Journal of Computer Science and Cybernetics, V.40, N.3 (2024), 233-248
DOI: 10.15625/1813-9663/20741

BYPASSING ANTI-EMULATION METHODS FOR MALWARE DETECTION

VAN LOI CAO[1,*], DINH DAI NGUYEN[2]

[1]*Institute of Information and Communication Technology, Le Quy Don Technical University, 236 Hoang Quoc Viet Street, Cau Giay District, Ha Noi, Viet Nam*
[2]*Institute of Cryptography Science and Technology, Government Cipher Committee, 24 Ly Thuong Kiet Street, Hoan Kiem District, Ha Noi, Viet Nam*

Abstract. Malware detection has played a crucial role in many cyberattacks in recent years. Due to the obfuscated nature of malware, the traditional static analysis technique tends to be ineffective. Additionally, modern malware often can identify dynamic analysis environments, posing challenges to dynamic analysis methods. Thus, feature extraction relies on analysis techniques that tend to be less effective in obfuscated malware, resulting in poor performance of subsequent machine learning-based detectors. This study introduces a Bypass Anti-emulation-based Malware Detection framework (BAE-MD) for enhancing the efficiency of obfuscated malware detection. In other words, BAE-MD includes a method that can bypass the anti-emulation mechanism of malware in a controlled dynamic environment. This forces the malware to decrypt and decompress its actual malicious code to memory. By doing so, Yara rules can be applied to memory dump to extract more than 60 features to feed into detectors. BAE-MD is evaluated on a malware dataset in comparison with others using static and dynamic analysis technique-based feature extraction. The experimental results can confirm that our method outperforms the others. More investigations are also carried out to illustrate the efficiency of BAE-MD. These results suggest that BAE-MD is a promising approach for dealing with the continuous evolution of malware.

Keywords. Malware analysis, Malware detection, obfuscation, anti-emulation, feature extraction.

1. INTRODUCTION

In contemporary times, cybercriminals leverage malware as a primary tool to perpetrate attacks on computer systems. The internet serves as the primary conduit for executing these malicious activities, facilitated through avenues such as emails, malicious websites, and the distribution of downloadable software [1]. Malware consists of harmful software deliberately engineered to execute destructive functions. [2, 3]. These programs are typically classified according to their behavior and operational methods, encompassing a diverse array of types such as viruses, worms, trojan horses, backdoors, rootkits, and ransomware. The objectives behind attacking computer systems are multifaceted, including resource destruction, financial gain, unauthorized data access, utilization of computing resources, and service disruption [4–

*Corresponding author.
E-mail addresses: loi.cao@lqdtu.edu.vn (V.L. Cao); nguyendinhdai01@gmail.com (D.D. Nguyen)

7]. The detection of malicious code stands as a critical component in the response and troubleshooting protocols within companies and organizations.

Furthermore, the continual evolution of malware, marked by diverse variations and increasingly sophisticated methods of concealment, presents a formidable challenge to malware detection systems. Zero-day malware presents a particular challenge, as these exploits target vulnerabilities yet to be discovered [8, 9]. Whenever new software is developed, malicious actors seek out vulnerabilities to compromise its security. Consequently, there is a pressing need to continually patch and refine malware detection systems to effectively combat such threats [10, 11].

Malware detection commonly employs two main approaches: signature-based and behavior-based methods. Signature-based systems are known for their speed and efficiency, yet they can be susceptible to evasion techniques employed by obfuscated malware. [12, 13]. Conversely, behavior-based techniques demonstrate greater resilience against obfuscation but tend to be more time-intensive. While both signature and behavior-based methods have seen significant development, hybrid approaches have also emerged to combine the strengths of both techniques. Hybrid methods aim to address the limitations inherent in signature and behavior-based approaches. Recently, machine learning/deep learning (ML/DL) has been utilized for malware detection. These approaches have offered a promising avenue for identifying new and variant strains of malware [14–16].

The feature extraction process plays a crucial role in developing effective deep learning (DL) and machine learning (ML) models for malware detection. Both benign and malware samples are analyzed using static and dynamic techniques, and distinctive features are extracted to identify malicious files from benign files. The effectiveness of malware detection systems hinges on the precise extraction of relevant and discriminative features through these analytical methods [1]. Static analysis involves scrutinizing the code and structure of software without execution to ascertain potential maliciousness and discern the behavior of any malicious code. Various properties such as PE header file information, import and export functions, strings, and API calls are extracted during static analysis. Conversely, dynamic analysis techniques often entail executing malicious code within a virtualized environment, such as a sandbox, emulator, VMWare, or VirtualBox, to extract information on invoked APIs, exhibited behaviors, newly created files, and altered registry values.

However, the evolution of malware also poses new challenges to feature extraction. To circumvent security systems, malware developers have employed various obfuscation techniques, also known as anti-analysis techniques [17–20]. In other words, through the utilization of encryption and encoding techniques, sophisticated malicious programs such as metamorphic, polymorphic, and packed malware are created, posing significant challenges for malware analysis and detection techniques [21–24]. Feature extraction that relies on well-known static and dynamic malware analysis can not retrieve the true behaviors of the malware, leading to the inefficiency of subsequent ML/DL models.

To improve the efficiency of detection methods on malware with obfuscated techniques, this study has proposed a Bypass Anti-emulation-based Malware Detection framework (BAE-MD). Our framework consists of three components: Bypass Anti-emulation (BAE), Feature Extractor, and ML-based Detector. The Bypass Anti-emulation method stands out as the novel and pivotal component of BAE-MD, representing our primary contribution to this study. BAE is proposed to bypass the anti-emulation mechanism of malware. As a result,

malware is deceived to execute its actual behavior in memory (i.e. decompress, decryption, and decoding its malicious code in memory). This allows us to dump the memory into files and apply static analysis techniques for extracting valuable features. The details of the framework are presented in Section 4.

