Non-normalizable Functions: A New Method to Generate Metamorphic Malware

Rodney Owens
SIS Dept. and CyberDNA
UNC Charlotte

Weichao Wang
SIS Dept. and CyberDNA
UNC Charlotte

***Abstract*—To successfully identify the metamorphic viruses oriented from the same base, anti-virus software has adopted the code normalization technique to transform the variations to a more uniform signature representation. Current code normalization technique focuses on the simplification of the arithmetical or logical operators. In this paper, we introduce a new technique of generating metamorphic viruses by embedding complicated manipulation functions that cannot be normalized into the malicious executables. Using encryption/decryption functions as an example, we present this evasion strategy that malware writers could employ in the future. We demonstrate the strategy's effectiveness in evading detection by current anti-virus technologies. We also discuss the potential mitigation mechanisms.**

I. Introduction

Signature based virus detection has made great progresses in the past years. Virus signatures can be created by scanning files for sequences of machine instructions that are unique to the virus [1]. For example, Symantec released 1.6 million virus signatures in 2008. Periodically, client anti-virus programs will check malware definition servers for signature updates and add the new signatures to their databases for future detection.

Attackers have developed many methods of preventing anti-virus products from detecting their malicious programs, such as polymorphic and metamorphic coding. Metamorphic malware is malware that changes sections of its own code to syntactically similar versions. This defeats antivirus scanners by preventing machine code sequences from looking the same so that a signature would not match in these altered sections.

While mechanisms [2], [3], [4] have been designed to reduce semantically similar machine instructions to enable more reliable detection of metamorphic variants of viruses, the performance of anti-virus software under more sophisticated manipulation functions remains an open problem. In this paper, we plan to investigate new methods to change the virus binaries to avoid detection. Specifically, we propose to embed public key encryption functions to change the signatures of malware. Through conducting decryption immediately after encryption on registers, we insert a sophisticated function into the virus image without changing its actual duty. We will demonstrate that by using complex encryption functions beyond simple arithmetical and logical operators, we can change the control graph of the malware as well as prevent code normalization from determining whether or not this function has no effect on the program execution.

To demonstrate the effectiveness of the mechanism, we choose a set of malicious executables that can be detected by many anti-virus products as the base files. We embed a hard-coded RSA algorithm over the AL register into the base files to change their control flows and sequences of instructions. The experiments show that many anti-virus products will be fooled by this simple technique. At the same time, the performance penalty on the malware is very limited. The study shows that new mechanisms are needed to detect such attacks.

The contributions of the paper are as follows. We explore possible ways of embedding complicated manipulation functions into malicious executables to generate new variations of malware. Our experiments on state-of-the-art anti-virus software programs show that this mechanism can deceive many of them. Therefore, the proposed techniques pose real threats to computer security. We also investigate mechanisms to defend against such attacks.

The remainder of the paper is organized as follows. In Section II, we will review related work. We will focus on the virus detection mechanisms and the anti-attack of the malicious parties. In Section III, we will present the proposed function embedding and malware generation mechanism. In Section IV, we will describe in detail the new techniques for modifying Portable Executable (PE) binary files of the viruses to avoid detection. Concrete examples of manipulation functions and experimental results on state-of-the-art anti-virus software will be presented to demonstrate the threats of the proposed mechanisms. In Section V, we will introduce our methods of preventing such attacks. Finally, in Section VI, we will discuss future extensions and conclude the paper.

II. Related Work

A. Malware Signatures

The easiest way for an anti-virus program to detect known malware is with signatures. A signature is a particular binary sequence that is unique to the malware program [1]. These signatures are usually defined by the anti-virus vendors, and are added into the local database as a definition update. As a user accesses files and programs, the anti-virus program checks the binary form of the file with its signature database. If there is a match, the file is said to contain malicious code. Virus signatures are made of patterns with thousands of bits to reduce false alarms caused by short binary strings. Methods of creating better signatures have been investigated by [5], where

the hexadecimal sequences of the binary code were chopped up into n-grams in order to use data mining and machine learning algorithms to analyze and classify the malware.

