International Journal of Security and Its Applications
Vol.8, No.1 (2014), pp.183-192
http://dx.doi.org/10.14257/ijsia.2014.8.1.17

Evading Anti-debugging Techniques with Binary Substitution

JaeKeun Lee, BooJoong Kang and Eul Gyu Im

Department of Computer and Software,
Hanyang University, Seoul, Korea
{jk890111, deviri, imeg}@hanyang.ac.kr

Abstract

Anti-debugging technology refers to various ways of preventing binary files from being analyzed in debuggers or other virtual machine environments. If binary files conceal or modify themself using anti-debugging techniques, analyzing these binary files becomes harder. There are some anti-anti-debugging techniques proposed so far, but malware developers make dynamic analysis difficult using various ways, such as execution time delay, debugger detection techniques and so on. In this paper, we propose a Evading Anti-debugging techniques method that can avoid anti-debugging techniques in binary files, and showed several samples of anti-debugging applications and how to detect and patch anti-debugging techniques in common utilities or malicious code effectively.

Keywords: *Malware Analysis, Anti-debugging detection, Static Analysis, Dynamic Analysis*

1. Introduction

Software has been vulnerable to copyright infringements due to illegal copies and distributions. Thus it is important for software developers to conceal the program's core sources code or flow when they build software binaries. So, many obfuscation techniques or anti-debugging techniques were applied to binary files [1]. Anti-debugging techniques refer to various ways of preventing binary files from being analyzed in debuggers or other virtual machine environments.

As many debuggers or analyzing tools being developed rapidly, some developers try to avoid debugging through anti-debugging APIs or other techniques [2]. To avoid anti-debugging techniques, analysts trick binary files as if they are not in the analyzing environments using plug-ins for debuggers. Using plug-ins for debuggers reduces the debugger's performance and the tracing speed. In addition, some anti-debugging techniques can still detect the debugging environments, so execution results may be different in the debugging environments.

In this paper, we propose a method to avoid anti-debugging techniques by analyzing assembly instructions. Our proposed method analyzes and traces the general-purpose register values to find out whether anti-debugging instructions exist, using anti-anti-debugging rule sets. Our rule-based method, each time a new technology appears, can add or remove anti-anti debugging rules quickly. In addition, because this method do not execute program in debugger environments, it does not be detected by dynamic anti-debugging techniques. After spotting the sections containing anti-debugging instructions, our Evading Anti-debugging techniques tool patches the instructions with new instructions. Experimental results showed that our method can remove anti-debugging instructions from malware.

ISSN: 1738-9976 IJSIA
Copyright ⓒ 2014 SERSC

The rest of paper is composed of following: In Section 2, background information of basic anti-debugging technologies was introduced. Section 3 addresses related work on handling anti-debugging technologies. Section 4 describes about our Evading Anti-debugging techniques method, and Section 5 includes various samples and malicious codes involving anti-debugging technologies. Section 6 concludes the paper.

2. Background

2.1. Assembly Instruction

Assembly language is a programming language directly corresponding to machine code, and for Windows PE (portable executable) files. There exist several disassemblers or debuggers, such as `borg` disassembler [3], `ollydbg` [4], `ImmunityDebugger` [5], and `IDA Pro` [6]. Malware can be analyzed with assembly instructions generated from these disassemblers. Additionally, some disassemblers provide address or register information for some instructions, so analysts can apprehend detailed execution process with this information. As shown in Figure 1, Anti-debugging technologies can also detect which Anti-debugging API is called from instructions.

In addition, assembly instructions include General Purpose registers [7], such as `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `ESP`, and `EBP`. These registers are used to save address values, and could be used in API calls or flag references. Thus, analysts can understand the program executions if they know the information in registers or the information that a certain instruction references.

