An Improved Process Supervision and Control Method for Malware Detection

Behnam Shamshirsaz
Department of Electrical and Computer Engineering, Kharazmi University, Iran
shamshirsaz@khu.ac.ir

Seyyed Amir Asghari
Department of Electrical and Computer Engineering, Kharazmi University, Iran
asghari@khu.ac.ir

Mohammadreza Binesh Marvasti
Department of Electrical and Computer Engineering, Kharazmi University, Iran
marvasti@khu.ac.ir

Abstract: *Most modern-day malware detection methods and algorithms are based on prior knowledge of malware specifications. Discovering new malwares by solely relying on computer based automatic solutions with no human intervention currently appears out of reach. Many malwares never decode harmful parts of their code until the triggering of a specific event. Others detect virtual machine or sandbox environments and hide their true nature. Detecting these kinds of malwares-specifically multi evented ones-are nearly impossible for fully automatic detection methods. Previous research found that about 75% of malwares studied did not react in a fully automatic environment without user intervention thus being undetectable. This paper introduces a near automated solution to detect malwares quickly by relying on a supervision and control method based on user level capabilities of the operating system. Improving on previous methods, this research can replace the need for debugging new malwares in almost all aspects. This solution forces malwares in automated environments to activate and be discoverable. Researcher intervention during malware code execution along with the malware's intent over calling sensitive operating system functions and parameters aid this process. Since operating system functions are virtualized malwares are incapable of physically harming the system during execution. The solution reached 98% overall accuracy in conjunction with reducing code size by 80% in comparison with similar techniques, improving simplicity and reliability.*

Keywords: *Mid-level code, malware detection, process supervision, microsoft windows, anti-virus, endpoint detection and response.*

Received November 12, 2019; accepted February 9, 2021
https://doi.org/10.34028/iajit/19/4/9

1. Introduction

Multiple detection methods have been proposed since the early days of computer malware. They broadly fall under either of two categories: static and dynamic methods [17]. Many solutions and algorithms have been proposed for each category. Static methods are based on detecting specifications of previously known malwares inside new code known as signatures. This approach is not fruitful on new malwares with no known previous signatures. Dynamic methods check for malware behavior while system applications are running via monitoring operating system function calls and reaching a conclusion on whether an application is behaving like malware or not. This type of decision making is key to new malware discovery. However, doing this task effectively within a reasonable timeframe is not trivial considering the rapidly growing number of newly generated malwares and their variety. Discovering new malwares quickly and with certainty are key requirements in a reliable anti-malware solution thus making investigation of different approaches necessary. Previous methods involving artificial intelligence, clustering and data mining alongside using virtual machines and sandboxes did not provide certainty in malware discovery either. This hints semi-automatic solutions as being a desirable approach. Anti-virus researchers commonly use debuggers to inspect suspiciously complex applications. However, working with debuggers is complex and time consuming. With tools developed based on this research an anti-malware developer can check suspicious applications much more easily than using a general debugger. Our method is based on virtual environments surrounding user level Operating System (OS) functions and does not use kernel drivers or kernel level coding. It neither relies on Artificial Intelligence (AI) methods nor mathematical formulations but rather by placing the researcher in control of the malware identification process. It should be noted however, that AI techniques can be combined with our technique. Microsoft Windows was chosen as our testing platform and code samples were developed in the C programming language with visual studio 6.0 for simplicity and robustness. The rest of the paper is organized as follows: Related work is presented in section 2. Section 3 describes the proposed approach. Results and analytical comparisons are presented in section 4. conclusions follow in section 5.

2. Related Works

Previous techniques have been based on either static or

dynamic approaches including methods such as Application Programming Interface (API) hooking [4, 5, 6, 7, 8, 9, 11, 17, 18], patching the OS service dispatcher [4, 16], discovering API calling sequences [2, 14, 15, 16, 17] O.S. Kernel System Service Dispatcher Table (SSDT) patching [3, 4, 5, 6, 14] and Intel Pin [10]. Some unorthodox methods such as malware detection via image code have also been presented in the past [12]. Such methods have not been tested in production environments and may suffer from a high error rate. In this paper we discuss methods relevant to our technique and will utilize foundational elements related to our approach.

