


2018

A Malware Analysis and Artifact Capture Tool

Dallas Wright
Dakota State University

Josh Stroschein
Dakota State University

Follow this and additional works at: <https://scholar.dsu.edu/ccspapers>

 Part of the [Information Security Commons](#), [Programming Languages and Compilers Commons](#), [Software Engineering Commons](#), and the [Systems Architecture Commons](#)

Recommended Citation

Wright, Dallas and Stroschein, Josh, "A Malware Analysis and Artifact Capture Tool" (2018). *Faculty Research & Publications*. 8.
<https://scholar.dsu.edu/ccspapers/8>

This Conference Proceeding is brought to you for free and open access by the Beacom College of Computer and Cyber Sciences at Beadle Scholar. It has been accepted for inclusion in Faculty Research & Publications by an authorized administrator of Beadle Scholar. For more information, please contact repository@dsu.edu.

A Malware Analysis and Artifact Capture Tool

Dallas Wright
Cyber Security
Dakota State University
Madison South Dakota, USA
e-mail: dallas.wright@trojans.dsu.edu

Josh Stroschein, D.Sc
Assistant Professor Cyber Operations
Dakota State University
Madison South Dakota, USA
e-mail:joshua.stroschein@dsu.edu

Abstract— Malware authors attempt to obfuscate and hide their code in its static and dynamic states. This paper provides a novel approach to aid analysis by intercepting and capturing malware artifacts and providing dynamic control of process flow. Capturing malware artifacts allows an analyst to more quickly and comprehensively understand malware behavior and obfuscation techniques and doing so interactively allows multiple code paths to be explored. The faster that malware can be analyzed the quicker the systems and data compromised by it can be determined and its infection stopped. This research proposes an instantiation of an interactive malware analysis and artifact capture tool.

Keywords—Malware, Malware Analysis, Windows, Debuggers, Decompilers, Disassemblers, Obfuscation, Sandbox

I. INTRODUCTION

Malware is malicious software intentionally designed to harm data, computers, and other devices or people. There are numerous classifications of malware including ransomware, trojans, rootkits, keyloggers, backdoors and others [1, 2]. Malware may be used to gather intelligence, alter data or even hold the files on a device or computer for ransom. It has been used to compromise individual financial records [3], financial markets [4], voting [5] and utility infrastructure [6]. The threat of malware infections has brought about the need for large investments of capital by utility companies, corporations and government entities at all levels [7].

Malware analysis is the study of the method of execution of malware and its effects on host systems. There are two general types of malware analysis, static and dynamic. Static analysis is the study of malware, without execution, of the program in a binary state. This analysis usually requires the disassembly and/or de-obfuscation of the malware's source or binary code. Dynamic malware analysis is the study of malware while it is executing. This typically involves executing the malware in a virtual test environment and studying its behavior [8]. Malware authors employ code obfuscation to deter both dynamic and static analysis [9]. This includes obfuscation of code by functionally encrypting source code, numeric representations of characters, non-base 10 numbers, unused variables, misleading variable names, unneeded instructions and modifying program flow to confuse, and encryption.

During the execution of malware numerous artifacts are often created. The artifacts can be useful in the analysis and

detection of the malware. Some artifacts are often short lived existing only for brief periods of time making the task of identifying and capturing such artifacts difficult [9]. Interactive debuggers can be used to stop the execution of the malware at the moment they exist, but this is a manual process. It can be quite difficult to identify the precise moment when an artifact is created and in a non-obfuscated state.

There are tools that attempt an automatic analysis of malware including the capture of their artifacts, but these tools can be costly to set up and use, may not trigger certain portions of the code and have limited interaction with the malware analyst. To address these issues, this work proposes a framework which could be used to automatically detect and capture malware artifacts, provide for customization, and log malware activity as well as provide a level of interactivity for the control of program flow and malware analysis focus. The artifact produced by this research will be a Malware Analysis and Artifact Capture Tool (MACT).