The main contribution of this study can be listed as follows:

1. Propose a novel Bypass Anti-emulation-based Malware Detection framework (BAE-MD). Under the control of BAE, the actual behavior of malware is explored for facilitating subsequent feature extraction and ML-based detectors.
2. Design a set of experiments to evaluate our BAE-based malware detection framework in comparison with other methods using static and dynamic malware analysis techniques. The performance of these methods is evaluated on benchmark malware datasets by common metrics.
3. Analyze the behavior of malware when executing them in the emulated environment controlled by BAE. The analyzing results such as execution time, number of API calls, entropy as well as strings, and the number of Yara rules matched on malware are compared to those in the case without using BAE.

The structure of this paper is shown as follows. Sections 2 and 3 introduce the two well-known malware analysis approaches as well as a brief discussion on recent ML-based malware detection methods. Section 4 presents our malware detection framework that tackles the challenges identified in this study. Section 5 describes the experimental analysis and compares it with relevant studies. Finally, Section 6 concludes the paper by discussing the findings and future works.

2. BACKGROUND

This section presents malware analysis techniques used in this study. This consists of static analysis on the PE header and dynamic analysis in sandbox environments. These techniques are commonly used for malware feature extraction [24].

2.1. Static analysis

Static analysis is a key method for examining source code without executing executable files, aiming to extract distinct signatures representing the files [4]. It gathers various static data types like PE-header information, string-based entropy, and compression ratios. Executable files, following the Common Object File Format (COFF), use the Portable Executable (PE) format on Windows. PE serves as a structured container conveying essential information for proper execution and management. Its sections include a DOS header, PE file header, section tables, PE sections, and a transport layer security (TLS) section. The PE header contains crucial details like code size, location, file header, optional header, and directories such as import, resource, and exception directories. These directories list DLLs, APIs, and catalog resources used by the software. Additionally, the PE header includes elements like Signature and NumberOfSections, facilitating the loading of executable files into memory during execution. The static analysis utilizes these elements to extract malware features, crucial for effective malware detection. By analyzing the structure and content of executable files, static analysis contributes significantly to identifying and mitigating potential threats.

2.2. Dynamic analysis

Dynamic analysis entails monitoring a program's activities while it is running [4]. In this method, the file under examination is executed within a sandbox environment, such as Cuckoo, which is designed to be controlled and monitored. This setup allows for the observation of the file's behavior and the collection of relevant dynamic data. Dynamic analysis can gather various types of information, including details about malicious activities evidenced by the executable's behavior and memory snapshots taken during execution. Key behaviors are identified by capturing invoked API calls, machine activities, file operations, as well as registry and network interactions. Additionally, opcode-based memory images can be captured to reflect dynamic malicious activities. The API calls recorded through dynamic analysis using tools like Cuckoo are subsequently converted into byte sequences for the purpose of binary feature classification.

3. RELATED WORK

This section will brief overview of recent malware detection approaches using machine learning methods. This aims to focus on how to extract malware behavior features using static analysis [25, 26] and dynamic analysis [16, 27–29].

In static analysis, the file being examined is not executed. Instead, the analysis relies exclusively on the file's contents and metadata. Consequently, the detection of its behavior is based solely on static features rather than dynamic execution characteristics. Recent studies, such as those by [25] and [26], exemplify machine learning (ML) approaches to malware detection using static analysis. These methods derive input features from executable files without running them. Static analysis involves scrutinizing the program's internal structure, including N-grams, opcodes, Portable Executable (PE) header information, strings, and import functions. However, modern malware frequently employs packers and encryption techniques to obfuscate its contents, leading to files with varied signatures. Such obfuscation can hinder static analysis by concealing the actual content and behavior of the malware. Consequently, the extracted static features may not accurately reflect the malware's behavior, resulting in diminished performance of subsequent ML-based detection methods.

Dynamic analysis examines the actions of a program during its execution [16]. In this process, the file under investigation is run within a sandbox- aan isolated and controlled environment-to observe its behavior, including changes to the file system, network activity, process management, and specific system calls [27]. This approach allows for the deobfuscation and unpacking of the file, extraction of memory dumps, and identification of its execution path. Beyond mere execution, dynamic analysis often involves debugging the binary to understand its capabilities, operational methods, and potential modifications of its execution flow to circumvent detection measures.

Recent studies indicate that sandbox environments, such as Cuckoo, are frequently employed to track all system calls, network connections, and file system modifications made by a binary. Deep learning methods are then utilized to identify malware [28, 29]. However, to counteract dynamic analysis, malware creators often incorporate checks within their binaries. These checks evaluate the execution environment and compare it to a typical user environment. They may detect monitoring mechanisms by analyzing factors such as user interactions (e.g., mouse movements, keyboard inputs), the presence of known analysis tools,

environmental variables indicative of a testing environment, specific hooks, or anomalies in execution timing. As a result, these sophisticated evasion techniques can significantly undermine the effectiveness of dynamic malware analysis.

In response to the challenges posed by sophisticated malware evasion techniques, a dynamic analysis-controlled environment is essential. This environment ensures malware behaves naturally for effective analysis, particularly as static analysis is hindered by malware's packing and obfuscation. This paper proposes a novel malware detection framework with a bypass anti-emulation method (BAE) for improving the performance of ML-based malware detection methods on obfuscated and packed malware. By executing malware within a controlled environment with BAE and extracting static features from memory dumps, insights into its behavior, network connections, and API calls are gained. These static features serve as input for ML-based models, enhancing malware detection accuracy by identifying patterns and malware.

4. PROPOSED APPROACH

This section will describe our proposed Bypass Anti-emulation-based Malware Detection framework (BAE-MD). As discussed in Section 1, the Bypass Anti-emulation method (BAE) is a crucial component of BAE-MD. It is designed to bypass the anti-emulation mechanism of malware, which involves monitoring API calls and measuring execution time to determine the characteristics of emulation environments. Consequently, the malware is compelled to execute its actual behavior in memory, enabling the extraction of valuable features from the memory. As illustrated in Figure 1, BAE plays a primary role in the BAE-MD framework. The second component is Feature Extractor which uses a static analysis technique, particularly Yara rules for extracting features from memory dump files. Finally, an ML-based Detector applies resulting features for classifying malware from benign. In the BAE-MD framework, the Bypass Anti-emulation method stands out as both a novel approach and the primary contribution. Details of these components are presented in Subsections 4.1, 4.2, and 4.3 as follows.

