COMPACK – AN LFSR BASED UNPACKING & EXECUTION SOLUTION FOR WINDOWS BINARIES

Saptarshi Laha
M.Sc. Cybersecurity
School of Computing
National College of Ireland
Dublin, Ireland
x18170081@student.ncirl.ie

Abstract—Software piracy is a significant concern in today's world. Software developers and software development companies aspire to prevent piracy of their products by incorporating various software licensing mechanisms into it. Additionally, the software is delivered to the customers in a packed state to make reverse-engineering of the same exceptionally complicated and laborious, thereby preventing individuals from accessing the underlying software licensing mechanism and executable code. This paper proposes a mechanism of packing which aims to be a strong contender for both traditional and modern packing techniques. The discussed method involves handcrafting a portable executable which enforces reliable protection due to its low section entropy, negligible performance overhead, small file size, pseudorandom instruction generation, concealment of non-executing instructions, and, the self-modifying behaviour of the resulting executable file. These properties allow the constructed portable executable to be competent at subverting reverse-engineering tools targeted at unpacking packed executables, while also making manual unpacking of the same an arduous toil. It further helps preserve the integrity and confidentiality of the software by retaining the logic of the software licensing mechanism and executable code present underneath, despite its polymorphic and metamorphic nature, while keeping it away from the prying eyes of the analyst.

Keywords—Manual Binary Construction, Data Decompression, Code Obfuscation, Anti-Debugging, Anti-Static-Analysis.

CONTENTS

I. INTRODUCTION

Piracy of software has massively risen over the past decade. This rise has led most developers and development companies to use software licensing solutions and packing of software code, to prevent analysis and subversion of the same by an analyst. The incorporation of such software protection mechanisms in the software, however, has not dampened the spirit of the exceptionally inquisitive members of the technical community, including security researchers and black-hat hackers. They attempt to crack the newly implemented secure packing methods, taking it as a challenge to further their knowledge in understanding of the technology, or rendering the protection mechanism and licensing solution set in place useless, and distribute software for fame and profit, respectively. Thus, there arises a need for the development of new packing algorithms from time to time that provides temporary security to the software products and their underlying software licensing mechanisms.

Before understanding the internals and working of the proposed packer, it is essential to have a brief idea regarding the general unpacking process of a packer. This knowledge will enable an analyst to write an automatic unpacker or allow them to unpack the code concealed underneath for manual analysis. The most common mechanism of unpacking an executable in Windows starts with the saving of the register context at the entry point usually with a ***PUSHA*** instruction. Reserving of a portion of heap memory equivalent to the size of the unpacked Portable Executable (PE) sections or higher than it using one or a combination of functions such as ***VirtualAlloc*** or ***VirtualAllocEx*** follows if a previously reserved section does not exist in the section table of the PE file. The access protection flags of the reserved portion of the memory or reserved section that maps to the memory in case of a previously reserved section in the PE file, are then changed to Read, Write and Execute using one or a combination of functions such as ***VirtualProtect*** or ***VirtualProtectEx***. The next step includes decryption and decompression of the code and data sections of the actual executable into the reserved memory, loading and linking of the libraries that the original executable imported, and restoration of the register context saved at the entry point using a ***POPA*** instruction. A jump instruction to the original entry point of the unpacked executable succeeds, executing the unpacked PE file as it would if it was executed without the unpacker performing the previous operations [1].

The information presented above helps an analyst devise their attack strategy against it. In this case, the attack strategy happens to be extremely simple irrespective of the complexity involved in packing the executable. The return value of the ***VirtualAlloc*** or ***VirtualAllocEx*** functions is the first address of the newly allocated memory. Additionally, the first parameter of the ***VirtualProtect*** and the second parameter of the ***VirtualProtectEx*** functions is the first address of the memory section whose access protection flags require alteration, which happens to be the same address as the one returned by ***VirtualAlloc*** or ***VirtualAllocEx*** functions if it exists and precedes this function. Analysing the disassembly generated by the packed executable and setting the appropriate breakpoints, an analyst can quickly unpack a packed executable for analysis if it uses the unpacking strategy mentioned above, and, can eventually also develop a custom toolkit to do so automatically, massively reducing the analysis time involved. The study of this scenario poses a challenging yet crucial question regarding the possibility of dynamically generating instructions and executing it during program runtime such that manual or automatic unpacking is not possible or involves tremendous effort. The method of packing and unpacking proposed by this paper attempts to answer this question in the Research Method and Specification section of the document.

An insight into the Literature Review section reveals that the methods currently used for packing do not hold good against the current advances in packer detection and automatic or manual unpacking of the same. Hence, there is a need to present a novel idea in PE file packing which incorporates techniques from various advances in general computing such as self-modification of code, pseudorandom generation of opcodes, handcrafting of tiny PE files and maintaining of low section entropy to achieve the desired security. This security achieved is practical only if it does not come bundled with its fair share of performance overhead, and thus, ensuring the same with the utmost care is also an addition to the initial proposal by the paper. It is essential to understand the working of currently used packers and how the security enforced by them meets their demise at the hands of current date unpackers, discussed in the Literature Review section of the document, before proceeding with the discussion entailing the proposition presented by this paper. This layout is the intended flow of the document to ensure that the reader is at a better position to grasp and conceptualise the techniques involved to gain an in-depth understanding of the subject matter. The understanding gained is, in turn, is crucial in suggesting a new packing scheme and its extensive analysis for potentially overlooked flaws outlined in the later sections of the document. In addition to the content mentioned above, a few more critical topics are also present in the Literature Review section of the document which helps in the formulation of the proposed packing method, while also serving as an informative read.

II. Literature Review

This section includes discussions of various types of packers and their unpacking methods followed by various automatic unpackers and their working. The first subsection dedicates itself to making the reader wary of the types of packers present and their unpacking methods. In contrast, the following subsections scrutinise the approach of detection of packing in a PE file, identifying the type of packing present, and the methodology involved in unpacking the same, done by different automatic unpackers. It further highlights the loopholes that each of these implementations has, which then gets leveraged to present the proposal for a new packing method. Practically, a security researcher would only be interested in sharing their findings in case the packing of malware utilises a packer. Hence, most of the automated unpacking tools mentioned here find its use in malware analysis; however, they can find their use for unpacking commercial software as well, packed using that specific packer or packing method.

