AntiViruses under the Microscope: A Hands-On Perspective

Marcus Botacin[1] Felipe Duarte Domingues[2] Fabrício Ceschin[1] Raphael Machnicki[1]
Marco Antonio Zanata Alves[1] Paulo Lício de Geus[2] André Grégio[1]

[1]Federal University of Paraná (UFPR-BR)
{mfbotacin,fjoceschin,mazalves,gregio}@inf.ufpr.br
rkmach17@gmail.com
[2]University of Campinas (UNICAMP-BR)
paulo@lasca.ic.unicamp.br
f171036@dac.unicamp.br

Abstract

AntiViruses (AVs) are the main defense line against attacks for most users and much research has been done about them, especially proposing new detection procedures that work in academic prototypes. However, as most current and commercial AVs are closed-source solutions, in practice, little is known about their real internals: information such as what is a typical AV database size, the detection methods effectively used in each operation mode, and how often on average the AVs are updated are still unknown. This prevents research work from meeting the industrial practices more thoroughly. To fill this gap, in this work, we systematize the knowledge about AVs. To do so, we first surveyed the literature and identified existing knowledge gaps in AV internals' working. Further, we bridged these gaps by analyzing popular (Windows, Linux, and Android) AV solutions to check their operations in practice. Our methodology encompassed multiple techniques, from tracing to fuzzing. We detail current AV's architecture, including their multiple components, such as browser extensions and injected libraries, regarding their implementation, monitoring features, and self-protection capabilities. We discovered, for instance, a great disparity in the set of API functions hooked by the distinct AV's libraries, which might have a significant impact in the viability of academically-proposed detection models (e.g., machine learning-based ones).

1 Introduction

AntiViruses (AVs) are one of the main solutions to defend users against malware, being present in the majority of computer systems [112]. The popularity of AVs has led to a myriad of research proposals to enhance and bypass them, but little attention was given to their internals and development decisions.

AVs are intricate pieces of software and their complexity evolves at the same pace that malware becomes more sophisticated. Current AVs have their own developed parsers for multiple file formats, they load multiple kernel drivers to monitor the system, and they have to protect themselves from attacks. Although all these factors are key to the operation of an actual AV, they are often not discussed by the academic literature, which is more focused on presenting prototypes of the proposed concepts than actual implementations.

When AV prototypes are proposed, they are often focused on single techniques (e.g., machine learning-based AVs, or a new signature matcher), whereas current AVs operate based on multiple engines, which are activated according to the requested scan type and context. While on-demand file checks might operate using signatures, real-time scans might be based on an API-based machine learning model, and suspicious files might be later submitted to cloud scans. We believe that all these operation modes should be considered on developed solution's threat models aiming to reach real scenario's usage.

We believe that a significant reason for the lack of information on AV internals is that most AVs are closed-source solutions. In most cases, the only way to have access to real AV's source code is when they are disclosed by attackers: past breaches disclosed source-code for distinct AV companies [191, 88]. However, since these events happened a decade ago, even if one had access to these codes, they would be too much outdated to reflect a current AV solution.

Though keeping their source-code private is an AV vendor's right in a very competitive market, the lack of information does not allow researchers to model solutions that fully resemble a real-world scenario. For instance, a researcher proposing a new AV is not fully aware of: (i) in which scenarios signatures are deployed by the companies (e.g., zero-days detection, false-negative mitigation, so on); (ii) how many signatures are added on average to a typical database (and old signatures are removed); (iii) how much overhead is accepted for a typical scan; (iv) how often an AV is updated on average; and so on.

To bridge this understanding gap, we surveyed the existing academic literature to identify what is known and unknown about AV internals. We further analyzed real, commercial AV solutions to fill the existing knowledge gaps with information about actual implementation. Our goal is to present a broad and representative analysis of current AV solutions.

To ensure broadness, we selected for analysis AVs covering the most popular platforms (Windows, Linux, and Android), such that we first present results regarding Windows, the most developed AV ecosystem, and later compare them with the results regarding the two most recent platforms. To ensure representativeness, we selected the most popular desktop AVs according to the AVTest's criteria [23] and mobile AVs according to the Google Play Store market share.

Analyzing AVs internals is hard, as it encompasses a multitude of subsystems (filesystem, network, processes, drivers, libraries, databases, and so on) that communicate among themselves. Thus, a single strategy and/or single inspection point would not be enough to fully understand AVs operation. Therefore, we opted to inspect AVs during their whole operation using the most suitable tools according to the task performed by the AVs at each moment. Our approach consisted of tracing AV operations on a virtual machine (VM) from the installation procedure to the update process via the multiple scan modes. We also fuzzed multiple AV interfaces to check their outputs against known threats, as well as for bug hunting purposes.

With this work, we expect to foster research in the AV internals field and help researchers to better model and parameterize their solutions. It is important to highlight that this work does not propose new detection mechanisms nor techniques to bypass AV's detection, but an analysis of their internals and implementation decisions, although these might also lead to new development and evasion opportunities as an associated outcome.

Contributions. In Summary, our main contributions are threefold:

- We survey the literature to summarize the existing knowledge about AV internals and existing knowledge gaps.
- We bridge the identified knowledge gaps by analyzing actual AVs and summarizing our findings.
- We discuss the current challenges of AV operation and pinpoint possible directions for future developments. More specifically, we discuss the following aspects of AVs operation:
  - We describe the multiple AV components, such as engines, browser plugins, and libraries, regarding their operations and implementation choices.
  - We highlight the differences in AV's implementations for multiple environments, from the reliance on event tracing for Windows, to the use of preloading on Linux, and the abuse of accessibility services for Android app's inspection.
  - We discuss the self-protection mechanisms employed by the AVs, including their strong points (e.g., DLL unload prevention) and weaknesses (e.g., integrity tampering in safe mode).
  - We evaluate the multiple detection methods and operation modes leveraged by the AVs. We conclude that although modern AVs indeed evolved to consider distinct information sources, such as cloud data and behavioral profiles, most of their detection capabilities are still provided by static checks.

Organization. This work is organized as follows: In Section 2, we motivate the study of AV internals; In Section 3, we present background information about AV operation; In Section 4, we present our experimental methodology; In Section 5, we analyze the anatomy of actual AVs; In Section 6, we analyze detection challenges; In Section 7, we analyze AV's self-protection mechanisms; In Section 8, we analyze the performance impact imposed by real AVs; In Section 9, we analyze the differences of AV's implementations for distinct platforms; In Section 10, we discuss our findings; and finally, we draw our conclusions in Section 11.

Disclaimer: Intellectual Property. The sole purpose of this work is to academically understand AV's operations, with no commercial implications. We conducted all analyses by inspecting AV components as they are installed in the user's machines, acting the same way as any skilled customer could act to check whether the product works as advertised. We did not extract nor decompile any code from AV's components; We only present pseudo-code representing our understanding of AV's operations. We also only display original files if they were available in clear in user's machines. Thus, we did not disclose any intellectual property-protected information in this work.

2 Why Studying AV Internals?

The need for researching AVs is clear: we need to develop more secure solutions. Even though, the need for researching AV internals is blurry for many minds. Why is it important? The answer is because the solutions proposed in a research context should migrate from prototypes to production at some time to actually protect users [30], but this is only possible when the research proposals fit the way that AVs actually operate. Therefore, it is important to study AVs internals to propose solutions compatible with them.

There are many cases where a better understanding of AVs internals would help research developments, for instance:

- When **hooking APIs**. Many researchers propose hooking APIs to collect data for machine learning-based detectors [172]. Is the number of APIs hooked by the prototype compatible with the number of APIs hooked by actual AVs?
- When **accelerating signature matching**. Many researchers propose mechanisms to speed up signature matching [77]. Are signatures still used by AVs? Are they prevalent? What kind of signatures are used?

- When **measuring performance**. Many research works proposed to account for the performance impact of running AVs [7]. But, Is the performance AV independent of their internal engines? Can distinct engines be compared? Do all AVs operate in the same modes?

This paper aims to answer both the aforementioned as well as related questions in the expectation of helping researchers in bridging the gap between prototypes and actual AVs in their future AV developments.

3 Background & Related Work

In this section, we present the security properties that AVs are expected to fulfill and discuss existing research work gaps in analyzing these properties.

3.1 AV Research Literature

There is no doubt that AVs are the main security solution deployed by most users. AVs have become so popular that even rogue AV solutions can be found in the market [105, 55]. This popularity naturally fostered varied research on the subject. AV research has been significantly evolving, both in quality as well as in quantity. In the past, the few existing research works used to look to individual threats, such as the MyDoom case [185]. Currently, many research works focus on large-scale approaches. Despite such evolution, AV research is still limited to the external AV factors, i.e., they do not cover AV's internal aspects, such as its implementation decisions. Table 1 summarizes the most studied aspects of AV's operations according to our literature search.

Table 1: **Related Work on AVs.** Summary of the most studied aspects.

Task	Aspect	Work
Assessment	Socio-Cultural Factors	[73, 62, 111]
	Labeling Problem	[114, 85, 163]
	Detection Evaluation	[29, 79]
Matching	ClamAV Engine	[61]
	Detection Mechanisms	[144]
	AV Bypasses	[80, 142]
Platforms	Mobile	[68]
	New Paradigms	[195, 32, 196]
Performance	Cloud AV	[60, 91]
	FPGA AV	[33, 77]
AV Internals	Overview	**This Work**

The first external factor most evaluated by related work is to assess the effectiveness of the AVs to detect malware samples. Whereas these works investigate relevant problems, such as the diversity of the labels assigned by distinct AVs, these works do not delve into the details about why the distinct engines flag the samples differently (we aim to discover in this work). The second class of evaluated factors covers the development of detection engines. Many works proposed distinct approaches to flag malicious activities, such as the inspiration on immune systems [196]. The major drawback of these approaches is that they are only proof of concepts and do not resemble a real engine. They do not operate, for instance, under the constrained conditions of a real engine (as evaluated in this work). Most of these works are developed on top of `ClamAV`. Whereas this is the open-source solution closest to a real AV, it still far away to be representative of a state-of-the-art engine (e.g., it does not support real-time monitoring, for instance). Other research work classes focus on the underlying platforms that support AV operation. A typical research work task is to port AVs to mobile environments [68] to operate in resource-constrained devices. The major drawback of these work is that they do not represent any research breakthrough, but implement existing detection techniques. Finally, some work focus on improving AV's performance. The most commonly adopted solutions are moving the AV to a cloud-server and/or providing an efficient hardware implementation to them (e.g., via dedicated FPGAs). Although all of these are important aspects of AV's operation, they all lack information about AV's internals. In this work, we aim to bridge this gap.

3.2 AV Internals Literature

The literature on AV internals is not as large as the one related to the proposals of new solutions, as previously shown. Only a few studies cover the particular aspects of AV's operation. As pointed by Aycock in his malware book [24]: "*the AV community tends to be very industry-driven and insular, and isn't in the habit of giving its secrets.*" Therefore, most reports of AV internals are found outside of the academic literature. Whereas fundamental to help to understand AV's internal, these reports lack scientific systematization. For instance, they focus on particular solutions (e.g., an analysis of hooks on the Kaspersky AV [165], and/or Windows defender reverse engineering findings [39]), but do not draw a landscape of the whole AV market (as this work does).

These landscapes started to be presented by the first academic work tackling the problem (e.g., a review of AVs using signatures [4], or ML detectors [193]). The major drawback of the academic literature is that most works adopt black-box

analyses procedures [156], exploiting the fact that still few solutions employ anti-black-box technique [70]. Whereas this approach provides interesting information, such as about the AV's energy consumption [152], they do not reveal the AV company's project decisions.

The closest work to reveal AV internals is the "Antivirus Hackers Handbook" [106], which presents a reverse engineer methodology for inspecting AVs and the findings of its application to multiple AVs. Whereas this is the most complete reference so far, it needs to be updated to cover the recent advances of this industry (e.g., cloud scans, machine learning, and so on) and also expanded to cover other platforms. Whereas the first step towards this direction was given in a recently released book chapter [138], this does not cover AV in deep details as the first book. Therefore, in this work, we aim to update the knowledge about AV internals by still considering the originally proposed methodology [106] as the basis to ours and complement their findings.

Table 2: **Related Work Summary.** Our work presents the most comprehensive and up-to-date analysis.

Work	Landscape	Avs	Studied Aspect	Modern AV
[165]	✗	Kaspersky	Function Hooks	✓
[39]	✗	Defender	Emulation	✓
[4]	✓	Multiple	Signatures	✓
[193]	✓	Multiple	Machine Learning	✓
[152]	✓	Multiple	Energy Consumption	✓
[138]	✓	Generic	Detection	N/A
[106]	✓	Multiple	Overall	✗
This	✓	Multiple	Overall	✓

In Table 2, we show a comparison of this work and the works available in the literature considering its coverage (landscape vs. single solution analysis), completeness (evaluated aspects), and representativeness. Our work is the only updated landscape article to cover all aspects of AVs operations.

3.3 AV Goals: Theory & Practice

The importance of studying AV's detection rates is reasonably clear to most people, as they directly affect the system's protection. However, the importance of studying AV's internals is sometimes overlooked, as they only indirectly affect security. Despite that, good implementation choices are essential to guarantee detection capabilities: For instance, a previous study showed that abusing AV internals leads AV's solutions to crash [74].

There are two key concepts to understand AV's internals: (i) the attack surface, and (ii) the Trusted Code Base (TCB) [34]. The first refers to the fact that the more exposure a system and/or application has, the greater the possibility of it being targeted, exploited, or vulnerable to any other event. The more services and/or components an application presents, there are more alternatives to a successful attack. The second refers to the fact that any component added to a software (e.g., library, module, so on) needs to be trusted by the main application. These concepts are strongly related, as each component added to the TCB increases the attack surface.

When AVs are added to systems, they increase the TCB of that system. Thus, the addition of the AV software themselves initially increases the attack surface of that system, as the AV adds libraries, modules, interact with subsystems, so on. Under the light of the presented concepts, an AV is only viable if the benefits of adding the AV as part of the TCB of a system is greater than the attack surface added by it.

The general goal of an AV is to reduce the system's attack surfaces by making them less exposed and exploitable. This can be done, for instance, by leveraging AV capabilities to sandbox applications [178]. However, this is not what happens in practice when the AV's internals fail to accomplish their goals.

There are multiple reports of AV failures and many of them are related to an increased attack surface. A typical failure case is related to format parsers. AVs implement parsers by themselves for multiple file protocols. Parsing is a very error-prone task and the security implications are giant if the errors happen inside an AV engine [17]. Besides parsing, another frequent AV task is to unpack protected code. In addition to error-prone, this task is also risky because in many cases the packed code needs to be executed within the AV. Bad decisions about unpacking routines might lead to a significant increase on the attack surface. When the unpacking is performed in kernel [154], a userland threat is elevated by the AV itself to a kernel threat. Privilege escalation by AVs can only be prevented by a careful design of their internals. Unfortunately, attacks are still seen in practice, such as in the case of a rootkit remover that in fact allowed unsafe drivers to be loaded in the kernel [58].

Recently AVs extended their inspection capabilities to cover other scenarios, such as web threats. As in previous cases, whereas increasing defenses, they also increase the attack surface. This might lead to unintended consequences. For instance, an attempt to inject a `Javascript` verification code in web pages to protect users ended up disclosing unique tokens that allowed tracking users over websites [78].

AVs also often intercept network communications to protect users against malicious downloads and data exfiltration. AVs usually set local proxies to the system to redirect traffic via the AV inspector. These proxies might even intercept encrypted traffic, which leads to privacy concerns [65]. Even worse, the proxies themselves might be attacked if they are not properly

implemented. Recently, an AV proxy was revealed vulnerable to `Freak` attacks [81]. Face to the presented scenario, in this work, we also evaluate AVs under the light of their attack surface.

3.4 Detection Mechanisms & Operation Modes

AVs have been reported for a long time as solutions that detect samples via signatures when on-demand checks are requested. This is far from an accurate description of a current AV. They have evolved to cover multiple attack surfaces and operate on distinct modes. The AV might be operating in multiple modes simultaneously, as defined by the AV policy. In many cases, these modes are progressively activated during system operation. In other cases, however, some modes might only be available in premium products, also according to AVs vendor's policies. According to our observations, AVs operate in the following modes:

- **On-demand Checks.** These are the typical checks performed when users request specific files to be checked. This type of scan is useful to detect malicious patterns that were not visible when the file was created and thus inspected by the other components operating in other modes.
- **Scheduled Checks.** These are a variation of on-demand checks that is activated only in predefined times aiming to scan the whole system. This type of check is often performed in the background and/or when the system is idle.
- **Real-time Checks.** These modules continuously inspect running processes' interactions with other OS components to find suspicious behaviors and immediately blocking threats. When this mode is enabled, performance overhead is imposed to the system as the processes' actions need to be tracked and intercepted by the AV.
- **Trigger-based Checks.** This mode executes inspection routines as soon as a specific action occurs in the system. For instance, AVs inspect executable files as soon as they are written on disk (e.g., downloaded from the Internet), or when they are about to be executed (e.g., after a double click).
- **Delayed Checks.** AVs might also perform additional checks in delayed periods of time when an inspected artifact (file and/or process) is not reported as clean with high-level confidence. The AV might use this additional time to wait for the process to exhibit more characteristics to be inspected or to request to an external party (e.g., cloud server) additional information about the file. Some AVs rely on collective information, such as those obtained via telemetry systems, to make their decisions.

One should not confuse these presented operating modes with the types of checks performed in each one of them. In the malware field, analysis (and detection) procedures are often classified as static and dynamic [164]. Therefore, AVs might present a combination of the following detection strategies:

- **Statically Triggered Checks,** When the scan was requested by the user (e.g., on-demand and/or scheduled scan modes).
  - **Static Detection.** This type of detection occurs without running the suspicious artifact. It is characterized, for instance, by the use of signatures and pattern matching techniques against static files. This is the most commonly used scan technique when an on-demand check is requested.
  - **Dynamic Detection.** This type of detection occurs when the suspicious artifact is executed to be scanned. Many AVs do not limit their on-demand checks to signatures, but in fact they run the suspicious binary in a sandbox to check its behavior before allowing it to execute in the main system. For instance, we found that the AVAST's `Sf2.dll` library implements a Dynamic Binary Instrumentation (DBI) solution for that purpose.
- **Dynamically Triggered Checks.** When the checks are triggered by the runtime monitors. These checks are dynamically triggered as they rely on the fact that the suspicious artifact is running.
  - **Static Detection.** Although these monitor rely on running artifacts, the detection method employed by a real-time monitor might be static. A file system filter might, for instance, detect a file creation in real time but launch a pattern matching procedure to detect it as malicious.
  - **Dynamic Detection.** These are the checks performed in the context of the running processes. AVs often monitor APIs arguments to detect suspicious actions as soon as they are started by the processes.

The presented operation modes and detection methods cover the following OS attack surfaces:

- **File System Scans.** The AV monitors the file system to inspect newly created and/or modified files. Files are the typically AV-inspected artifacts due to malware sample's persistence needs.
- **Process Scans.** The AV tracks processes interactions to establish relations between them. This allows tracking child processes of malware loaders and identify injection attacks via remote thread creations.

- **Memory Scan.** Some AVs (e.g., ClamWin [53]) are able to apply detection rules against loaded processes images. This allows detecting emerging threats, such as fileless malware. This type of inspection imposes significant performance penalties due to the memory access latency. Therefore, it is more common to find memory inspection in the on-demand operation mode than in the real-time mode.

- **Network Inspection.** AVs currently cover network-based threats since the Internet has become massively popular. To do so, AVs set proxies in the system to inspect the application's traffic. Whereas some applications such as browsers are almost always inspected, the proxy for other applications is often just a pass-through filter.

- **Browser Protection.** AVs have been increasingly adding inspection capabilities directly into the browsers. They are able to inspect network traffic and the loaded page's contents. The typical implementation of an AV's browser monitor is by leveraging the browser's plugins and extension facilities.

3.5 Understanding AV Structure

We previously presented the multiple operation modes and attack surfaces covered by the multiple AV components. We now detail these components, how they interact with each other, and the impact of potential flaws in each one of them.

Figure 1 presents an overview of the most common AV's components and their interactions. In an overall manner, AV's components interact in a client-server way [106]. However, depending on the perspective of the task at hand, the understanding of what is a client and a server might change.

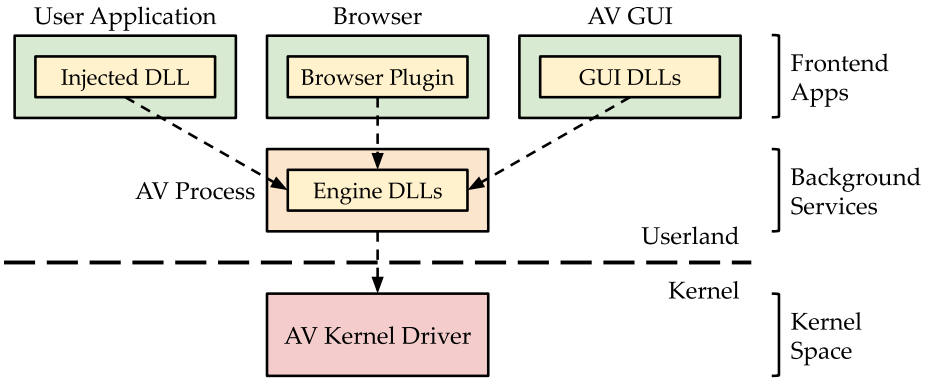

Figure 1: **AV Architecture.** Overview and main components.

When a user claims to have interacted with an AV, in fact, he/she interacted with a Graphic User Interface (GUI) application (a client) that just set parameters for the AV core running in another process (a server) that effectively adds threat intelligence to the system. Whereas the GUI is implemented as a typical user process, the server usually runs as a background service, with elevated privileges and sandboxed interactions. Therefore, whereas the GUI can be terminated and restarted by the user (or any application), the AV core should be resistant to termination to not be finished by a malware sample running in userland. The communication between the GUI mechanisms and the AV core is often performed via JSON or XML data sent and received via HTTP-like protocols. This allows clients built upon distinct frameworks (e.g., Windows binaries, Web-based applications) to communicate with the AV core.

The AV core is not a monolithic piece of software, but usually a host process that loads within its libraries that effectively implement the AV inspection capabilities (e.g., pattern matching, unpacking, so on). Tables 19 to 24 from Appendix A shows the multiple libraries used by the distinct AVs. In this sense, the AV core process is a client of the detection results provided by the libraries. This architecture opens space for attacks if one were able to load the AV core libraries within any process to inspect it and find ways to defeat it. Therefore, AVs should implement methods to prevent the loading of these libraries in external processes and/or to authenticate the communication with them. These protection mechanisms are described in Section 7.

Whereas some libraries are of AV's exclusive use, some libraries are designed to be injected into running processes to monitor them. These libraries provide information to the AV core processes (a client for this type of information, but a server of detection results) that decides what to do with this application (e.g., process termination if malicious). Unlike the previous case, the challenge here is to avoid the library being unloaded by a malicious process to evade detection.

Although the AV core processes run with administrator privileges, some information can only be obtained in the kernel space (e.g., reading foreign memory, I/O ports, so on). Therefore, AVs implement one or multiple kernel drivers to interact with and collect additional data to decide about the maliciousness of a given artifact. From the I/O point-of-view, the kernel serves the AV core client with data. As in the library's case, the kernel driver should be protected from attacks. The AV should ensure that the driver is not unloaded by third parties to reduce AV's inspection capabilities. The AV also should ensure that a third-party will not use the driver to elevate its privileges. For instance, the AV should authenticate the communication with the driver to avoid a third-party process to request the AV driver to read protected memory regions and thus disclose sensitive data via unprivileged IOCTLs.

Similar reasoning can be applied to AV's network clients and proxies. As their ports are openly available in the system to be connected by any processes, they should ensure that they only establish a connection with trusted entities, such as the AV entities. Otherwise, these clients might disclose sensitive information to any process that queries their state via these network ports.

4 Definitions & Methodology

In this section, we define our study object and describe the methodology to inspect it.

4.1 Definitions

Before presenting the strategies adopted to inspect the AVs, we first present a definition of the AV objects studied in this work: AV internals and AV engines.

AV Internals. We consider AV internals all components of an AV product that are not directly exposed to the user, including the AV engine, modules, libraries, databases, and configuration files.

AV Engine. We consider as the AV engine the modules implementing the functions responsible for detecting and removing malware. The AV engine is the core of an AV product and its working is, theoretically, independent of marketing issues—Non-functional AV features, such as for personal and enterprise versions, should not affect the AV engine operation.

4.2 Methodology

This work's goal is to shed light on AV's internals from a practical point of view. We are concerned whether the concepts reported in the literature are actually deployed by the off-the-shelf solutions. Therefore, to present a landscape of AV's implementations, we analyzed AVs regarding all their operation steps, from (i) installation; through (ii) scanning; until (iii) post-installation updates.

Our study aims broadness, thus we evaluated AVs for Windows, Linux, and Android. However, we pay special attention to the Windows OS because it is usually the most targeted OS by malware samples [14]. We analyzed the set of the 10 most popular Windows AVs ranked according to the AVTest's criteria [23] (checked in August/2019). All AVs but the built-in Windows Defender were evaluated from the installers downloaded from the AV vendor's websites. Freeware AVs were installed with their full capabilities and commercial AVs were installed in their trial versions. The installers are described in Table 3.

Table 3: **Analyzed AVs.**

AV	Version	MD5
Avast	19.7.4674.0	172ee63bf3e0fa54abd656193d225013
AVG	19.8.4793.0	0d19e6fc1a4d239e02117f174d00d024
BitDefender	24.0.14.74	0e54eab75c8fd4059f3e97f771c737de
F-Secure	21.05.103.0	2393777281f3a9b11832558f5f3c0bce
Kaspersky	20.0.14.1085	7dc4fb6f026f9713dca49fc1941b22ce
MalwareBytes	3.0.0.199	9c69b2a22080c53521c6e88bd99686a1
Norton	22.17.1.50	2f1f762658dc7e41ecc66abd0270df97
TrendMicro	12.0	f8b8a3701ec53c7e716cf5008fad9aa1
Vipre	11.0.4.2	77a9dbd31ed5ebe490011ffa139afe03
WinDefender	4.18.1902.5	Built-in W10

Since AVs encompass multiple subsystems (e.g., filesystem, network, processes, drivers, libraries, databases, so on), we have to conduct distinct analysis at the distinct steps of AVs operation to understand their whole operation. Each time a module was in action, we conducted a distinct investigation procedure to consider the the most interesting targets for that module. Table 4 summarizes and exemplifies the targets for each AV execution step.

Our analyses encompassed both static and dynamic procedures performed using distinct tools. The tools considered for the overall AV characterization are summarized in Table 5 (we present other, specific-purpose tools over the text when required). We performed static procedures to identify the files deployed by the AV in customer's machines. It included enumerating all executable binaries, kernel drivers, and libraries, along with their imports and exports. The drawback of this type of procedure is that although we can identify some key AV engine components, we cannot identify how they interact with other components nor when their capabilities are triggered. Thus, we performed dynamic analysis procedures to bridge this gap. The dynamic inspection consisted of actually interacting with the AV software and triggering multiple tasks, from scans to update procedures. We traced all AV's components, both from userland as well as from the kernel, during our interactive analysis sessions.

In addition to characterizing AV's typical operations, we also simulated some adversarial conditions for AV operations to assess their self-protection capabilities. For instance, we (i) impersonated AV's clients by loading their DLLs inside our controlled

Table 4: **Analysis Methodology.** Distinct aspects are checked according to the AV operation step.

Operation Step	Analysis Target	Operation Step	Analysis Target
Installation	File Identification	**On-demand scans**	Analysis Threads
	Installer Tracing		Scan Parameters
	Downloaded Files		Cache Databases
	Anti-Tampering Checks		Scan Routines
AV Loading	Created Processes	**Runtime Checking**	Process Creation
	Created Services		DLL Injection
	Loaded Drivers		Kernel Callbacks
	Configuration Files		DLL Unload Prevention
	Checksums and Self-Checks		Process Termination Prevention
Updates	Network Traffic	**Cloud Scans**	Hash Generation
	File Replacement		Network Traffic

Table 5: **Analysis Tools.** Summary.

Task	Tool
File Characterization	peid [8] + pefile [66]
Strings Identification	Strings (built-in)
Instruction Checking	objdump (binutils)
DLL Enumeration	DLL Export Viewer [145]
Driver Enumeration	DriverView [146]
Process Enumeration	ProcessHacker [153]
Hook Identification	HookShark [83]
Registry Inspection	Regshot [159]
Filesystem Checks	FileGrab [69]
Userland Tracing	SysInternals [129]
Kernel Tracing	Branch Monitor [31]
IOCTL fuzzing	Custom Solution
Network Inspection	Tcpdump [176] + mitmproxy [135]

processes to verify if they accept third-part commands; (ii) developed our own `IOCTL` fuzzer to verify whether their drivers answer to third-party requests; and (iii) deployed Man-In-The-Middle attacks to check whether AV's communication can be tampered or not.

We searched for Linux AVs similarly as we searched Windows ones. However, testing Linux AVs has been revealed as a harder task than testing other platform's AVs, mainly because fewer commercial solutions are targeting Linux. Many solutions are tied to single platforms and/or available only for enterprise customers (which is not the case for this study). Most of the AV versions we had access were not functional. Their installation processes can be considered as still undeveloped face to the current scenario of installers for other platforms, such as for Windows. For instance, we found installers that still do not automatically solve missing dependencies problems. Considering the above, we were able to inspect a fully-functional version of the **ESET** AV for Linux Desktops. In this scenario, our analyses were more focused on showing the differences from a real Linux AV to real Windows ones, since similar components were discussed in details for the Windows ones.

Whereas the Linux environment is characterized by a limited number of AV solutions, the Android ecosystem presents the opposite characteristic: it has a myriad of AVs and other security-related apps, such that they would deserve a specific research work to be fully analyzed. However, since our goal is not to provide an exhaustive analysis of Android AVs, which is left for future work, but to draw a landscape of their distinctions to the desktop AVs, we limited our evaluation to the top-5 most popular apps in the Google Play Store in July/2020 (apps versions are shown in Table 6). Most of the analysis procedure in the Android environment consisted of statically inspecting the distributed applications. In this case, the dynamic analysis procedure should be understood as the act of running the application in the device such that databases are populated. These databases were further retrieved and inspected offline.

Table 6: **Mobile AVs.** Tested Versions.

AV	Avast	AVG	Psafe	Kaspersky	ESET	AVIRA
Version	6.29.1	6.29.2	6.5.1	11.50.4.3277	5.4.13	6.7.2

AVs cannot be evaluated by themselves; they need to be exposed to malware samples to exhibit their defensive capabilities, Moreover, AVs react distinctly to distinct samples. Thus, we collected multiple malware samples and submitted them to AV scans during the monitoring to be able to observe AVs in action. For this study, we collected samples from Virustotal [186], Malshare [115], VxUnderground [189], and from a partner security company that opted to remain anonymous. In total, we considered 9M PE samples (among which 5K are kernel rootkits), 5K ELF samples, and 5K Android samples. We did not

balance these datasets in any way, since our goal is not to characterize the samples but the AVs. The datasets were not submitted as a whole for the AVs, but samples were individually tested until the AV exhibits the behavior we were interested in. We confirmed all collected samples as being malicious by submitting them to the Virustotal service. This service is mentioned all over this work whenever we need a great confidence level for an analysis procedure, which is provided by the Virustotal's AV committee.

In the next sections, we present our AV evaluation broken down by the distinct tasks performed by the AVs. We opted to present the results according to the performed tasks because it allows us to better describe specific AV's aspects that would remain hidden if mixed among the myriad of tasks performed by modern AVs. It also allowed us to perform the analyses with a greater focus, without being distracted by side operations. Even though, the analysis process has been revealed challenging because multiple of these tasks happened during the process. For instance, it was hard to distinguish the tasks performed by the multiple AV processes when update procedures were triggered along with file scans. We did our best to isolate such cases and expect to present the most accurate description possible of each AV component's dues.

Our research work is guided by two main analysis goals: broadness and correctness. Therefore, whenever possible, we present results covering all the AVs to present a broad panorama of AV operations. In a few cases, we focus our description on specific AV products. This ensures that we only report results for which we have a high confidence level on the outcomes of the analysis processes. This prevents us from reporting to the reader wrong results due to obfuscated code constructions that could not be fully interpreted[1].

5 Antiviruses Anatomy

In this section, we discuss the multiple components of actual AV products and the project decisions behind their implementations.

5.1 AV Ecosystem

Analyzing AVs is a double-edged sword. On the one hand, they are very particular solutions. Each company deploys distinct policies and the analysts that produced sample information are different. Therefore, analyses can hardly be generalized. On the other hand, AV engines are not so different in structure as they have to fit the same OS constraints. Therefore, we here aim to present a landscape of these common aspects.

In practice, the market of AV engines is not as broad as the AV's solution market itself. This happens because many solutions share the same engines (e.g., licensed versions of a major AV company engine). There are even companies specialized in selling detection engines instead of selling their own AV solution. Also, most of the main AV companies provide Software Development Kits (SDK) to their products [168]. These are often adopted by newcomers since creating an engine from the beginning is tough [143].

Face to this engine sharing scenario, one can still identify research work falling to significant pitfalls, such as referring to the number of AV solutions that detected a given sample as a confidence level on its maliciousness without considering that many of those detections occurred due to the usage of the same engine. These repeated detection reports do not add extra information about a given sample's maliciousness because all of them are repetitions of the same procedures leveraged by the shared engine.

We can have a long-term view of how the AV engines sharing evolved by looking to common labels present in the *VirusTotal* service [186]. The labels assigned by two AVs usually agree when they are originated from the same engine. We relied on this fact to cluster similar labels and identify AVs sharing their engines. We considered the set of 9M samples described in Section 4 for this experiment and that two engines are the same when their labels agree on at least 70% of all cases.

Figure 2 shows the clusters and their agreement rates for the AVs identified as sharing engines. This approach was able to identify real cases of engine sharing, such as ZoneAlarm outsourcing detection to the Kaspersky cloud [198] and the AVG's acquisition by Avast [18]. In the latter, the approach is even able to show the cooperation evolution: In 2016, when the agreement was announced, the two AVs were not clustered together. In the following year, the AVs started being clustered together, with a lower rate than in the last years, when the AVs are likely fully integrated. This label correlation was then observed in other research work [197].

When analyzing the AV's binaries, we discovered two cases that reflect the aforementioned integrations. More specifically, we discovered that: (i) Avast and AVG finished their integration, with the same core files (same hashes) distributed for the two Avs; and (ii) the VIPRE AV uses the `Active Threat Control` driver and the `scan.dll` library from Bitdefender and the `WebExaminer` driver from ThreatTrack, which is the root of many label's similarity.

5.2 Security Resources Integration

Face to the above-presented scenario, AV solutions tend to differ more due to the modules that are integrated into them. Each AV architecture defines how the modules are integrated. In some cases, information from the multiple modules might be correlated. However, the presence of a module in a given application should not be seem like a definitive indicator of the AV's operation mode. In a noticeable example, Google embedded the ESET NOD32 AV in its Chrome browser but instead of verifying web pages, the AV in fact checks the filesystem for artifacts potentially harmful to the browsing environment [151].

