Hindawi
Security and Communication Networks
Volume 2023, Article ID 8227751, 41 pages
https://doi.org/10.1155/2023/8227751

Review Article

Metamorphic Malware and Obfuscation: A Survey of Techniques, Variants, and Generation Kits

Kenneth Brezinski and Ken Ferens

Department of Electrical and Computer Engineering, University of Manitoba Winnipeg, Winnipeg, Canada

Correspondence should be addressed to Kenneth Brezinski; brezinkk@myumanitoba.ca

Received 15 September 2022; Revised 27 April 2023; Accepted 26 July 2023; Published 2 September 2023

Academic Editor: Konstantinos Rantos

The competing landscape between malware authors and security analysts is an ever-changing battlefield over who can innovate over the other. While security analysts are constantly updating their signatures of known malware, malware variants are changing their signature each time they infect a new host, leading to an endless game of cat and mouse. This survey looks at providing a thorough review of obfuscation and metamorphic techniques commonly used by malware authors. The main topics covered in this work are (1) to provide an overview of string-scanning techniques used by antivirus vendors and to explore the impact malware has had from a security and monetary perspective; (2) to provide an overview of the methods of obfuscation during disassembly, as well as methods of concealment using a combination of encryption and compression; (3) to provide a comprehensive list of the datasets we have available to us in malware research, including tools to obfuscate malware samples, and to finally (4) discuss the various ways Windows APIs are categorized and vectorized to identify malicious binaries, especially in the context of identifying obfuscated malware variants. This survey provides security practitioners a better understanding of the nature and makeup of the obfuscation employed by malware. It also provides a review of what are the main barriers to reverse-engineering malware for the purposes of uncovering their complexity and purpose.

1. Introduction

Digital resources and infrastructure have become some of the most crucial concerns in the field of cyber security. As we encourage a greater use of the Internet to delegate the tasks of everyday life, we expose ourselves and our information through potential exploitation by malicious actors. The biggest culprit is malware, a portmanteau for malicious software. Malware takes on many forms, but put simply, the ultimate goal of malware is to carry out a series of actions for nefarious purposes. Whether the end goal is espionage, disrupting services, or exploiting systems for financial gain, the costs associated with inaction are increasing every year as new malware variants are deployed on unsuspecting enterprises and victims. Every year several antivirus (AV) vendors publish their annual white papers regarding the current state of malware worldwide. From a research standpoint, researchers are concerned about three aspects of malware behavior: the ability for malware to disguise its own structure to avoid detection; modification and/or utilization of the host operating system (OS) resources; and the communication malware aims to establish externally [1] to so-called command and control servers (CnC). These aspects of malware behavior can be summarized as follows:

Obfuscation: Malware employs the use of various obfuscation techniques, such as packing and encryption, in order to avoid signature-based detection methods. Obfuscated malware also makes it cumbersome to disassemble and produce accurate control-flow graphs (CFG) when reverse engineering.

Resources: Malware will utilize various resources of the host operating system in order to carry out its predefined objectives. Malware will call several Windows application programming interfaces (APIs), make changes to the registry, read and write to the file system,

as well as create and spawn new daughter processes and threads.

Network: Malware will attempt to communicate with an outside command and control (CnC) server in order to relay information. Communication may be used to serve a greater botnet network and relay personal confidential details obtained from surveillance of the target operating system (OS) or used in detecting the presence of a sandbox environment in antiemulation and stealth malware.

The scope of malware worldwide is widespread and includes infections in both Macintosh and Windows OS, affecting businesses, governments, and individuals alike. A total of 20% of individuals have experienced a malware attack in one form or another, a 14% increase from 2018 [2]. Estimates obtained for 2019 identified 24 million Windows and 30 million Macintosh infections being recorded [3], with Kaspersky noting over 24 million unique malicious objects being detected in 2019 alone [4]. While infections recorded span several different types of OS, approximately 94% of malware developed is, in fact, Windows targeted [5, 6]. Malware takes on many shapes and sizes and includes archetypes such as Trojans, adware, spyware, viruses, worms, ransomware, rootkits, exploits, cryptojackers, and keyloggers. These all carry out some form of invasion, damage, or disabling of systems for the direct or indirect benefit of the malicious actor. More recently, the availability of free and open source software distributions has posed significant risks, as so-called "script kiddies," which are users who have little to no experience in writing software themselves, have made use of these tools for nefarious purposes. The readily available access to distributions such as Remnux and Kali Linux (Offensive Security, New York City, NY) has made it even easier for users to deploy various forms of reconnaissance and penetration testing tools with out-of-the box software. As natural language processing (NLP) tools become more sophisticated, chatbots such as ChatGPT can act as personal advisers in red-teaming and blue-teaming drills, which can also subsequently be used by black hats for their own vulnerability campaigns.

Businesses are some of the most susceptible recipients to malware attacks, as they are potential victims to ransomware attacks for monetary gain and experience service downtown due to denial of service (DOS) attacks. For example, in late 2019, the average downtime for a ransomware attack was 16.2 days and the average ransom payment was 81,116 USD, almost doubling from 41,198 USD seen earlier in 2019 [7]. The average cost of a data breach to a business was estimated at 3.8 million USD [8], and the average cost of a DOS attack was placed at 2 million USD [9]. The prevalence of malware in the business environment is evident, with 95% of organizations recording a malicious infection [10] and 81% having been affected by such an infection [11]. While total malware detections have seen a small increase of 1% year over year, the business sector has seen a 13% increase in 2019 [3, 12]. The top 10 malware variants which target business infrastructure saw triple digit increases in their number of infections between 2018 and 2019 [3]. Small businesses represent 43% of infected businesses reported, likely due to their inability to mitigate, flag, and respond to infections appropriately [7] and the fact that 37% of businesses spend less than 200,000 USD on Internet technology (IT) security and 78% do not have a formal incident response plan in place [13, 14]. Security experts encourage IT security personnel to adopt the 1-10-60 rule: threats are to be detected within the first minute, threats are to be investigated in 10 minutes, and an appropriate action must be taken within the first 60 minutes [15]. Businesses are prime targets for malware due to the financial motivation, with 71% of all breaches being financially motivated and 25% being motivated by espionage [7, 8]. Furthermore, North America is one of the leading regions where corporate ransomware is a pressing concern, with 68% of businesses having experienced attacks in the last year [16].

AV vendors are particularly interested in the emergence of new forms of malware because these represent unique instances of malware that have never been seen before and they pose a significant threat to security infrastructure. A report by FireEye noted over 100,000 unique malware signatures are being reported each day by AV vendors [10]. Zero-day attacks are of particular concern as they require AV vendors to develop signatures of these new malware instances, requiring significant domain-level knowledge and constant revision of their signature database. New and emerging threats are evident, with 60% of ransomware variants identified in the last 6 months of 2016 being developed in the last year [17]. Moreover, a small but mutable subset of malware variants, totaling only 50 malware families, were noted to make up 80% of all successful malware infections [10]. This propensity for malware infections to originate from a small family of malware instances is due to the polymorphism built into their development. Polymorphism allows for malware to change their signature upon each iteration of its propagation, leading to previously unseen threats and new instances of zero-day attacks [18–21]. As the stakes increase for both cybercriminals and businesses, so has the tools they develop to penetrate and mitigate threat vectors, respectively. The call for cyber security expertise has never been at its highest, with 62% of organizations planning on investing more in cyber security in 2020 [22]. The prevalence of polymorphic malware and its variants has expanded how we approach the field of cyber security for threat mitigation. Legacy methods, which classify new malware based on previously known signatures, are no longer effective in identifying polymorphic malware [23], lending credence to the development of a more adaptable, behavioral, and cognitive-based approach to how we detect malware [24]. The vast majority (93.6%) of malware observed today is polymorphic [25], and the necessary steps must be taken to ensure our instruction detection systems (IDS) and security information and event management (SIEM) systems are equipped to keep up with the ever-mutating nature of today's malware landscape.

This review will cover several aspects of metamorphic malware: starting from the limitations of current signature-based methods to the various obfuscation techniques employed by malware. This survey discusses the constantly

evolving threat characteristics of metamorphic malware, which provides the basis for building more sophisticated heuristic and analytically tools based on potential features sets. In addition, a broad discussion of metamorphic engines and antiarmoring techniques discusses the challenges researchers face in isolating malware variants in a controlled environment. We hope to improve the current understanding of metamorphic malware research by making the following core contributions in this work:

(i) Summarize the common obfuscation methods which, in turn, can be used to develop better heuristic techniques for feature engineering in machine learning pipelines.

(ii) Present the inner workings of a metamorphic engine and polymorphism more generally. Understanding how a malicious payload can persist in memory without ever be written to disk will allow researchers to find indicators of compression or encryption when a candidate binary is presented.

(iii) Outline the current metamorphic engines broadly available in the literature which can be used by researchers to obfuscate their own binaries to incorporate robustness into their own work.

Section 1.1 will cover basic signature techniques used by AV vendors including the most common scanning techniques and considerations for scanners. Section 2 builds on the limitations of these techniques by introducing malware obfuscation, which is the most commonly used routine used by metamorphic engines in its obfuscation stage. In Section 3, the idea of obfuscation is put into perspective with a deepdive into a metamorphic engine, which involves the ability of malware to unpack, obfuscate, compress, and encrypt its payload on the fly. Finally, Section 4 provides an overview of the most well-studied datasets used in malware research, with Section 4.2 covering popular metamorphic kits that can be used by researchers to create their own metamorphic binaries.

1.1. Signature Analysis and Creation. Signatures are used to help identify malicious code segments present, either existing as independent executables or attached to benign files known as benignware. It is imperative that AV vendors constantly update their signature databases in order to cross-reference known malicious binaries with files suspected of being malicious. Acting as unique fingerprints for malware, signatures are plagued with several fundamental issues. First, signatures are incapable of identifying emerging malware variants. In an environment where approximately 60% of new ransomware are never-before-seen variants according to the most recent estimates [3], this creates a significant shortfall in detection rates for new variants. In addition, when the vast majority of malware is polymorphic [25], signatures are sometimes not generalized to catch obfuscated instances of previously flagged malware.

The art of file scanning is in of itself a laborious process, requiring trade-offs between speed and specificity. Incorporating longer signatures provides a more specific identification of malware and malware families but is unable to catch the subtleties of minute changes [26]. Short signatures provide better coverage but results in more false positives [27, 28]. AV vendors therefore must come up with a series of rules to both generalize their signatures and improve their scanning efficiency. Some of the basic scanning strategies are shown in Table 1 and described in the following:

String scanning is the de facto standard for any string match scanning. The scanner is to look up the exact sequence of bytes in any offset.

Wildcards method allows for the use of wildcard variables. In the example shown in Table 1, the use of "??" acts as a placeholder for 2 bytes of any string, while %3 prompts the scanner to look for the subsequent byte sequence in any of the proceeding 3 byte positions. This is extremely effective for catching register swapping and instruction replacement obfuscations.

Mismatch method incorporates the idea of partial match of any given byte sequence. In the example provided in Table 1, if the scanner allows for up to 1 mismatch, as long as 2 of the 3 byte sequences are found, the scanner alerts to a match.

Generic method allows for the detection of malware families through the use of both wildcards and mismatch sequences. This method extracts the core malware artifacts of a malware family, thereby capturing any subtle alterations to the bytecode sequence that may arise in the future. For example, the Win95/Regswap virus uses similar opcodes between generations. Through a combination of wildcard string matching with mismatch, the entire Regswap generation can be flagged based on a few common signatures.

In addition to generating unique signatures as part of generating a greater signature database for malware, scanning files requires a dedicated strategy, and in some cases, dedicated hardware. For example, while a signature may be located in any one of the portable executable (PE) sections, such as *.idata*, it may also be located in the PE file header. In addition, if you wish to cross-reference thousands of malicious signatures with an incoming data stream using regex patterns, you would have to take advantage of intrapacket or interpacket scanning to process them effectively [29]. AV vendors utilize cheaper operations, such as checking the file length, before committing to the use of a more arduous task such as a checksum [28]. In practice, a signature can act as a representation for a series of bytes, a whole file, or certain sections. The ways in which AV vendors carry out simple scanning on a binary is described in the following sections.

1.1.1. Top-and-Tail Scanning. This mode of scanning used to extract signatures from the top and bottom of files. This is especially useful for viruses that append to the front or back of the targeted host program. Since the address of the main entry point of a program is in its header section,

Table 1: Summary of byte (strings) scanning techniques.

	Description	Example
String scanning	Searches for sequences of common strings characteristic of malware	AA, AB, AC, AD, AE, AF
Wildcards method	Searches for sequences of common strings while introducing wildcard variables	AA, ??, AC, %3, AE, AF
Mismatch method	Searches for sequences of common strings, regardless of position	AA, AC, AE, AF
Generic detection	Searches for common strings combinations typical of malware families	AA, ??, %2, AE, AF

Legend: (?) wildcard, (%) mismatch.

manipulation of this address to point to the appending malicious binary is possible [30]. As an example, the Polimer.512.A virus preappends itself at the front of the executable and shifts the original program content after itself. Alternatively, the Vienna virus is 1,881 bytes long and appends itself to the end of the host file.

1.1.2. Entry Point Scanning. This mode of scanning is used to extract signatures from the sequence at program entry points. Malware routinely alters program entry points as to avoid detection through rerouting of the execution flow to a decryptor stub which decrypts the original binary [31]. The Zmorph virus follows such behavior, whereby the decryptor aims to rebuild the instructions line by line by pushing the result into the stack memory. This can lead to "black hole" scenarios where useless operations are compiled early on in the process flow to burden the reverse engineering analysis.

In addition, an assembly encoder or an altered JUMP statement can be configured to run encoded information in a "code cave," as to not increase the file size of the binaries. This would normally impact the binary file header values, and any changes will alter relative/absolute offsets, so the pointers need to be changed accordingly. As previously mentioned, the Polimer.512.A virus appends itself to the infected program and in doing so is exactly 512 bytes long. This would raise flags and be easy to identify possible infected files due to the consistent file size differential.

Viruses such as the Win32/Simile is able to avoid changing the entry point of an infected file by altering call instructions which reference ExitProcess() to point to the virus code. This has the effect of not changing the entry point of the infected file. Other viruses such as W32/Bistro and W32/SMorph obfuscate their entry point [32]. SMorph is able to use existing API calls in the infected file to call to its own import address table containing references to API imports.

1.1.3. Integrity Checking. This mode of scanning can be an extremely powerful tool to detect manipulation of system files which should never change [27]. A checksum database can be used for reference when performing routine integrity checking of the system and files to detect any alterations [33, 34]. Common checksums include MD4, MD5, and CRC32. Checksums are routinely used on byte values suspected areas of a virus body, thereby reducing the number of total checksums required.

Alternatively, certain types of infections, such as companion infections, may attempt to mimic the name of an infected file and redirect the header section of an EXE which stores the address to the main entry point of a program to the start of the virus code [35]. The virus may also change the extension to COM as the Windows OS give a higher priority to COM over EXE extensions. In order to account for this, distributions such as McAfee's network security platform can assign a magic number to file types and will flag files whose extensions have been tampered with [36].

2. Obfuscation

This chapter will provide an overview of the common obfuscation techniques employed by malware. Examples of these techniques will be provided, along with some actual code snippets from popularized malware variants. Finally, a brief overview of encryption and compression is given, two very important techniques to familiarize yourself with. This chapter will focus on obfuscations made specifically via changes to the opcodes and operands, which serve as the CPU instruction set which specifies the data that are processed and how it is done. Opcode examples will include both Intel and ATT syntax, with the former being readily apparent as the source operand is always on the right side of the instruction and the destination on the left (e.g., mov eax, 1).

2.1. Dead-Code Insertion. Dead-code insertion, or sometimes referred to as garbage code insertion, is an obfuscation technique which inserts byte code sequences into a binary without affecting functionality [37–40]. This obfuscation relies on the fact that instructions can be added to code which do not perform any meaningful function, or in other scenarios, can carry out an instruction and perform the operation in reverse [41, 42]. An example of this type of obfuscation is shown in Table 2 where a series of nop instructions are used to pad the instructions. Typically, dead-code insertion is used to carry out one of three functions:

(1) Insertion of a pointless operation such as nop, mov eax, eax, add eax, 0, and eax, −1 or or eax, 0. In practice, these instructions do not change the content of CPU registers or memory as they are all semantically equivalent to nop; however, they may modify the status of the flag register in the CPU. These instructions also have different opcodes.

(2) Insertion of operations with the purpose of burdening the reverse engineering process by altering values in registries and then reversing the instruction. An example would be incrementing a registry add eax, 1 and then reversing the instruction by decrementing sub eax, 1. Other examples would be push and pop and inc and sub. This

Table 2: An example of dead code insertion using nop.

Before obfuscation	After obfuscation
	xor eax, eax
xor eax, eax	move eax, 0x2D
move eax, 0x2D	nop
mov ecx, 0xA	nop
	mov ecx, 0xA

form does change the values of the CPU registry but simply undoes the operation sometime afterwards.

(3) Insertion of dead code within branches of code that are never actually called, which may or may not make changes to variables in other branches of code which are never executed. An example would be a set of variables in Function A which are manipulated, but Function A is never executed because it is bypassed with a jmp statement.

Garbage code insertion is used successfully in the implementation of W95/Bistro, a later implementation of W32/Zperm, which utilizes a random block insertion engine which is placed directly after the virus entry point. Upon entering, this block of code millions of instructions is run, thereby overburdening the emulator before the virus instructions are even executed. Other popular examples of viruses utilizing garbage code insertion are W32/Evol and W32/Zmist. Zmist is notable for its use of the executable trash generator (ETG). W32/Evol in particular is able to utilize garbage code insertion to produce very different variants with different opcodes and string signatures, thereby evading signature scanning techniques as no sequence of bytes is similar between the two generations. An example of 3 variations of the same code is shown in Table 3.

The use of garbage code insertion techniques is useful in avoiding AV scanning for two reasons. First, the garbage code inserted is unique to each virus generation, thereby sidestepping previously seen AV signatures [44]. Secondly, garbage code from benignware can be inserted into malware to increase the false negative rate. In [45], the authors created binaries with approximately 30% of dead code along with 10% benign code and showed similar classification scores as benignware. In the work of [46], ranges of garbage code between 5 and 35% were used to determine their effectiveness at evading detection; with 10% being noted as being sufficient. In an earlier work [47], the authors combined various proportions of garbage code insertion with subroutine reordering to total 25 different combinations. Two different obfuscation engines, AVFUCKER and DSPLIT, also known as crypters, were used in [48] to produce obfuscated code with dead code insertion. Since there is a wide variety of permutations, from single nops to intermeshed garbage code blocks, upon which garbage code insertion can take form, string scanning is fairly ineffective against this form of obfuscation.

2.2. Registry Reassignment. Registry Reassignment, or sometimes referred to as Registry Renaming, is an obfuscation technique which swaps unused registers or memory variables with those currently used by the program [44]. In its simplest form, as demonstrated in Figure 1, registry reassignment can replace the eax registry with ebx, with no change in functionality.

Table 3: Three versions of the E32/Evol virus following obfuscation through garbage code insertion and encryption. Retrieved from [43].

	Opcode	After obfuscation
Version 1	C7060F000055	mov dword ptr [esi], 5500000Fh
	C746048BEC5151	mov dword ptr [esi + 0004], 5151EC8Bh
Version 2	BF0F000055	mov edi, 5500000Fh
	893E	mov [esi], edi
	5F	pop edi
	52	push edx
	B640	mov dh, 40
	BA8BEC5151	mov edx, 5151EC8Bh
	53	push ebx
	8BDA	mov ebx, edx
	895E04	mov [esi + 0004], ebx
Version 3	BB0F000055	mov ebx, 5500000Fh
	891E	mov [esi], ebx
	5B	pop ebx
	51	push ecx
	B9CB00C05F	mov ecx, 5FC000CBh
	81C1C0EB91F1	add ecx, F191EBC0h; ecx = 5151EC8Bh
	894E04	mov [esi + 0004], ecx

The downside to using registry reassignment is that string scanning techniques, such as wildcard or half-byte techniques, can be used to detect any possible combination of registry used. This in effect will provide a constant string between generations of registry reassignment, rendering them easily flagged by scanners. The virus W95/Regswap (hence the name) effectively made use of registry reassignment as demonstrated in Table 4.

In Table 5, the string signatures of version 1 and version 2 have a 60% similarity when it comes to their hexadecimal representation [49]. With the help of regex expressions, the accuracy is greatly increased with variations of a similar instruction set [50]. Along with garbage code insertion, these primary obfuscation techniques make it considerably harder to flag new variants of malware.

2.3. Instruction Substitution. The instruction-substitution technique introduces an additional layer of obfuscation on the existing techniques discussed. The power of instruction-substitution comes from the fact that there is a seemingly endless diversity to the substitutions you can introduce to an existing instruction framework. Table 6 demonstrates an example of a 2–4 instruction substitution (2 instructions are replaced with 4 to perform the same function) [51]. Another instruction substitution would be push eax; mov eax, ebx with push eax; push ebx; pop eax. Semantically, these are equivalent, but push, pop is in fact slower as it is quicker to direct registry write with mov. This exact substitution is utilized by the W95/Zmist virus, along with interchanging xor/sub and or/test instructions.

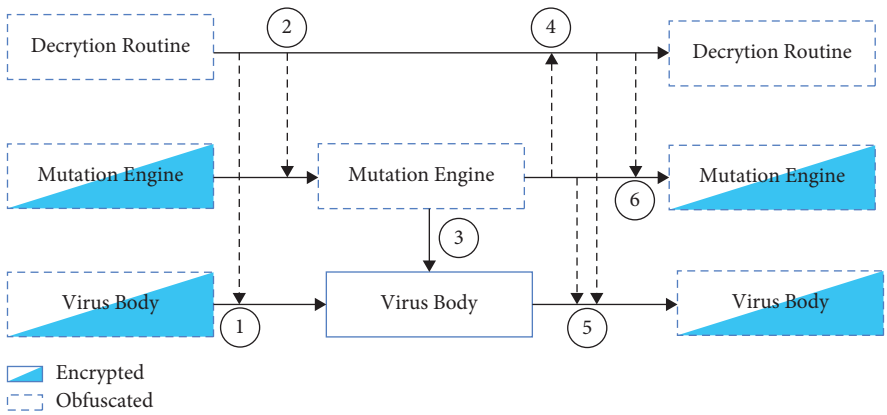

Figure 1: Graphical illustration for the decryption, obfuscation, and encryption carried out by a metamorphic mutation engine.

Table 4: An example of simple registry reassignment.

Before obfuscation	After obfuscation
mov eax, ecx	mov ebx, ecx
xor ebx, ebx	xor eax, eax
test eax, ebx	test ebx, eax

Table 5: An example of the Regswap virus. Adapted from [49].

	Opcode	After obfuscation
Version 1	5A	pop edx
	BF04000000	mov edi, 0004h
	8BF5	mov esi, ebp
	B80C000000	move eax, 00Ch
	81C288000000	add edx, 0088h
	8B1A	mov ebx, [edx]
	899C8618110000	move [esi + eax ∗ 4 + 00001118], ebx
Version 2	58	pop eax
	BB04000000	move ebx, 0004h
	8BD5	mov edx, ebp
	BF0C000000	move edi, 000Ch
	81C088000000	add eax, 0088h
	8B30	mov esi, [eax]
	89B4BA1811000	move [edx + edi ∗ 4 + 00001118], esi

Table 6: A simple example of instruction-substitution.

Before obfuscation	After obfuscation
	add eax, 01H
add eax, 05H	add eax, 05H
mov ebx eax	push ebx
	pop eax

Instruction-substitution is utilized very effectively in several high-profile viruses such as Evol, MetaPHOR, Zperm, and Avron. Since instructions-substitutions produce different opcode representations, this renders opcode frequency and accompanying n-gram techniques effectively useless. Researchers have attempted to draw from the basic set of fundamental operations in order to track the malware's original intentions. In [52], a clue set was established for the Evol virus in which all rewritten instructions were based upon. This approach was found to be very effective at characterizing the metamorphic engine Evol uses. A similar approach was taken by [53] where the complex instructions the virus would create were transformed back into their simple representations using their similar semantics. In Table 7, two versions of the W95/Bistro virus are shown, using different instruction substitutions in each generation. Similar to registry reassignment, the generations contain similar string signatures, making them susceptible to wildcard and half-byte scanning techniques. While this manuscript is focused on obfuscators based on the Intel x86 instruction set, compile-time instruction set obfuscators can also create semantically similar rule sets for basic operations in other instruction sets [54, 55].

2.4. Code Transposition. Code transposition, or sometimes called instruction permutation, is an obfuscation technique which utilizes conditional or unconditional jmp statements to reorder single or blocks of instructions [18]. Since jmp instructions can theoretically be used for every line of instruction, the total number of permutations $m!$ is proportional to the number of lines rearranged m [44]. Code transposition carries out a very similar function as subroutine reordering with the exception that there is a change in the process flow; therefore, they will be discussed together. Subroutine reordering, also known as block reordering, is an obfuscation technique that reorders the process flow by rearranging blocks of code that have independent subroutines [56]. If a program were to be categorized into n number of subroutines, then $n!$ permutations of subroutines are available for rearrangement [40, 50, 57, 58]. A simple program with 10 subroutines would therefore be able to produce over 3.6 million possible iterations. Subroutines require that the instructions' set are independent of one another, allowing them to be reordered without having an impact on functionality. In Table 8, an example of a set of instructions exhibiting multiple forms of obfuscation is shown. In the example code transposition, subroutine reordering, garbage code insertion, and instruction-substitution are all used.

Table 7: Instruction replacement used by the Win95/Bistro virus. Adapted from [49].