II. PAST WORK AND CURRENT CONTRIBUTION

Antivirus scanners attempt to detect malware using syntactic signature detection. This method is a form of static analysis that is fast and returns few false positives. One weakness associated with syntactic signature detection is its inability to effectively detect new malware or obfuscated variants of existing malware [10]. Obfuscation is a technique by which a program's functionality can remain intact while making it more difficult to detect and/or understand. Variants of malware are often modified to appear and execute in a different manner while still performing the same nefarious functions. Malware writers attempt to obfuscate their code and defeat analysis and detection by antivirus scanners [11].

A polymorphic virus transforms itself dynamically to try to evade detection. Metamorphic viruses attempt to evade detection techniques by changing their code [9]. Obfuscations can be employed to hide binary code, prevent accurate disassembly, and mask flow. There are various methods used to implement obfuscation including, code unpacking, code overwriting, ambiguous code and data, obfuscated calls and returns, call-stack tampering, calling convention violations, and no-op code [12]. The effect of obfuscation does not limit itself to antivirus scanner detection of malware it also complicates malware analysis.

Operating system Application Program Interfaces (API) are functions employed to abstract and perform common low-

level system functions. API hooking is a method by which function calls can be intercepted during execution and then allowing different or additional functionality to be employed. There are numerous research papers relating to the use of API hooking to perform malware detection and analysis. It is claimed that a program can be identified as malware by looking at the API calls a process makes along with the parameters used and the frequency and order in which they are called [13]. The research of Ye, Wang, et al. used API call analysis to demonstrate a 92% detection rate and a faster detection time with fewer false positives [14]. Further, it is claimed by many researchers that by using API analysis it is also possible to identify the family of malware to which the program belongs [15]. Lee, Jeong, et al. have proposed creating a Control Flow Graph (CFG) from the API call sequence to create a semantic signature representation of a binary [11]. The authors claim that using this semantic signature instead of the typical syntactic signature results in more effective detection of malware.

One class of tools that utilizes API hooking are sandboxes. Sandboxes are virtual environments used to isolate the effects of executing programs from the host environment on which they run. Sandboxes are commonly used tools and are therefore targeted for evasion by many malware creators [16]. Sandbox evasion techniques and limitations include requesting human interaction such as dialog boxes, sleeping, masking processes, limited code traversal, and modifying behavior after detecting the sandbox or execution in a virtual environment [17, 18]. Because of sandbox specific evasion techniques and limitations exploited by malware authors, dynamic analysis using sandboxes is not as reliable of a technique as it once was [19].

This research contributes to the analysis of malware by extending the use of API hooking and combining them with interactive functionality to overcome some limitations in the existing tools. A novel malware analysis focused tool that provides both interactivity during dynamic analysis and customizable functionality for automated logging, more comprehensive path execution and artifact capture is proposed.

III. PROBLEM RELEVANCE

Several automatic malware analysis systems and services exist including Cuckoo Sandbox, ProcMon, Malwr, ThreatExpert and others. While these tools should be utilized as they do provide useful information, they lack a high level of malware analyst interactivity, customization during dynamic analysis, limited path traversal and have shortcomings that are exploited by malware authors [19]. A framework that allows the analyst to control the execution path of the sample, customize the data logged during interaction with the OS and pause execution would address significant limitations of these existing tools.

Interactive dynamic debuggers, disassemblers, and decompilers are available that can be used to pause execution, examine memory, set breakpoints, examine instructions and code [9]. These tools include IDA, WinDbg, OllyDbg, Intel Debugger and others. These are effective tools, but they are general reverse engineering tools not tailored specifically toward malware analysis. They are designed to look at binaries and trace process execution. These tools have functionality

and customization that can be utilized to perform some functions required during malware analysis. Unfortunately, some of this customization and associated setup must be performed each time the tool is used and is not specialized to malware analysis. Additionally, the data these tools return must be interpreted, filtered and possibly be reanalyzed with other tools.