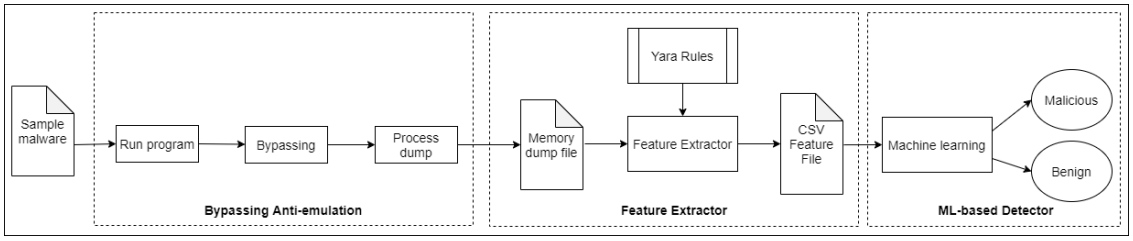

Figure 1: Malware Detection framework (BAE-MD) with Bypass Anti-emulation (BAE)

4.1. Bypass anti-emulation (BAE)

We chose the Qiling framework, which utilizes the Unicorn Engine, as our emulation tool for analysis. The Qiling framework [a] is an advanced binary emulation platform built

[a] Qiling framework. https://github.com/qilingframework/qiling, 2021.

on top of the Unicorn Engine [b]. Qiling serves as a high-level framework that leverages Unicorn to emulate CPU instructions while also managing operating system interactions. It supports various executable formats, including PE, Mach-O, and ELF, and incorporates dynamic linkers for loading and relocating shared libraries, as well as syscall and I/O handlers. Consequently, Qiling enables the execution of binaries without dependence on their native operating system. Unicorn provides a scriptable CPU emulator, and when a program makes a system call, Qiling endeavors to replicate the corresponding behavior of the host operating system (such as Windows or Linux). Details of the BAE method are presented in three steps as follows:

1. *Run program*: BAE starts with sending binary samples to the Qiling framework for execution. Qiling then utilizes the capabilities of the Unicorn Engine to emulate the execution of these binary samples. Qiling's orchestration manages the execution process, and it interacts with the Unicorn Engine to interpret the instructions within the binary. This process involves analyzing and executing each instruction step by step, thereby granting us the capability to intervene in these steps to bypass the anti-emulation function employed by malware.

2. *Bypassing*: It is a crucial task in the BAE method. Malware developers often implement various techniques to identify when their code is being executed within an emulation environment. Once malware has found the environment, it will hide its actual behavior leading to false analysis results. This study aims to bypass two typical techniques employed by malware to identify emulation environments, namely *monitoring API calls* and *measuring execution time*.

   Bypass API calls: The malware's use of API calls to specific functions is the characteristic of emulation environments. If these functions are not invoked, the malware may decide to execute its malicious code, such as GetWindowContextHelpId(), ImageListAdd(), CoReleaseMarshalData(), CreateEventA(), SetClassLongA(), and wgetmainargs(). To circumvent these APIs, we utilize API hooking techniques to intercept and modify API calls made by the malware. By altering the behavior of certain functions, we can deceive the malware into believing it's interacting with a real system. As demonstrated in the code below, we re-implement the GetWindowContextHelpId() API (as in line 2) in a format that Qiling can recognize.

```
1 @winsdkapi(cc=STDCALL, dllname="user32_dll")
2 def HookGetWindowContextHelpId(ql, address, params):
3   ERROR_INVALID_WINDOW_HANDLE = 0x578
4   ql.os.last_error = ERROR_INVALID_WINDOW_HANDLE
5   return 0
```

   Bypass measuring execution time: Emulators and sandboxes often execute code more slowly than real hardware. Thus, malware can use timing-based techniques to measure the execution time of certain operations to identify whether it's running in a controlled environment. To bypass this technique with Qiling, we use the Qiling search pattern (line 9) and patch this binary (line 13) in our below script.

[b] Unicorn engine. https://github.com/unicorn-engine/unicorn, 2021.

```
def patch_binary(qiling):
 patch_ = {
    'original': b'\x99\...',
    'patch': b'\x81\...'
 }
 patches.append(patch_)
 for patch in patches:
  #Search partern
  check = qiling.mem.search(patch['original'])
  if check:
   try:
     #Patch binary
     qiling.patch(check[0], patch['patch'])
     return
   except Exception as err:
        #Failed
```

3. *Process dump*: Once malware is actually executed in memory within our controlled environment, we will dump the memory process for feature extraction in Subsection 4.2. A process memory dump, also known as a memory snapshot or memory image, involves capturing the contents of a process's memory space at a specific point in time. This memory dump can provide valuable insights into the behavior of a running program.

As previously outlined, BAE incorporates the Run program, Bypassing, and Process dump procedures within the Qiling framework, creating a controlled environment that compels malware to execute and restore their malicious code. Utilizing Qiling, our sophisticated controlled environment can be more advanced than other sandbox systems like V-sandbox [30] and Tamer [31] designed specifically for IoT malware, which is based on Qemu. Qiling distinguishes itself from Qemu by offering dynamic analysis capabilities through its Python-based framework, facilitating dynamic instrumentation, runtime code patching, and cross-platform execution (e.g., Windows, Linux). In contrast, Qemu is confined to binary emulation without such extensibility and platform support, primarily targeting Linux and BSD environments [c]. Leveraging Qiling, the BAE method can be readily extended to various cross-platform executions.