2.2. Anti-debugging Techniques

Table 1. Anti-debugging Techniques [8]

Type	Name
API	IsDebuggerPresent
	CheckRemoteDebuggerPresent
	FindWindow
	ZwQueryInformationProcess
	NtQueryInformationProcess (ProcessDebugPort)
Flag	BeingDebugged flags
	Ntglobal flags
	Heap flag
others	RDTSC
	OllyDbg Memory Breakpoint
	SeDebugPrivilege
	INT 3 Exception
	GetTickCount

Anti-debugging technology refers to techniques that prevent analyzing some parts of binary files in debugging environments by executing different execution flows, or exiting the executions. In Table 1 shows various anti-debugging techniques such as API based anti-debugging, hardware based anti-debugging, timing based anti-debugging, *etc*.

 Copyright ⓒ 2014 SERSC

In case of the `IsDebubberPresent` API which is the most common anti-debugging technique, the API returns the `PEB`'s `beingDebugged` flag, and using this flag, the process can distinguish whether it is being debugged or not. Most anti-debugging APIs use certain data that judge what information flag contains. There are also time-based anti-debugging techniques that monitor execution delay to find out whether debugging environments are used or not.

3. Related Work

Most of the existing researches against anti-debugging suggested exploring multi-path or flags in debugger environments. However, these approaches fail to hide debugger environments, even though modifying the analyzing environments similar to PC execution environments. Artem Dinaburg *et al.*, [9] proposed "Ether" implementation hypervisor level of CPU. From controlling the analysis environment outside, the malware may not know the presence of an analyzer. L. Liu *et. al.*, [10] proposed Malyzer, using the shadow process, which monitored other processes running malicious code. Malyzer makes this shadow process invisible to the original suspicious process. As a result, Malyzer defeat anti-debugging techniques. M. N. Gagnon *et al.*, [11] focuses anti-debugging techniques and also suggested ways to protect software. Peter Ferrie [12] explained about various anti-unpacker tricks and described what features it has when the OS environment is differentiated. In addition, he made it easy to infer avoiding debugger environment by checking the flag value. Kawakoyal, Iwamura and Itoh [13] practiced Stealth Debugger, VMM with debugging function, and controlled Time Tick to avoid time checking anti-debugging technology. Xu Chen *et. al.*, [14] also used Stealthy Debuger - used to conceal Virtual Machine, signature or debugger environment – to suggest various ways of avoiding anti-debugging technology. `aadp` [15] is a plugin for `ollydbg` and `ImmunityDebugger` that aims to avoid anti-debugging techniques, such as anti-debugging APIs or flags. J. Lee *et al.*, [16] proposed a basic concept of a rule-based anti-anti-debugging system, but their paper did not have enough experimental data.

Most of the researches focused on concealing debugger usage in dynamic analyze environment or characteristics emerging from virtual machine execution. However, in most of dynamic environment, concealing signature or using other plug-ins takes more time than analyzing the binary itself without environment setting or plug-ins. Moreover concealing the existing analyze environment would make the framework itself useless when it is detected by new anti-debugging technology, and it will cost more expense and time to set new environment.

4. Our Proposed Method

This section suggests rule structure and whole composition of Evading Anti-debugging techniques method, various ways of detecting anti-debugging technology by analyzing assembly code, patching ways of byte sequence matching.