While malware analysis is a form of reverse engineering there are certain phases, techniques and execution properties that malware analysts systematically and repeatedly employ that could be better served with a tool more directly suited to their needs. Having a tool specifically designed to dynamically control execution flow, monitor system API's, monitor memory, log system events and capture artifacts would allow a malware analyst to more efficiently and accurately examine malware. It would increase efficiency by limiting configuration time with fewer and more malware specific options and defaults, provide a more functionally direct interface and more relevant reporting.

IV. DESIGNING MACT

To capture malware artifacts, a Malware Artifact Analysis and Artifact Capture Tool (MACT) instantiation is proposed. The general design objectives for the tool are 1) rules that identify specific calls of an API as calls of interest, 2) interception of relevant APIs, 3) for each API of interest a method of information capture must be defined and coded 4) detailed logging of events and tool actions and 5) user interactivity.

When performing the injected behavior, injected code will be programmed to differentiate normal execution behavior with that of malware. For example, allocation of memory on the heap with VirtualAlloc may be acceptable and hence its execution not monitored or logged if it doesn't attempt to make memory executable. These rules will need to be established and coded on a function by function basis.

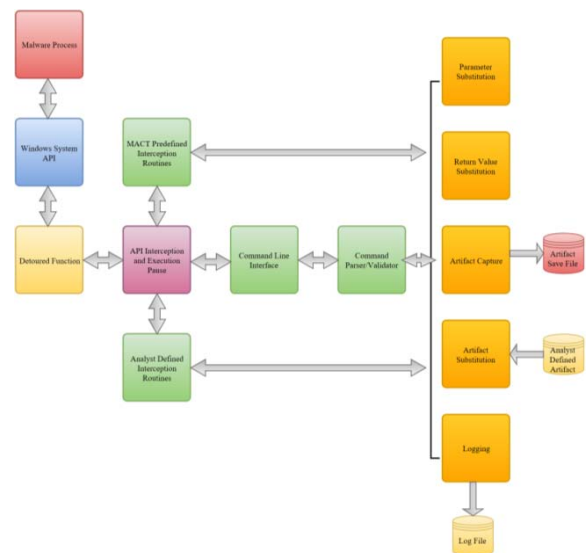


Figure 1. MACT System Flow

Once the Windows APIs of interest are chosen and their associated functions defined the tool can inject the hooks into the malware process. During execution, the injected code can perform the logging, capturing of artifacts and monitoring.

The captured malware artifacts will provide information useful to a malware analyst. By examining these artifacts, insight can be gained as to the methods and goals of malware. The tool can be fine-tuned for a specific sample of malware to capture additional and more relevant data.

The ability to identify specific artifacts can be used to simplify the analysis of malware by using previously analyzed artifact results. By using these more atomic malware artifacts it will be easier to identify encrypted, metamorphic, polymorphic and obfuscated versions of the malware. This can lead to faster analysis and source attribution.

Each injected function will be responsible for performing relevant logging. This includes standard logging items such as API name, parameters, event messages and API specific events such as the launching of a monitor for memory related APIs. Logging of events can be used to piece together the method of attack. It can also provide information that can guide the analyst to focus on different types of calls or process paths and to modify the hooked function to gather more relevant information.

Monitoring may need to be established based on specific API calls. The monitoring can capture what is written at specific memory locations and then serialize it. It can also be used to track memory, files, registry settings, and other processes. The monitoring can be performed by the spawning of threads from injected functions. For example, if the malware allocates memory and makes it readable, writable and executable it would be prudent to monitor writes to that memory location as it is possible that an artifact is being created and will be executed from that location.

MACT can also provide the capability to pause process execution when a Windows API is called. This pause can allow for the analysis of other processes, memory, files, system registry and external communication with the MACT tool or other tools.

V. THEORETICAL FRAMEWORK

The theoretical framework to be used for this research will be that of a single case mechanism study [20]. Each sample in a malware sample set will define a unique case for the purposes of this study. This framework can be used to emphasize the differences and similarities between selected commonly used malware analysis tools and the proposed instantiation that is the artifact created for this research. The comparisons will be used to evaluate MACT's effectiveness in performing malware analysis.