It has been proposed by [3] that metamorphic malware can be detected by examining executables using algebraic specification, by which malware machine instructions are analyzed for semantic equivalences by assigning instructions mathematical operation equivalences and comparing these equivalences to signatures. If they match, then the binary analyzed was a metamorphic variant of known malware.

B. Metamorphic Engine Analysis

In [6] and [7], the authors discuss methods malware uses to obfuscate its code. Some examples are "Garbage Insertion/Semantic Nops", "Code Reordering/Instruction Permutation", "Variable Renaming", "Instruction Substitution", and "Control Flow Alteration". With Garbage Insertion/Semantic Nops, processor instructions that have no net effect on the outcome of the program are added. Code Reordering/Instruction Permutation takes advantage of the fact that many instructions can be executed out of order and the net outcome is still the same. Variable Renaming is when an intermediate value changes which variable it is assigned to. This can most easily be accomplished by switching the registers that temporary variables are stored in. Instruction Substitution is when a sequence of instructions are substituted for another sequence of instructions that have the same effect. Finally, in Control Flow Alteration, conditional jumps are added that jump to sequences of instructions that permanently mutate variables and perform undesired operations. The conditions for this jump, however, are set to never be true, so these segments of code are never executed.

In [8], the authors use varying amounts of code reordering and semantic nops to obfuscate known malware to avoid detection. The authors proposed normalization schemes to detect such changes. In [6], the authors evaluate their normalization technique by defining the Euclidean distance between normalized code fragments and the malicious code archetype.

III. Constructing Non-Normalizable Metamorphic Virus

An effective approach to detecting metamorphic viruses is the code normalization technique [2], [9]. In this technique, different schemes are adopted to cancel out the mutations introduced by the malware. To embed robustness against the instruction level mutation into the detection tool, high level representations of the code and the control flow graph are used. Although the researchers have noticed that some of the code transformations are non-reversible, they focus on the schemes such as instruction permutation and substitution. The following research shows that some of the arithmetical or logical operations can be too complicated for the code normalization technique and can be used to construct metamorphic viruses that are beyond the detection capabilities of the state-of-the-art tools. Below we first introduce the basic idea of code normalization. We will then introduce the families of functions that can be used to manipulate the instructions and control flow graph, respectively.

A. Revisit of Code Normalization

Code normalization is the process of optimizing code to a single efficient normal form. The main idea of code normalization is to simplify different versions of an executable to a single form (or as few as possible), and then create a signature on this normal form. In the future, executables can be normalized and then compared to these signatures of normalized malware. If there is a match, then the executable performs the exact same actions as the previously identified malware. In this way, metamorphic malware can make thousands of variants, but as long as those variants can be normalized to the same form, this normal form is the only signature that the anti-virus program needs to have.

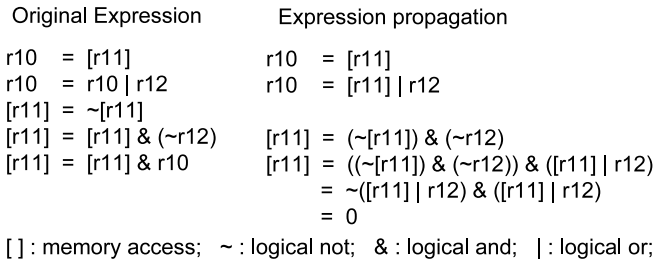

Fig. 1. Code normalization through expression propagation.

An example of a code normalizer was given in [2]. In [2], the authors described code normalization as a five step process. The first step, "Instructions meta-representation", is the raising of the processor instructions to a high-level representation. The second step is "Propagation", where the results of intermediate calculations are forwarded and compiled into their intended operations for further analysis. The third step is "Dead Code Elimination", through which any values that are computed, but then never used, are thrown away. The fourth step is "Algebraic Simplification", where algebra is used to simplify the high-level expressions from the previous steps. This removes statements like "add 1 to eax; then subtract 1 from eax". The last step is "Control Flow Graph Compression" where the program's possible flows are graphed. Then, based on previously computed values, branches of the graph that can never be executed are removed. After all of the processes have finished, the resulting graph is believed to be the reduced normal form of the program.

