Analysis, Anti-Analysis,
Anti-Anti-Analysis: An Overview of the Evasive Malware Scenario

Marcus Botacin[1], Vitor Falcão da Rocha[1], Paulo Lício de Geus[1], André Grégio[2]

[1]Instituto de Computação (IC)
Universidade Estadual de Campinas (Unicamp)
Campinas – SP – Brasil

[2]Departamento de Informática (DInf)
Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

Abstract. *Malicious programs are persistent threats to computer systems, and their damages extend from financial losses to critical infrastructure attacks. Malware analysis aims to provide useful information to be used for forensic procedures and countermeasures development. To thwart that, attackers make use of anti-analysis techniques that prevent or difficult their malware from being analyzed. These techniques rely on instruction side-effects and that system's structure checks are inspection-aware. Thus, detecting evasion attempts is an important step of any successful investigative procedure. In this paper, we present a broad overview of what anti-analysis techniques are being used in malware and how they work, as well as their detection counterparts, i.e., the anti-anti-analysis techniques that may be used by forensic investigators to defeat evasive malware. We also evaluated over one hundred thousand samples in the search of the presence of anti-analysis technique and summarized the obtained information to present an evasion-aware malware threat scenario.*

1. Introduction

Malicious software, also known as malware, is a piece of software with malicious purposes. Malware actions can vary from data exfiltration to persistent monitoring, causing damages to both private and public institution, either on image or financial aspects. According to CERT statistics [Cert.br 2015], malware samples may account by more than 50% of total reported incidents.

Given this scenario, analysts are required to analyze malicious samples in order to provide either defensive procedures or mechanisms to prevent/mitigate the infection, as well as to perform forensic procedures on already compromised systems. The set of techniques used for such kind of inspection is known as malware analysis. Analysis procedures can be classified into static, where there is no need to run the code, and dynamic, where code runs on controlled environment [Sikorski and Honig 2012]. The scope of this work is limited to static procedures, since they are the first line of detection against evasive malware.

Considering malware analysis capabilites and peculiarities, criminals started to protect their artifacts from being analyzed, equipping them with so-called anti-analysis (or anti-forensics) techniques. This way, their infection could last longer since they could make their samples stealth. Recent studies, such as [Branco et al. 2012], present scenarios in which 50% of samples contain at least one anti-analysis technique, and this number has been growing constantly.

In order to keep systems protected from such new armored threats, we need to understand how these anti-analysis techniques work so as to develop ways to effectively detect evasive samples before they can act. This is called anti-anti-analysis. In this paper, we present the *modus operandi* behind such kind of techniques, as well as possible detection methods in details. We evaluated the developed solution against over a hundred thousand samples, benign and malicious, which allowed us to build an evasive scenario panorama. We also compared evasive techniques used on different contexts (distinct countries), which can help analysts to be ahead of the next coming threats.

This work is organized as follows: Section 2 introduces basic concepts related to anti-analysis techniques and discusses related work and tools aimed at detecting anti-analysis techniques; Section 3 describes an study of how distinct evasion techniques work, and presents our detection framework; Section 4 shows the results obtained from applying our solution to distinct datasets; finally, Section 5 presents concluding remarks and future work.

2. Background and Related Work

In this section, we present the concepts related to anti-analysis and their detection counterparts as well as introduce the current state-of-the-art solutions.

2.1. Anti-analysis

The main idea of anti-analysis techniques is to raise the bar of counteraction methods. It can be done in many ways, e.g., leveraging theorethical hard-to-compute constructions. In this Section, we provide an overview of such anti-analysis techniques.

One common approach is to fingerprint the analysis environment. Known analysis solutions expose regular patterns, such as fixed IP addresses, host names, and serial numbers. Evasive samples can detect those patterns and suspend their execution [Yokoyama et al. 2016]. This type of approach was succesfully used against Cuckoo [Ferrand 2015] and Ether [Pék et al. 2011] sandboxes.

Another approach is to evade analysis by detecting execution side effects of virtual machines and emulators, which has been the most used enviroment for malware analysis. Those systems may exhibit a differing behavior when compared to their bare-metal counterparts, such as instructions not being atomic [Willems et al. 2012]. Currently, there are automated ways of detecting these side effects [Paleari et al. 2009]. Virtual Machines can also be detected by the changes that hypervisors perform on system internals (e.g., table relocations). Many tables, such as the Interrupt Descriptor Table (IDT), have their addresses changed on VMs when compared to bare-metal systems. These addresses can then be used as an indicator of a virtualized environment [Ferrie 2007].

There also approaches based not on evading the analysis itself, but on hardening the post-infection reverse engineering procedure. One notable technique is the anti-disassembly, a way of coding where junk data is inserted among legitimate code to fool the disassembler tool. Another variation of anti-disassembly techniques is to use opaque constants [Kruegel et al. 2007], constructions that cannot be solved without runtime information. Static attempts to guess resulting values of these expressions tend to lead to the path explosion problem [Xiao et al. 2010].

Finally, there are samples that make use of time measurement for analysis detection, since any monitoring technique imposes significant overheads [Lindorfer et al. 2011]. Although some solutions try to mitigate this problem by faking time measures, either on system APIs [Singh 2014], or on the hardware timestamp counter [Hexacorn 2014], the problem is unsolvable in practice, since an advanced attacker can make use of an externel NTP server over encrypted connections.

A notable example of anti-analysis tool is `pafish` [Pafish 2012], which consists of a series of modules that implement many of mentioned detection techniques, such as virtual machines detection and environment fingerprints. The tool's intention is to be used as verifier for any attempt of transparent solution, as well as to allow for a better understanding of practical malware evasion techniques.

2.2. Anti-Anti-Analysis

As well as the anti-analysis techniques, the anti-anti-analysis ones may also be classified as static or dynamic approaches. Static approaches can be applied in the form of pattern matching detectors of known anti-analysis constructions, for instance, address verifications and locations. However, due to its known limitations, some constructions can only be solved during runtime, which is accomplished when they run inside dynamic environments.

Dynamic solutions, in a general way, are based on faking answers for known anti-analysis checks, such as in COBRA [Vasudevan and Yerraballi 2006]. These approaches, however, turn into an arms-race, since new anti-analysis techniques are often released and these systems need to be updated. To minimze the impact of this issue, transparent analysis systems have been proposed, such as Ether [Dinaburg et al. 2008] and MAVMM [Nguyen et al. 2009]. These systems, however, impose high overheads and development costs.

In the following sections, we review the anti-anti-analysis techniques for the above presented anti-analysis classes and present static detectors for these techniques. We left dynamic detectors for future work, since they are not part of this work's scope.

2.3. State-of-the-art of anti-anti-analysis

Our work is related to many detection solutions. Two noticeable ones are`pyew` [Pyew 2012] and `peframe` [Peframe 2014], which aim to detect the evasive technique itself, and not whether a tool/system/environment may be evaded or not. They work by statically looking for known shellcodes and library imports related to analysis evasion. In this work, we have expanded theses detectors in order to provider a broader coverage.

In addition to the aforementioned tools, our work relates to the one presented by [Branco et al. 2012], which implemented several anti-anti-analysis detectors and analyzed evasive samples. In this work, we have implemented both the anti-analysis techniques as well as the presented static detectors, applying them against our distinct datasets, and enriching their analysis with the discussion of the working flow of the mentioned techniques. We also proceed in the same way regarding the work by [Ferrie 2008].

At the time we were writing this article, we have noticed a related work implementing similar techniques [Oleg 2016]. Such work, however, is limited to implementation issues whereas we present a comprehensive discussion and results evaluation.

Other related approaches, although more complex, are those which rely on using intermediate representations (IR) [Smith et al. 2014] or interleaving instructions [Saleh et al. 2014], cases not covered by this work. This work also does not cover obfuscation techniques based on encryption. This issue was addressed by other work, such as [Calvet et al. 2012].

3. Anti-Analysis Techniques and Detection

In this section, we summarize the anti-analysis techniques, their operation, and how they can be detected. The techniques were originally described in the previously presented works [Branco et al. 2012, Ferrie 2008, Pyew 2012, Peframe 2014, Pafish 2012, Oleg 2016] and are here classified according to their purpose: anti-disassembly, anti-debugging, and virtual machine detection. The complete discussion of each trick is presented on the appendix[1], due to space constraints.

3.1. Anti-disassembly

To understand how disassembly can turn into a hard task, we first introduce how current disassemblers work. After that, we present known tricks to detect evasion.

In general, disassemblers can be classified into `linear sweep` and `recursive traversal` approaches [Schwarz et al. 2002]. In the former, the disassembly process starts at the first byte of a given section and proceeds sequentially. The major limitation of this approach is that any data embedded in the code is interpreted as an instruction, leading to a wrong final disassembled code.

The latter approach takes into account the control flow of the program being disassembled, following the possible paths from the entry point, which solves part of problems presented by the linear approach, such as identifying `jmp`-preceeded data as code. The major assumption of this approach is that it is possible to identify all sucessors of a given branch, which is not always true, since any fail on identifying the instruction size can lead to incorrect paths and instructions.

3.1.1. Tricks

Table 1 shows a summary of anti-disassembly techniques and their detection methods[2].

Table 1. Anti-disassembly techniques and their detection methods.

Technique	Description	Detection
PUSH POP MATH	`PUSH` and `POP` a value on/from the stack instead of using a direct `MOV`	Detect a sequence of `PUSH` and `POP` on/from a register.
PUSH RET	`PUSH` a value on the stack and `RET` to it instead of the ordinary return.	Detect a sequence of `PUSH` and `RET`
LDR address resolving	Get loaded library directly from the PEB instead of using a function call	Check memory access referring the PEB offset.
Stealth API import	Manually resolving library imports instead of directly importing them.	Check for a sequence of access/compares of PEBs offsets.
NOP sequence	Breaks pattern matching by implanting NO-OPerations	Detect a sequence of `NOPs` within a given window
Fake Conditional	Create an always-taken branch	Check for branch-succeded instructions which set branch flags
Control Flow	Changing control flow within an instruction block	Check for the `PUSH-RET` instruction sequence
Garbage Bytes	Hide data as instruction code	Check for branch-preceeded data

[1] https://github.com/marcusbotacin/Anti.Analysis/tree/master/Whitepaper

[2] Described by Branco et al. 2012

3.2. Anti-Debug

In order to understand how anti-debug techniques work, we firstly introduce the basic idea of most tricks: using direct memory checks instead of function calls. Secondly, we present the tricks themselves.