This framework will allow a quantitative comparison of the artifacts of analysis produced by all the tools involved in the study. A set of malware samples will be chosen and analyzed using each tool to extract the artifacts. Each malware analysis related artifact identified by MACT will be captured and saved. Counts will be kept by each type of analysis element produced by tool and sample. A tabular representation of the results will be created to aid evaluation.

VI. WINDOWS APIS

Microsoft Windows APIs allow user programs to perform common functions in a uniform way as well as to access system functionality through the kernel [13]. They abstract the low-level functionality from an application. These include creating files, reading files, allocating memory, writing to memory, accessing the network, accessing the internet, reading and writing registries. Legitimate applications and malware both use APIs to perform their functions.

An application's functionality determines the types, parameters and sequences of API calls. There have been various studies that claim these API characteristics can be used to classify applications as malware and identify which class of malware. They can even be used in place of binary signature detection to identify malware and to improve upon the success of binary signature detection to identify previously unregistered polymorphic malware [15].

Malware often uses specific APIs to form the framework upon which it builds and obfuscates its attacks. Some of these API functions include VirtualAlloc to allocate memory, CreateProcess to create a new process, AdjustTokenPrivileges to modify access privileges etc. While these API functions have perfectly legitimate uses in benign software, a malware analyst may look for these calls in a program as they are often used by malware to perform its functions.

VII. API HOOKING

Hooking is the act of redirecting the program flow of a process by jumping to an address and executing injected code. Hooking is typically achieved by one of three methods, these are the Import Address Table (IAT) hook, debugger hook, and inline hook [13]. The Import Address Table hook modifies the Import Address Table in the header of the PE file. Inside this table are pointers to the API code. The pointers are modified to point to a stub that logs the API calls and returns to the actual API. The debugger hook occurs when a breakpoint is set from within a debugger. In this case the entry point is overwritten with an INT 3 that causes the CPU to throw a debug exception. Inline hooking modifies the entry point of an API with code that jumps to a different function called the *detour* function. Changing the entry point modifies the original code so it is important to save the code overwritten so it can be executed later [21].

Starting with Windows XP SP2, Microsoft introduced a method by which they could perform hot patching. To implement hot patching functionality Microsoft added five bytes of data in the preamble of each API function that they could use to redirect the flow of processing and execute different or additional code. This provided an easy mechanism for inline hooking. The dummy code inserted by Microsoft was MOV EDI, EDI. By replacing the dummy instruction with a jump to another address the flow of the process can be altered [22]. Fig. 2 illustrates the implementation of the patch code in the preamble.

```

arg_0      = dword ptr 8
arg_4      = dword ptr 0Ch

; FUNCTION CHUNK AT .text:63A90F44 SIZE
mov     edi, edi
push   ebp
mov     ebp, esp
push   ecx
mov     eax, [eax+8]
push   esi

```

Figure 2. Patch code in the function preamble.

VIII. USING API CALLS TO CAPTURE API ARTIFACTS

As mentioned previously, artifacts created by malware may only exist for a short time during the execution of a process before they are modified, deleted or hidden. Capturing these artifacts during dynamic analysis can be difficult and time consuming. Halting execution at the exact instruction during which the artifacts are complete and whole is challenging.

API hooking allows for altering the flow of a process and the execution of analyst code to perform other or additional functionality. After the analyst's code is executed the flow can be returned to the original function. Inline hooking can provide this functionality dynamically while not requiring the original program to be modified.

The user code jumped to can log API functions that are indicative of malware. These include allocation of memory, creation of threads, manipulation of files, modification of the Windows registry and adjustment of system privileges. With this information, the user code can further examine what is occurring and log the relevant information to be examined later. The user code may also be able to capture artifacts based on addresses used in memory allocation, process creation or thread creation.