Examination of various OS development efforts over time has identified four general capabilities critical for a process research environment. Some of these elements have been previously used by other solutions in a partial manner [2, 16, 17]. These elements are numerated below:

1) Process Modification: previous solutions allowed applications to run unmodified in sandbox or Virtual Machine (VM) environments except for perhaps logging capabilities. This was deemed sufficient for their purposes.
2) Process Injection: in sandbox or VM environments, applications injecting code into other processes are not stopped or modified at runtime.
3) Process Supervision: previous methods did not supervise application code at runtime in a sandbox or VM environment. This was to replicate a setting as close to a physical machine as possible. Often the only supervision available was the VM or sandbox supervisor.
4) Process Control: redirecting the flow of an application while it is running in a VM or sandbox environment is not possible. Consequently, encryption or decryption of code, hidden messages and other important information are not discoverable leaving only the final results for observation.

Most previous methods paid no attention to these elements and allowed VM environments to proceed as usual. The few that did, used kernel level code and drivers such as [16]. In general, previous methods can be divided into two major categories.

1. *VM and sandbox-based methods without specific supervision and control solutions*: these methods focus on AI solutions and mathematical formulations and usually do not provide a specific implementation. These methods are completely dependent on VM and Sandbox logging capabilities and environments. This limits their usability.
2. *Solutions with specific supervision and control mechanisms*: these solutions present an implementation method to supervise code flow inside operating systems. Until now almost all methods presented focused on mixed user level and kernel level coding, using kernel level code for full OS control and user level code for Import Address Table (IAT) filtering. In some cases, such as [16] user level code was used alone but such cases lacked critical control over process creation and chaining.

Usually, category 1 solutions focus on the ability to detect new malwares but those solutions are not focused on usability and effectiveness since they ignore sandbox usage limitations. This combined with their high AI error rate makes them undesirable. Also, a high number of malwares detect sandbox environments or require user intervention and are not detectable in automatic environments. Category 2 solutions are usually designed for end user systems and are not suitable for anti-malware solutions such as [16]. Unorthodox solutions such as image analysis or usage of disassemblers are not in our scope. These methods have limited effectiveness in certain cases and do not present a general solution.

3. Proposed Method

The method proposed will be described as follows and shall be called User Process Chaining (UPC) henceforth.

3.1. Overview

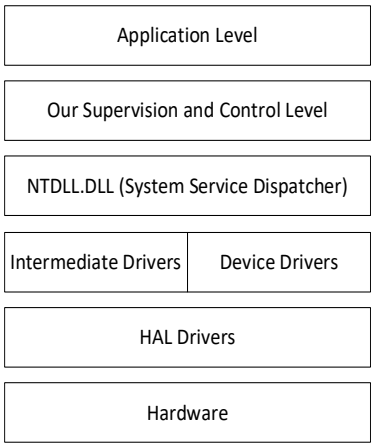
Today, malware detection mechanisms are almost entirely mixed mode applications. They have kernel mode device drivers along with their user mode applications since system control is often perceived as not possible without a kernel component. However, this may not be necessary. In modern operating systems (especially Windows) application processes are more isolated than ever before and can be seen as small VMs. Calling lower-level OS services is not possible directly or is strongly prohibited. User level code should call OS services via standard APIs. Given this, control over standard APIs allows us to make a controlled VM by only accessing user level environment specifications with no kernel level programming. At this point, kernel level programming is only needed when an application wants to install a new device driver. This however, for a normal application should be recognized as a highly suspicious attempt at system breach and should be reported as such. Supervisory kernel level code is only necessary when suspicious kernel level code is already running on the system and is in need of examination or a user mode application has exited supervision and should be stopped or terminated. UPC highlights that a complete user mode supervisor is enough to control any user mode application. Making such a supervisor is not difficult and could be done without kernel level coding or system level violations such as utilization of

undocumented functions or structures. In previous solutions we can see extensive use of New Technology (NT) Kernel patches, device drivers, disassemblers and sandboxes used. Despite using these tools previous techniques did not create sufficiently successful solutions. This could be seen at times due to similarities between harmful and non-harmful code.

A desirable anti-malware solution should be dynamic, smart and able to provide a level of certainty while allowing a researcher to determine the final verdict. Previous automated methods could not exceed 25% accuracy, despite only a 98% success rate within that margin using fully automated detection methods based on clustering and heavy data mining algorithms [19]. This paper aims to tackle all cases including the remaining 75%.

