Eliminate Evading Analysis Tricks in Malware using Dynamic Slicing

Peidai Xie, Xicheng Lu, Yongjun Wang and Jinshu Su

School of Computer, National University of Defense Technology,
Changsha Hunan, China
peidaixie@gmail.com, xclluu@163.com, wwyyjj1971@126.com

Abstract

In order to be a long time alive, modern malware often make anti-emulation check after launched for evading dynamic analysis. Malware authors gain fingerprint information of target environment through several API to detect whether their creations are running in monitored state or not. If an emulated analysis environment is detected, the malware will change its running to avoid malicious behavior exposing. The existing approaches are based on trace matching with a intuition that, given the same inputs, the execution of a program should be the same in simulated analysis environment and on a real hardware reference system. However those approaches are too fine-grained to be inefficient.

In this paper we propose an approach to deal with anti-emulation using instruction traces and dynamic slicing. With a difference from trace matching solutions presented in existing references, our approach is performed on one instruction trace derived from our dynamic analysis platform. We evaluate our approach with 189 malware samples collected in the wild. The experience shows that our proposed approach can spot efAPI used for anti-emulation check in an efficient way.

Keywords: *evading analysis, malware analysis, dynamic slicing, instruction trace, dynamic analysis*

1. Introduction

Malware, such as virus, worm, Trojan horse or bot, is still a major security threat to internet nowadays. It is basis to gain and understand malicious execution behaviors for detection and containment of malware effectively, which is performed by anti-virus software with proper malicious behaviors models captured by malware analysis tools. Static analysis and dynamic analysis are two prevalent malware analysis techniques. While static malware analysis is limited by runtime packing and code obfuscation in malware samples, dynamic malware analysis techniques are main approaches to gain malicious behavior.

Dynamic analysis, analyzing the actions performed by a program while it is being executed [1], also suffers from several limitations, such as incompleteness of the analysis result due to a single execution path. The root cause refers to two aspects, one is the virtual environment applied in dynamic analysis system, and another is the malware with intentionally subtle structure.

Virtual environment for malware analysis is not *transparent* to processes running in it. Meanwhile, malware is embedded intentionally with codes for detection of runtime environment, means anti-emulation. If a malware sample judges the running

environment to be a virtual environment or a debugger, it will quit simply. Malware also hides its malicious behaviors into conditional branches with a special input data to trigger its execution, *e.g.*, a logic bomb. The input data for triggering can be a message received from network communication or some information from running environment. The ultimate goal of malware is to stay a long time alive as much as possible.

The evading dynamic analysis approaches can be classified into detection of execution environment, attack of virtual environment and execution behaviors hidden. Detection of execution of environment is the action that malware check whether the running system is a real hardware platform or not by characteristic feature of it. Attacking a virtual environment means that malware will disturb the analysis environment or make it crash. An execution behavior hidden is the runtime behavior that some malicious code fragments will execute when expected conditions are satisfied. The last two methods will be studied in the future. This paper discusses the problem of detection of execution environment in malware, which is a runtime behavior that after a malware sample is launched, the running will get some fingerprint information of execution environment to make a decision that the process will quit or run continually. Ordinarily, the code fragment of anti-emulation check is executed at the beginning of an executable's running.

To defeat anti-emulation check in malware analysis, we can try to build a *full transparent* simulated analysis system, or detect the code fragment of anti-emulation check and then eliminate its running effectiveness. While building a full transparent simulated analysis system is nearly impossible. There is always some detectable fingerprint information in an actual virtual analysis system. The approaches proposed by existing references focus on how to detect anti-emulation actions. The [2] and [3] present an instruction trace matching approach which aligns two traces collected from a virtual platform and a *reference platform* to find a divergence point in order to build a dynamic state modification (DSM) which is used for fooling the malware's anti-emulation check. It is effective for instruction-level anti-emulation check, but the reference platform is impractical.

After analyzed a large number of malware samples in our iPanda[4] system, we find that malware can also performs anti-emulation check through fingerprint information of analysis environment. For example, the *dbot* worm invokes *GetUserName*() to check if the user is in its blacklist of user names. Windows operation system provides several API functions for gaining system-level information, which can be misused for identifying the running environment. We named those API **environment fingerprint API**, efAPI for short.

This paper focuses on such a type of evading dynamic analysis tricks in malware. A lightweight approach, which is based on dynamic slicing and execution pattern inspection, is proposed to detect and contain the anti-emulation actions that are achieved through efAPI in malware. This approach allows us to understand the efAPI based evading dynamic analysis techniques and also build a DSM for bypass such program branches in order to gain more malicious behaviors.