[1] We aimed to report results of marketed AVs whenever possible. Otherwise, the open-source AV ClamAV [52] is used as example

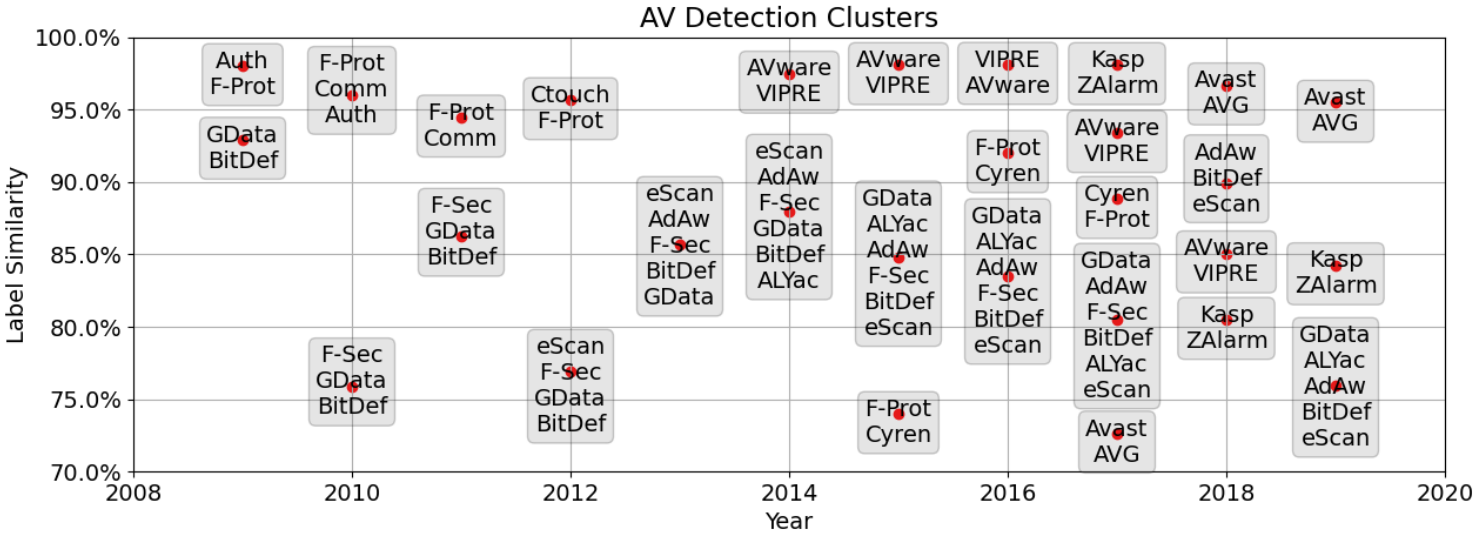

Figure 2: **Engine Sharing.** Identified clusters according to VirusTotal's labels sharing.

Table 7: **AV Resources.** A multitude of security resources is available in current AV solutions.

AV	Avast	F-Secure	Kaspersky	TrendMicro	VIPRE
Firewall	✓			✓	✓
Network Inspector	✓	✓	✓	✓	✓
Antispam			✓		✓
Secure Browser	✓				
Browser Protection	✓		✓	✓	
Real-Time Monitor	✓	✓	✓	✓	✓
Emulator	✓				
Safe Deletion				✓	✓
Safe Banking			✓		
Safe Search				✓	
Email Protection			✓	✓	✓
Social Protection				✓	✓
Password Manager	✓			✓	

The OS attack surfaces covered by the AVs should also not be confused with the threats that the AV aims to protect against. Current AVs are not only composed of detectors to suspicious executable files, but they also cover other security aspects. Table 7 summarizes the multiple components found in the AVs. We notice that current AVs are complete security solutions. They offer facilities such as filtering spam, acting as a firewall, sweeping files definitely, and even protecting emerging surfaces such as social networks. In the TrendMicro AV, we even found a protection tool (`TmopphYmsg.dll`) aimed to protect the deprecated Yahoo Messenger. Therefore, the AV agents deployed through the OS stack serve multiple security purposes.

5.3 AV's Implementation.

The fact that AV engines might be shared among distinct AV solutions highlights the importance of understanding and taking care of the development of these engines. On the one hand, the process of developing an AV engine does not differ significantly from developing any other software. The same project decisions adopted by popular solutions can also be found in the AVs. For instance, AV configurations are stored in SQLite databases, as in many popular projects. We found SQLite adapters for Avast (`aswSqLt.dll`), F-Secure (`sqlite3_32.dll`), and Kaspersky (`dblite.dll`) AVs.

However, as AVs are complex and critical pieces of software, they must follow the best development practices. For instance, their code is modular, with multiple helper functions and polymorphic implementations to support 32 and 64-bit systems and legacy standards. Interestingly, in the Kaspersky AV, we can even find a library referencing the Façade Design Pattern (`cf_mgmt_facade.dll`). Whereas there is no evidence that its code is implemented following this design pattern, this component does interface with other system components [101]. Interfaces are popular AV's components, as the AVs need to interact with multiple distinct subsystems.

A factor that complicates AV's development is that AVs cannot rely on the security of third party libraries and thus need to implement their own routines even for the most popular algorithms. For instance, in the Kaspersky AV, we found implementations of the MD5 (`hashmd5.dll`) and SHA1 (`hashsha1.dll`) hash algorithms. This project decision is essential to ensure that the AV is not considering a file as benign because the hash algorithm was also infected and subverted.

5.4 AV Installation & Removal

The first step to understand AVs is to observe their installation, as it reveals which are the components that they install and to which system components they interact with. A previous study shows that developing a secure application installer might be challeging [28], and we understand that this also applies to AVs as they need to ensure that they were correctly installed to properly protect users against attacks.

To understand how AV's installers work, we traced their installation in virtual machines. All AVs were successfully installed and did not require rebooting the system to finish. Even though, some components, such as extensions to third party applications, required the host application to be restarted. The AV files were not packed (although some of them are distributed in proprietary formats), which allowed us to inspect them. Table 8 summarizes the most installed components by AVs. Whereas EXEs and DLLs files were expected to be found, due to the software installation nature, we highlight the installation of XPI files (browser extensions) performed by most AVs. We also identified distinct signature files used to ensure file authenticity and integrity distributed via multiple file formats (e.g., `XML`, `TXT`, `SIG`, so on).

Table 8: **AVs Installers.** Mostly Installed File Types and Components.

AV	EXE	DLL	SYS	XPI	Certificates	databases
Avast	✓	✓	✓	✓	✓	
AVG	✓	✓	✓	✓		
BitDefender	✓	✓	✓	✓	✓	✓
F-Secure	✓	✓	✓	✓	✓	✓
Kaspersky	✓	✓	✓	✓		
MalwareBytes						
TrendMicro	✓	✓	✓	✓	✓	✓
VIPRE	✓	✓	✓		✓	

A key task for any installer is to ensure that the correct files are installed. Table 9 summarizes how the files are retrieved and verified. Most AVs opt to distribute online installers, that download the AV files from the Internet. Few AVs distribute standalone installers containing all installation files. An advantage of online installers is that they allow AVs to always install the most updated AV versions in the target machine.

To check the installer's robustness, we attempted to tamper the AV installers by adding bytes to these files to change their checksum and check whether they implement verification routines. We discovered that only the Norton AV checks the installer integrity. Most AVs opt to perform post-installation checks. In the case of online installers, they do not have to worry about file tampering at the installer level as the files are downloaded from the Internet and thus cannot be tampered locally. In turn, the files could be tampered during the download process if it is performed via non-encrypted (HTTP-only) connections. Whereas some AVs such as Kaspersky opt to download data via HTTPS connection from hardcoded IP addresses (not even DNS requests are performed to avoid hijacking), other AVs, such as Avast, opt to traffic data in clear. In fact, this is an interesting project

decision taken by many AVs. This was reported in previous studies [28] and was hypothesized to be due to legacy compatibility. Due to this decision, post-installation checks must be performed. Back to the Avast case, we confirmed that the AV performs post-download checks to confirm the file integrity and legitimacy.

Table 9: **Installers Summary.** Installers Types and Protection Mechanisms.

AV	Installer type	Installer Integrity Check	Encrypted Traffic
Avast	Online	✗	✗
AVG	Online	✗	✗
BitDefender	Online	✗	✓
F-Secure	Standalone	✗	✓
Kaspersky	Online	✗	✓
MalwareBytes	Standalone	✗	N/A
Norton	Online	✓	✓
TrendMicro	Standalone	✗	N/A
VIPRE	Hybrid	✗	✗

After AV modules are ready for use, AVs should register them in the Windows Security Center (WSC), an OS component that ensures there is an AV running in the system. The most recent Windows versions are shipped with a built-in AV, Windows Defender, such that the new AV should be registered in WSC so as Windows can safely deactivate the Defender AV and allows the new one to take control of the system. All evaluated AVs properly registrated themselves along WSC.

The Default Settings Problem. Another important aspect of an installer is that it defines default configurations for the AV operation. These configurations are not customized for user's specific needs and might not provide the best protection if they are not reviewed by the users. Default settings should be also be observed when performing comparisons and evaluations of AVs, as comparing two AVs operating with distinct features is unfair. In the Kaspersky AV, for instance, whereas cloud-based scans are implemented, it is not available by default. Acknowledging this issue is important because evaluations with and without cloud support will certainly lead to distinct results. Similarly, whereas the MalwareBytes AV provides a large set of configurations, including performance restrictions, its rootkit protection is not enabled by default. Acknowledging these configuration possibilities is important because performance measurements with and without detection restrictions will certainly lead to distinct results. For Avast, whereas real-time protection is enabled by default, firewall and sandboxes are disabled. Even components enabled by default need to be configured. For instance, although the ransomware protection is enabled by default, its default coverage is limited to a few user folders instead of operating system-wide.

It is important to highlight that the default configuration settings affect even the AV's detection rates. As already pointed by the literature [109]: "*Default configurations can sometimes leave systems less secure than recommended when adding them to a production network.*". In practice, the detection rates achieved by the AVs are bounded by the configured AV's sensibility. AVs present distinct sensibility levels, as well as most security solutions [166]. More specifically, the evaluated AVs present 3 distinct sensibility levels: low, medium, and high. Some detection capabilities are only available in the highest sensibility level. All evaluated AVs were shipped configured in the medium sensibility level by default, which reduces the FP rate and the performance overhead, but also limits the detection capabilities. Taking the Avast AV as an example, in this mode, the AV: (i) do not scan entire files, but only some parts (e.g., headers and chunks); (ii) do not follow links; (iii) do not scan removable media; (iv) skip scan for some known file extensions; and (iv) do not scan non-popular compressed files. These detection capabilities become available to the user if he/she configures a custom scan.

We consider that the issues related to default configurations are often overlooked in practice, although some aspects were described in the literature [139]. Thus, our goal is to present a real-world evaluation of AVs and their impact. To do so, all overview experiments and results presented in this work were performed using the default AV configuration. We expect to overcome popular claims about AV's detection capabilities that cannot be supported by empirical observations. More specifically, we believe that, as a general rule, if a security mechanism is not practical to be deployed by default, it is not an effective and efficient solution.

AV Removal. If AV installation procedures are poorly understood, AV removal procedures are completely obscure in most cases, which motivates our report. Although these procedures might worth an entire investigation, we here shed light on two key aspects of AV removal: detection and performance. When the AV license expires, the AV is not removed, it remains installed but their component's capabilities are limited (e.g., users cannot trigger on-demand scans anymore). Such limitations, however, does not imply that the AV is completely inactive. In fact, the components responsible for protecting the AV from tampering attempts (which includes attempts to tamper with the AV license, in this case) are still functional, thus the AV is still imposing performance overhead even in an expired state. We noticed that for the AVs in which the same kernel drivers are responsible for anti-tampering and runtime threat detection routines, the AVs might still detect some threats in real-time, even though their warnings are hidden from the user. Despite not completely unprotected, AV capabilities are significantly reduced when expired. In the past, when it happened, the system was left vulnerable. In recent Windows versions, as the AV license expiration is communicated by the AV to the WSC, Windows automatically re-enable the default Defender AV to protect the users.

5.5 Update System

Updating an AV is an essential security task to keep users protected against emerging threats (although a significant number of users neglect this aspect [112]). Whereas many (academic and industry) works claim that updating an AV is important, practical aspects such as how the update is delivered and how often it is performed are often overlooked. They are critical factors because if an update is delivered in an insecure manner it can be abused by attackers to defeat the AV solution. Therefore, in this section, we shed some light on the practical aspects of AV's update systems.

The first thing we should observe about AV updates is that current AVs perform two types of updates: application updates and malware detection updates. The first is performed to add new software functionalities to the AV and/or to migrate it to a new version. The second is performed to add new detection strategies and/or to fine-tune detection parameters using the already-deployed detection mechanisms. The difference between them should be highlighted because these two operations have significant differences, both in their frequency as well in their file sizes.

When we look at the updates from a file size perspective, software updates are large, with multiple MB, reaching up to 100MB in one of our observations. In turn, malware definitions are usually individually small, rarely exceeding an MB. However, as these definitions need to configure multiple, individual components, multiple of these definition files are downloaded each time, with their combined size reaching a couple of MB per update. The update size significantly varies over time, with the updates performed in some days presenting larger files than others, according to the distinct AV solutions. We were not able to compute a file size average for the AVs with statistical confidence.

When we look at the updates from a frequency perspective, software updates are "rare", occurring when new AV versions are available or when bugs are found. We observed these to occur from once a week to once a month. In turn, malware definition updates are more frequent, though dependent on the AV company's ability to generate new detection rules. In our experiments, we observed AV checks for malware definition updates ranging from every 3 minutes (Avast) to 30 minutes (VIPRE).

Although the update checking time is a good indicator of how fast updates are expected, we can only fully understand AVs vendor's capabilities in delivering new malware detection settings when we look to the actual updates performed by the AV. There is currently a paucity of studies in this field. To the best of our knowledge, the only work that presented statistics about updates was a 6-month observation performed by an AV comparative company dating back 2004 [1]. This study presented key results to characterize AV's operations, such as the heterogeneity of the updating process among the AVs, but this work needs to be updated to confirm or disprove these results when considering a modern AV.

Unfortunately, we do not have the same structure as an AV comparative company to perform a 6-month observation for all AVs. However, we were able to deploy a single AV for 30 days on a real user machine connected to the Internet 24h a day to observe all their updates occurring as soon as they are made available by the AV company. We expect that the obtained results could be extrapolated somehow to other AVs solutions and/or at least partially update our knowledge and statistics about the process of updating a modern AV.

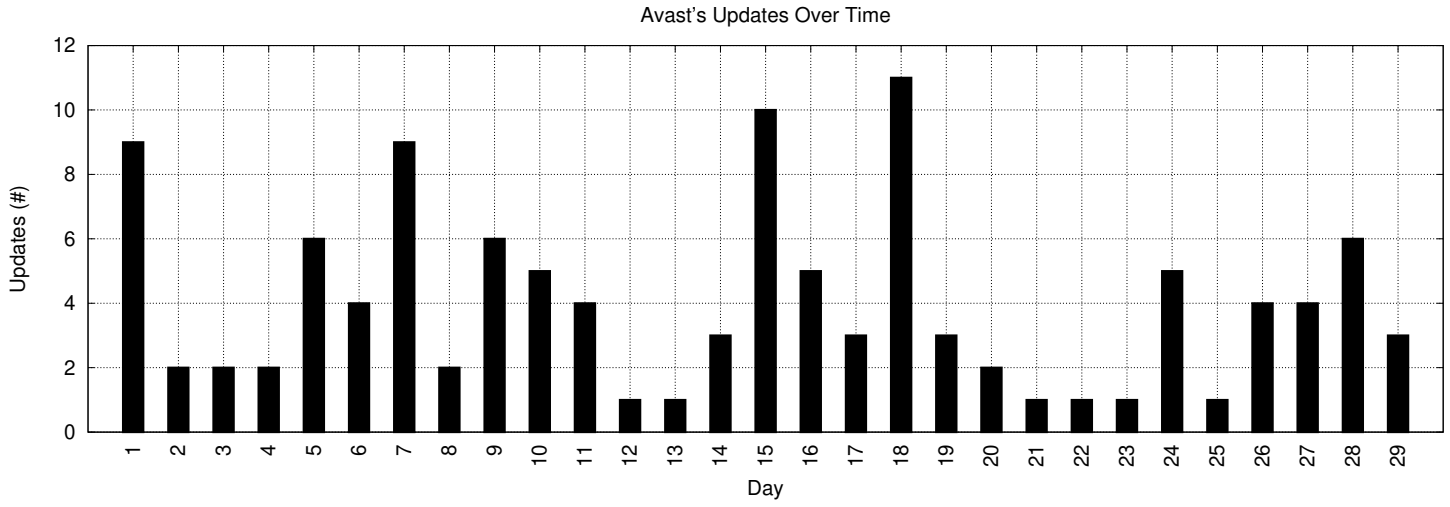

Figure 3: **Avast's updates over time.** The number of updates per day significantly varies over time.

Figure 3 shows the daily updating frequency for the Avast AV (chose at random for this test). We notice that at least 1 update occur every day. This shows that updates cannot be neglect in AV evaluations and/or even in academic developments, as one would be not paying attention to a process that happens every day. More than that, we observe that a distinct number of updates is performed each day, which extends the heterogeneity of AV updates reported in previous work from inter-AV to intra-AV observations. Finally, we observe that AVs might perform up to 10 updates in a given day. This reveals that the update policy of this AV is to deploy malware detection routines as soon as possible instead of consolidating all-new detection routines in a single update. This decision is a good strategy to keep users protected as soon as possible.

Despite occurring multiple times a day, AV's update processes are complex and require multiple steps. We following detail the update process of some of the investigated solutions.

Avast. In this AV, the update process starts with the AV querying its own servers for updates (e.g., `http://r0965026.vps18.u.avcdn.r`
This communication is performed via an unencrypted channel, as already identified in previous work [28]. Upon the download of the update files, AVs should check their legitimacy to avoid content tampering, which is eased in Avast's case due to the use of an unencrypted channel. For this task, Avast relies on the DSA algorithm to check the signature of each downloaded file.

The update files are delivered to the AV as VPX files, an Avast proprietary format. These files might deliver a new software component or new detection routines. According to our understanding, the VPX file is structured as shown in Code 1. The header stores the path and filename of the file to be updated with the content of this file. It also stores the version of this file, thus avoiding AV downgrades. When a software update is delivered, the data section directly stores a PE binary. The whole file is signed and the signing information is stored at the end of the file.

```
typedef VPX {
    typedef header {
        char filename[];
        int offset;
        int version;
    }
    typedef blob data[bytes];
    typedef signature {
        typedef hashes;
        typedef signatures;
        typedef certificates;
    }
}
```

Code 1: Avast's VPX file structure.

The delivery of new malware detection capabilities is performed via VPS files, whose structure, according to our understanding, is shown in Code 2. As for previous cases, the whole content is signed and verified before the actual update.

```
typedef VPS {
    typedef MAGIC_BOF = {"ASU!VPSz"};
    typedef blob data[bytes];
    typedef signature { ...
    typedef MAGIC_EOF = {"ASU!VPSz"};
}
```

Code 2: Avast's VPS file structure.

A key task for AVs is to ensure that their continuous operation, which challenges software updates. For instance, AVs should not be disrupted by unsuccessful updates (which was already demonstrated possible in the past [133]). To prevent such occurrences, Avast backups the files to be updated before their replacement. This allows the AV to recover the old configurations to remain operating if the new files lead to a crash. Due to this characteristic, the AV does not directly modify an existing file, but first creates a temporary file with the updated content and further moves it to the new destination. For instance, in our experiments, the file created at `C:\Windows\system32\drivers\asw7836f650432f0780.tmp` was further moved to `C:\Windows\system32\drivers\aswbidsdriver.sys`. Due to the AV's need to continuously operate, file modifications are not performed using ordinary API calls, but as filesystem transactions [126]. By making use of transaction APIs, the AV can rely on OS capabilities to ensure file integrity, concurrency control, and, in the last instance, that the transaction fails gracefully and the file is reverted to the previous, correct state.

The update of malware definitions is easier to be performed than the software update one. In this case, the AV creates a new folder to store all extracted files. Upon all files are stored there, the AV creates a malware definition database file (`C:\Program Files\Avast Software\Avast\setup\vps.def` that points to the recently created folder (the most recent definitions).

To keep track of this whole, complex process, all update steps are logged to a file (`C:\ProgramData\Avast Software\Persistent Data\Avast\Logs\Update.log`. Since this log file can grow significantly, the AV adopts a log rotation policy to store only the most recent update's data. In our tests, we identified that the AV log file is typically about 4MB of data, which corresponds to the last 30 days of updates.

MalwareBytes. The operation of this AV follows the same steps as the aforementioned one, with a few distinct implementation decisions. The first distinct implementation decision is observed right at the beginning of the update process when the AV servers are contacted by the host. MalwareBytes relies on third-party cloud servers (`ec2-52-54-175-12.compute-1.amazonaws.com` and `server-13-32-81-124.mia3.r.cloudfront.net`) to deliver their updates (via encrypted connections) instead of using their own servers. The second difference is observed in the delivered payload: Instead of a custom file format, such as VPX, this AV distributes updates via 7z files. This is a very interesting project decision to allow component reuse since the same engine used to extract 7z files for inspection can be used to extract the update files. Before replacing any file, the original files are backuped. For instance, the `C:\ProgramData\Malwarebytes\MBAMService\config\AeConfig.json` is backuped into the `C:\ProgramData\Malwarebytes\MBAMService\config\AeConfig.json.bak`.

VIPRE. This AV's operation is very similar to the previous ones. The updates are retrieved from a CDN (`map2.hwcdn.net`) via an encrypted connection in a gzip format (e.g., `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1602525813_PENDING\versi` to be further extracted. Before replacing files, the original files are backuped in multiple formats. For instance, the `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1602525813_PENDING\licences.cfg` file is backuped into `C:\Program Files (x86)\VI` `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1602525813_PENDING\licences.cfg.bak2`, and `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1602525813_PENDING\licences.cfg.gzip`. The new files are extracted in the malware definition folder (e.g., `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1599672068\heur.cfg.upd`). The whole update process is logged in the `\ProgramData\VIPRE\Logs\SBAMSvcLog.csv` log file.

5.6 On-demand Checks

Whereas some people still think that all scans are equal, the AV's reality is that scans vary a lot. When a user requests an AV scan for a given file and/or directory, the AV does not simply perform a pattern matching against the file. Instead, the AV follows a complex series of detection steps, as here described.

A generic on-demand pipeline starts by the AV reading the scan configuration files, which defines which detection routines will be performed. The AV loads the correct inspection modules in runtime upon parsing these configuration files. Then, the AV checks if the file really needs to be scanned, according to the multiple AV policies. A typical scan routine is launched if the file really needs to be scanned. If nothing is found in this case, the file can be analyzed in an emulated environment.

Emulated environments execute the suspicious file in an AV-provided sandbox for a limited amount of time to check for any Indicator of Compromise (IoC). There are multiple trade-offs and drawbacks to be discussed when implementing this kind of solution. However, as most of them were discussed in previous papers [26, 39], we limit ourselves here to present complementary information. In particular, we describe the emulator found in the Avast AV, which was not covered in these previous work.

Avast. The on-demand operation of the Avast AV starts with the reading of the `avast5.ini` file (detailed in Code 3 of the Appendix D). This file defines how scans are performed by setting which will be scanned and/or skipped, and which type of heuristic checks will be performed. The AV can, for instance, enable and/or disable packer detection, and/or code emulation. Research-wise, it is interesting to see how the AV has a fine-grained configuration level but do not expose this to the user, which ends up preventing AV comparatives to be performed in a more fair way [29].

After the AV is configured, it checks if the given payload needs to be scanned, which is performed by querying a set of `sqlite3` databases. If the payload is a file, the `C:\ProgramData\Avast Software\Avast\FileInfo2.db` (Figure 14 of the Appendix D) is queried. If the payload is an URL, the `C:\ProgramData\Avast Software\Avast\URL.db` (Figure 15 of the Appendix D) is queried. These databases store important information from previous scans. For instance, for each file (identified via their sha256 sums), the database stores when the last scan was performed. It allows the AV to compare this information with the file's last modification date and skip the scan if the file was already scanned after being modified. This database is not populated for every file in the device, but acts as a cache. The last cleanup field indicates when the data in the database was rotated, as in a typical log policy.

The AV does not instantly launch a local scanning procedure after it decides that the file really needs to be scanned. First, it queries the file reputation in the AV server (`filerep-prod-011.mia1.ff.avast.com`). If the file is known to be malicious at this point, the file is reported and the verification is finished.

The AV launches a local scanning procedure in the cases where no reputation information is available for the file. In this case, the AV starts by loading its malware detection capabilities (e.g., signatures, heuristics, so on) by reading the `Software\Avast\defs\aswdefs.ini` file. After that, the matching procedures are performed (see Section 5.9).

Finally, if the file was not detected using the previous approaches, the AV might run the payload in an "emulator" to inspect it dynamically. We noticed that this type of detection method is not triggered all the times, but we were not able to identify which is the triggering criteria. The AVAST "emulator" is, in fact, a Dynamic Binary Instrumentation (DBI) tool implemented by the `Sf2.dll`. The DBI solution is implemented by AVAST and seems to not rely on third-party components. It exports functions such as `StartInstrumentation` and `SelfInjectionPoint` that can be used to instrument the application in which this library is injected into. Most of the library's capabilities are only revealed in runtime. Its entry-point function performs recursive calls until setting methods such as `OnAPITraceChunkAPITracer`, `OnBeforeEmulationEndMachine`, and `OnLoadingModuleModuleManager` that can be used to trace applications at distinct levels.

TrendMicro. The operation of the TrendMicro AV is very similar to the presented for Avast. The AV starts reading its configuration from a file (`C:\Program Files\Trend Micro\AMSP\system_config.cfg`). Based on the configured routines, the proper modules are loaded. Objects are not immediately scanned, which only happens after a check to the `C:\ProgramData\Trend Micro\AMSP\data\10009\MBG.db` (shown in Figure 17). This is a sqlite3 database that acts as a scan cache.

VIPRE. The operation of the VIPRE AV is very similar, with the database of cached scans being placed in the `smartdbv2.dat` and `smartmd5cache.dat` files.

Other AVs. Although presenting similar characteristics with the aforementioned AVs, we were not able to fully characterize the operation of the remaining AVs, such that we opted to not discuss them in details in this section.

5.7 Signatures

Signatures were the first detection method employed by AVs to detect known samples. Over time, signatures were considered less attractive due to their significant drawbacks to detect malware variants and 0-days. These tasks are better performed by

Machine Learning (ML)-based detectors, for instance. This resulted in a pitfall often repeated by many people that current AVs do not use signatures anymore. However, signatures cannot be simply discarded by AVs since signatures are still the fastest way to respond to incidents caused by recently-uncovered threats (1-day attacks). Therefore, in practice, we can still find evidence of the application of signatures to counter malware. In many cases, users can even identify when an AV mistakenly identifies a text file as malicious due to the byte patterns present on it [2].

To shed some light on the use of signatures, we started our investigation on the use of signatures by the AVs by searching for strings related to the EICAR test file [63], as the AVs are required to detect this file for compliance with AV testing procedures. We found clear references to the EICAR file in the core files of the Avast, AVG, BitDefender, FSecure, Kaspersky, and TrendMicro. In the Avast's `algo64.dll` library, the full EICAR pattern was present, which suggests that an explicit byte-comparison is performed to detect this file. For the remaining AVs, the EICAR file seems to be treated as any other detection rule, which suggests that the AV engines have implemented byte-based pattern matching mechanisms to be able to detect this type of signature file. BitDefender, Kaspersky, and VIPRE AVs were able to detect the EICAR pattern at distinct file offsets, such as when appended and/or prepended to other files.

Once we confirmed that AVs indeed implement signature matching mechanisms, it is interesting to take a look at how these are implemented. Signatures can be implemented in multiple ways [4], but nobody is completely sure about which of these approaches are deployed in commercial AVs. To bridge this gap, we searched for the presence of known pattern matching mechanisms. For some AVs, we found references to the YARA [187] pattern matcher. For the Avast, where no direct reference is available in any file, memory dumps of running Avast processes present references to symbols (`RuleIsSilent@CYaraHelper` and `Scan@CYaraHelper`) that suggests that a wrapper for the framework is loaded in memory in runtime. Similarly, the Norton AV presents references to resources named `yarac`, which seems to be related to compiled YARA rules. Finally, the Trend Micro AV explicitly imports YARA rules. In addition to multiple references over the binaries, we were able to found even a debug print stating that the AV would: "*Begin to use yara to make a decision!*."

Another common AV pitfall is to consider that signatures are only byte-based, which is not true. Modern signature schemes are more like detection recipes, i.e., a series of steps that must be performed to trigger a detection warning. These steps might rely on distinct AV capabilities, as shown in Section 5.9, and also include byte patterns. For instance, it is common for a signature to require the scanned payload to be first unpacked, then a given section to be deobfuscated, for then applying a byte pattern matching against it. This allows AVs to be more precise and filter out false positives. This type of filtering can be seen even on most Yara rules released by security companies [160]. A frequently observed filtering criterion is to check if the scanned file starts with the `MZ` flag, thus indicating that the file is a Windows PE file. If it is not, the pattern matching procedure is skipped. AVs also implement this same filtering criterion. We observed that in practice in all AVs by patching the `MZ` bytes of previously-detected files and realized that the detection rules were not triggered anymore.

Once we gathered evidence that AVs indeed rely on signature matching for their detection procedures, we designed an experiment to quantitatively evaluate the importance of signatures for AV detection. We repeatedly submitted the PE samples described in Section 4 to AV scanning procedures while individually patching their distinct code sections. We assume that if a sample stopped being detected if-and-only-if when a specific section is patched, this is due to the use of signatures by a given AV to detect that specific sample. If the sample remains being detected even if their sections are individually patched, we considered that the detection occurs due to other mechanisms (e.g., header checking, ML detections, heuristics, so on). This experiment was repeated to all AVs present in the Virustotal service.

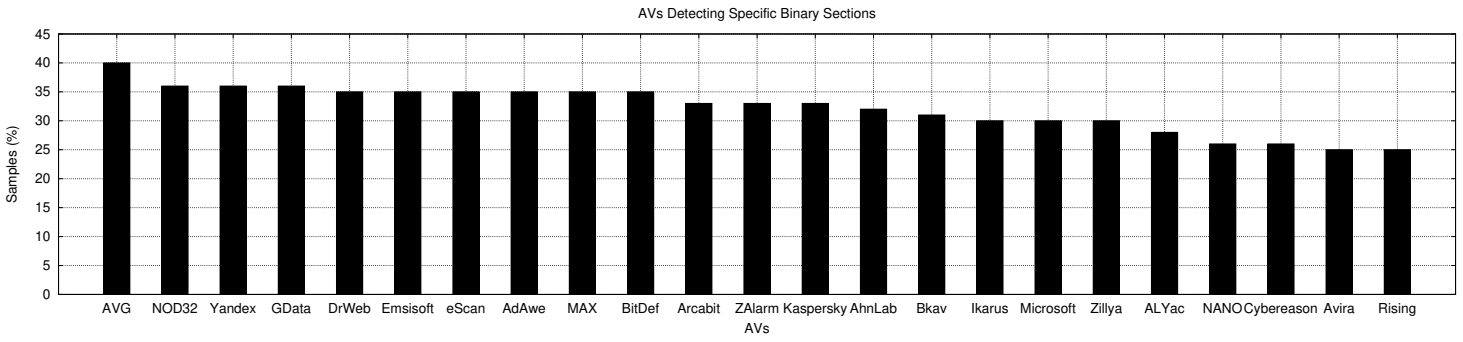

Figure 4: **Signature Prevalence.** Around a third of the AV's detections are based on specific section's contents.

Figure 4 shows this experiment's results for the AVs that detected all tested unmodified samples, thus mitigating detection biases. We discovered that around a third of all samples are detected via signatures. The rate is consistent among all AVs, varying from 25% to 40%. This shows that signatures cannot be discarded as a significant detection method for real AVs.

A side-effect of the presented experiment is that it ends up showing the sections in which signatures were applied by the AVs, as shown in Figure 5. Our first observation is that signatures are applied against all sections, which is compatible both with the expectation of position-unaware, byte-base pattern matching methods, as well as with detection recipes that check multiple sections. The prevalence of detected sections depends on how frequently given sections appear in the considered samples. As

[2]We present an example of this case in the video available at `https://www.youtube.com/watch?v=aKXiupiplbk`

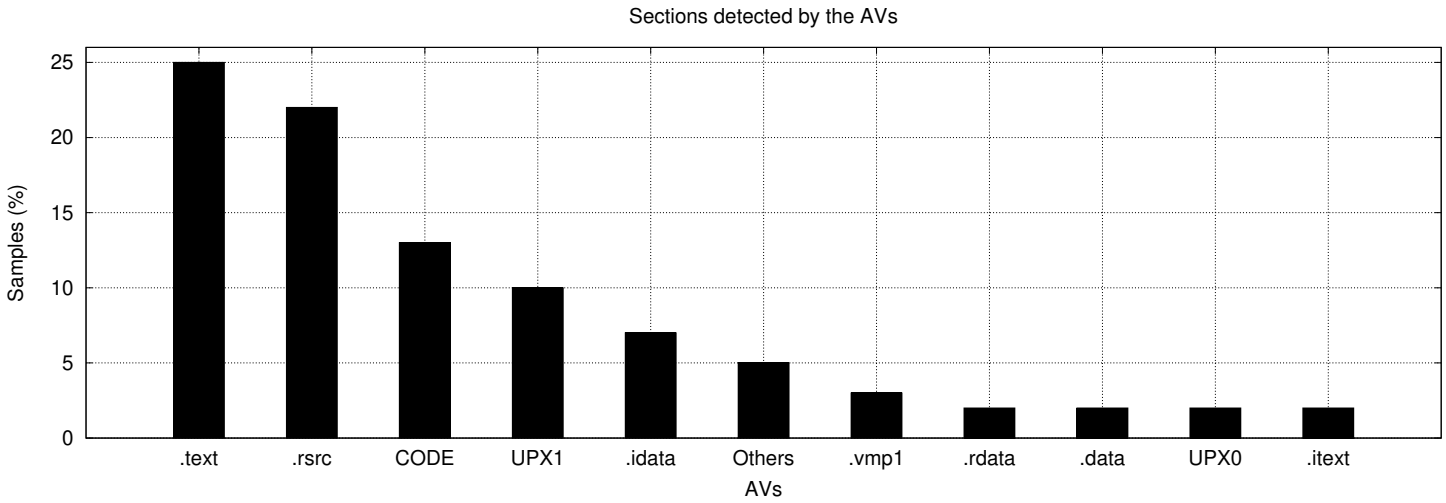

Figure 5: **Sections detected by the AVs.** Sections in which the specific payloads detected by the AVs are located.

expected, the `.text` section is the most detected since the malicious constructions are placed there in the form of instructions. The resource section is the second most detected one since it might embed malicious payloads. Interestingly, the `vmp` and `upx` are also flagged, which shows that many AVs might still use the presence of a packer as a proxy for malware detection instead of checking the actual file content.

A critical factor to develop a signature is its size: Short signatures will likely result in False Positives; Larger signatures are slower to match and require significant storage when millions of them are combined in a single database. Despite their importance, there is not a guideline for signature size definition, which makes researchers propose signature schemes in an ad-hoc manner. There is also little public information about the signature sizes really employed by the AV solutions.

Currently, we know that ClamAV signatures are on average 28 byte-long [64], which results in a database file of 112 MB [52] to store all its million signatures. However, this does not seem to be a standard, as the ESET AV has been reported to consider signatures up to 60KB [67]. To draw a landscape of the real signature size considered in marketed AV solutions, we deployed a methodology to extract the byte patterns used as signatures by the AVs on a large-scale dataset.