	Opcode	After obfuscation
Version 1	55	push ebp
	8BEC	mov ebp, esp
	8B7608	mov esi, dword ptr [ebp + 08]
	85F6	test esi, esi
	743B	je 401045
	8B7E0C	mov edi, dword ptr [ebp + 0c]
	09FF	or edi, edi
	7434	je 401045
	31D2	xor edx, edx
Version 2	55	push ebp
	54	push esp
	5D	pop ebp
	8B7608	mov esi, dword ptr [ebp + 08]
	09F6	or esi, esi
	743B	je 401045
	8B7E0C	mov edi, dword ptr [ebp + 0c]
	85FF	test edi, edi
	7434	je 401045
	28D2	sub edx, edx

Table 8: An example of code reordering and code transposition in combination with other obfuscation techniques.

Before obfuscation	After obfuscation
	mov ebx, 10
	jmp F1
	jnk
mov eax, ecx	F2: push edx; jnk
mov ebx, 10	pop ecx
Mul ebx	jmp F3
add eax, 5	F1: mul, ebx
mov ecx, eax	add ecx, 1; jnk
	add ecx, 5
	jmp F2
	F3: mul ebx

Several jmp statements are employed to permute blocks of instructions which can be run independently from each other. Instruction-substitution is used to add more sophisticated instructions based on the simple instruction set add eax 5; mov ecx, eax. jnk insertions are used to add complexity to the existing code, as well as added following the jmp F1 statement where it is never actually executed. This jnk could include code from benignware that would normally fail to compile if it were embedded within the existing obfuscated framework but may confuse scanning techniques nonetheless. Table 8 also displays another form of obfuscation called subroutine outlining [32]. This obfuscation explicitly turns instruction blocks into subroutines and uses the call instruction to perform an unconditional jump to the location indicated by the label operand. Subroutine inlining would carry out the reverse: where subroutines would be unraveled and placed in order to preserve the process flow. Unlike simple jmp instructions, call preserves the locations to return to when the subroutine is completed.

This sophisticated form of obfuscation is used by the W95/Zperm and W32/Ghost viruses, with the former employing the use of the real permutation engine to perform subroutine reordering. Zperm divides the code into frames which are independent subroutines, which are then repositioned randomly and connected using branch instructions to preserve process flow. When Zperm initializes, it allocates a buffer sized at 64 Kb filled with zeros and then fills it with obfuscated code and randomly positioned jmp statements [43]. This means that a constant body is never generated between generations and is never present in memory. Similar to Table 8, garbage code is inserted between frames to fool string detection similar to the Zmist virus. W95/Zmist also inserts jmp instructions after every instruction, making it the perfect shield to heuristic detection. In [39], 30% subroutine reordering was used to sidestep a developed similarity metric that compared benignware to malware based on the similarity of their transpositions. From a security analysis standpoint, it is extremely difficult to know when the virus begins when it is embedded within existing code and is encrypted. Partial emulation is one avenue whereby code can be reconstructed and then used to completely decrypt the virus. But when and how to decrypt during emulation is still a laborious process in of itself.

3. Encryption, Compression, and Metamorphism

Metamorphism, and more generally obfuscation techniques, makes up the backbone for most new and emerging malicious threats we see today. As the signature-based scanning techniques improved for AV vendors, so did the levels of obfuscation employed by malicious actors to thwart said techniques [49, 59]. Along with obfuscation came various forms of armoring, stealth-behavior and antiemulation tactics, which made the job of a security researcher that much more burdensome.

To understand how mutation came to be, it is worth mentioning the earliest forms of obfuscation and how they came into existence. Viruses make use of entry point obscuration (EPO) in order to avoid any consistency in the execution order of the virus code in relation to the infected file. As shown in Figure 2, the file header would point to an address that would execute virus code, which would then point back to the host file so that the virus execution would do so unknowingly.

The CASCADE virus in 1986 became one of the first known viruses to implement encryption, thereby requiring a separate decryption routine to carry out decryption and push the instructions into memory for execution. Since the form of encryption would become apparent as the virus propagated, the decryptor routine itself would have to be mutated, leading to the establishment of the first series of oligomorphic viruses.

3.1. Oligomorphism. Oligomorphism began as a reaction to the signature-based scanning techniques widely utilized for flagging possible virus infections. With the help of scanning

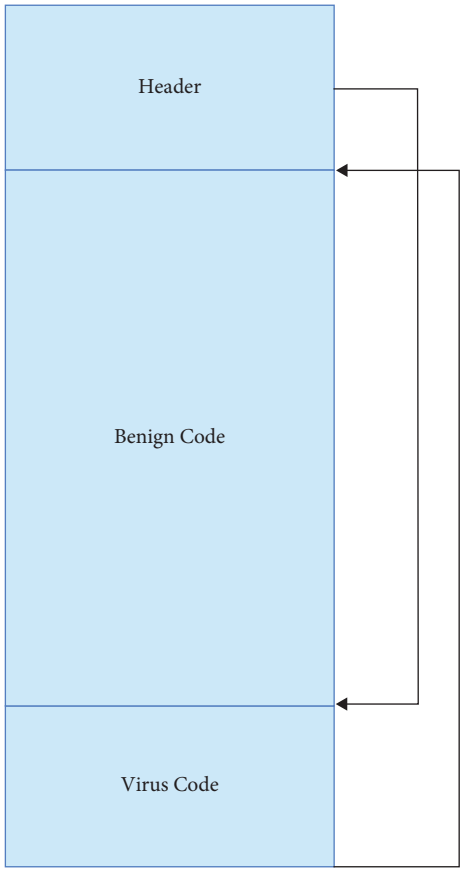

Figure 2: Illustration of an appending virus that latches onto the end of a benign file.

techniques such as wildcard and mismatch, a greater swath of possible infections could be characterized by a few unique signatures. Furthermore, since virus code would either append or preappend onto an existing file, top-and-tail scanning was an effective tool for extracting signatures from certain select sections of a file. Emulators could also be utilized to uncover the decryption routine used in the encryption, meaning that the decryption routine itself had to be altered in some form or another. Emulators wait as the virus is decrypted one instruction at a time and as it rebuilds itself by pushing the stack into memory. Once control is sent to the stack memory, the emulator monitors the stack, and the code can be dumped. Oligomorphic malware were the start of a new breed of malware which would involve obfuscation of the routine itself, meaning viruses were unique among their generation. The first oligomorphic virus was the whale DOS virus first identified in 1990. In Figure 3(a), an obfuscated, encrypted decryption routine is used to carry out decryption of the virus body and to avoid detection.

However, a major limitation to oligomorphism is that the loop of possible decryptors is finite. For example, the W95/Memorial virus had exactly 96 different decryptors to choose from. Once an oligomorphic generator is exhausted, the entirety of its possible generational variance is also exhausted and understood. The natural extension to this problem is to introduce obfuscation into the decryptor routine itself, leading to an infinite number of possible decryption routines [60]. This led to the first generation of polymorphic viruses such as 1260, and popularized generators such as Phalcon/Skism mass-produced code generator (PS-MPC) and virus creation lab (VCL), which are still used to this day.

3.2. Polymorphism. Polymorphic malware was seen as a complete package: complete with a compiler that could decrypt and obfuscate then recompile everything back together. The unencrypted virus body would create a new mutated decryptor using a random encryption algorithm and then allow the decryptor to encrypt itself before linking both sections back together. However, the core problem of emulation remains: the virus code section would be decrypted into memory and be able to be detected and flagged by security researchers. It was also the case that prior generations of obfuscators suffered from several limitations [61]:

(1) Constant size of virus code between generations (Polimer.512.A or Vienna viruses)

(2) Appending or preappending to the infected host file meant signature scanning could target these sections exclusively

(3) Similar virus code segments between generations mean the virus is subject to entropy analysis

In order to build on some of these deficiencies, the introduction of the metamorphic engine came to be.

3.3. Metamorphism. The introduction of metamorphic viruses introduced the idea for the first time that no two generations of viruses can have similar signatures, as no constant body is present like with polymorphic malware [43]. In Figure 3(b), an example of a metamorphic virus is shown. Unlike polymorphism, the virus code is obfuscated, meaning that the entirety of the virus is present in an obfuscated state. This introduces the fundamental issue since "metamorphics are body-polymorphics" [62] and as a result have no constant body and they reinforce the notion that anomaly-based detection is NP-complete [63, 64]. The first metamorphic viruses were W95/Regswap in 1998 [65] followed by W32/Ghost identified in 2000 [66]. W32/Ghost contained 10 submodules, so over 3.6 million possible variations were possible with subroutine reordering.

In light of the graphic shown in Figure 3(b), the separation between the decryptor and the virus body is no longer possible and the level of obfuscation means that encryption is no longer needed. Furthermore, as is typically the case, the decryption routine is scattered in the benign code. The executed code in the virus body mutates entirely along with the decryptor, and it does not need to unpack to create a new constant virus body like polymorphics [50]. One of the most utilized and effective metamorphic generators is W32/NGVCK created in 2001. Metamorphic viruses have a sophisticated mutation engine that contains many subprocesses. These will be discussed in the following section.

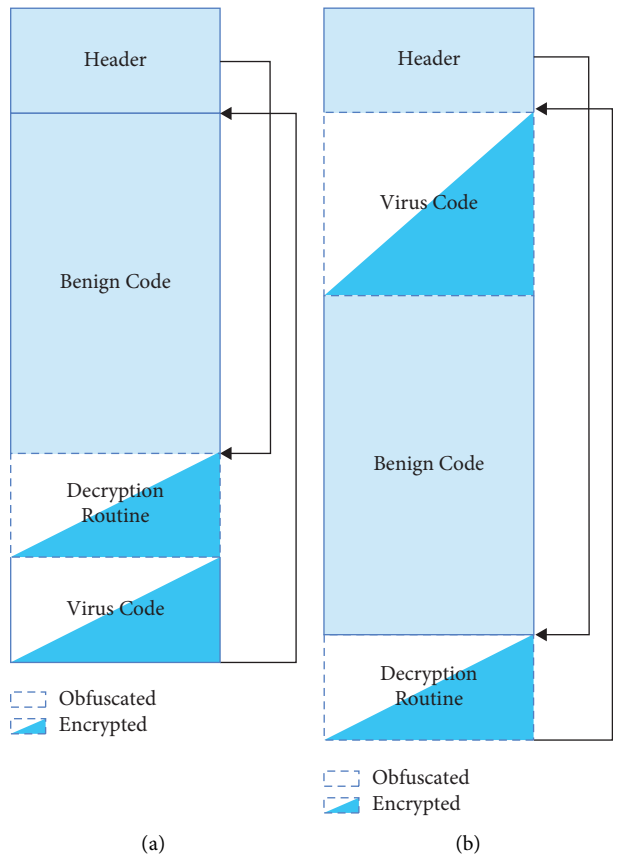

Figure 3: An illustration showing the variation in positioning and level of obfuscation found in (a) oligomorphic and (b) metamorphic malware.

3.4. Metamorphic Engine. A metamorphic engine is responsible for the obfuscation and reconstruction of the binary so that the file can remain operational. In Figure 4, an illustration of a complete metamorphic engine is shown. Some of the key components of the metamorphic engine are described as follows [65, 67]:

Disassembler is responsible for turning the source code into assembly instructions. This creates an intermediate form that is independent of the CPU architecture for future adoption with different OS and CPU architectures [43]. Within the disassembler a code analyzer provides info for a code transformer module that gathers information related to control flow, subroutines, variables, and registers.

Shrinker eliminates much of the garbage code produced from previous generations and mainly eliminates garbage and other nonsequential code that is produced from obfuscation. This step also carries out code shrinking, a form of code-substitution that will turn previous 1 to 2 or 1 to 3 instruction substitutions back to their semantically similar primitive equivalents [68].

Permutor carries out much of the obfuscation using permutations of subroutines, many times in a randomized fashion. Insertion of jmp instructions is also common to divert control flow.

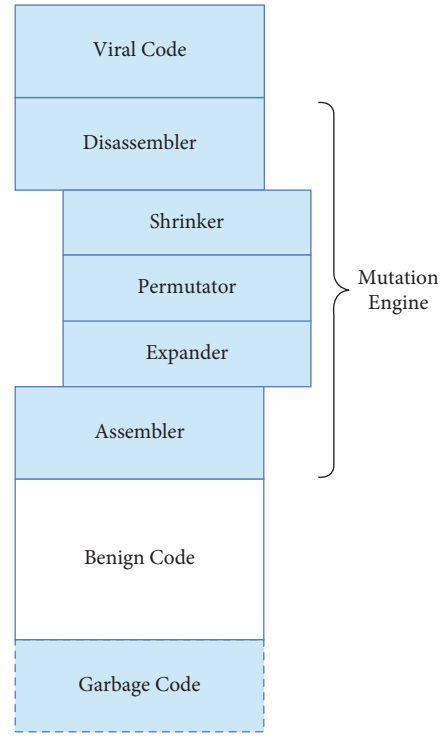

Figure 4: Overview of the major components of a metamorphic engine.

Expander performs instruction-substitution to convert instructions into another equivalent instruction set. In addition, registries are reassigned and variables are reselected according to the fixed probabilities using substitution tables [65, 69]. Garbage and other do-nothing codes are added, and functions are inlined/outlined [70, 71] Both the permutor and expander steps are quite sophisticated in the metamorphic W32/Etap and W32/Zmist viruses [60].

Assembler restructures the control flow and converts the assembly code back into machine binary code where it can become operational again.

Virus code contains the core instruction set that will execute on all new generations of the virus. It also contains the instructions that coordinate with the mutation engine and other components.

The mutation engine does not have to operate at the assembly and the source code level but can also operate at an intermediate representation (IR) bytecode level [70]. In [72, 73], morphing techniques are seen as deterministic automata, whereby transitions following formal grammar are made to symbols and new mutations are produced. In [69], a template is used which illustrates how simple representations of formal grammar can produce several possible mutations. The depiction shown in Figure 4 includes all the core components with the exception of a decryption routine. A metamorphic engine with the addition of a decryption routine is shown in Figure 1 and follows a sequence of steps to decrypt, obfuscate, and link everything back together. The steps are as follows in order:

(1) First, the decryption routine decrypts the virus body and executes an instance of it.

(2) The decryption routine then decrypts the mutation engine and executes it.

(3) The shrinker component of the mutation engine goes to work to deobfuscate the virus body.

(4) Obfuscation takes place by introducing a new and unique decryption routine using the various techniques discussed in Section 2.

(5) The virus body is then obfuscated by the mutation engine to produce a unique generation using the various techniques discussed in Section 2. The virus body is then encrypted using a unique algorithm, a static key or a host specified temporary key. More is given on this in the following section.

(6) Finally, the mutation engine is encrypted.

Once all three components are reobfuscated to seemingly new binaries, with the mutation engine and virus body decrypted, the virus relinks its components back up and can execute on a new host by decrypting its payload through it newly obfuscated decryption routine.

The authors in [57] provide a detailed summary of the production and considerations for creating a metamorphic generator, as well as in [74] for creating a metamorphic worm. One of the more sophisticated metamorphic viruses is W32/Simile, also known as MetaPHOR or Etap. The author, "Mental Driller," referred to the expansion, contraction, and permutation of instructions as the "Accordion Model" [61, 67] based on the changing form that garbage code takes when it becomes obfuscated. The Simile virus was also unique, and in that, 90% of the virus code was dedicated to the metamorphic engine itself, with the decryptor being placed at the end of the code section and the virus body being partitioned elsewhere [43, 52].

3.5. Encryption. While encryption was briefly touched upon at the beginning of Section 3, obfuscation engines make use of a variety of encryption techniques to avoid detection [49]. The earliest form of encryption was carried out by the CASCADE virus on DOS [40] and did so using a simple xor (see Figure 5).

The cascade virus, first identified in the early 1900s, was shown to increase the file size of infected files by 1701 and 1704 bytes and mainly comprised its encryption loop and main body. The virus uses a technique called "cascading" to conceal its presence. When the infected files are executed, the virus code is executed first, causing the virus to infect more files and directories. This creates a cascading effect, making it difficult for antivirus programs to detect and remove the virus [75]. The decryption routine in Figure 5 is fairly simple: the stack pointer, sp, acts as the key and the si register is used to keep track of which position of the virus body to point to. As the decryption process is carried out, both the si and sp counter increment and decrement by one, respectively, until sp returns to 0; otherwise, it will jump using jnz. For example, applying a simple xor operation to each byte using an 8-bit value as the encryption key will produce the encrypted text. The string 2D03 002E when xor'd with the key 0xFF will produce D2FC FFD1. Doing so in reverse with the same key will produce the original text, thereby performing encryption and decryption with only one key.

Conventional decryption relies on the virus' own decryptor loop to decrypt the virus body. It did not take long for malicious actors to rely on multiple decryptors instead of one, such as the DOS/whale virus in 1990, which utilized dozens of different decryptors and chose one randomly each infection. It may also be the case that rather than the encryption being performed serially, decryption can be performed in a random fashion, as is the case for W32/MetaPHOR which does so seemingly randomly, with each instruction only being decrypted once. In malware deployments, the use of a crypter is typically used, which carries out encryption for antianalysis and obfuscation purposes. A crypter contains a stub which carries out the decryption and does so while generating a new payload and key with each new generation [48, 76]. All of this occurs in memory, and nothing is written to disk. Decryption can take place in the stack, but then the key to it is not writable, as opposed to allocating to memory which is easily flagged by emulations that are monitoring memory. On Intel x86 platforms, 24 bytes or more of modified memory is characteristic of a decryption routine [28]. Once the stub passes

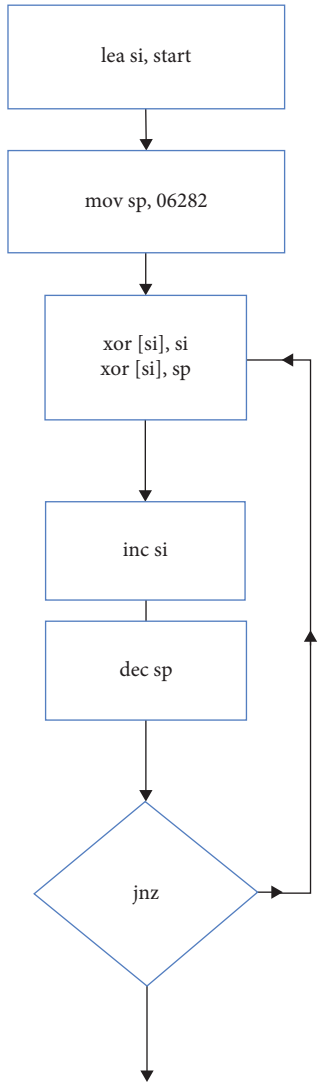

Figure 5: Simple xor decryptor which decrypts byte by byte using an increment counter and a jump not zero (jnz) loop.

control to the virus body after decryption, a new encryption key is created and all executables and .text sections are encrypted with the new key. Depending on the file type, a TEA cipher can be used for EXE and RC4 for DLLs as is the case for HackedTeam's core-packer [77]. The key is then stored in the decryptor stub or elsewhere.

Basic encryption can be performed as mentioned previously with a single decryptor key (see Figure 6), using 1 to 1 byte to byte mapping, with zero operand using inc or neg, or reversible instructions such as add or xor. Alternatively, sliding key encryption makes use of the starting key which updates as it progresses and may even utilize the characters most recently encrypted (see Figure 6(b)) or based on an algorithm, as shown in Figure 6(c). Flow encryption determines a key stream in advance equal to the size of the encrypted text and then encrypts the body instruction by instruction. Key generation can also be varied amongst decryptor routines, where a key(s) can be located in the decryptor stub itself, hidden among the virus body, generated uniquely from the host system, or alternatively, randomly generated and not stored at all.

The sources for the encryption key can vary but can either be hardcoded in one form or another or obtained through the host. In the case of variable key generation, the decryptor can develop the encryption key based on its own function calls. Alternatively, environmental key generation does not involve any descriptors from the viral payload or stub itself, but rather, retrieves them from the infected host. One example of environmental key generation is the use of a trusted platform module (TPM) chip, which is a hardware component built into many modern computers and devices [78]. The TPM can generate unique encryption keys that are tied to specific physical attributes of the device, such as the device's BIOS, firmware, or other hardware components. This makes it much more difficult for an attacker to access the key and decrypt the protected data even if they are able to physically access the device. In the case of the RDA.Fighter virus family, the virus checks the BIOS address at FFFF:000E0, and if it returns advanced technology (AT), as in AT-class computer, the time stamp is retrieved from the CMOS buffer; otherwise, it is retrieved from the system clock. The timestamp is then used to create a 16-bit number that is used to decrypt the next code section using a mirror table lookup as a mask. In addition to time, the current date, timer tick, host filename, and even the hard disk serial number can act as sources for developing the encryption key. As a form of armoring, the key can be stored on a distant web server, and outside of a typical host environment, such as in virtualization or emulation, the virus can disable itself and fail to run.

Decryptions and decryption loops are not limited to a single loop, or to a single key. For example, the RDA.Fighter virus family utilizes 16 layers of decryption and does so in a backward fashion, making it a laborious process to automate the disassembling process [28]. Multiple layers of encryption are also utilized by the W32/Harrier and Bradley viruses [79]. To avoid all form of local or external storage of the key, a random decryption algorithm (RDA) can be used to brute force the key. The key can be any generated word value, and the decoding method will check the checksum following the decoding procedure to identify when it has successfully found the key. In the RDA.Fighter family, RDA is used as secondary form of encryption on top of environmental key generation.

3.6. Compression. Compression represents an additional level of obfuscation on top of a possible decryption routine and other forms of obfuscation. A packer is defined as a utility which enacts some form of compression to the executable either to reduce files size to avoid entropy analysis or introduce a layer of obfuscation to the PE header. It has been estimated that 80% of all malware uses some form of packer [80], as well as 90% of all worms [43]. Two of the most popular packers are Ultimate Packer for eXecutables (UPX (https://upx.github.io/)) and ASPACK (https://www.aspack.com/). In addition to significant compression ratios and great performance, these packers work for a variety of executable formats with no memory overhead due to in-place decompression.

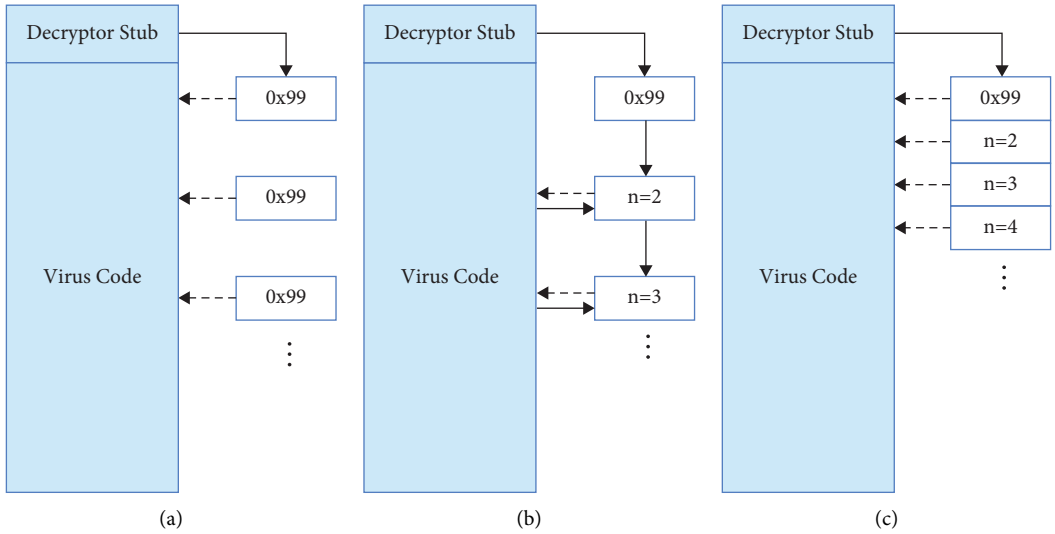

Figure 6: Illustration of different encryption archetypes, where (a) key is reused for each encrypted block; (b) encrypted block is used as nonce for next encrypted block; and (c) stream cipher is used to encrypt each block.

Packers are ultimately tasked with compressing executables with decompressed code and a compressed payload. Packers compress the code to avoid reverse engineering and bypass firewalls. Malware makes use of packers by initially converting an *Image Section* (see Figure 7(a)) into a *Packed Section* and *Unpacking Section* (see Figure 7(b)). The *Unpacking Section* is then set to be the initial point of entry once the file is executed. Upon execution, the packed section is decompressed to become the *Unpacked Section* (Figure 7(c)) and is executed on virtual memory [81]. One of the more devious uses of packers in malware analysis is that the original PE header is hidden as the visible import functions are those utilized by the packer itself. Since packers such as UPX, ASProtect, PECompact, and Themida are widely used for nonnefarious purposes as well, there is no sure indication that the file is malicious based on the import functions [82–84].

One of the more comprehensive tools for the detection of malicious packers is the use of entropy analysis [1]. In the work of [85], 28 different packers were used to classify a control flow graph as an image representation through the use of a convolutional neural network (CNN). The work of [86] used CNNs for a similar purpose, but was used to categorized 9300 malware variants into 25 malware families simply based on the malware binary. These techniques have the advantage of allowing the neural network to learn which PE sections are important in identifying maliciousness; and in doing so it uses an advanced form of entropy analysis which can identify malware family usage of packers, encryption and garbage code obfuscation [86]. When compression is coupled with encryption, as is the case with so-called *Protectors*, the resulting binary has high entropy levels, making it susceptible to classification. In [58], a file segmentation method that utilized entropy with wavelet analysis was used to classify metamorphic malware based on edit distance between file segments. This motivation was derived from the earlier work of [87] that established that the homogeneity of each malware's binary section is characteristic of the complexity of its data order. Along with this insight, polymorphic malwares are able to be identified using these techniques, albeit with a high rate of false positives [87].

In Figure 8, a historic summary is provided, which is complete with major milestones in obfuscation and new malware deployments.

