Towards the Detection of Emulated Environments via Analysis of the Stochastic Nature of System Calls

Tauhida Parveen
Dept. of Computer Sciences
Florida Institute of Technology
tparveen@fit.edu

William Allen
Dept. of Computer Sciences
Florida Institute of Technology
wallen@cs.fit.edu

Scott Tilley
Dept. of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Gerald Marin
Dept. of Computer Sciences
Florida Institute of Technology
gmarin@cs.fit.edu

Richard Ford
Dept. of Computer Sciences
Florida Institute of Technology
rford@cs.fit.edu

Abstract

One of the most powerful tools in the hacker's reverse engineering arsenal is the virtual machine. These systems provide a simple mechanism for executing code in an environment in which the program can be carefully monitored and controlled, allowing attackers to subvert copy protection and access trade secrets. One of the challenges for anti-reverse engineering tools is how to protect software within such an untrustworthy environment. From the perspective of a running program, detection of the emulated environment is not trivial, as the attacker can emulate the result of different operations with arbitrarily high fidelity. Thus, an emulated environment may be - prima facie - indistinguishable from a "real" environment. However, this conclusion may well be false: this paper demonstrates a mechanism that is able to detect even carefully constructed virtual environments by focusing on the stochastic variation of system call timings. A statistical technique for detecting emulated environments is presented, which uses a model of "normal" system call behavior to successfully identify two commonly used virtual environments under realistic conditions.

Keywords: reverse engineering, security, digital rights management, emulation, virtual machine

1. Introduction

Virtual or emulated execution environments are being applied to a variety of new applications, such as software testing [7], distributing pre-configured software [11], and enhancing computer science education [3]. While such an environment may be used for purely legitimate purposes, the unauthorized or malicious use of these technologies is also increasing. One such use is the reverse engineering of binaries that contain protected or proprietary information. This motivates the need for techniques that can determine programmatically whether an application or operating system is executing in an emulated or virtual environment.

This paper presents a technique that has shown promise in detecting when a program is executing in a virtual environment, without prior knowledge of the specific environment in use. This technique relies on changes in the distribution of system call timing that result from the additional processing time required to provide a non-native execution environment. A model of "normal" system call timing is derived for a range of hosts and that model is used to successfully detect program execution in two common virtual environments.

The next section of this paper briefly describes native and emulated execution environments. Section 3 details the development of the detection methodology and describes a proof-of-concept implementation of the approach. Section 4 outlines experiments that were conducted to test the methodology and provides an evaluation of the results. Section 5 summarizes the paper and discusses possible avenues for future work.

2. Background

Chikofsky & Cross define reverse engineering as "analyzing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information" [5]. Applications of reverse engineering include construction

of a new software product or maintenance of legacy applications, deconstructing software for the purpose of teaching, repairing malfunctioning systems, porting software to run on a different operating system, or developing new applications that run in conjunction with the legacy software [2].

Reverse engineering can also be used to reveal and circumvent protection mechanisms, possibly leading to unauthorized access to protected intellectual property or digital content. Because binary reverse engineering frequently requires monitoring instruction execution, tools such as debuggers and emulators are necessary. A debugger modifies the object code so that it can track program state or control program execution. However, anti-debugging techniques have been developed that can detect those modifications and prevent further monitoring of program execution [6].

Most emulators (or a dynamic debugger that provides emulated environments) use a compatibility layer that translates system calls to the native execution environment[1] into system calls for the emulated execution environment[2]. Because of this compatibility layer, the access restrictions present in a program may be by-passed in the emulated environment, thus exposing information that would otherwise be protected. Therefore, individuals wishing to circumvent protection mechanisms often use an emulated environment.

Several suggestions have been made to change the technology used to produce software so that it is more resistant to attempts to circumvent copyright protection schemes [4]. These techniques focus on building preventive measures into the application. They assume that the application is executing where it is expected to run, in a native environment. It is possible that these methods can be by-passed or security measures can be removed when the application is being executed in an emulated execution environment.

In this research, virtual machines were used to create emulated environments. Virtual machines are optimized; they generally execute code faster than emulators. The emulated environments created by virtual machines are usually better in performance than traditional emulators [13]. Two commercial virtual machine products VMware Workstation [12] and Virtual PC 2004 [8] were used in this research to provide consistent emulation of x86-based applications. Although these products can support a range of operating systems, the experiments described here were performed using the Windows XP system.