ALGORITHM 1: Candidate Signature Extraction Algorithm,

Data: Binary, Section, Start, End
Result: Candidate Signature
`/* Patch first half */`
1 upper = patch(binary,Section,Start,(Start+End)/2)
`/* Patch second half */`
2 lower = patch(binary,Section,(Start+End)/2, End)
`/* If only upper is detected, the signature is in the other part */`
3 **if** *only_ detected(upper)* **then**
4 return sig_extraction(Binary,Section,(Start+End)/2,End)
`/* If only lower is detected, the signature is in the other part */`
5 **if** *only_ detected(lower)* **then**
6 return sig_extraction(Binary,Section,Start, (Start+End)/2)
`/* If both or none is detected, no signatures */`
7 return NOT_FOUND

A common strategy to extract signatures from files is to split the file into multiple, smaller snippets and identify which one remains detected by the AV. This strategy was employed in previous literature work in many variations [192]. For our experiment, we opted to implement an alternative version of the `AVwhy` [59] tool. More specifically, we implemented a divide-and-conquer approach that at each iteration patches half of the considered binary snippet, as in a binary search algorithm, and uploads the patched binary to Virustotal for scanning (see Algorithm 1). We consider as the signature the unique, smallest sequence of non-patched bytes that makes the binary still be recognized as malicious by a given AV ((see Algorithm 2)).

Figure 6 exemplifies the operation of our algorithm when inputted with an originally-detected malware binary having two sections (1 and 2). The algorithm starts by independently patching Section 1 (**step 1**) and Section 2 (**step 2**). The algorithm concludes that the AV signature is not present in the first section because the binary remained detected despite the patch. In turn, a signature must be present in the second section because the AV stopped detecting the patched file as malicious. The algorithm then proceeds to refine the signature size identification by repeating the patching procedure now only with the two components of the second section (**steps 3** and **4**). Similarly, the algorithm concludes that a signature is present on the second

ALGORITHM 2: Signature Identification Algorithm,

```
   Data: Binary
   Result: Detection Signature
   /* Candidate Signatures                                                    */
1  Sigs = []
   /* Consider all sections                                                   */
2  for sections in binary.sections do
3      Sigs.add(extract_sig(binary,section,START,END)
   /* A signature is confirmed if a single candidate is found                 */
4  if len(Sigs)==1 then
5      return Sigs[0]
6  return NOT_FOUND
```

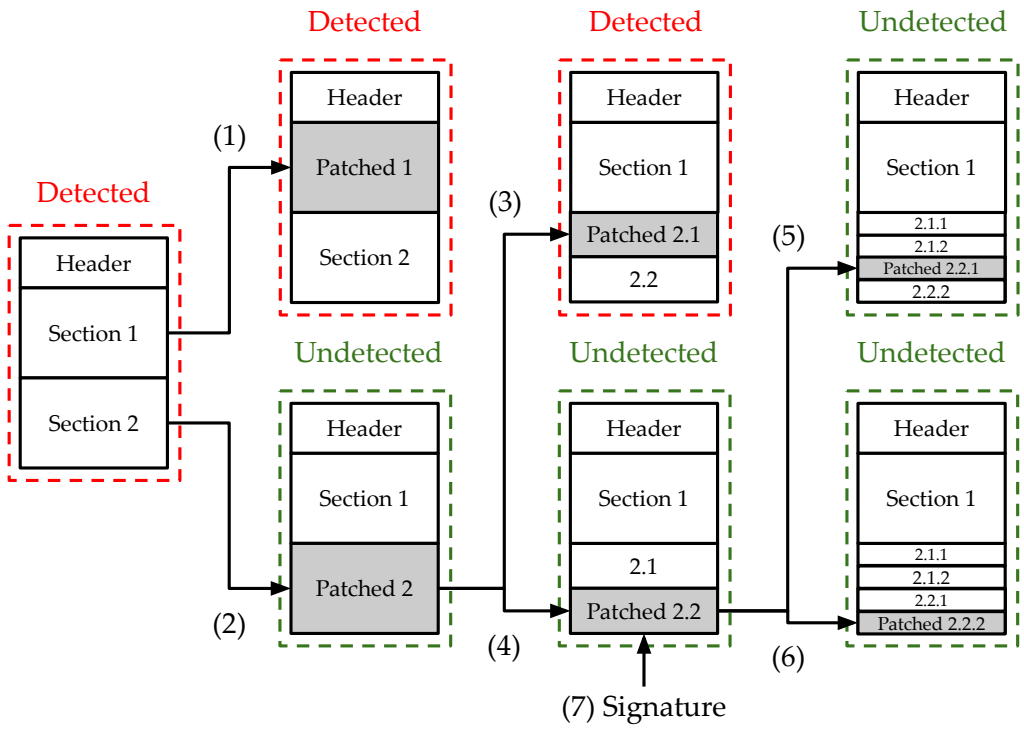

Figure 6: **Binary Search-Like Signature Identification.** Distinct patches are applied until the smallest required snippet is identified.

part of the patch and advances towards refining the patch size. However, in the last steps (**5** and **6**), the algorithm fails to refine the patch size because both patched files were not detected anymore. The algorithm then considers the last valid patch (obtained in **step 4**) as the most likely signature (**step 7**).

Previously, a similar approach to ours was used to identify the signatures used in practice by the Windows Defender AV [117]. We are aware that our approach is only limited to identify byte-based signatures and will not capture heuristic behaviors, but we still consider this approach interesting to reveal how byte-based signatures are used in practice.

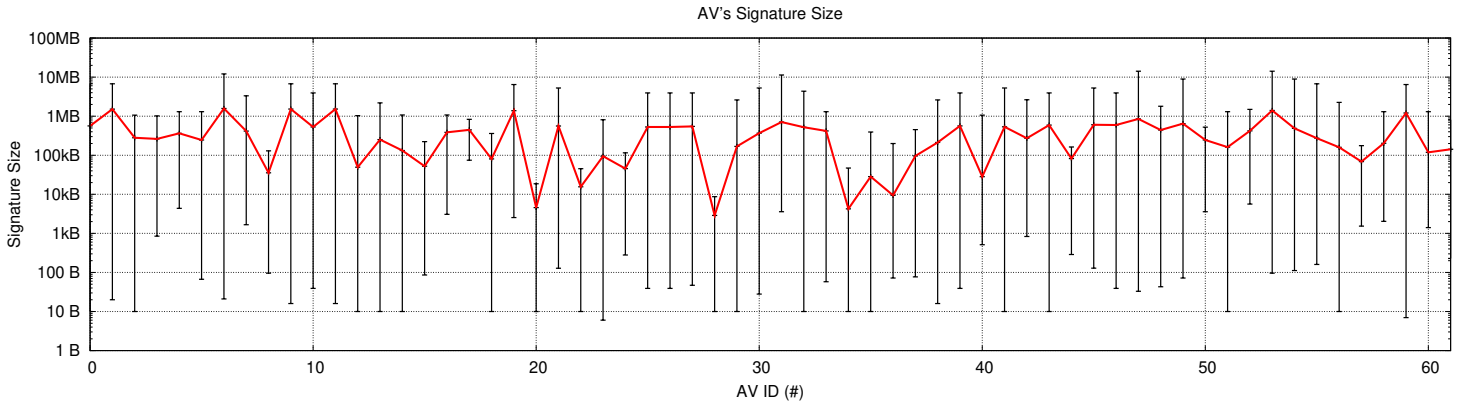

Figure 7: **Signature Size.** Whereas the average signature size is between 100KB and 1MB, minimum and maximum sizes may vary significantly.

Figure 7 shows the minimum, maximum, and average signature sizes for the multiple AVs present in the Virustotal service (represented by an ID). We first notice that a plausible explanation for the lack of guidelines for AV signature size definition is that there is no pattern that fits the reality. In practice, the identified signature sizes for all AVs presented a great variation. Almost all signatures fit in the interval between 10KB and 1MB, with a prevalence in the 100KB-1MB interval. Most AVs presented small signatures (e.g., 10B-long), which we credit to the search of specific patterns within specific sections (e.g., the search of the PE header in the resource section to identify embedded payloads). Some AVs also presented very large signatures (MB-long), which we credit not to a long byte signature itself, but to the expansion of regular expressions in the form of `prefix*suffix`, thus covering a large set of bytes.

5.8 Monitor's Implementation

A key part of an AV engine is the monitoring component, as it collects that data that will be analyzed by the intelligence component that judges whether an artifact is malicious or not. A failure in capturing data might result in detection evasion in the case where the intelligence component does not have enough data to make a decision. Given its importance, in this section, we delve into details about how monitors are implemented. Real-time AVs have two design choices for the implementation of an event data collector: (i) hooking APIs at userland, or (ii) monitoring events from the kernel. Each one has its pros and cons, as following discussed.

5.8.1 Userland Hooks

Hooking at userland is advantageous for real-time AVs in comparison to kernel-based monitors as userland hooks enable data collection without the overhead of diving into the kernel, with API granularity, and affecting only the monitored process. The major drawback of this choice is that the hooking API can be unloaded by the monitored process, and/or the hook can be detected and defeated, which requires extra AV protection. Face to this trade-off, most AVs opt to implement userland hooks.

Understanding how hooks are implemented is important to provide supporting information for the development of newer AV engines. Many research works propose API-call based detection mechanism based on the hypothesis that DLLs can be injected into any process and that any API function can be hooked. In practice, however, DLL injection even in benign processes might lead to crashes [11] and due to that some apps protect themselves from being monitored [38]. Moreover, only a subset of all existing API functions are hooked by the AVs due to multiple reasons (e.g., complexity increase and/or performance degradation). A correct evaluation of whether current models fit into reality can only be conducted having knowing the APIs functions hooked. We following present the identified hooked functions by each AV according to our analysis procedures.

Avast hooks system API functions by injecting the `C:\\Program Files\\AVAST Software\\Avast\\x86\\aswhook.dll` into the running processes. This DLL hooks the set of functions shown in Table 25 of the Appendix B. Avast hooks a limited set of functions (17 distinct functions from 2 distinct system libraries) that cover only explicit actions (e.g., `LoadDLL`) instead of indirect actions, such as DLL injection (e.g., `CreateRemoteThread`). This shows that complex detection models proposed in the literature to hook hundreds of functions would not completely fit in the actual operation model of this AV.

AVG shares the detection engine with AVG, thus it works by injecting the same library (now placed at `C:\\Program Files\\AVG\\Avas`
into running processes. The same previously presented API functions are hooked.

Bitdefender hooks system functions by loading the `C:\\Program Files\\Bitdefender\\Bitdefender Security\\atcuf\\26437514` DLL into running processes. The DLL is delivered using a custom packer and extracts itself in memory. This DLL hooks the set of functions shown in Table 26 of the Appendix B. Bitdefender is the AV that hooked the largest set of API calls (132 distinct functions from 11 distinct system libraries), supporting direct and indirect events. Thus, it is compatible with more complex real-time detection models. The AV hooks even into cryptography functions, likely to proactively defend the system against ransomware attacks.
Vipre monitors the running processes by injecting the `C:\\Program Files (x86)\\VIPRE\\Definitions\\aap\_core\\1.19.176.0\` library into them. This library is signed by BitDefender and unpacks from the same addresses as the previously presented BitDefender library. In fact, the installed hooks, shown in Table 27 of the Appendix B, are a subset of the hooks installed by the original BitDefender AV (45 distinct functions from 3 distinct libraries), thus suggesting that the VIPRE AV uses an alternative version of the BitDefender engine.
F-Secure monitors processes by injecting them with the `C:\\Program Files (x86)\\F-Secure\\SAFE\\Ultralight\\ulcore\\1576` library. Table 28 of the Appendix B shows that this library hooks a small subset of all API functions (17 distinct functions from 4 distinct libraries), similar to Avast does. As a noticeable difference, this AV worries about detecting privilege escalation attempts via the loading of kernel drivers, as can be inferred by the monitoring of the `services` subsystem.
Kaspersky monitors running processes by injecting them the library `C:\\System32\\klhkum.dll`. This library has a jump table-like construction that points to an obfuscated function that derives hooks for the original system functions. We were not able to identify a general rule for the hook installation.
Malware Bytes. Whereas most AVs opted to implement their own code hooking solutions, the most noticeable characteristic of MalwareBytes is that it relies on a third-party solution for this task. The presence of debug symbols (`\Users\Patxi\Documents\Malware` reveals the use of the `madcodehook` framework [158].
WindowsDefender We skipped the analysis of this AV as the Windows Defender AV has been previously analyzed [39]
Other AVs. We found no userland hooks for the remaining AVs. It does not imply that they do not hook API functions, but only that they were not detected by the considered hook detection tools (distinct research work reported distinct libraries and functions hooked in distinct AVs [57, 141]). Alternatively, these AVs might be leveraging kernel drivers for monitoring purposes [155], as following discussed.

5.8.2 Kernel Monitors

AVs do not monitor the system only from the userland but also from the kernel. Operating from kernel brings the advantage of protecting AVs from subversion by userland malware. In turn, drivers are more complex pieces of code to be developed, they can't rely on a wide range of libraries, and should be signed to be loaded by the OS.

From an AV perspective, kernel drivers are used for three tasks: (i) to deploy callbacks to collect data in a privileged manner, which allows, for instance, monitoring the file system in a wide manner and thus potentially detect ransomware due to intense filesystem activity; (ii) to attach to process to receive the same signals and interrupts that the process receives, which allows implementing, for instance, keylogging protection mechanisms by receiving the keys pressed in the context of a protected process; and (iii) to load an inspection mechanism at boot time (Early Launch Anti-Malware–ELAM), which aims to inspect the system before the loading of the malware.

We analyzed all AVs and found drivers implementing all these three functionalities. Each AV deploys multiple drivers but, in an overall manner, all AVs rely on almost the same OS callbacks, focusing on monitor processes creation and filesystem activity. Few drivers implemented callbacks for the Windows registry. Although the OS provides mechanisms for sharing data between drivers, AVs opted for each one of their drivers to reimplement all callbacks for each driver, likely due to performance reasons. The multiple AV modules need the same information, mostly processes and threads IDs, because these are used to reference detection tables and to whitelist processes operations.

The callback implementation for most AVs is very similar. Most of the data collected in the callback functions is queued on Deferred Procedure Calls (DPCs) to be analyzed out-of-band, without blocking the process execution. An exception to this rule is when the AV has active components that online check and block specific actions by making the callback to return an error code. To speed up the performance, the AVs implement caches for the collected information. In the specific case of file system monitoring, as I/O routines are dispatched in batches, it is very likely that the same objects are referenced in consecutive callbacks (e.g., file create, file open, and file write, for instance). Therefore, to avoid retrieving OS information about each artifact (e.g., owner ID, paths, tokens, permissions, so on) every time the callback is invoked, the data retrieved in the first callback is cached for further accesses. This design decision is essential to mitigate the performance overhead of interrupting the process execution for a long time inside a callback routine.
Avast. This AV implements 14 drivers that cover distinct attack surfaces, as shown in Table 29 of Appendix C. It monitors a wide range of system resources, including rootkits and keyloggers. Its drivers include not only monitoring mechanisms, but also a self-protection mechanism against termination.
AVG. This AV deploys the same drivers as the Avast AV. It also ships additional Microsoft drivers for compatibility, such as a `cdfs` driver to read CDROMs.
BitDefender. This AV deploys 5 distinct drivers, as shown in Table 30 of Appendix C. This AV seems to make a design decision to move a significant part of its detection capabilities to the userland, given the significant difference on the hooked functions at userland to the number of drivers and callbacks implemented at kernel.

F-Secure. This AV deploys 4 drivers, as shown in Table 31, being the one which implemented fewer callbacks. The AV is clearly modularized, with each one of the drivers responsible to monitor a subsystem independently.
Kaspersky. This AV deploys 22 distinct drivers, as shown in Table 32, It also ships Microsoft drivers for compatibility and an OpenVPN driver. It covers multiple attack surfaces, protecting from rootkits and key- and mouse-loggers. It also implements anti-tampering mechanisms.
MalwareBytes. This AV deploys 7 drivers, as shown in Table 33. it includes an ELAM driver. Most of the detection capabilities are centralized in the `swiss-army` driver.
Norton. This AV implements 10 drivers, as shown in Table 34. It includes an ELAM filter, which is basically a reimplementation of the other modules but targeting the operation in this specific context.
Trend Micro. This AV deploys 10 distinct drivers, as shown in Table 35. It covers multiple attack surfaces, with special attention to boot and OS startup.
VIPRE. This AV implements 5 distinct drivers, as shown in Table 36 of Appendix C. It includes two third-party drivers: the ATC driver from BitDefender, already presented, and the Activity Monitor from ThreatAttack.
WinDefender. We skipped the analysis of this AV as the drivers of this AV are mixed with OS drivers, which makes them hard to be distinguished. In total, this AV references more than 400 distinct Windows drivers.

5.9 Detection Routines

Whereas many think about AV detection as a single process, in fact, it has many steps, each one with their own challenges and drawbacks. AVs rely on multiple helper functions to perform each one of them, such that understanding them helps us to understand the AV detection process. Thus, we here shed some light on the key features of AV engines.
Deobfuscation. An AV detection routine can be described in a very high level as the process of matching an unknown payload against a known malicious pattern. However, this task is not as straightforward as it might sound when we dig into details. In most binaries, the patterns to be matched will not be clearly displayed, but obfuscated somehow, such that AVs must implement deobfuscation routines to be able to inspect the real payloads.

The strategies used by attackers to obfuscate malware vary significantly, such that AV's vendors perform a cost-benefit analysis to identify which techniques are the most prevalent and worth being addressed by the AVs. Popular techniques used by attackers that are handled by AVs are string manipulation, decoding of base64 payloads, XOR-encoded payloads, and the append of data in files.

However, the support for those routines does not mean that they will be applied all the time and for all files. AV's vendors also have to make decisions about other trade-offs, such as performance, and false positives. According to our observations, these helper functions are mostly used along with detection rules (e.g., signatures) and not in a standalone manner (e.g., to match entire binaries).

Table 10: **Deobfuscation Functions.** Not all techniques are applied to entire payloads.

Technique	**XOR**			**BASE64**			**RC4**			**Embedding/Carving**		
Mode	**Sig.**	**RT**	**OD**	**Sig.**	**RT**	**OD**	**Sig.**	**RT**	**OD**	**Sign.**	**RT**	**OD**
Avast		✗	✗	✓	✗	✓		✗	✗		✗	✗
MalwareBytes		✗	✗	✓	✗	✗		✗	✗		✗	✗
VIPRE		✗	✗	✓	✗	✗		✗	✗		✗	✗
Kaspersky		✗	✗	✓	✓	✓		✗	✗		✗	✗
TrendMicro		✗	✗	✓	✗	✗		✗	✗		✗	✗

Table 10 summarizes the AV operation in the distinct steps and modes—as part of signatures (Sig.), during real-time (RT), or on-demand (OD)—when considering entirely obfuscated payloads using distinct techniques. We notice, on the one hand, that XOR-ed binaries are not decoded by any AV solution. Similarly, AVs also do not reverse RC4-encoded binaries and binaries embedded into other files (pictures, in our experiments). On the other hand, we discovered that some AVs are really able to decode base64-encoded binaries in addition to using base64 in their signatures. The distinction occurs in the step in which the decoding is performed. Whereas in the Kaspersky AV the decode occurs already in the real-time mode, the Avast AV only decodes a base64-encoded binary upon an on-demand scan request.

Avoiding applying all deobfuscation tools to all files reduces AV's performance impact and likely the False Positive (FP) rate, but also opens space for attacks. For instance, whereas a malicious DLL can be detected by an AV in its plain version, it might not be detected when XOR-ed. This might allow an attacker to read the XOR-ed file content to the memory of the DLL loader and XOR it back to a PE file in memory, thus proceeding with the injection procedure. [3].
Unpacking. A special type of obfuscation tool is the so-called packers, executable binaries which embed other binaries within them while applying distinct transformation techniques [162] to protect the original payload from inspection. As for the aforementioned encoding techniques, AVs also have to choose which packers they will support (e.g., either for unpacking or direct inspection), in another trade-off decision. Table 11 summarizes the packers that we identified (via analysis) that are supported by distinct AVs. The absence of a packer for an AV entry does not mean that the AV does not support that packer, only that

[3]We implemented this attack as a proof of concept. A video of the attack is available at `https://www.youtube.com/watch?v=IXVMeRNC_F4`

we were not able to identify the component responsible for handling them, since many AVs implement custom mechanisms for handling packers (e.g., BitDefender's handling of UPX [110]).

Table 11: **AV's Supported Packers.** Not all AVs support the detection of the same packers.

Packer	UPX	Themida	Telock	PeLock	Armadillo	Morphine	VMProtect
Avast	✓	✓	✓	✓	✓	✓	✓
Bitdefender		✓	✓	✓	✓	✓	
Fsecure	✓	✓	✓		✓	✓	
TrendMicro	✓						

AVs are also varied in the way that they handle the packed samples. For instance, consider the case of the UPX packer [177], likely the most popular packing solution these days. We initially hypothesized that some AVs might be embedding the original UPX binary in their code or, at least, embedding part of the original algorithms, since UPX is an open-source solution. However, we did not find evidence of those practices in our observations. Instead, we discovered that each AV implements its own mechanism to detect and handle UPX-packed binaries. Inspecting the TrendMicro `atse64.dll`'s library, for instance, we discovered that the AV looks for the `UPX!` magic bytes within a file to classify it as UPX-packed. Inspecting the FSecure `aeheur.dll` library, we discovered that this AV checks not only the `UPX!` magic, but also the `UPX2`, `UPX1`, and `UPX0` in the section names, increasing the identification confidence. These same names are also checked for Avast. As a significant difference for the previous AV, Avast distributes its detection over multiple components, such as the `algo64.dll`, `aswBoot64.dll`, and `aswEngin.dll` libraries.

Table 12: **Detection of custom UPX packers.** Not all AVs handle UPX-packed binaries without the UPX magic bytes.

Packer	UPX		Custom UPX	
Payload	Goodware	Malware	Goodware	Malware
Avast	✓	●	✓	●
MalwareBytes	✓	●	✓	◓
TrendMicro	✓	●	✓	◓

Although the decision of supporting the standard UPX packer is interesting to fight the most usual malware samples, it does not mean that all UPX-packed files will be detected. Since UPX is open-source, anyone can obtain its code and modify its structure to not display magic numbers and bytes, thus evading the most usual detection solutions [199]. To understand the impact of this strategy, we repeated the scans presented in previous experiments now packing the malicious files with the standard and a custom [90] UPX solution. Table 12 shows that the files packed with the standard UPX packer were all correctly classified, both as goodware and malware. Goodware files were also correctly classified when using the custom UPX packer. This is good news since many previous study reported that AVs have been classifying files as malicious based on their packer and not on their content [182, 3]. However, the good FP rate seems to come at the cost of FNs, since many malware files packed with the custom UPX were not detected as such. According to our analyses, this happens because in the cases when the AV is not able to unpack the malware and reconstruct its IAT imports, it detects the AV only by the visible characteristics in the packed sections (e.g., strings), which significantly reduces AV's detection capabilities.

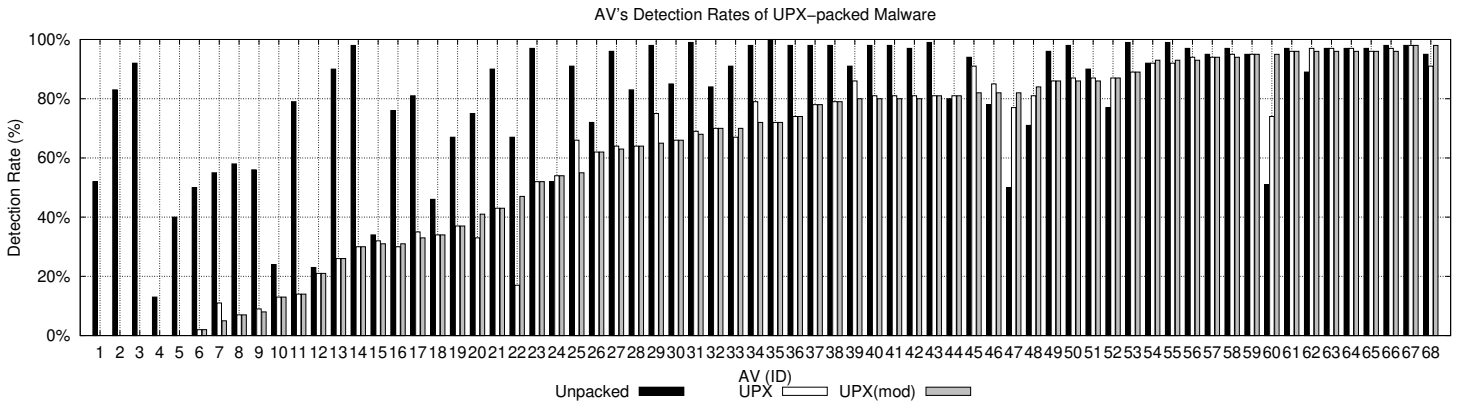

Figure 8: **Detection of UPX-packed Malware.** Distinct AV's implement distinct mechanisms, which leads to distinct detection rates.

To have a broader understanding of the impact of the distinct strategies implemented by the AVs to handle packed binaries, we submitted all the tested samples to the Virustotal service and retrieved detection labels for all the AVs available there. The obtained landscape is presented in Figure 8. We notice that AVs can be classified into three categories: (i) the ones that are not able to handle UPX samples at all, not even the standard version; (ii) the AVs that can handle the standard UPX but are

defeated by the custom modifications; and (iii) the AVs that completely handle UPX binaries, despite any custom modification. Luckily for the users, most AV solutions are placed in the last two categories. More specifically, most AVs are in the second class, such that their security capabilities suffice for catching the most usual threats, even though they might fail to detect more targeted threats developed by more skilled, motivated attackers. It is also interesting to notice that the AVs in the last category presented a greater detection rate for the modified UPX-packed version than for the standard UPX. This suggests that these AVs were able not only to (i) identify that the payload was packed with a modified UPX version, and (ii) unpack it, but (iii) the AVs also used this information as a bias to increase the detection score (a heuristic), which might have influenced in the final detection rates.

The AV that best performed in our Virustotal tests was the WindowsDefender. Although we have not considered this AV in many of the other experiments presented in this paper (see Section 4), we decided to take a specific look at its unpacking capabilities to understand why it was so effective in detecting the modified UPX packer. We discovered that this AV implements a generic and complete unpacker of UPX samples in the `mpengine.dll` library. This AV detects the LZMA compressor used by UPX and decompress it via the `AVUpx30LZMAUnpacker` function. If the content is XOR-ed, it is decoded via the `AVXorDecryptor` function. In many cases, the modification of UPX headers leads to corrupted images (e.g., zero-length section headers). To handle these cases, the AV can fix the binary entry point, via the `UpxEP` function, and even the binary disassembly, via the `AVUpxFixDissasm` function.

The aforementioned result highlights the fact that the malware detection problem is not only a technical issue but also a cost-benefit tradeoff. For instance, to achieve a greater detection rate, this AV also had to spend greater resources (e.g., developer's time, codebase size, testing coverage, architectural complexity) to implement mechanisms to handle constructions whose prevalence might or not justify its deployment.

5.10 AV's Threat Models

A threat model is a key security concept and should be considered in any security evaluation. It defines which, why, and how resources will be protected. More than that, it ends up revealing how one understands a problem. Antiviruses have their own threat models, but these are not often stated clearly. There are multiple implicit assumptions in their operation and understanding them helps to shed light on which aspects of their operation are critical and/or need to be improved.

An often implicit assumption is about their operation in pristine systems, i.e., many AVs assume that they will be installed on a clean system (e.g., right after OS installation). Therefore, the AV will operate by identifying differences from future system states to the original system state in which the AV was installed. We searched all AV's manuals but did not find a clear statement indicating that they suppose a pristine system to operate. Such reference was only found in a web tutorial of Kaspersky AV [97].

On the one hand, assuming pristine systems is reasonable face to the fact that an AV might not properly operate if an infection is taking place, since a malware sample might try to tamper with AV operation. On the other hand, it is not rare to identify users reporting that they installed an AV because they are unsure about the system state [15]. Should AVs protect them anyway?

To identify how AVs behave face this scenario, we compared the detection results of multiples AVs when a dataset of malware is added to the system before and after the AV installation. We discovered that all AVs suggest performing a system-wide scan right after their installation. This scan was able to detect all malicious files that were stored in the filesystem before the AV installation. However, we discovered that the AVs are not able to handle well-running threats started before the AV installation. To investigate this point, we developed an application to simulate an AV killer threat. It monitors the system and automatically owns any directory created with an AV name with exclusive access. It also creates mutexes with the same names used by AV resources. No AV was able to be successfully installed in this scenario, thus showing that the AVs do suppose their installation on clear(er) systems.

Another implicit threat model decision is that AVs will only protect users from threats targeting the same platform that the AV operates (e.g., same OS, same architecture). On the one hand, this is a reasonable assumption, since a malware sample compiled to a distinct platform will not cause harm to the AV running system. On the other hand, in the current world, it is very common to users to transfer files from one platform to another (e.g., download a file on a computer and copy it to a smartphone via USB). Should AVs detect a malware right after the download on the host device or is it entire responsibility of the mobile AV?

Detected File Types. Distinct AVs employs distinct policies for cross-platform threat detection.

FileType	Win		Linux		APK	
Detection Mode	Real Time	On-demand	Real Time	On-demand	Real Time	On-demand
Avast	✓	✓	✗	✓	✗	✓
BitDefender	✓	✓	✓	✓	✗	✗
Kaspersky	✓	✓	✓	✓	✓	✓
MalwareBytes	✓	✓	✗	✗	✗	✗
TrendMicro	✓	✓	✗	✓	✗	✓
VIPRE	✓	✓	✓	✓	✗	✗
ESET-Linux	✓	✓	✓	✓	✗	✗

To understand how current AVs operate in this scenario, we performed cross-platform scans (i.e., Linux AVs to scan Windows

files and vice-versa). The results are summarized in Table 5.10. We discovered that whereas all AVs are able to detect Windows threats both on-demand as well as in real-time, the same is not true for other threat types. For instance, the MalwareBytes AV does not detect samples for any other platform. Other AVs opt to detect only some types of threats. For instance, BitDefender detects ELF threats, but not APKs. Similarly, ESET for Linux AV opted to detect Windows threats but not APKs. Even when the AVs detect all threat types, they do it in different ways: Avast detects only Windows threats in real-time and the other threat types are only detected upon on-demand checks; Kaspersky AV, in turn, detects all threat types in real-time.

5.11 Rootkit detection

A particularly difficult decision when designing AV's threat models is the protection scope. Most solutions detect threats in the userland, thus they can benefit from kernel support to collect privileged information about the running processes. Few Avs also claim to detect kernel threats, such as rootkits. This is a challenging task because the rootkit can interfere with the AV interaction with OS components [6] as it runs in the same privilege level as the AV [161]. Therefore, it is plausible to hypothesize that AV's rootkit detection capabilities are not as effective as their capability of detecting userland threats.

To understand the actual rootkit detection capabilities of the evaluated AVs, we tried to understand in which operation step the detection occurs. We aimed to identify if the detection occurs via patterns when the rootkit files are placed in the filesystem, or via behavioral characterization when they are running. For such, we leveraged the kernel driver rootkits described in Section 4. Table 13 summarizes our findings.

Table 13: **Rootkit Detection.** Most detection is performed by file inspection modules and not by real-time monitors.

AV	**Real Time**	**On-Demand**	**RunTime**
Avast	✓	✓	✗
BitDefender	✓	✓	✗
Kaspersky	✓	✓	✗
MalwareBytes	✗	✓	✗
TrendMicro	✓	✓	✗
VIPRE	✗	✓	✗

We discovered that all AVs detected the malicious kernel drivers via patterns: some of them as soon as they were placed in the filesystem, and some of them upon a requested scan. This shows that AVs have a reasonable rootkit detection capability even without leveraging complex kernel detectors. However, after we modified a set of samples to bypass static detection and successfully loaded the drivers into the kernel, no AV raised a warning about their operation. This shows that the AV operation model is to prevent the rootkit from being loaded, but they cannot do much after they are in place.

While analyzing the AVs we found that Avast was the only AV that presents a module explicitly dedicated to detecting rootkits. It is composed by the `aswArDisk.sys` and the `aswArPot.sys` drivers. The first is a file system filter that exports a `ArDiskRegisterCallback` callback to be used by the second. The latter implements verifications leveraging its high privileged capabilities. For instance, its symbols suggest that it searches for SSDT hooking attempts by looking to the `SystemTable` and `ShadowSystemTable`. We did not fully understand these verification routines. We hypothesize that this protection might be targeting 32-bit Windows, since SSDT patching is already prevented in 64-bit systems. If this is true, verifications should also include other system tables, such as IDT, which can also be hooked.

This module also has functions that perform manual parsing of internal Windows structures (e.g., parsing the Process Environment Block–PEB, and/or the Thread Environment Block–TEB). We found manual parsing associated with the invocation of the `CreateProcess`, `CreateThread`, `GetProcessId`, `GetThreadId`, and `ZwSystemInformation` functions. Since treatment routines for these same functions are present in the userland, this suggests that the AV implements a mechanism similar to a lie detector, checking if the information collected in the kernel is the same presented to the userland. This approach is very interesting because a kernel rootkit might hide artifacts from the userland by hooking functions and performing a DKOM attack [82] but cannot hide these artifacts from the OS.

Despite collecting information at the kernel level, the rootkit protection also relies on userland modules to operate. All information collected by the presented modules is delivered to the `aswAR.dll` library that implements multiple verification routines. For instance, it exports methods for deleting files, registry keys, and service termination, all of them relying on the high-privileged capability of the kernel module. On the one hand, implementing the threat intelligence at userland eases the development process, as the AV can rely on other libraries, reuse code, and so on. On the other hand, this adds exposure to the AV. Since a code is able to escalate to the kernel, it is plausible to hypothesize that this same code is able (and has the permissions) to tamper with the userland module.

To effectively handle kernel rootkits, AV would have to be equipped with modules running in more privileged rings than the kernel (e.g., hypervisors, SMM mode extensions, so on). Whereas these solutions have been widely described in the literature [34], the only real-world solution fully leveraging these capabilities is a specific version of the Kaspersky security solution [103]. Moreover, we are not aware of previous descriptions of these solutions being deployed in the most popular AV versions. We then searched AV for any sign of these components to bridge this understanding gap. We discovered the presence of hypervisors in the Avast and in the Kaspersky AV. Whereas the Avast's `aswVmm.sys` file is clearly described as a hypervisor, the Kaspersky's `klhk.sys` omitted this fact, although it can be identified, for instance, by the presence of Intel VT-X's `vmlaunch` instructions

in its disassembly. When these components are enabled, the whole system is moved to a virtual state under the control of AV's hypervisor. However, this mode is never enabled by default. First, it is only available to premium customers. Second, it might conflict with other software, as reported many times [19, 104]. The major advantage for AVs when operating in these modes is that they have full OS control, even about kernel structs. For instance, AVs are then allowed to hook system tables without kernel noticing. A drawback of this approach is that third party can abuse that to also hook these tables, as already exemplified for both Avast [175] and Kaspersky [89].

5.12 Whitelist

For an AV, properly flagging benign artifacts as unsuspicious is as important as presenting high detection rates, since a solution that impedes users from using their legitimate software (a False Positive–FP) would be fast discarded. A possible solution for mitigating FPs would be for AVs to relax their detection policies, as it is preferable to not detect a sample that is less harmful than blocking a legitimate application that would block thousands or millions of users. This however would leave a fraction of users vulnerable to a threat that is known by the company. Whereas this trade-off is already implicitly performed while training ML models used by the AVs, we are not aware of AV companies explicitly making this choice.

AV's solution for the FP's cases is to add the legitimate software causing detection troubles to a list of known benign software (a.k.a. whitelist/allowlist). Therefore, if a scan for that software is requested, the whitelist will be first queried and immediately return that the file is safe without triggering a scan. This allows AVs to implement more aggressive heuristic and ML models since these will be triggered only for artifacts that passed by the whitelist checks. This strategy is very effective in practice because the AV can, for instance, whitelist the files related to the OS operation (e.g., Windows' System32 folder) and aggressively detect new files added to the system.