Later subsections discuss critical information regarding miscellaneous topics which find their utilisation in the implementation of the proposed packing method. Generally, these topics do not directly lead to the crafting of a new packing method, but aid in the process of refining the same against current advances in unpacking methods by making them automatic unpacking resistant. It also serves as an excellent primer for learning questionable techniques that usually get exploited by black-hat hackers or security researchers in performing non-conventional tasks that are extremely difficult or impossible to perform using regularly used methods of development. Hence the reader must be wary of the techniques presented in these subsections to evaluate the advantages and limitations that each of these fancy methods brings along.

A. Types of Packers

This section discusses the different types of packers categorised based on their complexities, as researched by Ugarte-Pedrero et al. [2]. This information is crucial to understanding the discussion of the automatic unpacking methods that follow. The division of packers based on its complexity results in six different types as described below:

1) *Type 1 Packers* – These types of packers perform a single layer of unpacking before transferring the control flow to the unpacked code.
2) *Type 2 Packers* – These types of packers contain multiple unpacking layers. Each unpacking layer is responsible for unpacking the subsequent unpacking routine. On the entire reconstruction of the actual program, the control flow transfers to the unpacked code for execution.
3) *Type 3 Packers* – These types of packers are like the previous types of packers with the exception being that the unpacking does not follow linearly but gets organised in a more complex topology that includes loops. Due to this structure, the original packed code may not reside in the last unpacking layer. Instead, the last layer generally contains various anti-debugging mechanisms, integrity checks or part of the obfuscated code of the packer. A tail jump still exists to separate the packer code from the actual program code.
4) *Type 4 Packers* – These types of packers are single-layered or multi-layered packers that have a part of the packer code, which is not responsible for unpacking, interleaved with the execution of the original program. There, however, exists a precise moment in time when the entire actual program code gets unpacked in memory. The tail jump responsible for the transition from packer code to program code can be challenging to locate.
5) *Type 5 Packers* – These types of packers have the unpacking code jumbled with the original program code, such that the layer containing the original code has multiple frames, and the packer unpacks them one at a time. Though these packers have a tail jump for the transition, only one frame of code gets revealed, and thus, there is a requirement for a snapshot of the process memory after the execution of the entire program for analysis of the original code.
6) *Type 6 Packers* – These types of packers are the most complex packers as they only unpack a single fragment of the original program at any point of time.

The next few sections focus on different approaches to automatic unpacking, as discussed by different researchers. These approaches play a crucial role in making the reader understand the unpacking process carried out by different automatic unpackers to help them comprehend the motivation behind the confident design choices that exist in the packing method proposed by the paper, before introducing them to its nitty-gritty details.

B. PolyUnpack

This subsection discusses a common malware unpacking mechanism proposed by an unpacker named PolyUnpack [3]. The method of unpacking used by Royal et al. in their PolyUnpack project includes disassembling of the program to identify code and data in the first step of the process. The executable instructions gathered in this process helps form a set of original instructions of the program. Next, it executes an instruction in the program, saves the current value of the instruction pointer and performs an in-memory disassembly starting from the current value of the instruction pointer until it encounters a non-executing instruction. Finally, it performs a check to verify if the instructions gathered in the previous phase are a subsequence of the original instruction set. In case instructions gathered do not happen to be a subsequence of the original instruction set, then the unpacked code starting from the current instruction pointer is returned. Otherwise, this process continues until the last instruction of the executable is extracted.

The algorithm mentioned above is ingenious but fails to work in some instances including if the executable detects that it is being executed by PolyUnpack, as it can alter its flow of action thereby preventing automatic unpacking or leading to unpacking of unintended code. Hence, it is vital to keep this property in mind when suggesting the design for the proposed custom packer. Apart from this, PolyUnpack also fails on step-by-step unpacking code as it extracts unpacked code starting from the first instruction change and exits. The algorithm can be slightly tweaked to run for the complete executable irrespective of an instruction change for better results. However, this will result in the extraction of ***N*** unpacked code, where ***N*** is the number of instruction changes that occur during the execution of the executable file. One further optimisation is possible, which is, during the extraction of the ***N*** unpacked code, the standard instruction subsequence undergoes omission and the changed instructions stitched together to form the actual unpacked code. This modification will eliminate the extraction of ***N*** unpacked code, and instead extract only once irrespective of the number of instruction changes occurring during program execution. This modification is helpful against packers which have multiple instruction changes during program execution. It is, however, essential to note that these modifications are not present in the original implementation or proposal of the same but are suggestions to cover up some of the flaws of the unpacker. Even in this case, if the instruction subsequence completely changes, then the unpacker will output garbage data that cannot undergo stitching to form a valid portable executable file.

The next subsection describes the working of another malware unpacker which serves as an improvement over PolyUnpack in detecting and extracting packed code and incorporates some of the functionalities suggested to make PolyUnpack better while taking an entirely different approach algorithmically. The paper presented by Royal et al. highlights a vital piece of information regarding the determinability of the presence of hidden code being an undecidable problem. This property gets leveraged in this paper to propose an anti-unpacking packer solution.

C. Renovo

This subsection discusses another automatic unpacking algorithm proposed by Kang et al. and commonly known as Renovo [4]. Although the approach of Renovo is substantially different from PolyUnpack discussed in the previous subsection, they do have their fair share of similarities as well. Renovo works by generating a memory map whenever an executable gets loaded onto memory. This generated memory map is labelled the clean state. On encountering a memory write instruction by the executable, the corresponding memory region gets marked dirty. If an instruction pointer jump occurs to any of these newly generated regions, it gets marked as the original entry point for the unpacked code and extracts the code present in the memory region. Renovo, however, adds functionalities to unpack multi-layered packing by analysing the code and data sections in the newly unpacked code and recursively performing the previous operations until the final unpacking occurs.

This approach takes into consideration the possibility of multi-layered packing to exist, unlike PolyUnpack. However, Renovo, just like PolyUnpack also extracts N copies of data in case of N different jumps to memory regions marked dirty in memory. Just like in the previous case, this methodology of unpacking is rather inefficient against step-by-step unpacking packers and also is ineffective against other anti-reversing methods implemented such as code obfuscation and detection of Renovo unpacking the code by the packed executable and modifying its behaviour accordingly to thwart analysis. Thus, even though Renovo is a significant upgrade from PolyUnpack, it is still not the best or most efficient approach to unpacking hidden code.

