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 capabilities 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 theoretical 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 successfully 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 environment 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 noticeable 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 external 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 general 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 verification 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 minimize 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 successors 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-succeeded 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-preceded 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 `POP`s
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 instructions 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 whether 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 distinct 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 ones.

4.2. Packer influence

In the last section, we could see that sections related to packer obfuscation were identified. 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`

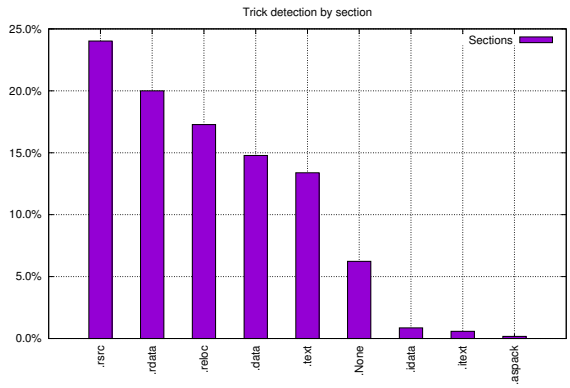

Figure 1. Tricks by section.

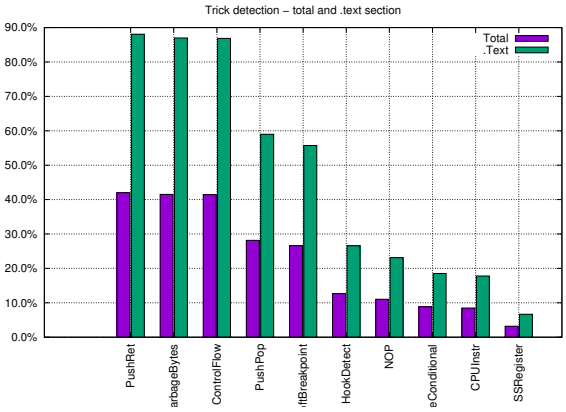

Figure 2. Tricks - total and `.text` section.

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.

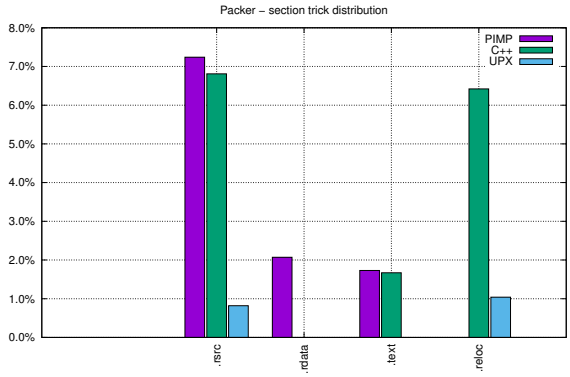

Figure 3. Packer distribution across binary sections.

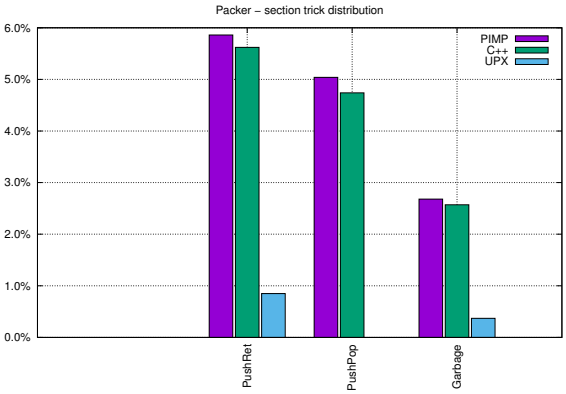

Figure 4. Tricks detected on distinct packers.

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 whether the detection of those aforementioned tricks could be used as a malicious program indicator, we compared the trick incidence 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 approaches 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.

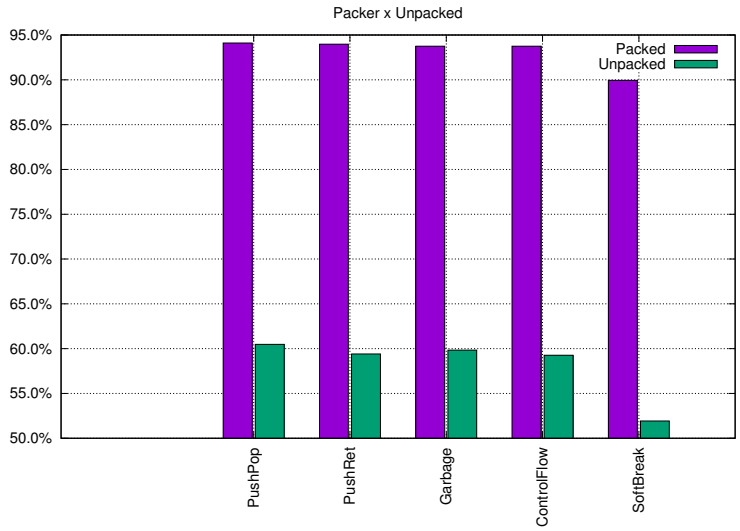

Figure 5. Packer influence on trick detection.