3.2.1. Known API x Direct call

Most O.S. provide support for debugging checks. Windows, for instance, provides the `IsDebuggerPresent` API [Microsoft 2016]. Most anti-debug tricks, however, do not rely on these APIs, but perform direct calls instead. The main reason behind such decision is that APIs can be easily hooked by analysts, thus faking their responses. Internal structures, in turn, such as the process environment block (PEB) [Microsoft 2017c], are much harder to fake — some changes can even break system parts.

3.2.2. Tricks

Table 2 presents a summary of anti-debug techniques and their detection counterparts[3][4][5].

Table 2. Anti-debug techniques and their detection methods.

Technique	Description	Detection
Known Debug API	Call a debug-check API	Check for API imports
Debugger Fingerprint	Check the presence of known debugger strings	CHeck known strings inside the binary
NtGlobalFlag	Check for flags inside the PEB structure	Check for access on the PEB offset
IsDebuggerPresent	Check the debugger flag on the PEB structure	Check access to PEB on the debugger flag offset
Hook Detection	Verify whether a function entry point is a `JMP` instruction	Check for a `CMP` instruction having `JMP` opcode as an argument
Heap Flags	Check for heap flags on the PEB	check for heap checks involving PEB offsets
Hardware Breakpoint	Check whether hardware breakpoint registers are not empty	Check for access involving the debugger context
SS Register	Insert a check when interruptions are disabled	Check for SS register's `POPs`
Software Breakpoint	Check for the `INT3` instruction	Check for `CMP` with `INT3`
SizeOfImage	Change code image field	Check for PEB changes.

3.3. Anti-VM

A summary of anti-analysis tricks used by attackers to identify and evade virtualized environments is shown in Table 3[6][7].

[3] API check implemented by Pyew and Peframe [4] SizeOfImage implemented by Ferrie 2008 [5] Other techniques implemented by Branco et al. 2012 [6] VM fingerprint implemented by Pafish [7] Other techniques by Branco et al. 2012

Table 3. Anti-vm techniques and their detection methods.

Technique	Description	Detection
VM Fingerprint	Check for known strings, such as serial numbers	Check for known strings inside the binary
CPUID Check	Check CPU vendor	Check for known CPU vendor strings
Invalid Opcodes	Launch hypervisor-specific instructions	Check for specific instrutions on the binary
System Table Checks	Compare IDT values	Look for checks involving IDT
HyperCall Detection	Platform specific feature	Look for specific instructions

3.4. Detection Framework

Given the presented detection mechanisms, we have implemented them by using a series of Python scripts[8]. They work by iterating over `libopcodes`-disassembled instructions, and performing a pattern matching on these, according the trick we are looking for. Our pipeline is able to provide the information whether a given technique was found on a binary or not, the number of occurrences per binary, and the section the trick was found.

Unlike Branco et al. 2012 approach, which considered the `RET` instruction as a code block delimiter, we have implemented a variable-size window delimiter to evaluate if the tricks may have been implemented by making use of multi-block constructions.

4. Results

In this section, we present the results of applying our set of detectors to distinct datasets and discuss how anti-analysis tricks have been applied in practice.

4.1. Binary sections

Here we show the binary section influence on the trick detection. In order to perform this evaluation, we considered a dataset of 70 thousand worldwide crawled samples.

Figure 1 shows the detection distribution along the binary sections. It is worth to notice that the usual instruction section (`text`) is only the 5^{th} more prevalent section. The presence of other sections can be due to samples moving their tricks to dinstinct sections in order to not be detected by anti-virus (AV). This fact can only be exactly determined through dynamic analysis. The presence of some section such as `.aspack`, for instance, is due to the presence of a packer to obfuscate the code.

Figure 2 shows that the tricks contained in the `.text` section correspond to half of the total tricks detected. The most prevalent techniques, such as `PushPop` and `PushRet`, are the most simple.

4.2. Packer influence

In the last section, we could see that sections related to packer obfuscation were identifyed. In this section, we discuss the packer influence on trick detection. The first noticeable situation is that the tricks detected on packed samples are not equally distributed among sections, as shown in Figure 3. We can observe that the `C++` compiler and the `PIMP` packer exhibit tricks on the `.rsrc` section, whereas the `UPX` packer presents tricks on distinct sections. A similar situation

[8] `https://github.com/marcusbotacin/Anti.Analysis`

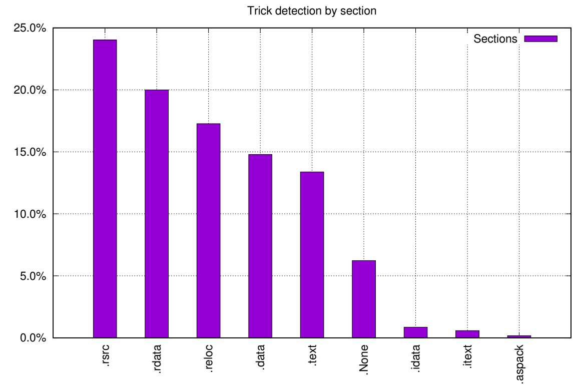


Figure 1. Tricks by section.

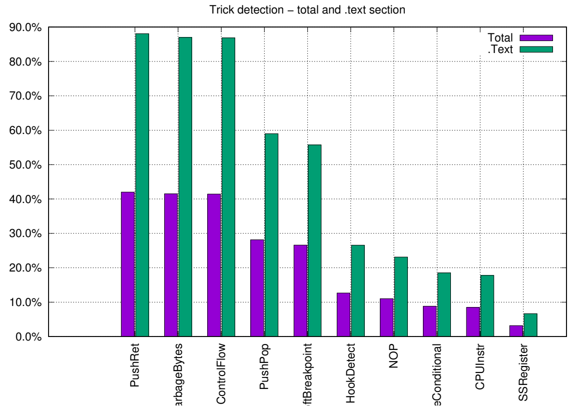


Figure 2. Tricks - total and `.text` section.

happens when considering the detected tricks, as shown in Figure 4. The `C++` compiler and the `PIMP` packer presents similar rates of tricks while the `UPX` packer presents distinct tricks. Finally, in order to evaluate the packer influence on trick detection, we unpacked all samples for which there are known unpackers (6 thousand samples), thus obtaining the results shown in Figure 5. We could confirm our expectations that the majority of the tricks are present on the packer, not on the original code. This fact is mostly due to the usage of malware kit generators.

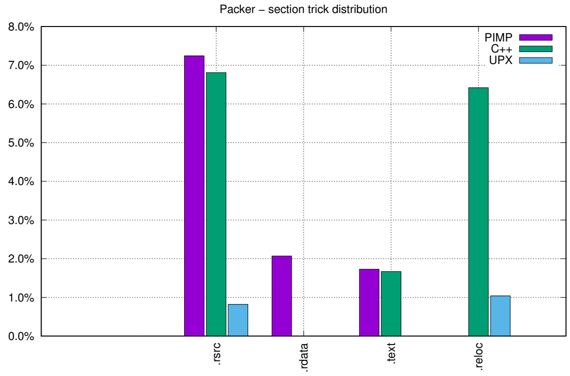


Figure 3. Packer distribution across binary sections.

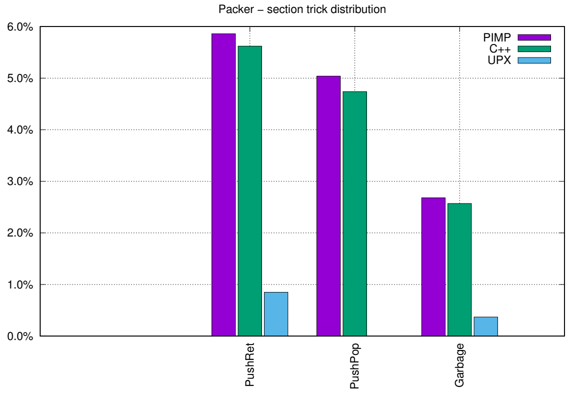


Figure 4. Tricks detected on distinct packers.

4.3. Malware and Goodware

Some of the presented tricks are widely used, so they can be found either on benign programs (goodware) and malicious ones. To verify if the detection of those aforementioned tricks could be used as a malicious program indicator, we compared the trick incident on both program classes, as shown in Figure 6. We performed our tests using as a benign dataset the binaries and DLLs from a clean Windows installation (binaries from the `System32` directory). We can observe that some general tricks (CPU identification) can also be found on system DLLs, but these are not present on the binaries. This fact is explained by the Windows architecture, which relies on DLLs for userland-kernel communication. This indicates that we need to employ distinct appraches when developing heuristics for executables and DLLs. We aim to extend this evaluation for general binaries, despite system ones. However, it is hard to ensure internet-downloaded binaries are not trojanized in any way, thus biasing the results.

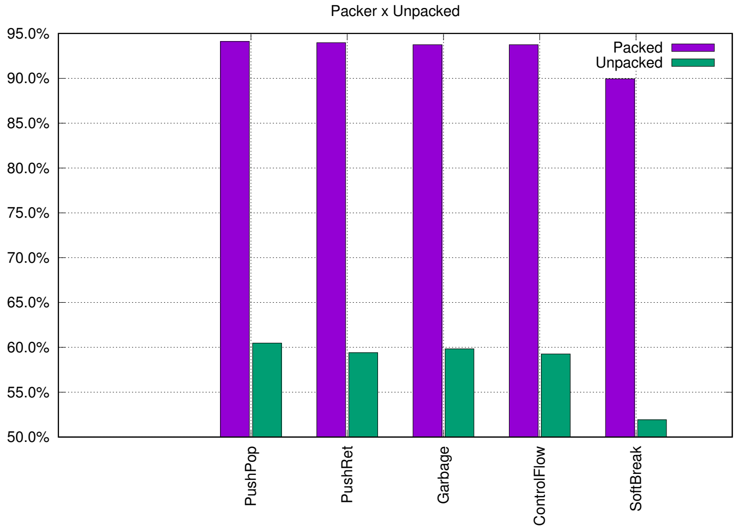


Figure 5. Packer influence on trick detection.