The discussion of the next subsection entails the use of heuristics as well as a statistical model to aid the unpacking process of the packed executable file. This method is far more complicated than the currently reviewed methods to unpack packed executables while having a lower performance overhead and executing faster than the previously discussed methods under general circumstances. There, however, is another unpacking mechanism called OmniUnpack [5] that deserves mentioning before proceeding with this transition. The reason why this mechanism lacks a detailed explanation in the current text is that it is more focused on analysing malware rather than being generalised to any packed executable, unlike the previously mentioned methods, while following a similar methodology as Renovo in the background to help facilitate the analysis of packed code during its unpacking stage. It is also important to mention that OmniUnpack is more efficient than Renovo due to choices made by the authors that allow for lesser performance overhead.

D. Eureka

This subsection dedicates to the analysis of another unpacking algorithm proposed by Sharif et al. known as Eureka [6]. Eureka combines heuristics-based and statistical-based unpacking to unpack hidden code from a packed executable during its runtime, along with child process monitoring. The heuristics-based unpacking used by Eureka waits for the termination of the running process by the interception of the ***NtTerminateProcess*** system call, for dumping a snapshot of the program's virtual memory address space. This approach assumes that since the program has unpacked and executed, the unpacked instructions are present in the memory. Besides, it also takes into consideration the creation of child processes by the executing process to aid in its unpacking or perform execution of a crucial subroutine and thus also waits for the interception of the ***NtCreateProcess*** system call to start monitoring the child process invoked and apply the same heuristics principles for its analysis. The statistical-based unpacking involves modelling the statistical properties of the unpacked code. This modelling bases itself on two assumptions which are – specific opcodes, usage of registers and instruction sequences are more prevalent than others in executables, and, the volume of code increases as the packed executable unpacks itself. When a particular section of memory defies the first assumption or behaves according to the second assumption, that memory section gets dumped.

Eureka is undoubtedly a much more complex unpacking algorithm compared to the previously discussed ones. It additionally has a negligible performance overhead compared to them as a result of the lack of continuous processing of the executing instructions of the program. Instead, it depends upon the invocation of one of the two pre-defined system calls or an anomaly in its statistical predictions, to extract the unpacked code from the memory region. Although Eureka is far more advanced in its unpacked code analysis patterns, it still fails to hold up against step-by-step unpacking and execution of instructions or analysis of the execution environment by the packed executable to modify its behaviour, as mentioned by the author. A simple modification involving the interception of the ***VirtualAlloc***, ***VirtualProtect*** and other calls with similar functionalities can, however, make this unpacker much more potent than it is. However, this would add to the performance overhead as these functions often invoked by non-packer based executables and could result in erroneous results and findings due to the same reason.

In the next subsection, the assessment of another automatic unpacking mechanism that utilises logging of every instruction executed and further parsing of the log to extract a partial unpacked file occurs. Thus, the next subsection marks the onset of a new generation of automatic unpacking algorithms, and hence these algorithms are more complicated compared to the previously discussed ones, apart from being modular in design. This modularity means that the algorithm depends on multiple tools at multiple points in the extraction process to assist with extracting the final unpacked code. However, before proceeding to the next subsection, there is another unpacking method that deserves special mention at this point, known as WaveAtlas [7]. This unpacker is however not discussed in much detail as this automatic unpacker works based on two hypothesis – the executed instructions starting from the initial memory image till the second to last memory image serve to protect the concealed program, and, the executed instructions in the last memory image contain the actual program. These assumptions do not hold in case of step-by-step extraction and execution of packed programs, and the paper also mentions the partial validation of the hypothesis, which leads to an uninteresting read. Hence, the creation of a separate subsection for the extensive analysis of the same did not seem worthwhile.

E. Mal-Xtract, Mal-XT and Mal-Flux

This subsection deals with the introduction of a rather complicated set of methods for hidden code unpacking. The first unpacking method discussed is proposed by Lim et al. and known as Mal-Xtract [8]. The methodology proposed utilises PANDA to execute the packed executable while recording the entire system emulation in a log file. The recordings in the log file get replayed to detect the memory addresses that got written to during the execution of the program, and, the written memory addresses and their adjacent memory addresses are marked as written. An empirical approach then gets used to determine the threshold value based on the number of instructions written for consideration of the memory sections for being dumped. Finally, Volatility is used on the physical memory dump to extract the sections that meet the threshold value to try and reconstruct the unpacked executable code. Further, the extracted section data gets parsed using IDA Pro, and their corresponding mnemonics gathered for comparison with the original executable file using a diff tool. The similarity percentage and the entropy then get used to determine the presence of packing in the executable.

Although this approach uses multiple toolkits and employs sophisticated methods for carrying out partial unpacking and validation of the presence of packing in an executable file, it still has its fair share of shortcomings. Before mentioning these shortcomings, it is crucial to highlight the improved methods that this automatic unpacker employs including logging of the entire emulated execution, which is very useful in analysing and extracting hidden data from packed executables. However, there is no check performed to verify if the extracted instructions at every stage get executed or not, leading to the generation of copious amounts of garbage data which thwarts analysis. Besides, the empirical approach used for the calculation of the minimum, and maximum threshold values also depend on the number of unpacked instructions at every given instance. It does not take much effort to dupe this mechanism into working incorrectly by merely extracting the same amount of code at any given instance of unpacking. Finally, the reconstructed executable cannot execute directly due to the Import Address Table (IAT) being damaged or missing, which further makes this automatic unpacker inept.

The following unpacking method of interest is an advancement from Mal-Xtract. Lim et al. propose it, and it is known Mal-XT [9]. Mal-XT replicates the previously discussed method but adds the functionality to track the execution of instructions. This addition further helps in separation of executable code from garbage code and data. In other words, the extraction procedure for hidden code only extracts the data from memory sections that undergoes execution. Although this additional enhancement to the initially proposed algorithm of Mal-Xtract helps in neglecting sections of code that do not get executed, it still does not take into account the presence of logically opposite pairs of code being present in the memory section that executes. Besides, multiple executions of a program are required with different parameters to follow all the branches and paths to derive the complete executable, unlike the previously discussed method. Also, just like the previously discussed method, this method results in the reconstruction of an unpacked executable file that cannot execute directly due to the IAT being damaged or missing.

Next, the discussion of the unpacking method proposed by Lim et al. is known as Mal-Flux [10] is considered. This method, just like Mal-XT, is an advancement from Mal-Xtract. It also analyses the same memory addresses for being rewritten with code apart from monitoring new memory or section writes and rewrites. This property makes it much more advanced than the previously discussed automatic unpackers. This additional recorded information gets utilised in the reconstruction of the executable file at the end of the process, although just like the previously discussed unpacking methods, this method also results in the generation of an executable that cannot execute directly due to the IAT being damaged or missing. Besides, the most impactful disadvantage of Mal-Flux is the fact that it is hugely performance heavy, apart from not tracking the execution of instructions to separate actual code from garbage code and data which is crucial to the refining of the results.

