Extended Abstract: Toward Systematically Exploring Antivirus Engines

Davide Quarta, Federico Salvioni, Andrea Continella, and Stefano Zanero

Politecnico di Milano
{davide.quarta,andrea.continella,stefano.zanero}@polimi.it
federico.salvioni@mail.polimi.it

Abstract While different works tested antiviruses (AVs) resilience to obfuscation techniques, no work studied AVs looking at the big picture, that is including their modern components (e.g., emulators, heuristics). As a matter of fact, it is still unclear how AVs work internally. In this paper, we investigate the current state of AVs proposing a methodology to explore AVs capabilities in a black-box fashion. First, we craft samples that trigger specific components in an AV engine, and then we leverage their detection outcome and label as a side channel to infer how such components work. To do this, we developed a framework, crAVe, to automatically test and explore the capabilities of generic AV engines. Finally, we tested and explored commercial AVs and obtained interesting insights on how they leverage their internal components.

1 Introduction

Antiviruses are still the major solution to protect end users. Despite the importance of malware detectors, there still is a need for testing methodologies that allow to test and evaluate them.

Current AV testing and comparison methodologies, rely on the capability of detection of samples [31, 7], and offer interesting insights on the time needed to detected a new sample for a product, but offer no insights in what are the capabilities implemented in an antivirus engine.

Moreover, most previous works focus on testing signature matching, and show how obfuscation techniques are effective against static analysis-based detectors [12, 6, 26, 27, 29]. However, modern AVs are complex systems that are not only based on static features and implement heuristics matching and emulation techniques (Figure 1) [27, 31, 32]. Other works focus on studying techniques to evade emulators [4, 5, 11, 14, 17, 16, 24, 30]. Neither of them focus specifically on AVs, or provide a comprehensive evaluation of AVs emulators. Rather, they look for new techniques to exploit AVs shortcomings. As a consequence, we lack of a recent and comprehensive study on modern AVs and of a complete understanding of how they really work. For instance, today it is usually believed that AVs mainly rely on signature matching while it is still unclear whether and how they leverage emulation engines. This sense of "obscurity" does not help protecting users, because it is hard to understand the features that AVs implement and evaluate their

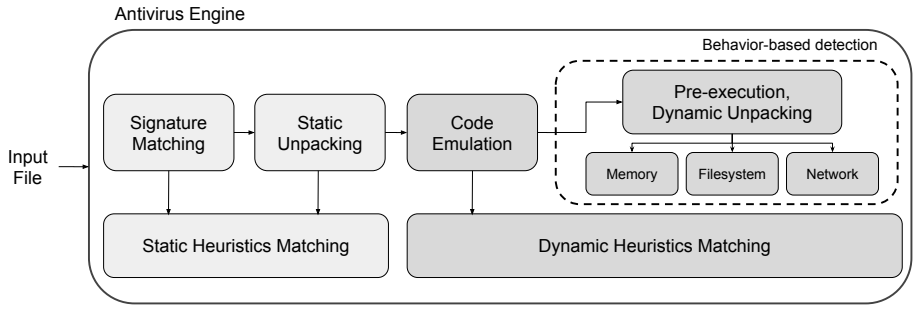

Figure 1: Overview of a typical AV engine.

capabilities. Instead, by exploring AV engines and understanding their internals, it is possible, for instance, to identify weaknesses, map the attack surface, or understand how stealthy samples can evade detection.

In this work, we move toward filling this gap by providing a comprehensive methodology to explore and test modern AVs, down to their core components. Our methodology builds upon the techniques devised so far in testing antiviruses [4, 6]. Differently from the techniques and frameworks proposed so far, our objective is to allow exploration of the features of antivirus engines both at a coarse and fine grained level. Doing so, we aim at answering the following questions: **(Q1)** Does an AV implement emulation? **(Q2)** Does it implement static unpacking? **(Q3)** Does it implement heuristics matching?