Copyright ⓒ 2014 SERSC

4.1. Evading Anti-anti-debugging Techniques Method Overview

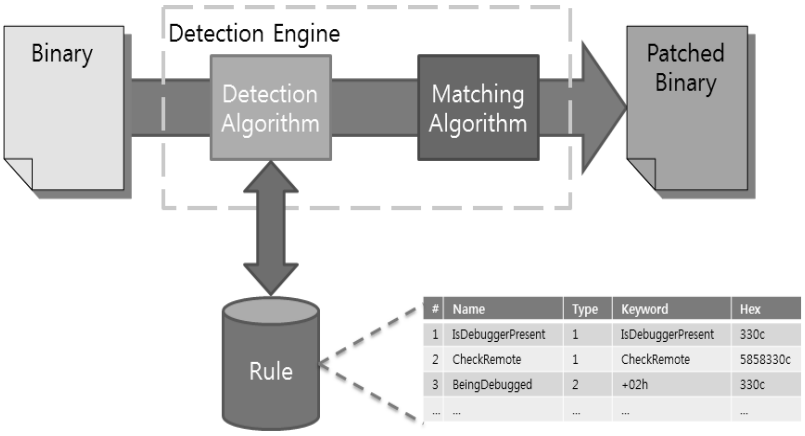

Figure 1. Evading Anti-anti-debugging Techniques Method Overview

Our Evading Anti-debugging method focuses on detecting and patching to avoid anti-debugging techniques based on static analysis. As shown Figure 1, the method can be split into three steps. First, our rule set is parsed, and an input binary file is disassembled. Next, detection signatures in the rules are searched in assembly instructions, and detected instructions are recorded with offset information. Lastly, locations of the recorded instructions from the previous step are identified in binary files, and the matched byte sequences are modified to new byte sequences according to the patching rules defined in our rule set.

4.2. Rule Composition

Table 2. Rule Composition

#	*Name*	*Type*	*Keyword*	*Parametesr*	*Patchhex*	*…*
1	IsDebuggerPresent	1	IsDebuggerPresent	0	33c0	…
2	BeingDebugged	2	+02h	0	33c0	…
3	CheckRemote	1	CheckRemote	2	585833c0	…
4	Ntglobal flags	2	+68h	0	33c0	
…	…	…	…	…	…	…

In Table 2, Rule refers to regulations for searching and patching anti-debugging. A rule is composed of 5 parts – Number, Name, Type, KeyWord, Patch_Hex. Number means input sequence when setting the rule, and Name means title of the following anti-debugging. Next, Type gets different assigned number according to whether the anti-debugging rule is API Type, TEB list referred flag, or other. KeyWord means the string value used when searching. It could hold string such as `IsDebuggerPresent`, or status such as +02h. In status case, sort of the Type is also searched – searching only real flag status. Lastly, Patch_Hex is searched by the rule, and if specific Hex Byte Sequence is evaluated as anti-debugging, the existing Patch_hex is conversed to byte sequence for patching.

 Copyright ⓒ 2014 SERSC

New anti-debugging techniques emerged when, creating new debugger plug-ins or the debugger's another scripting language to create the time-consuming need. While Depending on composition, add a new rule to the speed of our rule-based method is very fast and flexible.

4.3. Anti-debugging Detection Algorithm

Our anti-debugging detection algorithm handles three cases. The first case is to call APIs directly using their addresses. As shown Figure 1.A, it is the case when call `dword ptr [IsDebuggerPresent]` appears in assembly instructions. Like most of anti-debugging functions, the `IsDebuggerPresent` API returns value to the `eax` register after being called. If the return value is 0, it represents the program is not being debugged, while the return value 1 means the program is on debugging. Therefore, this instruction is widely used to detect debugging environments. In Figure 1.B, the '`call dword ptr [IsDebuggerPresent]`' instruction is replaced by '`xor eax, eax`' (33c0) - initializing `eax` to 0.