It is of great interest to mention that all these automatic unpacking techniques mentioned above use additional methods for activities such as analysis of malware or segregation of the malware executables and the benign executables. However, as these methods do not aid the formulation of the packer proposed in this paper, they are omitted. Additionally, all the previous subsections were related to dynamic unpacking methods of packed executable files. The next subsection deals with the static unpacking of packed executable files which follows a different approach than the ones currently employed.

F. Static Unpacking

This subsection deals with the static unpacking of packed executables, as proposed by Coogan et al. [11]. Unlike the previously discussed methods, this method of hidden code unpacking does not depend on the execution of the packed executable but rather depends on disassembly, static code analysis, alias analysis, possible transition point detection, static unpacker extraction and static unpacker transformation to achieve the desired result. It starts by generating a static disassembly of the code, followed by identification of basic blocks to construct a control flow graph of the disassembled code. Next, it performs binary-level alias analysis to identify the memory addresses where indirect memory operations occur. The results generated in the previous step get used to determine potential transition points where the unpacked code gets executed by the transferring of the control flow. The operations performed following any individual path corresponding to any unique transition point is analysed to identify the memory locations that may be modified apart from performing backwards static slicing to identify the static unpacker. This static unpacker then gets analysed to perform various transformations such as detection and removal of code protection mechanisms on it. Finally, the statically analysed and modified code gets executed to result in the unpacking of the hidden code.

This method of unpacking attempts to render any defence mechanisms set in place in the unpacker code useless and, further, execute the unpacker to extract the unpacked code. This technique finds utilisation in single-level unpacking, but if the packing of code is present in several layers, then each layer would only unpack the subsequent layer of code. Additionally, if it is a step-by-step unpacking packer, then such methods fail to work practically due to the extracted code having no meaning by itself. Due to these limitations, this type of automatic unpacker is not feasible for analysis of densely packed or radically instrumented binaries. This subsection marks the end of the discussions of various automatic unpacking techniques. In the following subsection, the discussion entails the importance of entropy for detection of packed executables. This information will help the reader understand the need behind keeping the entropy low in the packing algorithm proposed by the paper.

G. Entropy-based Packer Detection

This subsection deals with introducing the reader to the concept of entropy and further explains how it acts as a metric for packer detection. The entropy of a block of data describes the amount of information it contains, and its calculation is as follows:

$$H(x) = -\sum_{i=1}^{N} \begin{cases} p(i) = 0, & 0 \\ p(i) \neq 0, & p(i)log_2 p(i) \end{cases}$$

where ***p(i)*** is the probability of the $\boldsymbol{i^{th}}$ unit of information in event ***x***'s sequence of ***N*** symbols. For packed program analysis, the unit of information is a byte value, ***N*** is ***256***, and an event is a block of data from the packed program [12].

Compressed and encrypted code and data result in the calculation of high entropy values, whereas the general code and data result is much lower entropy. Hence identification of packed data is rather simple based on the calculation of its entropy value. Bat-Erdene et al. also propose a method to utilise entropy analysis for detection of multi-layer packing in executables [13]. This analysis calculates the entropy of all the unpacked sections recursively during to unpacking process to determine if any of the unpacked sections further contain packed code. If the entropy values for a section are above the threshold limit, the section is considered packed, and further unpacking of the section follows.

The information presented above is critical to the formulation of the method of packing proposed by this paper. However, a few more primers on essential topics are required to formulate the same altogether. The next subsection dedicates itself towards introducing the reader to the concept of self-modification of code and its linearization, which is one of the most critical concepts that need special mention due to the massive role it plays in the formulation of the proposed packing method by this paper.

H. Self-modifying Code

This subsection deals with the introduction of self-modification of code, while also explaining how the linearization of the same occurs using State Enhanced – Control Flow Graphs as proposed by Anckaert et al. [14]. Self-modification of code refers to the modification of any piece of the code within a running program, by the same running program for altering the operation it performs when it encounters that code. Data transfer operations in a writable and executable section in memory can facilitate self-modification of code to alter its behaviour. Malware writers generally exploit this method, but it can also lead to the generation of dynamic code in a specific section in memory and executing it to perform various fancy operations. In usual programs that do not exploit this behaviour, a control flow graph is constructible, which shows the various execution paths an executable can take during its execution. However, a general control flow graph is not constructible for a program utilising self-modification of code.

Anckaert et al. propose a method to generate a model of control flow graph that takes into consideration the different states the program is at and details the subsequent transition or transformation related to it. Unfortunately, this linearization using State Enhanced – Control Flow Graphs is only possible if the targets of indirect control transfer within a program are known. If they are not known, then the State Enhanced – Control Flow Graph model has to take into consideration every byte getting altered at any particular time within the program and the control flow getting passed to all of them which makes the problem infeasible to solve. Thus, in other words, if the addresses where the instructions are getting altered and executed are not constant, then it is challenging for this model to linearise the disassembly generated. In the case of the implementation presented by this paper, the VirtualAlloc or VirtualAllocEx type of functions find their utilisation, which in turn allocate a size of memory in the heap section starting from a random address, and thus the previous model cannot be used for analysis. Additionally, the control transfers are pseudorandom and can occur at any point leading to further complexities.

The next section discusses the obfuscation of executable code which helps thwart analysis of packers and packed code using disassemblers. This topic is of exceptional importance as it helps build the anti-analysis mechanisms of the packer, which is vital to its unpacking methodology.

I. Obfuscation of Executable Code

This subsection introduces the reader to intentional code obfuscation methods that thwart the static disassembly and analysis process by disassemblers and analysts, respectively, as proposed by Linn and Debray [15]. The authors highlight that the insertion of junk instructions at specific locations are extremely useful in confusing the disassembler. Additionally, partially or fully overlapping instructions also find their use in hindering the disassembly and analysis process. Linn and Debray suggest that to confuse the disassembler the junk code inserted needs to be inserted where the disassembler expects to find code, apart from these instructions being partial and inserted at locations where they are unreachable during runtime to preserve the program semantics.

There are three effective ways to introduce junk code to thwart linear sweep and recursive traversal methods of code discovery and disassembly followed by disassembler, these include, call conversion, opaque predicates and jump table spoofing. Call conversions utilise misbehaving call instructions that do not return to the caller after its execution, while opaque predicates rely on the fact that a conditional instruction leads to the generation of two branches of code, whereas only one branch of code always executes in reality. Jump table spoofing refers to the introduction of maliciously fabricated jump instructions into sections of code containing garbage data.

