A study of polymorphic virus detection

Vinh T. Nguyen*
PhD student, Computer Science

Abstract

Traditional viruses were computer programs with static structure exhibiting very limited functionality. Once identified for the first time, their structure is utilized by antivirus (AV) software as a tool for detecting the similar viruses with similar patterns. However, modern viruses are smart enough to self-configure and even change the pattern of their functionality making it hard for AV software detecting them. A polymorphic virus is a complicated computer virus that affects data types and functions making it difficult to inspect its internal structure. In this paper, we conduct a study of the polymorphic virus to answer three research questions: (1) What are the general techniques employed by these viruses to exhibit polymorphism? (2) What is the state-of-the-art of detecting polymorphic viruses? And (3) What should be made to help antivirus software detect these viruses? The result of this study may provide a good source of knowledge for polymorphic researchers and anti-virus software company getting the overall picture of this virus and thus provides a suitable solution to the problem.

Keywords: polymorphic virus, malware, anti-virus

1 Introduction

With the advent of science and technology, a computer has been one of the most advanced devices over centuries that helps human perform sophisticated work and save data. It is being used in our daily life activities using desktop computers, laptops, tablets, smart phones and hand-held devices. In the early day, computers are mostly used to speed up calculations by a set of instructions with limited storage capacity. Later on, this architecture was expanded to store data inside storage devices such as floppy disk, optical disk, hard disk, memory stick and so on. In competitive markets, this data may contain highly sensitive information and becomes on of the favorite targets for many attackers and tons of malicious programs were written to favor this data. These malicious codes are known by many different names such as a virus, malware, botnet, trojan, etc. for different purposes (i.e., for fun, for evil, or even for good). They are often operated by inserting or attaching themselves to another host program.

One typical harmless virus was known as Elk Cloner [Spafford et al. 1989] virus written by Richard Skrenta, a 15-year-old high school student, around 1982 which displayed a little poem on the screen. It did not damage any resources on computer but annoying people with the message as shown in Figure 1. This virus was able to spread to infect another operating system running Apache II.

Inspired by understanding the biological evolution and self-production, John von Neumann [Von Neumann and Burks 1996] created the first self-replicating computer programs to be known in the history. This program can be considered the foundation of many modern virus.

Virus can penetrate into host computers in many different ways, for example by email, text message attachments, social links, free apps, fun images, audio, video files. Once it was triggered, it stayed dormant and infect other computers in the networks. To avoid being detected, the virus author used various techniques to stealth the

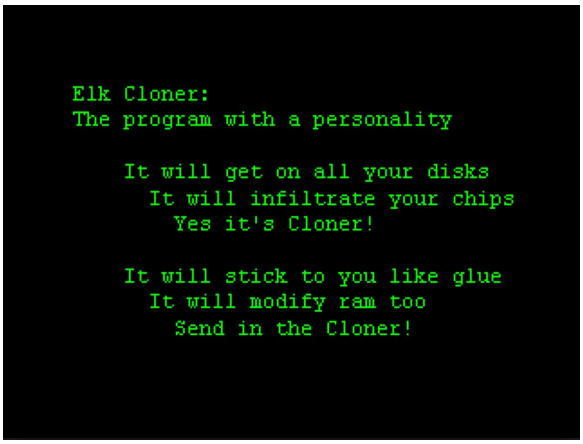

Figure 1: *A little poem message annoying users*

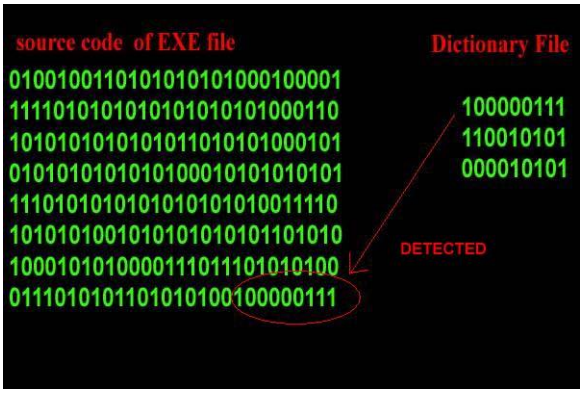

Figure 2: *Anti-virus signatures based detection*

code. Traditional method avoided detection by not modifying the "last modified" date of the host file when it was infected. Other virus, for example Chernobyl Virus [Christodorescu and Jha 2006] utilized the unused areas of executable files by overwriting them with malicious code, this allows keeping the same size of infected files. Another more advanced technique, Conficker [Porras et al. 2009] terminated the tasks associated with the anti-virus software before it was detected.