There are few literature reports about how whitelists are employed in practice by AVs and even their vendors do not fully disclose much information about their usage. We found few cases in which the companies clearly stated that a whitelisting mechanism is present in their products [99, 54, 20], even though we can hypothesize that similar mechanisms are used by all solutions due to FPs occurring due to the nature of the malware detection problem, despite all efforts of the vendors.

AVs usually refer to whitelists as a complementary resource to be used in special cases, such as when the AV is detecting software that users compiled themselves. However, there are evidences that AV companies start whitelisting software already in their detection routines generation step. We consider that understanding the impact of whitelisting in these procedures is essential, as they can significantly affect the detection results. They also significantly affect the detection rules generation itself, which become more complex than often proposed in multiple research work. A significant challenge of whitelisting software at this step is to keep up with the amount of data that legitimate software represents (e.g., a TB of database size for Symantec [76]). Another significant challenge is to scale analysis and fast respond to incidents face to the need of filtering our candidate detection rules that collide with benign software (e.g., it takes more than 30 minutes to be done for Ikarus solution [16]).

Despite all this impact, nobody is completely aware of how whitelisting mechanisms are implemented in the actual AVs. There are multiple possible implementations: (i) simply adding file hashes to a list of allowed files; (ii) consider files signed by trusted entities as clean; (iii) identifying some strings as indicators of the file's nature, and so on. During our analysis, we discovered that most solutions rely on some type of whitelist, although these significantly vary according to the AV. Avast, for instance, has a specific whitelist for its gaming mode to avoid detecting some protections as cheats. The BitDefender AV, in turn, whitelists web certificates to prevent warnings related to known certificates. In this AV, we can even find the strings used to log when an artifact was whitelisted (e.g., `WHITELIST_BY_POLICY`). In the Kaspersky AV, a `WhitelistManager` allows controlling individual processes. In practice, it is hard to identify the scope in which the whitelisting mechanisms are employed (e.g., during static matching or during runtime). For the VIPRE AV, we found that the whitelists are static, as suggested by the `StaticScanWhitelistForObject` function name. For Norton AV, there are both static checks implemented in the userland DLLs (to allow cross-site references in some web pages), as well as dynamic checks in the `IDSvia64.sys` kernel driver, which has a process whitelist option to be configured upon loading, identified by the `Application Whitelisting Enabled` parser message.

Although all these implementations are interesting and deserve attention, we limited ourselves in this paper to present the key AV operation points. Therefore, we opted to describe in more detail the case of the FSecure AV, as it illustrates some performance-wise project decisions. The `fsecr64.dll` core AV library exports the `FSE_checkFileInWhiteList` function symbol, which immediately suggested the use of whitelists by this AV. Delving into details, we identified that internal routines of this function are invoked right at the beginning of two other exported functions (`FPI_ScanFile` and `FPI_ScanMemory`), which confirms that not all system artifacts are verified, even in the case of scans requested by the users. When the artifacts are whitelisted, the scanning procedures immediately return. The `checkFileInWhiteList` function does not directly takes an artifact as argument. Instead, it receives a number that corresponds to an IDentifier for the artifact. Therefore, the AV does not effectively query a knowledge database to whitelist the artifact every time it is invoked. Instead, it keeps a lookup table of open resources during its operation. The knowledge database for that artifact is only effectively checked when the artifact is first open, created, or later modified. The information retrieved from the knowledge database is loaded in the whitelist lookup table, which is queried in the further invocations of the whitelist function.

6 Detection Challenges

In this section, we analyze the decisions taking by the distinct AVs when choosing scanning strategies and detection mechanisms.

6.1 What to scan?

A good AV is not only the one that has a good detection mechanism, but also the one that knows what to inspect and when to inspect. This implies a significant trade-off: On the one hand, inspecting all resources all the time imposes significant performance overhead. On the other hand, reducing the scanning capabilities opens attack opportunities. To evade detection, attackers often encapsulate their malicious payloads into other files and using varied formats. Ideally, these should also be inspected by the AVs, but this might have significant performance costs. A popular technique to hide payloads is to compress the malicious files, which would require AVs to extract them for inspection.

To verify if AVs are able to detect this type of construction and at which detection step, we selected a set of multiple malware samples originally detected by the AVs and compared their detection results before and after compression. The results are shown in Table 14. We discovered that AVs are able to extract multiple file formats (zip, rar, 7zip) to inspect their contents when a file scan is requested by the user. These extractors are implemented by the AVs themselves, as no standalone extraction tool was available on the tested systems. In fact, we found evidence of the presence of file extractor in some AVs: In the F-Secure engine, we found the `7z.dll` library; and in the Kaspersky AV, we identified the `minizip.dll` and `rar.dll` libraries.

Despite their extraction capabilities, AVs limit the analysis of compressed files to the on-demand scan modes. No AV was able to detect the compressed file in real-time, as soon as they were dropped in the file system, even though the AVs were able to detect the non-compressed version of the same files in real-time. This shows that AVs adapt their detection capabilities to the performance constraints of each detection mode and operational scenario.

This result has two implications for future research projects: (i) security analysis should not only consider whether an AV is able to scan a given artifact or not, but also in which time opportunity this check could be performed. For instance, it is not fair for a security evaluation to compare on-demand detection rates to online detection rates if their performance requirements are distinct; and (ii) there is space for future developments regarding improving the performance of AVs. For instance, AV accelerators would allow AVs to implement real-time inspection of compressed payloads.

Table 14: **Detection of Compressed Files.** Detection is performed only in on-demand mode.

File Type	**PE**		**ZIP**		**RAR**		**7z**	
Mode	**Online**	**Offline**	**Online**	**Offline**	**Online**	**Offline**	**Online**	**Offline**
Avast	90%	98%	0%	94%	0%	98%	0%	90%
MalwareBytes	0%	100%	0%	100%	0%	100%	0%	100%
Kaspersky	96%	96%	0%	16%	0%	0%	0%	0%
TrendMicro	26%	40%	0%	42%	0%	40%	0%	40%
VIPRE	100%	100%	0%	100%	0%	98%	0%	100%

There are detection challenges even when AVs are operating in the on-demand mode. Compressed files are not always simple to extract and, in many cases, the files are password-protected. We initially hypothesized that AVs would be able to brute-force passwords to crack these files. However, in our experiments, with the same samples considered for the previous experiment, we discovered that no AV cracks ZIP passwords (of any length). This result shows that there is also space for new AV architectures, such as the emerging cloud-based ones. In this hypothetical scenario, an AV would be able to upload files to a cloud to be cracked by a powerful computer without impact the performance and energy consumption of the endpoint machine.

Another challenge faced by AVs is to select what media to inspect. Currently, all AVs have filesystem filters to trigger scans of new files. They also have kernel drivers to interpose USB requests to block autorun malware [180]. However, there are other media types whose inspection is not enabled by current AVs. For instance, no evaluated AV inspects CDROMs when they are mounted, even though they scan files when copied from there to the filesystem and the processes created from the mounted device. An even more complicated case refers to the scan of network-mounted devices. In our tests, only the Kaspersky AV scans this type of media. The choice of inspecting network-mounted devices is a complex trade-off. On the one hand, not scanning them let other users vulnerable, especially if some user of such network is not protected by an AV. On the other hand, actively removing files from the network storage, as performed by Kaspersky, might remove third party files. In the worst case, a False Positive in an endpoint AV might cause the removal of files of any other user, even of those not running any AV solution in their endpoints.

In addition to the challenges currently faced by the AVs, new challenges are emerging and might pose significant threats in the future. For instance, the distribution of malicious code in multiple pieces might allow detection evasion [32].

6.2 GPUs & Machine Learning

GPUs emerged with a great potential for the development of security applications. Their Single Instruction Multiple Data (SIMD) characteristic naturally spans a myriad of applications based on pattern matching, which now could be performed in a massively parallel manner. The application of GPUs for signatures matching, for instance, is even suggested by NVIDIA itself [148]. Therefore, since the emergence of the first GPUs, many researchers proposed AVs based on GPUs [34]. In practice, however, the promise of a pure-GPU AV never concretized, and it is hard even to understand which parts of the promises become reality.

A scenario in which GPUs could help and that become real is the application of Machine Learning (ML) to security problems. Machine Learning is a trending topic in computer security and AVs are not unaware of this trend. One can be sure that modern AVs rely on some kind of ML technique, which can be discovered either by the AV's reports [193] or by indirect observations, such as the fact that attacking ML models in a standalone manner might have impact on the detection results of commercial AVs [40].

Although we can ensure that some kind of ML is used at some part of an AV operation, it is not clear which tasks are performed and in which manner. Whereas some often claim that "AVs always use ML", "ML are essential to AVs", "AV's detection is based on ML", "AVs rely on GPU for detection", and so on, we consider all of them as bold claims without further explanation and analysis. Therefore, to face this scenario, it is important to understand how ML is actually used by the AVs.

The first thing to understand GPU's usage by the AVs is to clarify where they are used in the security process. More than a decade ago, Kaspersky announced that the company was using GPUs to speed up malware similarity detection [94]. One should not confuse this usage with GPU application at client-side. The whole process is conducted at the server-side, under their full control, and clients only have access to the processing results.

Applying GPUs at client-side is much harder, since GPU programming is not standardized, with distinct vendors enabling distinct processing capabilities. Also, GPUs have access to a limited amount of memory, which might not suffice for loading the entire malware definition database. Moreover, the cost of offloading data from the CPU to the GPU is significant, which might limit the throughput of some real-time tasks. We believe that these drawbacks might have limited the development of GPU-based AVs so-far. Finally, even if these limitations did not exist, not all current systems are GPU-powered, which would not allow AVs to eliminate traditional processing routines.

It is also important to understand that not all applications of GPU are machine learning tasks. As far as we know, the only commercial solution that adopted GPUs for a security task is the Windows Defender, which partnered with Intel to run monitoring code in their GPUs [37]. The GPUs are used to perform memory scans using traditional methods, which is far from the application of any ML method.

In our experiments, we did not find an active use of GPUs by the AVs at the client-side for scanning purposes. We only found GUI-related components that internally make use of GPU for renderings, such as the `libGLESv2.dll` library (an OpenGL implementation) for Avast, and the `libcef.dll` (the Chromium Embedded Framework) for Norton and Bitdefender. Our investigation also did not reveal any use of traditional machine learning (ML) by the evaluated AVs. We looked for many traditional libraries used for ML processing (e.g., scikit-learn, tensorflow, mlpack, caffe, so on) and for log messages related to ML and no evidence of their use was found. This suggests that although ML is leveraged in AV's backend server to identify malware and generate detection rules [107], samples detection at the client-side is still performed using traditional signatures and heuristics.

The case of Next-Gen AVs. To fill the gap on the usage of ML by AVs, security companies have been promoting the called "next-generation AVs" [56, 188], which are basically AVs equipped with ML-based malware classifiers. These solutions are usually deployed in business settings, which are more controlled environments and with small software diversity than domestic environments, such that we are not sure that their transition to "home products" is easy. We tried to test these products to get a better understand of their actual detection capabilities. We subscribed for trial versions on many vendor's websites but, unfortunately, we did not get access to any solution.

Despite the lack of actual evaluation, we can highlight the fact that the application of ML on AVs is blurry even when they are indeed used by the solutions. The first thing to be clarified is how ML is applied: statically or dynamically. The static application of ML happens when a file is scanned without its execution, mostly via on-demand checks. The dynamic application happens when a process is monitored in runtime. Static applications are easier to be implemented and tend to be more widespread, even among the "next-gen" solutions, but suffer from multiple obfuscation drawbacks [140], which only can be solved via dynamic inspection. Dynamic ML approaches, however, are not so popular, although promised by some "next-gen" solutions, as they introduce greater performance overhead.

Regardless of the operation mode, the ML adoption indeed benefited AV products, as ML models generalize well, which helps to detect malware variants, and might be less susceptible than heuristics to trivial evasion attempts [71]. These characteristics led some to say that AV was dead face to the use of ML [157]. There are two reasons why we consider this statement wrong. The first is that despite all ML capabilities, it is not a silver bullet. There are tasks in which pure-ML approaches do not outperform traditional approaches, such as in the detection of fileless malware [75]. Therefore, a good detection solution cannot rely on a single detection method, but should be a layer of distinct approaches [92]. Second, an AV cannot be defined only by its detection engine. We have been demonstrating the multiple AV components and their roles over this paper. Thus, even if ML were a perfect detection mechanism, it could not operate solely, it would still require components to capture data to feed its algorithm and would rely on some other module to provide anti-tampering protection to it. Therefore, the ML detection would be just a component of a greater security solution, which is still an AV, in the anti-malware solution sense, whatever the commercial name it is actually called.

6.3 Detection on the Cloud

Cloud services have become widespread and currently it is easy to find AVs advertising the possibility of scans on the Web (e.g., Avast [21], Kaspersky [96]). It is also common in the academic field to find new proposals of cloud-based AVs and researchers on the AV field probably faced the claims that detection rates would be greater if cloud protection was enabled. Therefore, it is

important to take a look at the real aspects of this operational mode.

The first thing to have in mind is that cloud detection is not enabled by default in all products. In the Kaspersky AV, for instance, the customer is required to join the Kaspersky Secure Network (KSN) to enable such services. Otherwise, the AV fails with the *"TryGetActual disabled. User is not a member of KSN"* message. In the Avast AV, the customer must have the rights to perform cloud scans. The `Avast Asynchronous Virus Monitor (AAVM)`, implemented in the `Aavm4h.dll` library, calls the `AavmFmwDownloadUACloudEntitlement` and `AavmFmwGetUACloudAuthToken` functions to validate if cloud scans are allowed. If so, the `Antivirus engine loader`, implemented in the `aswEngLdr.dll` library, instantiates a cloud-enabled AV object via the `avscanEnableCloudServices` function. Therefore, whereas available in some AVs, cloud services should not be seen yet as the default scan mode.

In addition to Avast and Kaspersky, in our evaluations we also found cloud components in the BitDefender and the VIPRE AVs. BitDefender implements a cloud component (`bdcloud.dll` library) and VIPRE leverages the BitDefender's `AntiSpamThin.dll` library for AntiSpam Detection. As can be noticed, cloud services are not used only for the detection of malicious binaries. Some AVs also rely on cloud services for file backup and system telemetry. In this work, we will focus on the components responsible for threat detection.

All AVs structure their cloud detectors as objects to be instantiated by the AV engines. Therefore, the AVs present a pattern of object creation, usage, and deletion. In the VIPRE AV, the AntiSpam object is created using the `BDAntispamSDK_Initialize` function, configured using the `BDAntispamSDK_SetSettings`, and destroyed using the `BDAntispamSDK_Uninitialize` function. Similarly, BitDefender's is created and destroyed using, respectively, `Init@Cloud@Gambit` and `Uninit@Cloud@Gambit`.

All AVs operate in a similar manner. They upload a resource to be scanned in the cloud and wait for a detection response. In most cases, hashes are uploaded. In fewer cases, the objects to be scanned are directly uploaded. Most of the AVs operate over reputation scores provided by the cloud servers. In the VIPRE AV, it first submits an artifact, via `BDAntispamSDK_SubmitBuffer` or `BDAntispamSDK_SubmitPath`, and further retrieve detection results via `BDAntispamSDK_ScanBuffer`, `BDAntispamSDK_ScanPath`, or `BDAntispamSDK_GetIPReputation`. Similarly, BitDefender first uploads the artifact via `UploadFile@Cloud@Gambit` and later get results via `Query@Cloud`, `IsCloudRequestSuccesifull@Response`, or `GetResponsesCount@Response`.

Despite presenting these capabilities, we were not able to identify the use of cloud scan during typical AV usage for most AVa. An exception to that was Avast, which performs a query to `filerep-prod-011.mia1.ff.avast.com` to check the file reputation when an on-demand scan is triggered. For the other AVs, we were only able to trigger cloud scan on-demand and only when using custom configs. In the Kaspersky AV, for instance, a user can trigger a cloud scan via the Windows context menu. In this case, the AV queries on the cloud the reputation of the file and reports, for instance, how many other Kaspersky customers that joined the KSN have this file in their machines. The context menu triggers the `avpui` process, which is a GUI for the scan. It outsources the requests to the cloud to the `avp` process, which reads the entire file content, hashes it in a SHA-like manner, and sends it to the cloud to retrieve the reputation.

On the one hand, it is interesting to see how reputation-based methods become popular. They significantly contribute to increasing AV's detection capabilities, since updating a centralized database of reputation information is faster than updating individual endpoints, with the additional advantage of not requiring any storage space in customer's machines. It might enable, for instance, AVs to store an almost infinite number of signatures in their cloud servers. However, despite the cloud advantages, AVs do not (and in fact cannot) eliminate the traditional detection mechanisms, as they still have to protect the systems when the devices are not connected to the Internet. This might happen due to the device's operation on a constrained scenario/network, or even due to an attack, since it is plausible to hypothesize that in a scenario where only Internet-based scans are available, attackers would try to block Internet access to render devices vulnerable.

On the other hand, these reputation-based mechanisms cannot be classified as truly cloud-based "scans", since it would require them to upload the entire payload to a server and block its execution on the endpoint until some custom analysis is performed in the cloud server. We did not find evidence of this type of operation for any AV. Therefore, there is still a field of opportunities for researchers aiming to make these analysis procedures practical.

6.4 Real-Time

Behavioral detection is also strongly related to AVs. In the literature, we can find two frequent claims about AVs. Either that: (i) AVs only use signatures and not real-time monitors; or that (ii) AVs can implement complex behavioral detection routines. None of them are highly accurate, as the current state of AV's real-time detectors is heterogeneous. For instance, whereas some AVs have real-time monitors, this capability is not enabled by default to mitigate performance degradation [167]. Therefore, we here present a set of experiments and analyses of their results to draw a landscape of the actual usage of real-time monitors by AVs.

The first thing to have in mind is that current AVs do not use real-time monitors as a sandbox solution, tracing all API calls for generalized attack detection. Instead, only a subset of all API functions is hooked (see Section 5.8). These API functions are used for three distinct tasks: (i) enforce specific security policies (e.g., file access policies), (ii) ensure AV's self-protection, and (iii) detect some known, popular attack classes in runtime. It is also important to highlight that these tasks are not performed using a single method (userland blocking vs. kernel blocking), but a combination of them, according to the granularity level of the monitoring/blocking needs. We following describe some of these tasks in greater detail.

File Accesses. One of the main tasks that AVs perform in runtime is to enforce a security policy that establishes which files and directories can be accessed or not. This ensures the correct operation of multiple system components, from the browser to

the AV itself, which should not be tampered. To understand whether, how, and to which extent the distinct AVs monitor the filesystem, we developed a code that enumerates and tries to open all files in all directories. We then compared the results when using and not using an AV.

Table 15: **Filesystem accesses prevented by the AVs.** AVs block access to certain directories to avoid system infection and to ensure self-protection.

AV	Function	Paths
Avast	Self-Protection	C:\ProgramData\Avast Software\
		C:\Users\Win\AppData\Roaming\Avast Software\
	System Protection	C:\ProgramData\Microsoft\Crypto\RSA\MachineKeys\
		C:\ProgramData\Microsoft\RAC\StateData\RacMetaData.dat
Kaspersky	Self-Protection	C:\ProgramData\Kaspersky Lab\
		C:$Recycle.Bin\
	System Protection	c:\ProgramData\Menu Iniciar
		c:\Users\Default\AppData\Roaming\Microsoft\Windows\Start Menu\
		c:\ProgramData\Microsoft\Crypto\RSA\
		c:\Windows\System32\LogFiles\Fax\I
		c:\Windows\System32\LogFiles\Firewall
		c:\Windows\System32\LogFiles\WMI
	Internet Protection	c:\Users\Default\AppData\Local\Historico
		c:\Users\Default\AppData\Local\Temporary Internet Files
		c:\Users\Default\Cookies
MalwareBytes	Self-Protection	C:\Program Files\Malwarebytes\
TrendMicro	Self-Protection	c:\Users\Win\AppData\Local\Trend Micro\
		C:\ProgramData\Trend Micro\
	System Protection	c:\swapfile.sys
		c:\ProgramData\Microsoft\Crypto\RSA\
		c:\ProgramData\Microsoft\Windows\LocationProvider
		c:\ProgramData\Microsoft\Windows\Power Efficiency Diagnostics
		c:\ProgramData\Microsoft\Windows\Start Menu\
		c:\System Volume Information
		c:\Windows\System32\LogFiles\Fax\
		c:\Windows\System32\LogFiles\Firewall
		c:\Windows\System32\LogFiles\WMI
		c:\Windows\System32\networklist
		c:\Windows\SysWOW64\networklist
		c:\Windows\Temp
	Internet Protection	c:\Users\Default\AppData\Local\History
		c:\Users\Default\AppData\Local\Historico
		c:\Users\Default\AppData\Local\Temporary Internet Files
		c:\Users\Default\Cookies
VIPRE	System Protection	c:$Recycle.Bin
		c:\ProgramData\Menu Iniciar
		c:\ProgramData\Microsoft\Crypto\RSA
		c:\Windows\Logs\SystemRestore
		c:\Windows\MEMORY.DMP
		c:\Windows\System32\LogFiles\Fax\
		c:\Windows\System32\LogFiles\Firewall
	Internet Protection	\Users\Default\AppData\Local\History
		c:\Users\Default\AppData\Local\Historico

Table 15 shows that most of the prevented file accesses have three distinct goals: (i) AV's self-protection, with each AV protecting its own installation and configuration folders; (ii) System protection, with each AV protecting a distinct set of directories. In common, Windows configurations and logs are protected by all solutions; and (iii) Internet protection, with some AVs giving special attention to the browser history and cache. We highlight the fact that despite each AV deploying a distinct set of access rules, all of them implemented the same access control mechanism. This suggests that the OS might be lacking this type of protection mechanism as a native feature. In a scenario where the OS natively supports distinct policies, AVs would be required to distribute only their policy rules and not the mechanism itself to deploy them.

Process Accesses. Given the policies implemented for file access control, as presented above, one might hypothesize that AVs also implement access policies for handling processes, such as preventing a malicious process from opening a handle to a benign process. To evaluate this hypothesis, we implemented an application that enumerates all running processes and tries to open a

handle to them. We executed this application with and without a running AV and compared the outputs. We also varied the flags for file opening (e.g., read accesses, write accesses, so on) to identify whether AVs drop privileges or not. We discovered that the AVs do not interfere with process opening routines. All processes originally opened by our application without a running AV were also successfully opened under an AV with the same flags/attributes. Handlers to system processes were also successfully obtained. The only processes accesses that were effectively prevented by the AVs were the AV processes themselves. This shows that the contrast between the strong statements that people make about AVs is justified by the actual AV's behaviors, as some of the claimed properties are true (e.g., file accesses policies), but there is still room for improvement (e.g., process accesses policies).

DLL Injection Prevention. To understand AV's capabilities of detecting threats in real-time we need first to understand which classes of attacks require this type of detection. Whereas any attack detected statically could also be detected in real-time, AVs will likely implement dynamic detection mechanisms only for the ones that cannot be detected other way. The attacks that can be detected statically will likely be still detected this way since it is a more lightweight approach than running a monitoring infrastructure. Therefore, we consider that DLL injection a good case study to investigate dynamic detection mechanisms.

DLLs are not self-contained pieces of code–i.e., they do not run by themselves, but need to be injected into a host process to execute in their context. The injection can occur via multiple mechanisms (e.g., via the OS itself, or a custom loader). DLLs can be injected for benign purposes (e.g., providing legacy software compatibility, or extensions) or malicious purposes (e.g., tamper software execution, hijack control flow, so on). Due to this characteristic, it is hard to distinguish benign and malicious DLLs statically, as their behavior depend on the injected process. Thus, AVs tend to leverage dynamic monitoring mechanisms for this task.

To confirm that AVs use dynamic monitors and understand how they are used, we tried to load DLLs into diverse processes and check whether these were detected or not by the AVs and in which step. There are distinct ways one can load a DLL into a process [87]; some count on more OS support than others, and some are more documented than others. Our goal here is not to survey all existing injection techniques, but to exercise AVs face to distinct strategies. Thus, we considered distinct approaches, such as the most standard technique (CreateRemoteThread), Reflective Injection [169], Process Hollowing [113], and AtomBombing [174].

Table 16: **Code Injection Techniques Detection.** Distinct techniques are detected by the AVs using distinct methods. Some techniques are not detected at all.

Technique	**CreateRemoteThread**		**Reflective**		**Process Hollowing**		**AtomBombing**	
Detection	**Static**	**Dynamic**	**Static**	**Dynamic**	**Static**	**Dynamic**	**Static**	**Dynamic**
Avast	✗	✗	✗	✓	✓	✗	✗	✗
Kaspersky	✗	✗	✓	✓	✓	✗	✓	✗
MalwareBytes	✗	✗	✗	✗	✗	✗	✗	✗
TrendMicro	✗	✗	✗	✗	✓	✗	✗	✗
VIPRE	✗	✗	✓	✗	✓	✗	✗	✗

Table 16 shows how the AVs behaved in our tests when exposed to the aforementioned DLL injectors. We first notice that the traditional DLL injection method is usually not flagged as malicious by the AVs, except for VIPRE. We hypothesized that AVs assume these cases as likely benign due to the use of this method by many legitimate applications.

Unlike traditional DLL injection, it is hard to claim reflective injection as a legitimate use, since its manual mapping step aims to avoid using monitored system APIs. Thus, some AVs statically inferred the loaders implementing this mechanism as likely malicious (even though the libraries themselves were not flagged). We then modified the loaders to hide their imports and strings, such that they were not statically detected anymore. When trying to running these loaders, they were dynamically detected by Avast and Kaspersky AVs. We discovered that, in this case, the detection occurs because the AV identifies a lib that was loaded into the process space after the process startup but which did not use any of the functions hooked for monitoring by the AV (e.g., `LoadLibrary, CreateRemoteThread`), in a process similar to a lie detector.

Even when AVs implement dynamic detection mechanisms, there are still drawbacks that affect detection. For instance, in the Avast AV, the dynamic detection is performed by the called SmartScreen mechanism, which freezes the execution for a few seconds for scan, thus imposing some performance penalty. To speed up the performance, the mechanism caches scanning results. Thus, after a first clean scan, this result is cached and scans are not performed in subsequent launches of the same file/process. However, in the case of an injector, the scan result is very dependent on the injected payload. When launched with a valid DLL as an argument, the injector will call API functions with specific arguments that will trigger the dynamic detection. However, if the injector is run without a DLL as an argument, the injector will perform some calls without arguments and these will result in errors, such that the dynamic monitor will cache the information that this file/process is clean. In a subsequent execution of the injector with the actual malicious payload, the process will not be scanned, and the DLL will be successfully injected.

Similarly, process hollowing and atom bombing injectors are statically detected by some AVs. However, once we can hide their static fingerprints, the AVs are not able anymore to detect their execution as a malicious behavior. This result leads us to conclude that the AVs indeed have some capabilities of detecting threats in runtime, but these can still be significantly improved. As promising future approaches, we envision that the profiling of memory allocation activities, such as proposed by some "next-generation" AVs [120], are interesting strategies, as they would allow the detection of the constant code-page allocation instead

of the injection mechanism. We believe that for this approach become successful, an increased level of OS-AV cooperation is required.

6.5 Delayed Detection

In the delayed detection mode, the AV first captures a bunch of data and later reasons about it to raise (or not) a detection warning (e.g., collect thread creation information to detect injection attempts [137]). A noticeable source of information for delayed decisions is the Event Tracing for Windows (ETW) interface added by Microsoft in recent Windows versions. It allows the collection of thousands of events [123] about Windows applications, services, and drivers. The set of captured events includes a system-wide view of libraries loaded into the system's processes and the files created in the filesystem. Whereas these events are captured very fast by the ETW framework, we do not consider its operation as real-time because the AVs do not interpose functions to monitor them. Thus, AVs cannot block malicious actions directly. They are limited to act as passive listeners of the event loggers.

ETW was not available in the past, so it is not often described as part of a security solution. However, AVs recently started to use ETW as part of their detection routines and it is plausible to hypothesize that this mechanism will become each time more popular. For instance, McAffee provides a tool [119] to collect ETW events and security reasoning about it, even though in a standalone manner. Among the AVs we inspected, native integration with ETW was available for F-Secure and Vipre. In the F-Secure AV, we found the `fsetw_plugin64.dll` library as a host for the ETW plugin. However, it is not clear how it is used by the AV (although spoofed PID detection is supposed [86]).

In the VIPRE AV, ETW is used as a boot time monitor [122]. The driver registers its event monitors by writing to the `SYSTEM\CurrentControlSet\Control\WMI\GlobalLogger` registry key, as specified by Microsoft [121]. In the ETW format, events are generated by providers, managed by the controllers, and consumed by the clients. In the Vipre case, the `SbFwe.dll` library is the ETW controller. It exports multiple functions (`SbFweETW_BootLogging`, `SbFweETW_IsBootLoggingEnabled`, `SbFweETW_IsRunning`,`SbFweETW_Start`, `SbFweETW_Stop`, `SbFweETW_StopCurrentBootLoggingSession`, `SbFweETW_UpdateLogLevel` that allow the AV to manage the ETW collection.

6.6 Post-Detection

A frequent misconception is that the AV's job finishes when a threat is detected. In fact, there are still actions to be taken after it occurs. Ideally, an AV should allow users to report False Positives and Negatives, provide usage statistics to the vendor, and even restore the system to a clean state. To present an overview of the post-detection actions performed by the AVs, we investigated their behaviors and summarized them in Table 17.

Table 17: **Post-Detection Actions Summary.** We only considered the actions displayed in the GUI, although some of these actions are displayed via other channels (e.g., websites).

AV	Quarentine	FP Report	FN Report	Send to Analysis	Remediation
Avast	✓	✓	✓	✓	Limited
F-Secure	✓	✗	✗	✓	Limited
Kaspersky	✓	✗	✗	✓	Limited
MalwareBytes	✓	✗	✗	✓	Limited
TrendMicro	✓	✗	✗	✓	Limited
Vipre	✓	✓	✗	✓	Limited

Upon detecting malware files, all AVs move them to a quarantine. Although the name of the quarantine module has been changing over time (e.g., it is now called Virus Vault in the Avast AV), its operation principle remains unchanged since the creation of the first AVs. When a file is quarantined, it is hidden from the users by the AV, but it is not actually deleted: In most AVs, the file is only hidden from the user by using file system filters. For some AVs, such as TrendMicro, the original file is replaced by a modified version. The `C:\ProgramData\Trend Micro\AMSP\quarantine` directory of TrendMicro stores such modified versions, which consist of the original files encoded in a dynamic, XOR-like manner to avoid triggering further detection alerts. Upon moving files to there, the quarantine manager displays to the users a list of these detected files and allows users to restore (unhide) or actually delete them (in the TrendMicro's case, anyone with a decrypter or known the dynamic key generation algorithm can unencode the quarantine file [179]). If no action is performed, files are automatically deleted by all AVs after some time in the quarantine.

The quarantine should allow users to report that a detected sample was a False Positive (FP). Whereas some AVs really allow that, some of them opt to only whitelist that file locally. Reporting FPs globally is important because the same file misdetected here might prevent a legitimate software operation for other users in the future. Although AV companies do their best to generate unique detection patterns, it is hard to not conflict with any of the million possible software installed by a heterogeneous user base. Some AVs even keep a list of known FPs to alert users [72]. Ideally, AVs should allow users to report FPs as soon the threats are detected in a given file, but this capability was observed in only two AVs. The other AVs let this task for the users who check the quarantine. In the worst case, some AV vendors make this possibility only available via specific forms on their websites [116, 181]. This lack of integration with the main AV suite will likely make many users not report the cases.

Moreover, AVs should also allow users to report False Negatives (FNs). If a user somehow knows that a file is suspicious (e.g., because that file in the email already infected the user before, or infected a friend, so on) but this file is not detected by the AV, the user should be able to report it to the AV vendor. We found that only one AV provides this option upon a system scan with a clean result. The other AVs assume that it is very unlikely that users will have the knowledge to identify FNs. Despite that, all AVs provide some mechanism to upload a suspicious file to the AV servers to be inspected by analysts. This option is often placed in the context of getting a second opinion about the file, but it can also be used to report FPs and FNs. In addition to enhancing the AV's detection capabilities directly, this submission mechanism is also important because the files end up being part of malware feeds which will be analyzed by the AV companies in search of new attack trends [183].

Finally, AVs should also be able to clean the system after they detected that a malware sample executed there. All consider AVs report that they have disinfection and/or remediation capabilities. We tested these mechanisms by developing dropper malware [40] that was unknown to the AVs but that drops a known malware. Our goal was to check whether AVs were able not only to detect the dropped malware, but also the initially-undetected dropper malware which launched the known malware sample. We discovered that, in practice, the AV's remediation capabilities are limited. The AVs actually removed the malware dropper upon detecting the launch of the known dropped malware sample, but the registry keys written by the dropper malware remained untouched after the "disinfection". Even worse, when we split the malware dropping and the malware launching into two independent pieces of code, the AVs only removed the file responsible for launching the known malware sample, but not the one responsible for adding it to the filesystem. The difficulty of correlating events on AVs and other security solutions is a problem explored by the academic literature [32] and now even the AVs themselves acknowledge that their capabilities are limited [100]. Therefore, malware remediation is still an open problem and thus constitutes a field to be explored by future research work.

6.7 Network Inspection

Inspecting network traffic is a key component of AV detection engines. Whereas many studies point to new detection rules to be applied to network traffic, a few is known about how the traffic is inspected by the AVs. In this section, we shed some light on it by analyzing AVs to understand their network scanning routines.

AVs (can) leverage a mix of approaches to intercepting connections: (i) they (can) load browser extensions to read the content of the loading Web pages; (ii) they (can) proxy the traffic through an AV process to inspect it; and/or (iii) they (can) check connections after they were requested and/or established via kernel drivers.

In the first case, the browser extensions have access to the Web page's DOM and these can be inspected directly. These extensions can also retrieve the URL of the accessed Web pages and request the AV to scan for the URL reputation. We present an overview of browser extensions in Section 9.4.

In the second case, the AV redirects the traffic through a new process to allow scanning it, i.e., a legitimate Man-In-The-Middle (MITM) interception. We identified that this strategy is employed by the Kaspersky AV. In this AV, connections are forwarded to the `avp.exe` process, which is the AV proxy [95]. Traffic inspection is performed system-wide and includes encrypted traffic, which is enabled by the installation of an AV certificate (Kaspersky-signed, valid from 2010 to 2030) in the system. Therefore, connections to all websites appear to originate from the `avp.exe` process and to be signed by the Kaspersky certificate. Whereas this strategy allows traffic inspection, it reduces the network throughput, as as the establishment of a connection now requires the double of handshakes and encryption steps. Other drawbacks of this inspection solution are that: (i) the MITM performed by the AV might conflict with firewall solutions [95]; and (ii) application with custom certificate management, such as some popular Python scripts, might fail to connect to some websites with `INVALID_CERTIFICATE` messages.

When new network inspection processes are added to the system, either to receive browser plugin requests or to proxy the connections, these can be identified by the new ports open in the system. Using the `TCPView` tool from the SysInternals utility, we discover that the AVs open localhosts ports that can be connected by any application, not only the AV client.

Most AVs opt to inspect network connection after the requests were performed and/or after the connection was established. As shown in Section 5.8. most AVs load NDIS drivers, targeting the Microsoft networking filtering platform [128]. These drivers are not used to inspect the contents (as moving data from kernel to userland would be performance prohibitive), but to enforce security policies on the connections (e.g., blacklists addresses, protocols, so on). The firewalls embedded in the AVs are implemented via these drivers.