3.2. System Design

As previously stated, our design utilizes user level code only and sample applications are implemented on Microsoft Windows. Replicating this structure is feasible in a similar manner on other operation systems such as GNU/Linux. Figure 1 provides an overview of the OS layers in Windows.



Figuer 1. System position in microsoft windows OS layers.

3.3. Primitives

1. *No use for undocumented API or structures*: previous software tracers or VMs used undocumented or discouraged Windows APIs for controlling program execution. This behavior would cause serious problems with new Windows versions or patches.
2. *New Design for compatibility with Windows*: previous works mostly designed for Linux environments, had process control flow and chaining that applied poorly under Windows. Process memory mapping and control flow is now more enhanced and requires special care. Our paper aims to provide a comprehensive solution to this issue.

3.4. Process Handling in Windows

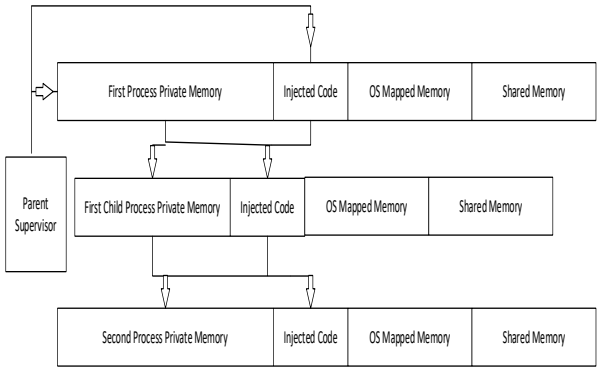
- Process Creation and Control Functions: UPC modifies application processes in a manner that sensitive OS functions fall under supervision and control. Windows API functions such as CreateProcess(), OpenProcess(), TerminateProcess(), ExitProcess(), CreateThread(), CreateRemoteThread() and ExitThread() provide a complete set that can be used to monitor and redirect process control flow under Windows. A supervision diagram is utilized that guides control over these without ignoring their interdependency.
- Process Memory Supervision: Functions such as WriteProcessMemory(), ReadProcessMemory() and VirtualProtectEx() can be used for all process handling requirements at application level. A complete overview of how Windows handles user level memory pages could be helpful in resolving memory access problems. UPC provides a local supervisor for processes under investigation.
- Process Code Injection: In Windows a parent user level supervisor injects extra code into a child process to control its API calls or memory mapping. Code injections however, may cause crashes or conflicts if done incorrectly. UPC aims to do this silently with a minimal footprint and controls injections requested by the monitored process.
- Process Runtime Redirection: UPC provides the ability to control API calls and their effects at runtime. We can allow the application to run a partial or complete subset of the Windows API with modified or unmodified parameters and halt some executions in need of review. Given these tools the supervisor can analyse rare or special events, maintain control over memory allocations/de-allocations, reveal secret codes/messages and much more. To summarize, UPC provides a virtual environment capable of redirecting application flow.

3.5. Design Properties

- Windows User Mode VM: given that UPC is limited to OS user mode the analysis is limited to WinAPI, its structures and CPU Protected Mode Level 3 capabilities. Undocumented structures will change in the future and are not suited for use. We use standard hook methods and link chaining of process creation and memory handling. Avoiding undocumented structures prevents incompatibility with future system developments.
- Control Flow Chaining: our CreateProcess() based control flow chaining mechanism and other compatible functions will guarantee that UPC will not lose control flow of application processes in any situation. Not using this approach, previous methods were forced to use kernel patches.

3.6. Process Chaining View

Figure 2 demonstrates how process chaining is handled by our system.



Figuer 2. Process chaining control diagram.

As demonstrated UPC supervisor application injects control code not only inside the process under investigation but will also replicate such code inside newly spawned processes as well. This ensures uninterrupted supervision until process termination.

3.7. Process VM View

Any process created by Windows's application loader has its own memory and a copy of system libraries. As such it resembles an independent VM. At execution start the function addresses needed by the process are known via its import table. Address of system functions not defined in the import table are unknown to the process image and are requested by GetProcAddress(). Guessing system addresses or calculating them by using undocumented structures is prohibited, not to mention in violation of system programming rules under Windows. As expected newer Microsoft Windows versions and patches eliminate all previous methods of non-standard calling of OS routines. Utilizing OS and application DLLs are portrayed in Figure 3.