As Operating Systems keep updating that do not allow to modify the files or kill process without proper authorization, the virus authors had to use another technique to hide their programming codes.

The first technique was known as **self-modification** [Anckaert et al. 2006], this technique was developed to counter the anti-virus software (AV) that scans the virus by signature as depicted in Figure 2. Basically, the AV will maintain a database that contains a list of signatures for every detected virus. When it scan a file, it compares the file's signature with its signature database, once a string is matched this file is considered to be infected then this file can be deleted, locked or cleaned (remove the signature). To avoid detection, the virus modified itself with a new signature on every infected file which can be shown as follow:

*e-mail:vinh.nguyen@ttu.edu

```
repeat N times {
    increase A by one
    do something with A
    when STATE has to switch {
        replace the opcode "increase" above
        with the opcode to decrease,
        or vice versa
    }
}
```

By doing this way, virus authors can create an infinite number of signatures.

Another method to avoid signature detection is through **encryption**, this technique uses simple encryption method to cipher the body of malicious code. Each encryption key will produce an encrypted text, so the virus can replicate itself to many different files by only modifying the encryption key. Each infected file will contain an encrypted malicious code, decryption module and encryption key. The unique encrypted malicious code will result in a different signature, thus make it difficult for AV to detect. The main drawback of this technique is the decryption module which remains constant through all infected files, opening a possible way for AV software to detect.

In order to overcome the limitation of encryption method with a constant decryption module, a new technique was developed to make the decryption module from *static* to *dynamic*, that is, this module will be modified in each infection. This method is called **polymorphic code** [Torrubia-Saez 2003]. This polymorphic virus has become one of the most challenging task for AV software to detect since it is a self-encrypted virus and is able to duplicating itself by creating slightly modified versions of itself.

A more advanced technique is **metamorphic code** [Borello and Mé 2008] in which the virus completely rewrite itself on every execution. However, this method is extremely expensive because it requires a *metamorphic engine*, making it impractical in practice.

Hence, in this study, we focus on understanding the polymorphic virus by addressing the following research questions:

- **Q1**: What are the general techniques employed by these viruses in order to exhibit polymorphism?
- **Q2**: What is the state-of-the-art of detecting polymorphic viruses?
- **Q3**: What should be made in order to help antivirus software detect these viruses?

The rest of the paper is organized as follows: section 2 reviews the state-of-the-art of detecting polymorphic viruses. section 3 presents the general techniques employed by these viruses in order to exhibit polymorphism. section 4 shows the potential approach to help antivirus software detect these viruses. And section 5 concludes our paper with recommendations.

2 Literature Review

Typically, to understand the pattern and behavior of a malicious program, two general approaches are used in analysis: (1) static analysis, and (2) dynamic analysis. Static analysis involves analyzing binary signatures of the malware without executing it; whereas, dynamic analysis observes the behavior of the running malicious code in a controlled environment.

2.1 Static Analysis

Signature based approach: Signature detection [Griffin et al. 2009] is the simplest method and is the most widely used for traditional malware detection. This method constructs a database that contains signatures of all known malware. When analyzing a new programming code, it compares the signature of the analyzed virus with its database, if the matching is found, the analyzed file is considered as virus. This approach is fast and has high positive rate, however the database needs to be updated with new signature. Although this technique is old but it was used in the early days of polymorphic detection when investigator/researcher analyzed the virus manually, one by one, line by line to detect various sequences of programming codes [Bondarenko and Shterlayev 2006]. As the number of virus has been increasing so fast, this technique quickly becomes time-consuming, expensive and impractical.