To answer them, the main challenge is that AVs are closed and very complex systems. Hence, performing an in depth analysis is a very complex task and requires deep reverse engineering knowledge. However, this would not scale and cannot be applied to efficiently evaluate many AVs. Instead, we use a generic, black-box methodology that does not make any assumption on the AVs implementation. We consider an AV as black-box system whose inputs are the scanned samples and whose outputs are the outcomes of the scans. First, we craft a set of samples aiming at triggering specific components in the AV engine. Then, observing how the AV reports and labels the input samples, we infer details on how the engine components work internally. Essentially, we leverage the scan outcome as a side channel to gain information about the detection process. In practice, we developed a set of tests in a framework, CRAVE, that, following our methodology, can be used to automatically retrieve the capabilities of a generic AV, requiring manual interventions only to generate peculiar samples such as an undetectable dropper. Specifically, we focused on and implemented three interesting tests: (1) Testing whether an AV adopts an emulation engine. (2) Testing whether an AV performs static unpacking. (3) Testing whether an AV relies on common heuristics.

Armed with CRAVE, we leveraged VirusTotal [1] to perform a large-scale experiment on 50 commercial AVs (including the popular Kaspersky, McAfee, Avast,

[1] `https://www.virustotal.com`

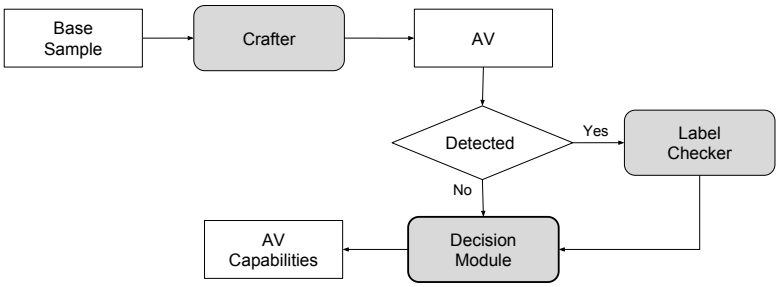

Figure 2: High-level methodology adopted to explore AVs and extract information about their internals and capabilities.

Symantec). In our experiments, we found that 4 AVs fully implement code emulation. Then, we successfully determined that 20 AVs implement static unpackers. Finally, we verified that our samples mutations trigger the heuristics matching engine and affect the detection outcome.

In summary, we make the following contributions:

- We demonstrated how variations to the samples submitted for analysis to an antivirus engine, combined with the resulting assigned label, constitutes a powerful side channel allowing to infer characteristics of the engines employed for scanning.
- Leveraging this side channel, we developed a framework, CRAVE, to automatically explore AV engines in a black-box fashion.
- Armed with CRAVE, we investigated the current state of AV engines. Specifically, we focused on understanding whether and how AVs leverage different components (heuristics, emulation engines, static unpackers) to process and detect malicious samples.

In the spirit of open science, we make the code developed for CRAVE publicly available [2].

2 CRAVE Testing Methodology and Framework

Our methodology is based on providing AVs with different samples as input and observing how they report such samples. As depicted in Figure 2, we first craft a new sample by obfuscating and mutating a base sample. Such mutations embed features that under analysis reveal whether specific AV components have been triggered. Then, we provide the AV with our crafted sample and observe the detection outcome. If the AV reports a detection, we also verify whether it labels the sample accordingly—same label of the initial base sample. Intuitively, if the AV

[2] `https://github.com/necst/crave`

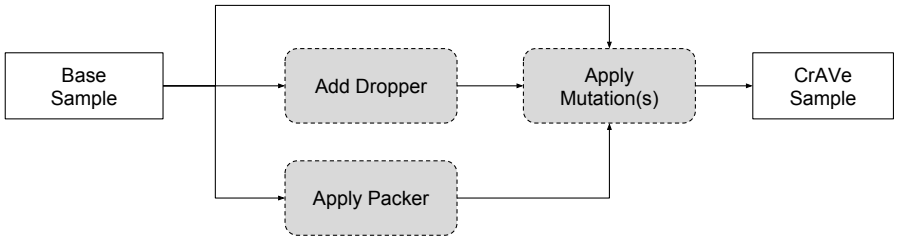

Figure 3: Crafting phase, different "mutations" are applied on the sample.

correctly detects and labels our crafted sample, it means it managed to properly handle all the mutations and spot the malicious payload.