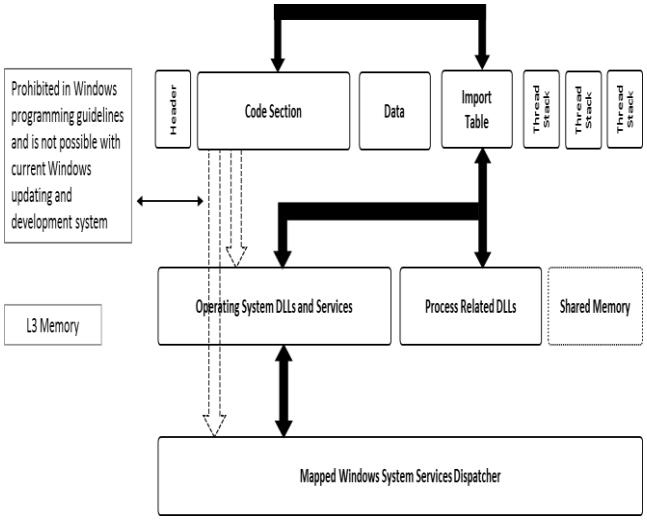


Figure 3. Process VM calls and addressing.

With this in mind, we can be sure that sensitive system calls will pass through our filters with a correct patch.

3.8. Process Supervision View

UPC process supervision routines will be inside the process memory undiscoverable by it via being accessed from inside an injected dll. At this point process call handling changes from Figures 3 to 4.

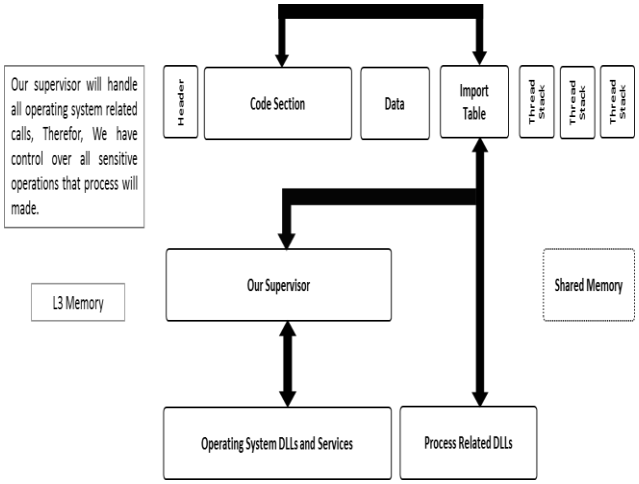


Figure 4. O.S. Filtering diagram with UPC supervision.

3.9. Sensitive APIs

1. Searching, Creating and Erasing file functions: in Microsoft Windows a process can create new files or modifying existing ones with the CreateFile() function. This function has two variants: the A and W extensions for American National Standards Institute (ANSI) and Wide character file names. Many other functions in Windows have the same variants. Since version 7.1, Windows places some base libraries under main ones such as kernelbase.dll under kernel32.dll. Some functions of main libraries have equivalents in their base versions and should be considered for filtering as needed. Functions such as CreateFile2(), ReadFile(), WriteFile(), FindFirstFile(), FindNextFile() and DeleteFile() should also be considered for filtering.
2. Creating new process functions: createProcess(), CreateProcessInternal(), ShellExecute(), CreateService() and StartService() should be considered for filtering. Any process that calls CreateService() is creating a new driver and if it is not a hardware installation application or a known filter driver it should be considered as malware.
3. Creating and controlling new threads functions: functions such as CreateThread(), ResumeThread() and CreateRemoteThread() should all be considered for controlling and filtering.
4. Process Memory Access and Modify functions: functions such as ReadProcessMemory(), WriteProcessMemory(), VirtualProtectEx(), VirtualAllocEx() and VirtualFreeEx() should all be filtered.

5. Device drivers access and control function: deviceIoControl() which is currently the main function to communicate between application level code and device driver or kernel level code should be filtered.
6. Process communication functions: libraries such as ws_32.dll and wsock32.dll that provide socket communication should be monitored. Control over LoadLibrary() and LoadLibraryEx() functions provides control over these system DLLs and their functions. Given that some processes use CreateFileMapping() for intercommunication, this function too should be supervised. This list however, is not exhaustive.
7. System time functions: getTickCount(), GetTickCount64(), GetSystemTime(), GetSystemTimeAsFileTime() and GetSystemTimes() should be controlled by the supervisor to prevent recognition of time elapses by the investigated application.
8. Process supervision and control / ACL functions: createToolhelp32Snapshot(), Process32First() and Process32Next() are in need of supervision. Windows ACL functions such as OpenProcessToken(), OpenThreadToken() and AdjustTokenPrivilages() must also be supervised. Please note that this is not an exhaustive list and there are unmentioned functions in ntdll.dll in need of supervision.
9. Registery access functions: functions such as RegOpenKey(), RegOpenKeyEx(), RegSetKeyValue() and RegQueryValue() should be all be controlled.

