas.ai/dlbase/Stuff/VX%20Heavens%20Library/static/vdat/creatrs1.htm (accessed on 14 February 2017).

Wong, W. and Stamp, M. (2006), 'Hunting for metamorphic engines', *Journal in Computer Virology*, Vol. 2, No. 3, pp. 211-219.

Xu, M., Wu, L., Qi, S., Xu, J., Zhang, H., Ren, Y. and Zheng, N. (2013), 'A similarity metric method of obfuscated malware us-ing function-call graph', *Journal of Computer Virology and Hacking Techniques*, Vol. 9, No. 1, pp. 35-47.

You, I. and Yim, K. (2010), 'Malware obfuscation techniques: a brief survey', *International Conference on Broadband, Wireless Computing, Communication and Applications,* Fukuoka, Japan, pp. 297-300.

Zbitskiy, P. (2009), 'Code mutation techniques by means of formal grammars and automatons', *Journal in Computer Virology*, Vol. 5, No. 3, pp. 199-207.

About the authors

Prasad Deshpande holds an undergraduate degree from Pune University in India, and he completed his Master's in Computer Science at San Jose State University in December 2013. Prasad currently works on disaster recovery and business continuity products at Symantec Corporation in Silicon Valley. E-mail address: prasad.0210@gmail.com

Mark Stamp can neither confirm nor deny that he spent more than seven years working as a cryptologic mathematician at the super-secret National Security Agency. However, he can confirm that he spent two years at a small Silicon Valley startup, developing a security-related product. For the past dozen years, Mark has been employed as a Professor of Computer Science at San Jose State University, where he teaches courses in information security, conducts research on topics in security and machine learning, writes security-related textbooks, and supervises ridiculously large numbers of Masters student projects. Perhaps not surprisingly, most of his students projects are security-related.

Corresponding Associate Professor, Department of Computer Science, San Jose State University, One Washington Square, San Jose, CA 95192. Tel: 408-492-5094. E-mail address: mark.stamp@sjsu.edu