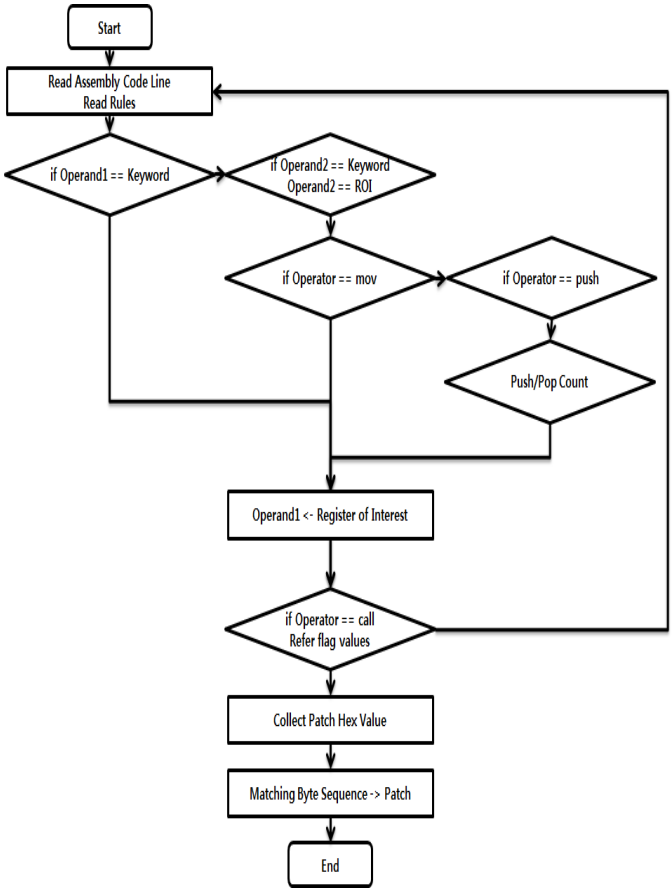

Figure 2. Flow Chart of Anti-debugging Detection Algorithm

The second case handles copying `dword ptr [IsDebuggerPresent]` to a general purpose register before this API is called. While in some cases malware developers copy the above API address to other register to avoid anti-debugging detection. The third case is the

Copyright ⓒ 2014 SERSC

case that flags in PEB (Process Environment Block) are directly examined to detect debugging.

The third case is when distinguishing whether it is anti-debugging by using flag values. Figure 3.D shows process of approaching PEB through fs, and copying beingDebugged flag value to register using +02h. If a register approaches PEB, different flag should be marked like Register of Interest (ROI) concept, and it should be decided whether the Rule could be applied at once, initializing the flag value to 0. A block Diagram and patching methods are described below.

Besides, it shows the process of copying ROI to other registers. The mov command copies ROI and calls it. Patch is necessary for copy call section, since it is doing the same task after all. push, pop and copy operations are executed alike this process.

After this basic detection algorithm, patch using the Hex value specified in the Rule. In the API function case, push operator is used according to the number of parameters, and the Windows API follows stdcall logic – therefore pop(58h) operator is needed accordingly. Thus in the rule, pop should be added according to the number of parameters, to make the pointer indicate proper spot when executing binary.

In addition, in some the operating system, anti-debugging APIs address has been changed by offset. In this case, the nearest conditional branches (JNZ) from anti-debugging APIs were replaced by normal branches (JMP) without modifying the call anti-debugging APIs Instruction.

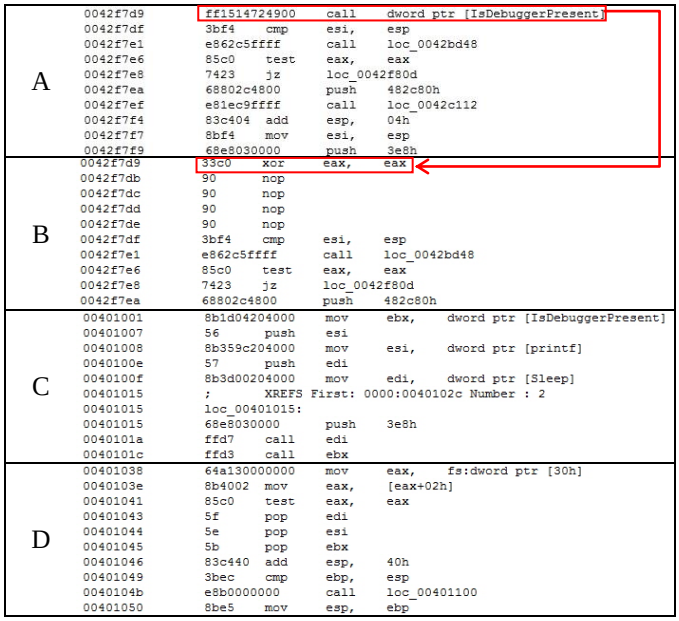

Figure 3. Cases of Anti-debugging

5. Experiments

This section explains experimental result of our anti-debugging method. The experiments were performed in a Windows XP environment, and the modified borg disassembler and ollydbg were used.