3.10. System Operation

UPC commences by loading the target application via a supervisor loader. Target is loaded by the CreateProcess() function with the CREATE_SUSPENDED flag set and conduct supervisor routine injection before ResumeThread() is called for the main process. This ensures that the main thread of the executable image won't be called before UPC filtering system and UPC supervisor routine will have full control. In rare cases where other libraries may contain the malware code supervisor should test those libraries with other methods such as debugging which is outside of the scope of this paper. Flow of the suspicious application is halted by any supervisor routine calls to sensitive OS APIs and a dialog box is displayed showing the function name and its parameters. The researcher can then allow function execution unimpeded, change function call parameters before the execution or prevent/allow function execution and report fake values to the caller.

Using this system, the supervisor can handle almost all scenarios at runtime for the application under investigation and find hidden actions under special events. In practice we have found that using the supervisor application is very simple and fast. Execution of the supervisor application can be stopped or restarted at any point. Memory view of the process under investigation is possible at runtime and the supervisor can inspect this view any time there is a sensitive OS API call.

4. Comparison Results

4.1. Implementation Comparison with Mixed Mode Solutions

A complete mixed mode solution like [16] contains different parts. These code sections have different complexity and execution times. In this case, code size and code execution time have a close relation. Considering the following variables:

SET=Service Execution Time, AI=Artificial Intelligence, CM=Control Module

For which the following exist:

IAT=Import Address Table, IRP=I/O Request Packet, IDT=Interrupt Descriptor Table, HYP=Hypervisor, EMU=CPU Emulation and SSDT=System Service Descriptor Table patched handler

We will have service execution times for each part calculated roughly via:

$SSDT\_SET \approx IAT\_SET * 2$	$IRP\_SET \approx SSDT\_SET * 7$
$IDT\_SET \approx SSDT\_SET * 2$	$HYP\_SET \approx SSDT\_SET * 7$
$EMU\_SET \approx SSDT\_SET * 7$	

In [15] execution time of all code parts is:

IAT_SET + *SSDT_SET* + *IRP_SET* + *IDT_SET* + *HYP_SET* + *EMU_SET* + *AI_SET* + *CM_SET* and our execution time is: IAT_SET+CM_SETConsider that our IAT_SET and CM_SET is very close in size and execution time to [16]. Removing AI_SET from computation, [16] code execution time demonstrates that IAT_SET+CM_SET are only 4% of overall code execution time and other parts occupy the remaining 96%. These parts also occupy 80% the code volume. With such smaller code size and complexity our method demonstrates the simplicity required for rapid development and reliability.

The AI method used in [16] is based on the weighted API mechanism presented in Equation (1) and its accuracy is measured with Equation (2) provided below:

M = A * Ax + B * Bx + C * Cx … , A,B,C,… are weighted APIs and Ax, Bx, Cx are their frequencies. (1)

Accuracy = Approximate Matching Index of (M) with (Malware Classification Database) (2)

UPC accuracy is however, based on computing and relying on multiple approximate matching indices by a human supervisor. This results in much higher accuracy compared to using a single M Approximate Matching Index such as in [16].

4.2. Comparison with Fully Automated Sandbox-Based Methods

Previous methods of malware detection, including Weighting Mechanisms [1, 2], Properties' Extraction [2, 15] and Classification [10, 11, 12, 13] concentrated on reaching maximum precision. In their research, samples were selected that could be run without user intervention in sandbox environments such as CW or Cuckoo. The important point missed was that many malwares are dormant without user intervention and some recently developed recognize sandbox and VM environments and do not activate. A recent quantitative Zhang *et al*. [19] inspected 60000 samples from which only 17400 samples activated in a sandbox environment. From the ones that activated some did not successfully operate in their sandbox environments. The article concluded that 75% of the sample set could not be tested on a fully automatic detection system. The automated detection system presented reached 98% precision on the remaining 25% of the sample set.