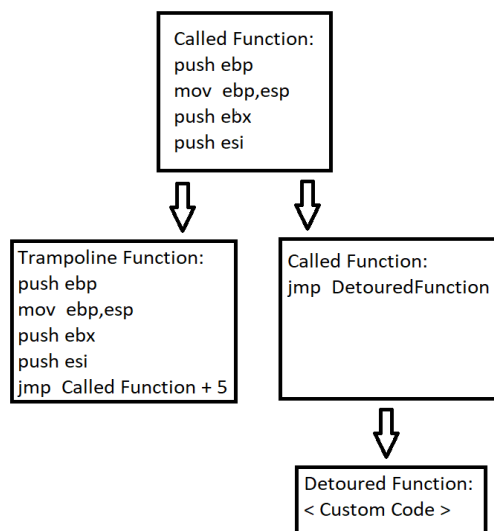


Figure 3. API Interception

Additionally, the user code could be used to check the system state by looking at changes in the Windows registry, memory, processes, etc. by comparing them to a previous system state [23][24]. These changes can be serialized, processes tracked, and malicious changes uncovered thus giving insight into what the malware is doing and how it is accomplishing its goals. These state checks could provide a method to detect malicious activity that is not apparent solely because of specific API calls.

IX. APIS OF INTEREST

There are numerous APIs malware could use to perform their tasks. For the purposes of this research, the categories of APIs for interception will be defined as the following:

- **Memory Management:** VirtualAlloc, VirtualProtect, CopyMemory, MoveMemory, HeapAlloc, etc.
- **Windows Registry Modification:** RegReplaceKey, RegSetValueEx, RegSaveKey, etc.
- **Process and Thread Handling:** CreateProcess, GetCurrentProcess, CreateThread, etc.
- **File Management:** CreateFile, GetTempPath, ReplaceFile, etc.
- **External Communication:** connect, recv, send, TransmitFile, TransmitPackets, etc.
- **Dynamic-Link Library Manipulation:** LoadLibrary, GetProcAddress, GetModuleFileName, etc.

These categories include some of the more common APIs used by malware [25]. The APIs to be intercepted by MACT can be adjusted to hook more APIs as deemed necessary. The list used may be tailored and modified during malware analysis to prevent the unnecessary interception of some calls or to add additional APIs to intercept.

X. VALIDATING MACT

To validate the Malware Analysis and Artifact Capture Tool, it will be necessary to verify its results against traditional malware analysis techniques. For the first phase of testing, a test program will be written which uses targeted and other APIs to create artifacts and trigger interception. Additionally, benign calls will be included in the test program as well as deploying different code obfuscation and encryption techniques. The second phase will consist of comparing the captured artifacts against the known artifacts of malware samples. These are available through various online repositories such as virustotal.com.

During the first phase of validation, additional APIs or enhanced interception functional requirements can be identified and implemented. Recursive testing, improvement and re-evaluation can be employed to fine tune the API interception functions as well as the monitoring and logging functionality of the tool.

The second phase of validation will consist of comparing analysis elements. To validate the effectiveness of MACT, it will be necessary to gather the totals of unique malware analysis elements identified by the commonly used tools. The elements captured will be categorized by type. The results of the research will be presented as shown in Table I for each sample in the set.

Table I. Malware Analysis Tools Comparison

Tool	Classification	Registry	File System	Memory	Processes	Communication	API	Artifact
MACT	c_0	r_0	e_0	m_0	p_0	u_0	i_0	a_0
ANLYZ	c_1	r_1	e_1	m_1	p_1	u_1	i_1	o_1
CWSandBox	c_2	r_2	e_2	m_2	p_2	u_2	i_2	o_2
HybridAnalysis	c_3	r_3	e_3	m_3	p_3	u_3	i_3	o_3
IDA	c_4	r_4	e_4	m_4	p_4	u_4	i_4	o_4
Malwr	c_5	r_5	e_5	m_5	p_5	u_5	i_5	o_5
Sandbox Pikker	c_6	r_6	e_6	m_6	p_6	u_6	i_6	o_6
ThreatExpert	c_7	r_7	e_7	m_7	p_7	u_7	i_7	o_7
ViCheck	c_8	r_8	e_8	m_8	p_8	u_8	i_8	o_8
WinDbg	c_9	r_9	e_9	m_9	p_9	u_9	i_9	o_9

The analysis elements by type are placed in a set with each element type per tool.