4.2. Feature extractor

The dump memory file for each sample from Qiling is collected for feature extraction using the static analysis technique. In this study, we utilize Yara rules [d] to extract characteristics of malware. The below script is an example of Yara's rule for extracting thread injection from malware:

```
rule inject_thread {
 meta:
  description = "Code injection with CreateRemoteThread in a remote process"
 strings:
  $c1 = "OpenProcess"
```

[c] https://qiling.io/comparison/
[d] https://github.com/Yara-Rules/rules/blob/master/capabilities/capabilities.yar

```
  $c2 = "VirtualAllocEx"
  $c3 = "NtWriteVirtualMemory"
  $c4 = "WriteProcessMemory"
  $c5 = "CreateRemoteThread"
  $c6 = "CreateThread"
  $c7 = "OpenProcess"
 condition:
  $c1 and $c2 and ($c3 or $c4) and ($c5 or $c6 or $c7)}
```

To enhance the analysis, we leverage Yara's capabilities alongside digital signatures as key features. Yara utilizes a rule-based methodology to identify patterns associated with malware within files. These rules generally include strings, regular expressions, and specialized operators that define distinct attributes of malware families, combined with boolean logic to refine detection. Attributes are assigned values of 0 or 1 by checking for malicious behavior according to Yara's rules. By using the Yara's rules, 62 features are extracted as shown in Table 1.

Table 1: Features extracted by Yara's rules

Feature	Description	Feature	Description
InjectThread	Code injection in a remote process	CredLocal	Steal credential
CreateProcess	Create a new process	SniffAudio	Record Audio
Persistence	Install itself for autorun at Windows startup	CredFf	Steal Firefox credential
HijackNetwork	Hijack network configuration	CredVnc	Steal VNC credential
CreateService	Create a windows service	CredIe7	Steal IE 7 credential
CreateComService	Create a COM server	SniffLan	Sniff Lan network traffic
NetworkUdpSock	Communications over UDP network	MigrateApc	APC queue tasks migration
NetworkTcpListen	Listen for incoming communication	SpreadingFile	Malware can spread east-west file
NetworkDynDns	Communications dyndns network	SpreadingShare	Malware can spread east-west using share drive
NetworkToredo	Communications over Toredo network	RatVnc	Remote Administration toolkit VNC
NetworkSmtpotNet	Communications smtp	RatRdp	Remote Administration toolkit enable RDP
NetworkSmtpRaw	Communications smtp	RatTelnet	Remote Administration toolkit enable Telnet
NetworkSmtpVb	Communications smtp	RatWebcam	Remote Administration toolkit using webcam
NetworkP2pWin	Communications over P2P network	WinMutex	Create or check mutex
NetworkTor	Communications over TOR network	WinRegistry	Affect system registries
NetworkIrc	Communications over IRC network	WinToken	Affect system token
NetworkHttp	Communications over HTTP	WinPrivateProfile	Affect private profile
NetworkDropper	File downloader/dropper	WinFilesOperation	Affect private profile
NetworkFtp	Communications over FTP	StrWin32Winsock2Library	Match Winsock 2 API library declaration
NetworkTcpSocket	Communications over RAW socket	StrWin32WininetLibrary	Match Windows Inet API library declaration
NetworkDns	Communications use DNS	StrWin32InternetAPI	Match Windows Inet API call
NetworkSsl	Communications over SSL	StrWin32HttpAPI	Match Windows Http API call
NetworkDga	Communication using dga	MysqlDatabasePresence	This rule checks MySQL database presence
Bitcoin	Perform cryptocurrency mining	HasTLS	Has Thread Local Storage
Certificate	Inject certificate in-store	HasASLR	Has Address Space Layout Randomization
EscalatePriv	Escalade privileges	HasSEH	Has Structured Exception Handling
Screenshot	Take screenshot	HasCFG	Has Control Flow Guard
LookupIp	Lookup external IP	HasDEP	Has Data Execution Prevention flag
DynDns	Dynamic DNS	HasManifest	Has manifest
LookupGeo	Lookup Geolocation	SuspiciousDebugTs	Suspicious debug timestamp
Keylogger	Run a keylogger	CodeIntegrity	Code integrity

4.3. ML-based detector

The extracted features are then input into ML-based methods. All malware categories are designated as the first class, while benign samples are categorized as the second class. For the experiments, we utilize well-known classification methods, namely XGBoost with gradient-boosted decision trees, Random Forest, and KNN, for constructing the ML-based Detector in our BAE-MD framework. The objective is to assess the strength and robustness of the BAE method when paired with various established machine-learning techniques.

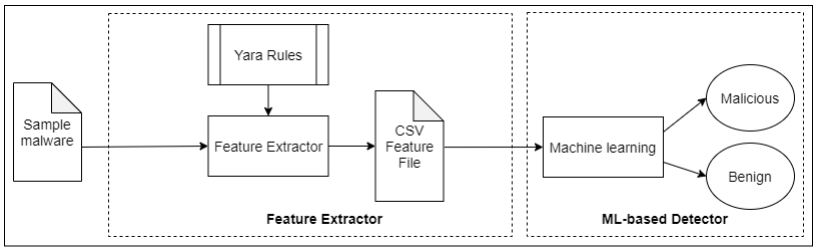

Figure 2: Malware Detection Framework with Static Analysis on PE (SAP)

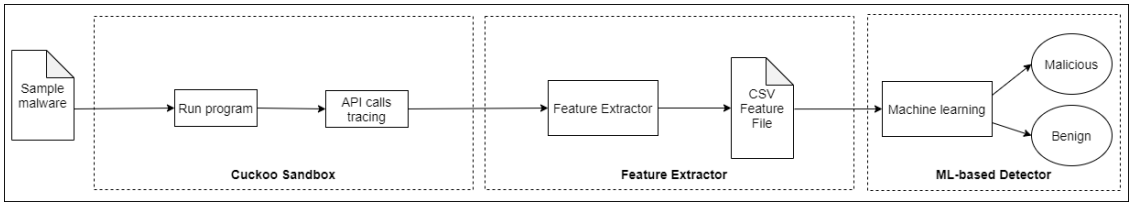

Figure 3: Malware Detection Framework with Dynamic Analysis on Cuckoo Sandbox (DAC)

The BAE-MD framework is an end-to-end solution designed to automatically identify malware. It can seamlessly integrate into malware detection systems, enhancing their detection capabilities, particularly against malware employing obfuscation techniques.