On the one hand, all AVs are very similar in terms of the implementation of HTTP inspection. All AVs make use of the `WinHTTP` Windows library to parse the collected domains. A typical inspection flow is to retrieve the URL from the NDIS drivers via `GetMessage`, tokenize the URL via `WinHttpCrackUrl`, then inspect the domains (e.g., trying to resolve the DNS for that domain/IP, as performed by Avast, for instance). On the other hand, AVs are very heterogeneous in the way they implement detection rules. All AVs but VIPRE implement custom inspection mechanisms. VIPRE relies on SNORT rules (enabled only in the premium version).

VIPRE's Snort Rules. The VIPRE AV stores its network signatures in the `\ProgramData\VIPRE\Rules\idsrules.dat` file. By the date we inspected it, this file was composed of **466** rules. The rules are typical snort rules with the addition of a custom threat level categorization, as shown in the Code 4 of Appendix D. From a research perspective, it is interesting to discover that an AV does not rely on custom detection mechanisms but in an open solution, since it allows any research developed on top of this same solution to be immediately transitioned to the market.

The 466 rules provided by the VIPRE AV cover both inbound as well as outbound connection. The inbound rules are intended

to prevent attacks. The outbound rules (named attack-response by the vendor) are intended to prevent attacks to proceed after an infection has already occurred. The rules are distributed over distinct protocols (58% TCP, 25% ICMP, 14% UDP, 3% IP) and cover their multiple use cases. For instance, whereas ICMP rules are used to handle ping to traceroute requests, TCP rules are used to detect known attacks and port scans.

Table 18: **VIPRE's AV snort rules.** Distribution of the rule's labels given by the vendor.

Prevalence	Label	Prevalence	Label
32,01%	misc-activity	4,19%	misc-attack
17,22%	attempted-admin	1,77%	attempted-user
9,71%	attempted-recon	0,88%	unsuccessful-user
9,71%	attempted-dos	0,88%	non-standard-protocol
6,62%	trojan-activity	0,66%	network-scan
5,08%	successful-user	0,66%	successful-admin
4,86%	web-application-attack	0,44%	protocol-command-decode
4,86%	bad-unknown	0,44%	denial-of-service

Table 18 shows the distribution of the attacks blocked by the VIPRE's rules according to the vendor's classification. Whereas some of them might be not immediately meaningful for the reader (and we don't want to disclose vendor's secrets here explaining their details), we can still have a broad picture about which tasks are performed by this module. In particular, we highlight the significant amount of rules aiming to block trojan-related activities, such that identifying the module that blocks the threats ends up revealing interesting characteristics of the blocked threat themselves (in this case, the large dependency on network services).

Among all rules, 28% are specific designed to handle known CVEs for multiple applications, ranging from Adobe to Microsoft products. Content is blocked by 11% of all rules via URL blocklisting. The remaining rules implement regular expressions and content matching.

7 AV Self-Defense & Monitoring

In this section, we analyze the attack surface exposed by the AVs, the risks of them being exploited, and the protection mechanisms leveraged to mitigate attack possibilities.

7.1 Attack Surfaces & Vulnerabilities

Many people refer to AV as secure solutions because they are security solutions. However, the two concepts should not be mixed. From a programming perspective, AVs are developed as any software, thus they might present bugs. The presence of bugs in actual AV solutions is revealed by the number of reports in the Common Vulnerabilities and Exposures (CVE) [136] list. Figure 9 shows the distribution of CVEs related to the antivirus keyword in the period between 1999 and 2020/September.

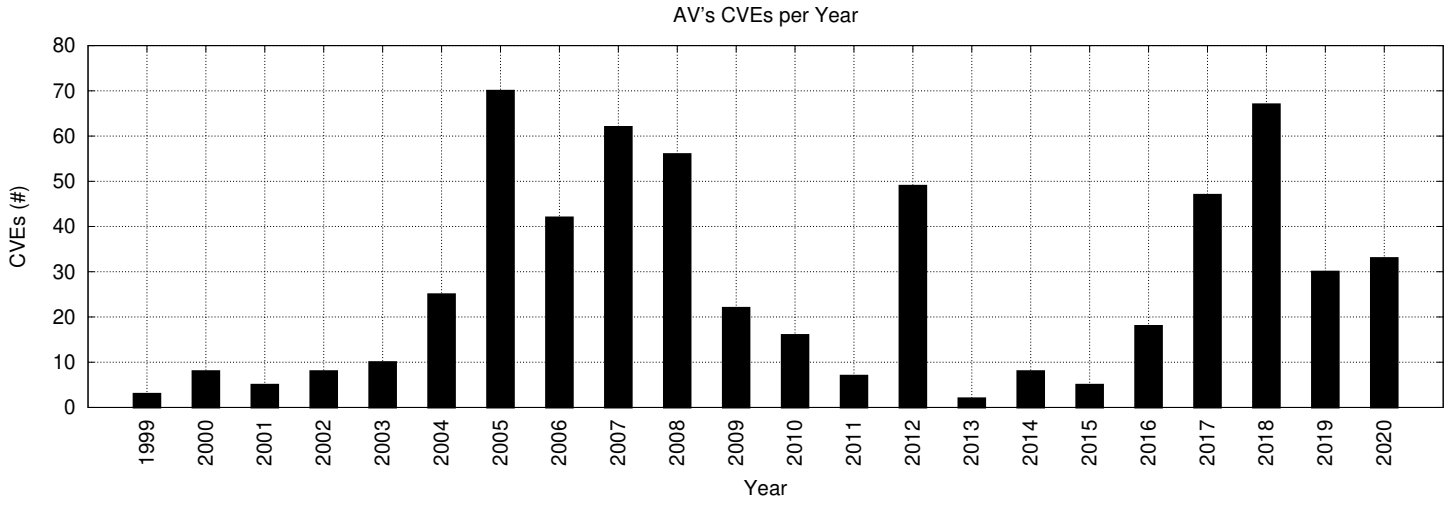

Figure 9: **AV Vulnerabilities.** CVEs per year.

Although the number oscillates significantly over time, we notice that vulnerabilities in AV products are reported and confirmed every year. These reports cover tens of distinct products and platforms (Windows, Mobile, and Mac). The vulnerabilities types are also varied, ranging from detection bypass (27%) to parsing, privilege escalation, and overflows (15% each one).

Considering the above, and the severe AV consequences presented in previous sections, we decided to investigate whether AVs might be potentially vulnerable to bug exploitation. Much research works have been done in the past about AV vulnerabilities [13, 190, 9], thus, we did not focus on finding specific bugs (as these might change as software are updated), but on identifying how exposed AV's API are and how accessible to testing AVs are.

We first checked whether we could load DLLs within our own process to manipulate them. We successfully loaded DLLs of all AVs into our host process and none of them was automatically unloaded, thus we believe that they do not check the host process that they are loaded into. Whereas implementing this type of check is desirable for a security solution, we cannot claim it as a problem by itself, since all AVs implement authentication mechanisms in the form of context objects that the host process must initialize before interacting with the libraries. However, this lack of loading checks allows us to invoke any exported function in an arbitrary manner, which includes passing invalid arguments to check for bugs/crashes. Therefore, we developed our own API fuzzer on top of our DLL host process to test AV's components. In our tests, we invoked the exported functions with multiple distinct and random parameters. We prioritized the test of the simplest APIs (i.e., the ones receiving no arguments or only integers, so on) since we believe that they are less dependent on the initialization of context objects and other components. We discovered many cases of crashes for all tested AVs. Even though many of the crashes might be due to the request of invalid options and/or non-implemented routines, this result shows that these functions are blindly trusting on previous validation steps, with functions not testing their own arguments for expected values. This opens significant opportunities for exploitation attempts. As a mitigation procedure, AVs could adopt defensive programming strategies [172]. We believe that the investigation of this possibility is an interesting open research question.

7.2 Anti-Tampering

As discussed in previous sections, AVs have to protect themselves from tampering to reduce the attack surface increased by their own installation. Self-protection is often overlooked in many research work and evaluations (we found only one research work tackling this problem [132]), but it is key to keep the protection mechanisms operating to secure a system. To better understand how AVs protect themselves, we analyzed their distributed packages and attempted to attack them, as following presented.

Installation Tampering. We started by investigating whether AVs are somehow vulnerable at installation time. As presented in Section 5.4. we discovered that most checks are not performed at installation time but rather post-installation. To evaluate post-installation checks, we performed the same checksum change experiment presented in Section 5.4, but now targeting the already-installed files. We first discovered that the files cannot be modified when the AV is running, as the AV installation directory is protected by the AV drivers. Booting in the safe-mode allows us to modify the files as the AV drivers are not loaded. We then discovered that the AVs present some integrity checks mechanisms to detect these modifications, which is a good practice. However, a drawback of this approach is that this mechanism renders the AVs vulnerable to DoS attacks if multiple files are corrupted (including the ones that ensure the integrity of other files in the AV's chain of trust). When we modified the checksum of all files, all AVs refused to start upon a reboot. This leads the system vulnerable. Malicious files that were previously detected in real-time by the AV as soon as they were added to the system could now be copied without problems. We observed that although the AV drivers raised a notification to userland, this notification could not be delivered as the AV components were not operating properly. This highlights the need of paying attention to physical security issues, as no AV can protect the system against an attacker that can manage the system. Certainly one can argue that if an attacker has access to the system safe-mode it could simply remove the AV. Whereas it is correct, an AV removal would be easier noticed than an AV problem. In the discussed attack, for some AVs, even their daemons remained displayed on the system tray, although not working. As a recommendation, AVs should emit clearer warnings when they are not working properly to allow users to identify the problem occurrence.

AV Loading & Reverse Engineering. Attackers are often investigating AVs to find ways to bypass them. Most attackers will adopt black-box methods to select a version of their malicious payloads that is able to bypass AV's detection. However, more sophisticated attackers might adopt gray-box methods, thus reversing engineering parts of the AV to understand why their payloads have been detected. To mitigate this type of attack, AVs might find ways to protect their code against improper usage. Koret and Bachaalany suggest in their book [106] that loading AV's libraries into an attacker and/or reverse engineer-controlled process was an effective way to interact with AV internals. They demonstrate that by reverse engineering some popular AV solutions of that time. To update their study and show the current status of today AV's protections we repeated their experiments. We discovered that the AVs do not prevent their libraries from being loaded into third party process in any way. We were still able to load their libraries into our processes (see code in our provided repository). However, they are all protected somehow. In most cases, the communication must be authenticated before an actual scanning routine could be executed. In some AVs, there are even libraries specialized into authenticate AV's usage (e.g., the `fs_ccf_client_auth_64.dll` library in the F-Secure AV).

In an overall manner, we would be able to replicate the book's experiments, but now with a greater complexity, as AVs evolved significantly. The AV detection routines are now not concentrated in a single library, as when the book was written but spread among multiple components. For instance, if we were going to replicate the Avast command-line tool (`ashcmd.exe`), we would have first to invoke the `aswProperty.dll` to create an object that defines the characteristics of the scans. Then, invoke the `shTask.dll` library to setup a scan (by passing the property object to the `tskInitActionContext` function) and start the scan (via the `tskExecData` function). If the scan takes a while to proceed and the user wants to query the scan progress, it should skip the first abstraction layer and directly query the engine loader (via the `avscanGetScanProgress` function in the

`aswEngLdr.dll` library).

As shown above, communicating with a modern AV is a hard task, but this should not be understood as a barrier for a motivated player, either an attacker or a researcher. Recently, a researcher demonstrated not only how to communicate with the Windows Defender engine but also ported it to work on Linux [150].

Processes Termination. Another possibility to tamper with an AV operation is to try to directly terminate it. Shields against terminators have been proposed academically [84], but it still unclear what real-world AVs actually implemented. Therefore, we implemented multiple strategies to terminate AV processes to evaluate their real characteristics. We discovered that, in an overall manner, all AVs employ some anti-termination mechanism. However, their implementation changed over time. More specifically, the protection mechanisms can be classified as before and after the Windows 8.1 release.

Before the launch of Windows 8.1, the OS provided no support for the anti-termination task, therefore AVs implemented their own solutions. The most usual one is to run the AV processes with elevated privileges (a.k.a. admin), thus only another elevated process could terminate it. Although offering some protection against the simplest threats, it was not enough to counter a malware able to escalate the first privilege barrier.

After the launch of Windows 8.1, Microsoft added support to anti-process termination, with the protection of anti-malware solutions being one of its main goals [124]. Microsoft introduced possibilities such as the protected processes concept, which cannot be terminated even by privileged processes. These processes are set by the early-launch boot drivers described in Section 5.8 and were used by all AV solutions considered in our evaluations. In all AV's strategy, not all processes are protected (e.g., UI processes are not protected), but only the key ones (e.g., AV engines, core services).

The rationale behind the adoption of the protected processes is not to eliminate the risk of AV termination but move the attack surface that would allow it for a more privileged ring. Now, in addition to escalating its privileges to administrator, a malware sample would also have to scale to the OS kernel to be able to defeat the AV. Once in the kernel, a malware driver/rootkit can disable the process protection [118] and further terminate the process.

Driver Unloading. A strategy that attackers might employ to render AV protections ineffective is to disable kernel protection, which would prevent AVs from collecting data and from securing critical resources. Therefore, AVs must prevent kernel drivers from being unloaded by third party processes.

We first hypothesized that AV could be applying rootkit-like technique techniques to hide drivers from the applications. For instance, AVs could employ Direct Kernel Object Manipulation (DKOM) to hide the kernel drivers from the OS list [82]. However, we discovered that, in practice, all kernel drivers are visible to the system. The AV focus is on their protection and not on their hiding.

We discovered that there are two strategies used by AVs to protect their drivers. Many AVs implement their drivers as non-PnP (Plug aNd Play) driver and/or do not implement the `DriverUnload` routines for their drivers. Therefore, attempts to unload them result in the `1052 error: unsupported operation`. To ensure that these drivers will always be loaded, AVs rely on the creation of their respective services with the `NOT_STOPPABLE` and/or `NOT_PAUSABLE` flags, which prevents even administrators from changing their characteristics. Attempts to exclude the services are blocked by the kernel-based filters denying access to the OS services' configuration files and registry keys.

In summary, the operation of driver protection mechanisms can be seen as a cycle, where a service prevents a driver from being unloaded and the driver prevents the service configuration to change.

DLL Unloading. Another strategy an attacker might employ to defeat an AV operation is to unload the AV inspection library from the memory of the malicious process. To avoid being defeated, the AVs should prevent their unloading. As for the driver's case, we first hypothesized that the AVs could be hiding the libraries from the OS linked lists to be invisible to the processes enumeration routines. This strategy could be implemented by a manual mapping DLL injection procedure[4]. However, in practice, we found no AVs employing DLL hiding, thus all injected libraries are visible to the malicious processes, that can use their presence as a fingerprinting mechanism for evasion purposes. Fortunately, despite visible, the DLLs cannot be unloaded by any standard mechanism, neither by directly calling the `FreeLibrary` function [130] nor via external tools [147]. We discovered that the AVs prevents the unloading of their libraries by pinning them via the `GET_MODULE_HANDLE_EX_FLAG_PIN` flag during the load. Therefore, those injected libraries behave as if they were linked at boot time and can only be unloaded at process termination. We can confirm that by looking at the reference counters of the injected libraries, which always exhibit the maximum allowed value (65535) and never decreases even after a `Freelibrary` invocation.

7.3 Telemetry & Logs

Security is a continuous process and thus, like any process, it requires feedback. In the AV's case, feedback information about the health of the protected systems is given by telemetry information. The good use of telemetry information might help AVs on identifying implementation bugs, open security breaches, new attack trends [41], and account actual exploitation cases [25]. If not well protected, telemetry data/logs can also be abused by criminals [42].

The value of telemetry has been shown in practice by Microsoft [171], with collects data from millions of customers to predict if one of them will be compromised. However, despite this study, not much information is available about how other companies use telemetry data in their solutions, which motivates us to take a further look at stored data and collection mechanisms.

[4]Undocumented injection technique where the injector manually sets internal OS structures to include the DLL without calling OS APIs to reference it

The telemetry system operates basically periodically sending to the AV servers information collected during the AV operation in the endpoint. A major source of information is the AV logs. The whole AV operation produces logs, whose content might give us an idea of what kind of information is collected and stored by the AVs. The logs can be used to improve AV's operation both locally as well as remotely. It is hard to identify which information is sent to the remote server, but we can still have insights.

The database of the Avast AV (Figure 16 of Appendix D), for instance, stores a history of the updates, scans, and most detected threats. It allows the AV company to identify weaknesses in their protection and to design new mechanisms. For instance, the information that the users are not updating their products within a reasonable time might indicate that new automatic update procedures should be developed. Similarly, a low scan frequency might indicate that the scan scheduler should be adjusted. For the other AVs, similar information is collected. TrendMicro is able to even separate events by the triggered detection engine (see Figure 18)

We believe that measurement studies relying on real logs collected from AV user's machines would present interesting insights to the security community about how computer users protect themselves via the use of AVs. These insights might help to guide the development of next-generation AV solutions. However, as far as we know, no study publicly presented such information so-far.

Ideally, all the information collected by the AVs should be available to the user, but this is not what happens in practice in most cases. Although the AVs have mechanisms to integrate their logging mechanism with the Windows native event viewer [98], we observed that this integration is not enabled by default in most cases. Therefore, there are still opportunities for developing better integration tools built upon the log of AV engines.

8 AV Performance

A frequent complaint about AVs over time is that they cause the system to slow down, which motivated (and still motivates) research on improving AV scans performance. According to Aycock [24], there are 4 strategies for accelerating an AV scan: (i) reducing the amount scanned; (ii) reducing the amount of scans; (iii) lowering resource requirements; and (iv) changing the algorithm. Whereas the strategies have been previously enumerated, no study evaluated how these have been applied to actual AV solutions and what is their impact on performance.

Although AVs have evolved to mitigate the performance penalty, the performance overhead imposed by AVs is still significant [184]. This is explained by the AV interaction with system components for interposition, which adds overhead to their operation (e.g., impacting the filesystem [7]). Whereas some literature work characterized AVs regarding the quantitative performance overhead [5], few qualitative analyses were performed to explain which parts of the AV operation most impact the system performance. Therefore, in this section, we aim to bridge this gap and characterize the overhead imposed by the multiple operation modes: real-time, on access, so on. We focus on the relative overhead imposed on the system and not on the absolute value since it would become fast outdated as the CPU's performance is ever increasing.

The first thing to have in mind about AVs is that their operation is not homogeneous, neither their imposed overhead. AV's operation can be characterized in distinct steps: idle, on-demand checks, and real-time monitoring.

Idle. For an AV, remaining idle means that no on-demand or scheduled scan is being performed and no new application to be monitored in runtime is launched by the user. For the OS, however, the idle time does not mean that no operation is performed. Instead, this time is used by background processes to perform their operations. For instance, update mechanisms are often launched by the OS and the applications when the system is idle. These operations will also be monitored by the AV, thus the idle time does not mean that the AV is inactive nor that it does not cause performance impact.

To understand this impact in practice, we used the Resouce Monitor (ResMon) application to measure the CPU usage of the multiple AVs components (engine processes, GUIs, associated background services) when idle. We considered a fresh Windows installation, with browsing and office applications. The CPU usage was repeatedly measured by consecutive 60 seconds.

Figure 10 shows that the CPU usage imposed by all AVs when idle is low, ranging from 5% to 10%), but not negligible. Moreover, the error bar indicates that even the idle operation has processing peaks, reaching up to 20% of CPU usage, which is caused by the creation of system process in the background and the writeback of cached files in the filesystem.

For some applications, even the overhead of background scans might make the AV operation prohibitive. For instance, AV scans during the execution of a game might be enough to significantly reduce the frame rate to the point of bothering the user/player. To avoid these cases, AVs developed the gaming modes [102, 22] to prevent background tasks to affect the system performance. Most AVs automatically trigger the gaming mode when a full-screen application is launched. Whereas this dynamic adaptation characteristic shows AV's flexibility to meet user's requirements, which might also indicate an opportunity of developing new scanning solutions that do not overload the main CPU (e.g., AV co-processors).

On-demand checks. A key constraint for AV's performance in the on-demand mode is the need of loading the knowledge (e.g., signature) database to scan the file. A strategy to mitigate this performance penalty is to preload the signature database to be used when required. This is often implemented by the AV daemons.

A drawback of this approach is that a significant amount of memory is spent during the whole system operation with AV signatures without immediate use. Unfortunately, as most AVs are closed-source solutions, we cannot recompile them with and without daemons to measure their impact in practice. However, we can understand this impact by inspecting the open-source ClamAV solution. ClamAV can natively operate with (`clamdscan`) and without (`clamscan`) a daemon that preloads the knowledge database.

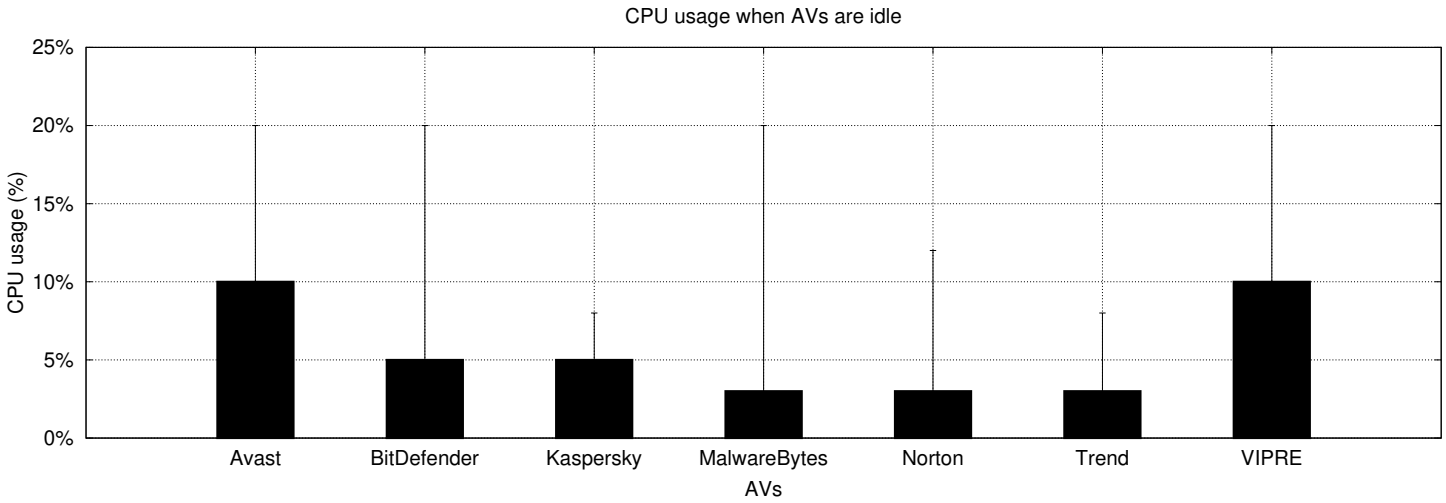

Figure 10: **AVs performance when idle.**

Figure 11 shows the memory and CPU usage during on-demand ClamAV scans with no database preloading. We notice that the AV scan of the same dataset considered in the previous experiments took 25 seconds. During the whole time, the memory consumptions kept increasing, as the database kept being uploaded in memory, until reaching the total of 1GB. The CPU usage rate indicates that the matching started since the beginning of the loading of the first signatures. However, the matching was limited by the availability of signatures to be matched, thus the CPU rate is limited by a memory upper-bound. When considering the operation of the ClamAV daemon, the scan of the same files took an average of 0.03s, an 800 times speedup. As a drawback, the same 1GB of data was preloaded by the daemon and kept resident in RAM during the whole system operation, even when no scan was active.

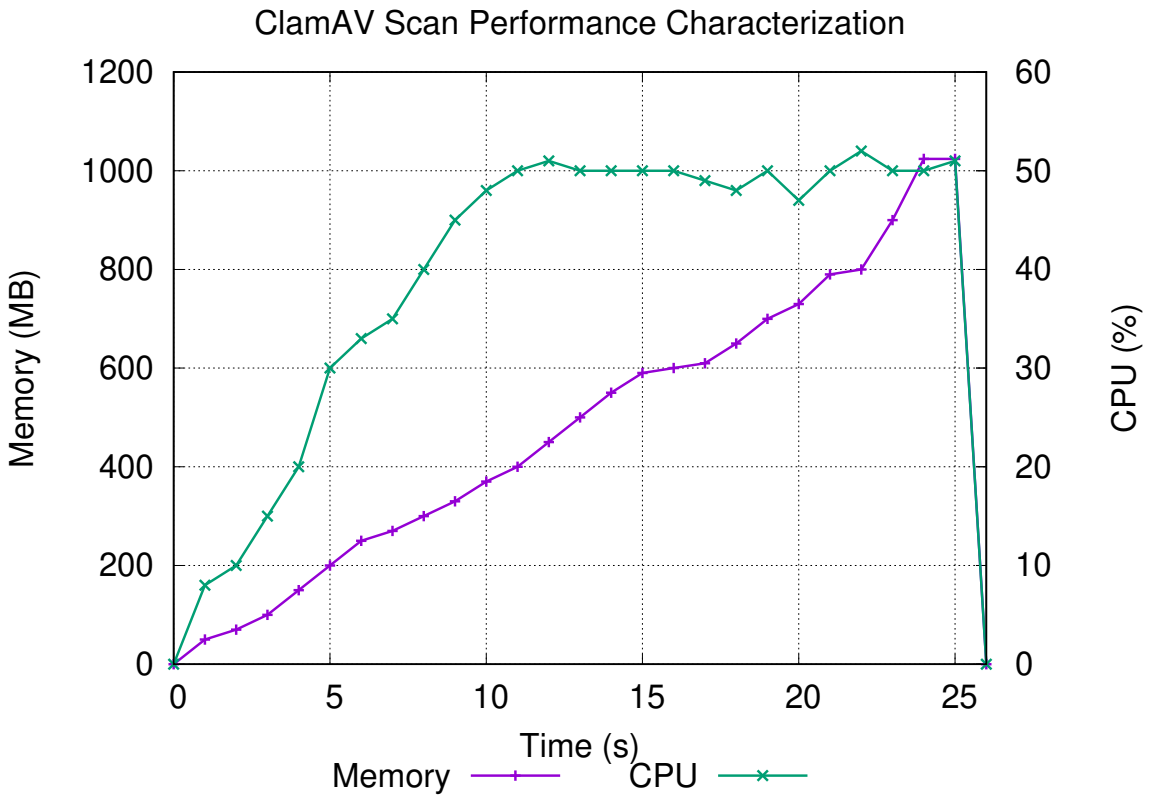

Figure 11: **ClamAV Performance.** Significant memory and CPU overheads are imposed to load the signature database.

After the loading of the signatures, the second AV task that most affects performance is the signature matching itself. The matching procedure is directly affected by the input files: the larger the files, the more CPU cycles are required to fully inspect them. Besides, the more complex the file format, the more complex the rules required to model a malicious pattern within them. Despite that, some performance-focused optimizations can be performed to speed up signature matching.

A possible optimization is to pre-compile the matching rules. For instance, regular expressions can be compiled into automata to be directly matched from memory. Once again, as AVs are closed-source solutions, we cannot recompile distinct signatures schemas to evaluate their performance impact. However, we can understand them by looking at/ a popular matching mechanism, the YARA framework. YARA rules can be compiled both on-demand or beforehand. Figure 12 shows the overhead of compiling typical YARA rules for malware detection [194] in comparison to pre-compiling them. As hypothesized, YARA rules are very distinct in complexity, thus the overhead of compiling them is also distinguished according to their performance requirements. The simpler the rule, the greater the relative performance impact of compiling the rule. The more complex the rule, the more the compilation time is mitigated in the matching time. In the worst case, a third of the total matching time is spent in compiling the rules for matching. This shows that there are still opportunities for the development of more efficient matching procedures and the investigation of distinct matching algorithms [134].

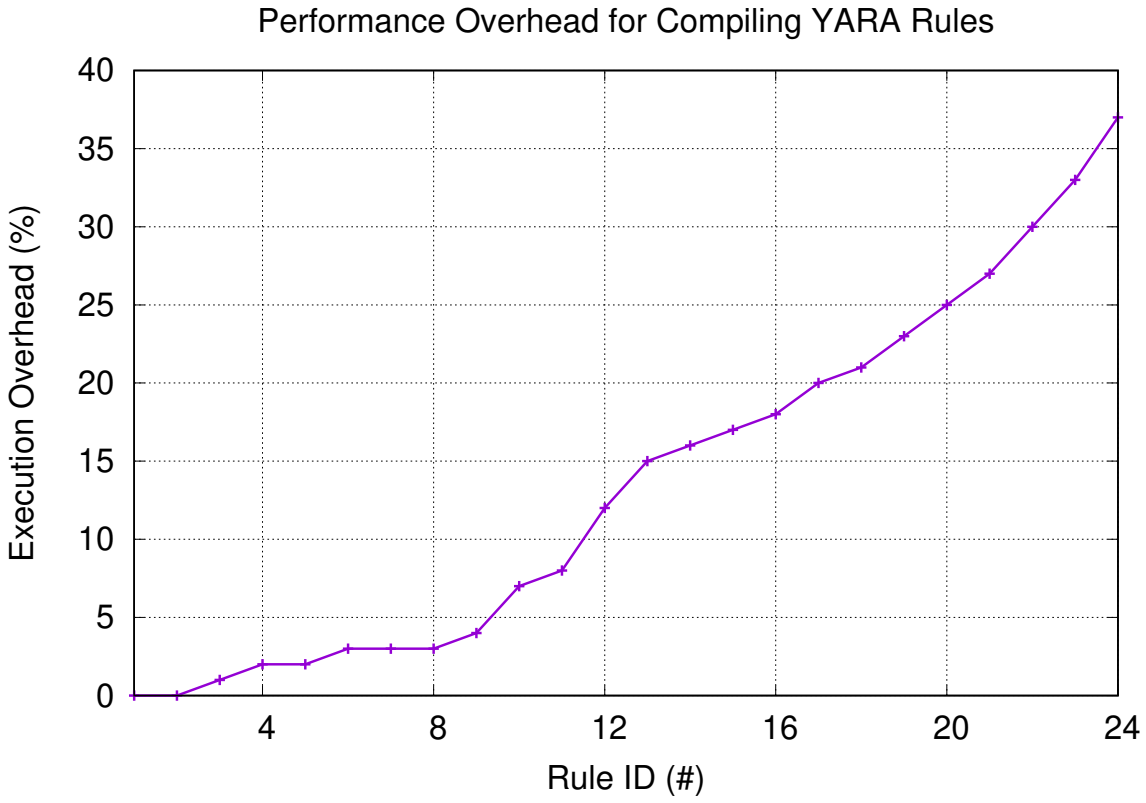

Figure 12: **The Matchign Cost.** Precompiling complex YARA rules might save significant CPU cycles.

Real-Time When operating in the real-time mode, AVs inject libraries in the running processes to hook into API functions. Since the AV code starts to be executed preloading the API functions whenever they are called, performance overhead is introduced.

Evaluating the imposed overhead is hard, since distinct AVs monitor a distinct set of APIs. To present a fair evaluation, we selected a subsystem that is monitored by the distinct AVs: the process subsystem. We developed an application that enumerates all running processes in the system, tries to open a handle to them, and queries basic process information, such as PID and paths. This triggers at least one monitored API call for each AVs, thus we can compare the overhead imposed by them.

Figure 13 shows the performance overhead in the number of CPU ticks considering an average of 50 repetitions. We notice that all AVs cause significant performance penalties in the application execution. For all cases, the AV monitoring process more than doubled the number of spent CPU cycles for the software execution. Although this result cannot be generalized to a whole-system operation, since it is a micro-benchmark, it shows that the performance impact imposed by AV's real-time monitoring solutions is a real issue, thus deserving attention for further research work.

More specifically, we observe that all AVs impose a similar performance penalty regardless of their distinct threat intelligence routines. This shows that the monitoring cost–the cost of injecting a library and hooking APIs–is responsible for the largest part of the processing time rather than the intelligence routines. Therefore, investigating alternatives for function interception–such as parallel scanning mechanisms–seems to be a promising way for future work on the field.

Speed ups & Caches. Regardless of the operation mode, performance is a concern for the AVs, so they try to mitigate the performance impact in multiple ways. A widespread strategy is to rely on caches. A cache of kernel data, as shown in Section 3.4, allows the AV to avoid resolving repeated queries (e.g., get process name from PID) for consecutive, repeated operations intercepted by it. A file cache, as shown in Section 5.6, allows the AV to repeatedly scan the same files that were scanned previously and were not modified.

On the one hand, it is interesting to see how AVs found an effective way to mitigate the scanning overhead. Although AVs

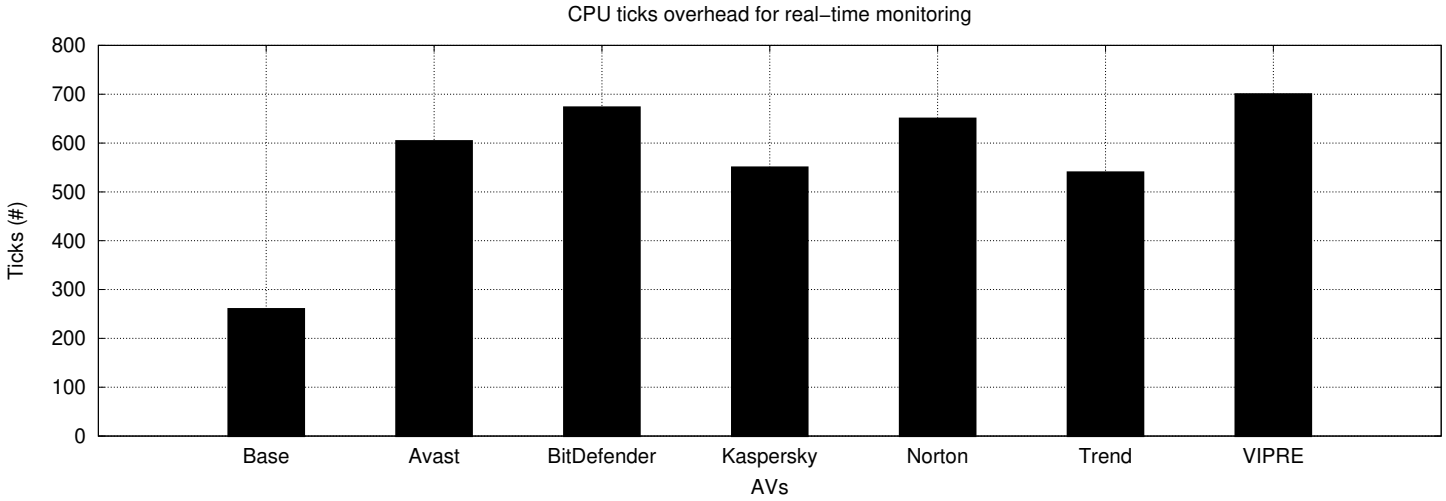

Figure 13: **Real-Time monitoring overhead.** The performance is dominated be the interception cost rather t han by the analysis routines.

implement many optimizations in their detection routines, the use of file caches shows that, in the last instance, not scanning is the best solution for mitigating the overhead. The saved cycles due to a skipped verification might be essential in the future to perform more complex detection routines. On the other hand, there is still room for improvement regarding the times when detection routines are actually executed, as following discussed.