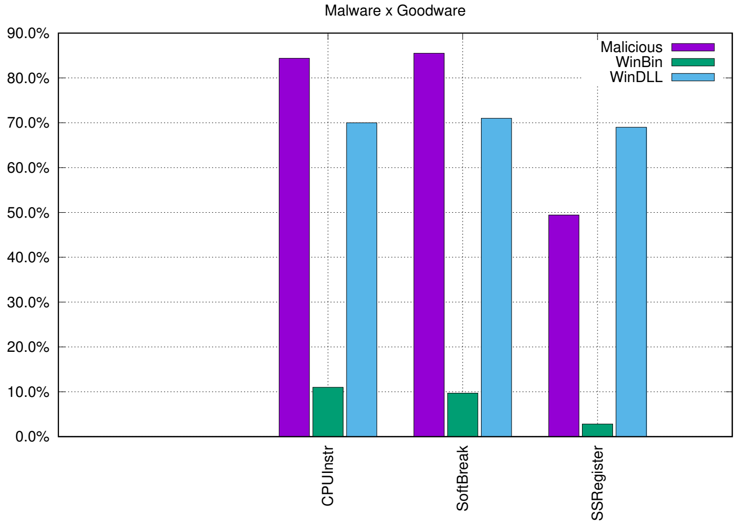


Figure 6. Tricks detection on malware and goodware.

4.4. Distinct Scenarios

The tricks prevalence differs across distinct datasets. In order to provide a view on how these differences affect user in practice, we compared the worldwide crawled dataset[9] to a dataset of 30 thousand brazillian collected samples[10]. Figure 7 shows the results of comparing the datasets using the PEframe tool. We can observe that the brazillian dataset presented higher detection rates for the `VmCheck` and the `VirtualBox` tricks and lower for the others. These rates are quite surprisingly, given the previous research results regarding the brazillian scenario. When performing the same checks using our developed tricks, as shown on Figure 8, we show that the brazillian scenario presents lower trick rates than the worldwide one. This differences can be explained by the fact that the knowledge behind the tricks detected by the PEframe are more spreaded, since they are easier. More advanced tricks, such as some of those we have presented in this work, are only

[9] From `http://malshare.com/` [10] The same as in Botacin et al. 2015

present on a broader scenario, i.e., the worldwide dataset.

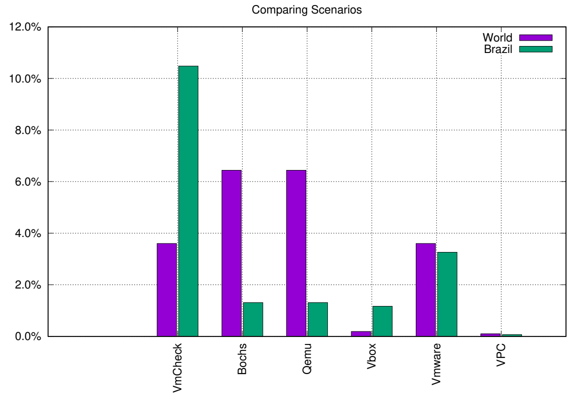


Figure 7. Comparing scenarios: PEframe detection.

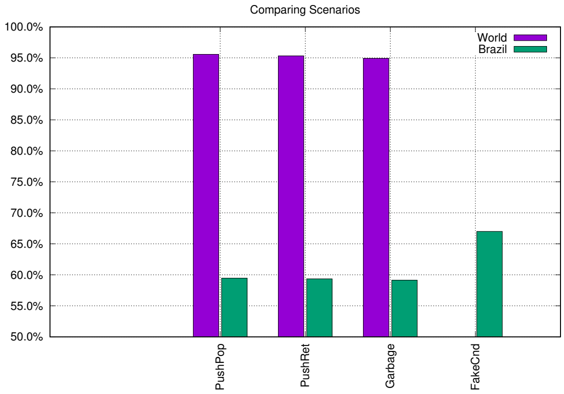


Figure 8. Comparing scenarios: Tricks detection.

The presented results are in agreement with previous research results regarding the brazilian scenario [Botacin et al. 2015].

4.5. Improving tricks and their detection

In this section, we present ways the tricks can be enhanced and how to detect them.

4.5.1. Trick splitting

A way of evading the trick detection is to split it across distinct blocks. Although we cannot check such usage in practice without dynamic analysis, we can look for signs of spllited-tricks by changing the detection window, as show on Figure 9. The initial value is the `RET` window, on which we traverse the block until the instruction is found. We considered the detection rate of this window as a groundtruth, thus presenting the `100%` detected value. The other values are fixed-size number of instructions which will be traversed, thus increasing the detection rate. We observed a maximum increase of `0.65%`.

4.5.2. Instruction disalignment

Another possible way of evading tricks detection is by using unaligned instructions, so the disassembler is not able to present the correct opcode. Although we could only check the effective usage of such approach on a dynamic system, we can look for static signs of such usage. In order to do so, we have implemented some detectors using YARA[11] rules and running them on the binary bytes. The tests results are shown in the Table 4. We have considered 300 random samples, being the `Aligned` considered as groundtruth. We can observe the `Unaligned` results are significantly higher, indicating it is a viable way of hidding code.

[11] `https://virustotal.github.io/yara/`

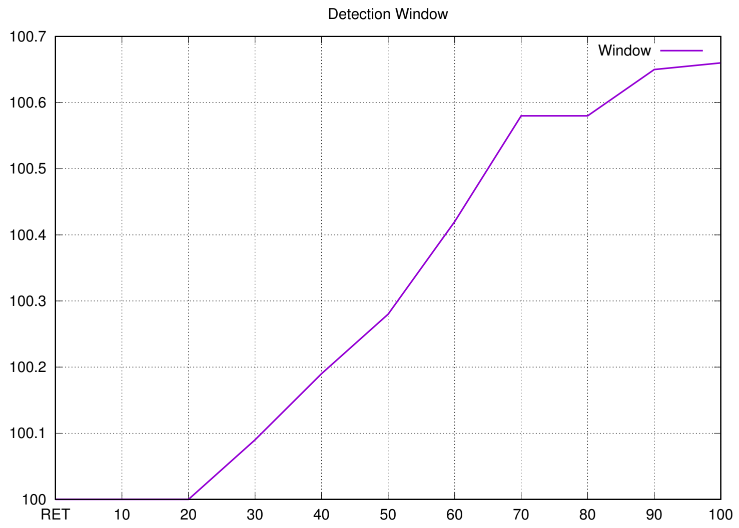


Figure 9. Evaluating block window effect on trick detection.

Table 4. Evaluating the occurrence of misaligned tricks.

Trick	Aligned	Unaligned
CPU	182	287
FakeJMP	63	203

4.5.3. Compiler-based tricks

Another way of hidding the trick is to compile the code using instructions unsupported by AVs and other tools or indirect constructions. The *ROP itself malware* [Poulios et al. 2015], for instance, suggested turning a malware sample into a ROP[12] payload, approach which was implemented by the *Ropinjector* tool [Poulios 2015]. The *SSexy tool* [Bremer 2012] compiles the code using `SSE`[13] instructions. The `Movfuscator` [domas 2015] does the same using `XOR` ones. Finally, the work [Barngert 2013] compiles a code to run using only `MMU` instructions[14]. In order to verify that in practice, we submited some known shellcodes from ExploitDB compiled using the ROPInjector solution, being the results reported in the Table 5. We can notice that the AV were not able to detect the payloads when compiled using the tool.

Table 5. Compilation-based evasion.

ShellCode	1[15]	2[16]	3[17]	4[18]	5[19]
Unarmored	4/57	15/58	9/57	7/68	9/53
ROPInjector	0/57	0/57	0/54	0/54	0/53

4.6. General AV detection

The results from the previous section suggests that AV are not able to handle some tricks. Problems on AV emulators were also described on other work [Nasi 2014]. We submitted to virustotal some

[12] Return Oriented Programming [13] Streaming SIMD Extensions [14] Memory Management Unit

shellcodes armored with our tricks, being the results shown in the Table 6. We can notice that the AVs do not presented the same efficiency to handle armored and unarmored tricks.

Table 6. Evaulating AV Evasion: unarmored and trick-armored samples.

Shellcode	SC1		SC2		SC3	
Technique	W/o Trick	W/ Trick	W/o Trick	W/ Trick	W/o Trick	W/ Trick
Fakejmp	10/58	6/57	20/58	17/58	15/58	10/57
PushRet		7/57		17/58		10/58
NOP		6/57		17/57		10/58

4.7. Discussion

We have presented anti-analysis tricks and ways of detecting them. We also presented some insights on how they can be enhanced as well their detector. The major limitation relies on the fact that the efectiveness of such approach can only be measured using dynamic analysis, which is a straightforward future work. Additionaly, we aim to implement some kind of rule-based remmediation [Lee et al. 2013]. More details about this solution's limitations are presented in the appendix.

5. Conclusion

In this work, we have studied anti-analysis techniques, their effect on malware analysis, and theorethical limitations. We also developed static detectors able to identify known evasive constructions on binaries. We have tested these detectors against multiple datasets and observed that there are significant differences between the Brazilian scenario compared to the global one.

The list of tricks presented in this paper is not exhaustive, since attackers keep testing, and consequently developing new ways of evading analysis and detecting environments. Hence, our future work consists on implementing new detectors, as they have been discovered, as well as evaluating distinct datasets, in order to identify other trends about malware.

Acknowledgements

This work was supported by the Brazilian National Counsel of Technological and Scientific Development (CNPq, Universal 14/2014, process 444487/2014-0) and the Coordination for the Improvement of Higher Education Personnel (CAPES, Project FORTE, Forensics Sciences Program 24/2014, process 23038.007604/2014-69).

References

Bachaalany, E. (2005). Detect if your program is running inside a virtual machine. https://www.codeproject.com/articles/9823/detect-if-your-program-is-running-inside-a-virtual.

Barngert, J. (2013). Trapcc. https://github.com/jbangert/trapcc.

Botacin, M., Grégio, A., and de Geus, P. (2015). Uma visão geral do malware ativo no espaço nacional da internet entre 2012 e 2015. *Anais do XV SBSEG*.

Branco, R. R., Barbosa, G. N., and Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti- vm technologies. https://github.com/rrbranco/blackhat2012/blob/master/blackhat2012-paper.pdf.