B. Code Manipulation Functions

Currently, the code normalization tools integrate the ordinary algebraic rules and intermediate result propagation to conduct simplification. The first technique can easily identify the operations such as adding a constant to the register and then subtracting the same value. In the second technique, as illustrated in Figure 1, the intermediate results are carried over to the later instructions for further simplification. Both techniques can identify and cancel out only the complementing operator pairs with straightforward impacts such as

add/subtract, times/divide, xor, and the logical operators. However, in the broad field of code manipulation, there are families of functions that will conduct a series of complex operations to the register or memory locations without changing their values. The net effects of these functions cannot be automatically identified by a computer program. Below we introduce two families of such functions.

The first family of functions that can be used to manipulate register and memory values are encryption and decryption algorithms. Here the toy schemes such as circular shift or bit permutation should be excluded. We consider only those product ciphers such as RSA, Elliptic Curve Cryptography, and AES. For symmetric encryption algorithms, we can use any register or memory values as the key and the plain text. For asymmetric encryption algorithms, we can pre-select the public/private key pair and embed them into the implementation. We can conduct encryption immediately followed by decryption to the selected values. In this way, the actual values at the selected locations will not change. At the same time, we inject a complex function between those originally consecutive instructions to avoid the detection of malware signatures.

One concern for using the product cipher algorithms as value manipulation functions is that many of these algorithms have standard implementations. Therefore, the anti-virus software can generate signatures of these algorithms to identify them in metamorphic viruses. Two schemes can be used to solve this problem. First, we know that for many encryption algorithms, once the secret key is selected, we can optimize the encryption procedure to improve its efficiency. Since the customization is built upon the special values of keys, the final results can be very different from the standard implementation. Second, there are many user-designed encryption algorithms that are never accepted as a standard because of their vulnerabilities. These algorithms can also be used as manipulation functions. Note that what we need here is a group of complex operations that have no net effects on the value. The safety of the algorithm is not a factor.

The second group of functions that can be used to manipulate register and memory values are built upon some special properties of the encryption functions. For example, some encryption algorithms have the commutative property. Under this property, when a value is encrypted for multiple times with different keys, the decryption operations with corresponding secrets can be conducted in any order and the final result will be the same. In this way, we can combine function permutation with encryption to further increase the difficulty of normalization. As another example, the self-healing property of the ECB mode of DES encryption can be used to inject some randomness into the manipulation procedure without changing the final results.

C. Control Flow Graph Manipulation Functions

The inserted value manipulation functions can be converged into segments of code with a straight-line structure. As the analysis in [2] shows, changing only the order of consecutive instructions is not enough for defeating the anti-virus software since under many conditions the control flow graph (CFG) of the metamorphic virus is constructed and compared to known malware. Therefore, some fake conditional jumps should be used to obfuscate the CFG. Since the state-of-the-art anti-virus software can recognize many kinds of fake jumps, we propose to use the code manipulation functions to impact the CFG. For example, we can conduct decryption immediately after the encryption operation and insert a fake conditional jump based on the comparison result between the two values.

We can also use some special properties of the encryption algorithms to control the program execution flow. Two examples are as follows. The unpadded RSA encryption has the homomorphic property, which means $E(m_1) \times E(m_2) = E(m_1 \times m_2)$. Therefore, the malware can conduct three encryption operations of m_1, m_2 and $m_1 \times m_2$ respectively. Then a fake conditional jump can be inserted based on the comparison of the encryption result. As another example, DES encryption exhibits the complemental property, which means that $E_k(P) = C \Leftrightarrow E_{\bar{k}}(\overline{P}) = \overline{C}$, where $\overline{x}$ is the bitwise complement of x. With this property, we can generate two cipher texts that are bitwise complements to each other. Then a fake conditional jump can be inserted into the malware based on comparison of the results.