Multi-Core Systems. Despite the distinct operation modes, a common characteristic of all AV's performance is that AV's loads are not well-balanced among distinct processor cores. When we shifted our experiments to multi-core machines, the obtained results were very similar to the aforementioned ones. In most cases, no multi-core processing was observed. In a few cases, we observed the same behavioral profiling in distinct cores, which we discovered to be due to duplicated AV processes running in distinct cores. The only AV operation step that effectively benefited from multiple cores was the matching step, which is performed in a multi-threaded way in all AVs. This step can be naturally parallelized and this fact is massively exploited by the AVs. In one specific case, we found an AV that launched 78 distinct threads to match the files in a directory. Therefore, improving the AV performance is still a relevant research task, especially if it involves other AV operation steps rather than the matching step.

9 AVs Platforms & Architectures

Although all AVs present in an overall manner the same architecture and the same operation modes, as presented over this work, some particularities significantly affect their detection capabilities, as following presented.

9.1 x32 and x64 Windows AVs

AVs have to adapt themselves to the modifications that their underlying systems undergo over time. A significant set of modifications were imposed in the transition from 32 to 64-bit Windows systems. The modification that most affected AVs was the introduction of the Kernel Patching Protection (KPP) mechanism, which prevented AVs from directly hooking into the system tables [36]. Currently, to overcome that, AVs started employing filters and callbacks made available by the OS to monitor the system operation. However, in the past, some companies faced significant trouble to transition (e.g., Sophos AV did not work in x64 systems [149]). Therefore, to understand the current state of kernel monitoring, we inspected the drivers deployed by 32 and 64 bit AV versions. We discovered that past AV versions did not update their drivers for the 32 bit systems, deploying hooks in 32 bit and callbacks in 64 bit systems. However, as newer AV versions were launched, AVs merged their 32 and 64-bit versions. Currently, all AVs operate the same way in 32 and 64 bits, relying on the same callbacks for the same drivers.

9.2 Windows vs. Linux AVs

Whereas designed for the same task of protecting critical system resources, the AVs for the distinct platforms are different in the same proportion as these platforms are different. For instance, the resources to be protected in Linux and Windows systems are different: whereas in Windows configuration information is stored in the Registry, in Linux they are stored directly in files (e.g. `/proc`), which implies in distinct protection mechanisms.

In common with its Windows counterparts, the Linux AV we evaluated (**ESET** AV for Linux Desktops) is also client-server-structured. On the other hand, its protection and working are mostly based on the OS native features. The AV adds shell scripts to the system to perform some checks in given key system operation points. Most of the script's protection is performed

via obfuscation. The AV does not add a driver to the system, but adds multiple `.so` libraries. The `libesets_pac.so` library is injected via `LD_PRELOAD` into running processes via the `/lib/pkg/postinstall` post-installation script that writes the library path at `/etc/ld.so.preload`. This library wraps the most common `libc` functions, such as `open`, `write`, `execv`, and `socket`. In addition to real-time checks, the AV also performs static binary checks. An interesting part of the Linux AV threat model is that it also detects Windows threats. In our threats, windows binaries (PE files) and scripts (VBS files) were detected as soon they were copied to the filesystem. Some filetypes, such as `docx` files are skipped from the checks by an explicit whitelist configuration.

9.3 Mobile AVs

As for the Linux vs Windows case, the mobile environments also have particular characteristics to be protected. Capabilities such as sending SMS, contact lists, so on are only present on mobile and not on desktop. Also, mobile environments present other restrictions, such as preventing superuser access (rooting). Therefore, it is plausible to hypothesize that mobile AVs would be significantly different or less effective than desktop AVs. We evaluated that in practice by analyzing the apps described in Section 4.

We discovered that the Android AVs are not as modular as the desktop ones. Whereas desktop AVs distribute multiple files to be plugged into each system component, the Android AV application is more self-contained and plugged into the system by the Android environment itself. This shows the impact of a distinct OS architecture over the security solution. Despite this fact, the Android AVs are also client-server-structured, since the most complex routines implemented by them (e.g., scanning routines) are placed into native libraries that are invoked via the Java Native Interface (JNI) [12]. Interestingly, native libraries are also often by malware samples that these same AVs aim to detect [2]. Unlike desktop AVs, Android AVs do not load any kernel driver (which is sometimes even prevented by the stock Android environment), thus they eventually monitor the system with the same privileges as the malware samples [161].

Since they cannot monitor the system from a more privileged ring, the AVs try to ensure good data tracking coverage by requesting almost all available permissions in their manifest file. They also register almost all existing intents and broadcast channels to be notified about system-wide events. An intriguing side-effect of this broad request policy is that although the AVs claim they aim to guarantee user's privacy, many of them declare third party components in their manifests whose tokens allow these third parties to track the user's interaction with the system (e.g., by adding the Facebook API).

The AVs register intents to receive messages when certain actions are performed in the system (e.g., when the device connects to a new WIFI network), which allows them to perform some basic checks. However, to offer the same protection level as in their desktop versions, the AVs need to inspect applications in a more fine-grained manner (e.g., check which URLs are accessed by the user). As no native support is offered for these tasks, the AVs collect these data by exploring the accessibility services. For instance, accessibility resources originally designed to read screen content for blind people are now used as a mechanism to collect data from web forms and application fields. Interestingly, this same strategy is used by some malware samples that the AVs aim to detect [108, 10, 93].

As the AVs do not have in-app access as in the desktop versions, most of their detection capabilities are presented in the form of pattern and signature matches. In the first cases, the AVs present some templates of know attacks against popular applications (e.g., Whatsapp scams). In the second case, traditional, byte-based signatures are employed. For some AVs, we were even able to find references to the EICAR test file [63] embedded in the apps. The mobile AVs try to overcome the limited real-time analysis capabilities in comparison to the desktop AVs with more full system scan checks. The solutions often schedule multiple full system scans per day to detect files that they have missed during runtime monitoring. Also, a significant part of these AV's detection is powered by reputation systems (e.g., of popular applications) and/or blacklists (e.g., spam-sending lists).

The mobile AVs present clearer assumptions than the desktop AVs. They assume the system is not `rooted` and most of their protection comes from this fact. In fact, many solutions actively seek for rooting applications. Similarly, they also trust the standard App Store and notify the user when this is not working properly. Similar to desktop AVs, mobile AVs also have to protect themselves against terminators. Most of them implement mechanisms to prevent their removal.

Some of the limitations when scanning files are not a major problem for most mobile AV's threat model as detection seems to be only a minor part of their protection goals. Most AVs provide complementary security resources to protect users, such as file wipers to safely delete files, file vaults to safely store files, applications lockers for access controls, VPNs for safe web navigation, and anti-theft mechanism to lock the device when it is lost and/or stolen.

From a security analysis perspective, most of the attack surface added by these AVs are due to validation, licensing, logging, signing, and billing routines. The interactions with the OS are in fact the smallest part of the added attack surface. Therefore, the development of these solutions should follow the same best practices adopted for any other application class that handles these same inputs. From an architectural point of view, mobile AVs can be understood as an intelligent agent that is added to the system to make decisions about security implications. Most of the monitoring part of their operation is implemented by the Android environment itself, and the OS cooperation might be a trend for future developments and emerging platforms (see Section 10). We following detail our findings of particular AV operations.

Kaspersky. This AV monitors the system in a broad manner. The phishing protection is applied even to the received SMS. The SMS monitoring is performed via accessibility services. This AV was the only one to implement its own monitoring solutions in addition to relying on the Android services. It monitors the filesystem via the `inotify` Linux framework, as revealed by the call to the `inotify_rm_watch` function in the `libapp_services.so` library. This is one of the 5 native libraries embedded in

this application. This AV was also the only one to specify it has the ability to scan artifacts in the cloud. This is an interesting alternative for the Android environment as the remote server can have deeper system introspection capabilities to a sandboxed Android environment than the limited access that the AV has to the local OS. The AV is periodically updated via the Internet. It requires a minimum available storage space of 2MB, which suggests that it is the maximum size of an individual database update. However, the update occurs via HTTP, with the AV explicitly asking the Android to not enforce HTTPS over that connection. Whereas this practice was already identified in desktop-based AVs [28], likely due to legacy compatibility, it is not clear why it is replicated for mobile ones. The AV app is shielded with the `libdexterprotector.so`, a third-party solution. Interestingly, the APK file drops at installation time the `android_wear_micro_apk` APK, which is a lightweight AV version intended to run on wearable devices, such as smartwatches.

PSafe. Whereas presenting most of the previously described characteristics of mobile AVs, this AV was the only one that does not implement its detection engine via native libraries. It also collects forms information using accessibility services and schedules a daily full system check. We identified that the real protection claimed by the AV is implemented via template matching of known attacks against popular applications (e.g., Whatsapp scams).

AVIRA. This AV collects data following the AV's usual accessibility collection mechanism. It uses this data, for instance, to be notified when new applications are installed and launched so it checks the application integrity and signature. The AV mixes blacklists and whitelists approaches: Whereas it blacklists phone numbers, it whitelists multiple popular applications (e.g., Facebook, Instagram, Waze, so on). Its detection capabilities are also based on signatures, as the presence of the EICAR file indicates. The signature database is hourly updated. The AV embeds 18 natives libraries, including the ones for OpenVPN and JDNS, in addition to the AV core. The AV core presents AV self-checks (e.g., `AVSIGN_IsAviraFile_CustompublicKeyA` function) and also reference signature generation (e.g., `ST_CreatePeFileSignature` function). In fact, there are multiple references to PE, the Windows executable format, over the AV code, which suggests that the AV communicates with a shared backend between mobile and desktop AVs. In practice, however, we did not find any PE detection case.

ESET. This AV is very explicit about its checks. It notifies the user that it will warn applications downloaded from unofficial sources and that it collects browsing information even when surfing in the anonymous mode. Browser monitoring is performed via accessibility services but is only available for some browsers (e.g., noticeably Chrome). Its detection is performed via signature, with the EICAR test file being identified on it. A full system check is scheduled by default every 6 hours. It communicates with a single native library that implements the AV core. It is a complex, threaded library that invokes multiple system calls and writes to a `sqlite3` database. An interesting finding is how this AV is concerned about not only the user device but also the surrounding environment security. It checks the connected network for outdated router firmware versions, DNS poisoning attacks, and even devices vulnerable to known attacks. Although the app clearly warns that these capabilities should not be used against third parties, this cannot be prevented.

Avast. This mobile AV presents most of the aforementioned characteristics. Its manifest file requires access to multiple resources, such as bookmarks, history, so on. Some of these resources are only used for data leakage prevention, such as periodically cleaning the clipboard. The app has distinct database files (e.g., `networksecurity.db`, `applocking.db`, `call_blocking.db`) that are used to load blacklists for IP addresses, phone numbers, and so on. The AV explicitly checks for rooting application using code generated by the `RootCheck` tool.

AVG. The AVG application embeds the AVAST backend. As a significant difference, it also embeds OpenVPN native libraries to provide VPN access support.

9.4 Browser Extensions

Many applications have been moving to the Web, and so the attackers. This requires AVs to also move to there to provide effective detection. We showed in Section 5.4 that many AVs dropped browser extensions (XPI files) during their installation. We here present an overview of these components.

In an overall manner, these extensions are a minified version of the main AV client. They are also organized in a client-server manner, with the extension opening a socket to send requests to the AV engine main process (e.g., querying a given URL reputation). Most of their action focus in detecting suspicious URLs, but they can also inspect scripts and even data placed into forms (e.g., passwords). Their monitoring largely relies on callbacks provided by the browsers (though some can also implement browser hooks). The most popular callback is related to browser's tab activities, which is used by the AVs to trigger new inspection procedures as soon as new tabs are created. Tab information could also be used to track malicious URLs paths, as suggested by the academic literature [173], although we cannot confirm that the AVs perform such kind of tracking in their backends. Some AV extensions can also inject scripts into the pages to be able to manipulate their DOM objects. In some cases, the monitoring is disabled by whitelisting mechanisms that skip popular and/or buggy URLs from checking. Most of their operation is autonomous, but in a few a cases user intervention is required (e.g., confirm he/she wants to visit a given suspicious site). As a significant difference from binary-based AVs, most of the extension's protection is provided by the browser infrastructure itself, whose manifest files already contain hashes and self-signed files that ensure their integrity and legitimacy. In most cases, no obfuscation was identified in the extension's files. We following detail specific cases.

Kaspersky. This is the most complex browser extension among the evaluated ones. It is structured in a client-server architecture, with websocket and XMLHTTP communication. A session with the main AV process is only open after the extension specifies an ID and a key. It prevents other processes to communicate with the AV engine process in the same port. This same protection is implemented in the binary-based version. The extension receives distinct detection codes for malware and for phishing detection.

The AV clearly specifying its attempt to detect phishing is important because AVs have already been reported to have very distinct detection rates for these two classes [29]. As optional features, this extension also blocks miners, removes advertisement banners, and offers password protection via password quality checks and virtual keyboards. The extension integrity is only self-protected by the hashes in the manifest files. Few obfuscation signs are identified in the app, except the password quality algorithm. We discovered it checks for the password length and for repeated keys in the password to give a password score. In addition to integrity checks, the extension also implements its own handling of MD5 hashes, which eliminates the need to trust external entities. The extension monitors the navigation by injecting a script into every page, which allows it to parse DOM objects in the visited pages. It also hooks the websocket APi and registers multiple browser callbacks to be notified when new URLs are requested. It parses not only the visited URLs but also the links contained in the pages. It also captures navigation cookies and checks their validity. Some domains are excluded from verification, such as Google–it avoids looking to the search parameter for this website. Domains are identified based on the URL prefix.

TrendMicro. In contrast, this extension is very simple. It is also structured in a client-server manner, but not authentication is required. It is only protected via the manifest file checks. It is not clear if it detects phishing in addition to malware. Its detection capabilities are fully powered by browser callbacks. They deliver the accessed URLs to the extension, which whitelists some of them, such as google.

F-Secure. This extension is similar to the previous one. It is also a client-server structure with no authentication. It is protected by a third-party signed manifest file. It is no clear if phishing is specially handled. The monitoring is performed via browser callbacks that deliver the accessed URLs, which triggers some code injection in special cases. The callbacks also allow the extension to check TLS connection parameters and certificates. The access is blocked if the certificate does not match the accessed domain. As an optional feature, the extension provides a safe search mode, that also acts as parental control. It operates based on some whitelists, which includes youtube URLs.

BitDefender. This AV deploys two distinct extensions: an anti-tracker and a password wallet. Both are client-server structured authenticated via an ID. In the wallet case, the server is requested to generate strong passwords that are pasted into web forms. The extensions are protected by a third-party signed certificate. They also rely on third-party components (URL package and webpack framework) that are obfuscated. It is not clear if the extensions handle phishing in addition to tracking. Some tracking URLs are whitelisted. They monitor the browser via callbacks. The wallet extension also injects scripts to manipulate the DOM and paste the passwords in the forms.

Avast. This AV also distributes two extensions: a page reputation checker and an option safe price plugin. They are client-server extensions authenticated via a user token. They are protected via a third-part signed manifest. The reputation system detects not only malware but multiple classes of phishing and harmful content. Both extensions collect the accessed URLs from tab callbacks. They implement complex logic to decode hidden URLs, including base64 decoding. The URLs are hashed and their reputation is queried via the main AV process. Optionally, the user can vote on the reputation of a page. User's votes are uploaded to a remote server for crowdsourced detection purposes. The optional safe search plugin implements the same capability and is structured in the same way (even same files) as the page reputation plugin, but does not have any security goal. Its goal is only to search the web and advertise prices. It collects user information for this search. It is implemented based on the protobuf protocol for communication and jquery framework for parsing, such that it is not clear if this wide attack surface is really required.

AVG. As for the main binary, the extensions distributed for AVG are the same distributed for Avast. Their distinction in the presentation for the user is performed via the locale file, which is used to display distinct messages for Avast and AVG.

9.5 Case Study: ClamAV

ClamAV is a very popular platform for AV development, with many research works built on top of it (see Section 3.1). Therefore, when we talk about the need for a better understanding of AV engines, it is common for someone to refer to ClamAV as an alternative since its source code is open [43]. However, we identified many limitations that make ClamAV not fully resemble a commercial AV. We discuss them here to reinforce our claim on the need of considering real AV issues.

ClamAV highlights AV's complexity to protect the whole system. A significant part of its code is dedicated to parsing the distinct file formats (Currently, 12 distinct file formats are supported [45]). Despite this significant implementation effort, it does not provide complete security guarantees against infections, since attackers exploiting other, unchecked file formats will still succeed in getting into the system. In practice, the academic literature already demonstrated that attackers often migrate file formats to evade AVs and infect systems [27].

ClamAV also puts significant development efforts on verifying the signatures of PE files [48], checking whether their certificates were issued by a trusted authority or not, or whether they are expired or not. In some sense, this mechanism acts as a kind of whitelist, as binaries signed by a trusted entity (e.g., Microsoft) will hardly be considered malicious. This verification is skipped for non-signed binaries, which leads to the surprising conclusion that a malicious binary file signed with a fake certificate is more likely to be detected than a non-signed malicious file.

ClamAV implements a wide set of static detection mechanisms, with the simplest one being the checksum verification. ClamAV allows MD5 and SHA digests to be matched against entire files and/or specific PE sections. These hashes are also used in the whitelist mechanism [50], which allows the AV to skip the scanning of some files. Whereas still supporting MD5 hashes is important for legacy compatibility, any AV making use of them might be vulnerable to collision attacks. State-of-the-art solutions for MD5 collisions are reasonably efficient [170] (in a cryptographic sense), thus it is plausible to hypothesize an attacker creating

a malware whose hash collides with a whitelisted one to evade detection in a targeted scenario.

In addition to the checksum, ClamAV also supports the called body signatures, which are byte sequences matched using regular expressions. Although the AV documentation has a rich guide on how to write good signatures [44], bad signatures might eventually be deployed. These signatures can be deactivated individually using the whitelist mechanism [46], thus mitigating false positives. Moreover, the AV also supports the called bytecode signatures, which are C functions written to match more complex patterns. These are compiled and integrated into the AV engine in runtime.

ClamAV also supports the called container signatures to allow the inspection of files compressed as RAR, TAR, 7z, and other formats. The AV distributes a list of passwords as part of its update process that are used to try to open password-protected containers. This allows the AV to detect malware into containers protected with known passwords (e.g., malware samples are often distributed in zip files protected with the "infected" password).

Over time, ClamAV implemented new detection features, being the adoption of YARA signatures one of the most significant ones [49]. ClamAV currently does not support the whole YARA framework, with some modules not being implemented. However, for the future, we can hypothesize that the YARA framework might even replace the ClamAV core since most of their matching strategies overlap significantly.

Whereas presenting matching capabilities very similar to commercial AVs, ClamAV starts differentiating from commercial AVs for the auxiliary detection routines. For instance, a good AV engine should be able to unpack a myriad of file formats to allow the scan of the clear binary. The analysis of ClamAV's source code revealed unpacking algorithms for the UPX and ASPACK, but not for other ones. Besides, no runtime-based, generic unpacking method was identified, which limits the AV detection capabilities. Similarly, a good AV engine should provide deobfuscation engines to allow the scan of clear strings and data. Whereas we found methods to deobfuscate base64-encoded and RC4-encoded files, no other methods (e.g., XOR-based variations) were identified.

Many of the signatures distributed by AV companies aim to match instruction patterns, thus AVs often implement disassemblers. We identified a custom-implemented disassembly in ClamAV's code, but it is limited to the x86 (32bit) architecture, which limits the application of rules to this platform. For the future, ClamAV's developers and researchers might rely on third-party disassemblers to extend the AV's capabilities. In addition to statically looking to instructions, good AV engines often have the ability to emulate code portions to reveal the real malware behavior. Unfortunately, ClamAV does not implement a code emulator. There are also third-party, open-source solutions for this task that could be eventually integrated into ClamAV's code by researchers and developers in the future.

Modern AVs also have been relying on ML-based techniques to detect a greater number of threats, as discussed in previous sections. Unfortunately, there is also no support for ML detectors in the ClamAV core. We believe that integrating ML capabilities into ClamAV is a great opportunity to test academic proposals in a practical scenario.

Despite the aforementioned limitations, ClamAV presents at least part of the static capabilities provided by commercial AVs. Unfortunately, when we talk about dynamic capabilities, ClamAV provides almost no resource comparable to real AVs. ClamAV does not have a real-time module or load kernel drivers to enforce security policies. Whereas some extensions aimed to bridge this gap, they are still limited by either (i) working only on Linux due to the need for the `inotify` framework [47], or (ii) when operating on Windows, limited to invoke ClamAV's static procedures to newly created files [51], with no real-time threat intelligence. The lack of real-time monitoring also limits the AV's self-protection capabilities. No effective anti-tampering countermeasure was identified in ClamAV's code.

Finally, ClamAV is also limited in the monitoring surface, i.e., in the number of distinct agents collecting data for scanning. ClamAV does not collect data from the network or from the browser, which limits its action to the file scans triggered and/or scheduled by the users.

In summary, whereas investigating ClamAV's structure is an interesting task to get the first insights about the internal working of an AV, it does not eliminate the need of looking to a real AV engine ao as to be able to transpose concepts to an actual scenario. Therefore, from a research perspective, ClamAV should be seen more as an underlying platform for the development of future solutions on top of it rather than the definite AV solution itself.

10 Discussion

In this section, we revisit our findings to discuss their implications and also point limitations of our approaches.

The AV concept changed significantly over time, but these solutions are still the most popular type of security solutions nowadays. This solution class has been renamed over time from Anti-Virus to Anti-Malware, to Anti-APT (Advanced Persistent Threats), and currently stands by the name of EDRs (Endpoint Detection & Response). Whatever the name they are called, it remains essential to understand how they work to increase the protection they offer to the users.

AV Development. The available material on how to develop an AV solution is still scarce. Microsoft published in 2019 the first example on how to write a kernel driver to support AV operations [127]. As far as we know, this is the only material available covering AV development aspects. Therefore, this work's main goal is to shed light on some important aspects of AV development procedures. We adopted an analytical approach that reveals some of the decisions that AV vendors make to implement their solutions. We expect that this information might be useful for anyone interested in developing an AV engine. We also hope that we might inspire future work on the development of AV solutions.

The impact of whitelist. Our findings revealed that the AV solutions rely on whitelists to enhance their detection procedures. This is not often considered in the academic design of detection methods, although its impact is significant. In practice,

comparing a whitelist-free approach to a whitelist-based approach is unfair. Whitelist-free approaches often lower their detection capabilities when tuning their parameters to not flag benign artifacts as malicious. Whitelist-based approaches, in turn, might apply tighter thresholds for detecting more artifacts while whitelisting any false positive case. Unfortunately, current AVs do not fully disclose when the detection of an artifact was whitelisted. Making this information available would help researchers to conduct experiments and perform more fair comparisons (e.g., only among samples that were or were not whitelisted in both the reference AV and in the new proposal).

Found strings and detection information. As for the whitelist, other factors influence experiments that measure AV detection (e.g., if detection was static or dynamic, due to signatures or heuristics, so on). A fair experiment should consider the same type of detection for both the reference AV and for the new proposal. Unfortunately, most of the current AVs do not disclose the reasons why a sample was detected. Recently, Microsoft started providing this type of information for some of their security solutions [125]. The strings found during our analysis procedures indicate the presence of symbols for the distinct detection aspects for all AV engines, thus suggesting that this information could be easily made available to the users. Therefore, we expect that all AV solutions could move towards this more open direction in a near future.

OS support for AVs. The procedure of detecting a malware sample can be classified into two steps: a monitoring step and a threat intelligence application. The monitoring step consists of collecting data for inspection. The threat intelligence consists of making a decision based on the collected data (e.g., blocking a process). Our results showed that whereas desktop AVs implement agents for both steps, mobile AVs are more focused on implementing threat intelligence agents, as many monitoring mechanisms are implemented by the OS itself. A drawback of an OS-provided monitoring mechanism is that it restricts AV coverage to the surfaced specified by the OS. An advantage of this approach is that the OS developers are capable to deliver monitoring mechanisms more safely (e.g., function hooking often leads to crashes due to race conditions with OS structures accesses). In our view, this movement towards OS-based mechanisms is a trend, which starts to affect even desktops, as seem in the Microsoft movement of preventing hooking into kernel tables via KPP in favor of the new callback interface. If this trend consolidates, we expect OSes to provide deeper inspection capabilities. For instance, Microsoft recently added an interface for drivers changing its memory pages permissions [131]. We expect that this kind of interface to become available to AVs to allow them to change page permissions of their protected applications, which would allow them implementing more complex security policies [35].

The AV Future. Our results highlighted the operational aspects in which AVs perform well but also show that there is room for improvement in many aspects. It is always hard to make predictions, but we believe that an emerging research topic that might help to improve the next generation AVs is hardware support. Distinct proposals suggest adding external monitors [33] or CPU extensions [35] might help to achieve greater security guarantees.

Limitations. In this work, we shed light on the importance of understanding the AVs internal aspects. Unfortunately, due to market reasons, AVs are closed-source solutions, thus information about their internals is not easily available. To overcome this limitation, we adopted a hands-on approach. Although we were able to present a broad landscape of their internals, some details might have been missed due to the intrinsic nature of the black-box analysis process. Moreover, protection mechanisms, such as obfuscation, make the analysis task harder. Face to these cases, we focused on providing an overview of the AV operation instead of delving into particular aspects. Therefore, we do not claim this work as exhaustive. Further investigations might reveal more fine-grained details about specific operational aspects and component's implementations.

Future Work. AVs are complex pieces of software and no single work would be able to address all their component's working. Most of the resources presented in the Section 3.4 deserve an investigation by themselves. For instance, the security of the password managers implemented by the AVs needs to be investigated. Therefore, we expect that this work might foster future research on AV internals.

11 Conclusion

In this work, we investigated the project decisions behind the implementation of AV's internals to characterize the operation of this type of security solution. We identified that only a limited set of research works in the literature investigated AV internals and bridged this gap by analyzing popular (Windows, Linux, and Android) solutions to present a landscape of their operation in practice. We discovered, for instance, a great disparity in the set of API functions hooked by the distinct AV's libraries, which might have a significant impact on the viability of academically-proposed detection models (e.g., machine learning-based ones). We also discovered that whereas AVs provide reasonable resilience against popular packers, they cannot handle well other data encodings (e.g., XORed files), which is highlighted as a significant open research question. Finally, we discovered that whereas all AVs claim rootkit detection capabilities, most of them are based on static detection checks, which significantly affect business threat models. We expect our study might foster further research in the field and that our findings might work as support for these.

Reproducibility. All scripts developed to analyse and test the AVs are available in the repository at: `https://github.com/marcusbotacin/reverse.AV`

Acknowledgments. This project was partially financed by the Serrapilheira Institute (grant number Serra-1709-16621) and by the Brazilian National Counsel of Technological and Scientific Development (CNPq, PhD Scholarship, process 164745/2017-3).

References

[1] R. Abrams and A. Marx. Scripting av signature file updates and testing. https://www.av-test.org/fileadmin/pdf/publications/avar_2004_avtest_paper_scripting_av_signature_file_updates_and_testing.pdf, 2004.

[2] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Krügel, G. Vigna, A. Doupé, and M. Polino. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *NDSS*, 2016.

[3] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel. When malware is packin' heat; limits of machine learning classifiers based on static analysis features. In *Proceedings of NDSS*, NDSS, 01 2020.

[4] M. Al-Asli and T. A. Ghaleb. Review of signature-based techniques in antivirus products. In *2019 International Conference on Computer and Information Sciences (ICCIS)*, pages 1–6, April 2019.

[5] M. I. Al-Saleh, A. M. Espinoza, and J. R. Crandall. Antivirus performance characterisation: system-wide view. *IET Information Security*, 7(2):126–133, 2013.

[6] M. I. Al-Saleh and H. M. Hamdan. On studying the antivirus behavior on kernel activities. In *Proceedings of the 2018 International Conference on Internet and E-Business*, ICIEB '18, page 158–161, New York, NY, USA, 2018. Association for Computing Machinery.

[7] M. I. Al-Saleh and H. M. Hamdan". Precise performance characterization of antivirus on the file system operations. *Journal of Universal Computer Science*, 25(9):1089–1108, 2019.

[8] alreid. Peid. https://www.aldeid.com/wiki/PEiD, 2016.

[9] S. Alvarez. Antivirus (in)security. https://fahrplan.events.ccc.de/camp/2007/Fahrplan/attachments/1324-AntivirusInSecuritySergioshadownAlvarez.pdf, 2007.

[10] Y. Amit. Accessibility clickjacking – android malware evolution, 2016. https://www.symantec.com/connect/blogs/accessibility-clickjacking-android-malware-evolution, accessed on 11. August 2018.

[11] L. An, M. Castelluccio, and F. Khomh. An empirical study of dll injection bugs in the firefox ecosystem. *Empirical Software Engineering*, 24(4):1799–1822, Aug 2019.

[12] Android. Native apis. https://developer.android.com/ndk/guides/stable_apis, 2019.

[13] A. Antivirus. Feng xue. https://www.blackhat.com/presentations/bh-europe-08/Feng-Xue/Whitepaper/bh-eu-08-xue-WP.pdf, 2008.

[14] I. Arghire. Windows 7 most hit by wannacry ransomware. http://www.securityweek.com/windows-7-most-hit-wannacry-ransomware, 2017.

[15] Ashwyn. Recommended method for installing avast on an infected computer. https://forum.avast.com/index.php?topic=147079.0, 2014.

[16] K. Ask. Automatic malware signature generation. http://www.gecode.org/~schulte/teaching/theses/ICT-ECS-2006-122.pdf, 2006.

[17] K. Askola, R. Puuperä, P. Pietikäinen, J. Eronen, M. Laakso, K. Halunen, and J. Röning. Vulnerability dependencies in antivirus software. In *2008 Second International Conference on Emerging Security Information, Systems and Technologies*, pages 273–278, 2008.

[18] Avast. Avast and avg become one. https://blog.avast.com/avast-and-avg-become-one, 2016.

[19] Avast. Aswvmm.sys problem. https://forum.avast.com/index.php?topic=205585.0, 2017.

[20] Avast. Avast threat lab - file whitelisting. https://support.avast.com/en-ww/article/Threat-Lab-file-whitelist, 2018.

[21] Avast. Cloud antivirus. https://www.avast.com/business/resources/cloud-antivirus, 2019.

[22] Avira. Avira antivirus: Game mode explained. https://www.avira.com/en/blog/avira-antivirus-game-mode, 2020.

[23] AVTest. Antivirus & security software & anti-malware reviews. https://www.av-test.org, 2018.

[24] J. Aycock. *Computer Viruses and Malware*. Springer, 2006.

[25] L. Bilge and T. Dumitraş. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 833–844, New York, NY, USA, 2012. Association for Computing Machinery.

[26] J. Blackthorne, A. Bulazel, A. Fasano, P. Biernat, and B. Yener. Avleak: Fingerprinting antivirus emulators through black-box testing. In *Proceedings of the 10th USENIX Conference on Offensive Technologies*, WOOT'16, page 91–105, USA, 2016. USENIX Association.

[27] M. Botacin, H. Aghakhani, S. Ortolani, C. Kruegel, G. Vigna, D. Oliveira, P. L. D. Geus, and A. Grégio. One size does not fit all: A longitudinal analysis of brazilian financial malware. *ACM Trans. Priv. Secur.*, 24(2), Jan. 2021.

[28] M. Botacin, G. Bertão, P. de Geus, A. Grégio, C. Kruegel, and G. Vigna. On the security of application installers and online software repositories. In C. Maurice, L. Bilge, G. Stringhini, and N. Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 192–214, Cham, 2020. Springer International Publishing.

[29] M. Botacin, F. Ceschin, P. [de Geus], and A. Grégio. We need to talk about antiviruses: challenges & pitfalls of av evaluations. *Computers & Security*, 95:101859, 2020.

[30] M. Botacin, F. Ceschin, R. Sun, D. Oliveira, and A. Grégio. Challenges and pitfalls in malware research. *Computers & Security*, page 102287, 2021.

[31] M. Botacin, P. L. de Geus, and A. Grégio. Leveraging branch traces to understand kernel internals from within. *Journal of Computer Virology and Hacking Techniques*, 16(2):141–155, Jun 2020.

[32] M. Botacin, P. L. de Geus, and A. Grégio. "vanilla" malware: vanishing antiviruses by interleaving layers and layers of attacks. *Journal of Computer Virology and Hacking Techniques*, 2019.

[33] M. Botacin, L. Galante, F. Ceschin, L. C. P. C. Santos, P. L. de Geus, A. Gregio, and M. Zanata. The av says: Your hardware definitions were updated! In *14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2019)*. IEEE, 2019.

[34] M. Botacin, P. L. D. Geus, and A. grégio. Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools, and methods for systems and binary analysis on modern platforms. *ACM Comput. Surv.*, 51(4), July 2018.

[35] M. Botacin, M. Zanata, and A. Grégio. The self modifying code (smc)-aware processor (sap): a security look on architectural impact and support. *Journal of Computer Virology and Hacking Techniques*, 16(3):185–196, Sep 2020.

[36] M. F. Botacin, P. L. de Geus, and A. R. A. Grégio. The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 14(1):87–98, Feb 2018.

[37] P. Bright. Intel, microsoft to use gpu to scan memory for malware. https://arstechnica.com/gadgets/2018/04/intel-microsoft-to-use-gpu-to-scan-memory-for-malware/, 2018.

[38] M. Brinkmann. Firefox will block dll injections. https://www.ghacks.net/2019/01/21/firefox-will-block-dll-injections/, 2019.

[39] A. Bulazel. Windows offender: Reverse engineering windows defender's antivirus emulator. https://i.blackhat.com/us-18/Thu-August-9/us-18-Bulazel-Windows-Offender-Reverse-Engineering-Windows-Defenders-Antivirus-Emulator.pdf, 2018.

[40] F. Ceschin, M. Botacin, H. M. Gomes, L. S. Oliveira, and A. Grégio. Shallow security: On the creation of adversarial variants to evade machine learning-based malware detectors. In *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*, ROOTS'19, Vienna, Austria, 2019. Association for Computing Machinery.

[41] S.-T. Chen, Y. Han, D. H. Chau, C. Gates, M. Hart, and K. A. Roundy. Predicting cyber threats with virtual security products. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, page 189–199, New York, NY, USA, 2017. Association for Computing Machinery.

[42] C. Cimpanu. Turla hacker group steals antivirus logs to see if its malware was detected. https://www.zdnet.com/article/turla-hacker-group-steals-antivirus-logs-to-see-if-its-malware-was-detected/, 2020.

[43] CiscoTalos. Clamav. https://github.com/Cisco-Talos/clamav-devel, 2003.

[44] ClamAV. Creating signatures for clamav. https://www.clamav.net/documents/creating-signatures-for-clamav, 2003.

[45] ClamAV. File types. https://www.clamav.net/documents/clamav-file-types, 2003.

[46] ClamAV. How do i ignore whitelist a clamav signature? https://www.clamav.net/documents/how-do-i-ignore-whitelist-a-clamav-signature, 2003.

[47] ClamAV. On-access scanning. https://www.clamav.net/documents/on-access-scanning, 2003.

[48] ClamAV. Trusted and revoked certificates. https://www.clamav.net/documents/trusted-and-revoked-certificates, 2003.

[49] ClamAV. Using yara rules in clamav. https://www.clamav.net/documents/using-yara-rules-in-clamav, 2003.

[50] ClamAV. Whitelist databases. https://www.clamav.net/documents/whitelist-databases, 2003.

[51] ClamAV. Realtime protection with clamav on windows. https://blog.clamav.net/2011/02/realtime-protection-with-clamav-on.html, 2011.

[52] Clamav. Clamav. https://www.clamav.net/downloads\#collapseCVD, 2018.

[53] ClamWin. Free antivirus for windows. http://www.clamwin.com/, 2018.

[54] Comodo. Antivirus whitelist. https://securebox.comodo.com/antivirus-whitelist/, 2018.