4. Metamorphic Datasets, Generation Kits, and Armoring

While metamorphic malware has grown in sophistication, so has the tools we have as available as researchers to thwart their actions. One of such tools and resources is the use of publicly available datasets, such as DARPA99, a popularized dataset released to improve intrusion detection systems. Datasets encourage the development of classification tools by leaving the details for collecting representative samples in a controlled environment and at scale to others. Secondly, datasets also provide a baseline in which to compare competing algorithms, usually with the aim of increasing true positive rates and decreasing false positives. One of the downsides is that these datasets are typically outdated and are not representative of new and emerging threats. If researchers make raw malicious binaries available, as is the case with the SOREL dataset [88], they cannot do the same for benign binaries due to issues with copyright. One workaround used in SOREL is to dump the entire metadata of the binary and use that metadata dump to create features for a model to learn from. This section will touch on some of the more useful malware datasets used historically and then transition into covering some aspects of malware generation kits and antiarmoring behavior.

4.1. Malware Datasets. The DARPA dataset was created in 1998 and contains 7 weeks of raw TCP/IP dumps of a simulated attack scenario to an Air-Force base. The dataset contains both host and network files. The KDD99 was created based off

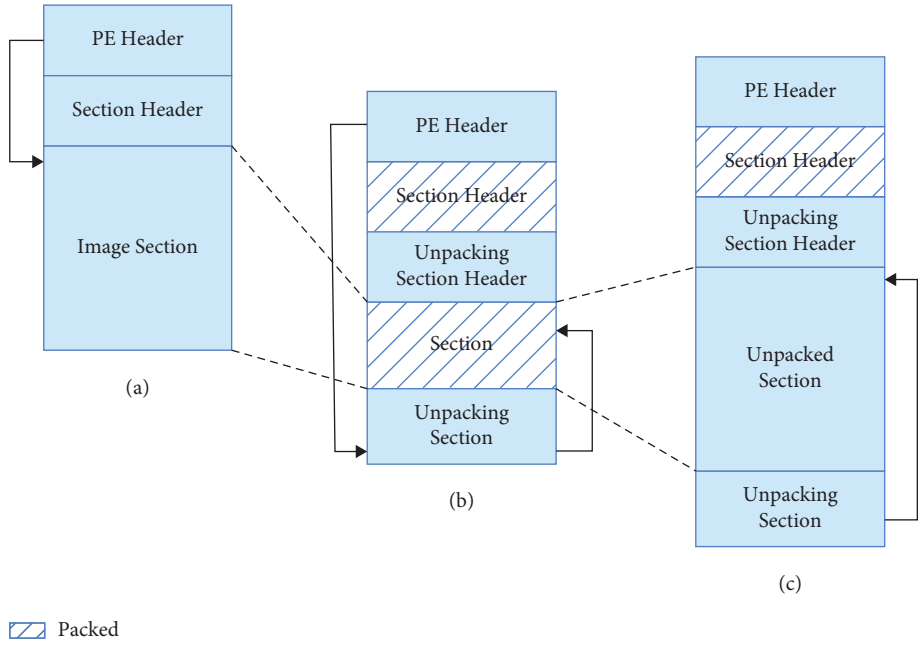

Figure 7: Overview of the main steps in a packer. Adopted from [81].

the DARPA dataset [89], with a reduced size and a total of 24 attack types and an additional 14 existing solely in the test dataset [90, 91]. Based on the observations of [91], KDD99 was the most widely used dataset in IDS research between the years 2010 and 2015. Several issues arose with the use of KDD99, namely, the time-to-live (TTL) values for benign and malicious packets were different [92, 93], and the data rates were not characteristic of real-world networks [94]. Many of these issues were exemplified in the critique carried out by [93], leading to a need to provide much needed modifications to the existing dataset. In addition, since the size of the KDD99 datasets was large for many trainable models and the dataset contained duplicates of attacks such as DOS, the dataset was further reduced to become its most recent version, NSL-KDD [93]. Another dataset containing network traffic is the UNSW-NB 15. The dataset was created by the IXIA PerfectStorm tool at the Cyber Range Lab at the Australian Center for Cyber Security [95]. A TCP Dump tool is used to capture 100 GB of raw traffic, with a total of 49 features generated using a set of tools and algorithms. Other lesser known network datasets include CAIDA [96] and ISCX 2012 [97] for network intrusion detection and CICIDS2017 [98]. The CICIDS2017 dataset is unique, and in that, the authors included behavior for Windows (XP, 7, 8, and 10), macOS, iOS as well as Linux operating systems, encompassing attacks from Botnets, DoS, DDos, Brute Force FTP, Brute Force SSH, Heartbleed, Web Attack, and Infiltration [98]. For a thorough summary of network-based datasets, the authors refer to the review carried out by [99].

Several datasets have been used to represent the content of the malware binary, versus relying on network activity. One of the more utilized datasets is the Microsoft Malware Classification Challenge dataset, which becomes popularized in a Kaggle competition back in 2015. The raw data of a virus' binary are represented in hexadecimal, with a compilation of metadata retrieved using the IDA disassembler tool. Binary representations of malware binary have also become popularized as a dataset in image analysis, with the Malimg dataset [100] having the greatest impact in recent years [101–109]. Other alternatives include the Malicia dataset [110] which contains 11,668 malicious binaries from 54 families retrieved from 500 drive-by downloads over 11 months. However, the project was ultimately discontinued in 2016. The Malsign dataset [111] contains 142,000 signed malware and potential unwanted products (PUP) binaries obtained from 2012 to 2015 for the Windows platform [112].

Mobile and internet-of-things (IoT) security plays a unique but important role in malware security, as these devices make up a larger proportion than ever in how we connect with others and exchange information. The Drebin dataset [113, 114] is one of the most used datasets in mobile security, with 5500+ malware being included in the dataset belonging to 20 families, collected from 2010 to 2012. The android adware and general malware dataset (AAGM) [115, 116] includes network activity of 1900 adware, general malware, and benignware running on android smartphones. The IoTID20 [117] is a more recent dataset used to simulate network attack retrieved from two smart home devices. The dataset consists of 42 pcap files encompassing simulated attacks produced from Nmap and from the Mirai botnet [118, 119].

Several datasets include features extracted directly from PE files, and this includes the ClaMP and EMBER dataset. ClaMP [120] includes features from the DOS header, file header and optional header of PE files. The integrated dataset includes 68 features:28 features are from the raw dataset, 26 features are Boolean (file and optional header), and 14 are derived features. A second version of the dataset exists which consists of 56 features. Finally, the largest dataset by far is the Ember dataset [121] with a total of 1.1 million binary files.

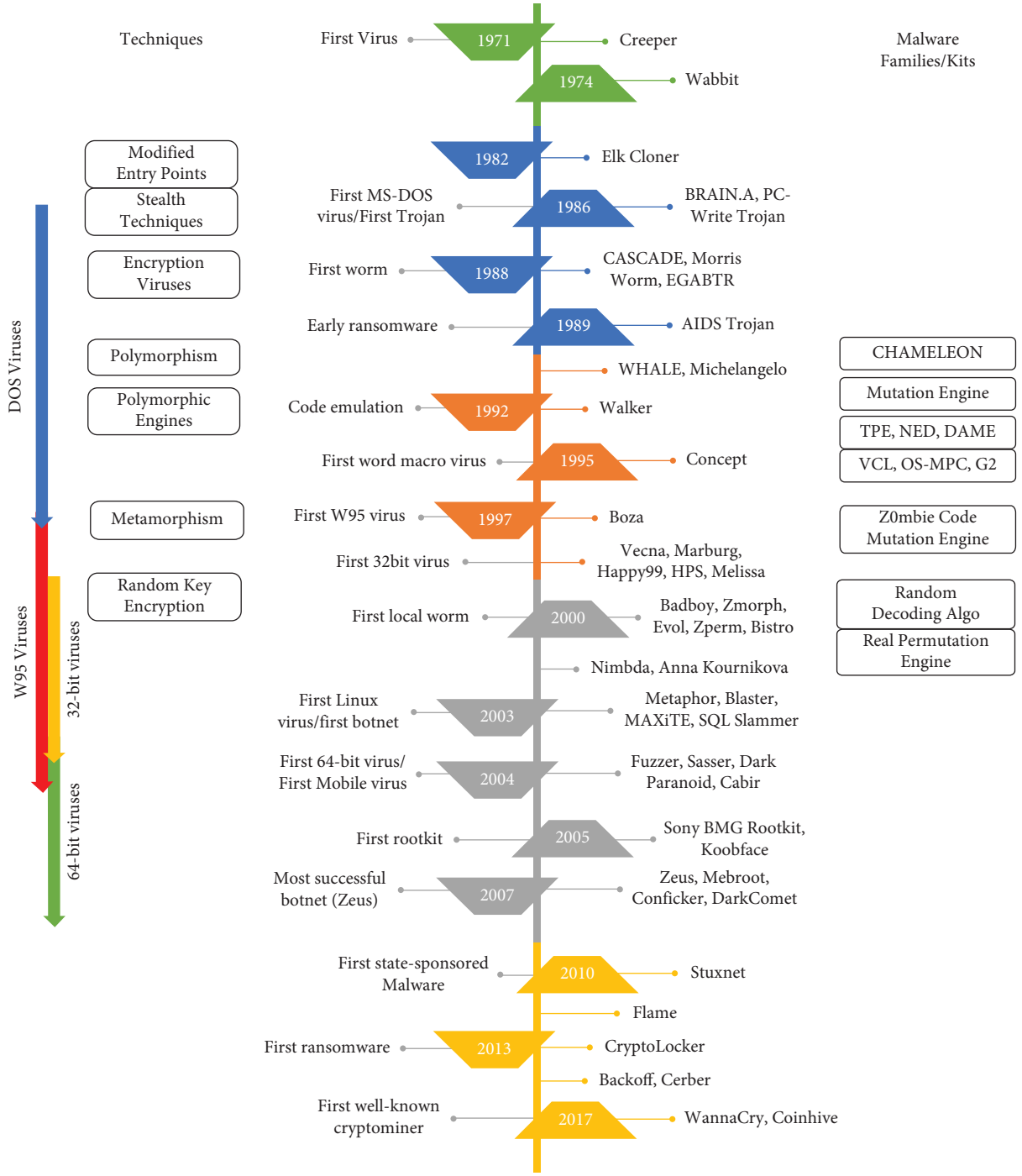

Figure 8: Timeline of major malware variants, techniques, and mutation engines.

The authors in [122] include additional tools to extract features from the PE files to further encourage the use of the dataset to train benchmark problems. The Ember dataset was the larges of such datasets until the introduction of the SOREL dataset in 2020, which expanded from 1.1 million binaries to 20 million binaries, including 10 million disarmed malware samples ready for feature extraction [88]. The Australian Defense Force Academy (ADFA) is the author of two datasets: the Linux dataset (LD) [123, 124] and Windows dataset (WD) [125]. Both datasets provide a comprehensive simulation of a HIDS based on the collection of system calls; however, a significant downside exists for the ADFA-WD as it was collected solely on Windows XP, which limits the applicability to future generations of Windows OS [125].

Insider threats are considered one of the more emerging sources of security vulnerabilities for government and firms. CERT identified that 15–24% of firms experience an insider incident perpetrated by a business partner [126]. It has also been noted that a quarter of cyber security risks are due to

insider threats, meaning that current or close business partners are considered as much of a threat as ransomware from a security standpoint [17]. That is why, a dataset such as the CERT insider threat V.2 dataset is so important in our understanding and tracing of threats that exist in network topologies [127]. The dataset includes several synthetic threat scenarios, accompanied with information related to HTTP records, employee info, and log on/off times, among other indicators. A summary of the datasets discussed along with some information on their makeup is shown in Table 9.

Virus repositories are also a source for millions of malicious binaries and source code for malware research. The Zoo (https://github.com/ytisf/theZoo) from [263] contains hundreds of malicious binaries that are updated on a regular basis as new threats emerge and as virus source code becomes available [264]. VirusTotal (https://www.virustotal.com/gui/home/) contains one of the most comprehensive repositories used in the industry today. Malicious binaries can be uploaded or searched via MD5 hash to provide a detailed summary of the threat and other metadata. VirusTotal also comes equipped with a public and private API that allows threats to be uploaded while returning a detailed report, along with which AV vendors have already developed a signature for the given binary. Virushare (https://virusshare.com/) is a searchable sample database, boasting 34 million + malware samples for use for analysts, researchers, and the security community [265]. Other less popularized repositories for sharing malware for research purposes include Malshare, VirusBay, and Das Malwerk.

4.2. Metamorphic Generation Kits. Virus generation kits facilitate the creation of a bulk of the newly generated virus signatures we see every day. These kits perform some, if not all, types of obfuscation outlined in Section 2 to evade signature-based techniques and are a significant problem for AV vendors and researchers alike. In addition, some kits even provide functionality whereby users can customize the level of obfuscation and encryption to introduce variation into the malware generation and are even able to enact antiemulation and armoring behavior. Some generation kits have been easily flagged by AV vendors since their generated code would contain similar code between generations; therefore, only a few signatures developed could flag the entire generation, rendering the generation kit obsolete. Depending on the generation kit, COM and EXE viruses can be produced directly, while other kits generate the virus assembly code. For example, Borland TurboAssembler TASM 5.0 can assemble an ASM file into an object file and then TLINK takes the object files and libraries and links them together to produce virus executables. As demonstrated in Figure 9, disassemblers such as IDA Pro can be used to produce the ASM files [266]. The ASM files can then be used to extract opcodes and other features sets for use in malware classification [267]. This section will discuss several popular generation kits used in research, with a brief description on some of the obfuscation techniques used by each generator.

The phalcon-SKISM mass produced code generator (PS-MPC) was developed in 1992 and includes over 25 options for different types of encryption and payload types, as well as having options to be memory resident. The generator employs its own decryption routine but lacks options for stealth techniques. PS-MPC generates files that reside in memory long enough to infect all COM and EXE files. The advantage of PS-MPC at the time of creation was the ability to carry out code generation in batches due to the generator operating as a code-morphing engine as it is script-driven [43]. While all PS-MPC-generated codes today are readily flagged by AV vendors, the generator is still used today for research on metamorphic malware [31, 51, 268–271]. The mass-produced code generation kit (MPCGEN) was first developed in 1993 and was used to create CFG files which were then passed to PS-MPC followed by TASM to produce 32-bit executables. The name "mass-produced" comes from the fact that the process of generating, compiling, and assembling can be carried out for 500 files in as little as 25 minutes. Similarly, MPCGEN is used to produce a high quality and quantity of metamorphic variants for research purposes [51, 56, 271–275].

The second-generation virus generator (G2) was developed in 1993 and produces COM and 16-bit EXE infectors. It employs several code substitution techniques, and as an extension to PS-MPC, introduces antidebugging and antiemulation features, as well as resident and nonresident viruses. G2 has an easily modifiable source code to allow customization by an advanced programmer, and the routines it uses are semipolymorphic. G2 to do this day is a go-to for generating polymorphic variants [31, 50, 51, 56, 58, 59, 67, 73, 268–278].

Virus creation lab for Windows 32 (VCL32) was created in 1992 but was revamped in 2003. Created by a virus writer named Nowhere Man, a member of a group called NuKE, this generator can produce the assembly source code of viruses. This means the assembly code needs to be compiled and linked afterwards before they are active. The versatility of VCL32 comes from being able to customize activation conditions based on date, time of day, number of infected files, computer country code, version of DOS, or the amount of RAM available. VLC32 supports COM file infections, generating companion viruses, as well as various encryption and infection strategies. As a complete package with a GUI and drop-down menus, the most recent version VCL32 released in 2004 is commonly used in research [31, 50, 51, 56, 268, 272, 274, 279, 280].

The next generation virus generation kit (NGVCK) is one of the more popular virus construction kits available. Developed in 2001 with the most recent version released in 2003, NGVCK has been widely adopted for use in developing 32-bit PE-EXE polymorphic malware, especially in a research environment [31, 39, 47, 50, 51, 56, 58, 67, 268–275, 277–288]. Options for encryption include rotate without carry ROR/ROL, Twos complement negation NEG, Ones complement Negation NOT, logical exclusive or XOR, and addition/subtraction ADD/SUB. NGVCK can carry out dead code insertion, subroutine reordering, code substitution, and registry renaming, and all are very effective

Table 9: Summary of the more prevalent malware datasets publicly available for use by researchers.

	Features	Description	References
NSL-KDD	21 attacks from 4 families (DoS, probe, root 2 local (R2L), user 2 root (U2R)), 41 features	125,973 training examples (19.85% benign), 41 features	[93, 95, 128–160]
KDD99	21 attacks from 4 families (DoS, probe, root 2 local (R2L), user 2 root (U2R)), 41 features	489,431 training examples (20% benign)	[91, 93, 95, 131, 161–185]
DARPA 99	Raw TCP/IP dump files; 58 attacks from 4 families (DoS, probe, root 2 local (R2L), user 2 root (U2R)), 41 features	6,591,458 training examples	[89, 186–193]
UNSW-NB15	Raw Traffic as Pcp files, 9 types of attacks (fuzzers, analysis, backdoors, DoS, exploits, generic, reconnaissance, shellcode, worms); 49 features	175,341 training and 82,332 testings examples, 49 features	[95, 152, 153, 177, 194–213]
MalImg	Malware binary, converted to 8 bit vector then 8 bit grayscale image	9,339 training examples; 25 Malware families	[100, 102–105, 107–109, 214–217]
CERT insider threat V.2	HTTP records, emails, employee info; 5 unique scenarios of suspicious activity. 191 suspected users	33,771,224 training examples, 33 features	[134, 218–235]
Drebin	Features extracted from application manifest and dex code. 8 core feature sets. 179 malware families	5,560 malicious and 123,453 benign applications	[112, 113, 134, 139, 158, 233, 236–241]
Microsoft malware classification	Hexadecimal representation of binary content with metadata manifest; 9 classes of polymorphic malware disassembled using IDA packet disassembler	20,000 malware samples	[101, 103, 104, 214, 242–256]
ClamMP	Header fields of PE headers; 54 raw features 15 derived	5210 examples (47.75% benign)	[120, 136, 257–259]
AAGM	Raw traffic from pcap files, 2 types of malicious applications (adware and general malware)	1,900 malicious applications (80% benign)	[115, 116]
EMBER	Raw features extracted from PE files in JSON format	900,000 training and 200,00 testing examples	[121, 260–262]
IoTID20	Raw traffic from pcap files, 12 features, 8 attacks types (DoS, ACK flooding, brute force, HTTP flooding, UDP flooding, ARP spoofing, scan host port, scan port OS)	40,070 benign and 58710 malicious examples	[118, 119]
ADFA-LD	Linux system calls; 6 attacks classes (FTP, SSH, poisoned executable, Adduser/Meterpreter, TikiWiki exploit, PHP remote vulnerability) 26 features	2,430,0162 benign and 317,388 malicious examples	[124]

Figure 9: Assembly and compilation of a virus executable.

techniques for obfuscation. In [51], NGVCK was compared to other popular generation kits, including G2, MPCGEN, and VCL32, and was noted to produce the highest rates of obfuscation compared to other kits. A similarity metric was used to compare assembly programs, and no similarity was found to have G2 and MPCGEN, up to 2.4% was found with VCL32, and normal files had similarities between 0.98% and 1.2%. In [271], only a 10% similarity was found between NGVCK when run over multiple iterations, meaning that the kit produces a large amount of variability between uses. An example of two virus variations produced by the NGVCK generation kit is shown in Table 10. Obfuscation produces two semantically similar variants using garbage code insertion, instruction substitution, and subroutine reordering as techniques.

A more recent polymorphic engine was introduced in [69] as the virus and metamorphic worm (MWOR) generation kit. The effectiveness of the generation kit was exemplified in [270] for being able to fool common statistical analysis. The kit has also found more recent interest in research as it is able to control for the proportion of garbage code and subroutine reordering possible [270, 271, 273, 282, 283, 286]. This is extremely effective because inserting a certain amount of garbage code from benign files has demonstrated an improved ability to thwart AV scanners [39]. This chapter does not provide an exhaustive list of generation kits, and on the contrary, these kits represent a small subset of available kits widely distributed. Websites such as VxHeavens were one of such sources until the website was taken down in March 2012 by Ukrainian police. Repositories containing over 200+ generation kits once hosted on VxHeavens can be found circulating online to this day. Included in these kits as discussed is antiarmoring and antiemulation capabilities. Some of these will be discussed in the next section.

4.3. Anti-Emulation, Stealth, and Code Protection. Antiemulation is an all-encompassing term that includes all the various armoring, stealth, and/or code protection techniques that are used to thwart or burden the process of reverse engineering of a malware sample. According to Symantec, approximately 28% of malware are VMware [12]. One of the shortcomings of virtual machines and other honeypot deployments is that the environment they are deployed in is static, with several configurations set to default. It is for this reason that antiemulation malware can check the environment for indicators of virtualization and fail to execute or burden the reverse engineering analysis with cumbersome instructions. This section will cover some of the actions taken by antiemulation malware to exploit their virtual environment and prevent security experts from understanding the full breadth of their behavior. Antiemulation checks fall into four categories: human interaction, configuration-specific, environment-specific, and VMware specific checks [289, 290].

4.3.1. Human Interaction. Checks to see if actions routinely carried out by a user are being performed. This includes mouse movements, use of the clipboard, and opening and closing windows. The Cuckoo Sandbox, for example, has a setting which provides this sort of functionality for each malware submission. Trojan Upclicker is a virus variant that monitors user input in the form of a left click in order to identify sandbox environments. It does this by using the SetWindowsHookEx() and GetLastInputInfor() API to determine the rate of user input over time. This would identify the presence of sandbox environments as automated analysis does not require the use of an auxiliary keyboard and mouse [291].

4.3.2. Configuration-Specific. Uses time periods or other configuration to execute at a later time and date only if certain conditions are met. The Duqu virus, which was first identified in 2011, included a series of antistealth techniques in the form of delays as a precautionary measure [292]. Code injection only occurs after approximately 10–15 minutes, and the lifespan of Duqu is set by an unknown communication module that removes its hooks, deletes its kernel driver, and removes its registry key once the timer has elapsed [292, 293]. The Kelihos botnet and Nap Trojan both make use of the SleepEx() and NtDelayExecution() for extended sleep calls, with the Kelihos botnet having affected 41,000 users before being identified and taken down. Hastati has a hardcoded check which is executed only at 2 pm on March 20, 2013. Otherwise, it does not execute if GetLocalTime() returns a time less than that, indicating the presence of a virtualized environment [294].

4.3.3. Environment-Specific. It looks at the settings and parameters of the host operating system and hardware and decides whether to execute based on those findings [295]. Virtual machines incorporate virtual hardware which tends to have consistent configurations between VM deployments. Hardware such as network adapters, USB controllers, and audio adapters are all virtualized, meaning that MAC addresses, USB controller types, and SCSI device types are all telling signs of virtualization. The *Scoopy Doo* tool developed by Tobias Klein uses Windows Script Host to read registry keys located in HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\ and HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class associated with SCSI and can also lookup keys that are associated with IO and ports for strings containing "VMware." In another application, malware can utilize the internal processor tick counter via the ReaD Time Stamp Counter (RDTSC) instruction. Based on a random bit value that is returned, the decryptor contained within the malware will decode and execute the virus body; otherwise, it will bypass and exit.

Table 10: Variations in code obfuscation used by the next generation virus generation kit. Adapted from [286].

Before obfuscation	After obfuscation (version 1)	After obfuscation (version 2)

Call function A
Function A: pop ebp
sub ebp, OFFSET function A
	Call function A
	Function A: sub dword ptr[esp], OFFSET function A
	pop eax
	mov ebp, eax
		add ecx, 0021751B; junk
		Call function A
		Function A: sub dword ptr[esp], OFFSET function A
		sub ebx, 00000909; junk
		mov edx, [esp]
		xchg ecx, eax; junk
		add esp, 00000004
		and ecx, 00005E44; junk
		xchg edx, ebp

4.3.4. VMware-Specific. It uses checks that add the ability for malware to look for specific indicators of virtualization based on the VMware software used by the host. One of the best examples is in the use of VMWare workstation's WinXP Guest virtual hardware which includes a running *VMtools* service and 300 references to *VMtools* in the registry. Another interesting adoption of VMware behavior is Pushdo. Pushdo uses PspCreateProcessNotify() to deregister sandbox routines [5, 290]. It also performs a check of the physical hard drive serial number and checks if it is set to a default value of 00 which is typically in virtual machines. In the work of [296], the authors looked at antiemulator behavior in android malware and noted volume identifiers, network interfaces, and invoking the GPU were all techniques used to obfuscate Dalvik virtual machines. Other evasion techniques, such as exception process timing, IMEI checking, and checking the variability in sensors have all been traced to emulation evasion in android malware [297–302].

Alongside the specific checks mentioned above, general antidebugging makes it difficult for researchers to extract signatures or strings to develop systems to protect against them. An example is the Bistro virus which inserts garbage code insertion and dummy loops before the decryptor stub. As a result, before the malware has even unpacked millions of instructions and burdens the emulator, and Bistro fails to run. During analysis, many malware variants are memory-resident, thereby requiring careful monitoring of viral payload to load itself into memory before it can be dumped and analyzed [61]. In the past, malware authors have been one step ahead in their efforts to thwart monitoring memory dumps or memory snapshotting. An example is the Zmorph virus which has its decryptor rebuilding its instructions line by line by pushing the result into stack memory. One of the earlier adopters of this sort of technique was the DOS/DarkParanoid which contained 10 different encryption functions which it used to encrypt its previously run instructions while only allowing its current instruction to be decrypted at any point in time. Without a conventional decryption loop, it is a true polymorphic memory-resident variant. The use of other so-called "stealth viruses" employed reconnaissance of the OS by waiting until AV products check-summed programs to check for changes. When a file was read, as opposed to executed as is the case with user input, it took that as indication of check-summing by the AV and removed itself from the target executable. Finally, once it waited until the file was closed, it then reinfects the file [303]. Using this process, it can follow the AV and infect every file on disk. A thorough summary of antidisassembly, antidebugging, and antiemulation techniques can be found in [43]. For a summary of android application hardening used by malware authors and developers, we refer the readers to the work of [304].