The remainder of this paper is structured as follows. In Section 2, we explain the problem in detail and give an actual example. In Section 3, we describe detailed the proposed approach, including an instruction tracer, the execution pattern inspection, dynamic backward slicing and building and applying DSM. Section 4 evaluates our approach. Section 5 gives a discussion, and in Section 6 we introduce some related work in detection of evading analysis techniques in malware. Finally, Section 7 concludes this paper.

2. Problem Statement

The evading dynamic analysis tricks, or anti-emulation checks, in a malware sample, are a number of instruction sequences embedded by attackers. Those code fragments are used to detect whether it is running in a virtual environment rather than on a user's real system and refuse to perform its malicious behaviors [5]. Therefore malware should be able to find the fingerprint information of a virtual environment firstly, and hide its malicious behaviors after identifying an emulator. To quit the running is a common way according our study, in addition raising an exception or executing a loop for a long time is usually used.

There are lots of tricks that can be played by malware for anti-emulation check of a motley variety of dynamic analysis platforms. At present, the analysis tools built on top of virtual machine increase in popularity and the corresponding anti-emulation check strategies are based on system fingerprint information gained through efAPI.

We present an actual sample of such efAPI based anti-emulation. We rewrite the corresponding instruction sequences using C-style pseudo code of an instance of *dbot.exe* of which the MD5 is c3238557177d8ffb2385caffbb6cc5ad and named HEUR:Trojan.Win32.Generic by Kaspersky anti-virus engine, after reversing engineering on it, shown in Figure 1.

```
bool isInVirEnv(){
        char *blacklist[] = {"sandbox", "honey", "vmware", "nepenthes"};
        char user[128];
        DWORD size = 128;
        int i;
        GetUserName(user, &size);
        for(i = 0; i < 5; i++) {
                if(strstr(user,blacklist[i]))
                        return true;
        }
        return false;
}

int main(int argc, char *argv[]){
        …
        if(isInVirEnv())
                return 0;
        …
}
```

Figure 1. The C-style Pseudo Code of a *dbot* Sample Instance

After the sample is launched, the function *isInVirEnv()* will be invoked in which a windows API *GetUserName()* is called and the *user* logged into current system is returned. And then a predefined blacklist in which user names are common set up in simulated analysis environment is retrieved to see if the *user* is in it. If the *user* is in the blacklist, the *isInVirEnv()* will return *true* and the running will quit immediately.

3. A Slicing-based Approach to eliminate Anti-emulation Checks

Inspired by dynamic program slicing techniques which is applied to find all statements that really affected a variable occurrence [6], we proposed a slicing-based approach to eliminate anti-emulation checks, named SliceBEAC shown in Figure 2.

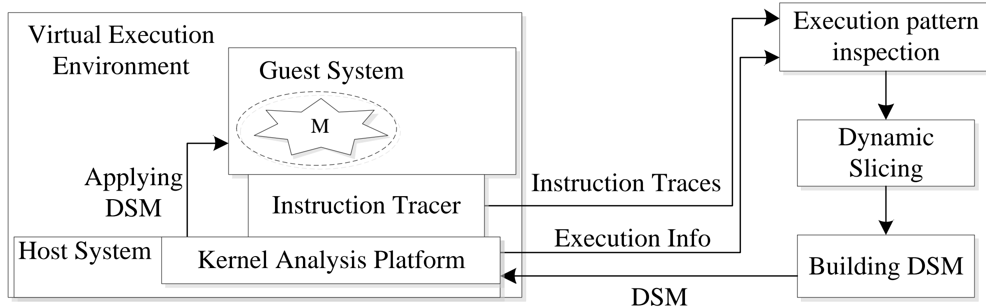

Figure 2. The Workflow of SliceBEAC

There are four steps to complete an elimination of a code fragment for anti-emulation check, corresponding four components in Figure 2, *i.e.*, a simulated analysis platform iPanda, the execution pattern inspection, dynamic slicing and building DSM.

The brief process is running a malware sample in iPanda, firstly, and recording instruction traces for execution pattern inspection and dynamic slicing, and then building and applying DSM to disable anti-emulation check.

3.1. The Instruction Tracer

The instruction tracer implemented in host system of iPanda is responsible for recording the instructions executed in the process of an executable for fine-grained malware analysis. If the target process does not stop its running, instruction tracer will kill it at the end of 5 minutes tracing.