Additional methods of code obfuscation include instruction equivalence which deals with the breaking up of a single instruction into multiple smaller instructions, or the method of performing logically opposite functions to confuse the analyst. Another novel idea in code obfuscation comes from the Turing-completeness of the ***MOV*** instruction in the Intel Instruction Set Architecture, as researched by Stephen Dolan [16]. This property of the instruction allows every other instruction in the Intel architecture to be replaced by a sequence of unconditional data transfers. Its usage results in extremely difficult to read assembly code that achieves obfuscation at the cost of a considerable performance overhead due to the need of every instruction having to execute due to the lack of conditional jump instructions being present.

This subsection marks the end of the literature review section. The code obfuscation methods learnt here are utilised in the formulation of the packer presented in the next section. The techniques learnt in this section help in some or the other way to build a robust packer. A few additional topics were worthy of mention, but since they do not require extensive analysis for the formulation, they were ignored and are instead lightly touched upon in the subsections of the next section for the sake of brevity.

III. Research Method and Specification

This section of the paper is responsible for outlining the specifics of the packing technique proposed and its analysis, followed by an in-depth look into the proposed method, an overview of the project plan and the testing to be carried out to validate the claims made.

A. Packer Specification

The loopholes of different unpacking mechanisms along with the additional information on fancy techniques such as code obfuscation and self-modification of code help in the formulation of the packing method presented here. The packer unpacks in a step-by-step instruction execution scheme, which means that ***VirtualAlloc*** or ***VirtualAllocEx*** and ***VirtualProtect*** and ***VirtualProtectEx*** get exploited massively throughout the code. Additionally, as inferred earlier, data compression or encryption leads to increase in entropy values and thus, such methods get entirely avoided in this approach. Instead, the implementation of an opcode generation algorithm occurs, which generates the subsequent opcode depending on the previous opcode value. The aim is to then place the generated instruction opcode into the newly allocated memory by ***VirtualAlloc*** or other such functions which is flagged readable, writable and executable by ***VirtualProtect*** or other such functions and executing it. Next, the ***VirtualFree*** or ***VirtualFreeEx*** function gets used to free the allocated memory to remove traces of the unpacked code, thereby preventing dynamic analysis of the same. However, if a single method of execution follows for all generated opcodes, the unpacking becomes extremely easy to track, and thus a combination of a condition jump such as ***JE*** or ***JNE***, ***CALL***, and ***RET*** find their utilisation to maintain the anonymity of the flow of code. The utilisation of these instructions for control flow allows the packer the freedom to insert junk code to thwart the static analysis by disassemblers and decompilers as mentioned earlier. A few of intermediate instructions are also replaced completely by unconditional data transfer operations using ***MOV*** to hinder the analysis further. However, an entirely ***MOV*** based unpacking approach does not find its use in this proposal due to the exceptional amount of performance overhead that it carries. Self-modification of code is additionally used in every step of the unpacking process to overwrite the preceding and following instructions in the unpacking process with logically equivalent instructions to make the executable polymorphic and metamorphic.

The packing method, on the other hand, utilises breaking down of the ***.text*** section of the PE file into opcodes of executable instructions used to construct the opcode generation function based on LFSR logic. Dealing with the other sections are a bit more complicated and are dealt with in the proposed method subsection of the paper. The entirety of the implementation will get done from scratch using ***C, C++*** and ***x86 Assembly*** languages with the only exception being the reuse of code from movfuscator [17] for the conversion of general instructions into their ***MOV*** counterparts. It is essential to mention that the implementation of this proposal targets the 32-bit Windows operating environment but executes on the 64-bit Windows operating environment as well, albeit not taking advantage of the ***fastcall*** mechanism and therefore underperforming compared to 64-bit standards. Additionally, the implementation can get extended to native 64-bit execution as well as to other platforms, but the extension of the same does not fall under the scope of this project. The analysis for the design choices made follows in the next subsection.

B. Analysis of Design Choices

It is simple to analyse why a step-by-step packer model is chosen over a layered or entirely unpacking packer model. This design choice exploits the fact that none of the automatic unpackers discussed can completely unpack such a packer. Additionally, the latest generation of automatic unpacker that employs logging of every executing instruction is rendered useless due to the polymorphism and metamorphism exhibited by the unpacking code of the packer. The decision to employ obfuscation of code is essential to thwart static disassembly or decompilation of the code difficult apart from the linearization of self-modifying blocks. Even in the case of successful disassembly, the linearization of self-modifying blocks is still not possible due to the overall unpacking and unpacker code getting modified at every step of unpacked code execution. Finally, ***C***, ***C++*** and ***x86 Assembly*** languages get used for the implementation of this project over their higher-level counterparts due to the sheer amount of low-level control needed for the fruition of this method that the latter fails to provide and to reduce the performance overhead from using the latter. This project aims the Windows 32-bit execution environment as it can then execute on both 32-bit Windows and 64-bit Windows environments without needing any changes to its initial implementation. However, none of these assertions claimed is guaranteed, and the actual testing will be the determiner of this method's true potential. Before proceeding to the testing and test parameters, an in-depth understanding of the packer and its unpacking method is essential, which happens to be the subject of the next subsection.

C. Proposed Method

A modular approach gets used to explain the proposed method to the reader in this subsection. The modular topics covered correspond to actual project modules that need to undergo development for the implementation of this packing method.

1) Instruction Extractor

The goal of the instruction extractor is to extract instructions from the ***.text*** section of the PE file that undergoes packing. This process is rather complicated due to the target platform of 32-bit Windows powered by x86 has a Complex Instruction Set Architecture. The instructions are difficult to determine due to the sheer number of total opcodes and the property of Intel x86 architecture to have two or more possible opcodes for performing some operation [18]. Additionally, there is no fixed opcode size which makes the process even more complicated when it comes to instruction decoding. In order to decode the instructions from a well-formed PE file, the implementation uses the suggestion from the Intel Software Developer Manual along with the rule that the instruction decoder that decodes and executes the opcodes can process a maximum of 15 bytes of information. The image below shows the process of opcode to instruction conversion, as suggested by the Intel Software Developer Manual.