Given these facts, the effective accuracy of the [19]'s technique is placed at 24.5%. This is not remotely acceptable by industry standards. Anti-malware software should be able to inspect and discover harmful code inside samples with a high success rate with a low rate of false positives. The overall accuracy of the system should not fall below 95%. Considering these facts, the industry has largely avoided fully automated detection systems and relies heavily on debugging. Since debugging is time consuming, this approach has not been able to keep pace with the rapid rate with which new malwares are appearing. The technique that we have presented here activated nearly all of the 75% of samples that [19]'s technique discounted. Since a researcher has control over the filtering process in UPC the only parameter that may affect results is to a small degree human error. This however, should be minimal given that the malware researcher should be a qualified domain expert. We have determined the worst-case error rate as being 2%. This leaves us with a 98% accuracy not to mention the time savings realized by the near-automated nature of the technique. Fully automated techniques may improve their results by complementing their technique with UPC. However, this may incur additional development costs avoided by simply using UPC technique alone. UPC can also be used in Endpoint Detection and Response (EDR) systems developed by anti-virus companies such as Kaspersky. Their solution is described in the detail in [18]. A combination of current Endpoint Protection Platform (EPP) and EDR systems developed by UPC mechanism will raise efficiency on detecting new malwares/Advanced Persistent Threat (APT) as shown in Figure 5. Simulation was done by a 2.00 GHZ Pentium 4 (8 Core), 6 GB RAM and 640 GB HDD.

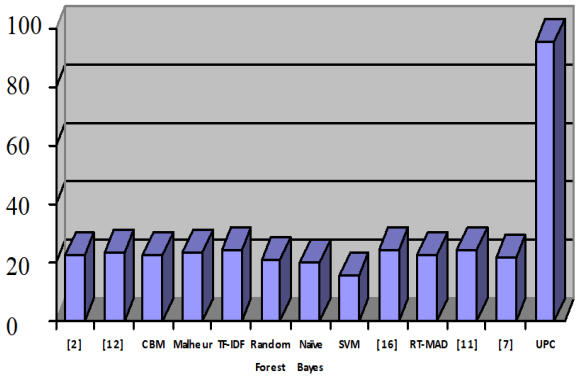


Figure 5. EPP plus EDR efficiency.

5. Conclusions and Future Works

Developing methods enhancing malware detection is a priority in cyber security. What we have demonstrated here was an enhancement on supervision and control methods presented and discussed in previous works. This method could be used as a standalone malware reconnaissance solution or developed as a module inside existing EDR tools. The method presented was designed for and capable of achieving the following goals: saving time, achieving simplicity in design, forward compatibility with future OS versions, drastically improving accuracy and greatly reducing dependence on debugging as a malware inspection tool. To this end UPC method reached 98% overall accuracy in comparison with the 24.5% reached via fully automated methods such as [19]'s. This was done in conjunction with reducing code size by 80% in comparison with [16] improving simplicity and reliability. Future avenues of exploration may pursue code execution in kernel mode for differentiating legitimate driver installations from malware system breach attempts or involve crash resistance and recovery methods described in [1, 19].

References

[1] Asghari S. and Taheri H., "An Effective Soft Error Detection Mechanism Using Redundant Instructions," *The International Arab Journal of Information Technology*, vol. 12, no. 1, pp. 69-76, 2015.

[2] Chen F. and Fu Y., "Dynamic Detection of Unknown Malicious Executables Based on API Interception," *in Proceedings of 1st International Workshop on Database Technology and Applications*, Wuhan, pp. 329-332, 2009.

[3] Cheng J., Tsai T., and Yang C., "An Information Retrieval Approach for Malware Classification Based on Windows API Calls," *in Proceedings of the International Conference on Machine Learning and Cybernetics*, Tianjin, pp. 1678-1683, 2013.

[4] Fu W., Pang J., Zhao R., Zhang Y., and Wei B., "Static Detection of API-calling Behavior from

Malicious Binary Executable," *in Proceedings of International Conference on Computer and Electrical Engineering*, Phuket, 2008.

[5] Javaheri D., Hosseinzadeh M., and Rahmani A., "Detection and Elimination of Spyware and Ransomware by Intercepting Kernel-Level System Routines," *IEEE Access*, vol. 6, pp. 78321-78332, 2018.