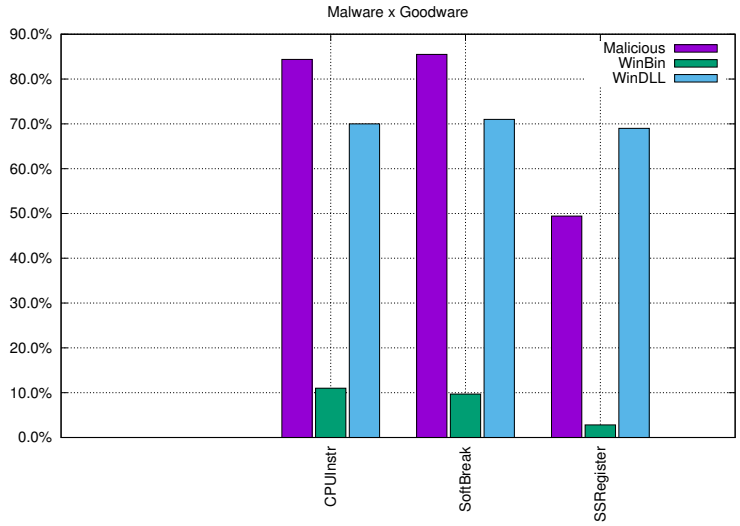

Figure 6. Tricks detection on malware and goodware.

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 Brazilian collected samples[10]. Figure 7 shows the results of comparing the datasets using the PEframe tool. We can observe that the Brazilian 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 Brazilian scenario [Botacin et al. 2015]. When performing the same checks using our developed tricks, as shown on Figure 8, we show that the Brazilian 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 simpler. More advanced tricks, such as some of those we have presented in this work,

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

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

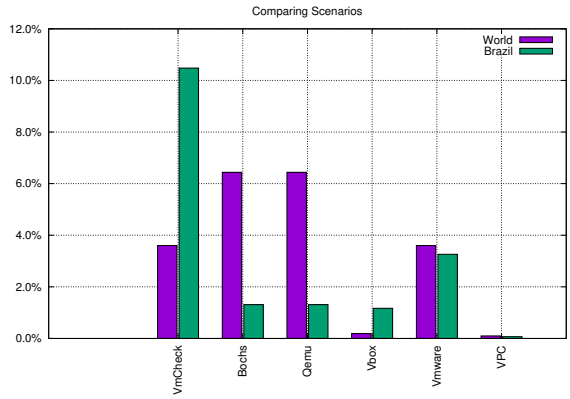

Figure 7. Comparing scenarios: PEframe detection.

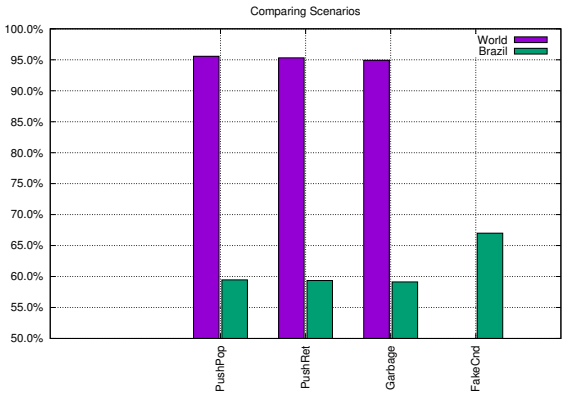

Figure 8. Comparing scenarios: Tricks detection.

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 splitted-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 this instruction is found. We considered the detection rate of this window as a ground-truth, 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 dis-alignment

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 ground-truth. We can observe the `Unaligned` results are significantly higher, indicating it is a viable way of hidding code.

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,

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

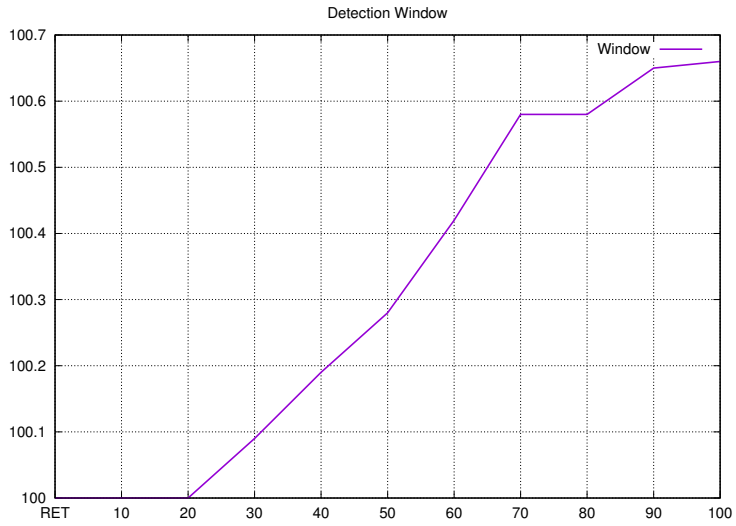

Figure 9. Evaluating block window effect on trick detection.

Table 4. Evaluating the occurrence of misaligned tricks.

Trick	Aligned	Unaligned
CPU	182	287
FakeJMP	63	203

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 such approaches in practice, we submitted 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 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.

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

Table 6. Evaluating 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 effectiveness of such approach can only be measured using dynamic analysis, which is a straightforward future work. Additionally, we aim to implement some kind of rule-based remediation [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 theoretical 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.