System call analysis: Sung et al. [Sung et al. 2004] proposed the Static analyzer for vicious executable (SAVE) to detect malware, mostly focus on polymorphic and metamorphic virus which run on Windows Operating System. This method works based on the assumption that all malware variants share a common core signature - a combination of several features of the programming code. In their method, two critical steps were involved: First, the Portable Executable (PE) decompressed and passed through a parser, this parser produced a list of Windows API calling sequence. Second, this API sequence will be compared against the signature database, a similarity measure was used to conclude the analyzed file. If the similarity is greater than a certain threshold, the detection is triggered.

Control-flow graph:: Graphs are also used in static analysis [Christodorescu and Jha 2006] and [Bonfante et al. 2007] where a set of control flow graphs (CFG) were constructed and reduced (where possible) and be used as a signature database. This method works based on the assumption that the control flow graph of the malware was not modified in most of the mutation engines. Detection is carried out by comparing the sub-GFGs of the malicious file against the signature database to find if any sub-CFG is matched with the database. However, this method does not work when analyzing the metamorphic virus (example Zmist [Szor and Ferrie 2001] because this virus can change the code itself for each execution or changes to the branching structures of that flow graph.

Model checking: This method assumes that systems have finite state or may be reduced to finite state by abstraction. Serge Chaumette et al [Chaumette et al. 2011] used context-free grammars as viral signatures and a process was designed to extract the simple virus signature. This method was based on two assumptions: First, most mutating engines generate code belonging to a language that is low complexity, that is, belonging to either natural language or context-free language. Second, the mutation engine has to be embedded inside the self-replicating malware, hence it is feasible to extract the grammar of the mutation engine via a static analysis.However, this method is very time-consuming. Another study was presented by Gerald R. Thompson and Lori A. Flynn [Thompson and Flynn 2007], they compared the program hierarchical structure and mapped this structure to a context-free grammar, normalizes the grammar, and finally, they used a fast check for homomorphism between the normalized grammars. This technique is resilient despite polymorphism that reorders instructions ,rewrites instructions, inserts instructions, or removes instructions. This approach did not address encrypted files but can be applied after the file is decrypted if the unencrypted virus is suspected to be polymorphic.

Data-flow analysis: This method gathers information about the possible set of values of objects and variables involved in the spec-

imen. Agrawal, Hira, et al. [Agrawal et al. 2012] proposed a Malware Abstraction Analysis (MAA) method. They used two stages to derive semantic signature of a binary instance: First, all functions was analyzes and abstracting away all unnecessary control flow artifacts from their flow graphs. Second, all local, function level signatures were combined into a single, global signature while abstracting away all call and return specific artifacts. This method is resistant to such large scale, global transformations.

Machine learning analysis: In recent years, machine learning has gained its popularity in many fields including security. Robert Moskovitch et. al. [Moskovitch et al. 2008] proposed a technique that monitors a small set of features that are sufficient for detecting malware without sacrifice accuracy. The result of the study showed that, only using 20 features, the mean detection accuracy was greater than 90 percent, and for specific unknown worms, this accuracy get over 99 percent, while maintaining a low level of false positive rate. The advantage of machine learning techniques is that it will not only detect a known malware but also act as a database for detecting new malware. Similar studies can also be found in another model such as Naive Bayes [Alazab et al. 2011], Decision Tree, Neural Network [Moskovitch et al. 2008]. Although this technique is practical but it may not replace the standard detection methods, rather than act as an add-on feature because machine learning techniques are computational and may not be suitable for end users.

2.2 Dynamic Analysis

Trevor YannOleg Petrovsky [Yann and Petrovsky 2006] proposed an architecture to detect polymorphic virus, this architecture includes three components: (1) an emulator that emulates a selected number of instructions of the computer program, (2) an operational code analyzer that analyzes a plurality of registers/flags accessed during emulated execution of the instructions and (3) an heuristic analyzer that determines a probability that the computer program contains viral code based on an heuristic analysis of register/flag state information supplied by the operational code analyzer.