- Classification = $\{c_1...c_n\}$ with $f(c)$ representing the malware effect on the classification
- Registry = $\{r_1...r_n\}$ with $f(r)$ representing the malware effect on the registry.
- File System = $\{e_1...e_n\}$ with $f(e)$ representing the malware effect on the file system.
- Memory = $\{m_1...m_n\}$ with $f(m)$ representing the malware effect on the memory.
- Processes = $\{p_1...p_n\}$ with $f(p)$ representing the malware effect on the processes.
- Communications = $\{u_1...u_n\}$ with $f(u)$ representing the malware effect on the communications.
- API = $\{i_1...i_n\}$ with $f(i)$ representing the malware effect on the API calls.
- Artifact = $\{a_1...a_n\}$ with $f(a)$ representing the malware effect on the artifacts.

The set of all analysis element type counts as listed above for each sample, is:

$$E = \{c_n, r_n, e_n, m_n, p_n, u_n, i_n, a_n\} \quad (1)$$

Z is the set of analysis element counts by type of all samples, and n is the number of malware samples.

$$Z = \{E_1...E_n\} \quad (2)$$

The set of all existing tools is:

$$Y = \{A, C, H, I, L, S, T, V, W\} \quad (3)$$

- A = ANLYZ
- C = CWSandBox

- H = Hybrid Analysis
- I = IDA
- L = Malwr
- S = Sandbox Pikker
- T = Threat Expert
- V = ViCheck
- W = WinDbg

MACT is represented by the variable M representing the set of all malware elements identified by MACT.

$$M = \{c_0, r_0, e_0, m_0, p_0, u_0, i_0, a_0\} \quad (4)$$

Ideally there will be an overall increase across the many elements of Z but the minimal level of success for MACT will be determined when any element count for MACT is greater than that of any existing tool or MACT identifies an element not found by the other tool. This will be represented by

$$\{M_x | x \in E_{1..n}, (\#M_x > \#E_x) \mid (M_x \setminus E_x > \emptyset)\} \quad (5)$$

When this equation is satisfied then the tool can be considered as contributing something unique to the malware analysis activity.

XI. CONCLUSION

Research shows that some currently used malware analysis tools can be difficult to configure, use, have limited code traversal, and due to a lack of interactivity provide limited ability to direct execution of malware during analysis. Other malware analysis tools in use are designed to focus on and perform debugging or reverse engineering of programs. The lack of malware analysis focus by these tools limits their ability to efficiently capture useful malware analysis data.

The use of APIs by malware provides an opportunity to enable interactive dynamic analysis, change the flow of processing, automated logging of system events, and document process activity. This functionality will assist in the analysis of malware by reducing the time and financial cost of the analysis. It can also enable the detection of artifacts not found using other automated methods or techniques. It is unlikely to be able to capture all artifacts for any given sample of malware but the capture of a relevant subset of captured artifacts will save hours or possibly days of analysis.

MACT will be able to adapt to changing malware behavior through its ability to inject new or modified analyst defined API interception functions. An archive of predefined interception functions could be established, added to and maintained by malware analysts. As malware changes and the approaches of its authors evolve so too can the tool adapt through these intercept functions.

REFERENCES

- [1] D. Uppal, V. Mehra, and V. Verma, "Basic survey on malware analysis, tools and techniques," *International Journal on Computational Sciences & Applications (IJCSA)*, vol. 4, no. 1, p. 103, 2014.
- [2] M. Wagner *et al.*, "A survey of visualization systems for malware analysis," in *EG Conference on Visualization (EuroVis)-STARs*, 2015, pp. 105-125.