5. Approaches to Feature Analysis

Malware features are typically categorized into two types: static and dynamic. Static features incorporate all the unique compositional information of the executable, irrespective of the contextual information of the target system [305–308]. That is to say, the static features of an executable would be the same regardless of what machine the malware is deployed on. Static features typically include the portable executable (PE) structure, assembly code instructions [5], list of DLLs, n-grams, and byte sequences. PE structure features would include information related to PE sections, resources, application programming interface (API) calls, as well as which dynamic link libraries (DLL) are imported/exported. Most modern antivirus (AV) products employ the use of a signature database which contains known signatures of the static features of malware. Alternatively, dynamic features include API and DLL call graphs, information gathered from the file system, registry, as well as process and thread activity and the consumption of kernel resources. Dynamic analysis can also include temporal snapshots of process execution, memory, network, and system call logs [309]. Dynamic analysis is OS-specific because depending on the system resources, account privileges, and other environmental variables, the malware will behave differently and have a different signature as a result.

The ability for malware to mutate has also presented a problem for researchers, which render many of the legacy static approaches to malware research obsolete. As a result, dynamic analysis has been presented as the de facto standard in classification approaches as it is impervious to routine obfuscation and packing carried out my mutating malware. Nowadays, dynamic analysis represents some 51% of the analysis methods in the body of literature examined [306], with a unique combination of feature sets and model architectures being used to perform classification. It has been noted that malware classification is not a trivial problem,

with some presenting it as an NP-complete problem [63] to identify a bounded-length mutating virus or a polymorphic variant of one [310]. Characterizing malware is the fundamental issue of concern, and researchers and practitioners are constantly refining their methods to stay ahead of the curve. Figure 10 provides an illustration of the feature pipeline used for most malware classification approaches. Both static and dynamic features form the bedrock in the characterization of malicious behavior. Any number of these features can be combined to form a hybridized model for feature analysis, which is unofficially the third form of characterization.

Many of these methods are covered in the comprehensive review of [308, 309], but this work will simply provide a narrow overview of malware detection approaches as it concerns API calls. While API calls are just of one of the many forms of static and dynamic behavior, it is one of the most consequential and information rich sources of discrimination. But first, an introduction to the source of APIs, files known as dynamically linked libraries, is required and will be the topic of the next section.

5.1. Dynamically Linked Libraries. Dynamically linked libraries, or DLLs, are libraries of code that are written by vendors such as Microsoft as well as third parties to coordinate and manage resources on the Windows OS. DLLs are fundamentally libraries of code that contain one or more functions, indicated in their Export Address Table (EAT), which identifies and whose functions are available for export to other processes. DLLs are structurally equivalent to executables, with the exception being that their main function is called DllMain, and they cannot be executed without the use of helper functions RUNDLL.exe or RUNDLL32.exe, for 64-bit and 32-bit, respectively. DLLs are useful because they allow multiple processes to share the same library of code loaded into memory, thereby reducing the time required to recompile each process and the amount of memory overhead if the same code segments had to be loaded in memory multiple times. Because each process does not need to include static code of its functions, it keeps file sizes smaller overall when it can connect to an already running copy of the library of functions. It also has the advantage of allowing the OS vendor to update a catalogue of core DLL libraries which can work with subsequent versions of the OS.

When a DLL is requested to be loaded by an EXE, it does so through by checking some default directories first. There is a known registry key in KnownDLLs that tells Windows that the well-known DLLs should be found in the System32 path; otherwise, it searches in the .exe directory, the current working directory, the %SystemRoot% directory, the 16-bit system path, and then the directories in your environment PATH. DLL order hijacking is the process by which malicious actors inject their own malicious DLLs somewhere in this load order so that their payload is loaded instead of a legitimate DLL. For example, ntshrui.dll is loaded by explorer.exe, but it is not a known DLL and therefore can be susceptible to load-order hijacking. DLLs that are fully protected can recursively load other DLLs that are not protected, which forces the next executable to follow the default search order and be prone to hijacking. The tool Dependency Walker (https://www.dependencywalker.com/) can be used to see the dependency tree between loaded DLLs on the OS. Legacy malware would change the Import Address Table (IAT) to point to a new address in memory for the DLL it needs. Changing pointers to new malicious address locations with malicious payloads has since been rectified on newer versions of Windows as it becomes apparent if all the address locations for functions are in higher memory space 0x7C86 and a single function is loaded into 0x3420 then most likely that IAT entry has been changed with a hook by a rootkit. Alternatively, malware can just modify the DLL inline, requiring no changes in pointers just the code, leading to a vulnerability commonly known as DLL proxying which is much harder to detect but can be alerted to using integrity checking.

Potentially vulnerable DLLs can be observed if using tools such as SysInternals' Process Monitor (Procmon (https://docs.microsoft.com/en-us/sysinternals/downloads/procmon)). In Procmon, if a DLL is not found and it is not core to the functionality of the process, it will return an entry NAME NOT FOUND. Using an out-of-the-box option like Metasploit's (https://www.metasploit.com/) msfvenom will produce a DLL than can be put in place of the missing DLL, thereby running the malicious payload and executing a successful DLL hijacking. Other tools such as the SANS (https://www.sans.org/blog/detecting-dll-hijacking-on-windows/) tool can be used to search for DLLs that appear multiple times, are unsinged, and are in unusual folders. More common in research, the Dependency Walker tool (https://www.dependencywalker.com/) makes it easy to view the mapping of imported DLLs and to even view a hierarchical view of all dependencies between modules by looking at the IAT. The authors in citewang 2008 separated DLL usage according to implicit dependency, delay-load dependency, and forward dependency, which are all responsible for the static loading of DLLs in 3 tiers of hierarchy. Tier 1 starts from those used by the main program, followed by Tier 2 which have DLLs invoked by other DLLs that are not in the main executable, with Tier 3 being the entire statically loaded tree. The authors created a one-hot encoded vector if the particular DLL existed in the program and used that feature mapping for classification. In [311], a similar approach was taken which relied on the DLL dependency tree but incorporated encoding tree string dependencies. The authors looked at all the tiers of DLLs which loaded and created a depth-first representation where the original executable is the root node and all nodes from root to leaf are assigned a unique integer value. They then used CMTreeMiner which extracts closed frequent subtrees that exist in a particular executable, and one-hot encoded a feature vector if a particular subtree exists in the executable. Looking at depths of subtrees from 3 to 6, accuracies as high as 98%+ were obtained following random forest and naive Bayes classifiers. The work of [312] did not go in as depth as [310], but the authors looked at the number of API calls by a DLL in addition to the list of DLLs used and the API calls

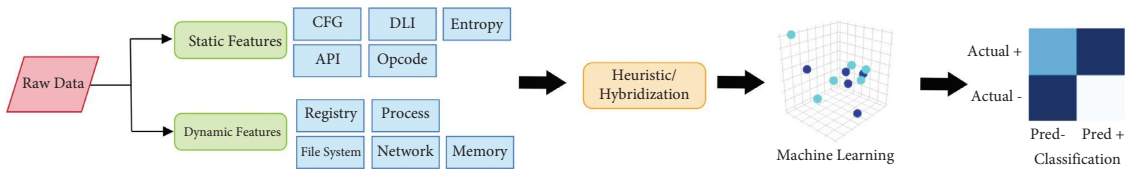

Figure 10: Summary of the feature pipeline for the classification of malware.

made. In any case, while DLLs do provide a good proxy of malicious intent, it is in fact the API calls that are made that are the real discriminator. For this reason, researchers turn their focus towards API calls and their usage among malware variants.

5.2. Windows Application Programming Interface. Windows API calls are interfaces provided by DLLs to access low-level resources [313]. API calls come in two flavors: user-level and kernel-level APIs. User level APIs operate at Ring 3 and provide the average user just enough privileges to access system resources to perform typical workloads. The actual hardware on the other hand runs in the kernel mode, which makes use of the kernel level APIs that are not directly available to users for the sake of security and stability of the OS. From the stability perspective, a user-level crash results in an error message, while a kernel-level crash results in the OS crashing. From the security side, malware could reside in the kernel and operate at a layer that is indistinguishable to the user or any Ring 3 defenses. Nowadays, it is much more unlikely to see malware residing in the kernel, as the Windows OS has made it more difficult to run code in the kernel and make use of rootkits. Ultimately, to make use of the kernel, all userland code uses Kernel32.dll as a gateway to communicate with Ntdll.dll which, in turn, communicates with the kernel.

The fascination with API calls comes down to the fact that API calls provides a higher resolution of analysis of the operation of any given process. It is the case that API functions and system calls are related to the services provided by the OS [309, 314, 315]. As the API is responsible for all system resource management, it is a particularly discriminating feature for malware classification as it provides the basic functionality for everything from networking to saving files to disk. The usage of APIs and patterns in usage can be very telling. Similar to the overarching view of static and dynamic analysis of behavior, APIs are approached from a static and dynamic perspective as well. In dynamic analysis, the run-time behavior is monitored, and ideally, all code segments are traced to reveal the behavior of the malware. This circumvents the obfuscation techniques of encryption, packing, and polymorphism [316]. Static analysis on the other hand can be fooled by adding fake API calls [317] or API calls typical of benign event activity [318]. It is also the case, as mentioned in Section 5.1, that the imported functions of a DLL may or may not ever be called, which can be used as a distraction from the real nefarious purpose of the malware.

Features such as the API call function names, parameters, and the return values of an executable can be extracted from the APIs [319]. Monitoring the API calls is an approach to detecting the malicious behavior of software; however, there is no clear distinction between malicious APIs and benign APIs as all native APIs are a helpful utility given the right context. The next section will outline some of the nefarious usages of APIs by malware authors and how they balance stealthiness with functionality.

5.2.1. Malicious Windows Application Programming Interface Usage. Broadly speaking, API usage can be categorized into 7 categories based on the functionality they provide to a process [314, 320]. Researchers have also made use of similar categories to classify malicious intent [184]. Some of the malicious functionality APIs can provide to executables and include the following:

File: create a file in sensitive folders; delete or hide files; file directory traversal

Process: inject DLL into a running system process; create mutex to prevent execution

Memory: free up or occupy memory; minimize memory usage

Registry: add or delete system service. Autorun, hide, and protect

Network: open and listen on a port, communicate over e-mail service, communicate with CnC server

Windows Service: terminate windows update, firewall, setup Telnet or SSH

Others: hooking keyboard, hiding window, scan for existing vulnerabilities and configuration

Code injection usually begins with the usage of third-part DLLs or injecting code into a Windows DLL. Malware makes use of Ntdll.exe indirectly to make use of kernel APIs, so checking the stack trace of event activity is important [321]. Malware authors have to balance gaining increased functionality at the cost of rising suspicion, so a careful deliberation of which APIs to use is always in mind [322]. Native Windows API calls that begin with NTtQuery are popular for malware, as they include functions such as NTtQuerySystemInformation and NTtQueryInformationProcess which provide much more information about the host system. More invasively, early rootkits would make changes to the System Service Descriptor Table (SSDT) which contains addresses to the kernel functions, which would instead be changed to malicious driver functions. If, for example, a typical address of a kernel function is set to 804d7000 for ntoskrnl.exe, then one can look at addresses which are not familiar and contained within the address space typical for

kernel drivers. With x64 bit versions of Windows starting with XP, *PathGuard* prevents modification of the kernel and the kernel code in the SSDT and the Interrupt Descriptor Table (IDT). The IDT takes care of exception handling, so rerouting the response to interrupts to malicious code would be highly disruptive. As a precaution to prevent making changes to native Microsoft DLLs and APIs, Windows Vista was the first Windows version to introduce digitally signed drivers. Some of the example use-cases and APIs used by malware are the following:

(a) File: if software wishes to make use of the file register, it can do so using CreateFile, ReadFile, and WriteFile. Malware can make use of CreateFileMapping or MapViewOfFile which loads the file into RAM, avoiding writing to disk all-together. Some malware types, like Ransomware, perform high volume file and encryption operations to carry out its function [323].

(b) Process: it is typical for malware to use OpenMutex to check if a mutex exists for a running malware executable. Malware can make use of DLL injection or direct injection. Code can be injected into a running process using VirtualAlloxEx and WriteProcessMemory. When the code is injected into an executable such as Explorer.exe, the same privileges hold for the executable it is injected into. Asynchronous procedure call (APC) is a process by which malicious code is attached to the APC queue of a process' thread. WaitForSingleObjectEx is the most common call, with QueueUserAPC being used for queues running on a thread. It can be run from the kernel using KeInitializeApc and KeInsertQueueApc. APC remains a known vulnerability on the MITRE ATTCK knowledge base [324].

(c) Registry: when it comes to making use of the Windows registry, malware can gain *persistence* so that it can load whenever Windows restarts [316, 325]. Most commonly the Run key located in HKLM\Software\Microsoft\Windows\CurrentVersion\Run can set executables to run automatically. The Sysinternals tool *Autoruns* (https://docs.microsoft.com/en-us/sysinternals/downloads/autoruns) can be used to check dozens of registry locations, drivers loaded into the kernel, and any other DLLs. Other options for persistence include running *Services* which are typically more powerful than administrator privileges. Other registry entries include *AppInit_DLLs*, which is a registry key that contains DLLs that are attached to processes that load User32.dll. This option has can be disabled in Windows 8 and later versions when Secure Boot is enabled. *WinLogon Notify* launches during log on, sleep, or when the lock screen is open. Adding a malicious DLL to the ServiceDll parameter in the registry allows a malicious service to start its malicious service DLL into a loaded svchost.exe [326].

(d) Networking: certain network API usage can be indicative of malicious intent as networking APIs provide different levels of flexible. For example, the APIs in Wininet.dll will use higher level APIs for HTTP and HTTPS communications. Malware might use the raw Winsock libraries located in ws2_32.dll if there is a need to provide further flexibility to their malicious arsenal. The Metasploit framework can produce shellcode that acts as a listener on a port by creating a simple process using CreateProcess. The configuration for STARTUPINFO is set to a socket, thereby creating a remote shell. This setup allows for I/O and error handling for cmd.exe and does so with the command window suppressed to remain stealthy.

(e) Other: malware downloaders and launchers use URLDownloadtoFileA to download a file from a URL and then execute the file by making a call to WinExec. Keyloggers use hooking or polling. Hooking uses an API such as SetWindowsHookEx to notify about a key press, while polling is conducted using GetAsyncKeyState and GetForegroundWindow to poll key states during any time period.

Researchers have looked beyond individual API calls and have investigated API call distribution [327]. A summary of some of these classes of API usage used by researchers is shown in Table 11. The issues arise in that, and it requires significant domain expertise to create and update a database of API calls for particular malware variants or families. It is also the case that there is significant overlap between malicious and benign API usage, thereby making it difficult to alert malware without alerting false positives. The work of [328] developed a similarity metric to trace the similarity between malware variants and Stuxnet based on groups of API calls. It comes to reason that groups of API calls in succession, or the distribution of API calls, can provide further insight into malicious behavior [334]. For this, we investigate some of these research methods in the following section.

5.2.2. Classification of Windows Application Programming Interfaces. The investigation of API calls in the context of feature extraction is sometimes referred to as API call sequence or API call traces. In either definition we are concerned with the patterns that arise in the sequence of API calls used one after another. Early adopters of this form of investigation used Hofmeyr API call sequences, whereby behavior profiles were established between two sequences of API calls based on Hamming distance [335]. Originally, UNIX system calls were traced, and the investigators were motivated by the immune system in their attempt to draw an analogy between sequences of system calls and chains of amino acids in the human body. API call sequences have been leveraged in several applications involving malware detection [160, 184, 316, 336–339], as well as in tracing the API call traces during event activity [316, 340–343]. Overall,

Table 11: Summary of malicious API usage by behavior type.

Behavior	APIs	References
General behavior	ShowWindow, GetWindow, WriteFile, WinExec, ShellExecuteA, OpenProcess, VirtualAlloc, *Hook, *Exception, *Shutdown, *Crypt, *Debugger, *Shellexecute, *Manager	[1, 328, 330]
Stealthiness	NtDelayExecution, FindFirstFileA, FindNextFileA, GetProcAddress, LoadLibraryA, OpenProcess, sleep	[5, 328]
Kernel	*Ldr*, *Section*, *DuplicateObject*, *Make*, *Object*, *Resource*, *UdiCreate*	[1]
Memory	*Memory*, *Volume*, *Space*, *Buffer*	[1]
Registry	CreateKey, OpenKey, CloseKey, RegOpenKey* RegSetValue, RegQueryValue, *EnumKey, *DeleteKey, *SetKey, *Enum*	[1, 316, 331]
Reproduction	*FindFirstFile, *CopyFile, GetFileType, SetFilePointer	[316, 331]
DLL injection	SetWindowsHookEx, CallNextHookEx, CreateRemoteThread, OpenProcess, LoadLibrary, GetProcAddress, VirtualAllocEx, WriteProcessMemory	[5]
Search files	FindClose, FindFirstFile, FindFirstFileEx, FindFirstFileName, TransactedW, FindFirstFileNameW, FindFirstFileTransacted, FindFirstStream, TransactedW, FindFirstStreamW, FindNextFile, FindNextFileNameW, FindNextStreamW, SearchPath	[1, 327]
Copy/delete files	CloseHandle, CopyFile, CopyFileEx, CopyFileTransacted, CreateFile, CreateFileTransacted CreateHardLink, CreateHardLink, Transacted, CreateSymbolicLink, CreateSymbolic, LinkTransacted, DeleteFile, DeleteFileTransacted	[327]
Get file information	GetBinaryType, GetCompressed, FileSize, GetCompressedFile GetFileInformation, ByHandleEx, GetFileSize, GetFileSizeEx GetFileType, GetFinalPathName, ByHandle, GetFullPathName, GetFullPathName Transacted, GetLongPathName, GetLongPathName, Transacted, GetShortPathName, GetTempFileName, GetTempPath SizeTransacted, GetFileAttributes, GetFileAttributesEx, GetFileAttributes, Transacted, GetFileBandwidth, reservation, GetFileInformation, ByHandle	[327]
Move files	MoveFile, MoveFileEx, MoveFileTransacted, MoveFileWithProgress [327]	
Read/write files	OpenFile, OpenFileById, ReOpenFile, ReplaceFile, WriteFile, CreateFile, CloseHandle	[327]
Change file attributes	SetFileApisToANSI, SetFileApisToOEM, SetFileAttributes, SetFileAttributesTransacted, SetFileBandwidthReservation, SetFileInformationByHandle, SetFileShortName, SetFileValidData	[327]
Metamorphic engines	HeapAlloc, LocalFree, HeapCreate, GetStartupInfoA, GetCommandLineA, GetEnvironmentStringsW, FreeEnvironmentStringsW, GetModuleFileNameA, GetCurrentProcess, CloseServiceHandle, GetCurrentProcessId, GetProcessHeap, HeapReAlloc, SetFilePointer, SetFileAttributesA, GetFileAttributesW, FindFirstFileA, FindClose, SetThreadPriority, GetCurrentThreadId, GetProcAddress, GetModuleHandleA, ResumeThread, GetEnvironmentVariableA, ExitThread	[275, 328]
G2	GetCurrentProcessId, GetConsoleMode, SetConsoleMode, FileTimeToDosDateTime, CreateFileW, GetFileSize, FileTimeToLocalFileTime, GetFileTime, LocalFileTimeToFileTime, SetFileTime, SetFilePointer, SetFileAttributesW, GetFileAttributesW, GetKeyState, ConsoleMenuControl, AppendMenuW, ReleaseMutex, FindFirstFileA, FindClose, SetThreadPriority, GetCurrentThreadId, GetProcAddress, GetModuleHandleA, ResumeThread, GetSystemTimeAsFileTime, GetTickCount, QueryPerformanceCounter, InitializeCriticalSection, LoadStringA, FormatMessageA	[332]
MPCGEN	HeapAlloc, LocalFree, GetVersionExA, HeapCreate, GetStartupInfoA, SetHandleCount, GetCommandLineA, GetEnvironmentStringsW, FreeEnvironmentStringsW, GetACP, GetCPInfo, GetStringTypeW, GetModuleFileNameA, LCMapStringW, MultiByteToWideChar, WideCharToMultiByte, GetEnvironmentStrings, LocalFileTimeToFileTime, SetFileTime, ReadProcessMemory, AppendMenuW, GetLastError, GetSystemTimeAsFileTime, GetTickCount, QueryPerformanceCounter, InitializeCriticalSection, FormatMessageA, GetCurrentProcess, DuplicateHandle, GetConsoleMode	[332]

Table 11: Continued.

Behavior	APIs	References
NGVCK	GetClassLongW, CreateFontIndirectW, DeleteCriticalSection, TlsFree, UnmapViewOfFile, CloseHandle, GetCurrentProcessId, EnumDesktopsW, EnumDesktopWindows, CloseDesktop, GetProcessHeap, SetUnhandledExceptionFilter, OpenDesktopW, GetProcessWindowStation, GetUserDefaultLCID, CombineRgn, OffsetRgn, ExtCreateRegion, CreateRectRgnIndirect, SetWindowRgn, DefWindowProcW, PeekMessageW, SetCapture, SendMessageW, ReleaseCapture, MsgWaitForMultipleObjectsEx, PtInRect, GetRgnBox, HeapReAlloc, LCMapStringW	[332]
Stuxnet	LoadLibraryW, LoadLibraryA, GetModuleHandle, GetProcAddress, VirtualAlloc, VirtualFree	[328, 333]

API call frequency and API sequences are effective techniques in identifying data-flow dependencies in a process [315].

5.2.3. Application Programming Interface Frequency. One of the more primitive approaches to API analysis is API frequency analysis. It stands to reason that if malware and benignware make use of similar API libraries, then malware must make use of certain libraries or "malicious" APIs more frequently than others. In [319], considering API frequency alone was effective in achieving 97% accuracy in a multicategorical classification problem involving metamorphic malware variants. One takeaway was that incorporating sequential information did improve accuracy of the models, so frequency analysis is certainly a useful preliminary step in behavioral analysis. The work of [344] developed an end-to-end malware detector based on the frequency of occurrence of opcode and API calls. Their detector coined OPEM, demonstrating an increased area under the curve (AUC) and lower FPs with static calls and a hybrid approach. Unfortunately, the authors did not account for obfuscated malware which tend to be packed and have polymorphic engines which obfuscates the opcode. Their hybrid approach, which included API execution trace, did outperform all other feature sets used in their work [344]. Certain works, like that of [245], decided to use a frequency of a subset of 794 API calls extracted from 500 thousand malware samples. The authors then fused this feature set with other static techniques such as entropy and features extracted from the PE file such as the total number of assembly instructions in the .data and .rsrc section. The drawback to these approaches is that taking the most frequent API calls leaves out information of potential edges cases; it is also a fact that frequent API calls by malware are still routine events carried out by benignware, such as reserving memory, creating a file, etc. The work of [345] approached the problem in a similar fashion, where they eliminated API calls with low frequency. Again, doing so removes important edge-cases and is used typically to reduce the size of the feature vector space to improve training times. These aforementioned works all made use of ML techniques to classify their malicious behavior. Other works make use of statistical similarity metrics to differentiate malicious versus benign by using one or more metrics of comparison. For example, in [304], the authors made use of information gain to select the features based on the sequence of opcodes from android applications. Based on some key obfuscation techniques discussed thus far, including control flow obfuscation, string encryption, in addition to advanced techniques such as class encryption and reflection, the authors found several ML approaches were effective in detecting obfuscated samples.

In [346], the *cosine similarity* was proposed to compare API call frequency between two vectors to represent the similarity in vector space of a known signature to a new malware sample. The expression for cosine similarity is shown in equation (1). The motivation for using cosine similarity is that the measure computes the similarity between two vectors while excluding their magnitude. This has the effect of ignoring the impact of magnitude if one vector were to use an API much more frequently than the other, as the θ angle in equation (1) is indifferent to their magnitude.

$$\cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}. \tag{1}$$

The *extended Jaccard measure* is another similarity metric than is useful in measuring the degree of overlap in two sets [346]. As an extension to Jaccard for use in continuous or count attributes, it is effective in demonstrating the similarity, or the ratio of *set intersection*, between two sets in the context of set theory. The equation for this relationship is shown in equation (2). The numerator can be seen as expressing the set intersection, while the denominator can be seen as the union which acts as a form of normalization.

$$J(x, y) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|^2\|\mathbf{y}\|^2 - \mathbf{x} \cdot \mathbf{y}}. \tag{2}$$

The cosine similarity was used effectively to create a similarity matrix between the rarest 20–30% raw security events and events of the training set [160]. This approach was used to significantly reduce their dimensionality of their set by focusing their efforts on the similarities between a baseline set of unusual events and their dataset more broadly. In [347], similarity metrics were computed for API sequences that appear frequently, and both assembly instructions and API calls were considered in their work. API calls were noted to be faster in having a smaller signature; however, the authors noted that the API approach is bad for network applications such as PuTTY and encrypted files which show few or do not show any API calls. Their work did rely on

unpacked executables as it was limited only to static analysis. In [346], an API call frequency similarity measure was used followed by a chi-square test to test the representation based on a distribution from a known signature. Families of APIs of known metamorphic mutation engines were categorized and compared to one another and to the same mutation engine using both the cosine similarity and the extended Jaccard measure. An interesting finding was that comparing a similarity metric between variants from the same mutation engine provided a measure of the degree of *obfuscation*, which was shown to be the largest for the next generation virus creation kit (NGVCK), a well-known mutation engine. The work of [275] completed similar work, whereby a proximity index table was setup to compare the similarities between mutation engine families. Due to the sheer number of possible API calls, feature dimensionality reduction was carried out on the original 1000 or so APIs according to frequency. The authors noted that common APIs were used between mass code generator (MPCGEN) and NGVCK-generated viruses. An approach that included data mining was taken in [320], whereby the calling frequencies of the raw features are calculated to select a subset of features, and then principal component analysis (PCA) is used for dimensionality reduction of the selected features. In total, 24,662 API function calls, 792 DLL features, along with PE header info, were considered in their feature set while considering only the top 30 DLLs according to frequency [320]. To address the issue with high-dimensional data, the authors in [336] developed a string-based malware detection system that focused on the top 3,000 interpretable strings that included API names using a max-relevance algorithm. Their feature parser extracted strings from 9,838 executables and classified them as Backdoors, spyware, Trojans, and worms, in addition to benignware. While these techniques have been proven useful in many controlled scenarios, frequency-based analysis is still prone to malware which can obfuscate themselves to avoid heuristic detection. For this reason, sequence analysis is used.