Polychronakis et al. [Polychronakis et al. 2006] presented a heuristic detection method that scans network traffic streams for the presence of polymorphic shellcode. This algorithm relied on a fully-blown IA-32 CPU emulator that makes the detector immune to runtime evasion techniques such as self-modifying code. Each incoming request was executed in a virtual environment. Their algorithm focused on identifying the decryption process that takes place during the initial execution steps of a polymorphic shellcode. The study result showed that the proposed approach is more robust to obfuscation techniques like self-modifications. One limitation of this approach was that it detected only polymorphic shellcodes that decrypt their body before executing their actual payload, it did not capture the shellcode that did not perform any self-modifications.

Antony et al. [Rogers et al. 2012] proposed an apparatus to detect malicious code that uses calls to an operating system to damage computer systems. This method will be creating an artificial memory region, this region may span one or more components of the operating system. The malicious file will be executed and the method try to detect whether the executable code attempts to access the artificial memory region. The method may comprise determining an operating system call that the emulated code attempted to access, and monitoring the operating system call to determine whether the code is viral.

Another apparatus was presented by Igor et al. [Muttik and Long 2005] where they patched additional program instructions into an emulator for detecting suspect code. During operation,a first emulator extension was loaded into the emulator then the suspect code was loaded into an emulator buffer within a data space of a computer system. The suspect code was executed in the first emulator extension. During this emulation, the system identifies whether the suspect code is likely to exhibit malicious behavior.

In another work, Ignor [Muttik 2004] presented an apparatus for detecting malicious software by analyzing patterns of system calls generated during emulation. The malicious file was executed in an isolated environment, the system calls pattern will be recorded and compared against database containing suspect patterns of system calls. Based upon the comparison result, the system identifies whether the software is likely to exhibit malicious behavior.

Stepan [Stepan 2005] proposed a method to detect malware by disassembling the malicious code dynamically then compiling this code to target the CPU host, the execution file will be executed safely on the host CPU. The code obtained can be used to compared with the original cost. This method increases the analysis speed significantly.

3 The polymorphic virus

The first polymorphic virus was written by Mark Washburn in 1990 [Szor 2005], it was known as **1260** or **V2PX** virus because of its length (1260 bytes). Inspired by Ralph Burger's publication and derived from the original Vienna virus, Mark wished to show the antiviral community why identification string scanners did not work in all cases. The length of the infected files will be increased by 1,260 bytes and be encrypted. The encryption key changes with each infection. The V2PX was not resident inside the memory, it infects *.COM files in the current or PATH directories upon execution Two sliding keys were used to decrypt the virus body, but more importantly, junk instructions were inserted into the decryptor. These instructions were useless in the code. They worked as a camourflag for the code. Depending on the number of inserted junk code, the decryptor can be shorter or longer. Furthermore, each group of instructions within the decryptor can be permutated in any order, thus decryptor's structure can change. Figure 3 shows an example of decryptor. It can be seen from Figure 3 that, in each group of instructions, a set of junk instructions are inserted (INC SI, CLC, NOP, and other do-nothing instructions)

The next milestone development of polymorphic virus was the advent of Mutation Engine (MtE) [Bontchev 1992], this engine was

```
inc     si      ; optional, variable junk
mov     ax,0E9B ; set key 1
clc     ; optional, variable junk
mov     di,012A ; offset of Start
nop     ; optional, variable junk
mov     cx,0571 ; this many bytes - key 2

; Group 2 - Decryption Instructions
Decrypt:
xor     [di],cx ; decrypt first word with key 2
sub     bx,dx   ; optional, variable junk
xor     bx,cx   ; optional, variable junk
sub     bx,ax   ; optional, variable junk
sub     bx,cx   ; optional, variable junk
nop     ; non-optional junk
xor     dx,cx   ; optional, variable junk
xor     [di],ax ; decrypt first word with key 1
; Group 3 - Decryption Instructions
inc     di      ; next byte
nop     ; non-optional junk
clc     ; optional, variable junk
inc     ax      ; slide key 1
; loop
loop    Decrypt ; until all bytes are decrypted - slide key 2
; random padding up to 39 bytes

Start:
```