Bremer, J. (2012). Ssexy. `https://github.com/jbremer/ssexy`.

Calvet, J., Fernandez, J. M., and Marion, J.-Y. (2012). Aligot: Cryptographic function identification in obfuscated binary programs. In *Proc. of the 2012 ACM Conf. on Computer and Communications Security*, CCS '12, pages 169–182, New York, NY, USA. ACM.

Cert.br (2015). Estatísticas dos incidentes reportados ao cert.br. `https://www.cert.br/stats/incidentes/`. Access Date: May/2017.

Dinaburg, A., Royal, P., Sharif, M., and Lee, W. (2008). Ether: Malware analysis via hardware virtualization extensions. In *Proc. of the 15th ACM Conf. on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA. ACM.

domas (2015). Movfuscator. `https://github.com/xoreaxeaxeax/movfuscator`.

Eliam, E. (2005). *Reverse: Secrets of Reverse Engineering*. Willey.

Ferrand, O. (2015). How to detect the cuckoo sandbox and to strengthen it? *Journal of Computer Virology and Hacking Techniques*, 11(1):51–58.

Ferrie, P. (2007). Attacks on virtual machine emulators. `https://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf`.

Ferrie, P. (2008). Anti-unpacker tricks. `http://pferrie.tripod.com/papers/unpackers.pdf`.

hexacorn (2014). Protecting vmware from cpuid hypervisor detection. `http://www.hexacorn.com/blog/2014/08/25/protecting-vmware-from-cpuid-hypervisor-detection/`.

Hexacorn (2014). Rdtscp - a recooked antire trick. `http://www.hexacorn.com/blog/2014/06/15/rdtscp-a-recooked-antire-trick/`.

Kruegel, C., Kirda, E., and Moser, A. (2007). Limits of Static Analysis for Malware Detection. In *Proc. of the 23rd Annual Computer Security Applications Conference (ACSAC) 2007.*

Laboratory, B. (2012). Detecting vmware. `https://brundlelab.wordpress.com/2012/10/21/detecting-vmware/`.

Lee, J., Kang, B., and Im, E. G. (2013). Rule-based anti-anti-debugging system. In *Proc. of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, pages 353–354, New York, NY, USA. ACM.

Lindorfer, M., Kolbitsch, C., and Milani Comparetti, P. (2011). Detecting environment-sensitive malware. In *Proc. of the 14th International Conf. on Recent Advances in Intrusion Detection*, RAID'11, pages 338–357, Berlin, Heidelberg. Springer-Verlag.

Liu, Q. (2012). Just another malware analyzer. `https://github.com/lqhl/just-another-malware-analyzer`.

Lyashko, A. (2011). Stealth import of windows api. `http://syprog.blogspot.com.br/2011/10/stealth-import-of-windows-api.html`.

Microsoft (2016). Isdebuggerpresent. `https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680345%28v=vs.85%29.aspx`.

Microsoft (2017a). Download detours express. `https://www.microsoft.com/en-us/download/details.aspx?id=52586`.

Microsoft (2017b). Getprocessheap function. `https://msdn.microsoft.com/pt-br/library/windows/desktop/aa366569%28v=vs.85%29.aspx`.

Microsoft (2017c). Peb structure. `https://msdn.microsoft.com/pt-br/library/windows/desktop/aa813706%28v=vs.85%29.aspx`.

Microsoft (2017d). Peb_ldr_data structure. https://msdn.microsoft.com/pt-br/library/windows/desktop/aa813708%28v=vs.85%29.aspx.

Microsoft (2017e). Setwindowshookex function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms644990%28v=vs.85%29.aspx.

Microsoft (2017f). Virtualquery function. https://msdn.microsoft.com/pt-br/library/windows/desktop/aa366902%28v=vs.85%29.aspx.

MNIN.org (2006). The torpig/anserin/sinowal family of trojans detect virtual machine information and abort infection when present. https://www.mnin.org/write/2006_torpigsigs.pdf.

Nasi, E. (2014). Bypass antivirus dynamic analysis. http://packetstorm.foofus.com/papers/virus/BypassAVDynamics.pdf.

Nguyen, A. M., Schear, N., Jung, H., Godiyal, A., King, S. T., and Nguyen, H. D. (2009). Mavmm: Lightweight and purpose built vmm for malware analysis. In *Computer Security Applications Conf., 2009. ACSAC '09. Annual*, pages 441–450.

Oleg, K. (2016). Anti-debug protection techniques: Implementation and neutralization. https://www.codeproject.com/articles/1090943/anti-debug-protection-techniques-implementation-an.

Pafish (2012). Pafish. https://github.com/a0rtega/pafish.

Paleari, R., Martignoni, L., Roglia, G. F., and Bruschi, D. (2009). A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proc. of the 3rd USENIX Conf. on Offensive Technologies*, WOOT'09, pages 2–2, Berkeley, CA, USA. USENIX Association.

Peframe (2014). Peframe. https://github.com/guelfoweb/peframe.

Pék, G., Bencsáth, B., and Buttyán, L. (2011). nether: In-guest detection of out-of-the-guest malware analyzers. In *Proc. of the Fourth European Workshop on System Security*, EUROSEC '11, pages 3:1–3:6, New York, NY, USA. ACM.

Poulios, G. (2015). Ropinjector. https://github.com/gpoulios/ROPInjector.

Poulios, G., Ntantogian, C., and Xenakis, C. (2015). Ropinjector: Using return oriented programming for polymorphism and antivirus evasion. https://www.blackhat.com/docs/us-15/materials/us-15-Xenakis-ROPInjector-Using-Return-
pdf.

Pyew (2012). Pyew. https://github.com/joxeankoret/pyew.

Radare (2008). Radare. https://github.com/radare/radare/blob/master/src/kradare/dtdumper/dtdumper.c. Access Date: May/2017.

Saleh, M., Ratazzi, E. P., and Xu, S. (2014). Instructions-based detection of sophisticated obfuscation and packing. In *2014 IEEE Military Communications Conf.*, pages 1–6.

Schwarz, B., Debray, S., and Andrews, G. (2002). Disassembly of executable code revisited. In *Proc. of the Ninth Working Conf. on Reverse Engineering (WCRE'02)*, WCRE '02, pages 45–, Washington, DC, USA. IEEE Computer Society.

Securiteam (2004). Red pill... or how to detect vmm using (almost) one cpu instruction. http://www.securiteam.com/securityreviews/6Z00H20BQS.html.

Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition.

Singh, A. (2014). Not so fast my friend - using inverted timing attacks to bypass dynamic analysis. http://labs.lastline.com/not-so-fast-my-friend-using-inverted-timing-attacks-to-bypass-dynamic-analysis.

Smith, A. J., Mills, R. F., Bryant, A. R., Peterson, G. L., and Grimaila, M. R. (2014). Redir: Automated static detection of obfuscated anti-debugging techniques. In *2014 International Conf. on Collaboration Technologies and Systems (CTS)*, pages 173–180.

Vasudevan, A. and Yerraballi, R. (2006). Cobra: fine-grained malware analysis using stealth localized-executions. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 15 pp.–279.

VMWare (2010). Mechanisms to determine if software is running in a vmware virtual machine (1009458). `https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1009458`.

Willems, C., Hund, R., Fobian, A., Felsch, D., Holz, T., and Vasudevan, A. (2012). Down to the bare metal: Using processor features for binary analysis. In *Proc. of the 28th Annual Computer Security Applications Conf.*, ACSAC '12, pages 189–198, New York, NY, USA. ACM.

Xiao, X., s. Zhang, X., and d. Li, X. (2010). New approach to path explosion problem of symbolic execution. In *Pervasive Computing Signal Processing and Applications (PCSPA), 2010 First International Conf. on*, pages 301–304.

Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., and Rossow, C. (2016). *SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion*, pages 165–187. Springer International Publishing, Cham.

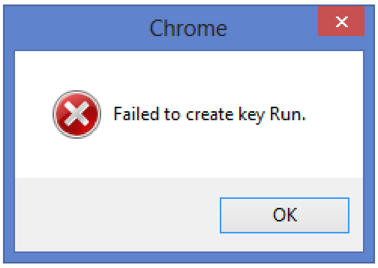
Appendix to the “Analysis, Anti-Analysis,
Anti-Anti-Analysis: An Overview of the Evasive Malware Scenario” paper, published in the proceedings of XVII SBSEG

Marcus Botacin, Vitor Falcão da Rocha, Paulo Lício de Geus, André Grégio

We added to this appendix additional information to support our findings on anti-analysis-armored malware.

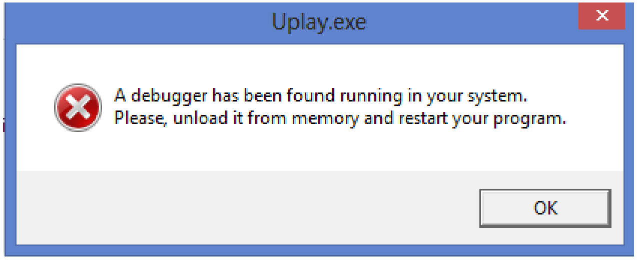
A. Real Malware Evasion

Some samples stealthly quit their operations when an anti-analysis trick is successful on identifying an analysis environment. However, some of them opt to display some information. We present below (Figure 10, 11, and 12) some examples we found when running real samples we have collected on the wild.



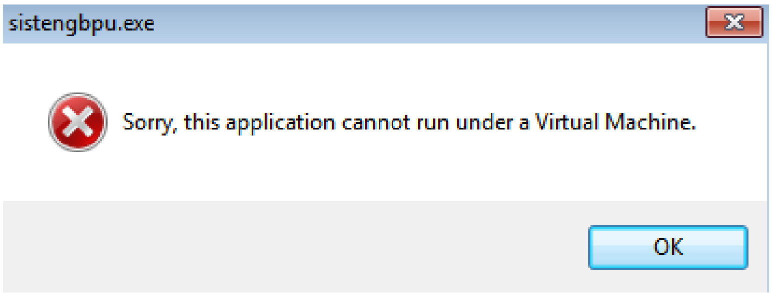
Failed to create key Run.

Figure 10. Real malware claiming a registry problem when an anti-analysis trick succeded.



A debugger has been found running in your system.
Please, unload it from memory and restart your program.

Figure 11. Commercial solution armored with anti-debug technique.