 Copyright ⓒ 2014 SERSC

5.1. Anti-debugging Patch Experiment

Table 3. Anti-debugging included Sample

API & Flag Name	*Detection & Patch*
IsDebuggerPresent	O
CheckRemoteDebuggerPresent	O
FindWindow	O
ZwQueryInformationProcess	O
NtQueryInformationProcess	O
BeingDebugged flags	O
Ntglobal flags	O
Heap flag	X

In Table 3, Initially, we create a sample binary file that has various anti-debugging techniques, such as `IsDebuggerPresent`, `CheckRemoteDebuggerPresent`, `FindWindow`, `ZwQuery(NtQuery)InformationProcess`, `BeingDebugged`, the `Ntglobal` flag and the Heap force flag. Then, we tested our Evading Anti-anti-debugging method with this sample and the method detected all the anti-debugging techniques except the Heap force flag. The Heap force flag can be detected but cannot be patched because Patch Hex value was oversized compared to the original Hex value.

Table 4. Anti-debugging included Malicious Code Families

Malicious Code Name	*Detection& Patch*
Trojan.Agent.a.b.c.d	IsDebuggerPresent ZwQueryInformationProcess CheckRemoteDebuggerPresent
Trojan.Antavmu.a.b.c	IsDebuggerPresent
Backdoor.Agent.a.b	IsDebuggerPresent
Worm.Autorun.a.b.c.d	IsDebuggerPresent

Table 5. Anti-debugging included Commercial Software

Program Name	*Detection& Patch*
AcroRd32	IsDebuggerPresent
Alzip	IsDebuggerPresent
Ggpo	IsDebuggerPresent
DaumPotPlayer	IsDebuggerPresent
Winrar	IsDebuggerPresent
Chrome	IsDebuggerPresent

Tables 4 and 5 shows experimental results with malware and commercial software. Most malware was found to have only the `IsDebuggerPresent` API. But the `ZwQuery InformationProcess` API was used to in the Trojan.Agent family. Most malware focuses on infecting rather than anti-debugging, so only a simple anti-debugging technique is used. Anti-debugging techniques were also used in commercial software, and in most cases, only the `IsDebuggerPresent` API was detected.

Copyright ⓒ 2014 SERSC

5.2. Patch Verification Experiment

Table 6. Trace Change of Worm.Autorun.A

Tracing Patching Before			
	00402155	CALL 00401E65	
Target Pos	00401E65	CALL DWORD PTR DS:[<&KERNEL32.IsDebuggerPresent>	
		EAX=00000001	
	00401E6B	TEST EAX,EAX	
	00401E6D	JE SHORT 00401E77	
	00401E6F	PUSH 0 ExitCode = 0	
	00401E71	CALL DWORD PTR DS:[<&KERNEL32.ExitProcess>]	
		EAX=00000000, ECX=7C7D0000, EDX=77C11AE8,	
		EBP=0012FEE4, ESI=7C93DE6E, EDI=00000000	
		Process terminated, exit code 0	
Tracing Patching After			
	00402155	CALL 00401E65	
Target Pos	00401E65	XOR EAX,EAX	EAX=00000000
	00401E67	NOP	
	00401E68	NOP	
	00401E69	NOP	
	00401E6A	NOP	
	00401E6B	TEST EAX,EAX	
	00401E6D	JE SHORT 00401E77	
	00401E77	RETN	
	0040215A	CALL 004021B0	
	004021B0	PUSH EBP	
	004021B1	MOV EBP,ESP	EBP=0012FF00
	004021B3	PUSH ECX	
	004021B4	MOV EAX,DWORD PTR FS:[18]	EAX=7FFDF000
	004021BA	MOV DWORD PTR SS:[EBP-4],EAX	