D. Analysis of Normalizability

The normalizability analysis is essential for the proposed metamorphic virus construction mechanism since it will determine whether or not the designed code manipulation functions can be simplified or identified by the state-of-the-art normalization algorithms. Such analysis will be conducted at both the instruction level and the abstract function level. At the instruction level, we consider the techniques proposed in [2]. For example, through expression propagation, the decryption procedure of RSA can be represented as $(m^{r_1} \ mod \ n)^{r_2} \ mod \ n$, where r_1 and r_2 are the public and private keys respectively. Different from those simple operators such as add/subtract, the values of r_1, r_2, and n must satisfy some special requirements for simplification. Since there are an unlimited number of prime integers, the anti-virus software cannot pre-generate all possible values of r_1, r_2, and n. Therefore, it will be very difficult for anti-virus software to conduct normalization through expression propagation.

Making sure that the embedded code manipulation functions are non-normalizable at the instruction level alone will not prevent the anti-virus software from detecting the metamorphic malware. For example, although the anti-virus software cannot figure out the secret keys, it may be able to figure out that a function in the virus is actually conducting RSA encryption/decryption. Unfortunately, the latest achievements in automatic programming [10], [11] and automatic program verification [12], [13] focus on the relationship between high level specifications and their implementations. For example, in [12], the C2BP tool can generate the predicate abstraction for only the boolean operations in a C program. In [13], the logical formulas are generated to determine the soundness of the code with respect to the given specification. The Concurrent

Automatic Programming System [11] focuses on the formal verification of the high level specification. Given two random code segments $CS1$ and $CS2$, none of these approaches can determine whether or not the two segments will actually cancel out each other's impacts. Based on these observations, we conclude that it will be very difficult to normalize the code manipulation functions to detect the metamorphic viruses.

IV. Concrete Examples and Experiment Results

To demonstrate the practicability of the proposed approach, in this section we plan to design a metamorphic virus generator by embedding asymmetric encryption and decryption algorithms into malware executables. Specifically, we choose malware in Windows' Portable Executable (PE) format and RSA as the code manipulation functions. Below we describe the details of the implementation and the experiment results.

A. Implementation

Expanding executable files by embedding new instructions into them is more complicated than it appears. For example, all target address of the jump instructions should be carefully examined so that they will not jump into the middle of the injected code segments. Our metamorphic viruses are usually constructed through the following two steps.

Step 1: Expanding PE Executables

Windows' PE binary executables are composed of several sections. These include sections that contain the raw processor instructions (code section), DLL import tables, and multiple data sections to store initialized variables (such as some memory locations and strings output to the user). The code section is almost always one of the first sections in the executable file. The code section cannot simply be made larger since the data sections will be pushed to higher memory addresses thus making all references to the data sections invalid (they will point to the code section). We cannot add a new code section to the end of the executable file or increase the last section's size and mark it as executable because anti-virus vendors are well aware of these schemes, so heuristic scanners will flag the file as containing injected code.

From an attacker's point of view, we propose to expand the existing code section to allow room for more processor instructions, and change all the pointers within an executable to reflect the new data section's memory addresses. During this procedure, the PE header information needs to be modified to reflect this move. We use a PE file editor such as PEExplorer [14] to edit the PE header information. We then use a debugger such as OllyDbg [15] to extract the opcodes and operands from the program. A script is then written to search these operands for references to the memory addresses that have been moved. The data section itself sometimes contains references to other locations in the data section, so these values should also be searched and incremented by the script as needed. Once this is complete, the code section can be made large enough to add an arbitrary amount of code, while maintaining the integrity of the existing executable.