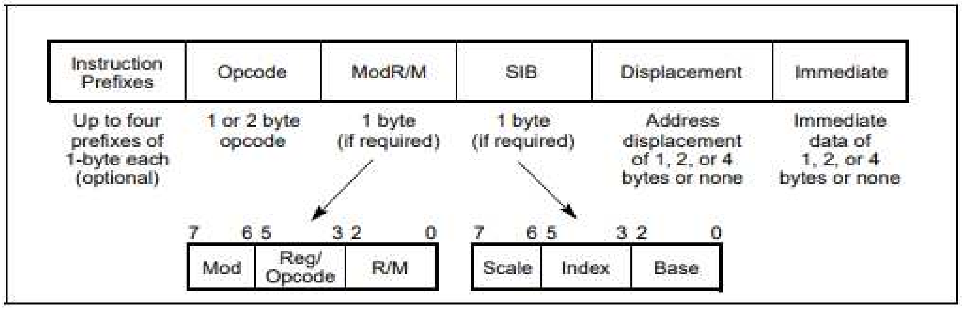
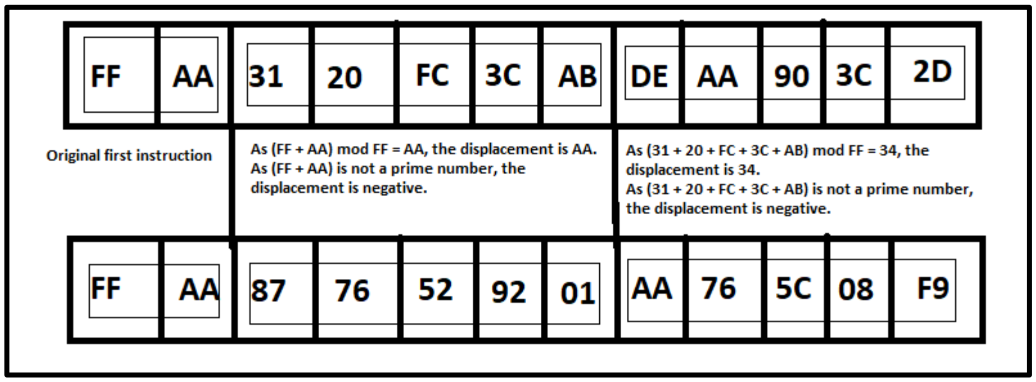


Fig. 1. Intel Microprocessor General Instruction Format

Although this specification image for the general instruction format allows for up to 16 bytes, the instruction decoder does not allow it and only processes up to 15 bytes for any single instruction as mentioned earlier. It is also important to note that there is a lack of opcode ambiguity due to which two separate instructions may have similar opcodes. This phenomenon is essential for both decoding the instructions correctly while introducing junk code seen later in this subsection.

Another essential feature that is implemented by the Instruction Extractor is to generate an LFSR-like sequence of instructions given the starting opcode, which finds its use for the unpacking routine. In other words, the function ***Gen(value, size)*** should be able to produce the next instruction of the specified size on an input of its current opcode value. There are multiple methods possible for achieving this, and the method chosen here for the sake of simplicity involves an array storing all the opcodes from the actual PE file. The sections of the array that correspond to different instructions have a displacement equivalent to the sum of the opcodes in the previous instruction mod 255. Moreover, the positive or negative displacement of the same is dependent on if the number generated is prime or not. A depiction of the process is present in the figure below.



FF AA 31 20 FC 3C AB DE AA 90 3C 2D
Original first instruction
As (FF + AA) mod FF = AA, the displacement is AA. As (FF + AA) is not a prime number, the displacement is negative.
As (31 + 20 + FC + 3C + AB) mod FF = 34, the displacement is 34. As (31 + 20 + FC + 3C + AB) is not a prime number, the displacement is negative.
FF AA 87 76 52 92 01 AA 76 5C 08 F9

Fig. 2. Opcode Hiding

2) *Section Loader*

While some sections can be removed entirely from the packed file and its crux placed inside the packer to reconstruct the same, other sections as the ***.data*** section cannot be processed similarly. This difference is due to the presence of essential program data in such sections that cannot be reconstructed by the packer, or, even if it undergoes reconstruction, it will leak potential data related to the packed program which is undesirable. This issue leads to the noteworthy handling of such sections using a section loader to maintain the integrity of the data while being concealed. The method used is the same as the instruction encoding in the previous module, except for the fact that this time, the data gets encoded. Since data sizes are not limited to 15 bytes, it leads to more complications in the encoding than the instruction. This issue is solved by dereferencing the lookups or writes to such sections from the opcodes in the ***.text*** section and then using this knowledge to encode the data appropriately.

Additionally, section loader is also responsible for the loading of encoded instructions onto newly allocated memory and executing it. A combination of ***JMP***, ***CALL*** and ***RET*** instructions get employed for the transfer of control flow to make the execution process ununiform. The decision on which instruction gets executed for control flow depends on an addition mod 3 of the opcode values. Since the array values are self-modified at every step as will be seen further in the proposal, the section loader needs to handle the lingering stack and heap values carefully in order for the program to maintain its integrity and semantic flow.

In case of a simple ***JMP*** instruction, the execution gets directly transferred to the newly generated block of code. The corresponding return to the unpacker code should be using ***JMP*** as well to maintain the integrity of the stack values. However, an additional return address needs pushing, for calling ***RET*** instead of ***JMP*** to return from the program execution code to the unpacker code. Similarly, a ***CALL*** instruction can also undergo execution without a subsequent ***RET*** being present in the unpacker code. The issues that arise when a ***CALL*** instruction gets used for control flow transfer with a subsequent ***CALL*** instruction executed from within the unpacked code. In this case, the stack is off by 4-bytes due to the saving of the return address of the caller. Hence, an additional ***POP Register*** instruction gets placed into the memory for balancing the stack, and to avoid altering the semantic flow of the program.

3) *Addition of Code Obfuscation*

As explained earlier, the addition of code obfuscation can occur by the insertion of junk code after jumps, or by transferring control flow to bad sections in the file with partial opcodes being present. A simple approach proposed to achieve obfuscation, in this case, involves the conversion of all ***JMP*** instructions to their conditional jump variants and performing an operation that sets the appropriate flag before the execution of the conditional jump so that it always gets executed. Additionally, all the padding code, including instructions like ***INT 3*** needs replacement by ***JMP***, ***RET***, and ***CALL*** instructions to throw off the disassembler of the decompiler used to analyse the code statically. Furthermore, the corresponding memory locations that falsely get jumped to have partial opcodes placed in them to confuse the disassembler or decompiler employed and throw off its analysis of the file.