Table 6 shows Worm.Autorun.A in the environment of the debugger is terminated immediately. After patching the debugger is not detected in `IsDebuggerPresent` API. As a result, Worm.Autorun shows a different behavior. On the other hand, in Table 7, 8 AcrobatReader and Trojan.Agent.a did not shut down right in the debugger environment. But after both programs are patched, Worm.Autorun.A shows different behaviors.

Table 7. Trace Change of AcrobatReader

Tracing Patching Before			
	0043EF60	Main MOV DWORD PTR SS:[EBP-18],ESP	
Target Pos	0043EF63	CALL DWORD PTR DS:[<&KERNEL32.IsDebuggerPresent>	
		EAX=00000001	
	0043EF69	TEST EAX,EAX	
	0043EF6B	JE SHORT 0043EFAD	
	0043EF6D	MOV DWORD PTR SS:[EBP-28],1000	
	0043EF74	MOV EAX,DWORD PTR SS:[EBP+8]	EAX=00AB4328
	0043EF77	MOV DWORD PTR SS:[EBP-24],EAX	
	0043EF7A	CALL DWORD PTR DS:[<&KERNEL32.GetCurrentThreadId>]	
		EAX=00000A68	
Tracing Patching After			
	0043EF60	MOV DWORD PTR SS:[EBP-18],ESP	
Target Pos	0043EF63	XOR EAX,EAX	EAX=00000000
	0043EF65	NOP	
	0043EF66	NOP	
	0043EF67	NOP	
	0043EF68	NOP	
	0043EF69	TEST EAX,EAX	
	0043EF6B	JE SHORT 0043EFAD	
	0043EFAD	MOV ECX,DWORD PTR SS:[EBP-10]	ECX=0012FB98
	0043EFB0	MOV DWORD PTR FS:[0],ECX	
	0043EFB7	POP ECX	ECX=7620CCAB
	0043EFB8	POP EDI	
	0043EFB9	POP ESI	
	0043EFBA	POP EBX	
	0043EFBB	MOV ESP,EBP	
	0043EFBD	POP EBP	EBP=0012FBA4
	0043EFBE	RETN	
	004085EF	MOV DWORD PTR SS:[EBP-20],EBX	

 Copyright ⓒ 2014 SERSC

Table 8. Trace Change of Trojan.Agent.A

Tracing Patching Before			
	00407533	SUB ESP,0CC	
Target Pos	00407539	CALL DWORD PTR DS:[<&KERNEL32.IsDebuggerPresent>]	
		EAX=00000001	
	0040753F	TEST EAX,EAX	
	00407541	JNZ 00407644	
	00407644	MOV ESP,EBP	
	00407646	POP EBP	EBP=0012FF28
	00407647	RETN	
	00407C25	MOVZX EAX,AL	
	00407C28	CMP EAX,1	
	00407C2B	JE 00407CE4	
Tracing Patching After			
	0043EF60	MOV DWORD PTR SS:[EBP-18],ESP	
Target Pos	00407539	XOR EAX,EAX	EAX=00000000
	0040753B	NOP	
	0040753C	NOP	
	0040753D	NOP	
	0040753E	NOP	
	0040753F	TEST EAX,EAX	
	00407541	JNZ 00407644	
	00407547	PUSH 105	
	0040754C	CALL 00401072	
	00401072	PUSH EBP	
	00401073	MOV EBP,ESP	EBP=0012FE14
	00401075	SUB ESP,0C	

6. Conclusion

In this paper, we proposed a rule-based patching method to avoid anti-debugging techniques by analyzing assembly instructions. Our rule-based method, each time a new technology appears, can add or remove anti-anti debugging rules quickly. In addition, because this method do not execute program in debugger environments, it does not be detected by dynamic anti-debugging techniques. After spotting the sections containing anti-debugging instructions, our Evading Anti-debugging techniques tool patches the instructions with new instructions. Experimental results showed that our method can remove anti-debugging instructions from malware. Our future studies will be multi-byte sequence matching to improve processing speed.