In order to monitor more anti-emulation check, we deliberately setup an easier check guest system. For example, in the guest system we reserve a lot of characteristics of QEMU, install several debuggers, and log in using a user named ‘vmware’ with administrator rights, *etc.*

3.2. Execution Pattern Inspection

A instruction trace is too rough to be used for slicing. Execution pattern inspection is responsible for trimming the trace, identifying windows API, building dynamic control flow graph (d-CFG) and deciding if a slicing will be performed or not. Trimming traces and API identification are the process to replace instructions in DLL modules with API prototype as well as spot original entry point (OEP) of the executable if runtime packing is detected. After that, a subtle trace consisted of instructions in executable module is done. The d-CFG building is for slicing as well as program dependent graph (PDG) on the trimmed trace that will be done through well-known algorithms. If execution is quit with at most γ APIs invoked and the number of instructions executed is at most δ, it is likely to be anti-emulation check and slicing will be performed. The algorithm is shown in Table 1.

Table 1. Execution Pattern Inspection Algorithm

Algorithm 1: execution pattern inspection.
Input：*T*, M_{dll}, M_{exe}
T is an instruction trace <I_1, I_2, …>, *M is address layout of modules*, M_{dll} *is dll module*, M_{exe} *is the malware instance executable.*
Output：*d-CFG, isSlicing*

```
Stage 1: trimming the trace
begin
   Sub_T ← Φ
   foreach I in T do
      if I is CALL and Target_I is F∈M_dll then
         Sub_T = Sub_T ∪ { I }
         until I is RET and NEXT_I∈M_exe
      end
   end
   substitute F for Sub_T
   repeat the procedure on T
end

Stage 2: building d-CFG
begin
   building a d-CFG on T' using common CFG algorithm
end

Stage 3: slicing or not
begin
   num_F = 0
   size = 0
   foreach I in d-CFG do
      size += Size(I)
      if I∈{F|F is in d-CFG} then
         num_F++
      end
   end
   if num_F<=γ and size/Size(M_exe)<=δ
then
      isSlicing=true
   else
      isSlicing=false
   end
end
```

The input of the execution pattern inspection algorithm includes an instruction trace, the address layout of DLL and EXE modules, and it outputs a d-CFG and a decision for performing slicing or not.

Firstly, the trace *T* is trimmed to be a short one that lots of instructions in DLL are removed and substituted with known APIs. Secondly, using common CFG algorithm, a dynamic CFG on *T* is builting. And finally a decision is made according invoked APIs and instructions. The γ and δ should be figured out with experiments.

3.3. Dynamic Backward Slicing Algorithm

The last step is to perform a dynamic slicing algorithm on a trimmed trace. The slice *criterion* is <I_c, *var*> that the I_c is the memory address of a conditional branch instruction and *var* is its conditional variable. Sometimes the slicing should be repeated on several criterions to find a root cause efAPI. For example, several functions are invoked to delete files of malware samples in dictionary *c:\windows\system32* after the root cause efAPI is returned successfully and then quit the running. We present the algorithm in Table 2.

This is actually a dynamic backward slicing algorithm. The input of it includes the instruction trace T, dynamic control flow graph d-CFG and the slice criterion and it outputs a dynamic slice on this criterion. This algorithm traverses the *T* to build program dependence graph with d-CFG to produce the slice. This procedure will iterate until the root cause efAPI is spotted.

Table 2. Dynamic Backward Slicing Algorithm

Algorithm 2: dynamic slicing algorithm on condition variable.
Input: *T, d-CFG,* $<I_c, var>$.
T is an instruction trace, *d-CFG* is a dynamic CFG on *T*, $< I_c , var >$ is slice criterion, I_c is condition branch instruction in *d-CFG*, *var* is condition variable in *T.*
Output: A dynamic program slice *S* on *var*.

```
Begin
  S ← Φ
  Define(var) ← Φ
  Reference(var) ← I_c
  foreach I in Define(var) do
    foreach r when I ∈ Reference(r) do
      foreach I from I_c to I_call-self backward do
        if I Reference var
        then
          Reference(var) ← I
          S ← I
        elif I Define var then
          Define(var) ← I
          S ← I
        endif
      end
    end
  end
  return S
end
```

3.4. Building and Applying DSM

If the efAPI is spotted, we build corresponding DSM and apply it to our dynamic analysis tool for mining more malicious behaviors. A DSM is a dynamic state modification $<M_{addr}$, *type*, *value*$>$, means that change the M_{addr} into *value* with *type* length when the running is reached it. There are several complex memory state modifications that can be used to indicate some code is executed or not. We will study such scenarios in the future.

An anti-emulation check code fragment can derive a DSM usually. A DSM will be used to apply simulated analysis environment to eliminate such checks. Meanwhile, next workflow of SliceBEAC will start for other anti-emulation checks.