5.2.4. Application Programming Interface Sequences. The investigation of API sequences has become the de facto standard for many behavioral approaches as the information contained within sequences is too powerful to rely on the API frequency alone. It has also led to the adoption of natural language approaches which will be discussed in Section 5.4. The work of [316] provided an example of the flow of information surrounding a process that can act as a template for how to carry out sequence analysis of APIs. The three flow paths are as follows:

(1) The API call GetModuleFileName takes a NULL character as its first argument which returns the malware file path

    (1.1). the path can be passed to CopyFile to open the executable and run its processes
    (1.2). or, if desired, a process can call CopyFile on itself with the share permission shared to NULL, thereby preventing applications from opening and scanning the file

This example serves to demonstrate that two very different uses of CopyFile can indicate malicious behavior, and only once the whole context is understood can a detection system alert it. An application that performed this successfully was in [337] where 2,727 unique APIs were categorized into 26 groups based on functionality such as hooking, file and directories, registry modification, and others. Based on the sequence of the APIs, critical patterns were uncovered which were essential for core functionality such as screen capturing and DLL injection. Results demonstrated $F1$ scores as high as 0.999 with a focus on the longest common subsequence between existing malicious signatures and those of unknown variants. A similar approach was taken in [1] where 11 hand-crafted signatures of dynamic and static behaviors were created based on malicious operations spanning registry operations to device operation to kernel operations. These signatures were converted into semantic blocks based on the largest common subsequences between dynamic and static APIs. The work of [348] created a formulation that includes API sequences as part of a temporal domain, and pointers passed to API calls as spatial information. The motivation being similar to [316] in that an API call such as LocalAlloc takes in uBytes as an argument that is statistically lower for malicious files than benign files during allocation of the heap. Capturing this information in the spatial domain, while modeling the sequences of APIs in the temporal domain were effective in classifying 516 executables with accuracies as high as 0.966. Rather than focusing on API sequences as it pertains to general malicious behavior, researchers have explored common API sequence usage among malware variants and types. In [330], five classes of malware including Worm, Trojan-Downloader, Trojan-Spy, Trojan-Dropper, and Backdoor were associated based on the presence of 26 API categories and sequences. 534 malware variants were hooked and then categorized based on the presence of these API sequences, which were characteristically different for different malware types that aim to pursue different objectives through their API usage. In [349], the authors considered 9 behaviors based on sequences of 2–4 APIs in succession, while [315] looked at combinations of 3 APIs (such as CreateFile, WriteFile, and CloseHandle). The work of [350] obtained a 99.7% detection rate using several API calls sets, which included sequences of different lengths.

When it comes to determining appropriate sets of API calls for classification, researchers have pursued approaches in the data mining space to optimize for a set of association patterns towards a particular objective [351] and in this case, optimizing an objective that a sample belongs to a malicious or benign sample. Several papers have been published in this area, in particular those published out of the Xiamen University [352–354] focused on malware classification. Ultimately, regardless of the particular mining algorithm used, the idea is to find a set of API calls that support the objective of classifying malware from benignware. In [353], this was performed using a frequency pattern growth algorithm [355]. The goal is to create a frequency pattern tree which encodes sequence in a tree-like structure similar to a Huffman coding where parents of a node are encoded as

longer extensions of the child sequences. So, for a given API call API_i, it would exist as a leaf node, while its parent nodes would contain sequences that contain API_i such as (API_i, API_j) or (API_i, API_k). This is performed recursively up the tree, and frequencies are stored as satellite information at each node, and this is how rules are generated. A new sample is then matched against the rules according to the descending order of the rules' confidence and support [356]. The motivation is to maximize the likelihood that rules exist which can discriminate one objective from the other. This procedure was further described in [352] and used successfully to generate rules which parse 29,850 Windows PE files, half of which were malicious. In the approach of [356], the authors compared frequency mining approaches to ML approaches including SVM, decision trees, and naïve Bayes and noted a 2–9% improvement in classification accuracy. Because these approaches did extract the APIs from the PE files, this static approach is not effective for packed malware or APIs which are imported by the executable but never used. In a later paper by Ye and Yu [143], rule pruning was used for duplicate rules and only elected to use the top 100 API calls as no further improvement was shown beyond 100. While using a linear SVM, Aassociate classifier and novel hierarchical associative classifier, 26 thousand malicious samples were parsed and a precision value as high as 96% was achieved but with a low recall value of 34%. A thorough examination of the state of data mining approaches as it pertains to cyber security are covered in [357]. While handcrafting sequence signatures can be time-consuming and require knowledge of specific patterns in API usage, the alternative is to consider all possible subsequences of a given length and consider the usage patterns of all sequences simultaneously. While data mining does provide a compact representation to do this, more innovative works allow models to discern these rules on their own when coupled to ML approaches. For this purpose, n-gram representation is used.

5.2.5. Application Programming Interface n-Grams. One of the earliest forms of sequence analysis in the malware domain was carried out in [358]. It was also the first successful application of n-grams, which involves translating a sequence of L APIs into subsequences n long and doing so for every possible subsequence that exists in the original API sequence. This has the effect of incorporating information about the sequences of APIs with little preprocessing required. For any given API sequence, a sequence of length L would have $L - n + 1$ n-grams, where n is the length of the subsequences and assuming a stride length of one. So, for an API sequence 10 APIs long, we would have $(10 - 5) - 1$, subsequences for $n = 5$. The number of possible n-gram combinations would be $|C|^5$, which represents all the unique combinations of five APIs in sequence that are possible in the set of APIs C. The authors in [358] looked at short byte string n-grams of the PC boot sector which was 512 bytes long. They utilized an ML approach that removed the sigmoid activation and stored the weights as 5/6-bit integers. The technique became part of the IBM AV package and was successfully deployed to millions of machines.

The versatility of n-grams means that one can look at smaller n to generate shorter signatures which are noisy but more generalizable or use larger n to create more specific signatures which lead to lower false positives (FP) but at a cost of lower true positives (TP). The application of n-grams is known to have low FP rates with increasing sequence length L; however, the space complexity of n-gram sequences is exponential in the length of the sequences $O(|C|^5)$ [71]. The work of [359] focused their attention of the PE header and body and carried out static analysis using the top 500 most common 4-grams [360], representing DLL names. Results demonstrated that the header-only features are as relevant as body information and that separately, they both have a use-case [359]. Similarly, in [361], a 4-gram representation was used to model API sequences. The authors developed average confidence values of benign and malicious activity and used the average confidence of malware as a threshold. This simple thresholding obtained 90% accuracy; however, the work provided no indication of FP rates to support their findings. The work of [342] went one step further and carried out n-gram modeling of API call sequences based on the file system, network, and registry activity. This work was unique in that, and it separated API events based on the file system, network, and registry, to provide a further analysis of how these event categories fare in acting as discriminators. In all, the authors looked at over 17,900 malicious executables and obtained 92.5% test accuracy. Finally, [345] resorted to 3- and 4-gram representations but focused on the dynamic API usage after process execution. This resulted in 94% accuracy, but when coupled with static feature sets based on frequency, it improved the accuracy beyond 97%. The shortfall of n-grams is that sequences exceeding that of 4 or 5 are impractical to model due to the number of permutations of API calls, which significantly hinders the ability for models to attend to different behaviors. For this reason, we can pursue graph-based approaches in an attempt to consider different behaviors simultaneously.

5.3. Graph-Based Approaches. Graph-based approaches to malware detection have a long history. The earliest application of graph-based includes the use of control flow graphs (CFG) to evaluate unique control flow sequences of a program. A CFG is created as a directed graph where the nodes represent individual or blocks of program instructions and the edges represent the control flow between statements [310]. Within each CFG, we have a subgraph that is isomorphic to the whole graph. Trying to map a subgraph from one sample to another is part of the set of problems which includes the subgraph isomorphism problem which is NP-complete [362]. In Figure 11, we can see an illustration for the control flow from the Trojan.Emotet virus. This instruction segment belongs to the set of instructions that are responsible for spawning a child process which depends on the initial call to CreateEvent at the top of Figure 11. When examining such a control flow, the question becomes which

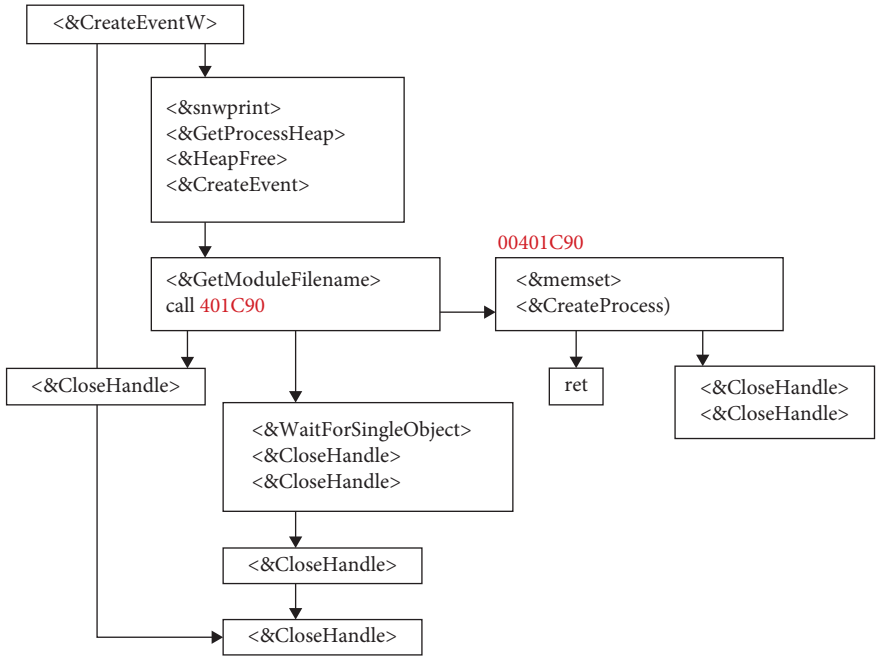

Figure 11: A CFG representation of the disassembled instructions for Trojan.Emotet produced in Ghidra.

segment(s) of instructions are responsible for malicious behavior. While this segment was carefully selected to show the behavior of Emotet, extracting similar segments from the entire malicious execution is cumbersome, especially when they include diversions and dead-ends. Extracting such segments as signatures and generalizing these signatures to flag future malware samples is the goal of CFG-based malware classification.

Most applications of CFGs look at extracting some subset of the flow of sequences to compare to other samples to establish a baseline for malicious control flow. One approach used by [363] looked at jmp, jcc, call, ret, inst, and ret opcode instructions and built the CFG based on only these instructions, thereby creating a reduced graph and leaving placeholders for the rest. Based on these, the authors created unique signatures for malware detection. In [364], the authors looked at the system call functions, which included call, jump, and conditional jump expressions in the x86 Intel instruction set. In [365], the authors looked at the most frequent subgraphs and simply excluded the rest. The sample set used by [366] included 25,145 functions which were 5 nodes (simple instructions) large and 15,439 unique functions which were 5 nodes long. Setting the threshold at 5 ensures that only atypical calls and procedures are included. One of the issues associated with CFGs is that the control flow is either (a) similar among all executables, regardless of malicious activity (also known as boilerplate code) or (b) is sometimes appended with benign code segments that are never executed but can confuse string-based scanning techniques [366]. This was considered by [367] in their CFG reconstruction based on system call logs extracted using *Procmon*. Their approach did not look at functions that were not loaded by the dynamic linker in order to remove boilerplate code. However, this is a double-edged sword as malware does not only rely on its Import Address Table (IAT) to fetch the APIs it needs, it can load those statically as well. An alternative approach used in [368] looked at contrast subgraphing [369], which is the opposite of graph isomorphism since it looks for the smallest subgraph of G_1 that does not belong in G_2. This approach lends itself well to looking for characteristically significant differences between malware and benignware, rather than developing signatures that look for similarities among classes. Alternatively, one can consider creating signatures as coopcode graphs that belong to malware families and therefore create high-level signatures that can be used to classify malware families based on the coopcode graph similarity [319]. While opcodes have been investigated extensively, Windows API usage has been shown to perform well at detecting polymorphic variants, [143, 160, 364] but the large size of potential subgraphs remains a limitation to graph-based approaches. Going more in depth, [370] examined not just the API functions used but also their function input arguments among file system, registry, socket, and process operations. This provides additional insight into the calling process, such as through bytes written to when using WriteFile or destination key when setting a registry value using RegSetValue. The work of [289] looked at the opcode similarity to detect polymorphic variants. The authors developed a weighted directed graph where the edges were probabilities that one opcode followed the next. They then computed scores between metamorphic viruses and between viruses and benign files and developed a threshold score for maliciousness. This approach performed well since metamorphic viruses are created with a selected few metamorphic engines; therefore, the signatures developed are in fact tracing obfuscation used by a given mutation engine [364, 371].

Another factor to consider when using CFGs is how to establish a comparison between CFGs from malicious and nonmalicious control-flows. The authors in [362] examined the detection of metamorphic code based on a cross-comparison of the control flow graphs of known malware. The authors normalized the code to remove dead or unreachable code, removed common subexpressions, removed dead paths, and analyzed indirect control flow transitions to remove longer chains of control flow and avoid misdirections. The authors recorded a 96.5% true positive rate while producing almost no false positives. The Jaccard similarity matrix was used in [367] between system call subsequences. The cosine similarity is another approach used [372], but all similarity metrics suffer from drawbacks because they are all subject to the selection of subgraph as discussed earlier. Even with reliable subgraphs that perform well on a particular set of malware, the work of [373] demonstrated that 23 algorithmic graph features including betweenness centrality, closeness, degree centrality, density, and number of edges and nodes can be used in adversarial analysis and result in a 100% misclassification rate. Their approached target IoT malware, but android malware, is also an ongoing field of study [374–376]. With all the shortcomings that come with the graph-isomorphism problem, newer advances in this field remove the need for graphs altogether and convert the entire graph into feature vectors [373, 377]. Once features are vectorized, this opens up the door for other machine learning models to act as discriminators for the classification step.

5.4. Natural Language Processing Approaches. The use of natural language processing (NLP) approaches applied to API call sequences was a natural extension to developing models that can predict malicious behavior. Malicious behavior is not simply a product of individual API usage or frequency of APIs, but it is rather a consideration of the pattern in the API usage over time. Similar to how word usage and context can provide an indication of whether or not an email is spam or not, the context of API called in succession can tell you something about malicious intent. This has the effect of being able to attend to different behaviors simultaneously and allows the model to learn what malicious behaviors exist on its own.

Many popularized vectorization techniques used in NLP applications have also been migrated for the purpose of malware research. Two of these techniques were displayed in the work of [378] which used a bag-of-words (BoW) model and term frequency-inverse document frequency (tf-idf). The background specifics of these techniques will be discussed in the next section. Their work created fixed lengthened vectors from behavioral reports produced in virtual machines and automated the feature extraction step. Finally, an ensemble of ML techniques, such as random forest, k-nearest neighbors (k-NN), support vector machine (SVM), and XGBoost, were used, with majority voting summarizing the end predictions over the models. An application that did involve APIs was carried in [1] who looked at both dynamic and static behaviors and hand-crafted groups of signatures based on operation. The authors created 11 different types of malicious operations, spanning from registry operations to device I/O to kernel operations. APIs were converted to semantic blocks which looked at the largest common subsequences between dynamic and static behavior. Following the sequencing, tf-idf was used to vectorize the contribution of each API, with a focus on rarely used APIs that drive malicious behavior. In [160], tf-idf was used to convert the sequence of a unique event name to a representation for a machine learning mode to learn which included both 1-dimensional convolutional neural network (CNN) and long short-term memory (LSTM) architectures. A similar line of work was used in [379] where a LSTM was used to model sequential API usage of 20 thousand malware samples run on a Windows 7 machine using the Cuckoo sandbox. The authors only considered 342 API calls but limited their investigation to those that were used at least 10 times among all samples in the training set. When coupled with tf-idf, this has the effect of focusing more on rarely used APIs, and by limiting the minimum threshold to 10, there are enough training examples for the model to learn the importance of those features. In a more recent work in [380], graph neural networks (GNN) were used to identify dynamic malware execution in a sandbox using the techniques developed in [315] and used in [381]. Windows APIs were vectorized with *n-gram* and td-idf, with malware execution being performed in sandbox snapshots with different benignware excecutions to simulate different potential host environments. The use of GNNs allowed the model to learn patterns in API usage by combining learned patterns from neighboring nodes that represent differnet hierachies in process execution. This has the effect of not only learning the API usage of a single process, but that of all the processes that are daughter or parent processes of any given running process - thereby magnifying the discriminatory power of the model in identifying malicious behavior.

In addition to the form of vectorization, modern NLP models allow the model itself to learn the importance of each word (or API) relative to the context of the surrounding words. For this purpose, word embeddings were developed which can learn the semantic relationship between words and map that relationship to vector space [382]. This has the effect of allowing models that are closely related to have similar cosine-similarity scores. A modest application by [383] used 300-dimensional word embeddings followed by a similarity matrix to cluster malware and benignware using *k*-means. This way, the cluster index was a dense representation of malware and benignware. A more end-to-end approach was used in [381] whereby API stack traces were modeled as an NLP problem. Embedding dimensions of size 50 to 200 were used to map the API stack trace that included APIs that communicated all the way to the kernel. With the use of a transformer architecture which learns latent representation of the sequences, *F*1 scores as high as 96.2% were obtained when considering registry APIs. The authors in [384] looked at developing a semantic transition matrix to segregate API calls which have similar contexts into clusters. This was conducted by capturing the relationship between API calls that represent malware and benignware using

Word2Vec [382], a word embedding technique which has more powerful encoding ability than vanilla word embedding approaches. More powerful encoders translate to better ability to learn context, which was evident in their FP rate of only 1%. A similar use of Word2Vec was followed by an LSTM in [385] to analyze opcodes and API function names. In total, 1369 API function names and opcodes were used, of which 958 were API calls.

Several works have made use of the Windows PE malware API sequence dataset [379], a dataset of over API call sequence extracted from 7017 malicious binaries from 8 malware classes including Adware, Backdoors, Downloaders Droppers, Apyware, Trojans, Viruses, and Worms. For this dataset, [386] achieved poor results with a 0.38 $F1$ score when using a 32-dimensional embedding to represent the API sequences followed by a 2-layer LSTM. Their approach used 342 API calls and discarded those that were used less than 10 times. Similar poor results were obtained in [387] which reported $F1$ scores ranging from 0.33 to 0.72 for the 8 malware types based on a similar LSTM approach. The work of [388] went one step further and compared an LSTM approach to that of a transformer and finally to a bidirectional encoder representation from transformers (BERT). BERT relies on learning latent representations from both directional contexts from before and after sequences, meaning that it does a better job encoding context of the API sequence. In [388], they also used the Windows PE malware dataset and found similar issues classifying the 8 classes with a weighted $F1$ score of 0.51 on their best performing BERT model. One approach that did find success using BERT was that of [389] who implemented *fastText* [390], a text vectorizing technique based on n-gram. While removing redundant API calls, such as NtDelayExecution, accuracies as high as 96.76% using BERT were obtained.

6. Conclusions

This paper provides a systematic review of commonly used obfuscation techniques used by malware variants and mutation engine kits. This survey of the literature touched upon several key indicators of obfuscation employed by malware, which serves to better understand the nature of the reverse-engineering process. Our work makes four core contributions.

We noted the scope of malware and obfuscation worldwide and presented some of the key red-flags noted by antivirus (AV) vendors and researchers. The numbers suggest an aggressive increase in the number of threats and the monetary cost associated with breaches, system intrusions, and downtime. In addition, we discussed some of the string scanning techniques that are still very much in use by AV vendors to this day.

We provided an examination of the popular obfuscation techniques used to translate the opcode sequences of malware into semantic equivalent but different instructions. These techniques have been integrated into popular mutation engines for over a decade now and render much of the reverse-engineering and legacy signature-based techniques obsolete if used effectively. This presents a fundamental problem for researchers and practitioners, but it has led to the field of dynamic analysis which examines the run-time behavior of malicious executables. We also touched upon the structure of metamorphic mutation engines, along with encryption and compression, two very important behaviors that serve as key indicators of maliciousness for a given binary.

We provided a review of popularized malware datasets that are commonly used in malware research. These datasets span applications in mobile malware, intrusion detection, networking, and binaries. We also touched upon some antiemulation and antiarmoring tactics in use by malware to protect from examination under virtualized environments.

Finally, some common approaches to feature analysis are introduced which discusses the various ways Windows APIs are categorized and vectorized to identify malicious binaries, especially in the context of identifying obfuscated malware variants.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon reasonable request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research has been financially supported by Mitacs Accelerate (IT15018) in partnership with Canadian Tire Corporation and is supported by the University of Manitoba.

References

[1] W. Han, J. Xue, Y. Wang, Z. Liu, and Z. Kong, "MalInsight: a systematic profiling based malware detection framework," *Journal of Network and Computer Applications*, vol. 125, pp. 236–250, 2019.

[2] A. A. Gillespie, *Cybercrime: Key Issues and Debates*, Routledge, Milton Park, UK, 2015.

[3] Malwarebytes, *2020 State of Malware Report*, Malwarebytes Labs, Santa Clara, CA, USA, 2020.

[4] Kaspersky, *Kaspersky Security Bulletin 2019*, White Paper, Moscow, Russia, 2019.

[5] E. M. Rudd, A. Rozsa, M. Günther, and T. E. Boult, "A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions," *IEEE Communications Surveys Tutorials*, vol. 19, no. 2, pp. 1145–1172, 2016.

[6] H. P. Enterprise, "HPE security research cyber risk report 2016," Technical Report, Feb. 2016.

[7] Verizon, *2019 Data Breach Investigations Report*, Verizon, New York, NY, USA, 2019.

[8] Ibm, *2019 Cost of Data Breach Report*, IBM, Armonk, NY, USA, 2019.

[9] D. Berard, "DDoS breach costs rise to over $2M for enterprises finds Kaspersky lab report," 2018, https://usa.kaspersky.com/about/press-releases/2018_ddos-breach-costs-rise-to-over-2m-for-enterprises-finds-kaspersky-lab-report.

[10] FireEye, *APT: A Window into Russia's Cyber Espionage Operations?*, FireEye, Milpitas, CA, USA, 2014.

[11] CyberEdge, *Cyberthreat Defense Report*, CyberEdge Group, Annapolis, MD, USA, 2020.

[12] Symantec, *ISTR 20 Internet Security Threat Report*, Symantex Corporation World Headquarters, Mountain View, CA, USA, 2020.

[13] Cso, *2018 U.S. State of Cybercrime," IDG Communications*, Survey, Framingham, MA, USA, 2018.

[14] PwC, *Key Findings from the Global State of Information Security Survey 2018*, International Data Group Inc, London, UK, 2017.

[15] Crowdstrike, "Crowdstrike global threat report 2020," 2020, https://go.crowdstrike.com/rs/281-OBQ-266/images/Report2020CrowdStrikeGlobalThreatReport.pdf.

[16] Mimecast, *The State of Email Security Report*, Mimecast, London, UK, 2019.

[17] Malwarebytes, *White Hat Black Hat and the Emergence of the Gray Hat: The True Costs of Cybercrime*, Osterman Research Inc, Black Diamond, WA, USA, 2019.

[18] B. Wanswett and H. K. Kalita, "The threat of obfuscated zero day polymorphic malwares: an analysis," in *Proceedings of the 2015 International Conference on Computational Intelligence and Communication Networks (CICN)*, pp. 1188–1193, Jabalpur, India, December 2015.

[19] N. Innab, E. Alomairy, and L. Alsheddi, "Hybrid system between anomaly based detection system and honeypot to detect zero day attack," 2018, https://www.semanticscholar.org/paper/Hybrid-System-Between-Anomaly-Based-Detection-and-Innab-Alomairy/5ca4f26c185280593e6b5a287cbe809f53610077.

[20] F. Azzedin, H. Suwad, and Z. Alyafeai, "Countermeasureing zero day attacks: asset-based approach," in *Proceedings of the 2017 International Conference on High Performance Computing & Simulation (HPCS)*, Genoa, Italy, July 2017.

[21] A. Almomani, B. B. Gupta, T.-C. Wan, A. Altaher, and S. Manickam, "Phishing dynamic evolving neural fuzzy framework for online detection zero-day phishing email," 2013, http://arxiv.org/abs/1302.0629.

[22] Esg, "2020 technology spending intentions survey," 2020, https://www.esg-global.com/research/esg-master-survey-results-2020-technology-spending-intentions-survey.

[23] R. Kaur and M. Singh, "Efficient hybrid technique for detecting zero-day polymorphic worms," in *Proceedings of the 2014 IEEE International Advance Computing Conference (IACC)*, pp. 95–100, Gurgaon, India, February 2014.

[24] P. Ren, Y. Xiao, X. Chang et al., "A Survey of Deep Active Learning," 2021, http://arxiv.org/abs/2009.00236.

[25] Webroot, "Webroot Threat Report," 2020, https://mypage.webroot.com/rs/557-FSI-195/images/2020.