We implemented CRAVE to support Windows executables. However, our approach is generic and can be applied to any other operating system and executable format. CRAVE is composed by three key components: a *samples crafter*, a *label checker*, and a *decision module*.

2.1 Sample Crafter

CRAVE leverages a set of base samples described in Table 1, on which it applies different mutations to stimulate the AV engine under analysis. Figure 3 shows each step of the crafting process. The base sample goes through a set of different (optional) steps: addition of a dropper, applying a packer, and a set of mutations.

(1) Dropper. We developed a custom dropper with low detection rate to test emulation features. At run-time, our dropper decrypts a malicious sample contained in the resources section using a long (30 bytes) key, stores it on the filesystem and executes it. CRAVE uses the dropper to identify AVs that perform emulation. If an AV is able to correctly identify and label the malicious payload embedded in our dropper, we infer it implemented dynamic analysis features (emulation).

(2) Mutations. We test four popular heuristics usually implemented by AVs [27]. Table 2 describes the samples features that we modify to trigger the heuristics matching. We compare how the detection outcome changes after applying our mutations. We do this for two different purposes: First, applying them to a benign

Table 1: Base samples used in our analysis

Sample class	Description
goodware	goodware samples that is not mis-detected
malware	known malware sample (detected)
dropper	simple dropper that decrypts a sample, stores in on the filesystem and finally executes it

sample to determine whether AV heuristics match such features. Second, on the other hand, applying them to a malicious sample to determine whether such features can be used to evade detection.

(3) Packers. We apply known packers to our base samples in order to understand whether AVs implement static unpackers. Specifically, we pack a malicious sample with a given packer. Then, we edit the packer's stub overwriting its entry point with a `RET` instruction. Finally, leveraging the detection outcome and label as a side channel, we can infer if the AV employs static unpacking. In fact, if the AV correctly labels the packed malicious sample, it means it successfully unpacked it. However, since we broke the packer's stub, this implies the unpacking has been performed statically.

2.2 Label Checker

The *crafting* phase might change the detection outcome of certain AVs in such a way that the submitted sample is still correctly recognized as malicious, but as a variant of the sample (e.g., `W32/Virut.Gen` and `W32/Virut.X` after applying mutations). Note that, CRAVE needs to consider labels matches in its approach. For instance, when testing if an AV performs emulation, we want to verify that the AV correctly labels the dropped payload, and does not simply detect a `GenericDropper`. Therefore, CRAVE needs to handle little, and irrelevant, differences in the labels of the same sample.

AVClass [28], in its actual state, cannot be used for a direct comparison of two labels, rather it is used to label a sample as a variant of a known family. Thus, we devised a simple method for comparing the labels based on two steps: First a filtering phase in which we remove generic and heuristic labels. Second, a matching phase, based on the aliases for families leveraging AVClass and Metaphone [25]. Metaphone is a phonetic matching algorithm commonly used for indexing and matching text. In our empirical tests, Metaphone performed well on the assigned AV labels. For example, `W32/Virut.Gen` and `W32/Virut.X` are both encoded into *FRT*, making the matching process easy and straightforward. When a sample is labeled with a generic signature, this phase discards it to avoid imprecise detections.

Table 2: Features implemented to test heuristics.

Class	Feature	Description
Section Names	`random`	Randomly generated (alphanumeric)
	`randomdot`	`Random`, starts with a dot
	`infer`	Based on section characteristics and inferred content
Permissions	`rwx`	Force all sections to have *rwx* permissions
Checksum	`correct`	Force a correction of the checksum
	`broken`	Do not correct the checksum

2.3 Decision Module

The decision module compares the detection outcomes and the labels leveraging our Label Checker. In practice, it implements the inferring process described earlier to determine AVs capabilities.

(1) Test Emulation. Comparing the labels assigned to our dropper and malicious payload, the decision module determines whether an AV performs emulation.

(2) Test Heuristics. Looking at new or missed detections of the mutated sample in respect to its original base specimen, the decision module determines whether a certain heuristic can be employed to change the detection outcome.