Sorry, this application cannot run under a Virtual Machine.

Figure 12. Real malware impersonating a secure solution which cannon run under an hypervisor.

B. Techniques

This section details how the techniques and their detectors are implemented.

B.1. Anti-disassembly

How disassembly, anti-disassembly, and anti-disassembly detectors are implemented.

B.1.1. Disassembly techniques used by distinct solutions

As presented on Section B.1.1, disassembly can be performed on two ways: `Linear Sweep` or `Recursive Traversal`. The disassembler operation mode affects the obtained results. In order to make the scenario clearer, we summarized, in the Table 7, the disassembly approach used by distinct tools.

Table 7. Disassembly technique used by distinct solutions (Based on [Eliam 2005]).

Tool	Technique
OllyDbg	Recursive traversal
NuMega	SoftICE Linear sweep
Microsoft WinDbg	Linear sweep
IDA Pro	Recursive traversal
PEBrowse	Recursive traversal
Objdump	Linear Sweep

B.1.2. Tricks

This section show how the anti-disassembly tricks and their detectors are implemented.

B.1.3. PUSH POP MATH

In order to obfuscate a value, samples can make use of indirect values manipulation, such as using `PUSH` and `POP` instruction. On this technique, a known immediate is pushed into the stack and then pop-ed to a register. This register is then used on further computations, as shown on Listing 1.

Listing 1. PUSH POP MATH trick.

```
        push    0x1234
        pop     rax
        xor     rax, 0xFFFF
```

In order to detect this technique, we can look for the sequence of a `PUSH`, `POP` to a register, and a computation over such register, as shown on Listing 2.

Listing 2. PUSH POP MATH trick detection.

```
 if 'push' in instruction and not op1 in ['ax','bx','cx','dx']:
            self.found_push=True
        elif 'pop' in instruction and self.found_pop==False:
            self.found_pop=True
            self.found_op=op1
        elif self.found_pop==True and instruction in ['and', 'or','xor']:
            if self.found_op in op1 or self.found_op in op2:
                self.found_comp=True

        if self.found_comp==True:
            self.clear()
            print "\"PushPopMath\" Detected
               ! Section: <%s> Address: 0x%s" % (section, address)
```

B.1.4. PUSH-RET Instruction Replacement

Instruction replacement is a common evasive technique since it makes harder to follow the execution control flow. One of many variations of this technique is to replace the ordinary `CALL` instruction by a sequence of `PUSH` and `RET`, on which an address is inserted into the stack and thus the function returns to it. This sequence is shown on Listing 3.

Listing 3. PUSH RET trick.

```
        push    0x12345678
        ret
```

The presence of a sequence of `PUSH` and `RET` can be used to detect this trick usage, as shown on Listing 4.

Listing 4. PUSH RET trick detection.

```
        def check(self, section, address, instruction, op1, op2):

        if 'push' in instruction:
            self.found_push=True
        elif self.found_push==True and 'ret' in instruction:
            self.found_ret=True
        else:
            self.found_push=False

        if self.found_ret:
            self.clear()
            print "\"PushRet\" Detected
               ! Section: <%s> Address: 0x%s" % (section, address)
```

B.2. LDR Address resolving

The LDR is a PEB internal structure which contains information about loaded modules [Microsoft 2017d], as shown on Listing 5.

Listing 5. LDR struct definition.

```
typedef struct _PEB_LDR_DATA {
        BYTE        Reserved1[8];
        PVOID       Reserved2[3];
        LIST_ENTRY  InMemoryOrderModuleList;
      } PEB_LDR_DATA, *PPEB_LDR_DATA;

typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

When looking to the PEB structure, the LDR information can be found at the `0x0c` offset, as shown on Listing 6.

Listing 6. PEB's LDR entries.

```
typedef struct _PEB {
        BYTE                            Reserved1[2];
        BYTE                            BeingDebugged;
        BYTE                            Reserved2[1];
        PVOID                           Reserved3[2];
        PPEB_LDR_DATA                   Ldr;
```

This way, some samples could try to access this information directly, by loading the respective addresses, as shown on Listing 7.

Listing 7. Direct LDR handling.

```
        mov     eax, [fs:0x30]
        mov     eax, [eax+0x0c]
```

In order to detect this trick, we can check for the usage of the PEB and LDR offsets, respectively, as shown on Listing 8.

Listing 8. Detecting LDR direct access.

```
def check(self, section, address, instruction, op1, op2):

        if instruction in ['mov', 'movsx', 'movzx']:
            if 'fs:0x30' in op2:
                self.found_op1 = op1
                self.found_keyword = True
                return False

        if self.found_keyword:
            if instruction
                in ['cmp', 'cmpxchg', 'mov', 'movsx', 'movzx']:
                if '[' + self.found_op1 + '+0xc]'
                    in op1 or '[' + self.found_op1 + '+0xc]' in op2:
                    self.clear()
                    print "\"LDR\" Detected! Section
                       : <%s> Address: 0x%s" % (section, address)
```

B.2.1. Stealth Windows API Import

The Windows API is the basic toolchain for development tasks on this system, being provided by system libraries (DLLs) to be imported by the programs. However, such imports can reveal many information about a program behavior. Thus, directly using system API increases samples' stealthness.

One way to implement such stealth function imports is to rely on `ntdll` and `kernel32` libraries which are automatically mapped into processes, wven without any explicit import. Those libraries are accessible through a walk over the process memory [Lyashko 2011].

This trick starts by retrieving handlers through the SEH chain offset (`0x0`), as shown on Listing 9. Secondly, by iterating over it, until the end, to retrieve the handler at the `0x4` offset, as also shown on Listing 9.

Listing 9. SEH base address and library handler retrievals.

```
            mov eax,[fs:0]
            .search_default_handler:
            cmp dword[eax],0xFFFFFFFF
            jz .found_default_handler
            ;go to the previous handler
            mov eax,[eax]
            jmp .search_default_handler
            mov eax,[eax+4]
            and eax,0xFFFF0000
```

Given such memory location, it is required to look for the MZ signature over the pages, as shown on Listing 10.

Listing 10. Looking for the PE on memory pages.

```
.look_for_mz:
    cmp word [eax],'MZ'
    jz .got_mz
    sub eax,0x10000
    jmp .look_for_mz
```

At this point, a handle for an unknown library was retrieved. In order to identify it, the trick starts with a check of the `0x3c` offset, which contains an offset for a PE string signature followed by `COFF` data, as shown on Listing 11.

Listing 11. Finding PE signature and COFF data.

```
    mov bx,[eax+0x3C]
    movzx ebx,bx
    add eax,ebx
    mov bx,'PE'
    movzx ebx,bx
    cmp [eax],ebx
    jz .found_pe
```

Once the register holds the COFF header, the trick can find the exports of the `IMAGE_DATA_DIRECTORY` at offset `0x78` and thus read the `RVA` and add it to the base address, allowing the use, as shown on Listing 12.

Listing 12. Getting image data and handling its RVA.

```
    add eax,0x78
    mov eax,[eax]
    add eax,[image_base_address]
```

By accessing the `0xC` offset, it gets the `NAME RVA`, a string containing the library address, and so discovers whether it is `kernel32` or `ntdll`, as shown on Listing 13.

Listing 13. Accesing name rva.

```
        mov eax,[eax+0x0C]
        add eax,[image_base_address]
```

A detector can be implemented by checking whether all these steps appear in sequence on a given code. Our detector looks like the one presented on Listing 14.

Listing 14. Stealth import trick detector.

```
    def check(self, section, address, instruction, op1, op2):

        if self.found_seh==False and instruction in [’mov’, ’movsx’, ’movzx’]:
            if ’fs:0x0’ in op2:
                self.found_op1 = op1
                self.found_seh = True

        elif self.found_seh==True and self.found_handler==False and instruction in [’mov’, ’movsx’, ’movzx’]:

            if ’fs:0x30’ in op2:
                self.found_seh=False
            elif self.found_op1 + ’+0x4’ in op2:
                self.found_handler=True
                self.found_op2 = op1

        elif self.found_handler==True and instruction in [’cmp’,’cmpxchg’]:
            if self.found_op2 in op1:
                self.found_cmp=True

        elif self.found_cmp==True and instruction in [’mov’,’movsx’,’movzx’]:
            if self.found_op2+’+0x3c’ in op2:
                self.found_pe=True

        elif self.found_pe==True and instruction in [’and’,’or’,’xor’,’add’,’sub’,’cmp’]:
            if ’0x78’ in op1 or ’0x78’ in op2:
                self.found_img=True

        if self.found_img:
            self.clear()
            print "\"StealthImport\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

B.2.2. NOP

Another anti-disassembly technique is to add dead-code to the binary. Dead code are construction which are unreacheable or effectless, intended only to make the analysis procedures harder and to evade pattern matching detectors. A commmon dead code construction is a NOP sequence, as shown on Listing 15.

Listing 15. NOP sequence trick.

```
mov     eax, 0
nop
nop
nop
nop
nop
pop     rbp
```

A detector for this technique consists on finding a N-sized window ROP sequence, as shown on Listing 16.

Listing 16. NOP sequence trick detection.

```
def check(self, section, address, instruction, op1, op2):
        if instruction == 'nop':
            self.counter += 1
            if self.counter is 5:
                self.counter = 0
                print "\"NOPSequence\" Detected!
                   Section: <%s> Address: 0x%s" % (section, address)
```

B.2.3. Fake Conditional

Many solutions try to follow control flow in order to apply their detectors. Making conditional control flow harder is a powerful anti-analysis technique, since not all paths can be followed due to the path explosion problem.

One possible implementation for this evasive techique is to rely on flags computed by a previous known instruction. The example of Listing 17 shows a known-result instruction, since XOR-ing the registers will always result on zero, thus triggering the zero-conditioned jump.

Listing 17. Fake Conditional trick.

```
xor eax, eax
jnz main
```

Detectors for this technique rely on detecting XOR instructions followed by these kind of construction, such as JMP, STC or CLC, as shown on Listing 18.