[1] In this research, the term *native* execution environment describes a configuration in which an operating system is installed on a physical machine and interacts directly with the hardware of that machine to support program execution.

[2] An *emulated* execution environment provides an imitation of a native environment where the operating system actually interacts with the physical machine only through a layer of software (a virtual machine or other emulation program) that is itself running in a native execution environment.

3. Detecting an Emulated Environment

The timing of a process' execution is affected by small fluctuations and variations by the execution environment's behavior. These microscopic fluctuations in performance provide the key behavior used in this research to gain a better understanding of the nature of an execution environment.

To improve security and reliability, application software is restricted from gaining direct access to most system resources. System calls are used to request the operating system for restricted actions such as accessing I/O devices or system memory. During the execution of a system call, interrupts and other unpredictable events can cause timing delays. Thus, if one were to measure the duration of individual system calls executed in a native environment, the timing of those individual calls could be expected to show artifacts that represent the interaction of physical and software processes.

This behavior is different in an emulated execution environment and should show variability in timing that is distinct from the durations measured when executing the same system calls in a native environment. This is due to the addition of the compatibility layer and the presence of other applications running in the same native environment as the emulation. Therefore, the detection methodology developed in this research is based on a statistical analysis of the timing properties of system calls in both environments.

The Windows XP environment provides a number of different methods for measuring the passage of time. While the default low-level timer is the *GetTickCount()* function, Windows XP also provides a high resolution counter that can produce more accurate timing measurements. The *QueryPerformanceCounter()* function [9] can be used to retrieve the current values of a high resolution counter. In this research, timing data from system calls executed in both native and emulated environments was gathered by calling the *QueryPerformanceCounter()* function at the beginning and at the end of a section of a code.

It should be noted that the emulation environment might intentionally alter any internal source of timing information in an attempt to hide its presence. However,

the goal of this research is to determine if an emulated environment impacts the system call timing significantly enough for that characteristic to be used for detection, not to present a foolproof detection mechanism.

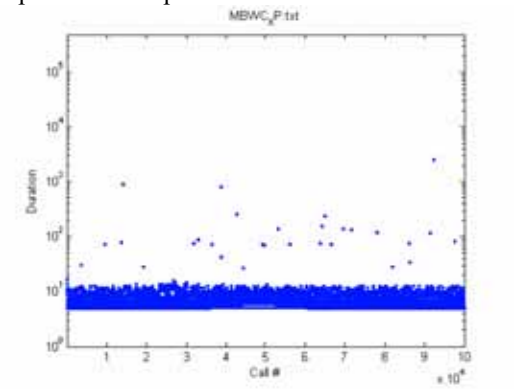

Figure 1 (a): System Calls in Windows XP (log scale)

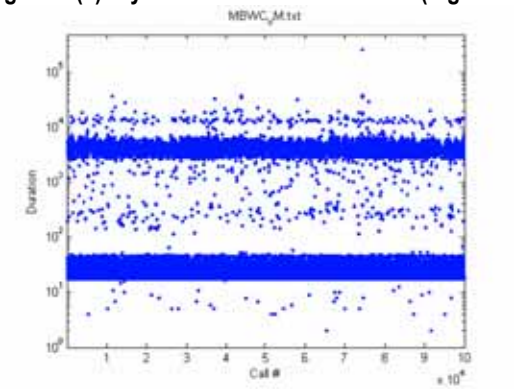

Figure 1 (b): System Calls in VMWare (log scale)

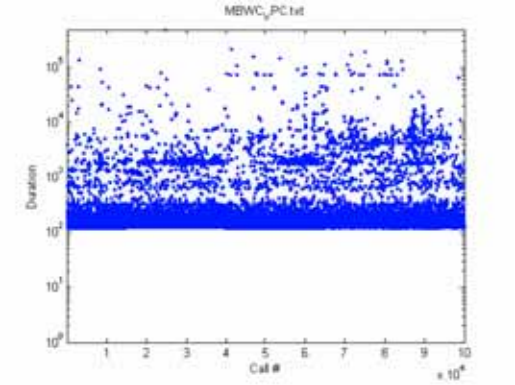

Figure 1 (c): System Calls in Virtual PC (log scale)

The hypothesis that timing artifacts can be used to distinguish between a native and emulated execution environments came from observing the timing variations of system calls from both environments, as shown in Figure 1 (a-c). These plots show the durations (on a log scale) of the same set of system calls executed in each of the three environments (native XP, VMWare and Virtual PC) running on the same platform.