(3) Test Static Unpacking. Comparing the labels assigned to our packed sample and malicious payload, the decision module determines whether an AV performs static unpacking.

3 Experimental Results

First, we verified whether AVs perform emulations. Then, we verified whether AVs perform static unpacking. Finally, we verified how sample features affect the heuristic engine.

Dataset and Setup. We leveraged VirusTotal to perform a large-scale experiment. Our methodology requires a malware sample that is detected by the tested AV. For this reason we looked for a sample to maximize the number of reported detections in VirusTotal. For our experiments, we used a variant of `Virut` [3], which at the time of our experiments was detected by 64 out of 67 AVs.

Requirements. Our methodology needs three requirements to be satisfied:
1) The goodware sample must not be detected as malicious by the AV.
2) The known malicious sample must be detected by the tested AV.
3) The dropping logic must not be flagged as malicious by the AV.
These assumptions are easy to meet and we can verify that they hold as a first step of our methodology.

3.1 Testing Emulation

Following our methodology described in Section 2, we tested if AVs perform emulation (Table 3). After filtering out all the AVs from VirusTotal that do not satisfy our aforementioned requirements, we reduced the initial list of 64 AVs to 50. Among these 50 AVs, CRAVE identified 4 of them performing full emulation (`AV4`, `AV15`, `AV16`, `AV19`). Six more AVs were able to detect our crafted dropper, but they reported an inconsistent label. Hence, we could not determine whether they performed emulation.

Dropper Variation (No Execute). We repeated this experiment changing our crafted dropper. Specifically, our new dropper did not execute the dropped file

[3] SHA256: 06c62c4cb38292fb35f2c2905fce2d96f59d2d461fa21f7b749febfed3ef968d

anymore, but it only decrypted it and dumped it on a file. As a consequence, this caused `AV4` and `AV15` to report it as benign. Interestingly, a new, different AV correctly detected and labeled the new dropper, showing it performed emulation. We could not speculate the reason behind the latter, as this would require deeper investigation.

3.2 Testing Static Unpacking

Following our methodology described in Section 2, we tested if AVs perform *static* unpacking. As shown in Table 3, we tested AVs against 5 different, known packers (`UPX`, `MEW`, `ASPack`, `kkrunchy`, `Petite`). After filtering out the AVs from VirusTotal that do not satisfy our requirements, we obtained a list of 64 AVs. All in all, we found 17 AVs that statically unpack `UPX`, 0 `MEW`, 6 `ASPack`, 0 `kkrunchy`, 4 `Petite`.

Stressing Static Unpacking. We repeated the same experiment as above applying the `infer` mutation to our crafted, packed samples. The results of this test shows that one AV (`AV18`) did not unpack `ASPack` and `Petite` anymore, suggesting it might rely on sections names to understand the type of packer—for instance, `ASPack` introduces a section named `.aspack`.

Goodware. Then, we performed a different experiment by packing our benign `helloword` and testing whether AVs detect the packers independently if they embed goodware or malware. We found 2 AVs detecting `UPX`, 16 `MEW`, 2 `ASPack`, 26 `kkrunchy`, 7 `Petite`.

3.3 Testing Heuristics

We tested if AVs match popular heuristics (Table 3).

Goodware. First, we tested whether applying our mutations (Table 2) to our benign `helloworld` triggers detection. We found that our `infer` and `checksum` do not trigger any new detection. On the other hand, `RWX` triggers 1 new detection, `random` 9, and `randomdot` 10.

Malware. Second, we verified whether applying our mutations (Table 2) to our malicious payload (i.e., `virut`) causes missed detections. We found that `checksum` causes 1 missed detection, `RWX` 1, `random` 2, `randomdot` 4, and `infer` 5.

This experiments show that applying mutations to trigger heuristics matching can affect the AVs detection outcomes, and demonstrate the need for a deeper exploration of such behaviors in future work.