Figure 3: *An Example Decryptor of 1260*

written by the Bulgarian Dark Avenger.The idea of the mutation engine was based on modular development.The concept of MtE was to make a function call to the MtE function and passed control parameters in predefined registers. The MtE will build a polymorphic shell around the simple virus inside it. When a virus uses the engine to write itself to a file, the MtE encryptor modifies the virus code so it will look like random garbage. The decryptor will ungarble this code once it is executed. The decryptor is the one part of the virus that remains unencrypted. When an infected file is run, the decryptor first gains control of the system, then decrypts both the virus body and the MtE. Then, it will transfer control of the system to the virus, which in turn will locate a new file to infect. The parameters to the MtE engine include the following:

- A work segment
- A pointer to the code to encrypt
- Length of the virus body
- Base of the decryptor
- Entry-point address of the host
- Target location of encrypted code
- Size of decryptor (tiny, small, medium, or large)
- Bit field of registers not to use

The Decryptor Generated by MtE as shown in Figure 4 will return a decryptor with an encrypted virus body in the supplied buffer. From this point, the MtE and the virus itself are copied in random access memory (RAM). The mutation engine was invoked then it randomly generated a new decrytor capable of decrypting the virus. Next, the MtE and the virus are encrypted. Finally, the virus appended this new decryptor, along with the newly encrypted virus and MtE onto a new target.

As a result of this stage, the decryptor varied in each infectioin making it difficult for virus scanner searching for the tell-tale sequence of bytes that identifies a specific decryptor because there is no fixed signature, decryptor or no alike two infections.

```
        ; to "Start"-delta
mov     cl,03   ; (delta is 0x0D2B in this example)
ror     bp,cl
mov     cx,bp
mov     bp,856E
or      bp,740F
mov     si,bp
mov     bp,3B92
add     bp,si
xor     bp,cx
sub     bp,B10C ; Huh ... finally BP is set, but remains an
        ; obfuscated pointer to encrypted body

Decrypt:
mov     bx,[bp+0D2B]    ; pick next word
        ; (first time at "Start")
add     bx,9D64 ; decrypt it
xchg    [bp+0D2B],bx    ; put decrypted value to place

mov     bx,8F31 ; this block increments BP by 2
sub     bx,bp
mov     bp,8F33
sub     bp,bx   ; and controls the length of decryption

jnz     Decrypt ; are all bytes decrypted?

Start:
```

Figure 4: *An Example Decryptor Generated by MtE*

The complexity of each polymorphic virus ranges from simple to complex. Generally, it can be classified into different levels:

- **Level 1:** To generate a polymorphic virus, a scheme is chosen from a set of encryption/decryption schemes. An instance of the virus will have one of these schemes in plain text as shown in Figure 5. The public key for this encryption can be distributed to many takers to encrypt the message. This simple is so called "semi-polymorphic".

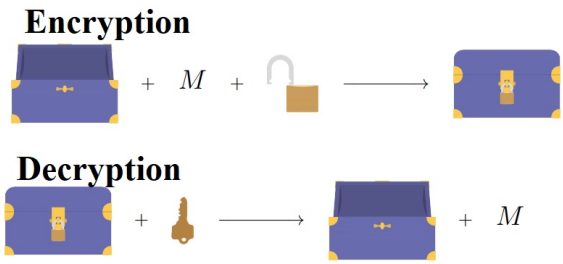

Figure 5: *A simple semi-polymorphic virus method*

- **Level 2:** Virus decryption routine contains one or several constant instructions, the rest is changeable as shown in Figure 6, the algorithm using the variables A and B but not the variable C, allowing C to be changed endlessly.

```
    lots of encrypted code
    ...
Decryption_Code:
    C = C + 1
    A = Encrypted
Loop:
    B = *A
    C = 3214 * A
    B = B XOR CryptoKey
    *A = B
    C = 1
    C = A + B
    A = A + 1
    GOTO Loop IF NOT A = Decryption_Code
    C = C^2
    GOTO Encrypted
 CryptoKey:
    some_random_number
```