5. EVALUATION AND DISCUSSION

This section evaluates our proposed BAE-based malware detection framework (BAE-MD) on a malware dataset. The performance of BAE-MD is compared to other malware detection frameworks using the static analysis on PE files (SAP) and dynamic analysis on Cuckoo Sandbox (DAC) trace API calls [e]. The overview of the malware detection frameworks using SAP and DAC are presented in Figures 2 and 3 respectively.

Therefore, we designed two experiments to investigate the efficiency of our proposed method against the others. Firstly, we evaluate the performance of Random Forest when working with three different analysis techniques Bypass Anti-emulation method (BAE), SAP, and DAC to illustrate the efficiency of BAE. To demonstrate the consistency of BAE, we evaluate three different ML methods such as Random Forest, XGBoost, and KNN when working on BAE. Secondly, we measure the characteristics of malware in two execution cases with/without our Bypass Anti-emulation method to confirm the influence of our method. The details of the experiments and resulting discussion are presented in Subsections 5.1, 5.2, and 5.3.

5.1. Dataset

In this paper, we use 3,020 malware samples collected from MalwareBazaar [f] and Any.run [g]. The data consists of various malware categories such as Ransomware, Spyware, and Trojan

[e] https://github.com/mohamedbenchikh/MDML/blob/master/
[f] MalwareBazaar - https://bazaar.abuse.ch/browse/
[g] ANY.RUN - https://any.run

Horse as listed in Table 2. Similarly, benign samples collected only PE files from various applications in the Windows machine. The dataset contains a total of 6,040 records with 3,020 benign samples.

Once PE files are collected, BAE is operated to produce memory dump files for the dataset as illustrated in Figure 1. Then, the feature extractor component will transfer each memory dump file into a 62-feature record, and store it in a CSV format file. The dataset is randomly split into 70% for training and 30% for evaluation as shown in Table 2.

Table 2: Malware samples for training and testing

Lable data	Trojan Horse				Spyware				Ransomware									Benign	Total
	Zeus	Emotet	Refroso	Scar	TIBS	Coolwebsearch	Gator	Transponder	Conti	MAZE	Pysa	Ako	Revil (Sodinokibi)	GandCrab	SamSam	Ryuk	BadRabbit		
Count	199	200	198	195	194	197	190	199	193	201	207	190	132	124	130	129	142	3020	6040
Training	141	138	142	135	142	139	135	138	137	141	143	136	92	87	91	90	99	2114	4240
Testing	58	62	56	60	52	58	55	61	56	60	64	54	40	37	39	39	43	906	1800

5.2. Experimental settings

Qiling framework with Unicorn Engine is set up as the emulation environment for our experiments. It is installed on a machine with OS Ubuntu version 20.04, Intel Core i7-9700 CPU, and 16GB RAM. For machine learning methods, the Sklearn library is employed to construct ML-based models such as Random Forest, XGBoost, and KNN. The hyper-parameters of the ML-based models are set with default values (i.e. $n\_estimators = 50$ for Random Forest, $max\_depth = 6$ for XGBoost and $n\_neighbors = 3$ for KNN). To evaluate the performance of Malware detection frameworks in this paper, different metrics such as Accuracy, Weighted Average Precision, Weighted Average Recall, F1-score, and the Area Under the ROC Curve (AUC) are utilized.

5.3. Results and discussion

As mentioned above, two experiments are conducted for the evaluation of our proposed framework. Tables 3 and 4 present the performance of BAE-MD against these others using SAP and DAC, and three ML-based methods on BAE. The analysis results from malware in the two execution cases (with/without BAE) are shown in Tables 5, 6, and 7.

5.3.1. Performance in terms of detection accuracy

Table 3 shows that Random Forest works in conjunction with three malware analyses for feature extraction, SAP, DAC, and BAE. The performance of Random Forest (RF) is measured with five different metrics. The table clearly shows that Random Forest working with dynamic analysis environments such as DAC and BAE outperforms that with SAP. Additionally, RF with BAE produces the highest performance on all metrics (Accuracy, Precision, Recall, F1-Score, and AUC).

The results show that analyzing malware during execution in dynamic environments, like DAC and Qiling with BAE, can explore more characteristics than just using PE files. This can facilitate extracting valuable features for ML-based detectors like Random Forest. The Bypass Anti-emulation method (with the two functions of bypassing monitoring API calls and bypassing measuring execution time) forces malware to actually execute in our control environment. Thus, we can examine malicious code from obfuscated malware for feature extraction that can be done by DAC. This can result in a better performance of Random Forest on BAE.

Moreover, Table 4 demonstrates the consistency of three different ML-based methods working on BAE. Both Random Forest, XGBoost, and KNN produce competitive performance when measuring with the five metrics. Amongst them, KNN performs slightly worse than the others.