Based on the analysis of this variations in system call timing shown in Figure 1, a methodology was developed that could be used in real-time to determine whether the underlying execution environment was native or emulated. The detection methodology follows three steps which are described in more detail below:

3.1 Gathering System Call Timing Data

The model of normal behavior was created by the execution of system calls in native environments on several different platforms. This information was gathered by a program written in the C language which executes a series of system calls, logging the start and end timestamps for each call. Fifteen non-I/O system calls were chosen from the Win32 API. Table 1 lists the system calls that were used in the experiments. Specific information on individual calls is available from the MSDN Library [10]. System calls that are associated with input/output (I/O) activities were purposely avoided since their timing measurements can be affected by unpredictable hardware or software-related events. Note that this set of system calls is not intended to represent the calls most commonly used in current applications; they were chosen randomly from the non-I/O calls provided by the Win32 API. Further work is needed to determine the best mix of calls to use for reliable detection.

Table 1: The set of Win32 system calls used to create the model of 'normal' behavior

GetVersion	*GetTickCount*
MutiByteToWideChar	*GetComputerName*
IsCharAlpha	*GetSystemTime*
CharLower	*GetTimeZoneInformation*
GetTimeFormat	*CharUpper*
GetSystemInfo	*GetSysColor*
GetSystemDirectory	*GetNumberFormat*
GetLocaleInfo	

Each system call was executed a total of 10,000 times, but the calls were made in a random order so that interactions between calls were not a factor. Each run of the experiment resulted in a total of 150,000 individual calls. The *QueryPerformanceCounter()* function was used to record the start and end timestamps for each system call as they executed in the native execution environment. From the start and end timestamps, the durations of each call were calculated (Duration = end timestamp – start timestamp). The mean and standard deviation of the durations were also calculated.

The durations of the system calls gathered from the native execution environment did not fit any well-known statistical distribution. Therefore, a threshold was determined such that the threshold would, with high probability, contain the system call durations from the native environment but would not contain many of the durations of the system calls executed in non-native execution environments.

3.2 Selecting a Detection Threshold

A number of trials were conducted to determine an optimum threshold that would separate the system call durations gathered from a native execution environment from those gathered from a non-native environment. The range of trial thresholds varied from 1 to 4 standard deviations above the mean of the durations from the native environment. The following steps show how the optimum threshold level was reached.

1. Let T_i be the random variable that is considered to be the duration of the system calls.
2. The sample mean and standard deviation for T_i is calculated.
3. Let C be a constant value from 1 to 4 with an increment of .01
4. Let X_C be the thresholds -
   X_C = *Sample Mean of durations + (1+C * .01)* sample standard deviation*

From the set of thresholds generated by this method, an optimum threshold was chosen such that it contains a significant percentage of the native system call durations with low probability of false alarm. To determine this optimum threshold, the probability that the durations of system calls in the native environment would exceed that threshold level was calculated for each increment in the range of potential threshold values:

1. let P_τ be the probability of system call duration greater than the threshold, τ
2. $$P_\tau = \frac{\text{count the number of duration values that are} > \tau \text{ (threshold)}}{total\_number\_of\_runs}$$

It was found that the probability of durations exceeding 4 standard deviations above the mean was less than 0.05. Therefore, the threshold of four standard deviations above the mean was chosen to distinguish between the native execution environment and the emulated execution environment. (It was determined that this threshold produced an acceptably low false alarm rate, so no calculations were performed above the range of 1 to 4 standard deviations, however it is possible that higher threshold values may produce acceptable results as well.)

However, because the system call durations do not follow a particular distribution there is no guarantee that the probability of durations exceeding the threshold would always be below 0.05. Therefore, Chebyshev's inequality [1] was used to determine the probability of durations that would be allowed to exceed the threshold without generating a false alarm.

Chebyshev's inequality states that in any data sample or probability distribution, nearly all the values are close to the mean value, and provides a quantitative description of terms like *nearly all* and *close to*. For example, no more than 1/4 of the values are more than 2 standard deviations away from the mean, no more than 1/9 are more than 3 standard deviations away and no more than 1/16 are more than 4 standard deviations. Although Chebyshev's theorem states that no more than 1/16 of the data should be *away* from the mean, in this research only the value *above* the mean was considered. Therefore, in a native environment, no more than 1/16 (or 0.0625) of the system call durations will be more than 4 standard deviations above the mean. If measurements determine that the fraction of system calls above this threshold is greater than 0.0625, then a significant environmental change (such as the introduction of an emulator) is likely.