[26] H. Carvey, "Chapter 6 malware detection," in *Windows Forensic Analysis Toolkit*, H. Carvey, Ed., Syngress, Boston, MA, USA, 4th edition, 2014.

[27] M. Gregg, *Build Your Own Security Lab: A Field Guide for Network Testing*, John Wiley & Sons, New York, NY, USA, 2010.

[28] J. Aycock, *Computer Viruses and Malware*, Springer, Berlin, Germany, 2006.

[29] E. Seamans and T. Alexander, "Fast Virus Signature Matching on the GPU," 2007, https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-35-fast-virus-signature-matching-gpu.

[30] D. Mellado, *IT security governance innovations: theory and research: theory and research*, IGI Global, Pennsylvania, PA, USA, 2012.

[31] D. K. Mahawer and A. Nagaraju, "Metamorphic malware detection using base malware identification approach," *Security and Communication Networks*, vol. 7, no. 11, pp. 1719–1733, 2014.

[32] S. Raghu and V. Mohit, *Cyber Security, Cyber Crime and Cyber Forensics: Applications and Perspectives: Applications and Perspectives*, Idea Group Inc (IGI), New York, NY, USA, 2010.

[33] S. Yu, W. Lou, and K. Ren, "Chapter 15 data security in cloud computing," in *Handbook on Securing Cyber-Physical Critical Infrastructure*, S. K. Das, K. Kant, and N. Zhang, Eds., pp. 389–410, Morgan Kaufmann, Boston, MA, USA, 2012.

[34] S. K. Shivakumar, "Securing enterprise web application," in *Architecting High Performing, Scalable and Available Enterprise Web Applications*, S. K. Shivakumar, Ed., Morgan Kaufmann, Boston, MA, USA, 2015.

[35] E. A. Daoud, I. H. Jebril, and B. Zaqaibeh, "Computer virus strategies and detection methods," *International Journal of Open Problems in Computer Science and Mathematics*, vol. 1, no. 2, p. 8, 2008.

[36] McAfee, "McA. Network Security Platform 9.1.x," 2017, https://docs.mcafee.com/bundle/network-security-platform-9.1.x-product-guide/page/GUID-D00D67EA-5EAE-4461-ACFC-A2B2A78C3E50.html.

[37] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Deceiving End-To-End Deep Learning Malware Detectors Using Adversarial Examples," 2018, http://arxiv.org/abs/1802.04528.

[38] A. Rohan, K. Basu, and R. Karri, "Can Monitoring System State + Counting Custom Instruction Sequences Aid Malware Detection?" in *Proceedings of the 2019 IEEE 28th Asian Test Symposium (ATS)*, Kolkata, India, December 2019.

[39] D. Lin and M. Stamp, "Hunting for undetectable metamorphic viruses," *Journal in Computer Virology*, vol. 7, no. 3, pp. 201–214, 2011.

[40] I. You and K. Yim, "Malware Obfuscation Techniques: A Brief Survey," in 2010 International Conference on Broadband," in *Proceedings of the Wireless Computing, Communication and Applications*, pp. 297–300, Fukuoka, Japan, November 2010.

[41] D. C. Park, H. Khan, and B. Yener, "Generation & Evaluation of Adversarial Examples for Malware Obfuscation," in *Proceedings of the 2019 18th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Boca Raton, FL, USA, December 2019.

[42] L. Durfina and D. Kolar, "C source code obfuscator," *Kybernetika*, vol. 48, no. 3, pp. 494–501, 2012.

[43] P. Szor, *The Art of Computer Virus Research and Defense*, Addison-Wesley, Upper Saddle River, NJ, USA, 2005.

[44] B. J. Kumar, H. Naveen, B. P. Kumar, S. S. Sharma, and J. Villegas, "Logistic regression for polymorphic malware detection using ANOVA F-test," in *Proceedings of the 2017 International Conference on Innovations in Information,*

Embedded and Communication Systems (ICIIECS), pp. 1–5, Coimbatore, India, March 2017.

[45] A. H. Toderici and M. Stamp, "Chi-squared distance and metamorphic virus detection," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 1, pp. 1–14, 2013.

[46] X. Li, P. K. K. Loh, and F. Tan, "Mechanisms of Polymorphic and Metamorphic Viruses," in *Proceedings of the 2011 European Intelligence and Security Informatics Conference*, Athens, Greece, September 2011.

[47] A. H. Toderici, *Chi-Squared Distance and Metamorphic Virus Detection*, San Jose State University, San Jose, CA, USA, 2012.

[48] C. I. V. Barría, D. G. Cordero, C. Cubillos, and R. Osses, "Obfuscation procedure based in dead code insertion into crypter," in *Proceedings of the 6th International Conference on Computers Communications and Control (ICCCC)*, Oradea, Romania, May 2016.

[49] B. B. Rad, M. Masrom, and S. Ibrahim, "Evolution of computer virus concealment and anti-virus techniques," *A Short Survey*, vol. 7, no. 6, p. 9, 2010.

[50] M. Patel, "Similarity Tests for Metamorphic Virus Detection," 2011, https://scholarworks.sjsu.edu/etd_projects/175.

[51] W. Wong, *Analysis and Detection of Metamorphic Computer Viruses*, San Jose State University, San Jose, CA, USA, 2006.

[52] M. R. Chouchane and A. Lakhotia, *Using Engine Signature to Detect Metamorphic Malware*, WORM, Alexandria, VI, USA, 2006.

[53] R. Jin, Q. Wei, P. Yang, and Q. Wang, "Normalization towards instruction substitution metamorphism based on standard instruction set," in *Proceedings of the 2007 International Conference on Computational Intelligence and Security Workshops (CISW 2007)*, pp. 795–798, Harbin, China, December 2007.

[54] Z. Yu-jia and P. Jian-min, "A new compile-time obfuscation scheme for software protection," in *Proccedings of the 2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, Chengdu, China, October 2016.

[55] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM Software Protection for the Masses," in *Proceedings of the 2015 IEEE/ACM 1st International Workshop on Software Protection*, Florence, Italy, May 2015.

[56] A. Kayem, *Information security in diverse computing environments*, IGI Global, New York, NY, USA, 2014.

[57] S. Venkatachalam and M. Stamp, *Detecting Undetectable Metamorphic Viruses*, San Jose State University, San Jose, CA, USA, 2011.

[58] D. Baysa, R. M. Low, and M. Stamp, "Structural entropy and metamorphic malware," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 4, pp. 179–192, 2013.

[59] A. Sharma and S. K. Sahay, "Evolution and detection of polymorphic and metamorphic malwares: a survey," *International Journal of Critical Accounting*, vol. 90, no. 2, pp. 7–11, 2014.

[60] B. B. Rad, M. Masrom, and S. Ibrahim, "Camouflage in Malware: From Encryption to Metamorphism," 2012, https://www.semanticscholar.org.

[61] P. Beaucamps, "Advanced polymorphic techniques," *International Journal of Computer and Information Engineering*, vol. 1, no. 10, p. 12, 2007.

[62] P. Szor and P. Ferrie, *Hunting for Metamorphic*, White Paper, Cupertino, CA, USA, 2001.

[63] D. Spinellis, "Reliable identification of bounded-length viruses is NP-complete," *IEEE Transactions on Information Theory*, vol. 49, no. 1, pp. 280–284, 2003.

[64] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology*, vol. 4, no. 3, pp. 211–220, 2008.

[65] M. Stamp, *Information Security Principles and Practice*, John Wiley & Sons, New York, NY, USA, 2nd edition, 2011.

[66] M. Tuba, S. Akashe, and A. Joshi, *Information and Communication Technology for Sustainable Development: Proceedings of ICT4SD 2018*, Springer, New York, NY, USA, 2019.

[67] M. Saleh, A. Mohamed, and A. Nabi, "Eigenviruses for metamorphic virus recognition," *IET Information Security*, vol. 5, no. 4, pp. 191–198, 2011.

[68] R. G. Fiñones and R. Fernandez, *Solving the Metamorphic Puzzle*, https://www.virusbulletin.com/virusbulletin/2006/03/solving-metamorphic-puzzle, 2006.

[69] S. M. Sridhara, *Metamorphic Worm That Carries Its Own Morphing Engine*, San Jose State University, San Jose, CA, USA, Apr. 2012, https://scholarworks.sjsu.edu/etd_projects/240.

[70] S. Deshpande, Y. Park, and M. Stamp, "Eigenvalue analysis for metamorphic detection," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 1, pp. 53–65, 2014.

[71] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A Survey on Heuristic Malware Detection Techniques," in *Proceedings of the 5th Conference on Information and Knowledge Technology*, Shiraz, Iran, May, 2013.

[72] P. V. Zbitskiy, "Code mutation techniques by means of formal grammars and automatons," *Journal in Computer Virology*, vol. 5, no. 3, pp. 199–207, Aug. 2009.

[73] T. Tamboli, T. H. Austin, and M. Stamp, "Metamorphic code generation from LLVM bytecode," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 177–187, Aug. 2014.

[74] S. Madenur Sridhara and M. Stamp, "Metamorphic worm that carries its own morphing engine," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 2, pp. 49–58, May 2013.

[75] D. Harley, R. Slade, and U. Gattiker, *Viruses Revealed*, McGraw Hill Professional, New York, NY, USA, Dec. 2002.

[76] Ec-Council, *Ethical Hacking and Countermeasures: Threats and Defense Mechanisms*, Nelson Education, Ontario, Canada, Sep. 2009.

[77] H. Team, "Hackedteam/core-packer," Jun. 2020, https://github.com/hackedteam/core-packer.

[78] J. L. Mauri, S. M. Thampi, D. B. Rawat, and D. Jin, *Security in Computing and Communications: Second International Symposium, SSCC 2014, Delhi, India, September 24-27, 2014. Proceedings*, Springer, Berlin, Germany, Aug. 2014.

[79] E. Filiol, "Strong cryptography armoured computer viruses forbidding code analysis: the bradley virus," 2004, https://hal.inria.fr/inria-00070748inria-00070748, Report.

[80] J. Al-Enezi, M. Abbod, and S. Alsharhan, "Artificial Immune Systems – Models, Algorithms and Applications," *Artificial Immune Systems*, vol. 14, 2010.

[81] D. M. M. K. Al-Anezi, "Generic packing detection using several complexity analysis for accurate malware detection," *International Journal of Advanced Computer Science and Applications*, vol. 5, no. 1, 2014.

[82] X. Ma, Q. Biao, W. Yang, and J. Jiang, "Using multi-features to reduce false positive in malware classification," in *Proceedings of the 2016 IEEE Information Technology,*

Networking, Electronic and Automation Control Conference, pp. 361–365, Chongqing, China, May 2016.

[83] A. Liska and T. Gallo, *Ransomware: Defending against Digital Extortion*, O'Reilly Media, Inc, Sebastopol, CA, USA, 2016.

[84] H. Bos, F. Monrose, and G. Blanc, "Research in attacks, intrusions, and defenses," in *Proceedings of the 18th international symposium, RAID 2015*, Springer, kyoto, Japan, November, Oct. 2015.

[85] M. H. Nguyen, D. L. Nguyen, X. M. Nguyen, and T. T. Quan, "Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning," *Computers and Security*, vol. 76, pp. 128–155, Jul. 2018.

[86] K. Brezinski and K. Ferens, "Complexity-based convolutional neural network for malware classification," in *2020 International Conference on Computational Science and Computational Intelligence*, Las Vegas, USA, October 2020.

[87] I. Sorokin, "Comparing files using structural entropy," *Journal in Computer Virology*, vol. 7, no. 4, p. 259, Jun. 2011.

[88] F. N. Ducau, E. M. Rudd, T. M. Heppner, A. Long, and K. Berlin, "Automatic malware description via attribute tagging and similarity embedding," 2020, http://arxiv.org/abs/1905.06262.

[89] R. Lippmann, D. Fried, I. Graf et al., "Evaluating intrusion detection systems: the 1998 DARPA off-line intrusion detection evaluation," in *Proceedings of the DARPA Information Survivability Conference and Exposition. DISCEX'00*, pp. 12–26, Hilton Head, SC, USA, January. 2000.

[90] W. Lee and S. J. Stolfo, "A framework for constructing features and models for intrusion detection systems," *ACM Transactions on Information and System Security*, vol. 3, no. 4, pp. 227–261, 2000.

[91] A. Özgür and H. Erdem, "A review of KDD99 dataset usage in intrusion detection and machine learning between 2010 and 2015," *PeerJ*, vol. 4, 2016.

[92] M. V. Mahoney and P. K. Chan, "An analysis of the 1999 DARPA/lincoln laboratory evaluation data for network anomaly detection," in *Recent Advances in Intrusion Detection*, G. Goos, J. Hartmanis, J. van Leeuwen, G. Vigna, C. Kruegel, and E. Jonsson, Eds., vol. 2820, pp. 220–237, Springer, Berlin, Germany, 2003.

[93] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," in *Proceedings of the 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, pp. 1–6, IEEE, Ottawa, Canada, July. 2009.

[94] J. McHugh, "Testing Intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory," *ACM Transactions on Information and System Security*, vol. 3, no. 4, pp. 262–294, 2000.

[95] N. Moustafa and J. Slay, "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," in *Proceedings of the 2015 Military Communications and Information Systems Conference (MilCIS)*, pp. 1–6, Canberra, Australia, November. 2015.

[96] W. Bhaya and M. E. Manaa, "A proactive DDoS attack detection approach using data mining cluster analysis," *Journal of Next Generation Information Technology*, vol. 5, no. 4, p. 12, 2014.

[97] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *Computers and Security*, vol. 31, no. 3, pp. 357–374, May 2012.

[98] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *Proceedings of the 4th International Conference on Information Systems Security and Privacy*, Kenitra, Morocco, April, 2018.

[99] M. Ring, S. Wunderlich, D. Scheuring, D. Landes, and A. Hotho, "A survey of network-based intrusion detection data sets," *Computers and Security*, vol. 86, pp. 147–167, Sep. 2019, http://arxiv.org/abs/1903.02460.

[100] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: visualization and automatic classification," in *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, pp. 1–7, Association for Computing Machinery, Pittsburgh, PA, USA, July. 2011.

[101] D. Gibert, C. Mateu, and J. Planes, "An end-to-end deep learning architecture for classification of Malware's binary content," in *Artificial Neural Networks and Machine Learning – ICANN 2018, Ser. Lecture Notes in Computer Science*, V. Kuringrková, Y. Manolopoulos, B. Hammer, L. Iliadis, and I. Maglogiannis, Eds., pp. 383–391, Springer International Publishing, Berlin, Germany, 2018.

[102] N. Bhodia, P. Prajapati, F. Di Troia, and M. Stamp, "Transfer learning for image-based malware classification," Jan. 2019, http://arxiv.org/abs/1903.11551.

[103] T. K. Tran, H. Sato, and M. Kubo, "Image-based unknown malware classification with few-shot learning models," in *Proceedings of the 2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*, pp. 401–407, Nagasaki, Japan, November. 2019.

[104] M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang, and F. Iqbal, "Malware classification with deep convolutional neural networks," in *Proceedings of the 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, Paris, France, February, 2018.

[105] S. Akarsh, P. Poornachandran, V. K. Menon, and K. P. Soman, "A detailed investigation and analysis of deep learning architectures and visualization techniques for malware family identification," in *Cybersecurity and Secure Information Systems: Challenges and Solutions in Smart Environments, Ser. Advanced Sciences and Technologies for Security Applications*, A. E. Hassanien and M. Elhoseny, Eds., pp. 241–286, Springer International Publishing, Berlin, Germany, 2019.

[106] S. Akarsh, K. Simran, P. Poornachandran, V. K. Menon, and K. Soman, "Deep learning framework and visualization for malware classification," in *Proceedings of the 2019 5th International Conference on Advanced Computing Communication Systems (ICACCS)*, pp. 1059–1063, Coimbatore, India, March, 2019.

[107] F. Abdullayeva, "Malware detection in cloud computing using an image visualization technique," in *Proceedings of the 2019 IEEE 13th International Conference on Application of Information and Communication Technologies (AICT)*, pp. 1–5, Baku, Azerbaijan, October, 2019.

[108] B. N. Vi, H. Noi Nguyen, N. T. Nguyen, and C. Truong Tran, "Adversarial examples against image-based malware classification systems," in *Proceedings of the 2019 11th International Conference on Knowledge and Systems Engineering (KSE)*, pp. 1–5, Da Nang, Vietnam, October, 2019.

[109] W. W. Lo, X. Yang, and Y. Wang, "An xception convolutional neural network for malware classification with transfer learning," in *Proceedings of the 2019 10th IFIP international conference on new technologies, mobility and security (NTMS)*, pp. 1–5, Canary Islands, Spain, June, 2019.

[110] A. Nappa, M. Z. Rafique, and J. Caballero, "Driving in the cloud: an analysis of drive-by download operations and abuse reporting," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, D. Hutchison, T. Kanade, J. Kittler et al., Eds., vol. 7967, pp. 1–20, Springer, Berlin, Germany, 2013.

[111] P. Kotzias, S. Matic, R. Rivera, and J. Caballero, "Certified pup: abuse in authenticode code signing," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, CO, USA, October, 2015.

[112] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVclass: a tool for massive malware labeling," in *Research in Attacks, Intrusions, and Defenses*, F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds., vol. 9854, pp. 230–253, Springer International Publishing, Berlin, Germany, 2016.

[113] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: efficient and explainable detection ofandroid malware in your pocket," Georg-August Institute of Computer Science, Technical Report, 2013.

[114] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: having a deeper look into android applications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing- SAC '13*, p. 1808, ACM Press, Coimbra, Portugal, June, 2013.

[115] A. H. Lashkari, A. F. A. Kadir, H. Gonzalez, K. F. Mbah, and A. A. Ghorbani, "Towards a network-based framework for android malware detection and characterization," in *Proceedings of the 2017 15th Annual Conference on Privacy, Security and Trust (PST)*, Calgary, Canada, August, 2017.

[116] M. Murtaz, H. Azwar, S. B. Ali, and S. Rehman, "A framework for android malware detection and classification," in *Proceedings of the 2018 IEEE 5th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*, pp. 1–5, Bangkok, Thailand, November, 2018.

[117] H. K. Kim, "Iot network intrusion dataset," 2019, https://ieee-dataport.org/open-access/iot-network-intrusion-dataset.

[118] C. Goutte, *Advances in Artificial Intelligence*, Springer Nature, Berlin, Germany, 1987.

[119] I. Ullah and Q. H. Mahmoud, "A scheme for generating a dataset for anomalous activity detection in IoT networks," in *Advances in Artificial Intelligence, Ser. Lecture Notes in Computer Science*, C. Goutte and X. Zhu, Eds., pp. 508–520, Springer International Publishing, Berlin, Germany, 2020.

[120] Github, "CMP," 2020, https://github.com/urwithajit9/ClaMP.

[121] H. S. Anderson and P. Roth, "EMBER: an open dataset for training static pe malware machine learning models," 2018, http://arxiv.org/abs/1804.04637.

[122] endgameinc, "Ember," 2020, https://github.com/endgameinc/ember.

[123] G. Creech, *Developing a high-accuracy cross platformhost-based intrusion detection system capableof reliably detecting zero-day attacks*, University of South Wales, Australian Defence Force Academy, Ph.D. dissertation, 2014.

[124] G. Creech and J. Hu, "Generation of a new IDS test dataset: time to retire the KDD collection," in *Proceedings of the 2013 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 4487–4492, Shanghai, China, April. 2013.

[125] G. Creech and J. Hu, "A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 807–819, 2014.

[126] CERT, *CERT Insider Threat Center*, Carnegie Mellon University Software Engineering Institute, Pittsburgh, PA, USA, 2017.

[127] J. Glasser and B. Lindauer, "Bridging the gap: a pragmatic approach to generating insider threat data," in *Proceedings of the 2013 IEEE Security and Privacy Workshops*, pp. 98–104, IEEE, San Francisco, CA, USA, May 2013.

[128] P. S. Bhattacharjee, A. K. Md Fujail, and S. A. Begum, "A comparison of intrusion detection by K-means and fuzzy C-means clustering algorithm over the NSL-KDD dataset," in *Proceedings of the 2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, pp. 1–6, Coimbatore, India, December, 2017.

[129] B. Ingre and A. Yadav, "Performance analysis of NSL-KDD dataset using ANN," in *Proceedings of the 2015 International Conference on Signal Processing and Communication Engineering Systems*, pp. 92–96, Guntur, India, January, 2015.

[130] L. Hakim, R. Fatma, and Novriandi, "Influence analysis of feature selection to network intrusion detection system performance using NSL-KDD dataset," in *Proceedings of the 2019 International Conference on Computer Science, Information Technology, and Electrical Engineering (ICOMITEE)*, pp. 217–220, Jember, Indonesia, October, 2019.

[131] N. Kunhare and R. Tiwari, "Study of the attributes using four class labels on kdd99 and nsl-kdd datasets with machine learning techniques," in *Proceedings of the 2018 8th International Conference on Communication Systems and Network Technologies (CSNT)*, pp. 127–131, Bhopal, India, November, 2018.

[132] G. Meena and R. R. Choudhary, "A review paper on IDS classification using KDD 99 and NSL KDD dataset in WEKA," in *Proceedings of the 2017 International Conference on Computer, Communications and Electronics (Comptelix)*, pp. 553–558, Jaipur, India, July. 2017.

[133] R. Thomas and D. Pavithran, "A survey of intrusion detection models based on NSL-KDD data set," in *Proceedings of the 2018 Fifth HCT Information Technology Trends (ITT)*, pp. 286–291, Dubai, United Arab Emirates, November, 2018.

[134] C. Zhang, F. Ruan, L. Yin, X. Chen, L. Zhai, and F. Liu, "A deep learning approach for network intrusion detection based on NSL-KDD dataset," in *Proceedings of the 2019 IEEE 13th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, pp. 41–45, Xiamen, China, October, 2019.

[135] H. Benaddi, K. Ibrahimi, and A. Benslimane, "Improving the intrusion detection system for NSL-KDD dataset based on PCA-fuzzy clustering-KNN," in *Proceedings of the 2018 6th International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pp. 1–6, Marrakesh, Morocco, October, 2018.

[136] K. Singh and K. J. Mathai, "Performance comparison of intrusion detection system between deep belief network (DBN)algorithm and state preserving extreme learning machine (SPELM) algorithm," in *Proceedings of the 2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pp. 1–7, Coimbatore, India, February, 2019.

[137] L. A. Álvarez Almeida and J. Carlos Martinez Santos, "Evaluating features selection on NSL-KDD data-set to train a support vector machine-based intrusion detection system,"

in *Proceedings of the 2019 IEEE Colombian Conference on Applications in Computational Intelligence (ColCACI)*, pp. 1–5, Barranquilla, Colombia, June, 2019.

[138] N. Paulauskas and J. Auskalnis, "Analysis of data preprocessing influence on intrusion detection using NSL-KDD dataset," in *Proceedings of the 2017 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pp. 1–5, Vilnius, Lithuania, April, 2017.

[139] A. R. Yusof, N. I. Udzir, A. Selamat, H. Hamdan, and M. T. Abdullah, "Adaptive feature selection for denial of services (DoS) attack," in *Proceedings of the 2017 IEEE Conference on Application, Information and Network Security (AINS)*, pp. 81–84, Miri, Malaysia, November, 2017.

[140] S. Rodda and U. S. R. Erothi, "Class imbalance problem in the network intrusion detection systems," in *Proceedings of the 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, pp. 2685–2688, Chennai, India, March, 2016.

[141] S. Samdani and S. Shukla, "A novel technique for converting nominal attributes to numeric attributes for intrusion detection," in *Proceedings of the 2017 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pp. 1–5, Delhi, India, July. 2017.

[142] M. A. Jabbar and S. Samreen, "Intelligent network intrusion detection using alternating decision trees," in *Proceedings of the 2016 International Conference on Circuits, Controls, Communications and Computing (I4C)*, pp. 1–6, Bangalore, India, October, 2016.

[143] Z. Ye and Y. Yu, "Network intrusion classification based on extreme learning machine," in *Proceedings of the 2015 IEEE International Conference on Information and Automation*, pp. 1642–1647, Lijiang, China, August, 2015.

[144] F. Yihunie, E. Abdelfattah, and A. Regmi, "Applying machine learning to anomaly-based intrusion detection systems," in *Proceedings of the 2019 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pp. 1–5, New York, NY, USA, May, 2019.

[145] T. S. Kala and A. Christy, "An intrusion detection system using opposition based particle swarm optimization algorithm and PNN," in *Proceedings of the 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pp. 184–188, Faridabad, India, February, 2019.

[146] Z. Xiaofeng and H. Xiaohong, "Research on intrusion detection based on improved combination of K-means and multi-level SVM," in *Proceedings of the 2017 IEEE 17th International Conference on Communication Technology (ICCT)*, pp. 2042–2045, Chengdu, China, October, 2017.

[147] A. Gül and E. Adalı, "A feature selection algorithm for IDS," in *Proceedings of the 2017 International Conference on Computer Science and Engineering (UBMK)*, pp. 816–820, Antalya, Turkey, October, 2017.

[148] Z. Chen, C. K. Yeo, B. S. Lee, and C. T. Lau, "Autoencoder-based network anomaly detection," in *Proceedings of the 2018 Wireless Telecommunications Symposium (WTS)*, pp. 1–5, Phoenix, AZ, USA, April, 2018.

[149] J.-H. Woo, J.-Y. Song, and Y.-J. Choi, "Performance enhancement of deep neural network using feature selection and preprocessing for intrusion detection," in *Proceedings of the 2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pp. 415–417, Okinawa, Japan, February, 2019.

[150] I. Mikhail, K. S. Kh, A. Klimenko, and E. Balenko, "Neural nets to detect abnormal traffic in communication networks," in *Proceedings of the 2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*, pp. 1–3, Vladivostok, Russia, October, 2019.