Acknowledgements

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning (2011-0029924).

References

[1] P. C. Van Oorschot, “Revisiting software protection”, Information Security, Springer, **(2003)**, pp. 1-13.
[2] M. Brand, C. Valli and A. Woodward, “Malware Forensics: Discovery of the intent of Deception”, Proceedings of the 8th Australian Digital Forensics, **(2010)**, pp. 39-45.
[3] borg disassembler, http://www.caesum.com/.
[4] O. Yuschuk, Ollydbg. http://www.ollydbg.de/.
[5] Immunity inc, Immunity Debugger, http://www.immunityinc.com/.
[6] C. Eagle, “The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler”, No Starch Press, **(2008)**.
[7] Intel® 64 and IA-32 Architectures Software Developer Manuals. www.intel.com/products/processor/manuals/.
[8] M. V. Yason, “The art of unpacking”, Retrieved, **(2008)** February 12.
[9] A. Dinaburg, P. Royal, M. Sharif and W. Lee, “Ether: malware analysis via hardware virtualization extensions”, Proceedings of the 15th ACM conference on Computer and communications security, ACM, **(2008)**, pp. 51-62.

Copyright ⓒ 2014 SERSC

[10] L. Liu and S. Chen, "Malyzer: Defeating anti-detection for application-level malware analysis", Applied Cryptography and Network Security, Springer, **(2009)**, pp. 201-218.
[11] M. N. Gangon, S. Taylor and A. K. Ghosh, "Software protection through anti-debugging", IEEE Security & Privacy, vol. 5, no. 3, **(2007)**, pp. 82-84.
[12] P. Ferrie, "Anti-unpacker tricks", Proceedings of the CARO Workshop, Amsterdam, **(2008)**.
[13] Y. Kawakoya, M. Iwamura and M. Itoh, "Memory behavior-based automatic malware unpacking in stealth debugging environment", Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software (MALWARE), **(2010)**, pp. 39-46.
[14] X. Chen, J. Andersen, Z. M. Mao, M. Bailey and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware", Proceedings of the IEEE International Conference on Dependable Systems and Networks, **(2008)**, pp. 177-186.
[15] Anti-Anti-Debugger Plugins, https://code.google.com/p/aadp/.
[16] J. K. Lee, B. J. Kang and E. G. Im, "Rule-based Anti-anti-debugging System", Proceedings of the 2013 ACM Research in Adaptive and Convergent Systems, Monteal, Canada, **(2013)** October 1-4.

Authors

JaeKeun Lee, is a master student of Hanyang University, Korea. He got the B.S. degree in computer engineering from Hanyang University, Seoul, Korea, in 2012.

Research interests: Malware Analysis and Detection, Network Security. Parallel Computing, and Cloud Computing

BooJoong Kang, is a research engineer of Division of The Research Institute of Industrial Science at Hanyang University, Seoul, Korea. He got B.S. and M.S. from Hanyang University in 2007 and 2009 each, and Ph.D. from Hanyang University in 2013

Research interests: Malware Analysis, RFID and SCADA Security

Eul Gyu Im, is a faculty member of Department of Computer and Software at Hanyang University, Seoul, Korea. He became a Member of IEEE in 1994. He got B.S. and M.S. from Seoul National University in 1992 and 1994 each, and Ph.D. from University of Southern California in 2002. Before joining Hanyang University, he worked for National Security Research Institute in Daejeon, Korea. He is also a member of ACM

Research interests: Malware traffic Analysis, Malware Binary Analysis, RFID Security, and SCADA Security

 Copyright ⓒ 2014 SERSC