3.3 Measuring System Call Timing Data

The last step for the detection of emulated environments is to compare probabilities of the timing being above the set threshold from the native and the emulated environments.

4. Experiments and Results

Several experiments were carried out to determine if the model created from the system call timing data gathered from native execution environments could be used to reliably determine whether a new execution environment was native or non-native. The model was also tested in a variety of native execution environments to determine if false alarms would occur.

For these experiments, the native environment under study was Windows XP (SP2), installed on a range of machines, each with a different processor speed and mix of installed software. Virtual machine environments, VMware and Virtual PC 2004, were used for these experiments and the operating system used within both of these emulated environments was also Windows XP.

Table 2: Threshold calculated for each processor by the method described in Section 3.2 (columns 2, 3 are based on system call durations measured by *QueryPerformanceCounter()*)

Processor (GHz)	Native Mean	Native Standard deviation	Threshold (Mean + 4*standard deviation)
1.5	24.51	55.79	247.67
1.8	21.71	46.38	207.23
2.26	14.32	29.73	133.24
2.5	14.21	29.16	130.85
3.0	13.58	40.69	176.34
3.4	11.99	35.85	155.39

Table 3: Proportion of system calls exceeding the thresholds shown in Table 2

Processor (GHz)	Threshold from Table 2	Native environment proportion	VMware proportion	Virtual PC proportion
1.5	247.67	0.011133	0.159933	0.304373
1.8	207.23	0.014260	0.167987	0.305567
2.26	133.24	0.008559	0.101880	0.342140
2.5	130.85	0.007613	0.131040	0.113933
3.0	176.34	0.007307	0.176887	0.074800
3.4	155.39	0.009547	0.098053	0.117660

The set of test machines included six PCs, each with a Pentium 4 processor, which were chosen because they provide a representative sample of contemporary hardware. Two of the machines were equipped with a comparatively slow processor (1.5 GHz and 1.8 GHz), two were mid-range machines (2.26 and 2.53 GHz), and the other two are higher performance machines (Hyper-Threaded processors running at 3.0 and 3.4 GHz, respectively). Features such as RAM, disk space, software version, the mix of applications installed on the system, and processor type were evaluated to determine if they would have any impact on the detection methodology, but no evidence was found to support this.

When conducting the experiments to validate the model, the same C program described in Section 3.1 was used to gather timing data from all six sample machines in all three environments (native Windows XP, VMware and Virtual PC). Once the data was gathered from each machine, the technique described in Section 3.2 was used to calculate a threshold for each machine.

Table 2 shows the threshold from each of the six sample machines. The values shown in the second and third columns are calculated from the system call durations measured using *QueryPerformanceCounter()*. The thresholds presented in the fourth column are based on the calculation described in Section 3.2.

In the next step, system call durations gathered from the native and emulated execution environments on each machine were checked against these thresholds. The hypothesis was that most of the system call durations gathered from the native environment (Windows XP) would fall within the threshold. Using Chebyshev's inequality, durations measured in native execution environments should not exceed the threshold by more than 0.0625 (based on the calculations in Section 3.2).

If the proportion of measured durations exceeded the threshold by more than 0.0625, the environment would not fit the model for a native environment and would be classified as an emulated environment.

For each of the execution environments, Table 3 shows the proportion of system call durations that exceeded the corresponding threshold (from Table 2). Figure 2 graphically depicts the information shown in columns three, four and five of Table 3, clearly showing that the proportion is well below 0.0625 (established with Chebyshev's rule) for the native environments and above that value (in many cases significantly above) for each of the emulated environments.

From this observation, it can be deduced that all six native execution environments fit the model and the data from the emulated execution environments do not fit the model. However, some of the emulated environments that were running on high performance processors approach the threshold. To the casual observer, it may appear that a threshold might be found that more evenly separates the native and emulated environments. However, lowering the threshold without knowing the

true distribution of the system call durations risks increasing false alarms.