Listing 18. Fake Conditional trick detector.

```
def check(self, section, address, instruction, op1, op2):

self.cycle_count += 1

if instruction == 'xor' and op1 == op2:
        self.found_xor = True
        self.xor_cycle = self.cycle_count
        return
elif instruction =='stc':
        self.found_stc= True
        self.stc_cycle=self.cycle_count
        return
elif instruction=='clc':
        self.found_clc=True
        self.clc_cycle=self.cycle_count
        return

if(instruction=='jnz' or instruction=='jne') and self.found_xor and self.cycle_count==self.xor_cycle+1:
        self.clear()
        print "\"FakeConditionalJumps\" Detected! Section: <%s> Address: 0x%s" % (section, % address)
elif (instruction=='jnc' or instruction== 'jae') and self.found_stc and self.cycle_count == self.stc_cycle+1:
        self.clear()
        print "\"FakeConditionalJumps\" Detected! Section: <%s> Address: 0x%s" % (section, % address)
elif (instruction =='jc' or instruction=='jb') and self.found_clc and self.cycle_count==self.clc_cycle+1:
        self.clear()
        print "\"FakeConditionalJumps\" Detected! Section: <%s> Address: 0x%s" % (section, % address)
```

B.2.4. Control Flow

An anti-analysis variation of the JMP construction is to replace the unconditional JMP by other constructions, which can fool a linear disassembler tool. A common replacement is to pushing a value in to stack and then launching a RET instruction. This construction can be seen on Listing 19.

Listing 19. Control Flow trick.

```
mov     eax, 0
        push 0x2
        ret
        pop     rbp
```

A detector for this technique is to match a sequence of PUSH and RET, as shown on Listing 20.

Listing 20. Control Flow trick detection.

```
    def check(self, section, address, instruction, op1, op2):

        self.cycle_counter += 1

        if instruction == 'push':
            self.found = True
            self.found_cycle = self.cycle_counter
            return

        if self.found and instruction
            == 'ret' and self.cycle_counter == self.found_cycle + 1:
            self.clear()
            print "\"ProgramControlFlowChange\" Detected
               ! Section: <%s> Address: 0x%s" % (section, address)
```

B.2.5. Garbage Bytes

A way to hide data inside binaries is to add the data right after an unconditional `JMP`, since it will be unreachable as code, as shown on Figure 13.

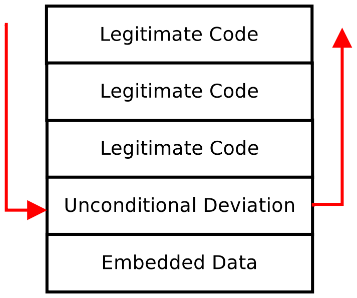


Figure 13. Binary data as a Dead Code.

The dead code insertion is intended to fool disassemblers which try to interpret such unreacheable bytes as code. Its usage is often associated with other control-flow-deviation-based anti-analysis techniques, such as `indirect jump`. An implementation of this technique can be seen on Listing 21.

Listing 21. Garbage Bytes trick.

```
mov     eax, 0
push 0x3
ret
.data
```

The detector for this technique is based od the previously presented `Control Flow` and `Fake Jump` detectors, followed by additional bytes, as shown on Listing 22.

Listing 22. Garbage Bytes tricks detection.

```
def check(self, section, address, instruction, op1, op2):

        self.cycle_counter += 1

        if instruction == 'push':
            self.found_push = True
            self.found_push_cycle = self.cycle_counter
        elif instruction == 'xor' and op1 == op2:
            self.found_xor = True
            self.found_xor_cycle = self.cycle_counter
        elif instruction == 'stc':
            self.found_stc = True
            self.found_stc_cycle = self.cycle_counter
        elif instruction == 'clc':
            self.found_clc = True
            self.found_clc_cycle = self.cycle_counter

        if self.found_push and instruction == 'ret' and self.cycle_counter == self.found_push_cycle + 1:
            self.clear()
            print "\"GarbageBytes\" Detected! Section: <%s> Address: 0x%s" % (section, address)
        elif self.found_xor and instruction == 'jnz' and self.cycle_counter == self.found_xor_cycle + 1:
            self.clear()
            print "\"GarbageBytes\" Detected! Section: <%s> Address: 0x%s" % (section, address)
        elif self.found_stc and (instruction == 'jnc' or instruction == 'jae') and self.cycle_counter == \
                        self.found_stc_cycle + 1:
            self.clear()
            print "\"GarbageBytes\" Detected! Section: <%s> Address: 0x%s" % (section, address)
        elif self.found_clc and (instruction == 'jc' or instruction == 'jb') and self.cycle_counter == \
                        self.found_clc_cycle + 1:
            self.clear()
            print "\"GarbageBytes\" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

B.3. Anti-debug

In this section, we present techniques aimed to detect if a sample is being debugged.

B.3.1. Known Debug APIs Usage

The most direct way to detect a debuggers presence is to rely on debug-related function provided by the O.S. API. On Windows O.S., for instance, many debug related APIs, such as `IsDebuggerPresent`, are available on its default libraries.

A straightforward countermeasure is to check the presence of such functions on binary imports section. This approach is implemented in tools like PEframe. Listing 23 shows an excerpt of PEFrame's code file used for pattern matching.

Listing 23. PEframe's anti-debug detection.

```
"antidbg": [
                "CheckRemoteDebugger",
                "DebugActiveProcess",
                "FindWindow",
                "GetLastError",
                "GetWindowThreadProcessId",
                "IsDebugged",
                "IsDebuggerPresent",
                "IsProcessorFeaturePresent",
                "NtCreateThreadEx",
                "NtGlobalFlags",
                "NtSetInformationThread",
                "OutputDebugString",
                "pbIsPresent",
                "Process32First",
                "Process32Next",
                "RaiseException",
                "TerminateProcess",
                "ThreadHideFromDebugger",
                "UnhandledExceptionFilter",
                "ZwQueryInformation"
            ],

            for lib in pe.DIRECTORY_ENTRY_IMPORT:
                for imp in lib.imports:
                    for antidbg in antidbgs:
                        if antidbg:
```

B.3.2. Debugger Fingerprint

As well as API function imports, one can also checks sample's strings in order to find known debugger symbols, which indicates the sample may use such values to check system properties. Tools such as JAMA [Liu 2012] implements such kind of pattern matching checking, as shown on Listing 24.

Listing 24. JAMA's anti-debug fingerprint.

```
DEBUGGING_TRICKS = {
    "SICE":"SoftIce detection",
    "REGSYS":"Regmon detection",
    "FILEVXG":"Filemon detection",
    "TWX":"TRW detection",
    "NTFIRE.S":"'DemoVDD By elicz' technique",
    "OLLYDBG":"OllyDbg detection",
```

B.3.3. NtGlobalFlag

The Process Environment Block (PEB) [Microsoft 2017c] is a system internal structure related to process management. Among its internal data, there is the `NtGlobalFlag`, which holds data related to the process heap. When a process is being debugged, specific flags of this field are enabled, thus allowing the debugger's presence check.

Specifically, the flags shown in the Table 8 are set. When a process is not being debugged, the typical value of the field is 0 whereas the value changes when a debugger is attached.

`NtGlobalFlags` can be accessed directly by using the undocumented function `RtlGetNtGlobalFlags` from `ntdll.dll`, as shown on Listing 25.

Table 8. NtGlobalFlag's heap flags.

Flag	Value
FLG_HEAP_ENABLE_TAIL_CHECK	0x10
FLG_HEAP_ENABLE_FREE_CHECK	0x20
FLG_HEAP_VALIDATE_PARAMETERS	0x40
Total	0x70

Listing 25. Undocumented NtGlobalFlag API Usage.

```
    _RtlGetNtGlobalFlags GetNtGlobalFlags =
    (_RtlGetNtGlobalFlags
        )(GetProcAddress(GetModuleHandle(_T("ntdll.dll")),
    "RtlGetNtGlobalFlags"));
```

This function makes a straightforward PEB reading implementation, as shown on Listing 26[20].

Listing 26. Wine's implementation of NtGlobalFlag API.

```
ULONG WINAPI RtlGetNtGlobalFlags(void)
{
    return NtCurrentTeb()->Peb->NtGlobalFlag;
}
```

Using an API function, however, can ease the sample's detection, since API imports are shown on PE structure and can also be monitored in runtime. Some authors, instead, prefer to access the PE structure directly in memory. The PEB base address is located at the `fs:0x30` offset, and `NtGlobalFlags` at `0x68`, being directly accessible. Listing 27 shows a possible implementation of this evasion technique.

Listing 27. NtGlobalFlag trick.

```
        call    puts
        mov     eax, [fs:0x30]
        mov eax, [eax+0x68]
        mov     eax, 0
        pop     rbp
```

A possible detector for this technique is to check for the PEB base address's load followed by a comparison on the `NtglobalFlags`'s offset. Listing 28 shows the implemented detector.

Listing 28. NtGlobalFlag trick detector.

```
    def check(self, section, address, instruction, op1, op2):

        if instruction in ['mov', 'movsx', 'movzx']:
            if 'fs:0x30' in op2:
                self.found_op1 = op1
                self.found_keyword = True
                return False

        if self.found_keyword:
            if instruction
                in ['cmp', 'cmpxchg', 'mov', 'movsx', 'movzx']:
```

[20] Wine implementation

```
                if '[' + self.found_op1 + '+0x68]'
                   in op1 or '[' + self.found_op1 + '+0x68]' in op2:
                    self.clear()
                    print "\"PEB NtGlobalFlag\" Detected! Section
                       : <%s> Address: 0x%s" % (section, address)
```

B.3.4. IsDebuggerPresent

Besides the global flags, the PEB struct has also an specific flag which indicates whether a process is being debbuged, as can be seen on Listing 29.

Listing 29. PEB structure.

```
        typedef struct _PEB {
                BYTE                            Reserved1[2];
                BYTE                            BeingDebugged;
```

The `BeingDebugged` flag is set when a debugger is attached to the process or when an debug-related API call is made on the process. This way, malware samples can check the presence of a debugger, and thus evade the analysis. This verification can be performed by using native API calls, such as `IsDebuggerPresent` [Microsoft 2016], as shown on Listing 30.

Listing 30. API-based debug detection.

```
                if(IsDebuggerPresent())
                        printf("debugged\n");
                else
                        printf("NO DBG\n");
```

This API implementation is a direct read from the PEB data, as can be seen on Listing 31.

Listing 31. Wine's IsDebuggerPresent implementation.