- [3] R. Lawler. (2017, 10/01/2017). *Equifax security breach leaks personal info of 143 million US consumers*. Available: <https://www.engadget.com/2017/09/07/equifax-hack-143-million/>
- [4] D. J. Lynch. (2017, 10/05/2017). *Hackers tapped personal information in SEC breach*. Available: <https://www.ft.com/content/d767d516-a78a-11e7-ab55-27219df83c97?mhq5j=e6>
- [5] N. Y. Times. (2017). *U.S. Tells 21 States That Hackers Targeted Their Voting Systems*. Available: <https://www.nytimes.com/2017/09/22/us/politics/us-tells-21-states-that-hackers-targeted-their-voting-systems.html>
- [6] A. Greenberg. (2017, 10/01/2017). *Your Guide to Russia's Infrastructure Hacking Teams*. Available: <https://www.wired.com/story/russian-hacking-teams-infrastructure/>
- [7] M. Amin, "A Survey of Financial Losses Due to Malware," presented at the Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies, Udaipur, India, 2016.
- [8] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32-39, 2007.
- [9] M. Christodorescu and S. Jha, "Testing malware detectors," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, pp. 34-44, 2004.
- [10] C. K. Patanaik, F. A. Barbhuiya, and S. Nandi, "Obfuscated malware detection using API call dependency," presented at the Proceedings of the First International Conference on Security of Internet of Things, Kollam, India, 2012.
- [11] J. Lee, K. Jeong, and H. Lee, "Detecting metamorphic malwares using code graphs," presented at the Proceedings of the 2010 ACM Symposium on Applied Computing, Sierre, Switzerland, 2010.
- [12] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 1-32, 2013.
- [13] S. Z. M. Shaid and M. A. Maarof, "In memory detection of Windows API call hooking technique," in *2015 International Conference on Computer, Communications, and Control Technology (I4CT)*, 2015, pp. 294-298.
- [14] Y. Ye, D. Wang, T. Li, and D. Ye, "IMDS: intelligent malware detection system," presented at the Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, San Jose, California, USA, 2007.
- [15] V. P. Nair, H. Jain, Y. K. Golecha, M. S. Gaur, and V. Laxmi, "MEDUSA: MEtamorphic malware dynamic analysis usingsignature from API," presented at the Proceedings of the 3rd international conference on Security of information and networks, Taganrog, Rostov-on-Don, Russian Federation, 2010.
- [16] J. A. Marpaung, M. Sain, and H.-J. Lee, "Survey on malware evasion techniques: State of the art and challenges," in *Advanced Communication Technology (ICACT), 2012 14th International Conference on*, 2012, pp. 744-749: IEEE.
- [17] A. Singh and Z. Bu, "Hot knives through butter: Evading file-based sandboxes," *Threat Research Blog*, 2013.
- [18] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," presented at the Proceedings of the 15th ACM conference on Computer and communications security, Alexandria, Virginia, USA, 2008.
- [19] M. Mehra and D. Pandey, "Event triggered malware: A new challenge to sandboxing," in *2015 Annual IEEE India Conference (INDICON)*, 2015, pp. 1-6.
- [20] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. 2014.
- [21] A. Kumar and S. Goyal, "Advance Dynamic Malware Analysis Using Api Hooking."
- [22] B. Mariani, "Inline hooking in windows," by *High-Tech Bridge SA dated Sep*, vol. 6, p. 26, 2011.
- [23] Y. Han, Z. Hao, L. Cui, C. Wang, and Y. Sang, "A Hybrid Monitoring Mechanism in Virtualized Environments," in *2016 IEEE Trustcom/BigDataSE/ISPA*, 2016, pp. 1038-1045.
- [24] R. Mosli, R. Li, B. Yuan, and Y. Pan, "Automated malware detection using artifacts in forensic memory images," in *2016 IEEE Symposium on Technologies for Homeland Security (HST)*, 2016, pp. 1-6.
- [25] M. Griffin, *Assessment of run-time malware detection through critical function hooking and process introspection against real-world attacks*. The University of Texas at San Antonio, 2013.