4. Experiment

We conducted the experiment as follows. Firstly, a set of malware samples gathered in the wild on Mar. 2011 are executed in iPanda for recording traces, and then execution pattern inspection algorithm is performed for picking proper traces for slicing. The set of samples and their traces are shown in Table 3. We have 189 samples and the number of them with execution normal termination (#ENT) is 73. #UIns.inTrace means the total numbers of instructions uniquely in a trimmed trace and CC is the code coverage in instruction level.

Table 3. The malware samples and their traces

#Sum	# ENT	#UIns.inTraces			
		ID	#samples	#UIns.	CC
189	73	S1	12	<1k	2.31%
		S2	54	1k ~ 3k	8.26%
		S3	7	>3k	43.03%

The execution pattern inspection hits 8 in 73 samples for performing slicing (#ForSlicing) when γ is 20, δ is %5, and 7 samples are successful sliced (#SuccSlicing), as is shown in

Table 4. The failed one when performing slicing is because that the sample uses SEH-based anti-emulation check, which installs an exception handler and triggers an arithmetic exception to execute the handler function to check analysis environment.

Table 4. The result of slicing whenγ is 20 and δ is %5

ID	#ForSlicing	#SuccSlicing	Average Time
S1	5	4	13s
S2	3	3	61s
S3	0	0	-

The efAPI list found by our approach is shown in Table 5. We can see that all 7 samples invoke *IsDebuggerPresent*() to detect debuggers, and *CreateMutex*(), *RegQuery*() and *CreateFile*() are used to detect some existing software fingerprint information. The #Q stands for the number of samples that quit running at the API and the #A stands for the number of samples that invoked the API.

Table 5. The efAPIs used by malware samples in our experiment

API	#Q	#A	API	#Q	#A
IsDebuggerPresent()	2	7	CreateMutex()	1	1
CheckRemoteDebuggerPresent()	0	1	RegQuery()	2	1
GetComputerName()	0	1	CreateFile()	1	3
GetUserName()	1	2			

In Table 5, the API *CheckRemoteDebuggerPresent()* and *GetComputerName()* are not spotted by our SliceBEAC directly because the samples with such a type of anti-emulation checks are not aim at our dynamic platform. The API *CreateMutex()* is often used to check if a debugger is present or not. One sample uses *RegQuery()* to make a check that if VMTools is installed into current system. In addition, there is other API which can be used for environment fingerprint API.

We evaluate our proposed SliceBEAC approach on 189 malware samples and we can figure out which efAPIs are used for anti-emulation checks.

5. Discussion

It is a try for detection of evading dynamic analysis in malware using SliceBEAC, a slicing-based approach to eliminate anti-emulation check by efAPIs. There are some advantages as well as limitations and countermeasures. The reference platform is not used in the approach for efficiency and easiness, while several types of evading dynamic analysis tricks such as instruction-based emulated analysis environment attacking code fragments cannot be detected. In addition, the approach is weak for attacks with complex program control flow since a time-consuming program dependence graph should be determined when doing dynamic slicing. Even so, we still think that it is a good beginning for future study on defeat evading dynamic analysis techniques in malware.

6. Related Work

Malware analysis is the process of determining the purpose and functionality of a given malware sample [7, 8]. Automated dynamic malware analysis techniques[1] are promising approaches for modern malware that impeded lots of evading analysis tricks[9, 10]. Dynamic analysis platform is built on top of virtual machine or full system emulators [11, 12]. Unfortunately, a simulated analysis system can be always detected or attacked successfully [13].

Evading analysis techniques in malware are instruction sequences for hindering the malware analysis process [14]. Runtime packing, anti-debugging, anti-simulation and hiding of malicious behaviors are all evading analysis tricks [15, 16]. Only those evading analysis tricks are eliminated that more malicious behaviors of a malware sample can be mined for effective malware detection and containment.

7. Conclusion

Anti-emulation check is nearly essential component in modern malware for evading dynamic analysis by malicious behavior hidden in order to be a long time alive. Execution environment fingerprint information extracted by several API is used by malware to perform anti-emulation check. In this paper we present a slicing-based approach for potting such efAPI. We evaluate the approach in our malware analysis tool with 73 samples. The result show that the slicing-based approach can find out efAPI within several seconds. In the future we will enhance this approach to deal with more evading dynamic analysis techniques.

Acknowledgements

This work was partially supported by the National Natural Science Foundation of China under Grant No. 61003303 and No. 60873215, Hunan Provincial Natural Science Foundation of China No.s2010J5050 and 11jj7003, the PCSIRT (NO.IRT1012), and the Aid Program for Science and Technology Innovative Research Team in Higher Educational Institutions of Hunan Province “network technology”, and National High Technology Research and Development Program of China (No.2011AA01A103).