4 Limitations and Future Works

Setup Limitations. While leveraging VirusTotal allows to easily perform large-scale experiments, it also has the limitation of not knowing how each AV is configured. The AVs included in VirusTotal might be parameterized with a different

Table 3: Summary of the results of our experiments on a selection of 20 AVs. For each heuristic, '+' means it triggers new false detections on goodware, '-' means it causes missed detections for a malware.

		Static Unpacking					Heuristics				
AV	**Emulation**	**UPX**	**MEW**	**ASPack**	**kkrunchy**	**Petite**	**RWX**	**random**	**randomdot**	**infer**	**checksum**
AV1								+	+		
AV2									-	-	
AV3		✓								-	
AV4	✓	✓									
AV5		✓		✓							
AV6		✓		✓		✓					
AV7							-	-	-	-	-
AV8		✓									
AV9				✓							
AV10		✓									
AV11		✓									
AV12							+				
AV13		✓									
AV14		✓		✓							
AV15	✓										
AV16	✓										
AV17								-	-	-	
AV18				✓		✓					
AV19	✓	✓									
AV20		✓									

heuristic/aggressiveness level than the official end-user default configuration [1]. Indeed, we do not aim at providing a quality evaluation of commercial products.

Methodology Limitations. Our methodology cannot fully work if AVs only use generic labels. For instance, in our experiments we found two AVs, that labeled different samples in the same way (e.g., `''Malicious.HighConfidence''`).

Some AVs might emulate only samples showing specific features (e.g., having a RWX memory area, or a specific sequence of instructions). In this case, we would need to apply or fuzz typical trigger conditions.

Since we cannot control what happens inside AVs engines, when we test if an AV performs emulation, we might face a case in which the emulator fails because of an implementation flaw (e.g., missing emulated API). This would make our methodology infer that the AV does not perform emulation, while it actually does. However, from the detection capabilities perspective this has the same effect: the dropped malware is not detected.

Future Works. Other than addressing the above limitations, we foresee two other research directions. First, studying the differences between local and cloud AVs, and between their free and premiums versions. Second, exploring different side channels that can be used to extract information from AV engines. For instance, writing in memory the extracted information and then reading such information by dumping and inspecting the memory of the AV process.

5 Related Works

Antivirus Testing: Signature Matching. Researchers studied how syntactic obfuscation techniques can defeat signature matching and evaluated AVs capabilities of detecting malware samples that implement such obfuscation techniques [6, 9]. However, they did not focus on heuristics and emulation features. The same apply for many other works that focus on techniques to defeat static analysis [12, 26].

Evasion of Emulators-based Detection. Several works described techniques to escape from emulators [2, 15, 30, 17, 13, 10, 16, 5, 8, 4]. However, while they show how emulators can be effectively bypassed, they do not propose any generic methodology or comparison between AVs.

Attacking Antivirus Software. Ormandy revealed several implementation vulnerabilities in commercial antivirus products showing how their complexity often exposes a large attack surface [18, 19, 20, 21, 22, 23]. Wressnegger et al. derived AV signatures from malware and proposed a novel class of attacks called "antivirus assisted attacks" that, abusing the byte-pattern based signature matching flow, allow adversaries to remotely instruct AVs to block or delete content on the victim machine [33]. Al-Saleh et al. determined through time channel attacks whether the database of an AV has been updated with certain signatures or not [3].

Emulators Fingerprinting. AVLeak [4] extracts artifacts from AV emulators using a black-box approach. It maps known malware samples to bytes and then leverages custom droppers to leak data by exploiting AVs labeling.

6 Conclusions

In this work, we performed an exploratory study of modern AVs by testing their capabilities. We adopted a black-box methodology that leverages the detection outcome and label as a side-channel to obtain info about the AVs internals. In our experiments on 50 AVs, we found that not all the AVs perform full emulation, that most of AVs implement static unpackers for known packers, and that applying mutations to input samples in order to trigger heuristics matching affects the detection outcome. We believe that testing and exploring AV engines helps reducing that sense of "obscurity" that is often hidden behind AVs. In fact, testing AVs features is a fundamental step in order to evaluate their capabilities, identify weaknesses, or map their attack surface. In conclusion, while we only scratched

the surface of AV testing and exploration, our results are promising and show that it is interesting to extend our study in future work. We envision our framework to be extended and used as a reference to test how AVs behave, and how they rely on each internal component.