[55] M. Cova, C. Leita, O. Thonnard, A. D. Keromytis, and M. Dacier. An analysis of rogue av campaigns. In S. Jha, R. Sommer, and C. Kreibich, editors, *Recent Advances in Intrusion Detection*, pages 442–463, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[56] CrowdStrike. Ngav defined. https://www.crowdstrike.com/epp-101/next-generation-antivirus-ngav/, 2020.

[57] D3VI5H4. Antivirus artifacts. https://github.com/D3VI5H4/Antivirus-Artifacts, 2020.

[58] D4stiny. How to use trend micro rootkit remover to install a rootkit. https://d4stiny.github.io/How-to-use-Trend-Micro-Rootkit-Remover-to-Install-a-Rootkit/, 2020.

[59] deresz. A script to reverse-engineer anti-virus signatures. https://github.com/deresz/avwhy, 2012.

[60] D. Deyannis, E. Papadogiannaki, G. Kalivianakis, G. Vasiliadis, and S. Ioannidis. Trustav: Practical and privacy preserving malware analysis in the cloud. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, CODASPY '20, page 39–48, New York, NY, USA, 2020. Association for Computing Machinery.

[61] N. K. Dien, T. T. Hieu, and T. N. Thinh. Memory-based multi-pattern signature scanning for clamav antivirus. In T. K. Dang, R. Wagner, E. Neuhold, M. Takizawa, J. Küng, and N. Thoai, editors, *Future Data and Security Engineering*, pages 58–70, Cham, 2014. Springer International Publishing.

[62] M. Dodel and G. Mesch. An integrated model for assessing cyber-safety behaviors: How cognitive, socioeconomic and digital determinants affect diverse safety practices. *Computers & Security*, 86:75 – 91, 2019.

[63] EICAR. Eicar test file. https://www.eicar.org/?page_id=3950, 2015.

[64] EMSISOFT. Why antivirus uses so much ram – and why that is actually a good thing! https://blog.emsisoft.com/2016/04/13/why-antivirus-uses-so-much-ram-and-why-that-is-actually-a-good-thing/, 2015.

[65] EricLaw. Spying on https. https://textslashplain.com/2019/08/11/spying-on-https/, 2019.

[66] erocarrera. pefile. https://github.com/erocarrera/pefile, 2016.

[67] ESET. Types of updates. http://support.eset.com/kb309/?viewlocale=en_US, 2018.

[68] R. Fedler, M. Kulicke, and J. Schütte. An antivirus api for android malware recognition. In *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, pages 77–84, 2013.

[69] FileGrab. Filegrab. https://sourceforge.net/projects/filegrab/, 2016.

[70] E. Filiol. Malware pattern scanning schemes secure against black-box analysis. *Journal in Computer Virology*, 2(1):35–50, Aug 2006.

[71] W. Fleshman, E. Raff, R. Zak, M. McLean, and C. Nicholas. Static malware detection subterfuge: Quantifying the robustness of machine learning and current anti-virus. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–10, 2018.

[72] FSecure. False positives. https://www.f-secure.com/v-descs/false_positive.shtml, 2019.

[73] S. Furnell and N. Clarke. Power to the people? the evolving recognition of human aspects of security. *Computers & Security*, 31(8):983 – 988, 2012.

[74] Geek. Defcon race to zero contest angers antivirus vendors. https://www.geek.com/news/defcon-race-to-zero-contest-angers-antivirus-vendors-57 2008.

[75] M. Gorelik. Machine learning can't protect you from fileless attacks. https://securityboulevard.com/2020/05/machine-learning-cant-protect-you-from-fileless-attacks/, 2020.

[76] K. Griffin, S. Schneider, X. Hu, and T.-c. Chiueh. *Automatic Generation of String Signatures for Malware Detection*, pages 101–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[77] N. B. Guinde and R. B. Lohani. Fpga based approach for signature based antivirus applications. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, page 1262–1263, New York, NY, USA, 2011. Association for Computing Machinery.

[78] HackerNews. Kaspersky antivirus flaw exposed users to cross-site tracking online. https://thehackernews.com/2019/08/kaspersky-antivirus-online-tracking.html, 2019.

[79] J. Haffejee and B. Irwin. Testing antivirus engines to determine their effectiveness as a security layer. In *2014 Information Security for South Africa*, pages 1–6, 2014.

[80] K. W. Hamlen, V. Mohan, M. M. Masud, L. Khan, and B. Thuraisingham. Exploiting an antivirus interface. *Computer Standards & Interfaces*, 31(6):1182 – 1189, 2009.

[81] Hanno. How kaspersky makes you vulnerable to the freak attack and other ways antivirus software lowers your https security. https://blog.hboeck.de/archives/869-How-Kaspersky-makes-you-vulnerable-to-the-FREAK-attack-and-other-ways-Antivirus-software-lowers-your-HTTPS-s
html, 2015.

[82] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[83] HookShark. Hookshark. https://www.unknowncheats.me/forum/pc-software/72799-hookshark64-beta-0-1-a.html, 2019.

[84] F. Hsu, M. Wu, C. Tso, C. Hsu, and C. Chen. Antivirus software shield against antivirus terminators. *IEEE Transactions on Information Forensics and Security*, 7(5):1439–1447, 2012.

[85] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. Le Traon, J. Klein, and L. Cavallaro. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 425–435, 2017.

[86] N. Hyvärinen. Detecting parent pid spoofing. https://blog.f-secure.com/detecting-parent-pid-spoofing/, 2018.

[87] N. Hyvärinen. Memory injection like a boss. https://blog.f-secure.com/memory-injection-like-a-boss/, 2018.

[88] InfoSecurity. Kaspersky lab hit by av software source code leak. https://www.infosecurity-magazine.com/news/kaspersky-lab-hit-by-av-software-source-code-leak/, 2011.

[89] iPower. Kasperskyhook. https://github.com/iPower/KasperskyHook, 2020.

[90] James. Upx visual studio. https://github.com/james34602/UPX-Visual-Studio, 2020.

[91] C. Jarabek, D. Barrera, and J. Aycock. Thinav: Truly lightweight mobile cloud-based anti-malware. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, page 209–218, New York, NY, USA, 2012. Association for Computing Machinery.

[92] Jareth. The pros, cons and limitations of ai and machine learning in antivirus software. https://blog.emsisoft.com/en/35668/the-pros-cons-and-limitations-of-ai-and-machine-learning-in-antivirus-software/, 2019.

[93] A. Kalysch, D. Bove, and T. Müller. How android's ui security is undermined by accessibility. In *Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium*, ROOTS, pages 2:1–2:10, New York, NY, USA, 2018. ACM.

[94] Kaspersky. Kaspersky lab utilizes nvidia technologies to enhance protection. https://www.kaspersky.com/about/press-releases/2009_kaspersky-lab-utilizes-nvidia-technologies-to-enhance-protection, 2009.

[95] Kaspersky. Features of using kaspersky anti-virus 2017 with third-party firewalls. https://support.kaspersky.com/12956, 2017.

[96] Kaspersky. How to run a scan task in kaspersky security cloud. https://support.kaspersky.com/us/13393#block6, 2018.

[97] Kaspersky. How to run a virus scan the right way: Step-by-step guide. https://www.kaspersky.com/resource-center/preemptive-safety/how-to-run-a-virus-scan, 2018.

[98] Kaspersky. Kaspersky security events in windows event log. https://support.kaspersky.com/KS4Exchange/9.4/en-US/127197.htm, 2018.

[99] Kaspersky. Whitelist program. https://usa.kaspersky.com/partners/whitelist-program, 2018.

[100] Kaspersky. About remediation engine. https://support.kaspersky.com/KESWin/11/en-us/151136.htm, 2019.

[101] Kaspersky. Configuring the facade module supporting application interaction with utilities and administration systems. https://support.kaspersky.com/KLMS/8.2/en-US/82367.htm, 2019.

[102] Kaspersky. Gaming mode on. https://www.kaspersky.co.in/gaming-mode-on/, 2020.

[103] Kaspersky. An immune-based approach to information system security. https://os.kaspersky.com/, 2020.

[104] Kaspersky. Installation error 27300 klhk.sys_x64 error code 2147024891. https://community.kaspersky.com/kaspersky-anti-virus-12/installation-error-27300-klhk-sys-x64-error-code-2147024891-8516, 2020.

[105] D. W. Kim, P. Yan, and J. Zhang. Detecting fake anti-virus software distribution webpages. *Computers & Security*, 49:95 – 106, 2015.

[106] J. Koret and E. Bachaalany. *The Antivirus Hacker's Handbook*. Wiley Publishing, 1st edition, 2015.

[107] P. Kováč. Fighting malware with machine learning. https://blog.avast.com/fighting-malware-with-machine-learning, 2018.

[108] J. Kraunelis, Y. Chen, Z. Ling, X. Fu, and W. Zhao. On malware leveraging the android accessibility framework. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 512–523. Springer, 2013.

[109] R. Kraus, B. Barber, M. Borkin, and N. J. Alpern. Chapter 6 - internet information services – web service attacks. In R. Kraus, B. Barber, M. Borkin, and N. J. Alpern, editors, *Seven Deadliest Microsoft Attacks*, pages 109 – 128. Syngress, Boston, 2010.

[110] Landave. Bitdefender: Upx unpacking featuring ten memory corruptions. https://landave.io/2020/11/bitdefender-upx-unpacking-featuring-ten-memory-corruptions/, 2020.

[111] F. L. Lévesque, S. Chiasson, A. Somayaji, and J. M. Fernandez. Technological and human factors of malware attacks: A computer security clinical trial approach. *ACM Trans. Priv. Secur.*, 21(4):18:1–18:30, July 2018.

[112] F. L. Levesque, A. Somayaji, D. Batchelder, and J. M. Fernandez. Measuring the health of antivirus ecosystems. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 101–109, 2015.

[113] m0n0ph1. Process hollowing. https://github.com/m0n0ph1/Process-Hollowing, 2015.

[114] F. Maggi, A. Bellini, G. Salvaneschi, and S. Zanero. Finding non-trivial malware naming inconsistencies. In S. Jajodia and C. Mazumdar, editors, *Information Systems Security*, pages 144–159, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[115] Malshare. Malshare. https://malshare.com/, 2018.

[116] MalwareBytes. Report false positive found with malwarebytes endpoint security. https://support.malwarebytes.com/hc/en-us/articles/360038523234-Report-false-positive-found-with-Malwarebytes-Endpoint-Security, 2019.

[117] Matterpreter. Defendercheck. https://github.com/matterpreter/DefenderCheck, 2019.

[118] Mattiwatti. Pplkiller. https://github.com/Mattiwatti/PPLKiller, 2016.

[119] McAffee. How to collect event trace logs, error tracing logs, and boot log tracing logs for host intrusion prevention 8.0 for windows. https://kc.mcafee.com/corporate/index?page=content&id=KB72868, 2018.

[120] Microsoft. Detecting reflective dll loading with windows defender atp. https://www.microsoft.com/security/blog/2017/11/13/detecting-reflective-dll-loading-with-windows-defender-atp/, 2017.

[121] Microsoft. How to create a boot-time global logger session. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/how-to-create-a-boot-time-global-logger-session, 2017.

[122] Microsoft. Tracing during boot. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/tracing-during-boot, 2017.

[123] Microsoft. Event_trace_properties structure. https://docs.microsoft.com/en-us/windows/win32/api/evntrace/ns-evntrace-event_trace_properties, 2018.

[124] Microsoft. Protecting anti-malware services. https://docs.microsoft.com/en-us/windows/win32/services/protecting-anti-malware-services-, 2018.

[125] Microsoft. Review event logs and error codes to troubleshoot issues with microsoft defender antivirus. https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-antivirus/troubleshoot-microsoft-defender-antivirus, 2018.

[126] Microsoft. When to use transactional ntfs. https://docs.microsoft.com/en-us/windows/win32/fileio/when-to-use-transactional-ntfs, 2018.

[127] Microsoft. Avscan file system minifilter driver. https://docs.microsoft.com/en-us/samples/microsoft/windows-driver-samples/avscan-file-system-minifilter-driver/, 2019.

[128] Microsoft. Ndis network interface architecture. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/ndis-network-interface-architecture, 2019.

[129] Microsoft. Sysinternals. https://docs.microsoft.com/en-us/sysinternals/, 2019.

[130] Microsoft. Freelibrary. https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-freelibrary, 2020.

[131] Microsoft. Introducing kernel data protection, a new platform security technology for preventing data corruption. https://www.microsoft.com/security/blog/2020/07/08/introducing-kernel-data-protection-a-new-platform-security-technology-for-preventing-data-corruption/, 2020.

[132] B. Min and V. Varadharajan. A novel malware for subversion of self-protection in anti-virus. *Software: Practice and Experience*, 46(3):361–379, 2016.

[133] B. Min, V. Varadharajan, U. Tupakula, and M. Hitchens. Antivirus security: naked during updates. *Software: Practice and Experience*, 44(10):1201–1222, 2014.

[134] F. Mira and W. Huang. Performance evaluation of string based malware detection methods. In *2018 24th International Conference on Automation and Computing (ICAC)*, pages 1–6, 2018.

[135] mitmproxy. mitmproxy is a free and open source interactive https proxy. https://mitmproxy.org/, 2017.

[136] MITRE. Cve. https://cve.mitre.org/, 2020.

[137] D. Mohammadbagher. Detecting thread injection by etw & one simple technique. https://www.peerlyst.com/posts/detecting-thread-injection-by-etw-and-one-simple-technique-damon-mohammadbagher, 2020.

[138] A. Mohanta and A. Saldanha. *Antivirus Engines*, pages 785–817. Apress, Berkeley, CA, 2020.

[139] M. Montanari and R. H. Campbell. Multi-aspect security configuration assessment. In *Proceedings of the 2nd ACM Workshop on Assurable and Usable Security Configuration*, SafeConfig '09, page 1–6, New York, NY, USA, 2009. Association for Computing Machinery.

[140] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, 2007.

[141] Mr-Un1k0d3r. Edrs. https://github.com/Mr-Un1k0d3r/EDRs, 2021.

[142] K. Murad, S. N.-u.-H. Shirazi, Y. B. Zikria, and N. Ikram. Evading virus detection using code obfuscation. In T.-h. Kim, Y.-h. Lee, B.-H. Kang, and D. Ślęzak, editors, *Future Generation Information Technology*, pages 394–401, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[143] S. Mustaca. Challenges for young anti-malware products today. https://www.virusbulletin.com/uploads/pdf/conference_slides/2019/VB2019-Mustaca.pdf, 2019.

[144] M. H. Nguyen, D. L. Nguyen, X. M. Nguyen, and T. T. Quan. Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning. *Computers & Security*, 76:128 – 155, 2018.

[145] Nirsoft. Dll export viewer. https://www.nirsoft.net/utils/dll_export_viewer.html, 2016.

[146] Nirsoft. Driverview. https://www.nirsoft.net/utils/driverview.html, 2016.

[147] NoVirusThanks. Dll uninjector. https://www.novirusthanks.org/products/dll-uninjector/, 2016.

[148] Nvidia. Chapter 35. fast virus signature matching on the gpu. https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-35-fast-virus-signature-matching-gpu, 2010.

[149] T. Ormandi. Sophail: A critical analysis of sophos antivirus. https://lock.cmpxchg8b.com/sophail.pdf, 2011.

[150] T. Ormandy. Loadlibrary. https://github.com/taviso/loadlibrary, 2017.

[151] PCMagazine. Google adds eset malware detection to chrome. https://www.pcmag.com/news/356830/google-adds-eset-malware-detection-to-chrome, 2017.

[152] I. Polakis, M. Diamantaris, T. Petsas, F. Maggi, and S. Ioannidis. Powerslave: Analyzing the energy consumption of mobile antivirus software. In M. Almgren, V. Gulisano, and F. Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 165–184, Cham, 2015. Springer International Publishing.

[153] ProcessHacker. Processhacker. https://github.com/processhacker/processhacker, 2016.

[154] G. ProjectZero. How to compromise the enterprise endpoint. https://googleprojectzero.blogspot.com/2016/06/how-to-compromise-enterprise-endpoint.html, 2016.

[155] Quarkslab. Guided tour inside windefender's network inspection driver. https://blog.quarkslab.com/guided-tour-inside-windefenders-network-inspection-driver.html, 2021.

[156] D. Quarta, F. Salvioni, A. Continella, and S. Zanero. Extended abstract: Toward systematically exploring antivirus engines. In C. Giuffrida, S. Bardin, and G. Blanc, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 393–403, Cham, 2018. Springer International Publishing.

[157] R. Raghunarayan. Antivirus is dead: How ai and machine learning will drive cybersecurity. https://techbeacon.com/security/antivirus-dead-how-ai-machine-learning-will-drive-cybersecurity, 2019.

[158] S. M. Rauen. Madcodehook description. http://www.madshi.net/madCodeHookDescription.htm, 2020.

[159] RegShot. Regshot. https://sourceforge.net/projects/regshot/, 2018.

[160] ReversingLabs. Reversinglabs yara rules. https://github.com/reversinglabs/reversinglabs-yara-rules, 2020.

[161] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. v. Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *2012 IEEE Symposium on Security and Privacy*, pages 65–79, 2012.

[162] K. A. Roundy and B. P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1), July 2013.

[163] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, pages 230–253, Cham, 2016. Springer International Publishing.

[164] M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, USA, 1st edition, 2012.

[165] sindoni. Kaspersky hooking engine analysis. https://quequero.org/2014/10/kaspersky-hooking-engine-analysis/, 2014.

[166] J. J. Singh, H. Samuel, and P. Zavarsky. Impact of paranoia levels on the effectiveness of the modsecurity web application firewall. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pages 141–144, 2018.

[167] Sophos. Default anti-virus scanning options for sophos central. https://community.sophos.com/kb/en-us/119637, 2016.

[168] Sophos. Sophos antivirus sdk. https://www.sophos.com/en-us/medialibrary/pdfs/factsheets/oem-solutions/sophos-antivirus-sdk-dsna.pdf, 2016.

[169] stephenfewer. Reflectivedllinjection. https://github.com/stephenfewer/ReflectiveDLLInjection, 2010.

[170] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. In S. Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 55–69, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[171] J. W. Stokes, J. C. Platt, H. J. Wang, J. Faulhaber, J. Keller, M. Marinescu, A. Thomas, and M. Gheorghescu. Scalable telemetry classification for automated malware detection. In S. Foresti, M. Yung, and F. Martinelli, editors, *Computer Security – ESORICS 2012*, pages 788–805, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[172] R. Sun, M. Botacin, N. Sapountzis, X. Yuan, M. Bishop, D. E. Porter, X. Li, A. Gregio, and D. Oliveira. A praise for defensive programming: Leveraginguncertainty for effective malware mitigation. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2020.

[173] T. Takahashi, C. Kruegel, G. Vigna, K. Yoshioka, and D. Inoue. Tracing and analyzing web access paths based on user-side data collection: How do users reach malicious urls?, 2020.

[174] talliberman. atom-bombing. https://github.com/BreakingMalwareResearch/atom-bombing, 2016.

[175] tanduRE. Avasthv project overview. https://github.com/tanduRE/AvastHV/tree/master/AvastHV, 2019.

[176] tcpdump. Tcpdump & libpcap. https://www.tcpdump.org/, 2014.

[177] U. Team. the ultimate packer for executables. https://upx.github.io/, 1999.

[178] TheHackerNews. Windows built-in antivirus gets secure sandbox mode – turn it on. https://thehackernews.com/2018/10/windows-defender-antivirus-sandbox.html, 2018.

[179] TrendMicro. Decrypt encrypted quarantine files. https://docs.trendmicro.com/all/ent/iwsva/v6.5_sp2/en-us/iwsva_6.5_sp2_online_help/decrypt_encrypted_quarantine_files.htm, 2007.

[180] TrendMicro. Autorun. https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/autorun, 2012.

[181] TrendMicro. Reporting a false positive issue in deep security. https://success.trendmicro.com/solution/1119869-reporting-a-false-positive-issue-in-deep-security, 2018.

[182] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*, pages 659–673, 2015.

[183] X. Ugarte-Pedrero, M. Graziano, and D. Balzarotti. A close look at a daily dataset of malware samples. *ACM Trans. Priv. Secur.*, 22(1), Jan. 2019.

[184] D. Uluski, M. Moffie, and D. Kaeli. Characterizing antivirus workload execution. *SIGARCH Comput. Archit. News*, 33(1):90–98, Mar. 2005.

[185] Unspecified. Mydoom: Do you "get it" yet? *Network Security*, 2004(2):13 – 15, 2004.

[186] VirusTotal. Virustotal. https://www.virustotal.com/gui/home/upload, 2019.

[187] Virustotal. Yara - the pattern matching swiss knife for malware researchers (and everyone else). https://virustotal.github.io/yara/, 2019.

[188] VMware. What is next-generation antivirus (ngav)? https://www.carbonblack.com/definitions/what-is-next-generation-antivirus-ngav/, 2020.

[189] VxUnderground. Vxunderground. https://vx-underground.org/samples.html, 2020.

[190] A. Wheeler and N. Mehta. 0wning anti-virus: Weaknesses in a critical security component. https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-wheeler.pdf, 2005.

[191] Z. Whittaker. Anonymous leaks symantec's norton anti-virus source code. https://www.zdnet.com/article/anonymous-leaks-symantecs-norton-anti-virus-source-code/, 2012.

[192] C. Wressnegger, K. Freeman, F. Yamaguchi, and K. Rieck. Automatically inferring malware signatures for anti-virus assisted attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 587–598, New York, NY, USA, 2017. Association for Computing Machinery.

[193] Xiao-bin Wang, Guang-yuan Yang, Yi-chao Li, and Dan Liu. Review on the application of artificial intelligence in antivirus detection systemi. In *2008 IEEE Conference on Cybernetics and Intelligent Systems*, pages 506–509, Sep. 2008.

[194] Yara. Yara rules. `https://github.com/Yara-Rules/rules`, 2018.

[195] I. Zelinka, S. Das, L. Sikora, and R. Šenkeřík. Swarm virus - next-generation virus and antivirus paradigm? *Swarm and Evolutionary Computation*, 43:207 – 224, 2018.

[196] Y. Zhang, L. Wu, F. Xia, and X. Liu. Immunity-based model for malicious code detection. In D.-S. Huang, Z. Zhao, V. Bevilacqua, and J. C. Figueroa, editors, *Advanced Intelligent Computing Theories and Applications*, pages 399–406, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[197] S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, and G. Wang. Measuring and modeling the label dynamics of online anti-malware engines. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2361–2378. USENIX Association, Aug. 2020.

[198] ZoneAlarm. Zonealarm cloud scanning policy. `https://www.zonealarm.com/about/cloud-scan-policy`, 2018.

[199] A. Zsigovits. Upx anti-unpacking techniques in iot malware. `https://cujo.com/upx-anti-unpacking-techniques-in-iot-malware/`, 2020.

A AV's Libraries

Table 19: **Avast.** Libraries.

Library	Description
aswScan	Low level antivirus engine
aswBoot64	start-up scanner
ring_lient	Ring module
burger_lient	Burger Client
aswUtil	Utility
aswJsFlt	Script Blocking filter
fwAux	Firewall Helper
lim	License Manager
streamback	StreamBack
asOutExt	AsOutExt Module
aswStrm	Streaming Update
ashShell	Shell Extension
aswSqLt	SQLite
Base	English Basic Module
event_anager_a	Google Analytics Event Consumer
aswRvrt	aswRvrt support
uiext	UI extension
aswW8ntf	metro notification
tasks_ore	task core
aswEngin	High level antivirus engine
swhealthex2	Software Health extension
aswRegLib	Registry editor
aswwinamapi	Metro Application Healer
aswntsqlite	NT SQLite
CommChannel	Communication Channels
AavmRpch	AAVM Remote Procedure Call
aswdetallocator	Det
TuneupSmartScan	Cleanup Smartscan extension
libcef	Chromium Embedded Framework
anen	Adapter Network Event Notifier.
libcrypto-1_-x64	OpenSSL
shepherdsync	Shepherd Syncer
process_onitor	Process Monitor
serialization	Serialization
event_outing	Event Routing
aswCmnBS	Common functions
chrome_lf	Chromium
wsc	security center dll
libEGL	ANGLE libEGL Dynamic Link
browser_ass	Browser Pass
aswProperty	Property Storage
aswRep	Reputation services access
aswLog	Log
libssl-1_-x64	OpenSSL
aswSecDns	SecureDNS engine
aswIP	IP Dynamic Link
aswDld	aswDld Dynamic Link
rescue_isk	Rescue Disk
aswidpm	IDP Monitor
pam	Password Manager
ArPot	ArPot usermode dll component
mfc140u	MFCDLL Shared - Retail Version
Aavm4h	Asynchronous Virus Monitor (AAVM)
algo64	Low level antivirus engine

Table 19: **Avast.** Libraries (continued from previous page)

Library	Description
aswCmnIS64	Independent functions
vaarclient	vaarclient
aswEngLdr(1)	Engine loader
aswPatchMgt	Software Health
gui_ache	GUI cache
aswEngLdr	Antivirus engine loader
firefox_ass	Firefox Pass
aswCmnIS	Antivirus independent functions
aswremoval	Removal engine
health	Property Storage
aswPropertyAv	AV Property Storage
CommonUI	Common UI layer
fltlib_rapper	Property Storage
event_anager_urger	Burger Event Consumer
aswAux	Auxiliary
dll_oader	dll loader
libGLESv2	ANGLE libGLESv2 Dynamic Link
ashServ	antivirus service
module_ifetime	module lifetime
gaming_ode_i	Gaming Mode
aswJsFlt64	Script Blocking filter
uiLangRes	UILangRes
Boot	Portuguese Boot Scanner Module
custody	Cyber-Capture
aswsys	SYS
network_otifications	network notifications
BCUEngine	Browser Cleanup Engine
ffl2	FF v2
CommonRes	Common UI resources
dnd_elper	Gaming Mode DND helper
aswData	UI Layer
event_outing_pc	Event Routing RPC
aswRawFS64	Raw disk access
gaming_ode	Gaming Mode
HTMLayout	HTMLayout
aswcomm	Communication Module
aswidplog	Logging
aswBrowser	SafeZone Browser
aswpsic	Persistent Stream Information Client
ashBase	Basic Functionality Module
ashTask	Task Handling Module
event_anager	Event Manager
PushPin	PushPin
aswAMSI	AMSI COM object
Cef_enderer	Property Storage
Edge_enderer	Property Storage
ashTaskEx	TaskEx
gaming_robe	Gaming Mode Probe
aswhook	Hook
snxhk	snxhk
aswDataScan	DataScan
aswVmm	aswVmm comm
aswHds	Home Network Security
aswcml	CML
instup	Antivirus Installer

Table 19: **Avast.** Libraries (continued from previous page)

Library	Description
event_anager_r	Event Consumer
Sf2	Dynamic binary instrumentator
log	Logging
exts	Antivirus Scanner Extension
aswAR	anti-rootkit module
aswCmnOS	Antivirus HW dependent
aswCleanerDLL	Virus/Worm Cleaner
aswsecapi	Secure API
aswFiDb	File information database access

Table 20: **F-Secure.** Libraries

Library	Description
ICUDT54	ICU Data DLL
dbghelp	Windows Image Helper
fs_e_ttps	Enhanced HTTPS support for IE
orspapi64	ORSP API DLL 32-bit (Release)
gkhsm64	Gatekeeper Handler 64-bit
aevdf	Avira Engine Module for Windows
spapi64	Scanning API 64-bit
fs_cf_lient_uth_2	Client Authentication API
fs_vents_pi_2	Product Events API
Qt5Core_SC	C++ application development framework.
fsvirgo64	Virgo engine
hashlib_64	Hashing 32-bit
F-Secure.Ipc	.Ipc
fs_icense_i_2	Licensing UI
OnlineSafety	Online Safety plug-in for CUIF
F-Secure.Tools	.Tools
HelpPlugin	Help Plugin
spapi32	Scanning API 32-bit
SystemInfo	System Info Plug-in, 32 bit
fs_vents_pi_4	Product Events API
aeoffice	Avira Engine Module for Windows
aecrypto	Avira Engine Module for Windows
qico	C++ application development framework.
hotfix_lugin	Ultralight Hotfix Plugin
qwindows	C++ application development framework.
fsetw_pi64	ETW API 64-bit
ICUIN54	ICU I18N DLL
CuifApi64	CuifApi
SupportView	.Settings.SupportView
fs_cf_lient_uth_4	Client Authentication API
fs_cf_anager_lugin_2	Host Process Manager Plugin
fs_ustomization_eader_4	Customization Reader
F-Secure.ClientAuth.Api	.ClientAuth.Api
aeemu	Avira Engine Module for Windows
ICUUC54	ICU Common DLL
FsEventsPlugin	Product Events Plugin
qgif	C++ application development framework.
xvdfmerge	AVIRA XVDF merge
fs_cf_ush_lugin_2	Push Notification Plugin
fs_cf_osmos_lugin_2	COSMOS plugin

Table 20: **F-Secure.** Libraries (continued from previous page)

Library	Description
F-Secure.Settings.Model	.Settings.Model
fs_cf_osmos_4	COSMOS API
qsvg	C++ application development framework.
F-Secure.Latebound	.Latebound
fs_ush_otif_lugin_2	Push Notification plugin
fsclm	Crypto
fsliball	fslib full bundle
F-Secure.Settings.Api	.Settings.Api
CCFDLLHosterAPI_4	Host Process API
fships	HIPS Logic module (Release)
settings_pstream_lugin_2	Settings Upstream Plugin
Qt5Sensors_SC	C++ application development framework.
fs_ubscription_eminder_2	Subscription Reminder
apcfile	APC SDK
F-Secure.Cuif.Api	.Cuif.Api
fs_ls_i_lg_ontentfilter64	Network Interceptor Content Filter plugin, 64 bit
fs_ettings_onverter_lugin_2	Settings Converter Plugin
CommonSettingsWidgets	Common Settings Widgets
aelibinf	Avira Engine Module for Windows
daas2inst_4	daas2inst
CuifSimpleAction	Simple Action plug-in
CuifWidgets	CuifWidgets
daas2	daas2
F-Secure.AutomaticUpdateAgent.Api	.AutomaticUpdateAgent.Api
qsqlite	C++ application development framework.
ManagementAgent	Management Agent Plug-in, 32 bit
Help	Help Plug-in, 32 bit
FsShellExtension32	Anti Virus Shell Extension Plug-in, 32 bit
Qt5Sql_SC	C++ application development framework.
fslynx	Lynx Engine 64-bit
fs_estart_lugin_2	OneClient Restart Plugin
fs_cf_i_lg_anking_rotection64	Network Interceptor Banking Protection plugin, 64 bit
ParserFramework	ParserFramework
FsPiscesClient	Pisces Client x64
Qt5Multimedia_SC	C++ application development framework.
fs_neclient_ore_lugin_2	OneClient Core Plugin
daas2_64	daas2
CommonSettingsPlugin	Common Settings Plugin
fs_cf_id64	Network Interceptor Daemon, 64 bit
Qt5Help_SC	C++ application development framework.
fs_ult_lugin_2	EULT plugin
Qt5Xml_SC	C++ application development framework.
senddump_shoster_lugin64	Senddump Hoster Plugin
fs_cf_atapipeline_pi_2	Data Pipeline API
aeheur	Avira Engine Module for Windows
F-Secure.Settings.NotificationsView	.Settings.NotificationsView
fshook32	HIPS user-mode hooking module (Release)
Qt5MultimediaWidgets_SC	C++ application development framework.
wsc_lugin64	WSC Plugin
aehelp	Avira Engine Module for Windows
Qt5WebKit_SC	C++ application development framework.
json_64	json-c Dynamic Link
ControlLayer	ControlLayer
fs_cf_uts2_lugin_2	GUTS2 Plugin
aescript	Avira Engine Module for Windows

Table 20: **F-Secure.** Libraries (continued from previous page)

Library	Description
ExpressionEngine	ExpressionEngine
fs_in_tore_pp_pi_4	Winstore Application API 32-bit
fshook64	HIPS user-mode hooking module (Release)
ssleay32	OpenSSL Shared
Qt5Svg_SC	C++ application development framework.
aepack	Avira Engine Module for Windows
fs_lyer_pi_4	Flyer API
fsamsi32	AMSI Client
F-Secure.Sp.Api	.Sp.Api
DataLayer	DataLayer
qjpeg	C++ application development framework.
Qt5Network_SC	C++ application development framework.
fs_ecl_2	Service Enabler Client
fm4av	File Management x64
Qt5Quick_SC	C++ application development framework.
fs_cf_atapipeline_pi_4	Data Pipeline API
fsaua_pi_ll	AUA API
F-Secure.Settings.ContentControlView	.Settings.ContentControlView
zlib_2	zlib data compression
libeay32	OpenSSL Shared
fs_ecl_4	Service Enabler Client
aerdl	Avira Engine Module for Windows
fs_cf_i_lg_lockpage64	Network Interceptor Block Page plugin, 64 bit
DeclarationHandler	DeclarationHandler
fs_ustomization_eader_2	Customization Reader
apchash	AVIRA APC hash file calculator
ControlPanelTools	Online Safety Control Panel Tools plug-in for CUIF
F-Secure.NLog.Extension	.NLog.Extension
fs_oaster_2	Toaster
CuifApi	CuifApi
fsclm64	Crypto
Localization	Localization Framework
sqlite3_2	SQLite
orspplug64	ORSP Client DLL 32-bit (Release)
fs_e_ttps64	Enhanced HTTPS support for IE
fs_lu_oster_lugin64	ULU Hoster Plugin
F-Secure.CrashDump	.CrashDump
F-Secure.Settings.Commands	.Settings.Commands
Qt5WebKitWidgets_SC	C++ application development framework.
Qt5Widgets_SC	C++ application development framework.
fsusscr	Universal System Scanner Core 64-bit
CuifTypes	CuifTypes
F-Secure.OneClient.Api	.OneClient.Api
aescn	Avira Engine Module for Windows
fs_lyer_lugin_2	Flyer Plugin
aeexp	Avira Engine Module for Windows
sqlite3_4	SQLite
Qt5WebChannel_SC	C++ application development framework.
fsetw_lugin64	ETW hoster plugin 64-bit
fs_neclient_pi_4	OneClient API
LocaleInfo	Locale Info Plug-in, 32 bit
fs_cf_ction_enter_pi_2	Action Center API
fs_vents32	Product Events
daas2inst_2	daas2inst
CuifWebKit	CuifWebKit

Table 20: **F-Secure.** Libraries (continued from previous page)

Library	Description
CCFIPC64	IPC
Qt5PrintSupport_SC	C++ application development framework.
fs_s_tatus_otification	Computer Security Status Notification Plug-in, 32 bit
NLog	NLog for .NET Framework 4.5
aedroid	Avira Engine Module for Windows
Qt5Gui_SC	C++ application development framework.
fs_cf_ush_pi_2	Push Notification API
fsaua_pi_ll64	AUA API
fs_cf_ownload_2	Download
fs_otfix_lugin_2	Hotfix Plugin
fs_cf_etrics_lugin_2	CCF Metrics Plugin
F-Secure.SettingsUI.Plugin.Api	.SettingsUI.Plugin.Api
CCFDLLHosterAPI	Host Process API
qrt	Qrt dll for WinNT
F-Secure.Styles	.Styles.Consumer
obusclient2_4	OBUS Client
F-Secure.Wpf.Converters	.Wpf.Converters
fs_ray_con_2	Tray Icon Plugin
CCFIPC	IPC
fs_cf_lient_uth_lugin_2	Client Authentication Plugin
HelpWidgets	Help Widgets
Qt5Positioning_SC	C++ application development framework.
avdaemon	Antivirus Daemon
7z	7z Plugin
fs_neclient_pi_2	OneClient API
JsonParser	JsonParser
F-SecureLoader	.DllLoader
ActionCenterPlugin	Action Center Plugin
Newtonsoft.Json	Json.NET
aesbx	Avira Engine Module for Windows
fs_cf_oster_ontrol_lugin_2	Host Process Control Plugin
Licensing	Licensing Plug-in, 32 bit
ProductInfo	Product Info Plug-in, 32 bit
fs_aming_ode_2	Gaming Mode
capricorn64	Engine
aebb	Avira Engine Module for Windows
F-Secure.Settings.SecurityView	.Settings.SecurityView
fsamsi64	AMSI Client
aemobile	Avira Engine Module for Windows
AntiVirus	Anti Virus Feature Plug-in, 32 bit
savapi	Avira Savapi
Qt5OpenGL_SC	C++ application development framework.
OnlineSafetyWidgets	Online Safety Widgets plug-in for CUIF
FsShellExtension64	Anti Virus Shell Extension Plug-in, 64 bit
aegen	Avira Engine Module for Windows
F-Secure.Datapipeline.Api	.Datapipeline.Api
F-Secure.Cosmos.Api	.Cosmos.Api
fshive2	Anti-Virus 64-bit
fs_lyer_pi_2	Flyer API
json_	json-c Dynamic Link
Qt5Qml_SC	C++ application development framework.
aecore	Avira Engine Module for Windows
fs_cf_osmos_2	COSMOS API
fsecr64	Hydra Scan Engine

Table 21: **Kaspersky.** Libraries.