Enforcing the strategy mentioned above requires the presence of code pads between subsequent opcodes filled with partial opcodes and garbage data. The number of code pads ***N*** should be equal to or greater than the sum of the number of original code pads that are replaced by ***JMP***, ***RET*** and ***CALL*** instructions and the number of unconditional jumps present in the code that gets converted to a conditional jump instruction. This methodology will also exhibit code overlaps which are further useful in attaining robust obfuscation. Apart from that, the conversion of instructions to unconditional data transfers using the ***MOV*** instruction should occur for non-transfer control related operations to subdue further attempts to disassemble or decompile the code statically or dynamically.

An additional method to perform obfuscation is to use the Tiny PE method of creation of an executable file at the very last stage [19]. This modification performs two vital operations, including reduction of the PE size on disk apart from stripping away headers and making the PE file seem invalid further thwarting the process of analysis of the same. The implementation of the last type of code obfuscation possible is already well defined by Section Loader subsection and including faulty ***CALL*** and ***RET*** instructions. Hence additional incorporation of the same is not covered in the scope of the current implementation due to the complexities that arise with maintaining the stack with such misbehaving instructions, especially when paired together with unpacked code ***CALL*** instructions.

4) *Self-Modification of Code*

Self-modification of code is one of the most potent methods employed by this proposal in building a robust unpacker for the packer. Introduction of polymorphism and metamorphism in code gets done using this method. The unpacking routine processes the instructions present in the array from the instruction extractor and executes them. Although the instructions are encoded, they still pose a chance of data leak. Hence another layer of protection finds its need in enforcing strong security of the encoded instructions. Self-modification of code gets used in a way that the instructions remain logically equivalent to their previous states, although the resulting opcodes get changed. This property gets implemented by changing the ***NOP*** instructions to their equivalent of ***XCHG RAX***, ***RAX*** and other such transformations.

Additionally, the unpacker code itself also changes during the runtime through the use of self-modification of code. The process in which this gets enforced is by the use of multiple variants of the compiled code embedded in the original instruction array. The goal is then to swap out the preceding and following instructions with a copy of their original code which results in a different opcode. An issue that needs addressing here is that since the compilations are different, the opcode lengths might differ and the result in erroneous instructions getting executed. This issue gets fixed by using the most massive opcode variant of a particular instruction for an operation in the first copy of the packer and subsequently replacing the same by smaller alternatives. Then the usage of ***JMP*** can lead back to balancing of the offset into the code as a disbalance might occur due to the changing of opcodes on every instruction execution of the original program.

The same self-modification of code concept can find application in the modification of the packed code while keeping the semantic flow of the packed program intact. This extension needs the modification of the ***Gen(value, size)*** function defined in the Instruction Extraction section. The ***Gen(value, size)*** should deterministically be able to generate the resulting opcode gives the current opcode and the next opcode size. It requires the definition of a function instead of an array for returning the same based on the current offset of execution. Hence, if ***Gen(value, size, offset)*** is defined, then the offset parameter is used to retrieve the offset into the packed program and allows conversion of opcodes in the function to their equivalent instruction with encoding by just balancing every previous and subsequent opcode by the new opcode generated. This method was previously not possible as there was only a single instruction copy of every instruction of the packed program, however, adding the offset parameter to the equation allows enormous amounts of freedom that is needed to convert an operation resulting in a single instruction to an operation resulting in multiple instructions. Here the offset acts as a specifier of the number of instructions from the packed program that have been executed and prevents re-execution of the same instruction twice if a single instruction gets replaced by multiple instructions in the resulting self-modified logically equivalent code.

The last step involves cumulating all the advances from the different modules together to form the packer and test its stability. The testing planed in outlined in the next subsection.

D. Test Plan

This subsection describes the test plan for the general development of the packer and the claims made regarding its functionality. The development of the proposed packer follows a modularised approach and hence a subsequent test follows the development of each module. The tests check for the general stability of the module and report the observations carried out. The observed behaviour of these modules gets utilised to form the report of the packer. The completion of the project and its modular testing phase leads to the assorting of the developed modules to make a complete and stable packer. Finally, the developed packer gets tested for the values of entropy it generates for different sections of the packed executable, the execution time compared to the regular executable executing without the packer, the success rate of automatic unpacking with the use of current unpackers, and the generation of correct disassembly by different static disassembly tools. These tests help in solidifying the claims of the proposal laid down in the packer specification and analysis of design choices subsection.

The testing of the instruction extractor includes testing for the extraction of the correct instruction from the ***.text*** section of the executable to be packed and also to analyse if it is successful in separating code and data. Section loader testing involves checking if the packer is successfully able to load the required libraries for the use by the packed program and link them correctly to it. The unpacker routine tests involve checking if the unpacker is correctly able to unpack the packed program in memory and link the data related to it, for the instructions to execute correctly. The packer stability test after adding code obfuscation techniques to it includes the checks performed to verify the correct working of the unpacker while producing garbage data for the static disassemblers and decompilers. The self-modification of code testing involves the validation of the correct execution of the program after performing in-memory alterations to the executable code of the unpacking and opcode generating function. The next section highlights the duration for these phases in the form of a Gantt chart to provide a better overview to the reader.

E. Gantt Chart

This subsection outlines the project development and testing plan described in the previous subsections in the form of a Gantt chart. The Gantt chart, however, does not have any separate demarcation for the writing of the report as it gets undertaken from the very first date until the very last date of the development and testing plan. This workflow ensures minimum wastage of time in the production of the report while not missing any crucial or relevant details at every stage of its update.