References

[1] M. Egele, T. Scholte, E. Kirda and C. Kruegel, "A Survey on Automated Dynamic Malware Analysis Techniques and Tools", ACM Computing Surveys, **(2010)**.
[2] M. G. Kang, H. Yin, S. Hanna, S. McCamant and D. Song, "Emulating Emulation-Resistant Malware", The 2nd Workshop on Virtual Machine Security. **(2009)** November 9; Chicago, Illinois, USA.
[3] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda and G. Vigna, "Efficient Detection of Split Personalities in Malware", Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10). **(2010)**. San Diego, CA, USA.
[4] P. D. Xie, X. C. Lu, Y. J. Wang, J. S. Su and M. J. Li, "iPanda: A Comprehensive Malware Analysis Tool", The International Conference on Information Networking(ICOIN'13), **(2013)**, Bangkok, Thailand.
[5] D. Shin, C. Im, H. Jeong, S. Kim and D. Won, "The New Signature Generation Method based on an Unpacking Algorithm and Procedure for a Packer Detection", International Journal of Advanced Science and Technology, vol. 27, **(2011)**.
[6] H. Agrawal and J. R. Horgan, "Dynamic Program Slicing", Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, **(1990)** June 18, USA.
[7] U. Bayer, C. Kruegel and E. Kirda, "Dynamic Analysis of Malicious Code", Journal in Computer Virology, vol. 2, no. 1, **(2006)**.
[8] H. Lu, X. Wang and J. Su, "CCS: Collaborative Malware Clustering and Signature Generation using Malware Behavioral Analysis", International Journal of Hybrid Information Technology, vol. 5, no. 2, **(2012)**.

[9] P. D. Xie, M. J. Li, Y. J. Wang, J. S. Su and X. C. Lu, "Unpacking techniques and tools in malware analysis", 2012 International Symposium on Electronical Engineering and Information Processing (ISEEIP'12), **(2012)**, Shen Yang, China.
[10] F. Guo, P. Ferrie and T. Chiueh, "A Study of the Packer Problem and Its Solutions. Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID' 08), **(2008)**.
[11] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis", Proceedings of the 4th International Conference on Information Systems Security (ICISS'08, keynote invited paper), **(2008)** December, Hyderabad, India.
[12] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík and A. Meduna, "Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis", International Journal of Security and Its Applications, vol. 5, no. 4, **(2011)**.
[13] L. Martignoni, R. Paleari, G. F. Roglia and D. Bruschi, "Testing System Virtual Machines". Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'10), **(2010)** Trento, Italy.
[14] X. Chen, J. Andersen, Z. Mao, M. Bailey and J. Nazario, "Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malwar", IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN'08), **(2008)**.
[15] M. G. Kang, P. Poosankam and H. Yin, "Renovo: A Hidden Code Extractor for Packed Executables", Proceedings of the ACM Workshop on Recurring Malcode, **(2007)** New York, NY, USA.
[16] H. C. Kim, D. Inoue, M. Eto, Y. Takagi and K. Nakao, "Toward Generic Unpacking Techniques for Malware Analysis with Quantification of Code Revelation", Joint Workshop on Information Security, **(2009)** August; USA.

Authors

Peidai Xie received the B.S. degree and M.S. degree from the National University of Defense Technology (NUDT) in 2006 and 2008, respectively, all in school of computer. He is a PhD candidate in Institute of Network and Information Security, NUDT since March 2009. His current research interests are in automated dynamic malware analysis, malicious behavior mining, malware detection and containment, program reverse engineering, and vulnerability analyzing, *etc*.

Xicheng Lu is a full professor and a Ph.D. supervisor, a member of Chinese Academy of Engineering. His research interests include distributed computing, computer networks and communications.

Yongjun Wang received the MSc degree and PhD degree in Computer Science from National University of Defense Technology, China, in 1995 and 1998 respectively. After that, he has worked at School of Computer, National University of Defense Technology as a lecturer, associate professor and full professor in 1998, 2000 and 2006 respectively. His current research interests are network security, system security and high performance network.

Jinshu Su received the B.S. degree in Mathematics from Nankai University in 1983, the M.S. degree and PhD degree in Computer Science, NUDT in 1989 and 1999, respectively. He is a full professor, the head of the Institute of network and information security, NUDT. His research interests include high performance routers, high performance computing, wireless networks and information security. He is a member of the ACM and IEEE.