Step 2: Injecting RSA based code manipulation functions

As we described earlier, new instructions cannot be inserted into the code section without any restrictions because of the destinations of the jump instructions. If the code has been evenly expanded throughout the code section, the jump instructions will not point to the correct positions and the program will not operate as desired. To solve this problem, we have experimented with two mechanisms. In the first mechanism, we insert the RSA encryption/decryption functions at the end of the code section. At each place that we want to activate the function, we will substitute the current instruction with an unconditional jump. An unconditional jump is five bytes long and is composed of a one-byte opcode and a four-byte operand. To prevent the memory locations of other instructions from being impacted, we usually choose instructions that occupy five or more bytes as the substitution targets. Each of these instructions can be replaced with an unconditional jump to the rear of the enlarged code section, and the original instruction moved to the jump destination. The next instructions at the jump destination after the original instruction is executed can be a public key encryption function of our choice, followed by its corresponding decryption algorithm. After the decryption, an unconditional jump back to where we originally jumped out is written. An example is shown in Figure 2, where the original instruction has been moved to memory location 0x4085DC.

```
Address          Command                    Address          Command
0x4024BE:  push offset str2          0x4024BE:  push offset str2
0x4024C3:  mov eax, dword_40BE10     0x4024C3:  jmp 0x4D85DC
0x4024C8:  max ecx, [edp+arg_4]      0x4024C8:  max ecx, [edp+arg_4]
                                                  -
                                                  -
                                     0x408463:  RSA func
                                                  -
                                                  -
                                     0x4085DC:  mov eax, dword_40BE10
                                     0x4085E1 :  call 0x408463
                                     0x4085E5 :  jmp 0x4024C8
```

Fig. 2. A section of assembly codes showing before (left) and after (right) the encryption/decryption functions are added.

The second code injection mechanism is more flexible but having tighter restrictions on the original executable file and heavier workload for modification. Here we will add the code manipulation functions directly into the code section, shift the instructions that are executed afterwards, and then update all the references to these instructions. For this mechanism to work properly, all jump destinations in the original executable file must be identified and updated through static scan. If the file contains a reference that can only be determined dynamically during execution, depending on the nature of the calculation, this mechanism may not safely be applied.

B. Experimental Results

To demonstrate the practicability of the proposed approach to generating metamorphic viruses, we have chosen a group of well studied malware and state-of-the-art anti-virus programs to conduct experiments. We use RSA encryption/decryption algorithms as the code manipulation functions. We choose a special key pair so that the RSA computation can be conducted upon the AL register. The evaluation focuses on the reduction in performance of the malware after the functions are injected

and the detection rate of the generated metamorphic viruses. Below we presented the details of the experiments and the evaluation results.

Selected Malware

We chose a variety of malware in PE format as our test cases to demonstrate that the proposed approach provides a generic mechanism to construct metamorphic viruses. These malicious programs are selected from different packages and their functionality spans in a wide range. Table I illustrates the details of these malware programs.

malware	package	functionality
iam-alt.exe	pass the hash	find out the password hashes of currently logged in users
whosthere-alt.exe		
genhash.exe		
abel.exe	Cain and Abel	steal logon hashes from a Windows computer and crack the hashes
john-mmx.exe	John the Ripper	remote password cracker

TABLE I
SELECTED MALWARE FOR EVALUATION.

Performance Results

Inserting new instructions, especially complicated functions such as RSA encryption/decryption, into the malware executable files could degrade the performance of the malware. Depending on the injection points, the code manipulation functions will be executed at different frequencies and reduction in performance could be different. We have embedded the RSA functions at different places in the malware. On all our test cases except for John the Ripper, the RSA function was executed on the order of thousands of times with no noticeable difference in delay. For John the Ripper password cracker, we embedded the RSA functions into many inter loops of the cracker so they are called very frequently. In our experiments, the RSA functions were run approximately 141,824,000 times with a possible measurement error up to 0.0014%. We test both the multi-salt mode and the single-salt mode of different password hash algorithms. Table II shows the results of the two runs when performed back to back. The computation efficiency is measured by the number of [user name, password] combinations that the cracker can test in each unit of time.

hash algorithm	Computation efficiency ([usr, passwd]/sec)		performance degrade (clm 2 / clm 3)
	before injection	after injection	
DES, multi-salt	1022K	2897	352.8
DES, single-salt	955K	2842	336
BSDI DES multi-salt	33267	95	350.2
BSDI DES single-salt	32941	95	346.7
Blowfish	404	2.1	192.4
AFS DES short	319259	1810	176.4
AFS DES long	827228	5677	145.7
NT DES	9732K	56096	173.5

TABLE II
JOHN THE RIPPER PERFORMANCE BEFORE AND AFTER RSA FUNCTION INJECTION.