7 Acknowledgements

This work has been supported by the Italian Ministry of University and Research FIRB project FACE (Formal Avenue for Chasing malwarE) – grant agreement N. RBFR13AJFT, and by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie – grant agreement N. 690972

References

[1] VirusTotal, About Page. `https://www.virustotal.com/en/about/`.
[2] Just-In-Time Malware Assembly: Advanced Evasion Techniques. Invincea white paper, 2015.
[3] M. I. Al-Saleh and J. R. Crandall. Application-Level Reconnaissance: Timing Channel Attacks Against Antivirus Software. In *LEET*, 2011.
[4] J. Blackthorne, A. Bulazel, A. Fasano, P. Biernat, and B. Yener. AVLeak: Fingerprinting Antivirus Emulators through Black-Box Testing. In *USENIX Workshop on Offensive Technologies (WOOT)*, Austin, TX, 2016. USENIX Association.
[5] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. June 2008.
[6] M. Christodorescu and S. Jha. Testing Malware Detectors. *SIGSOFT Softw. Eng. Notes*, July 2004.
[7] A. comparatives. Intipendent tests of anti-virus software.
[8] M. Cova. Uncloaking Advanced Malware: How to Spot and Stop an Evasion, 2015.
[9] M. Dalla Preda and F. Maggi. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*, 2017.
[10] K. Economou. Escaping the avast sandbox using a single ioctl. 2016.
[11] P. Ferrie. Attacks on more virtual machine emulators. 2007.
[12] Y. Ilsun and Y. Kangbin. Malware obfuscation techniques: A brief survey. 2010.
[13] P. Jung. Bypassing sandboxes for fun. 2014.
[14] D. Keragala. Detecting malware and sandbox evasion techniques. 2016.
[15] J. A. P. Marpaung, M. Sain, and H.-J. Lee. Survey on malware evasion techniques: State of the art and challenges. Feb 2012.
[16] H. Mourad. Sleeping your way out of the sandbox. 2015.
[17] E. Nasi. Bypass antivirus dynamic analysis. 2014.
[18] T. Ormandy. Comodo antivirus: Emulator stack buffer overflow handling psubusb packed subtract unsigned with saturation.
[19] T. Ormandy. Comodo: Integer overflow leading to heap overflow in win32 emulation.
[20] T. Ormandy. Eset nod32 heap overflow unpacking epoc installation files.
[21] T. Ormandy. Symantec/norton antivirus aspack remote heap/pool memory corruption vulnerability cve-2016-2208.

[22] T. Ormandy. Sophail: A critical analysis of sophos antivirus. 2011.
[23] T. Ormandy. Sophail: Applied attacks against sophos antivirus. 2012.
[24] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A Fistful of Red-pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, Berkeley, CA, USA, 2009. USENIX Association.
[25] L. Philips. Hanging on the metaphone. *Computer Language*, 1990.
[26] B. B. Rad, M. Masrom, and Ibrahim. Camouflage in malware : from encryption to metamorphism. *IJCSNS*, 2012.
[27] B. B. Rad, M. Masrom, and S. Ibrahim. Evolution of Computer Virus Concealment and Anti-Virus Techniques:AShort Survey. *CoRR*, 2011.
[28] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. AVclass: A tool for massive malware labeling. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016.
[29] A. Sharma and S. K. Sahay. Evolution and detection of polymorphic and metamorphic malwares: A survey. *International Journal of Computer Applications*, March 2014.
[30] S. Singh. Breaking the sandbox. 2014.
[31] O. Sukwong, H. Kim, and J. Hoe. Commercial antivirus software effectiveness: An empirical study. *Computer*, March 2011.
[32] P. Szor. *The art of computer virus research and defense*. Pearson Education, 2005.
[33] C. Wressnegger, K. Freeman, F. Yamaguchi, and K. Rieck. Automatically inferring malware signatures for anti-virus assisted attacks. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*. ACM, 2017.