```
BOOL WINAPI IsDebuggerPresent(void)
{
        req->handle = GetCurrentProcess();
        ret = req->debugged;
        return ret;
}
```

Likewise Ntglobalflag's case, authors often avoid using API calls, since they can be traced, and opt to implement their own PEB checkers. One way of performing such checks is to load the PEB base address at the `fs:0x30` offset and thus reading the `BeingDebugged` flag at the `0x2` offset. This verification was implemented and can be seen on Listing 32.

Listing 32. IsDebuggerPresent trick.

```
mov eax, [fs:0x30]
        mov eax, [eax+0x2]
        mov     eax, 0
        pop     rbp
```

The detector for this technique should check whether these two access are performed, as shown on Listing 33.

Listing 33. IsDebuggerPresent trick detector.

```
    def check(self, section, address, instruction, op1, op2):

        if instruction
            in [’mov’, ’movsx’, ’movzx’] and ’fs:0x30’ in op2:
            self.found_op1 = op1
            self.found_keyword = True
            return

        if self.found_keyword:
            if instruction in [’mov’, ’movsx’, ’movzx’]:

                substring = ’[’ + self.found_op1 + ’+0x2]’

                if substring in op1 or substring in op2:
                    self.clear()
                    print "\"IsDebbugerPresent\" Detected! Section
                        : <%s> Address: 0x%s" % (section, address)
```

B.3.5. Hook Detection

Hooking is a technique on which the execution control flow is deviated to a trampoline function having arbitrary code. This deviation can be used to action logging or system subversion, for example. This modification can be done by using system facilities, such as API calls, in the case of `SetWindowsHook` [Microsoft 2017e], or by performing binary changes, in the case of `detours` [Microsoft 2017a].

A variation of this technique, called `inline hooking`, consists of patching a binary with a `JMP` instruction. An evasive sample can check whether its own binary was in-memory patched by checking whether a given snippet of code starts with a `JMP` instruction. Due to this check, a comparison to the`E9` opcode (`JMP`) can be seen on the generated code, as shown on Listing 34.

Listing 34. Hook detection trick.

```
cmp [eax+0xe9], eax
pop     rbp
```

The `CMP` instruction having this operand (`E9`) can be used to build an anti-analysis detector, as shown on Listing 35.

Listing 35. Hook detection trick detector.

```
     def check(self, section, address, instruction, op1, op2):

        if instruction == ’cmp
          ’ and (’0xe9’ in op1.lower() or ’0xe9’ in op2.lower()):
            print "\"HookDetection\" Detected
                ! Section: <%s> Address: 0x%s" % (section, address)
```

B.3.6. Heap Flags

Likewise global flags, PEB's heaps have also their own flags, as shown on table 9, which can be used as a analysis indicators by evasive malware.

Table 9. PEB's heap flags

Flag	Value
HEAP_GROWABLE	2
HEAP_TAIL_CHECKING_ENABLED	0x20
HEAP_FREE_CHECKING_ENABLED	0x40
HEAP_SKIP_VALIDATION_CHECKS	0x10000000
HEAP_VALIDATE_PARAMETERS_ENABLED	0x40000000

Samples can perform Heap checks by using the API call to `GetProcessHeap` [Microsoft 2017b] or by implementing their own checks, retrieving the PEB base address at `fs:0x30` offset and them referencing the default heap at `0x18` offset, as shown on Listing 36.

Listing 36. Heap Flags trick.

```
        mov     eax, [fs:0x30]
        mov eax, [eax+0x18]
        mov     eax, 0
        pop     rbp
```

As a detector, one can save the resulting address associated to the PEB query and then search for this value plus the heap offset on the proceeding instructions, as shown on Listing 37.

Listing 37. Heap Flags trick detection.

```
def check(self, section, address, instruction, op1, op2):

        if op1 is not None and 'fs:0x30' in op1:
            self.found_op = op1
            self.found_first = True
            return
        elif op2 is not None and 'fs:0x30' in op2:
            self.found_op = op2
            self.found_first = True
            return

        if self.found_op is not None and ((op1 is not None
           and '[' + self.found_op + '+0x18]' in op1) or (op2 is not
                                                                                None
                                                                                   and
                                                                                   '['
                                                                                   +
                                                                                   self
                                                                                   .
found + '+0x18]' in op2)):
```

```
                self.clear()
                print "\"HeapFlags\" Detected
                   ! Section: <%s> Address: 0x%s" % (section, address)
```

B.3.7. Hardware Breakpoint Detection

Modern processors provide hardware-assisted debugging facilities, such as hardware breakpoints, which allow instruction addresses to be stored on special registers, and thus stop the execution when one of such address is fetched by the processor.

Listing 38 shows an excerpt of code used to manipulate debugging data when a hardware debugger is attached. The `0xc` offset represents the debugger context struct whereas the `0x4` offset means an access to the debug register number 0.

Listing 38. Hardware debugger detection trick.

```
        mov     [fs:0x0], rsp
        mov     rax, [rsp+0xc]
        cmp     rbx, [rax+0x4]
```

A detector can be implemented by checking whether such kind of manipulation is performed on a given function. On the implementation provided on Listing 39, the detector checks for the following offsets: `0x4,0x8,0xc,0x10`, representing debug registers 0 to 3.

Listing 39. Hardware debugger trick detector.

```
if instruction
    in ['mov','movsx','movzx'] and 'fs:0x0' in op1 and 'rsp' in op2:
            self.seh = True

        elif self.seh==True and instruction
            in ['mov','movsx','movzx'] and 'rsp+0xc' in op2:
            self.found_first=True
            self.found_op=op1

        elif self.found_first
           ==True and instruction in ['mov','movsx','movzx
           ','cmp','cmpxchg'] and self.found_op in op2 and ('0x4'
           in op2 or '0x8' in op2 or '0xc' in op2 or '0x10' in op2):
                self.found_second=True

            if self.found_second==True:
                self.clear()
                print "\"HardwareBreakpoint\" Detected
                   ! Section: <%s> Address: 0x%s" % (section, address)
```

B.3.8. SS register

When running on a debugger, it replaces the first byte of given instructions by a trap flag. In order to be more transparent, many debugger solutions try to hide this trap flag. However, when the SS register is loaded through a `POP` instruction, the interruption is disabled until the end of the next instruction, avoiding invalid stack issues. This way, an evasive sample could insert a check right after popping the SS. The code presented on Listing 40 illustrates an implementation for this technique.

Listing 40. SS register trick

```
        pop      ss
        pushf
```

The implemented detection technique is to check the usage of the SS register immediately after it was pop-ed. The implementation of this detector is shown on Listing 41.

Listing 41. SS register trick detection.

```
             if instruction in ['mov', 'movsx', 'movzx']:
                if 'ss' in op1:
                    self.found_ss = True
            elif instruction== 'pop':
                if 'ss' in op1:
                    self.found_ss = True
            elif 'pushf' in instruction:
                self.found_flag=True
```

B.3.9. Software breakpoint

Unlike hardware breakpoints, which are register-based, and thus limited in number, software breakpoints are unlimited in practice. In order to identify the distinct points where the execution will be stopped, the debugger changes the first byte of the instruction to the `0xCC` byte, which represents the `INT3` instruction.

An evasive sample can scan its own memory and check for the `0xCC` byte, detecting the debugger, as shown on Listing 42.

Listing 42. Software debugger detection trick.

```
cmp      rax, 0xCC
```

As a detector for this technique, we can check the code for comparisons to the `0xCC` byte, as shown on Listing 43.

Listing 43. Software debugger detection trick detector.

```
if 'cmp' in instruction:
    if '0xcc' in op1 or '0xcc' in op2:
        self.found_cmp = True
if self.found_cmp==True:
    self.clear()
    print "\"Software Breakpoint\"
        Detected! Section: <%s> Address: 0x%s" % (section, address)
```

B.3.10. SizeOfImage

Another trick able to defeat some debuggers consists on changing the image size field, so a debugger becomes unable to parse its content. This technique can be seen on Listing 44.

Listing 44. SizeOfImage trick.

```
        mov     eax, [fs:0x30]
        mov     eax, [eax+0xc]
        mov     eax, [eax+0xc]
        addw    [eax+20],0x1000
```

As a detector for this technique, one can look for signals of value changes om this field, as shown on Listing 45.

Listing 45. SizeOfImage trick detection.

```
 if instruction in ['mov', 'movsx', 'movzx']:
            if 'fs:0x30' in op2:
                self.found_op1 = op1
                self.found_keyword = True
                return False

        if self.found_keyword:
            if instruction in ['mov', 'movsx', 'movzx']:
                if '[' + self.found_op1 + '+0xc]' in op2:
                    self.found_op2 = op1
                    self.found_keyword2 = True

        if self.found_keyword2:
            if instruction in ['mov', 'movsx', 'movzx']:
                if '[' + self.found_op2 + '+0xc]' in op2:
                    self.found_op3 = op1
                    self.found_keyword3 = True

        if self.found_keyword3:
            if instruction in ['addw','add','sub']:
                if '['+self.found_op3 + '20]':
                    print "\"SizeOfImage\" Detected! Section
                       : <%s> Address: 0x%s" % (section, address)
                    self.clear()
```

A way to defeat such trick is to recompute the image size. This calculation can be performed using the `VirtualQuery` [Microsoft 2017f] API.

B.4. Anti-VM

In this section, we present details from the techniques related to virtual machine detection.

B.4.1. VM Fingerprint

Some solutions leave presence indicators in the system, such as known strings. Thus, a straightforward way to detect an hypervisor is to check the presence of such strings on system properties. VMware, for example, presents a code, shown on Listing 46 which detects the VMware solution by the presence of its strings on the BIOS code.

Listing 46. VMware fingerprint.

```
int dmi_check(void)
        char string[10];
        GET_BIOS_SERIAL(string);

        if (!memcmp
           (string, "VMware-", 7) || !memcmp(string, "VMW", 3))
                return 1;
                   // DMI contains VMware specific string.
        else
                return 0;
```

The same way, a straightforward evasion detection technique is to verify the presence of such verification strings on the suspicious binary. PEframe, for instance, implements such kind of verification. Listing 47 shows an excerpt of PEframe's implementation which performs such checks.

Listing 47. PEFrame's VM fingerprint detection.