Table 3: Performance of Random Forest on the features extracted by SAP, DAC, and BAE

Feature Extraction with	Static Analysis on PE (SAP)	Dynamic Analysis on Cuckoo Sandbox (DAC)	Bypass Anti-emulation (BAE)
Classifiers	Random Forest		
Accuracy	0.725	0.949	**0.964**
Precision	0.724	0.942	**0.964**
Recall	0.723	0.956	**0.962**
F1-Score	0.724	0.959	**0.963**
AUC	0.725	0.951	**0.964**

Table 4: Performance of three classifiers on the features extracted by BAE

Feature Extraction with	Bypass Anti-emulation (BAE)		
Classifiers	KNN	XGBoost	Random Forest
Accuracy	0.952	**0.964**	**0.964**
Precision	0.939	0.961	**0.964**
Recall	0.961	**0.964**	0.962
F1-Score	0.950	0.962	**0.963**
AUC	0.953	**0.964**	**0.964**

5.3.2. Investigation characteristics of dumped files with BAE

To clarify the results in Subsection 5.3.1, we choose several malware files for investigation. Malware often employs obfuscated techniques that compress and encrypt their malicious code to evade malware analysis and detection methods. We execute these malware files in Qiling controlled by BAE for investigation. The resulting information is compared to that obtained from Qiling without BAE and from their PE files. Details of the investigation are presented as follows:

Evaluate strings, the execution time, API calls, and Entropy: An obfuscated malware [h] is chosen for investigating strings, the execution time, API calls, and Entropy. The

[h] f5bf0c3e96b075995e0551785367891eea641dd9e1092c3808210753542d11e7

Table 5: Strings extracted from the obfuscated file and the memory dump file

Strings			
Obfuscated file		Memory dumped file	
LoadLibraryA	ExitProcess	CloseHandle	UnmapViewOfFile
GetProcAddress	VirtualProtect	IsBadReadPtr	MapViewOfFile
l;c	5it]{	CreateFileMappingA	CreateFileA
Xcp4c	{ms	FindClose	FindNextFileA
t_fdi9Hcommode	_typl	FindFirstFileA	CopyFileA
_h1I37	olfp	KERNEL32.dll	malloc
7 dy| Wv	Il.t	exit	MSVCRT.dll
B`.rd	@.	_exit	_XcptFilter
0'?lJ	u A	__p___initenv	__getmainargs
GIu	PTj	_initterm	__setusermatherr
XPTPSW	KERNEL32.DLL	_adjust_fdiv	__p__commode
MSVCRT.dll		__set_app_type	_except_handler3
		_controlfp	_stricmp
		.exe	C:*

strings extracted from the obfuscated file are compared to those extracted from the memory dump file controlled by BAE. The results from Table 5 confirm that when executing in our controlled environment, a larger number of strings are found. Additionally, these strings seem to be more likely real than extracted from the obfuscated file.

Moreover, the execution time, the number of API calls, and the entropy value are analyzed. Firstly, the above malware is executed in the Qiling environment in two cases with and without BAE. We then measure the execution time, the number of API calls, and the entropy as shown in Table 6. The results show that applying the bypass anti-emulation method influences on the malware behavior such as the execution time, the number of API calls, and the entropy. Entropy is a measure of the randomness or disorder within data. Low entropy values often indicate high structured and regular patterns, while high entropy values can suggest low randomness or complexity. Therefore, the lower entropy value (1.95) on the memory dump file using BAE in comparison with that (6.11) not using BAE can indicate that the BAE method has forced the malware to decompress and decrypt its content to the memory. In addition, the execution time and number of API calls (6.5s and 12) are larger than those not using BAE (1.5s and 4).

Again, these investigating results on strings, execution time, number of API calls, and entropy can confirm the efficiency of the BAE method.

Table 6: The execution time, the number of API calls, and the entropy from malware when it runs in Qiling with/without BAE

Type	Without Bypass Anti-emulation	With Bypass Anti-emulation
Time execution	1.5s	6.5s
Number of API call	4	12
Entropy	6.11	1.95

Yara rules: We randomly choose 25 malware samples from the training data shown in Table 2 for investigating the number of matching rules. The same Yara rules used for feature extraction in Subsection 4.2 are employed. These rules are applied to the original PE

files and the memory dump files extracted from the Qiling environment controlled by BAE. For each malware, Table 7 presents the SHA-256 hash along with the number of matching rules on its PE file and its memory dump file. The values in bold indicate a larger number of matching rules found on the memory dump files.

Because original samples are obfuscated programs. Obfuscated programs often compress and encrypt their malicious codes which cannot be analyzed. This data can be decompressed and decrypted as the program runs in our controlling environment with BAE. This can restore the malicious code by creating a memory region and writing the decoded data. Therefore, the number of matching rules on the memory dump files obtained from our controlled environment is often larger than that done on the original PE files. This can be explained that why our proposed framework performs more efficiently than others using SAP and DAC.

Table 7: Numer of Yara rules matched on the PE file and memory dump file from the Qiling environment with BAE

No.	Hash SHA-256 of malware	Number of rules matched on malware	
		Original PE file	Memory dump file
1	1af55c95620f18d4ad92ff28e83dd14ce1daaa77b7709beaa4cfe8652bde6f36	5	5
2	1c06a90840409f4635e8f9d959f482ed40379571595e2546926429f974998d21	12	12
3	2f86f8b37ce86cfe16419d0fbeb1d0ef2f37032f6c630ed1d14649f327c27aa0	0	**4**
4	3cf18715e4e52e503845221167ad276e90f614dd8142172d5a4a4f76b937bfe4	0	**5**
5	5e9f2794e4c145fbd68caacb0ddf07d1d35fc9b8b748b2efd1b4063e489f8223	5	**7**
6	6c6491a4d68635154b4b1ebcb72bd6f89493c0b44ed769b9a558888244efec18	5	**8**
7	6e855e9b6706e7b583345ce8fff8776c63ef5b36cb897cc4ed4a452722d3ad89	0	**2**
8	07bd5beb8d2042ce158e3debe0e63d1494816827384d31c87361ba8fd24b2d55	0	**1**
9	08a840677baaa0b14152850e1e0923aaa819ae8a2ecd7923f9510e1141962f16	0	**3**
10	8cc440eff0de4c70b4427d2d0332dd8ccbadb36ead79bd1db5bc67b665bd3fe2	0	**4**
11	9ab21141018bf9b7884f4dba96fcd4d184bc1fb913f38e8267decba39f78b376	0	**2**
12	9acefbb6638612b03847da2c1652e3a1aae9d677b22f5c5d0e43022d8d08c6bb	4	**5**
13	9e0a919d3d424638da51b979be48222f565f97a1f21a536e3fe56e067ae80401	13	13
14	16d662bcb526f0bc319671cb02488c7d37a70925d73a04d79c25e3ce0abb5253	0	**3**
15	23a188b67111d6c67ba62e1588479154ca23c4c65d768a662b873757a3419ed0	0	**2**
16	34a0f848bbcf609398fbffbc14a3b070f6e5c15c4987785c29db8de7d46f9bd6	1	**3**
17	49b2e08cf7fb9bceaf2721ef24c9ab795c984403c258af9df3914dee1f3225a0	0	**2**
18	58e6a469f1ace9ec112de054209783ad6dd469a0794f20a998a0dcdf02a4834e	0	0
19	59da9f40387363fb12e59349fa8f47535f80abe5ac07d87e20f42c547e176864	5	5
20	76de16b596ad3700130d2d2c02a9ca144ace99bef78a7088b93f069673cbe972	0	**2**
21	82a294aa5072baca70b941c44def34063e052ef781a1673ebc65071cffba647e	1	1
22	226f1fce2b39e4769507402b282864222d091d786344511d4fdf4cf9c3d2c049	0	0
23	236c73a241d229cc820b4fa2aa914403151deb84b90939ac4760460fc107dda4	5	5
24	664e98a05e0cdd62d0f97525f2255f1c19b5b8a1d8091a362ef5fbc007c2715c	0	**2**
25	954fe5a029ddc55acc658311beb82b95c6755a07101efb9cbe631a42e2bd00ef	0	**4**