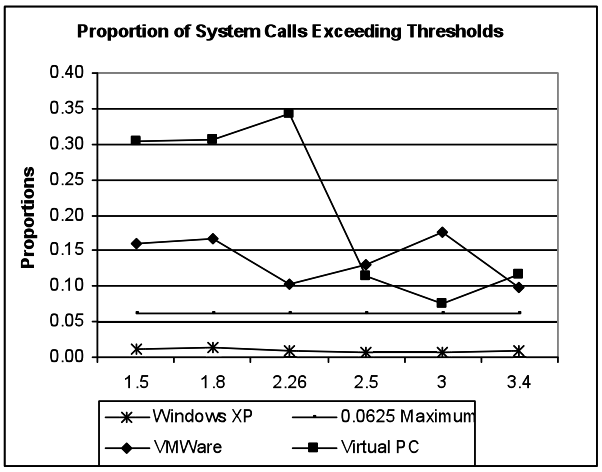

Figure 2: Graphical depiction of the results in Table 3

One solution to this problem would be to use a decision rule for determining when a false alarm occurs. A decision rule could be set such that, if the probability of the durations exceeding the threshold was checked *n* times, then so long as the probability did not exceed the threshold more than *k* out of *n* times, it could still be considered as a native environment.

5. Summary and Future Work

One of the most powerful tools in the hacker's reverse engineering arsenal is the virtual machine. These systems provide a simple mechanism for executing code in an environment in which the program can be carefully monitored and controlled, allowing attackers to subvert copy protection and access trade secrets. Detection of the emulated environment is not trivial, as the attacker can emulate the result of different operations with arbitrarily high fidelity.

This paper presented an approach to countering this threat by providing a means of automatically detecting the attempted reverse engineering. If it can be determined that the execution environment is monitoring the behavior of an application for the purpose of revealing or circumventing protection mechanisms, preventive measures may be taken to protect the application from such threats. The paper also described a proof-of-concept implementation of the approach. An evaluation of the results from experiments conducted to test the methodology suggested that the methodology is sound, and that the approach can successfully detect execution in an emulated environment.

The preliminary research described in this paper also shows that additional work is needed to produce an approach that is refined enough to be of general use. For example, it was mentioned that this approach was developed using non-I/O system calls that were chosen at random. The research could be expanded to include other non-I/O system calls, to determine if some system calls are more useful for building the model than others.

There is also an obvious need to evaluate the approach on non-Windows XP operating systems (and with emulated environments other than VMware and Virtual PC). It would be interesting to try the approach on other platforms, such as Linux, Mac OS X, or even gaming platforms such as the Xbox 360 and with other emulation environments, such as Bochs, Xen or WINE.

References

[1] Arnold A. *Probability, Statistics and Queuing Theory with Computer Science Applications*. Academic Press, 1990.

[2] Bennet, K. and Rajlich, V. "Software Maintenance and Evolution: A Roadmap." In *Proceedings of the 22nd International Conference on Software Engineering* (ICSE 2000). Future of Software Engineering Track, pp 73–87. New York, NY: ACM Press, 2000.

[3] Bullers, W. I.; Burd, S.; and Seazzu, A. "Virtual machines - an idea whose time has returned: application to network, security, and database courses", *Proceedings of the 37th SIGCSE Symposium on Computer Science Education*, 2006

[4] Collberg C. and Thomborson, C. "Watermarking, TamperProofing, and Obfuscation - Tools for Software Protection." *IEEE TSE,* Vol. 28, No. 8, August 2002.

[5] Chikofsy, E.; and Cross, J. "Reverse Engineering and Design Recovery: A Taxonomy." *IEEE Software* 7(1):13-17, January 1990.

[6] Eilam, E. *Reversing: Secrets of Reverse Engineering*, Wiley, 2005

[7] Magnusson, P.S, "The Virtual Test Lab", *IEEE Computer*, Vol. 38, No. 5, pp. 95–97, May 2005

[8] Microsoft Corp. Virtual PC. URL: www.microsoft.com/windows/virtualpc/default.mspx

[9] Microsoft Corp. "How to use QueryPerformanceCounter to Time Code." URL: support.microsoft.com/kb/q172338

[10] Microsoft Corp. MSDN. URL: msdn.microsoft.com

[11] Sapuntzakis, C.; Brumley, D.; Chandra, R.; Zeldovich, N.; et al., "Virtual Appliances for Deploying and Maintaining Software", *Proceedings of the 17th Large Installation Systems Administration Conference (LISA)*, October 2003

[12] VMware Inc., URL: www.vmware.com

[13] Efrem G. Mallach, "On the relationship between virtual machines and emulators", *Workshop on Virtual Computer Systems*, March 1973.