From the experiment results, we find that when the code manipulation function is inserted into the inner loop of the password cracker, the performance may degrade several hundred times. Although this performance penalty may be intolerable for many user applications, its impacts on many viruses are not as severe. Since many malicious programs use idle CPU cycles to accomplish their computation, attackers usually have all the time they need to compromise the system. At the same time, RSA is very computationally expensive. We can use other code manipulation functions to alleviate the performance penalty.

Detection Results

The ultimate goal of the proposed research is to generate metamorphic malware to deceive the anti-virus programs. To evaluate our approach from this aspect, we use state-of-the-art anti-virus programs to scan the newly generated variants. The adopted anti-virus programs are shown in Table III.

anti-virus program	Product version	Virus database version
Avast! Home Edition	4.8.1351	090906-1
AhnLab	8.0.1.6	
Avira	9.0.0.407	9.0.0.407
Symantec	10.1.5.5000	
Dr. Web	5.0.1.06018	
Mcafee VirusScan	13.15.101	5733.0000
Ikarus	1.0.97	73486
Sophos	7.3.0	4.28E
TrendMicro	17.50.1366.0000	6.423.50

TABLE III
ADOPTED ANTI-VIRUS PROGRAMS.

We conduct the experiments as follows. For each malware, we will first determine whether or not it can be detected by an anti-virus program. Only when it can be detected, will we generate metamorphic variants through function injection and resubmit them for scan. The experiment results are shown in Table IV. Please note that we illustrate only the anti-virus programs that can detect the original malware.

During our experiments, we find that sometimes we need to combine function injection with other techniques to deceive the anti-virus programs. For example, Symantec uses the attacker's name embedded in "pass the hash" toolkit to detect the malware. Therefore, we combine "unique string replacement" with function injection to generate metamorphic variants to deceive it. Some anti-virus programs use the DLL call tree to detect malware. For these programs, we encapsulate the DLL access with another layer of function calls. The required additional techniques to deceive the anti-virus programs are illustrated in parenthesis "()" in Table IV.

From the results in Table IV, we find that the function injection technique can generate metamorphic variants to deceive all tested anti-virus programs even when they can detect the original malware. Our proposed approach provides a new technique that could be adopted by attackers in the future to generate metamorphic viruses. New mechanisms must be designed to defend against such attacks.

V. DETECTING METAMORPHIC VARIANTS THROUGH SEMANTIC NOP FUNCTION RECOGNITION

Enabling an anti-virus program to detect complicated functions such as encryption and decryption algorithms is harder

than it appears. The anti-virus program cannot generate a signature for the encryption algorithm since it can change based on the secret keys or implementation. Further, new algorithms may appear as cryptographic technology evolves.

malware: IAM alt, Whosthere and Genhash		
anti-virus program	can detect the original malware	deceived by metamorphic virus generated through func injection
Avast!	yes	yes
AhnLab	yes	yes
Avira	yes	yes (with DLL relay)
Symantec	yes	yes (with unique string replacement)
McAfee VirusScan	yes	yes (with DLL relay)
Ikarus	yes	yes (with unique string replacement)
TrendMicro	yes	yes (with unique string replacement)

malware: Abel		
anti-virus program	can detect the original malware	deceived by metamorphic virus generated through func injection
Symantec	yes	yes (with unique string replacement)
McAfee VirusScan	yes	yes (with DLL relay)
Ikarus	yes	yes (with unique string replacement)
Sophos	yes	yes (with DLL relay)
TrendMicro	yes	yes (with unique string replacement)

malware: John the Ripper MMX		
anti-virus program	can detect the original malware	deceived by metamorphic virus generated through func injection
Avira	yes	yes (with unique string replacement)
Symantec	yes	yes (with unique string replacement)
McAfee VirusScan	yes	yes (with DLL relay)
Ikarus	yes	yes (with unique string replacement)
Sophos	yes	yes
TrendMicro	yes	yes (with unique string replacement)

TABLE IV
DETECTION RESULTS OF THE METAMORPHIC VARIANTS GENERATED THROUGH FUNCTION INJECTION.