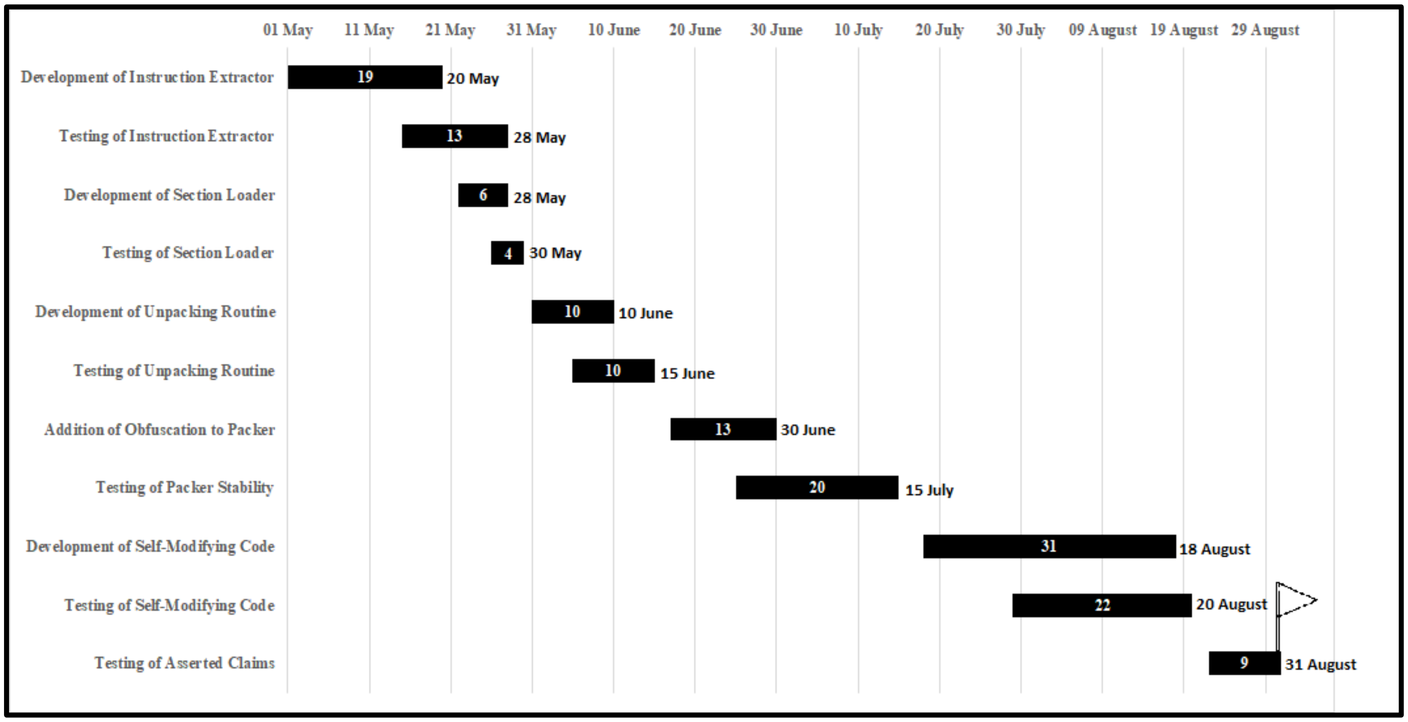


Fig. 3. Gantt Chart – Project Development and Testing Plan

REFERENCES

[1] W. Yan, Z. Zhang, and N. Ansari, “Revealing Packed Malware,” IEEE Secur. Privacy Mag., vol. 6, no. 5, pp. 65–69, Sep. 2008, doi: 10.1109/MSP.2008.126.

[2] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, “SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers,” in 2015 IEEE Symposium on Security and Privacy, San Jose, CA, May 2015, pp. 659–673, doi: 10.1109/SP.2015.46.

[3] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, “PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware,” in 2006 22nd Annual Computer Security Applications Conference (ACSAC’06), Miami Beach, FL, Dec. 2006, pp. 289–300, doi: 10.1109/ACSAC.2006.38.

[4] M. G. Kang, P. Poosankam, and H. Yin, “Renovo: a hidden code extractor for packed executables,” in Proceedings of the 2007 ACM workshop on Recurring malcode - WORM ’07, Alexandria, Virginia, USA, 2007, p. 46, doi: 10.1145/1314389.1314399.

[5] L. Martignoni, M. Christodorescu, and S. Jha, “OmniUnpack: Fast, Generic, and Safe Unpacking of Malware,” p. 10.

[6] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, “Eureka: A Framework for Enabling Static Malware Analysis,” in Computer Security - ESORICS 2008, vol. 5283, S. Jajodia and J. Lopez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 481–500.

[7] J. Calvet, F. L. Lévesque, J. M. Fernandez, E. Traourouder, F. Menet, and J.-Y. Marion, “WAVEATLAS: SURFING THROUGH THE LANDSCAPE OF CURRENT MALWARE PACKERS,” p. 7.

[8] C. Lim, Y. Syailendra Kotualubun, Suryadi, and K. Ramli, “Mal-Xtract: Hidden Code Extraction using Memory Analysis,” J. Phys.: Conf. Ser., vol. 801, p. 012058, Jan. 2017, doi: 10.1088/1742-6596/801/1/012058.

[9] C. Lim, Suryadi, K. Ramli, and Suhandi, “Mal-XT: Higher accuracy hidden-code extraction of packed binary executable,” IOP Conf. Ser.: Mater. Sci. Eng., vol. 453, p. 012001, Nov. 2018, doi: 10.1088/1757-899X/453/1/012001.

[10] C. Lim, Suryadi, K. Ramli, and Y. S. Kotualubun, “Mal-Flux: Rendering hidden code of packed binary executable,” Digital Investigation, vol. 28, pp. 83–95, Mar. 2019, doi: 10.1016/j.diin.2019.01.004.

[11] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, “Automatic Static Unpacking of Malware Binaries,” in 2009 16th Working Conference on Reverse Engineering, Lille, France, 2009, pp. 167–176, doi: 10.1109/WCRE.2009.24.

[12] S. Cesare and Y. Xiang, “Classification of Malware Using Structured Control Flow,” Parallel and Distributed Computing, vol. 107, p. 10, 2010.

[13] M. Bat-Erdene, T. Kim, H. Park, and H. Lee, “Packer Detection for Multi-Layer Executables Using Entropy Analysis,” Entropy, vol. 19, no. 3, p. 125, Mar. 2017, doi: 10.3390/e19030125.

[14] B. Anckaert, M. Madou, and K. De Bosschere, “A Model for Self-Modifying Code,” in Information Hiding, vol. 4437, J. L. Camenisch, C. S. Collberg, N. F. Johnson, and P. Sallee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 232–248.

[15] C. Linn and S. Debray, “Obfuscation of Executable Code to Improve Resistance to Static Disassembly £,” p. 10.

[16] S. Dolan, Stedolan.net, 2013. [Online]. Available: http://stedolan.net/research/mov.pdf. [Accessed: 06- Apr- 2020].

[17] C. Domas, "xoreaxeaxeax/movfuscator", GitHub, 2015. [Online]. Available: https://github.com/xoreaxeaxeax/movfuscator. [Accessed: 06- Apr- 2020].

[18] “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4,” p. 5038, 2019.

[19] A. Sotirov, "Tiny PE", Phreedom.org, 2006. [Online]. Available: http://www.phreedom.org/research/tinype/. [Accessed: 06- Apr- 2020].