```
VM_Str  = {
                "Virtual Box":"VBox",
                "VMware":"WMvare"
        }

for string in VM_Str:
        match = re.findall
           (VM_Str[string], buf, re.IGNORECASE | re.MULTILINE)
        if match:
```

B.4.2. CPUID check

Another fingerprint approach is to make use of the `CPUID` instruction, which fills CPU registers with the vendor string. On VM cases, the hypervisor name is supplied. This way, a traditional evasive approach is to compare `CPUID` results to known VM-vendor strings, such as `Xen` or `QEMU`. The same way, a strightforward check for evasive samples is to locate such strings on the binaries, such as implemented by PEframe, as shown on Listing 48.

Listing 48. PEFrame's CPUID trick detection.

```
VM_Sign = {
        "Xen":"XenVMM",
```

Another detection possibility is to check the 31^{th} returned bit from `CPUID` instruction, which should return whether the processor has hypervisor capabilities or not [hexacorn 2014]. As presented by Vmware [VMWare 2010], this verification could be implemented as shown on Listing 49.

Listing 49. Vmware's hypervisor capabilities check.

```
int cpuid_check()
{
        unsigned int eax, ebx, ecx, edx;
        char hyper_vendor_id[13];

        cpuid(0x1, &eax, &ebx, &ecx, &edx);
        if (bit 31 of ecx is set) {
                cpuid(0x40000000, &eax, &ebx, &ecx, &edx);
                memcpy(hyper_vendor_id + 0, &ebx, 4);
                memcpy(hyper_vendor_id + 4, &ecx, 4);
                memcpy(hyper_vendor_id + 8, &edx, 4);
                hyper_vendor_id[12] = '\0';
                if (!strcmp(hyper_vendor_id, "VMwareVMware"))
                        return 1; // Success - running under VMware
        }
        return 0;
}
```

B.4.3. Invalid Opcodes

Hypervisors often support special opcodes and parameter values not accepted on physical machines. An evasive sample can try to execute such special instructions in order to verify whether the environments answer properly or not. The code from [Bachaalany 2005], reproduced on Listing 50, shows how this technique can be used to detect the VirtualPC hypervisor.

Listing 50. Virtual PC detection.

```
__try
  {
        _asm __emit 0Fh
        _asm __emit 3Fh
        _asm __emit 07h
        _asm __emit 0Bh
}catch{
        //real machine
```

Those bytes can be used as pattern for anti-anti-analysis techniques, as in PEframe, shown on Listing 51.

Listing 51. PEframe's VirtualPC detection.

```
VM_Sign = {
                "VirtualPc trick":"\x0f\x3f\x07\x0b",
```

B.4.4. System Table Checks

As previously mentioned, virtual machines change tables addresses, such as IDT and GDT. Thus, table relocations can be interpreted as virtual machine identifiers by malware samples. We can detect these kind of checks by verifying the presence of instructions related to table addresses on binaries. The instructions of interest are those which store the table addresses on given memory locations. The store meaning is due to the fact that a system address is stored on memory. On the other side, when a new table address is defined, this address is load-ed into the system. Listing 52 shows the detector for this technique.

Listing 52. Detecting instructions related to table addresses checking.

```
if instruction.lower() in ['sidt', 'sldt', 'sgdt', 'str']:
print "\"CPUInstructionsResultsComparison
    \" Detected! Section: <%s> Address: 0x%s" % (section, address)
```

According to [Radare 2008], the hypervisors' tables presents the values shown in the Table 10.

Table 10. Hypervisor's tables values.

Hypervisor	GDT	IDT
VMware 3.2	0xFFC05000	0xFFC6A370
VMware 4.0	0xFFC07000	0xFFC17800
VMware 4.5 Windows	0xFFC07C00	0xFFC18000
VMware 5.0/5.5 Windows	0xFFC07C80	0xFFC18000
VMware 5.0.0 (13124) Linux	0xFFC075A0	0xFFC18000
VMware GSX 3.1 Linux	0xFFC07000	0xFFC18000
VMware GSX 3.1 Linux	0xFFC07E00	0xFFC18000
VMware Player 1.0.1 Linux	0xFFC07880	0xFFC18000
Xen-2.0.7 (dom0 or domU)	0xFF400000	0xFC571C20
Kqemu 0.7.2	0xF903F800	0xF903F000
MS Virtual PC 2004	0xE80B6C08	0xE80B6408

This kind of IDT check can be found in practice on many samples. It first appear is credited to Joanna Rutkowska, on the non-academical literature. Listing 53 shows how this kind of check is usually implemented [Securiteam 2004].

Listing 53. IDT check implementation.

```
int swallow_redpill () {
            unsigned char m[2+4], rpill[] =
            "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
                    *((unsigned*)&rpill[3]) = (unsigned)m;
                    ((void(*)())&rpill)();
                    return (m[5]>0xd0) ? 1 : 0;
}
```

The hex-encoded data launches the IDT check instruction. Many detectors use these bytes as signatures, such as on PEframe's implementation, shown on Listing 54.

Listing 54. PEframe's IDT check detection.

```
VM_Sign = {
                "Red Pill":"\x0f\x01\x0d\x00\x00\x00\x00\xc3",
```

B.4.5. VMware Hypercall Detection

Similarly to the O.S syscalls, hypervisors have their own ways to be invoked by the running systems. These ways are usually named hypercalls. On Vmware-based systems, an hypercall is made by generating an I/O operation on a specific port (`VX`), present only on their virtualized systems. An evasive sample can try to write on this port and, if sucessful, identify it is a VMware-powered system. A detector for this technique consists on detecting the `IN` instruction on the `VX` port. Listing 55 presents the detector implementation.

Listing 55. Vmware trick detector.

```
def check(self, section, address, instruction, op1, op2):

        if instruction
            == 'in' and ('vx' in op1.lower() or 'vx' in op2.lower()):
             print "\"VMWareINInstruction\" Detected
                 ! Section: <%s> Address: 0x%s" % (section, address)
```

In practice [Laboratory 2012], the code to check the `VX` port looks like the one presented on Listing 56

Listing 56. VMware detection trick.

```
__asm
{
        mov eax, 0x564D5868 ; ascii: VMXh
        mov edx, 0x5658 ; ascii: VX (port)
        in eax, dx ; input from Port
        cmp ebx, 0x564D5868 ; ascii: VMXh
        setz ecx ; if successful -> flag = 0
        mov vm_flag, ecx
}
```

Detectors like PEFrame often consider the `VMx` string as a signatures of an anti-vm techniques, as shown on Listing 57.

Listing 57. PEframe's VMX detection.

```
VM_Sign = {
        "VMware trick":"VMXh",
```

B.4.6. A bit more about signatures

A similar approach is taken on `torpig` detection [MNIN.org 2006]. Its IDT check is used as a signature on PEframe (Listing 58) and on Snort (Listing 59).

Listing 58. PEframe's torpig detection.

```
VM_Sign = {
        "Torpig VMM Trick": "\xE8\xED\xFF\xFF\xFF\x25\x00\x00\x00\
            xFF\x33\xC9\x3D\x00\x00\x00\x80\x0F\x95\xC1\x8B\xC1\xC3",
        "Torpig (UPX) VMM Trick": "\x51\x51\x0F\x01
            \x27\x00\xC1\xFB\xB5\xD5\x35\x02\xE2\xC3\xD1\x66\x25\x32
            \xBD\x83\x7F\xB7\x4E\x3D\x06\x80\x0F\x95\xC1\x8B\xC1\xC3"
        }
```

Listing 59. Snort's torpig detection.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"BLEEDING-EDGE VMM
Detecting Torpig/Anserin/Sinowal Trojan"; flow:to_client,established;
content:"|51 51 0F 01 4C 24 00 8B 44 24 02 59 59 C3 E8 ED FF FF FF 25
00 00 00 FF 33 C9 3D 00 00 00 80 0F 95 C1 8B C1 C3|"; classtype:trojan-
activity; sid:20060810; rev:1;)
```

B.5. FakeRet

We have previously discussed what happens when the detection window changes. We explain here how a inter-block trick could be implemented. In order to give a clear idea of what happens on such case, Listing 60 and Listing 61 show the `fakeret` code and the splited disassembly,respectively.

Listing 60. Fakeret implementation.

```
xor eax, eax
jmp     fakeret
call    puts
ret
fakeret:
jnz     main
```

Listing 61. Fakeret disassembly.

```
  27:   31 c0                   xor    %eax,%eax
  28:   eb 26                   jmp    31 <fakeret>
  29:   e8 00 00 00 00          callq  2e <main+0x2e>
  30:   c3                      retq

0000000000000031 <fakeret>:
  31:   0f 85 00 00 00 00       jne    37 <fakeret+0x6>
  37:   b8 00 00 00 00          mov    $0x0,%eax
```

As can be noticed, the trick is placed right after the `RET`. In order to detect this trick, the detector would have to follow the `JMP` flow, not stopping at the end of the block. We can notice that the trick flows from the byte `28` to the `31` and then `37`. A linear check would proceed to the byte `29` instead.

B.6. Unaligned Evasion Techniques

We have previously mentioned YARA rules to detect unaligned tricks. These rules are presented in this section. Listing 62 shows the rule which checks for the `str` and `sidt` instructions, present on the `CPU Instruction` trick.

Listing 62. CPU Instruction detector implemented on YARA.

```
rule CPU_Detector : CPU
{
        meta:
        description = "CPU Instruction Detector"

        strings:
                $str={0F 00}
                $sidt={0F 01}

        condition:
                $str or $sidt
}
```

Listing 63 shows the rule corresponding to the `FakeJump` trick. The wildcards (?) are responsible for ignoring instruction immediates.

Listing 63. FakeJump detector implemented on YARA.

```
rule FakeJump_Detector : FakeJump
{
        meta:
        description = "FakeJump Detector"

        strings:
                $seq={31 ?? 0F}

        condition:
                $seq
}
```

B.7. 32-bit Limitation

We have implemented the tricks for the x86 (32 bits) architecture. In order to detect the tricks on the x86-64 (64 bits) architecture, the corresponding offsets should be updated. The Table 11 shows the correspondence between known offsets.

Table 11. Mapping: x32 to x64

Value	x32	x64
PEB	fs:0x30	fs:0x60
NtGlobalFlag	0x68	0xbc
_HEAP	0x40	0x70

In addition, some techniques, such as the `Stealth Import` are only functional on 32-bits systems.