In order to detect the metamorphic viruses generated through function injection, we propose to develop a more intelligent scanner to identify the semantic nop functions. The detection procedure will be conducted in two steps. In the first step, we will locate the suspects of the nop functions. This can be achieved at both the instruction level and function call level. At the instruction level, we will locate the code segments that contain instructions deviating from the purpose of the software. For example, many asymmetric encryption algorithms are built upon some hard mathematical problems and their computation procedures are different from most user applications. We can identify these code segments through the density of such instructions. At the function call level, we will tally the number of times that each function is called. Then, we will setup a threshold and pick the most frequently used functions as the candidates. Once the suspect functions are located, we can conduct the second step. Our scanner will select a group of random numbers as the input to these code segments and monitor if any of the values in the storage locations change. For those semantic nop functions, all of the values must be preserved to prevent any changes to the program control flow. When we experiment with the suspect function with a large enough group of random numbers and none of their values change, we can conclude with high confidence that it is a semantic nop function and can be removed for subsequent virus analysis techniques.

VI. Conclusion and Future Work

Anti-virus products have come a long way with the detection of malware, but malware writers can continue to devise new strategies to overcome these more powerful anti-virus scanners. In this paper, we brought to life how malware can be made more sophisticated to evade detection and still perform the same functions as before. As the malware/anti-virus war continues to play out, anti-virus software needs to be armed with several different detection mechanisms in order to successfully detect evasive malware.

The register/memory manipulation functions may become so complicated that running test values through them would be impractical. For example, a complicated register/memory manipulation function that employs multiple nested calls with if/then tests would take too much time and have too much uncertainty within them for test values to uncover them in a reasonable amount of time. We plan to explore advanced semantic detection algorithms to include encryption/decryption and other sophisticated register/memory manipulation functions to improve signature based malware detection.

References

[1] J. Kephart and W. Arnold, "Automatic extraction of computer virus signatures," in *Inter. Conf. Virus Bulletin*, 1994, pp. 178–184.

[2] D. Bruschi, L. Martignoni, and M. Monga, "Code normalization for self-mutating malware," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 46–54, 2007.

[3] M. Webster and G. Malcolm, "Detection of metamorphic computer viruses using algebraic specification," *Journal in Computer Virology*, vol. 2, no. 3, pp. 149–161, 2006.

[4] ——, "Detection of metamorphic and virtualization-based malware using algebraic specification," *Journal in Computer Virology*, vol. 5, no. 3, pp. 221–245, 2009.

[5] J. Kolter and A. Maloof, "Learning to detect and classify malicious executables in the wild," *J. Mach. Learn. Res.*, pp. 2721–2744, 2006.

[6] D. Bruschi, L. Martignoni, and M. Monga, "Using code normalization for fighting self-mutating malware," in *Proc. Inter. Sympo. Secure Software Engineering*, 2006.

[7] M. Christodorescu and S. Jha, "Testing malware detectors," in *Proc. of ACM Inter. Sympo. Software Testing and Analysis*, 2004, pp. 34–44.

[8] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith, "Malware normalization," University of Wisconsin, Madison, Tech. Rep. 1539, 2005.

[9] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia, "Normalizing metamorphic malware using term rewriting," in *IEEE International Workshop on Source Code Analysis and Manipulation*, 2006, pp. 75–84.

[10] F. Miao, S. Akihiro, T. Tomohiro, K. Masahiro, U. Kazuhiro, and K. Seiichi, "An automatic programming system by composition of reusable program components," *SIG-KBS*, vol. 76, pp. 19–24, 2007.

[11] E. Kennedy, "A concurrent automatic programming system," in *Proc. Annual Southeast Regional Conference*, 2008, pp. 94–98.

[12] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani, "Automatic predicate abstraction of c programs," *SIGPLAN Not.*, vol. 36, no. 5, pp. 203–213, 2001.

[13] J.-C. Filliatre and C. Marche, "The why/krakatoa/caduceus platform for deductive program verification," *Computer Aided Verification*, pp. 173–177, 2007.

[14] "Pe explorer," http://www.heaventools.com, 2011.

[15] "Ollydbg," http://www.ollydbg.de/, 2011.