Figure 6: *A simple polymorphic virus method*

- **Level 3:**The virus decryptor contains unused functions or instructions like NOP, CLI, and STI so on as shown in Figure 3
- **Level 4:** The virus decryptor uses interchangeable instructions and changes their order (instructions mixing) as shown in Figure 7
- **Level 5:** At this level, the polymorphic virus utilized all of the above techniques. In addition, the decryption algorithm is subject to change.
- **Level 6:** Per-mutating viruses. This is the highest level of polymorphic virus and is to be known as body-polymorphic virus or metamorphic virus. At this stage, the whole main

```
Decrypt:
 add  $4, %bh          ; junk
 xor  %edx, %edx       ; junk
 xor  %al, (%esi)      ; decrypt a byte with key in AL
 inc  %al              ; slide the key up
 xchg %edx, %ebx       ; junk
 inc  %esi             ; go to next byte
 cmp  %ecx, %edx       ; junk
 dec  %ecx             ; decrement the byte counter
 jnz  Decrypt          ; loop back if more to decrypt
```

Figure 7: *Instructions are order-independent*

code of the virus is subject to change as shown by generations in Figure 8.

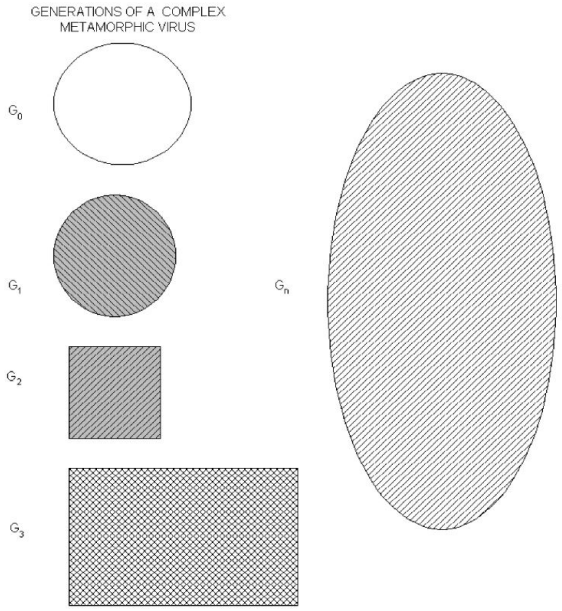

Figure 8: *Generations of complex metamorphic virus*

4 Changes to AV software

Due to the changes of polymorphic virus at different levels in the previous section, the anti-virus scanner and researcher have different strategies to fight back the virus.

The most handy work is to analyze the virus one by one, line by line, but this method is time-consuming, costly and impractical and also leads to mistakenly identifying one polymorphic as another. A generic method is employed [Nachenberg 1996], generally, this method assumes that:

- To avoid detection, the body of a polymorphic virus is encrypted.
- Decryption process must be performed before the virus can execute normally.
- Once an infected file executes, a polymorphic virus must take control of the system to decrypt the virus body, then yield control of the computer to the decrypted virus.

Based on this behavior, the anti-virus scanner loads the infected file into a self-contained virtual computer created from RAM. The infected file run as if it is running on a real computer. The execution is controlled by the scanner so that the virus can not do any damage to the real computer. When the virus runs, it exposes its body to the scanner, which in turn can search for signatures in the virus body that precisely identify the virus strain. If there is no virus to expose, the AV quickly stops running the file, removes it from RAM, and proceeds to scan the next file.

Speed is the key problem with generic decryption approach. It will be impractical if the polymorphic decrypts and executes in RAM for several hours. On the other hand, if the process of detection stops shortly, it may miss the main malicious code before it is able to reveal enough of itself for the scanner to detect a signature.

To overcome this drawback, a heuristics-based is employed. This method contains a set of rules that helps differentiate non-virus from virus behavior.This method works based on the assumption that normal operation will perform some math computations and uses these results. On the other hand, polymorphic virus may perform similar computations but throw away the results. The heuristic-based generic decryption looks for such inconsistent behavior to decide whether to extend the length of time a suspect file executes inside the virtual computer, giving a potentially infected file enough time to decrypt itself and expose a lurking virus. However, this method gives high false negative when it alters its rule base to detect new viruses. When virus authors try to make virus look like a clean program causing the scanner lengthen the time it needs to examine a suspicious file. Hence, this approach quickly become inaccurate, inefficient and obsolete in practice.