[6] Liu Y., Lai Y., Wang Z., and Yan H., "A New Learning Approach to Malware Classification Using Discriminative Feature Extraction," *IEEE Access*, vol. 7, pp. 13015-13023, 2019.

[7] Musavi A. and Kharrazi M., "Back to Static Analysis for Kernel-Level Rootkit Detection," *IEEE Transactions on Information Forensics And Security*, vol. 9, no. 9, pp. 1465-1476, 2014.

[8] Muthumanickam K. and Ilavarasan E., "Behavior based Authentication Mechanism to Prevent Malicious Code Attacks in Windows," *International Conference on Innovations in Information, Embedded and Communication Systems*, Coimbatore, pp. 1-5, 2015.

[9] Pektaş A. and Acarman T., "Malware Classification Based on API Calls and Behavior Analysis," *IET Information Security*, vol. 12, no. 2, pp. 107-117, 2018.

[10] Qiao Y., He J., Yang Y., and Ji L., "Analyzing Malware by Abstracting the Frequent Item sets in API call Sequences," *in Proceedings of 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Melbourne, pp. 265-270, 2013.

[11] Qu-Nguyen L., Demir T., Rowe J., Hsu F., and Levitt K., "A Framework for Diversifying Windows Native APIs to Tolerate Code Injection Attacks," *in Proceedings of the 2nd ACM Symposium on Information, computer and Communications Security*, New York, pp. 392-394, 2007.

[12] Skaletsky A., Devor T., Chachmon N., Cohn R., Hazelwood K., Vladimirov V., and Bach M., "Dynamic Program Analysis of Microsoft Windows Applications," *in Proceedings of International Symposium on Performance Analysis of Systems and Software*, White Plains, pp. 2-12, 2010.

[13] Shevchenko Y., "EPP Plus EDR: The Future of Endpoint Cybersecurity" *Kaspersky Corporation EPP-EDR Importance.* https://www.kaspersky.com/blog/epp-edr-importance/22366/, Last Visited, 2019.

[14] Sun H., Wang H., Wang K., and Chen C., "A Native APIs Protection Mechanism in the Kernel Mode Against Malicious Code," *IEEE Transactions on Computers*, vol. 60, no. 6, pp. 813-823, 2011.

[15] Sun S., Fu X., Ruan H., Du X., Luo B., and Guizani M., "Real-Time Behavior Analysis and Identification for Android Application," *IEEE Access*, vol. 6, pp. 38041-38051, 2018.

[16] Tsaur W. and Chen Y., "Exploring Rootkit Detectors' Vulnerabilities Using a New Windows Hidden Driver Based Rootkit," *in Proceedings of 2nd International Conference on Social Computing*, Minneapolis, pp. 842-848, 2010.

[17] Volynkin A., Skormin V., Summerville D., and Moronski J., "Evaluation of Run-Time Detection of Self-Replication in Binary Executable Malware," *in Proceedings of the IEEE Workshop on Information Assurance*, West Point, pp. 184-191, 2006.

[18] Xu S., Ma X., Liu Y., and Sheng Q., "Malicious Application Dynamic Detection in Real-Time API Analysis," *in Proceedings of IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, Chengdu, pp. 788-794, 2016.

[19] Zhang F., Leach K., Stavrou A., and Wang H., "Towards Transparent Debugging," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 2, pp. 321-335, 2016.

Behnam Shamshirsaz received his B.Sc. degree in 2017 (IT engineering) from Azad University and M.Sc. (Computer Architecture) in 2019, Kharazmi University, Tehran. His research interests include Computer Architecture and Software Security system design.

Seyyed Amir Asghari received his B.Sc. degree in 2007 (hardware engineering major), M.Sc. and Ph.D. in 2009 and 2013 respectively (computer architecture major) from Amirkabir University of Technology. His current research interests include fault-tolerant design and real-time embedded system design. He has served as a faculty member in the Department of Electrical and Computer Engineering at Kharazmi University.

Mohammadreza Binesh Marvasti received the M.Sc. degree from Department of ECE University of Tehran, Iran, in 2007 and the Ph.D. degree in ECE from McMaster University, Canada, in 2013. His research interests include Computer Architecture, Low-Power Digital Design, FPGAs, Approximate Computing, and On-chip Interconnection Network. He has served as a faculty member in the Department of Electrical and Computer Engineering at Kharazmi University.