[151] E. Zyad, C. Khalid, and B. Mohammed, "An effective network intrusion detection based on truncated mean LDA," in *Proceedings of the 2017 International Conference on Electrical and Information Technologies (ICEIT)*, pp. 1–5, Rabat, Morocco, November, 2017.

[152] J. Gao, S. Chai, C. Zhang, B. Zhang, and L. Cui, "A novel intrusion detection system based on extreme machine learning and multi-voting technology," in *Proceedings of the 2019 Chinese Control Conference (CCC)*, pp. 8909–8914, Guangzhou, China, July, 2019.

[153] M. Al-Hawawreh, N. Moustafa, and E. Sitnikova, "Identification of malicious activities in industrial internet of things based on deep learning models," *Journal of Information Security and Applications*, vol. 41, pp. 1–11, 2018.

[154] M. C. Belavagi and B. Muniyal, "Game theoretic approach towards intrusion detection," in *Proceedings of the 2016 International Conference on Inventive Computation Technologies (ICICT)*, vol. 1, pp. 1–5, Coimbatore, India, August, 2016.

[155] S. Naseer, Y. Saleem, S. Khalid et al., "Enhanced network anomaly detection based on deep neural networks," *IEEE Access*, vol. 6, p. 48246, 2018.

[156] S. K. Sahu, S. Sarangi, and S. K. Jena, "A detail analysis on intrusion detection datasets," in *Proceedings of the 2014 IEEE International Advance Computing Conference (IACC)*, pp. 1348–1353, Gurgaon, India, February, 2014.

[157] R. Patgiri, U. Varshney, T. Akutota, and R. Kunde, "An investigation on intrusion detection system using machine learning," in *Proceedings of the 2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1684–1691, Bangalore, India, November, 2018.

[158] P. Wu, H. Guo, and R. Buckland, "A transfer learning approach for network intrusion detection," in *Proceedings of the 2019 IEEE 4th International Conference on Big Data Analytics (ICBDA)*, pp. 281–285, Suzhou, China, March, 2019.

[159] S. Dong and B. Zhang, "SVDD-Based network traffic anomaly detection method with high robustness," in *Proceedings of the 2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, pp. 1522–1526, Chengdu, China, December, 2019.

[160] J. Lee, J. Kim, I. Kim, and K. Han, "Cyber threat detection based on artificial neural networks using event profiles," *IEEE Access*, vol. 7, pp. 165607–165626, 2019.

[161] M. Essid and F. Jemili, "Combining intrusion detection datasets using MapReduce," in *Proceedings of the 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 004724–004728, Budapest, Hungary, October, 2016.

[162] Z. Li and G. Yan, "A spark platform-based intrusion detection system by combining MSMOTE and improved adaboost algorithms," in *Proceedings of the 2018 IEEE 9th International Conference on Software Engineering and Service Science*, pp. 1046–1049, Beijing, China, November, 2018.

[163] K. Ibrahimi and M. Ouaddane, "Management of intrusion detection systems based-KDD99: analysis with LDA and PCA," in *Proceedings of the 2017 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pp. 1–6, Rabat, Morocco, November, 2017.

[164] P. Singh and A. Tiwari, "An efficient approach for intrusion detection in reduced features of KDD99 using ID3 and

classification with KNNGA," in *Proceedings of the 2015 Second International Conference on Advances in Computing and Communication Engineering*, pp. 445–452, Dehradun, India, May, 2015.

[165] J. Jiang, X. Jing, B. Lv, and M. Li, "A novel multi-classification intrusion detection model based on relevance vector machine," in *Proceedings of the 2015 11th International Conference on Computational Intelligence and Security (CIS)*, pp. 303–307, Shenzhen, China, December, 2015.

[166] H. A. Karande and S. S. Gupta, "Ontology based intrusion detection system for web application security," in *Proceedings of the 2015 International Conference on Communication Networks (ICCN)*, pp. 228–232, Gwalior, India, November, 2015.

[167] T. Chandak, C. Ghorpade, and S. Shukla, "Effective analysis of feature selection algorithms for network based intrusion detection system," in *Proceedings of the 2019 IEEE Bombay Section Signature Conference (IBSSC)*, pp. 1–5, Mumbai, India, July. 2019.

[168] R. Uikey and M. Gyanchandani, "Survey on classification techniques applied to intrusion detection system and its comparative analysis," in *Proceedings of the 2019 International Conference on Communication and Electronics Systems (ICCES)*, pp. 1451–1456, Coimbatore, India, July. 2019.

[169] N. Elisa, L. Yang, and N. Naik, "Dendritic cell algorithm with optimised parameters using genetic algorithm," in *Proceedings of the 2018 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, Rio de Janeiro, Brazil, July. 2018.

[170] A. A. Aburomman and M. Bin Ibne Reaz, "Ensemble of binary SVM classifiers based on PCA and LDA feature extraction for intrusion detection," in *Proceedings of the 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pp. 636–640, Xi'an, China, October, 2016.

[171] Y. Danane and T. Parvat, "Intrusion detection system using fuzzy genetic algorithm," in *Proceedings of the 2015 International Conference on Pervasive Computing (ICPC)*, pp. 1–5, Pune, India, January, 2015.

[172] T. Bjerkestrand, D. Tsaptsinos, and E. Pfluegel, "An evaluation of feature selection and reduction algorithms for network IDS data," in *Proceedings of the 2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, pp. 1-2, London, UK, June. 2015.

[173] T. Mehmood and H. B. M. Rais, "Machine learning algorithms in context of intrusion detection," in *Proceedings of the 2016 3rd International Conference on Computer and Information Sciences (ICCOINS)*, pp. 369–373, Kuala Lumpur, Malaysia, August, 2016.

[174] R. U. Khan, X. Zhang, M. Alazab, and R. Kumar, "An improved convolutional neural network model for intrusion detection in networks," in *Proceedings of the 2019 Cybersecurity and Cyberforensics Conference (CCC)*, pp. 74–77, Melbourne, Australia, May 2019.

[175] J. Song, Z. Zhu, and C. Price, "A new evidence accumulation method with hierarchical clustering," in *Proceedings of the 2016 IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pp. 122–126, Chengdu, China, July, 2016.

[176] H. Cui, "Research on eliminating abnormal big data based on PSO-SVM," in *Proceedings of the 2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pp. 2460–2463, Chongqing, China, October, 2018.

[177] A. Nagisetty and G. P. Gupta, "Framework for detection of malicious activities in IoT networks using keras deep learning library," in *Proceedings of the 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*, pp. 633–637, Erode, India, March, 2019.

[178] D. Shao-Bo, "Intrusion feature selection method based on neighborhood distance," in *Proceedings of the 2017 International Conference on Computer Systems, Electronics and Control (ICCSEC)*, pp. 748–751, Dalian, China, December, 2017.

[179] S. M. Almansob and S. S. Lomte, "Addressing challenges for intrusion detection system using naive Bayes and PCA algorithm," in *Proceedings of the 2017 2nd International Conference for Convergence in Technology (I2CT)*, pp. 565–568, Mumbai, India, April, 2017.

[180] A. Chandra, S. K. Khatri, and R. Simon, "Filter-based attribute selection approach for intrusion detection using k-means clustering and sequential minimal optimization techniq," in *Proceedings of the 2019 Amity International Conference on Artificial Intelligence (AICAI)*, pp. 740–745, Dubai, United Arab Emirates, February, 2019.

[181] Y. Jia, M. Wang, and Y. Wang, "Network intrusion detection algorithm based on deep neural network," *IET Information Security*, vol. 13, no. 1, pp. 48–53, 2019.

[182] T. Mehmood and H. B. M. Rais, "SVM for network anomaly detection using ACO feature subset," in *Proceedings of the 2015 International Symposium on Mathematical Sciences and Computing Research (iSMSC)*, pp. 121–126, Ipoh, Malaysia, May, 2015.

[183] X. Zhao, G. Wang, and Z. Li, "Unsupervised network anomaly detection based on abnormality weights and subspace clustering," in *Proceedings of the 2016 Sixth International Conference on Information Science and Technology (ICIST)*, pp. 482–486, Dalian, China, May, 2016.

[184] A. Alazab, M. Hobbs, J. Abawajy, A. Khraisat, and M. Alazab, "Using response action with intelligent intrusion detection and prevention system against web application malware," *Information Management and Computer Security*, vol. 22, no. 5, pp. 431–449, 2014.

[185] S. Duque and M. N. B. Omar, "Using data mining algorithms for developing a model for intrusion detection system (IDS)," *Procedia Computer Science*, vol. 61, pp. 46–51, 2015.

[186] F. Harrou, B. Bouyeddou, Y. Sun, and B. Kadri, "A method to detect DOS and DDOS attacks based on generalized likelihood ratio test," in *Proceedings of the 2018 International Conference on Applied Smart Systems (ICASS)*, pp. 1–6, Medea, Algeria, November, 2018.

[187] F. Harrou, B. Bouyeddou, Y. Sun, and B. Kadri, "Detecting cyber-attacks using a CRPS-based monitoring approach," in *Proceedings of the 2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 618–622, Bangalore, India, November, 2018.

[188] B. Bouyeddou, F. Harrou, Y. Sun, and B. Kadri, "An effective network intrusion detection using hellinger distance-based monitoring mechanism," in *Proceedings of the 2018 International Conference on Applied Smart Systems (ICASS)*, pp. 1–6, Medea, Algeria, November, 2018.

[189] B. Bouyeddou, F. Harrou, Y. Sun, and B. Kadri, "Detection of smurf flooding attacks using Kullback-Leibler-based scheme," in *Proceedings of the 2018 4th International Conference on Computer and Technology Applications (ICCTA)*, pp. 11–15, Istanbul, Turkey, May, 2018.

[190] G. Fan and Y. Min, "Automatic attack scenario construction by mining meta-alert sequences," in *Proceedings of the 2009 Second Pacific-Asia Conference on Web Mining and Web-Based Application*, pp. 149–153, Wuhan, China, June, 2009.

[191] G. Fan, Y. JiHua, and Y. Min, "Design and implementation of a distributed ids alert aggregation model," in *Proceedings of the 2009 4th International Conference on Computer Science Education*, pp. 975–980, Nanning, China, July, 2009.

[192] H. Salehi, H. Shirazi, and R. A. Moghadam, "Increasing overall network security by integrating signature-based NIDS with packet filtering firewall," in *Proceedings of the 2009 International Joint Conference on Artificial Intelligence*, pp. 357–362, Hainan, China, April, 2009.

[193] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, "The 1999 DARPA off-line intrusion detection evaluation," *Computer Networks*, vol. 34, no. 4, pp. 579–595, 2000.

[194] N. Moustafa and J. Slay, "The significant features of the UNSW-NB15 and the KDD99 data sets for network intrusion detection systems," in *Proceedings of the 2015 4th International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pp. 25–31, Kyoto, Japan, November, 2015.

[195] D. Jing and H.-B. Chen, "SVM based network intrusion detection for the UNSW-NB15 dataset," in *Proceedings of the 2019 IEEE 13th International Conference on ASIC (ASICON)*, pp. 1–4, Chongqing, China, October, 2019.

[196] A. Husain, A. Salem, C. Jim, and G. Dimitoglou, "Development of an efficient network intrusion detection model using extreme gradient boosting (XGBoost) on the UNSW-NB15 dataset," in *Proceedings of the 2019 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pp. 1–7, Ajman, United Arab Emirates, December, 2019.

[197] L. Zhiqiang, G. Mohi-Ud-Din, L. Bing, L. Jianchao, Z. Ye, and L. Zhijun, "Modeling network intrusion detection system using feed-forward neural network using UNSW-NB15 dataset," in *Proceedings of the 2019 IEEE 7th International Conference on Smart Energy Grid Engineering (SEGE)*, pp. 299–303, Oshawa, Canada, August, 2019.

[198] T. Janarthanan and S. Zargari, "Feature selection in UNSW-NB15 and KDDCUP'99 datasets," in *Proceedings of the 2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, pp. 1881–1886, Edinburgh, UK, June, 2017.

[199] A. Divekar, M. Parekh, V. Savla, R. Mishra, and M. Shirole, "Benchmarking datasets for anomaly-based network intrusion detection: KDD CUP 99 alternatives," in *Proceedings of the 2018 IEEE 3rd International Conference on Computing, Communication and Security (ICCCS)*, pp. 1–8, Kathmandu, Nepal, October, 2018.

[200] C. Wheelus, E. Bou-Harb, and X. Zhu, "Tackling class imbalance in cyber security datasets," in *Proceedings of the 2018 IEEE International Conference on Information Reuse and Integration (IRI)*, pp. 229–232, Salt Lake City, UT, USA, July, 2018.

[201] M. Al-Zewairi, S. Almajali, and A. Awajan, "Experimental evaluation of a multi-layer feed-forward artificial neural network classifier for network intrusion detection system," in *Proceedings of the 2017 International Conference on New Trends in Computing Sciences (ICTCS)*, pp. 167–172, Amman, Jordan, October, 2017.

[202] K. Sethi, R. Kumar, N. Prajapati, and P. Bera, "Deep reinforcement learning based intrusion detection system for cloud infrastructure," in *Proceedings of the 2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*, pp. 1–6, Bengaluru, India, January, 2020.

[203] N. Moustafa, G. Creech, E. Sitnikova, and M. Keshk, "Collaborative anomaly detection framework for handling big data of cloud computing," in *Proceedings of the 2017 Military Communications and Information Systems Conference (MilCIS)*, pp. 1–6, Canberra, Australia, November, 2017.

[204] S. Siddiqui, M. S. Khan, and K. Ferens, "Multiscale Hebbian neural network for cyber threat detection," in *Proceedings of the 2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 2161–4407, ISSN, Anchorage, AK, USA, May 2017.

[205] M. A. M. Aravind and V. Kalaiselvi, "Design of an intrusion detection system based on distance feature using ensemble classifier," in *Proceedings of the 2017 Fourth International Conference on Signal Processing, Communication and Networking (ICSCN)*, pp. 1–6, Chennai, India, March 2017.

[206] S. Yang, "Research on network behavior anomaly analysis based on bidirectional LSTM," in *Proceedings of the 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pp. 798–802, Chengdu, China, March 2019.

[207] G. Dlamini, R. Galieva, and M. Fahim, "A lightweight deep autoencoder-based approach for unsupervised anomaly detection," in *Proceedings of the 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, pp. 1–5, Abu Dhabi, United Arab Emirates, November, 2019.

[208] M. Azizjon, A. Jumabek, and W. Kim, "1D CNN based network intrusion detection with normalization on imbalanced data," in *Proceedings of the 2020 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pp. 218–224, Fukuoka, Japan, February, 2020.

[209] U. Ahmad, H. Asim, M. T. Hassan, and S. Naseer, "Analysis of classification techniques for intrusion detection," in *Proceedings of the 2019 International Conference on Innovative Computing (ICIC)*, pp. 1–6, Lahore, Pakistan, November, 2019.

[210] B. A. Tama, M. Comuzzi, and K. H. Rhee, "Tse-IDS A two-stage classifier ensemble for intelligent anomaly-based intrusion detection system," *IEEE Access*, vol. 7, pp. 94497–94507, 2019.

[211] O. O. Olasehinde, O. V. Johnson, and O. C. Olayemi, "Evaluation of selected meta learning algorithms for the prediction improvement of network intrusion detection system," in *Proceedings of the 2020 International Conference in Mathematics, Computer Engineering and Computer Science (ICMCECS)*, pp. 1–7, Ayobo, Nigeria, March, 2020.

[212] H. He, X. Sun, H. He, G. Zhao, L. He, and J. Ren, "A novel multimodal-sequential approach based on multi-view features for network intrusion detection," *IEEE Access*, vol. 7, pp. 183207–183221, 2019.

[213] M. H. Kamarudin, C. Maple, T. Watson, and N. S. Safa, "A LogitBoost-based algorithm for detecting known and unknown web attacks," *IEEE Access*, vol. 5, pp. 26190–26200, 2017.

[214] D. Gibert, C. Mateu, and J. Planes, "A hierarchical convolutional neural network for malware classification," in *Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, Budapest, Hungary, July, 2019.

[215] V. R. S. Selvin, *Malware Scores Based on Image Processing*, San Jose State University, San Jose, CA, USA, 2017.

[216] A. Makandar and A. Patrot, "Malware class recognition using image processing techniques," in *Proceedings of the 2017 International Conference on Data Management, Analytics and Innovation (ICDMAI)*, pp. 76–80, Pune, India, February, 2017.

[217] A. Makandar and A. Patrot, "Detection and retrieval of malware using classification," in *Proceedings of the 2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, pp. 1–5, Pune, India, August, 2017.

[218] W. R. Claycomb, C. L. Huth, B. Phillips, L. Flynn, and D. McIntire, "Identifying indicators of insider threats: insider IT sabotage," in *Proceedings of the 2013 47th International Carnahan Conference on Security Technology (ICCST)*, pp. 1–5, Medellin, Colombia, October, 2013.

[219] A. P. Moore, T. M. Cassidy, M. C. Theis, D. Bauer, D. M. Rousseau, and S. B. Moore, "Balancing Organizational Incentives to Counter Insider Threat," in *Proceedings of the 2018 IEEE Security and Privacy Workshops (SPW)*, pp. 237–246, San Francisco, CA, USA, May, 2018.

[220] D. A. Mundie, S. Perl, and C. L. Huth, "Toward an ontology for insider threat research: varieties of insider threat definitions," in *Proceedings of the 2013 3rd Workshop on Socio-Technical Aspects in Security and Trust*, pp. 26–36, Orlando, FL, USA, June. 2013.

[221] O. Igbe and T. Saadawi, "Insider threat detection using an artificial immune system algorithm," in *Proceedings of the 2018 9th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, pp. 297–302, New York, NY, USA, November, 2018.

[222] L. Flynn, C. Huth, R. Trzeciak, and P. Buttles, "Best practices against insider threats for all nations," in *Proceedings of the 2012 Third Worldwide Cybersecurity Summit (WCS)*, pp. 1–8, New Delhi, India, October. 2012.

[223] W. R. Claycomb and A. Nicoll, "Insider threats to cloud computing: directions for new research challenges," in *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference*, pp. 387–394, Izmir, Turkey, July. 2012.

[224] M. Aldairi, L. Karimi, and J. Joshi, "A trust aware unsupervised learning approach for insider threat detection," in *Proceedings of the 2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pp. 89–98, Los Angeles, CA, USA, July, 2019.

[225] L. Liu, C. Chen, J. Zhang, O. De Vel, and Y. Xiang, "Insider threat identification using the simultaneous neural learning of multi-source logs," *IEEE Access*, vol. 7, pp. 183162–183176, 2019.

[226] F. Meng, F. Lou, Y. Fu, and Z. Tian, "Deep learning based attribute classification insider threat detection for data security," in *Proceedings of the 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, pp. 576–581, Guangzhou, China, June. 2018.

[227] L. Lin, S. Zhong, C. Jia, and K. Chen, "Insider threat detection based on deep belief network feature representation," in *Proceedings of the 2017 International Conference on Green Informatics (ICGI)*, pp. 54–59, Fuzhou, China, August. 2017.

[228] A. Saaudi, Z. Al-Ibadi, Y. Tong, and C. Farkas, "Insider threats detection using CNN-LSTM model," in *Proceedings of the 2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 94–99, Las Vegas, NV, USA, December, 2018.

[229] J. Wang, L. Cai, A. Yu, and D. Meng, "Embedding learning with heterogeneous event sequence for insider threat detection," in *Proceedings of the 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 947–954, November. 2019.

[230] F. L. Greitzer, A. P. Moore, D. M. Cappelli, D. H. Andrews, L. A. Carroll, and T. D. Hull, "Combating the insider cyber threat," *IEEE Security Privacy*, vol. 6, no. 1, pp. 61–64, 2008.

[231] P. A. Legg, "Visualizing the insider threat: challenges and tools for identifying malicious user activity," in *Proceedings of the 2015 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pp. 1–7, Chicago, IL, USA, October, 2015.

[232] A. J. Hall, N. Pitropakis, W. J. Buchanan, and N. Moradpoor, "Predicting malicious insider threat scenarios using organizational data and a heterogeneous stack-classifier," in *Proceedings of the 2018 IEEE International Conference on Big Data (Big Data)*, pp. 5034–5039, Seattle, WA, USA, December, 2018.

[233] J. Jiang, J. Chen, T. Gu et al., "Warder: online insider threat detection system using multi-feature modeling and graph-based correlation," in *Proceedings of the 2019 IEEE Military Communications Conference*, pp. 1–6, Norfolk, VA, USA, November, 2019.

[234] A. Liu, X. Du, and N. Wang, "Recognition of access control role based on convolutional neural network," in *Proceedings of the 2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, pp. 2069–2074, Chengdu, China, December, 2018.

[235] D. Noever, "Classifier Suites for Insider Threat Detection," 2019, http://arxiv.org/abs/1901.10948.

[236] C.-W. Chen, C.-H. Su, K.-W. Lee, and P.-H. Bair, "Malware Family Classification Using Active Learning by Learning," in *Proceedings of the 2020 22nd International Conference on Advanced Communication Technology (ICACT)*, pp. 590–595, Phoenix Park, Korea (South), February, 2020.

[237] A. Karim, R. Salleh, and M. K. Khan, "SMARTbot: a behavioral analysis framework augmented with machine learning to identify mobile botnet applications," *PLoS One*, vol. 11, no. 3, 2016.

[238] X. Zhang, M. Marwah, I.-T. Lee, M. Arlitt, and D. Goldwasser, "Ace – an anomaly contribution explainer for cyber-security applications," in *Proceedings of the 2019 IEEE International Conference on Big Data (Big Data)*, pp. 1991–2000, Los Angeles, CA, USA, December, 2019.

[239] P. Buono and P. Carella, "Towards secure mobile learning. Visual discovery of malware patterns in android apps," in *Proceedings of the 2019 23rd International Conference Information Visualisation (IV)*, pp. 364–369, Paris, France, July. 2019.

[240] A. Skovoroda and D. Gamayunov, "Automated static analysis and classification of android malware using permission and API calls models," in *Proceedings of the 2017 15th Annual Conference on Privacy, Security and Trust (PST)*, pp. 243–24309, Calgary, AB, Canada, August, 2017.

[241] A. Demontis, M. Melis, M. Pintor et al., "Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks," Jun. 2019, http://arxiv.org/abs/1809.02861.

[242] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," 2018, http://arxiv.org/abs/1802.10135.

[243] S. Bhattacharya, H. D. Menendez, E. Barr, and D. Clark, "Itect: scalable information theoretic similarity for malware detection," 2016, http://arxiv.org/abs/1609.02404.

[244] J. Drew, T. Moore, and M. Hahsler, "Polymorphic malware detection using sequence classification methods," in *Proceedings of the 2016 IEEE Security and Privacy Workshops (SPW)*, pp. 81–87, San Jose, CA, USA, May 2016.

[245] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," 2016, http://arxiv.org/abs/1511.04317.

[246] M. Hassen and P. K. Chan, "Scalable function call graph-based malware classification," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Ser. CODASPY '17*, pp. 239–248, Association for Computing Machinery, New York, NY, USA, March. 2017.

[247] D. Gibert, C. Mateu, J. Planes, and R. Vicens, "Classification of malware by using structural entropy on convolutional neural networks," in *Proceedings of the Thirty-Second AAAI Conference on Artificial, Intelligence, (AAAI-18)*, S. A. McIlraith and K. Q. Weinberger, Eds., pp. 7759–7764, AAAI Press, New Orleans, LA, USA, February, 2018.

[248] D. Gibert, "Using convolutional neural networks for classification of malware represented as images," *Journal of Computer Virology and Hacking Techniques*, vol. 15, no. 1, pp. 15–28, 2019.

[249] T. M. Kebede, O. Djaneye-Boundjou, B. N. Narayanan, A. Ralescu, and D. Kapp, "Classification of malware programs using autoencoders based deep learning architecture and its application to the microsoft malware classification challenge (big 2015) dataset," in *Proceedings of the 2017 IEEE National Aerospace and Electronics Conference (NAECON)*, pp. 70–75, Dayton, OH, USA, June. 2017.

[250] T. Messay-Kebede, B. N. Narayanan, and O. Djaneye-Boundjou, "Combination of traditional and deep learning based architectures to overcome class imbalance and its application to malware classification," in *Proceedings of the NAECON 2018- IEEE National Aerospace and Electronics Conference*, pp. 73–77, Dayton, OH, USA, July. 2018.

[251] B. N. Narayanan, O. Djaneye-Boundjou, and T. M. Kebede, "Performance analysis of machine learning and pattern recognition algorithms for Malware classification," in *Proceedings of the 2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*, pp. 338–342, Dayton, OH, USA, July. 2016.

[252] M. Kumari, G. Hsieh, and C. A. Okonkwo, "Deep learning approach to malware multi-class classification using image processing techniques," in *Proceedings of the 2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 13–18, Las Vegas, NV, USA, December, 2017.

[253] H. Safa, M. Nassar, and W. A. Rahal Al Orabi, "Benchmarking convolutional and recurrent neural networks for malware classification," in *Proceedings of the 2019 15th International Wireless Communications Mobile Computing Conference (IWCMC)*, pp. 561–566, Tangier, Morocco, June. 2019.

[254] V. S. Priyamvada Davuluru, B. Narayanan Narayanan, and E. J. Balster, "Convolutional Neural Networks as Classification Tools and Feature Extractors for Distinguishing Malware Programs," in *Proceedings of the 2019 IEEE National Aerospace and Electronics Conference (NAECON)*, pp. 273–278, Dayton, OH, USA, July. 2019.

[255] E. Burnaev and D. Smolyakov, "One-class SVM with privileged information and its application to malware detection," in *Proceedings of the 2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pp. 273–280, Barcelona, Spain, December. 2016.