6. CONCLUSIONS AND FUTURE WORK

This study introduces the Bypass Anti-emulation-based Malware Detection framework (BAE-MD) for improving the efficiency of detecting obfuscated malware. By circumventing malware's anti-emulation mechanisms within dynamic environments, BAE-MD enables the decryption and decompression of malicious code directly into memory. This can facilitate obtaining valuable features for ML-based malware detection methods. Our proposed framework is evaluated on a malware dataset and compared to alternative approaches employing static and dynamic analysis techniques for feature extraction. The experiments demonstrate that

BAE-MD outperforms existing methods in terms of detection accuracy and efficiency. The task of investigating other execution format files and adapting BAE-MD on Linux systems is postponed to future work.

REFERENCES

[1] J. Singh and J. Singh, "A survey on machine learning-based malware detection in executable files," *Journal of Systems Architecture*, vol. 112, p. 101861, Jan. 2021. [Online]. Available: https://doi.org/10.1016/j.sysarc.2020.101861

[2] W. Han, J. Xue, Y. Wang, L. Huang, Z. Kong, and L. Mao, "MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics," *Computers & Security*, vol. 83, pp. 208–233, Jun. 2019. [Online]. Available: https://doi.org/10.1016/j.cose.2019.02.007

[3] R. Chaganti, V. Ravi, and T. D. Pham, "A multi-view feature fusion approach for effective malware classification using deep learning," *Journal of Information Security and Applications*, vol. 72, no. C, pp. 103 402–103 417, Feb. 2023. [Online]. Available: https://doi.org/10.1016/j.jisa.2022.103402

[4] D. Amir, B. Ahmed, R. Saddaf, and M. M. Ibrahim, "Artificial intelligence-based malware detection, analysis, and mitigation," *Symmetry*, vol. 15, no. 3, Feb. 2023. [Online]. Available: https://doi.org/10.3390/sym15030677

[5] S. Yan, J. Ren, W. Wang, L. Sun, W. Zhang, and Q. Yu, "A survey of Adversarial attack and defense methods for malware classification in cyber security," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 467–496, Mar. 2023. [Online]. Available: https://doi.org/10.1109/COMST.2022.3225137

[6] Ö. Aslan, S. S. Aktuğ, M. Ozkan-Okay, A. A. Yilmaz, and E. Akin, "A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions," *Electronics*, vol. 12, no. 6, p. 1333, Mar. 2023. [Online]. Available: https://doi.org/10.3390/electronics12061333

[7] K. Khan, A. Mehmood, S. Khan, M. A. Khan, Z. Iqbal, and W. K. Mashwani, "A survey on intrusion detection and prevention in wireless ad-hoc networks," *Journal of Systems Architecture*, vol. 105, p. 101701, May 2020. [Online]. Available: https://doi.org/10.1016/j.sysarc.2019.101701

[8] M. A. Fatemeh Deldar, "Deep learning for zero-day malware detection and classification: A survey," *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–37, Sep. 2023. [Online]. Available: https://doi.org/10.1145/3605775

[9] M. S. Akhtar and T. Feng, "Malware analysis and detection using machine learning algorithms," *Symmetry*, vol. 14, no. 11, p. 2304, Nov. 2022. [Online]. Available: https://doi.org/10.3390/sym14112304

[10] V. D. Quang, "Enhancing obfuscated malware detection with machine learning techniques," *Communications in Computer and Information Science*, vol. 1688, pp. 731–738, Nov. 2022. [Online]. Available: https://doi.org/10.1007/978-981-19-8069-5_54

[11] F. Jannatul, I. Rafiqul, M. Arash, and I. Md Zahidul, "A review of state-of-the-art malware attack trends and defense mechanisms," *IEEE Access*, vol. 11, pp. 121 118–121 141, 2023. [Online]. Available: https://doi.org/10.1109/ACCESS.2023.3328351

[12] M. M. Alani, A. Mashatan, and A. Miri, "XMal: A lightweight memory-based explainable obfuscated-malware detector," *Computers & Security*, vol. 133, p. 103409, Oct. 2023. [Online]. Available: https://doi.org/10.1016/j.cose.2023.103409

[13] H. J. Asghar, B. Z. H. Zhao, M. Ikram, G. Nguyen, D. Kaafar, S. Lamont, and D. Coscia, "Use of cryptography in malware obfuscation," *Journal of Computer Virology and Hacking Techniques*, vol. 20, no. 1, pp. 135–152, Mar. 2024. [Online]. Available: https://doi.org/10.1007/s11416-023-00504-y