The Striker System

This system is provided by Symantec Cooperation Anti-virus company. The first step is similar to previous approach, that is, it loads the infected into the virtual computer from RAM. However, it does not rely on heuristic guess but on the profiles of the virus or rules specified to each virus, not to differentiate from non-virus and virus behavior.

When examining a new file, the system first tries to exclude as many viruses as possible from consideration. For example, some virus may only infect .COM files or .EXE files, or .SYS. When checking the infected file with extension .EXE, the Striker elaborates polymorphic virus that infects only .COM or .SYS files. If all viruses are excluded from consideration, then the file is considered to be clean the system will close and scan to the next file. After the preliminary step, if no infection is detected, the Striker keeps running the file in the virtual computer as long as this file has the behavior mapping with at least one known polymorphic virus or MtE.

The advantage of the Striker's approach is speed because the virus profiles not only enable the system quickly excludes some polymorphic viruses but also to process uninfected files quickly, hence minimizing the work load for the system.

So far, the generic decryption has been considered as the single most effective method of detecting polymorphic virus and the striker system improves on this approach.

As new anti-virus systems have been developed, virus authors also have new ways to write code, making this battle never ends.

In recent years, artificial intelligence has emerged as a new trends in many fields including malware detection. This promising method has been proved in the study of Asiru et al.[Asiru et al. 2017]. Their proposed model is shown in Figure 9 with the average accuracy detection rate of 80 percent.

5 Conclusion and Recommendation

In this paper, we have studied the polymorphic virus. Detection methods are analyzed based on static and dynamic analysis. Poly-

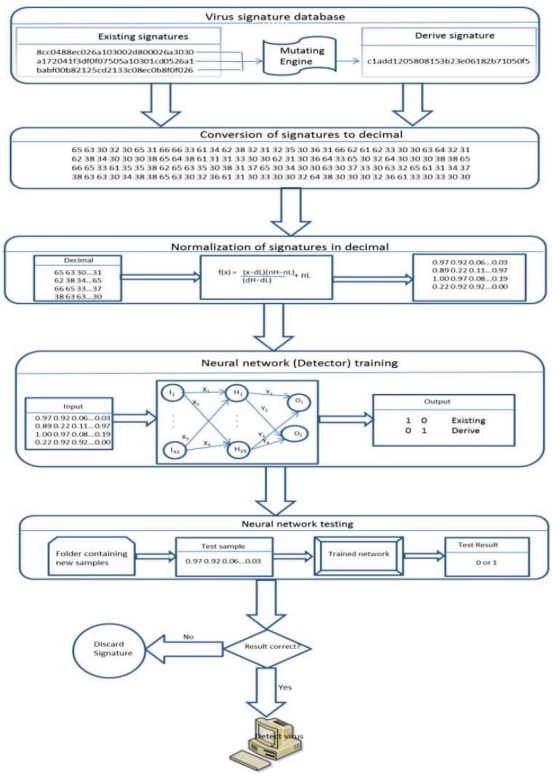

Figure 9: *The proposed model of using artificial intelligence for malware detection*

morphic virus are studied from low level to high level with some example codes. We also investigated how anti-virus software analyzes the infected file and shows pro-missing approach for malware detection in the future. To combat the never ending virus generation, the anti-virus software company should work closely with researchers to find potential approach that both work efficiency and accuracy.

References

AGRAWAL, H., BAHLER, L., MICALLEF, J., SNYDER, S., AND VIRODOV, A. 2012. Detection of global, metamorphic malware variants using control and data flow analysis. In *MILITARY COMMUNICATIONS CONFERENCE, 2012-MILCOM 2012*, IEEE, 1–6.

ALAZAB, M., VENKATRAMAN, S., WATTERS, P., AND ALAZAB, M. 2011. Zero-day malware detection based on supervised learning algorithms of api call signatures. In *Proceedings of the Ninth Australasian Data Mining Conference-Volume 121*, Australian Computer Society, Inc., 171–182.