[256] S. J. Kim, B. J. Kim, H. C. Kim, and D. H. Lee, "Update state tampering: a novel adversary post-compromise technique on cyber threats," in *Proceedings of the 15th International Conference Detection of Intrusions and Malware, and Vulnerability Assessment 2018*, pp. 141–161, Springer Verlag, Saclay, France, January. 2018.

[257] M. Alkasassbeh and S. Al-Daleen, "Classification of malware based on file content and characteristics," 2018, https://arxiv.org/abs/1810.07252.

[258] S. A. Roseline and S. Geetha, "An efficient malware detection system using hybrid feature selection methods," *International Journal of Engineering and Advanced Technology*, vol. 9, no. 1S3, pp. 224–228, 2019.

[259] E. G. Dada, J. S. Bassi, Y. J. Hurcha, and A. H. Alkali, "Performance evaluation of machine learning algorithms for detection and prevention of malware attacks," *IOSR Journal of Computer Engineering*, vol. 21, no. 3, pp. 18–27, 2019.

[260] S. Pramanik and H. Teja, "Ember- analysis of malware dataset using convolutional neural networks," in *Proceedings of the 2019 Third International Conference on Inventive Systems and Control (ICISC)*, pp. 286–291, Coimbatore, India, January. 2019.

[261] Y. Oyama, T. Miyashita, and H. Kokubo, "Identifying useful features for malware detection in the ember dataset," in *Proceedings of the 2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*, pp. 360–366, Nagasaki, Japan, November. 2019.

[262] A. T. Nguyen, E. Raff, and A. Sant-Miller, "Would a file by any other name seem as malicious?" in *Proceedings of the 2019 IEEE International Conference on Big Data (Big Data)*, pp. 1322–1331, Los Angeles, CA, USA, December. 2019.

[263] Tisf, "ytisf/theZoo," 2020, https://github.com/ytisf/theZoo.

[264] J. Choi, D. Shin, H. Kim, J. Seotis, and J. B. Hong, "AMVG: Adaptive malware variant generation framework using machine learning," in *Proceedings of the 2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 246–24609, Kyoto, Japan, December. 2019.

[265] R. M. Verma and D. J. Marchette, *Cybersecurity Analytics*, CRC Press, Boca Raton, FL, USA, 2019.

[266] C. Eagle, "The IDA Pro book," *The Unofficial Guide to the World's Most Popular Disassembler*, No Starch Press, San Francisco, CA, USA, 2nd edition, 2011.

[267] P. Khodamoradi, M. Fazlali, F. Mardukhi, and M. Nosrati, "Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms," in *Proceedings of the 2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*, pp. 1–6, Tehran, Iran, October. 2015.

[268] N. Çarkacı and B. Soğukpınar, "Frequency based metamorphic malware detection," in *Proceedings of the 2016 24th Signal Processing and Communication Application Conference (SIU)*, pp. 421–424, Zonguldak, Turkey, May 2016.

[269] S. K. Sahay and A. Sharma, "Grouping the executables to detect malwares with high accuracy," *Procedia Computer Science*, vol. 78, pp. 667–674, 2016.

[270] A. G. Kakisim, M. Nar, N. Carkaci, and I. Sogukpinar, "Analysis and evaluation of dynamic feature-based malware detection methods," in *Proceedings of the Innovative Security Solutions for Information Technology and Communications: 11th International Conference, SecITC 2018*, Springer, Bucharest, Romania, February. 2019.

[271] B. Vishal, *Data Mining and Analysis in The Engineering Field*, IGI Global, Hershey, Pennsylvania, 2014.
[272] M. Yassine, *Security and Privacy Management, Techniques, and Protocols*, IGI Global, Hershey, Pennsylvania, 2018.
[273] A. Khalilian, A. Nourazar, M. Vahidi-Asl, and H. Haghighi, "G3MD: mining frequent opcode sub-graphs for metamorphic malware detection of existing families," *Expert Systems with Applications*, vol. 112, pp. 15–33, 2018.
[274] W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, 2006.
[275] V. P. Nair, H. Jain, Y. K. Golecha, M. S. Gaur, and V. Laxmi, "MEDUSA: MEtamorphic malware dynamic analysis usingsignature from API," in *Proceedings of the 3rd International Conference on Security of Information and Networks*, pp. 263–269, Association for Computing Machinery, New York, NY, USA, September. 2010.
[276] M. F. Zolkipli and A. Jantan, "A Framework for Malware Detection Using Combination Technique and Signature Generation," in *Proceedings of the 2010 2nd International Conference on Computer Research and Development*, pp. 196–199, Kuala Lumpur, Malaysia, May 2010.
[277] S. K. Agarwal and V. Shrivastava, "An opcode statistical analysis for metamorphic malware," 2013, https://www.semanticscholar.org/paper/An-Opcode-Statistical-Analysis-for-Metamorphic-Agarwal-Shrivastava/2c5f5c82b4d814ec1327e97404e79945bde7a354.
[278] M. Shelar and S. Rao, "Malicious threats detection of executable file," *The International Journal of Innovative Technology and Exploring Engineering*, vol. 9, no. 3, pp. 3257–3262, 2020.
[279] E. A. Daoud, "Metamorphic viruses detection using artificial immune system," in *Proceedings of the 2009 International Conference on Communication Software and Networks*, pp. 168–172, Chengdu, China, February. 2009.
[280] R. Chouchane, N. Stakhanova, A. Walenstein, and A. Lakhotia, "Detecting machine-morphed malware variants via engine attribution," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 3, pp. 137–157, Aug. 2013.
[281] A. S. Bist and A. Sharma, "Analysis of computer virus using feature fusion," in *Proceedings of the 2016 Second International Conference on Computational Intelligence Communication Technology (CICT)*, pp. 609–614, Ghaziabad, India, February. 2016.
[282] R. Mirzazadeh, M. H. Moattar, and M. V. Jahan, "Metamorphic malware detection using linear discriminant analysis and graph similarity," in *Proceedings of the 2015 5th International Conference on Computer and Knowledge Engineering (ICCKE)*, pp. 61–66, Mashhad, Iran, October. 2015.
[283] J. Raphel, "Pruned feature space for metamorphic malware detection using Markov Blanket," in *Proceedings of the 2015 Eighth International Conference on Contemporary Computing (IC3)*, pp. 377–382, Noida, India, August. 2015.
[284] J. Kuriakose and P. Vinod, "Ranked linear discriminant analysis features for metamorphic malware detection," in *Proceedings of the 2014 IEEE International Advance Computing Conference (IACC)*, pp. 112–117, New Delhi, India, February. 2014.
[285] A. Venkatesan, *Code Obfuscation and Virus Detection*, San Jose State University, San Jose, CA, USA, 2008.
[286] M. Stamp, *Introduction to Machine Learning with Applications in Information Security*, CRC Press, Boca Raton, FL, USA, 2017.
[287] S. Attaluri, S. McGhee, and M. Stamp, "Profile hidden Markov models and metamorphic virus detection," *Journal in Computer Virology*, vol. 5, no. 2, pp. 151–169, 2009.
[288] N. Runwal, R. M. Low, and M. Stamp, "Opcode graph similarity and metamorphic detection," *Journal in Computer Virology*, vol. 8, no. 1, pp. 37–52, 2012.
[289] M. A. Davis, S. Bodmer, and A. LeMasters, *Hacking Exposed Malware & Rootkits: Malware & Rootkits Security Secrets & Solutions*, McGraw Hill, New York, NY, USA, 2010.
[290] A. Singh and A. Islam, "An encounter with trojan nap," 2020.
[291] A. Singh and Y. Khalid, "Don't click the left mouse button: introducing trojan upclicker," 2020.
[292] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi, "Duqu: a stuxnet-like malware found in the wild," Technical Report, Budapest University of Technology and Economics, Budapest, 2011.
[293] V. Kamluk, "The mystery of duqu: part six (the command and control servers)," Kaspersky, APT Reports, 2011, https://securelist.com/the-mystery-of-duqu-part-six-the-command-and-control-servers-36/31863/.
[294] A. Singh and Z. Bu, *Hot Knives through Butter: Evading File-Based Sandboxes*, White Paper, Milpitas, CA, USA, 2014.
[295] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: VMM detection myths and realities," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, pp. 1–6, USENIX Association, San Diego, CA, USA, May 2007.
[296] P. Faruki, A. Bharmal, V. Laxmi, M. Gaur, M. Conti, and M. Rajarajan, "Evaluation of android anti-malware techniques against Dalvik bytecode obfuscation," in *Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 414–421, Beijing, China, September. 2014.
[297] D. Kirat and G. Vigna, "Malgene: automatic extraction of malware analysis evasion signature," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, CO, USA, October, 2015.
[298] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of Android malware," in *Proceedings of the Seventh European Workshop on System Security*, Amsterdam, The Netherlands, May, 2014.
[299] P. Faruki, S. Bhandari, V. Laxmi, M. Gaur, and M. Conti, "DroidAnalyst: synergic app framework for static and dynamic app analysis," in *Recent Advances in Computational Intelligence in Defense and Security, Ser. Studies in Computational Intelligence*, R. Abielmona, R. Falcon, N. Zincir-Heywood, and H. A. Abbass, Eds., Springer International Publishing, pp. 519–552, New York, NY, USA, 2016.
[300] Androguard, "androguard/androguard," 2020, https://github.com/androguard/androguard.
[301] P. Lantz, "Pjlantz/droidbox," 2020, https://github.com/pjlantz/droidbox.
[302] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. V. D. Veen, and C. Platzer, "Andrubis – 1,000,000 apps later: a view on current android malware behaviors," in *Proceedings of the 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pp. 3–17, Kyoto, Japan, September. 2014.
[303] S. Bosworth and M. E. Kabay, *Computer Security Handbook*, John Wiley & Sons, Hoboken, NJ, USA, 2002.

[304] V. Sihag, M. Vardhan, and P. Singh, "A survey of android application and malware hardening," *Computer Science Review*, vol. 39, Article ID 100365, 2021.

[305] H. El Merabet and A. Hajraoui, "A survey of malware detection techniques based on machine learning," *International Journal of Advanced Computer Science and Applications*, vol. 10, 2019.

[306] A. Souri and R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," *Human-centric Computing and Information Sciences*, vol. 8, no. 1, p. 3, 2018.

[307] N. Idika and A. Mathur, "A survey of malware detection techniques," in *Proceedings of the 2007 International Conference on Information Technology (ICIT)*, Amman, Jordan, July, 2007.

[308] M. Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE Access*, vol. 8, 2020.

[309] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: research developments, trends and challenges," *Journal of Network and Computer Applications*, vol. 153, Article ID 102526, 2020.

[310] D. M. Chess and S. R. White, "Un undectable computer virus," *Proceedings of Virus Bulletin*, vol. 5, 2000.

[311] M. Narouei, M. Ahmadi, G. Giacinto, H. Takabi, and A. Sami, "DLLMiner: structural mining for malware detection," *Security and Communication Networks*, vol. 8, no. 18, pp. 3311–3322, 2015.

[312] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy. Science Progress 2001*, Oakland, CA, USA, May, 2001.

[313] S. P. Choudhary and M. D. Vidyarthi, "A simple method for detection of metamorphic malware using dynamic analysis and text mining," *Procedia Computer Science*, vol. 54, pp. 265–270, 2015.

[314] H. S. Galal, Y. B. Mahdy, and M. A. Atiea, "Behavior-based features model for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 12, no. 2, pp. 59–67, 2016.

[315] K. Brezinski and K. Ferens, "Sandy toolbox: a framework for dynamic malware analysis and model development," in *Transactions on Computational Science and Computational Intelligence*, Springer Nature, Berlin, Germany, 2021.

[316] Z. Salehi, A. Sami, and M. Ghiasi, "MAAR: robust features to detect malicious activity based on API calls, their arguments and return values," *Engineering Applications of Artificial Intelligence*, vol. 59, pp. 93–102, 2017.

[317] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying dormant functionality in malware programs," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pp. 61–76, Oakland, CA, USA, May 2010.

[318] A. Firdaus, N. B. Anuar, M. F. A. Razak, and A. K. Sangaiah, "Bio-inspired computational paradigm for feature investigation and malware detection: interactive analytics," *Multimedia Tools and Applications*, vol. 77, no. 14, pp. 17519–17555, 2018.

[319] A. G. Kakisim, M. Nar, and I. Sogukpinar, "Metamorphic malware identification using engine-specific patterns based on co-opcode graphs," *Computer Standards and Interfaces*, vol. 71, Article ID 103443, 2020.

[320] U. Baldangombo, N. Jambaljav, and S.-J. Horng, "A static malware detection system using data mining methods," 2013, http://arxiv.org/abs/1308.2831.

[321] K. Brezinski and K. Ferens, "An adaptive tribal topology for particle swarm optimization," in *Transactions on Computational Science & Computational Intelligence, Ser. Advances in Security, Networks, and Internet of Things*, Springer Nature, New York, NY, USA, 2018.

[322] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, No Starch Press, San Francisco, CA, USA, 2012.

[323] M. E. Ahmed, S. Nepal, and H. Kim, "Medusa: malware detection using statistical analysis of system's behavior," in *Proceedings of the 2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pp. 272–278, Philadelphia, PA, USA, October. 2018.

[324] J. Pavithran, M. Patnaik, and C. Rebeiro, "{D-TIME}: distributed threadless independent malware execution for runtime obfuscation," 2019, https://www.usenix.org/conference/woot19/presentation/pavithran.

[325] S. Zamir, Y. Margalit, and D. Margalit, "Method for detecting unwanted executables," US Patent US20060015940A1, 2006.

[326] R. T. Experiments, "Persisting in svchost.exe with a service dll," 2022, https://www.ired.team/offensive-security/persistence/persisting-in-svchost.exe-with-a-service-dll-servicemain.

[327] M. Alazab, S. Venkataraman, and P. Watters, "Towards understanding malware behaviour by the extraction of API calls," in *Proceedings of the 2010 Second Cybercrime and Trustworthy Computing Workshop*, pp. 52–59, Ballarat, Australia, July 2010.

[328] M. B. Y. Bahtiyar and C. Y. Altıniğne, "A multi-dimensional machine learning approach to predict advanced malware," *Computer Networks*, vol. 160, pp. 118–129, 2019.

[329] S. Gupta, H. Sharma, and S. Kaur, "Malware characterization using windows API call sequences," in *Security, Privacy, and Applied Cryptography Engineering*, C. Carlet, M. A. Hasan, and V. Saraswat, Eds., pp. 271–280, Springer International Publishing, New York, NY, USA, 2016.

[330] S. Gupta, D. R. D. O. Sag, I. Delhi et al., "Malware characterization using WindowsAPI call sequences," *Journal of Clinical Sleep Medicine*, vol. 7, no. 4, pp. 363–378, 2018.

[331] M. Belaoued and S. Mazouzi, "Statistical study of imported APIs by PE type malware," in *Proceedings of the 2014 International Conference on Advanced Networking Distributed Systems and Applications*, pp. 82–86, Bejaia, Algeria, June 2014.

[332] P. Vinod, H. Jain, Y. Golecha, M. Gaur, and V. Laxmi, "MEDUSA: MEtamorphic malware dynamic analysis usingsignature from API," in *Proceedings of the 3rd International Conference of Security of Information and Networks*, Rostov-on-Don, Russia, January 2010.

[333] A. Matrosov, E. Rodionov, D. Harley, and J. Malcho, "Stuxnet under the microscope," 2010, https://www.esetnod32.ru/company/viruslab/analytics/doc/Stuxnet_Under_the_Microscope.pdf.

[334] Y. Ding, X. Xia, S. Chen, and Y. Li, "A malware detection method based on family behavior graph," *Computers and Security*, vol. 73, pp. 73–86, 2018.

[335] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[336] Y. Ye, L. Chen, D. Wang, T. Li, Q. Jiang, and M. Zhao, "SBMDS: an interpretable string based malware detection system using SVM ensemble with bagging," *Journal in Computer Virology*, vol. 5, no. 4, p. 283, 2008.

[337] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on API call sequence analysis," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, Article ID 659101, 2015.

[338] A. Hellal and L. Ben Romdhane, "Minimal contrast frequent pattern mining for malware detection," *Computers and Security*, vol. 62, pp. 19–32, 2016.

[339] W. Han, J. Xue, Y. Wang et al., "Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics," *Computers and Security*, vol. 83, pp. 208–233, 2019.

[340] D. Uppal, R. Sinha, V. Mehra, and V. Jain, "Malware detection and classification based on extraction of API sequences," in *Proceedings of the 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 2337–2342, Delhi, India, September 2014.

[341] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proceedings of the Network and Distributed System Security Symposium*, p. 18, San Diego, California, USA, February 2009.

[342] A. Pektaş and T. Acarman, "Classification of malware families based on runtime behaviors," *Journal of Information Security and Applications*, vol. 37, pp. 91–100, 2017.

[343] M. Rhode, L. Tuson, P. Burnap, and K. Jones, "LAB to SOC: robust features for dynamic malware detection," in *Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Industry Track*, pp. 13–16, Portland, OR, USA, June. 2019.

[344] I. Santos, J. Devesa, F. Brezo, J. Nieves, and P. G. Bringas, "OPEM: a static-dynamic approach for machine-learning-based malware detection," in *Proceedings of the International Joint Conference CISIS'12-ICEUTE´12-SOCO´12 Special Sessions*, l. Herrero, V. Snášel, A. Abraham et al., Eds., pp. 271–280, Springer, Berlin, Germany, March, 2013.

[345] P. Shijo and A. Salim, "Integrated static and dynamic analysis for malware detection," *Procedia Computer Science*, vol. 46, pp. 804–811, 2015.

[346] K. Kaushal, P. Swadas, and N. Prajapati, *Metamorphic Malware Detection Using Statistical Analysis*, vol. 2, no. 3, p. 5, 2012.

[347] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala, "Malware detection using assembly and API call sequences," *Journal in Computer Virology*, vol. 7, no. 2, pp. 107–119, 2011.

[348] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq, "Using spatio-temporal information in API calls with machine learning algorithms for malware detection," in *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence, ser. AISec '09*, pp. 55–62, Association for Computing Machinery, New York, NY, USA, November 2009.

[349] C. Wang, J. Pang, R. Zhao, W. Fu, and X. Liu, "Malware detection based on suspicious behavior identification," in *Proceedings of the 2009 First International Workshop on Education Technology and Computer Science*, pp. 198–202, IEEE Computer Society, Hubei, China, March. 2009.

[350] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, "Malware detection based on mining API calls," in *Proceedings of the 2010 ACM Symposium on Applied Computing, Ser. SAC '10*, pp. 1020–1025, Association for Computing Machinery, Sierre, Switzerland, March 2010.

[351] Y.-D. Shen, Z. Zhang, and Q. Yang, "Objective-oriented utility-based association mining," in *Proceedings of the 2002 IEEE international conference on data mining*, pp. 426–433, Maebashi, Japan, December 2002.

[352] Y. Ye, D. Wang, T. Li, and D. Ye, "Imds: intelligent malware detection system," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Jose, CA, USA, August 2007.

[353] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang, "An intelligent PE-malware detection system based on association mining," *Journal in Computer Virology*, vol. 4, no. 4, pp. 323–334, 2008.

[354] Y. Ye, T. Li, Q. Jiang, and Y. Wang, "CIMDS: adapting postprocessing techniques of associative classification for malware detection," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 40, no. 3, pp. 298–307, 2010.

[355] A. A. G. Al-Hamodi, S. Lu, and Y. Alsalhi, "An enhanced frequent pattern growth based on map reduce for mining association rules," *International Journal of Data Mining and Knowledge Management Process*, vol. 6, 2016.

[356] Y. Ding, X. Yuan, K. Tang, X. Xiao, and Y. Zhang, "A fast malware detection algorithm based on objective-oriented association mining," *Computers and Security*, vol. 39, pp. 315–324, 2013.

[357] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications Surveys Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2016.

[358] G. Tesauro, J. Kephart, and G. Sorkin, "Neural networks for computer virus recognition," *IEEE Expert*, vol. 11, no. 4, pp. 5-6, 1996.

[359] A. Walenstein, D. J. Hefner, and J. Wichers, "Header information in malware families and impact on automated classifiers," in *Proceedings of the 2010 5th International Conference on Malicious and Unwanted Software*, pp. 15–22, Nancy, France, October 2010.

[360] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, 2006.

[361] C. Ravi and R. Manoharan, "Malware detection using windows api sequence and machine learning," *International Journal of Computer Applications*, vol. 43, 2012.

[362] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Detection of Intrusions and Malware and Vulnerability Assessment*, R. Büschkes and P. Laskov, Eds., pp. 129–143, Springer, Berlin, Germany, 2016.

[363] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, *Control flow graphs as malware signatures*, International Workshop on the Theory of Computer Viruses, Nancy, France, 2007.

[364] K. Jeong and H. Lee, "Code graph for malware detection," in *Proceedings of the 2008 International Conference on Information Networking*, pp. 1–5, Busan, Korea (South), January. 2008.

[365] M. Eskandari and H. Raesi, "Frequent sub-graph mining for intelligent malware detection," *Security and Communication Networks*, vol. 7, no. 11, pp. 1872–1886, 2014.

[366] P. Faruki, V. Laxmi, M. S. Gaur, and P. Vinod, "Mining control flow graph as API call-grams to detect portable executable malware," in *Proceedings of the Fifth International Conference on Security of Information and Networks*, pp. 130–137, Association for Computing Machinery, Jaipur, India, October 2012.

[367] K. Blokhin, J. Saxe, and D. Mentis, "Malware similarity identification using call graph based system call subsequence

features," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, pp. 6–10, Philadelphia, PA, USA, July. 2013.

[368] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 5–14, Association for Computing Machinery, New York, NY, USA, September. 2007.

[369] R. M. H. Ting and J. Bailey, "Mining minimal contrast subgraph patterns," in *Proceedings of the Sixth SIAM International Conference on Data Mining*, Bethesda, MD, USA, April 2006.

[370] T. Wüchner, M. Ochoa, and A. Pretschner, "Malware detection with quantitative data flow graphs," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, pp. 271–282, Association for Computing Machinery, New York, NY, USA, June 2014.

[371] J. Lee, K. Jeong, and H. Lee, "Detecting metamorphic malwares using code graphs," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 1970–1977, Association for Computing Machinery, Sierre, Switzerland, March 2010.

[372] V. Mehra, V. Jain, and D. Uppal, "DaCoMM: detection and classification of metamorphic malware," in *Proceedings of the 2015 Fifth International Conference on Communication Systems and Network Technologies*, pp. 668–673, Gwalior, India, April. 2015.

[373] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen, "Adversarial learning attacks on graph-based IoT malware detection systems," in *Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1296–1305, Dallas, TX, USA, July 2019.

[374] K. Vinayaka and C. Jaidhar, "Android malware detection using function call graph with graph convolutional networks," in *Proceedings of the 2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*, pp. 279–287, Jalandhar, India, May 2021.

[375] P. Feng, J. Ma, T. Li, X. Ma, N. Xi, and D. Lu, "Android malware detection based on call graph via graph neural network," in *Proceedings of the 2020 International Conference on Networking and Network Applications (NaNA)*, pp. 368–374, Haikou, China, December 2020.

[376] R. Taheri, M. Ghahramani, R. Javidan, M. Shojafar, Z. Pooranian, and M. Conti, "Similarity-based Android malware detection using Hamming distance of static binary features," *Future Generation Computer Systems*, vol. 105, pp. 230–247, 2020.

[377] M. Eskandari and S. Hashemi, "A graph mining approach for detecting unknown malwares," *Journal of Visual Languages and Computing*, vol. 23, no. 3, pp. 154–162, 2012.

[378] E. B. Karbab and M. Debbabi, "MalDy: portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports," *Digital Investigation*, vol. 28, pp. S77–S87, 2019.

[379] F. O. Catak, A. F. Yazı, O. Elezaj, and J. Ahmed, "Deep learning based Sequential model for malware analysis using Windows exe API Calls," *PeerJ computer science*, vol. 6, p. e285, 2020.

[380] K. Brezinski and K. Ferens, "Graph-oriented modelling of process event activity for the detection of malware," in *2023 International Conference on Computational Science and Computational Intelligence*, Las Vegas, USA, July 2023.

[381] K. Brezinski and K. Ferens, "Transformers-Malware in Disguise," in *Transactions on Computational Science & Computational Intelligence*, Springer Nature, Berlin, Germany, 2021.

[382] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, http://arxiv.org/abs/1301.3781.

[383] E. Amer, S. El-Sappagh, and J. W. Hu, "Contextual identification of windows malware through semantic interpretation of API call sequence," *Applied Sciences*, vol. 10, no. 21, p. 7673, 2020.

[384] E. Amer and I. Zelinka, "A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence," *Computers and Security*, vol. 92, Article ID 101760, 2020.

[385] J. Kang, S. Jang, S. Li, Y.-S. Jeong, and Y. Sung, "Long short-term memory-based Malware classification method for information security," *Computers and Electrical Engineering*, vol. 77, pp. 366–375, 2019.

[386] F. O. Catak and A. F. Yazı, "A benchmark api call dataset for windows pe malware classification," 2021, http://arxiv.org/abs/1905.01999.

[387] A. F. Yazi, F. Z. Çatak, and E. Gül, "Classification of methamorphic malware with deep learning(lstm)," in *Proceedings of the 2019 27th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, Sivas, Turkey, April 2019.

[388] S. McDonnell, O. Nada, M. R. Abid, and E. Amjadian, "CyberBERT: a deep dynamic-state session-based recommender system for cyber threat recognition," in *Proceedings of the 2021 IEEE Aerospace Conference (50100)*, pp. 1–12, Big Sky, MT, USA, March 2021.

[389] S. Yesir and B. Soğukpinar, "Malware detection and classification using fastText and BERT," in *Proceedings of the 2021 9th International Symposium on Digital Forensics and Security (ISDFS)*, pp. 1–6, Elazig, Turkey, June 2021.

[390] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext.zip: compressing text classification models," 2016, http://arxiv.org/abs/1612.03651.