[14] R. Ali, A. Ali, F. Iqbal, M. Hussain, and F. Ullah, "Deep learning methods for malware and intrusion detection: A systematic literature review," *Security and Communication Networks*, vol. 2022, no. 1, p. 2959222, Oct. 2022. [Online]. Available: https://doi.org/10.1155/2022/2959222

[15] U.-e.-H. Tayyab, F. B. Khan, M. H. Durad, A. Khan, and Y. S. Lee, "A survey of the recent trends in deep learning based malware detection," *Journal of Cybersecurity and Privacy*, vol. 2, no. 4, pp. 800–829, Sep. 2022. [Online]. Available: https://doi.org/10.3390/jcp2040041

[16] M. T. Nguyen, V. H. Nguyen, and N. Shone, "Using deep graph learning to improve dynamic analysis-based malware detection in PE files," *Journal of Computer Virology and Hacking Techniques*, vol. 20, no. 1, pp. 153–172, Sep. 2024. [Online]. Available: https://doi.org/10.1007/s11416-023-00505-x

[17] M. Kim, H. Cho, and J. H. Yi, "Large-scale analysis on anti-analysis techniques in real-world malware," *IEEE Access*, vol. 10, pp. 75 802–75 815, Jul. 2022. [Online]. Available: https://doi.org/10.1109/ACCESS.2022.3190978

[18] A. Sharma, B. B. Gupta, A. K. Singh, and V. Saraswat, "Orchestration of apt malware evasive manoeuvers employed for eluding anti-virus and sandbox defense," *Computers & Security*, vol. 115, p. 102627, Apr. 2022. [Online]. Available: https://doi.org/10.1016/j.cose.2022.102627

[19] P. Rehida, G. Markowsky, A. Sachenko, and O. Savenko, "State-based sandbox tool for distributed malware detection with avoid techniques," in *2023 13th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. IEEE, Oct. 2023, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/DESSERT61349.2023.10416467

[20] G. I. Stoleru and D. T. Gavrilut, "A practical approach for malware identification based on anti-emulation techniques and feature to image translation," in *2021 31st International Conference on Computer Theory and Applications (ICCTA)*. IEEE, Oct. 2021, pp. 133–140. [Online]. Available: https://doi.org/10.1109/ICCTA54562.2021.9916624

[21] M. Dener, G. Ok, and A. Orman, "Malware detection using memory analysis data in big data environment," *Applied Sciences*, vol. 12, no. 17, p. 8604, Aug. 2022. [Online]. Available: https://doi.org/10.3390/app12178604

[22] W. Qiang, L. Yang, and H. Jin, "Efficient and robust malware detection based on control flow traces using deep neural networks," *Computers & Security*, vol. 122, p. 102871, Nov. 2022. [Online]. Available: https://doi.org/10.1016/j.cose.2022.102871

[23] K. Shaukat, S. Luo, and V. Varadharajan, "A novel deep learning-based approach for malware detection," *Engineering Applications of Artificial Intelligence*, vol. 122, p. 106030, Jun. 2023. [Online]. Available: https://doi.org/10.1016/j.engappai.2023.106030

[24] M. Gopinath and S. C. Sethuraman, "A comprehensive survey on deep learning based malware detection techniques," *Computer Science Review*, vol. 47, p. 100529, Feb. 2023. [Online]. Available: https://doi.org/10.1016/j.cosrev.2022.100529

[25] A. R. Pandey, T. Sharma, S. Basnet, and S. Setia, "Static analysis approach of malware using machine learning," in *International Conference on Recent Developments in Cyber Security*. Springer, Mar. 2023, pp. 109–121. [Online]. Available: https://doi.org/10.1007/978-981-99-9811-1_9

[26] H. K. Singh, J. P. Singh, and A. S. Tewari, "Static malware analysis using machine and deep learning," *Proceedings of International Conference on Computing and Communication Networks*, Jul. 2022. [Online]. Available: https://doi.org/10.1007/978-981-19-0604-6_41

[27] J. von der Assen, A. H. Celdrán, A. Zermin, R. Mogicato, G. Bovet, and B. Stiller, "Secbox: A lightweight container-based sandbox for dynamic malware analysis," *IEEE/IFIP Network Operations and Management Symposium*, pp. 1–3, Jun. 2023. [Online]. Available: https://doi.org/10.1109/NOMS56928.2023.10154293

[28] N. M. Tu, N. V. Hung, P. V. Anh, C. Van Loi, and N. Shone, "Detecting malware based on dynamic analysis techniques using deep graph learning," in *Future Data and Security Engineering: 7th International Conference, FDSE 2020, Quy Nhon, Vietnam, November 25–27, 2020, Proceedings 7*. Springer, Nov. 2020, pp. 357–378. [Online]. Available: https://doi.org/10.1007/978-3-030-63924-2_21

[29] F. A. Aboaoja, A. Zainal, A. M. Ali, F. A. Ghaleb, F. J. Alsolami, and M. A. Rassam, "Dynamic extraction of initial behavior for evasive malware detection," *Mathematics*, vol. 11, no. 2, p. 416, Jan. 2023. [Online]. Available: https://doi.org/10.3390/math11020416

[30] H.-V. Le and Q.-D. Ngo, "V-sandbox for dynamic analysis IoT botnet," *IEEE Access*, vol. 8, pp. 145 768–145 786, Aug. 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.3014891

[31] S. Yonamine, Y. Taenaka, Y. Kadobayashi, and D. Miyamoto, "Design and implementation of a sandbox for facilitating and automating IoT malware analysis with techniques to elicit malicious behavior: case studies of functionalities for dissecting IoT malware," *Journal of Computer Virology and Hacking Techniques*, vol. 19, no. 2, pp. 149–163, May 2023. [Online]. Available: https://doi.org/10.1007/s11416-023-00478-x

Received on May 07, 2024
Accepted on July 10, 2024