ANCKAERT, B., MADOU, M., AND DE BOSSCHERE, K. 2006. A model for self-modifying code. In *International Workshop on Information Hiding*, Springer, 232–248.

ASIRU, O., DLAMINI, M., AND BLACKLEDGE, J. 2017. Application of artificial intelligence for detecting derived viruses. In *European Conference on Cyber Warfare and Security*, Academic Conferences International Limited, 647–655.

BONDARENKO, Y., AND SHTERLAYEV, P., 2006. Polymorphic virus detection technology.

BONFANTE, G., KACZMAREK, M., AND MARION, J.-Y. 2007. Control flow graphs as malware signatures. In *International workshop on the Theory of Computer Viruses*.

BONTCHEV, V. 1992. Mte detection test. *Virus News Int*, 26–34.

BORELLO, J.-M., AND MÉ, L. 2008. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology 4*, 3, 211–220.

CHAUMETTE, S., LY, O., AND TABARY, R. 2011. Automated extraction of polymorphic virus signatures using abstract interpretation. In *Network and System Security (NSS), 2011 5th International Conference on*, IEEE, 41–48.

CHRISTODORESCU, M., AND JHA, S. 2006. Static analysis of executables to detect malicious patterns. Tech. rep., WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES.

GRIFFIN, K., SCHNEIDER, S., HU, X., AND CHIUEH, T.-C. 2009. Automatic generation of string signatures for malware detection. In *International workshop on recent advances in intrusion detection*, Springer, 101–120.

MOSKOVITCH, R., ELOVICI, Y., AND ROKACH, L. 2008. Detection of unknown computer worms based on behavioral classification of the host. *Computational Statistics & Data Analysis 52*, 9, 4544–4566.

MUTTIK, I., AND LONG, D. V., 2005. Detecting computer viruses or malicious software by patching instructions into an emulator, June 14. US Patent 6,907,396.

MUTTIK, I., 2004. Detecting malicious software by analyzing patterns of system calls generated during emulation, Aug. 10. US Patent 6,775,780.

NACHENBERG, C. 1996. Understanding and managing polymorphic viruses. *The Symantec Enterprise Papers 30*, 16.

POLYCHRONAKIS, M., ANAGNOSTAKIS, K. G., AND MARKATOS, E. P. 2006. Network–level polymorphic shellcode detection using emulation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 54–73.

PORRAS, P., SAIDI, H., AND YEGNESWARAN, V. 2009. Conficker c analysis. *SRI International*.

ROGERS, A. J., YANN, T., AND JORDAN, M., 2012. Detection of viral code using emulation of operating system functions, Dec. 25. US Patent 8,341,743.

SPAFFORD, E. H., HEAPHY, K. A., AND FERBRACHE, D. J. 1989. A computer virus primer.

STEPAN, A. E. 2005. Defeating polymorphism: beyond emulation. In *Proceedings of the Virus Bulletin International Conference*.

SUNG, A. H., XU, J., CHAVEZ, P., AND MUKKAMALA, S. 2004. Static analyzer of vicious executables (save). In *Computer Security Applications Conference, 2004. 20th Annual*, IEEE, 326–334.

SZOR, P., AND FERRIE, P. 2001. Hunting for metamorphic. In *Virus Bulletin Conference*.

SZOR, P. 2005. *The art of computer virus research and defense*. Pearson Education.

THOMPSON, G. R., AND FLYNN, L. A. 2007. Polymorphic malware detection and identification via context-free grammar homomorphism. *Bell Labs Technical Journal 12*, 3, 139–147.

TORRUBIA-SAEZ, A., 2003. Polymorphic code generation method and system therefor, July 8. US Patent 6,591,415.

VON NEUMANN, J., AND BURKS, A. W. 1996. *Theory of self-reproducing automata*. University of Illinois Press Urbana.

YANN, T., AND PETROVSKY, O., 2006. Detection of polymorphic virus code using dataflow analysis, June 27. US Patent 7,069,583.