Library	Description
klsihk64l	
encryption_rypto_isk_egacy	Container reader library
Nemerle.Peg	Nemerle.Peg
wdiskio	WDiskIO
kasperskylab.ui.common	KasperskyLab.UI.Common
winlibhlpr	WINLIBHLPR
crypto_sl__	OpenSSL library
ushata	Ushata module
ie_lugin	Kaspersky Protection plugins
uds	
kasperskylab.ui.platform.toasts	KasperskyLab.UI.Platform.Toasts
licensing_roduct_acade	Licensing PDK facade
avzkrnl	AVZ Kernel
task_cheduler_andler	Kaspersky Anti-Virus Task Scheduler Handler
fsdrvplg	Plugin for FSDrv
ksdeinst	Modularity configurator
parental_ontrol_acade	Parental control facade component
kas_roduct	KASEngine EKA library
explode	Explode Transformer plugin
base64	Base64
cf_ngines	Content Filtering Engines
ac_acade	Application Control Facade
Microsoft.Practices.Prism.Interactivity	Microsoft.Practices.Prism.Interactivity
report	Report System
wifi_rotection	Wifi Protection
shellex	Shell Extension
Microsoft.Practices.Prism	Microsoft.Practices.Prism
kpcengine	KPC Engine
dblite	SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine.
remote_ka_rague_oader	Helper Library
icuin58	ICU I18N DLL
passdmap	PASSDMAP
apuhttps.dbf98b5663cedeeca7c9fb257aec8496_	
kasperskylab.ui.platform.ipm	
kasperskylab.ui.platform.htmltoinlinesconverter	KasperskyLab.Ui.Platform.HtmlToInlinesConverter
kasperskylab.ui.platform.balloons	KasperskyLab.UI.Platform.Balloons
kasperskylab.kis.ui.balloons	KasperskyLab.Kis.UI.Balloons
content_iltering_eta	Kaspersky content filtering pdk meta
kasperskylab.platform.localization.core	KasperskyLab.Platform.Localization.Core.Pipeline
icuuc58	ICU Common DLL
buffer	BUFFER
regmap	REGISTRY_APPER
msoe	MSOE
kasperskylab.ui.platform.reports.dataaccess	KasperskyLab.UI.Platform.Reports.DataAccess
kas_pconvert	Convert dynamic library
prseqio	SEQIO
xorio	ZIP MiniArchiver plugin
sw_eta	System Watcher Meta Information
am_ore	
backup_acade	Backup service facade
kasperskylab.kis.ui.shell	KasperskyLab.Kis.UI.Shell
encryption_rypto_isk_acade	Encryption Crypto Disk Facade

Table 21: **Kaspersky.** Libraries (continued from previous page)

Library	Description
unshrink	Unshrink Transformer plugin
kas_ngine	KAS-Engine dynamic library
icudt58	ICU Data DLL
backup_etainfo	Backup metainfo
app_ontrol	Application Control EKA
vkbd2x64	Virtual Keyboard
unstored	Unstored Transformer plugin
sys_ritical_bj.dfc5f69e9651d6738c03b143749621b9_	System Critical Objects
bl_sde	KL Product Business Logic
dumpwriter	Kaspersky Dump Writer DLL
browser_ntegration	Browser Integration
backup	Backup service
weak_ettings	Weak Settings Monitor
stdcomp	STDCOMPARE
heurap.03133393e594e8575b749797bdbb245a_	Heuristic anti-phishing service component
kasperskylab.ui.platform.safemoney	
kasperskylab.ui.core.visuals	KasperskyLab.UI.Core.Visuals
cf_acade	Content filtering facade component
icuio58	ICU I/O DLL
shell_ervice	Shell Service
hashsha1	Hash SHA1 algorithm implement
prutil	Utility Object Library
kas_oader	KASEngine EKA library
winreg	WINREG
klhkum	System Interceptors PDK usermode service interceptor
um_nterceptors_ontroller.cb0ca46bf7ea334ea0760a193400b9bb_	
klsihk64	
Microsoft.Practices.ServiceLocation	Microsoft.Practices.ServiceLocation
mdmap	Multipart Direct Mapper plugin
unreduce	Unreduce Transformer plugin
avpservice	Kaspersky Anti-Virus Service library
ucp_gent	UCP agent service
getsysteminfo	Kaspersky Get System Information
dtreg	DTREG
kl_ervice	Component service provider
cbi	KAV CBI DLL
ckahrule	
pxstub	Proxy Stubs
am_atch_anagement	
backup_ngine	Backup engine
ntfsstrm	NTFSSTREAM
avengine.1cc4216c3de3e9ae3f99b74e56a3763c_	AV engine component
kasperskylab.ksde.ui	KasperskyLab.Ksde.UI
klfphc	Filtering Platform Helper Class
klavasyswatch.2b5073fd10ee2a70bbc6f311743d63e0_	Heuristics proactive detection module
office_ntivirus	Kaspersky OfficeAntivirus Component
kasperskylab.pure.restoretool.nativeinterop	Restore tool native interop
deflate	Deflate Transformer plugin
kerneltracecontrol	Performance Analyzer Kernel Tracing Control Library
kasperskylab.platform.nativeinterop	Native interop assembly
system_ervice_ilter	
kasperskylab.ui.platform.views	KasperskyLab.UI.Platform.Views
cf_gmt_acade	Content filtering facade management component
updater_acade	
uniarc	UniArchiver plugin

Table 21: **Kaspersky.** Libraries (continued from previous page)

Library	Description
base64p	Base64P
plugins_eta	Kaspersky plugins pdk meta
params	Structure Serializer
antimalware_rovider	Kaspersky AntiMalwareProvider Component
schedule	Scheduler
kasperskylab.pure.ui.backup	KasperskyLab.Pure.UI.Backup
am_in_ux	
uninstallation_ssistant	Uninstallation assistant
encryption_rypto_isk_eta	Encryption Crypto Disk Meta
avpmain	Kaspersky Anti-Virus
mailmsg	MAILMSG
dmap	Direct Mapper plugin
ckahum	
crypto_omponents	
swpragueplugin	System Watcher PRAGUE proxy
unlzx	UnLZX Transformer plugin
minizip	ZIP MiniArchiver plugin
interprecz.b9ad9b743eed4c3a69ff722e60301e79_	Application Control Interpreter Recognizer
ndetect	Nertwork Detection
inproc_gent	Kaspersky Inproc Agent
system_nterceptors	
Nemerle	Nemerle Library
System.Windows.Interactivity	System.Windows.Interactivity
localization_anager	Localization Manager
hashmd5	HASHMD5
product_nfo	Kaspersky Product Info library
clldr	CLLDR Protection Library
ksdeuimain	Kaspersky Secure Connection
openssl_erifier	
propmap	PROPMAP
stored	Stored Transformer plugin
bi_acade	Browser Integration PDK facade
kasperskylab.ui.platform.services	Loader
kasperskylab.kis.ui.loader	Loader
installation_ssistant_eta	Installation assistant meta
winevent_nterceptor_ontroller	WinEvent Interceptor Controller
rar	RAR
plugins_acade	Kaspersky Anti-Virus Plugins PDK facade
pctrlex	Parental Control
network_ervices	Network services library
restore_ool_ervice	Restore tool service
nfio	NFIO
volenum	Volume enumeration
cd_ervice_rovider	
timer	Timer
si_onitor.d37220ecb715f59a66c797dafb8b265a_	
reportdb	Report DB System
activated_rocess_ategorization	Activated Process Categorization
bl	Product Business Logic
storage	
kas_ds	UDS dynamic library
prremote	PR_EMOTE
klsihk	
kasperskylab.ui.platform.reports	
tun_acade	

Table 21: **Kaspersky.** Libraries (continued from previous page)

Library	Description
ksn_acade	
System.Data.SQLite	System.Data.SQLite Core
crpthlpr	CryptoHelper
kasperskylab.pure.backupdiskscanner	KasperskyLab.Pure.BackupDiskScanner
app_ontrol_rague	Application Control Prague
mailer	Mailer library
kasperskylab.ui.core	KasperskyLab.UI.Core
avzscan	AVZ Scanner
mapiedk	MAPI and EDK library
kasperskylab.ksde.ui.loader	Loader
vkbd2	Virtual Keyboard
inifile	IniFile
ckahcomm	
superio	SUPERIO
installation_ssistant	Installation assistant
ipm_ervice	
kas_iltration	Content Filtration dynamic library
app_ore_eta	
wlengine.5074250125131bd6a0842583c51cbd6d_	Application Control Whitelist Engine
instrumental_eta	Instrumental Meta Library
wmihlpr	wmi helper
system_nterceptors_eta	
kpm_ntegration	KPM integration module
instrumental_ervices	Instrumental services
kasperskylab.kis.ui.visuals	KasperskyLab.Kis.UI.Visual
mdb	MDB
application_nvestigator	Application Control Application Investigator
antispam	AntiSpam mail fiter
mcou	Kaspersky Anti-Virus Mail Checker Outlook Plug-In
kasperskylab.kis.ui	KasperskyLab.Kis.UI
quantum	QUANTUM
safe_anking	Safe Banking
inflate	Inflate Transformer plugin
kasperskylab.ksde.nativeinterop	Native interop assembly
ekasyswatch	System Watcher EKA Task
avpuimain	Kaspersky Anti-Virus
prcore	Prague Core
app_ore_egacy	
kasperskylab.kis.ui.reports.dataaccess	KasperskyLab.Kis.UI.Reports.DataAccess
product_etainfo	Product Metainformation
cm_m	Cryptographic Module x86 (56 bit)
crypto_rovider	
kas_sg	GSG dynamic library
btdisk	Disk boot area parser
thpimpl	Thread Pool
fssync	
traffic_rocessing	Traffic Processing PDK
avpinst	Modularity configurator

Table 22: **Symantec.** Libraries.

Library	Description
FWCore	Firewall Core Component
Engine	InstallToolBox Engine
coActMgr	coActMgr
UISSSH	file description missing
wpMcPlg	Webcam Protection MC Plugin
FWHelper	Firewall Utilities
ccScanW	Symantec Scan Engine
rcEmlPxy	Symantec Email Proxy Resources
sds_ppendix__64	Symantec Static Data Scanner Component Library
cctFW	Norton Protection Center cctFW
SymHTML	Symantec HTML Interface
SymRdrSv	Symantec Redirector Service Plugin
NPCTray	Norton Protection Center System Tray
AVPSVC32	Norton Security Antivirus Product Service Module
speng64	Symantec Platform Component Library
EventSvc	Event Service
IronMigr	Symantec Iron Data Migration
ELAMCli64	Symantec ELAM
nsWscCtl	Norton Security WSC Control
wpNotify	Webcam Protection Notify
NISPInst	NIS Patch Installer
ccVrTrst	Symantec Trust Validation Engine
IPSEng32	IPS Script Engine DLL
SDKWrap	Security SDK Wrapper
muis	Shortcut MUI Resource
jwNCU	Browser and Temporary File Cleaner Job Worker
coSvcPlg	coServicePlugIn
buUIPlg	Backup UI Plugin
sds_ppendix__64	Symantec Static Data Scanner Component Library
csdklog	Client SDK Log
IDSxpx86	Intrusion Detection Interface DLL
coIDSafe	coIDSafe
CSDKSH	Symantec CSDKSH
tuUI	Tuneup UI
coDataPr	coDataProvider
MClnTask	M Client Task
SNDSvc	Symantec Network Service Plugin
CoIEPlg	coIEPlugIn
SpocClnt	SPOC Client
libcef	Chromium Embedded Framework (CEF) Dynamic Link Library
cuTFPlg	Temporary File Cleanup Plugin
rcSvcHst	Symantec ccServiceHost Resources
FwSesAl	Firewall Session Component
Eraser64	Symantec Eraser Engine
chrome_lf	Chromium
ccGLog	Symantec ccGenericLog Engine
diLueCbk	InstallToolBox LUE Callback
csdkprod	Client SDK Product Integration
BHSvcPlg	BASH Service Plugin
cceraser	Symantec Eraser Engine
sds_ppendix__86	Symantec Static Data Scanner Component Library
muis.mui	Shortcut MUI Resource
sticprxy	Submission Library
buComm	Backup Common

Table 22: **Symantec.** Libraries (continued from previous page)

Library	Description
buProv	Backup Providers
SQLite	SQLite
ncpBrExt	Norton Communication Platform NCPUI
buFScsdk	Backup FScsdk
ccEmlPxy	Symantec Email Proxy
SymDltCl	SymDelta client DLL
rcErrDsp	Symantec Error Display Resources
csdktu	Tuneup Client SDK Service
uiMetroN	Norton Metro Notifications
NspEng	NSP Client Backup Engine
coWPPlg	coWebAuthPlugIn
QBackup	Quarantine/Backup Engine
FWGenPlg	Firewall Generic Plug-in
nasascr	file description missing
ISDataSv	IS Data Service
SDKCmn	Security Status Server
NUMEng	Norton Update Manager Engine
spsvc	Symantec Platform Component Library
cltLMS	Symantec Shared Component
RuleXprt	Rule Database Upgrader library
InsImage	InstallToolBox Setup
buUI	Backup UI
QSPlugin	QuickStart Service Plugin
ScanLess	Norton Protection Center ScanLess UI Library
patch25d	Microdefs Apply Engine
ProxyClt	Proxy Client
buVssVst	Backup Volume Shadow Support For Vista
v2Client	v2 Client
ccIPC	Symantec ccIPC Engine
ccSvc	Symantec ccService Engine
DSCli	Symantec Data Store
ccJobMgr	Symantec ccJobMgr Engine
TaskWiz	Norton Protection Center N360 Task Wizard
cltFE	Symantec Shared Component
csdkaux	CSDK Client Auxiliary Interface
asEngine	AntiSpam Engine
tuMCFPlg	Tuneup Message Center Plugin
MsouPlug	AntiSpam MS Outlook Plugin
asHelper	AntiSpam Helper
buShell	Backup Shell
DefUtDCD	Symantec Definition Utilities
SHUIROL	file description missing
SymNeti	Symantec Network Driver Interface
DSCli64	Symantec Data Store
coParse	coParse
OEHeur	Symantec OEH
hsui	Norton Protection Center Help and Support
UMEngx86	SONAR Engine
coMCPlug	Message Center PlugIn
SymHTTP	Symantec HTTP Transport
PatchUI	InstallToolBox Setup
DuLuCbk	Symantec Definitions Deployment
ccGEvt	Symantec ccGenericEvent Engine
EFACli64	Symantec Extended File Attributes
msl	Symantec MS Light Library

Table 22: **Symantec.** Libraries (continued from previous page)

Library	Description
ccSubEng	Submission Engine
diStRptr	Stat Reporter Job Worker
srtsp64	Symantec AutoProtect
csdk	Client SDK
NavShExt	Norton Security Shell Extension Module
cltAlDis	Symantec Shared Component
FWSetup	Firewall Setup Utility
bbRGen64	Rule Preprocessor
sds_ngine_64	Symantec Static Data Scanner Component Library
wpCSDK	Webcam Protection CSDK Service
AVPAPP32	Norton Security Antivirus Product Application Module
AppMgr32	Symantec Application Core Manager
coUICtlr	Norton Password Manager
NScClt	Scandium Client
DiagRpt	Diagnostic Report
AVModule	Symantec AntiVirus Module
symhtml	Symantec HTML Interface
fwMCPlug	Firewall Message Center Plug-in
NAVLogV	Norton Security NAVLogV
coSfShre	coSafeShare
tuTW	Tuneup Task Wizard Plugin
FFPrefs	N360 FireFox Preferences Component
IDSAux	Intrusion Detection Auxiliary DLL
BuEng	Backup Engine x64
cuEng	Cleanup Engine
NSSSH	Symantec HTML Interface
IPSPlug	Symantec Intrusion Prevention Plugin
uiAlert	Norton Protection Center Alert Provider
AVifc	Symantec AntiVirus Interface
IPSEng64	IPS Script Engine DLL
srtspscan	Symantec AutoProtect
RuleUI	Rule UI
diArkive	InstallToolBox Archive
coShdObj	coShdObj
NumGui	Norton Update Manager Gui
symv8hst	Symantec Support Library
PeekUI	Norton Protection Center Peek User Interface Component
IDSXpx64	Intrusion Detection Interface DLL
diMaster	InstallToolBox Service
SymRedir	Symantec Redirector Interface DLL
o2ncpscr	NCP Main Script
asDcaCl	AntiSpam Delta Custom Action Client
sds_ppendix__86	Symantec Static Data Scanner Component Library
SvcDePlg	Service Dependency Plugin
Lue	Symantec LiveUpdate Engine
Comm	Communications Service
NCW	Norton Community Watch Component
BHClient	BASH Client
IronUser	Symantec Iron User Session
buMC	Backup MC
Avifc	Symantec AntiVirus Interface
AVExclu	Symantec AntiVirus Exclusion Manager
srtsp32	Symantec AutoProtect
isPwd	Password Manager
Iron	Symantec Iron Engine

Table 22: **Symantec.** Libraries (continued from previous page)

Library	Description
avScnTsk	Norton Security avScnTsk Module
ncpClient	NCP Client Service
cuIEPlg	Internet Explorer Cleanup Engine Plugin
naHelper	Norton Account Helper
QStartUI	QuickStart UI
RptCrdUI	Report Card UI
ccAlert	Symantec Alert and Notification
avScanUI	Norton Security Scan UI
ProdCbk	DING Product Callback DLL
SecureVPN	Secure VPN Proxy Feature
NCOLUE	NCO LUE Handler
spifc	Symantec Platform Component Library
sds_oader_64	Symantec Static Data Scanner Component Library
ccSEBind	Submission Engine Connection Library
sqscr	Norton Settings User Interface
sqsvc	Symantec Error Service Plugin
rcAlert	Symantec Alert and Notification Resources
InstUI	InstallToolBox Setup
ccSet	Symantec Settings Manager Engine
AppMgr64	Symantec Application Core Manager
Datastor	Data Store
AppState	Norton Metro App State
diFVal	InstallToolBox File Validation
cltLMJ	Symantec Shared Component
cltJSH	Consumer Licensing Technologies cltJSH
BHEng64	BASH Engine
wpSvc	Webcam Protection System Service
uiMain	Norton Protection Center NPC Status Plugin
AVMail	Symantec AntiVirus Email Filter
ccErrDsp	Symantec Error Display
jwWDF	Windows Defragmentation Job Worker
MCUI	Symantec Security History
isDataPr	IS Data Provider
coChrmSv	ChromiumPlugin
EFACli	Symantec Extended File Attributes
ccLib	Symantec Library
sds_ngine_86	Symantec Static Data Scanner Component Library
coFeatSv	NCO Feature Service
Settings	Norton Settings User Interface
buSvc	Backup Service
symv8	Symantec Support Library

Table 23: **TrendMicro.** Libraries

Library	Description
TMLCE64	Trend Micro Local Correlation Engine
TmopphSmtp	Trend Micro SMTP Handler
TmopsmHttp	Trend Micro Scan Manager for HTTP
ciussi64	ciussi Dynamic Link Library
Ssapi64	Anti-Spyware Engine
AIMURLRatingPlugin	Trend Micro TrendSecure
Tmopcfscan	Trend Micro String Scan Utility Module
helperTMEBCDriver	Trend Micro TMEBC Helper DLL

Table 23: **TrendMicro.** Libraries (continued from previous page)

Library	Description
smv64.old	smv64
TmopphPop3.old	Trend Micro POP3 Handler
Nitro	Trend Micro Nitro Engine
plugTmv	Trend Micro Vault PlugIn DLL
tmsa_ore64	TMSACore Dynamic Link Library
helperOspreyDriver	Trend Micro Anti-Malware Solution Platform
Tmopcfscan.old	Trend Micro String Scan Utility Module
utilTitaniumLuaHelper	Titanium LUA Helper
utilUniClient	Trend Micro Client Utility
plugSponge	PLUGSPONGE
Tmelapi	Trend Micro ELAM Communication Module
utilComponentInfo	Trend Micro Anti-Malware Solution Platform
coreFrameworkBuilder	Trend Micro Anti-Malware Solution Platform
plugEngineLCE	Local correlation Engine Plugin for AMSP
plugEventHub	Trend Micro Client Common Plug-in
ICRCHdler	ICRCHdler
tmeedbg.old	Trend Micro EagleEye Debug Log DLL
TmoppeUrlF	Trend Micro URL Filter Engine
TmoppeVS	Trend Micro Virus Scan Engine
TmSystemChecking	TmSystemChecking
tmncieco	Trend Micro NCIE Coordinator (amd64-fre)
plugFeedback	Plugin_eedback
TmoppeSsF	Trend Micro Safe Search Filter Engine
utilAccessControl	Trend Micro Anti-Malware Solution Platform
TMLCE64.old	Trend Micro Local Correlation Engine
inner_MSP_lientLibrary	Trend Micro Anti-Malware Solution Platform
coreTaskManager	Trend Micro Anti-Malware Solution Platform
tmopsent	Trend Micro Osprey Sentry
plugEngineDCE	Trend Micro Anti-Malware Solution Platform
plugSystemInfo	Plugin_ystemInfo
TmopCfg.old	Trend Micro Osprey Configuration DLL
DCEBootConfig.old	DCEBoot Config
TmopsmIm.old	Trend Micro Scan Manager for Instant Message
plugUtilLowConfDB	Trend Micro Anti-Malware Solution Platform
tmncieco.old	Trend Micro NCIE Coordinator (amd64-fre)
plugEngineVSAPI	Trend Micro Anti-Malware Solution Platform
TMAS_FAgent	Trend Micro Anti-Spam Dynamic Link Library
helperTMUMHDriver	Trend Micro UMH Driver Helper
plugEngineTmCDE	Trend Micro PlugEngineCDE
TmoppeHosF	Trend Micro Hosts Filter Engine
utilIPC	Trend Micro Anti-Malware Solution Platform
plugEngineFalcon	Trend Micro Anti-Malware Solution Platform
plugEngineTMSA	plugEngi Dynamic Link Library
helperEagleEyeDriver	Trend Micro Anti-Malware Solution Platform
TmopPlgAdp.old	Trend Micro Plugin Adapter Module
plugEngineAEGIS	Trend Micro Anti-Malware Solution Platform
ciuas64	ciuas Dynamic Link Library
TmOverlayIcon	Trend Micro Folder Shield Shell Extension
plugServiceBundle	Trend Micro Service Bundle PlugIn DLL
helperUCInstallation	Trend Micro Client Installation Library
TmopsmHttp.old	Trend Micro Scan Manager for HTTP
tmumhmgr.old	Trend Micro UMH Engine
plugSecureErase	Trend Micro Secure Erase PlugIn DLL
Tmopsent.old	Trend Micro Osprey Sentry
plugManualScan	Trend Micro Client Common Plug-in

Table 23: **TrendMicro.** Libraries (continued from previous page)

Library	Description
plugScan	PlugScan
TmUmEvt.old	Trend Micro User-Mode Hook Event Module
TmNetworkCost	Trend Micro Network Cost Dynamic Link Library
TMAS_LA.mui	Trend Micro Anti-Spam Agent for Outlook
libcef	Chromium Embedded Framework (CEF) Dynamic Link Library
TMPEM	Trend Micro Policy Enforcement Module
DRE	Damage Recovery Engine
tmsa_ore64.old	TMSACore Dynamic Link Library
PtSdk	PtSDK
plugEngineSSAPI	Trend Micro Anti-Malware Solution Platform
plugAdapterTMUMH	Trend Micro UMH Engine Adapter
TmopIEPlg32.old	Trend Micro Osprey IE Plug-In
plugLogHub	PlugLogHub
plugEngineTrxHandler	Trend Micro Anti-Malware Solution Platform
plugFeatureToggle	Trend Micro Client Common Plug-in
fcScan	fcScan
plugVizor	PlugVizor
coreActionManager	Trend Micro Anti-Malware Solution Platform
TmCDEngine	Trend Micro Collaberative Detection Engine
SEHelper	Trend Micro Secure Erase Helper DLL
plugTrendxScanFlow	Trend Micro Anti-Malware Solution Platform
tmmon64	Trend Micro UMH Monitor Engine
TmUmEvt64.old	Trend Micro User-Mode Hook Event Module (64-Bit)
utilUIProfile	Trend Micro Client Utility
ICRCHdler.old	ICRCHdler
DLLForVersionDisplay	DllForVersionDisplay
TmopsmIm	Trend Micro Scan Manager for Instant Message
TmopphPop3	Trend Micro POP3 Handler
helperSystemDriver	Trend Micro Anti-Malware Solution Platform
trxhandler	trxHandler
plugUtilException	Trend Micro Anti-Malware Solution Platform
CustomActUninst	Remove Application
TmopphSmtp.old	Trend Micro SMTP Handler
tmufeng.old	Trend Micro URL Filtering Engine
TmvHelper	Trend Micro Vault Helper DLL
TmToastNotification	Trend Micro Toast Notification Dynamic Link Library
TmvExt	Trend Micro Vault Extersion DLL
TmOsprey	Trend Micro Module
plugParentControl	Trend Micro Parent Control DLL
tmwk64.old	TMWK Dynamic Link Library
TmopphHttp.old	Trend Micro HTTP Protocol Handler Module
util3rdComponentInstall	Trend Micro Anti-Malware Solution Platform
TmNSCIns	Trend Micro NSC Driver Installation Module
utilInstallation	Trend Micro Anti-Malware Solution Platform
tmeectx.old	Trend Micro EagleEye Controller (X)
coreConfigRepository	Trend Micro Anti-Malware Solution Platform
npToolbarChrome	TrendMicro Toolbar Rating Plugin
TmopCtl.old	Trend Micro Osprey Control Module
plugWorkflowHost	Trend Micro Client Common Plug-in
TmopsmMail	Trend Micro Scan Manager for Mail
atse64	ATSE DLL for AMD64
TMAS_LA	Trend Micro Anti-Spam Agent for Outlook
TmopphMsn.old	Trend Micro MSN Protocol Handler Module
utilRPC	Trend Micro Anti-Malware Solution Platform
tmwlchk.old	Trend Micro White Listing Module

Table 23: **TrendMicro.** Libraries (continued from previous page)

Library	Description
plugBigFileScan	plugBigFileScan
tmsa64	TMSAEng Dynamic Link Library
Corridor	Corridor Dynamic Link Library
TmMsg.old	TMMSG with C interface
tmptfb	Trend Micro Platinum Feedback Module
plugCensus	Trend Micro Anti-Malware Solution Platform
TmoppeSAL	Trend Micro Script Analyzer
TmopphMsn	Trend Micro MSN Protocol Handler Module
Tmopsent	Trend Micro Osprey Sentry
TmopIEPlg.old	Trend Micro Osprey IE Plug-In
plugEngineWL	Trend Micro Anti-Malware Solution Platform
wccclient	wccclient
atse64.old	ATSE DLL for AMD64
fcTmJsFoundation	fcTmJsFoundation
plugSha1Cache	plugSha1Cache
FtpHandler	FtpHandler_D
plugUtilEnum	Trend Micro Anti-Malware Solution Platform
pbld64	RTPatch Executable
plugAppDelayLoad	Plugin_ppDelayLoad
TmopphHttp2.old	Trend Micro HTTP Protocol Handler Module
outer_MSP_lientLibrary	Trend Micro Anti-Malware Solution Platform
plugDTP	PlugDTP
tmufeng	Trend Micro URL Filtering Engine
plugManualScanFlow	Trend Micro Anti-Malware Solution Platform
TmMetroPkgMgr	Trend Micro Metro Package Manager Dynamic Link Library
ToolbarHelper	ToolbarH Dynamic Link Library
DCEBootConfig	DCEBoot Config
TmoppeEvts.old	Trend Micro Network Events Engine
plugWIFIAdv	WIFIAdvP Dynamic Link Library
plugDataShaper	Plugin_ataShaper
ssleay32	OpenSSL Shared Library
plugRealtimeScanFlow	Trend Micro Anti-Malware Solution Platform
plugRealTimeScanCache	Trend Micro Anti-Malware Solution Platform
TmopphYmsg	Trend Micro Yahoo Messenger Protocol Handler Module
tmfbeng	Trend Micro Feedback Engine
tmfbeng.old	Trend Micro Feedback Engine
plugAdapterSystem	Trend Micro Anti-Malware Solution Platform
TmoppeSAL.old	Trend Micro Script Analyzer
tscdll64	Trend Micro Damage Cleanup Engine (64-Bit)
TmOsprey32.old	Trend Micro Module
plugCommonScanCache	Trend Micro Anti-Malware Solution Platform
tmdshell	Trend Micro Client Shell Extension
plugAdapterEagleEye	Trend Micro Anti-Malware Solution Platform
utilThread	Trend Micro Anti-Malware Solution Platform
TmCDEngine.old	Trend Micro Collaberative Detection Engine
TmOsprey.old	Trend Micro Module
plugCloudBroker	plugCloudBroker
tmumhmgr	Trend Micro UMH Engine
AsSdk	Trend Micro Air Support
coreScanManager	Trend Micro Anti-Malware Solution Platform
utilServiceTag	utilServiceTag Dynamic Link Library
helperELAMDriver	Trend Micro Anti-Malware Solution Platform
libeay32	OpenSSL Shared Library
plugTaskManager	Plugin_askManager
plugFwOpt	PlugFWOpt

Table 23: **TrendMicro.** Libraries (continued from previous page)

Library	Description
plugPasswordProtection	PlugPasswordProtection
plugAdapterNCIE	Trend Micro Anti-Malware Solution Platform
luaWSC	Trend Micro Client Utility
TmoppeUrlF.old	Trend Micro URL Filter Engine
utilJsonHandle	Trend Micro Client Utility
TmSysEvt	Trend Micro Driver Communication Module (64-Bit)
plugEngineTMFBE	Trend Micro Anti-Malware Solution Platform
plugAdapterOsprey	Trend Micro Anti-Malware Solution Platform
TmOsprey32	Trend Micro Module
eextuins.old	Trend Micro EEXT Uninstaller
TmoppeHosF.old	Trend Micro Hosts Filter Engine
tmeectv	Trend Micro EagleEye Controller (V)
TmoppePDP	Trend Micro Privacy Data Protection Engine
plugLuaEngine	Plugin_uaEngine
TmConfig	TmConfig
TmVizorShortCut_8	VizorShortCut Dynamic Link Library for Win8
plugAdapterTMEBC	Trend Micro TMEBC Plug In DLL
ToolbarIE	Trend Micro TrendSecure
TmMetroTTM	TiThreatMap
TMPEM.old	Trend Micro Policy Enforcement Module
Redemption	Outlook Redemption COM library
tmsa64.old	TMSAEng Dynamic Link Library
tmeectv.old	Trend Micro EagleEye Controller (V)
tmtap	Trend Micro Firewall API Module
tmdbglog	TmDbgLog Dynamic Link Library
tmmon64.old	Trend Micro UMH Monitor Engine
TmSysEvt.old	Trend Micro Driver Communication Module (64-Bit)
tmaseng	Trend Micro Anti-Spam Engine
libeay32.old	OpenSSL Shared Library
TmopphYmsg.old	Trend Micro Yahoo Messenger Protocol Handler Module
DRE.old	Damage Recovery Engine
plugEngineDre	Damage Recovery Engine
trxhandler.old	trxHandler
TmvLib	Trend Micro Vault Lib DLL
TmMsg	TMMSG with C interface
helperNCIEDriver	Trend Micro Anti-Malware Solution Platform
plugAdapterELAM	Trend Micro Anti-Malware Solution Platform
paCoreProductAdaptor	Trend Micro Client Framework
utilNetCtrl	libNetCt DLL
plugEventLog	Trend Micro Client Common Plug-in
TmopPlgAdp	Trend Micro Plugin Adapter Module
plugEADAgent	Trend Micro EAD Agent(64-Bit)
tmmon.old	Trend Micro UMH Monitor Engine
tmxfalcon.old	Trend Micro Falcon Core Engine
tmwk64	TMWK Dynamic Link Library
plugCfgProxy	Trend Micro Client Common Plug-in
plugUpdater	Trend Micro Client Common Plug-in
plugUtilSysInfo	Trend Micro Anti-Malware Solution Platform
TmopIEPlg32	Trend Micro Osprey IE Plug-In
TmopphHttp	Trend Micro HTTP Protocol Handler Module
utilMsgBuffer	Trend Micro Anti-Malware Solution Platform
tmwlchk	Trend Micro White Listing Module
coreReportManager	Trend Micro Anti-Malware Solution Platform
plugToolbar	plugTool Dynamic Link Library
plugLocalCorrelationFlow	Trend Micro Anti-Malware Solution Platform

Table 23: **TrendMicro.** Libraries (continued from previous page)

Library	Description
fcTmJsTitanium	fcTmJsTitanium
plugTMAS	PlugTMAS
VizorUniclientLibrary	VizorUniclientLibrary
plugScheduler	Plugin_cheduler
7z	7z Plugin
TmoppePDP.old	Trend Micro Privacy Data Protection Engine
plugConfigManager	plugConfigManager
utilETW	Trend Micro Anti-Malware Solution Platform
tmtap.old	Trend Micro Firewall API Module
tmxfalcon	Trend Micro Falcon Core Engine
utilGenericLoader	Trend Micro Anti-Malware Solution Platform
TmopCtl	Trend Micro Osprey Control Module
plugDaemonHost	PlugHttpSrv
TmDbgLog	TmDbgLog Dynamic Link Library
plugPlatinum	PlugPlatinum
iaucore	Trend Micro ActiveUpdate Module
plugUtilRCM	Trend Micro Anti-Malware Solution Platform
TmopIEPlg	Trend Micro Osprey IE Plug-In
TmUmEvt64	Trend Micro User-Mode Hook Event Module (64-Bit)
TmopDbg.old	Trend Micro Osprey Debug Log DLL
tmmon	Trend Micro UMH Monitor Engine
coreEventManager	Trend Micro Anti-Malware Solution Platform
tmeesent	Trend Micro EagleEye Sentry
TmopsmMail.old	Trend Micro Scan Manager for Mail
TmoppeEvts	Trend Micro Network Events Engine
plugEngineSMV	Trend Micro Anti-Malware Solution Platform
plugSdkStub	Trend Micro Anti-Malware Solution Platform
coreCommandManager	Trend Micro Anti-Malware Solution Platform
TmopphHttp2	Trend Micro HTTP Protocol Handler Module
TMAS_LShare	Trend Micro Anti-Spam Sharor for Outlook
instInstallationLibrary	Trend Micro Anti-Malware Solution Platform
TmoppeVS.old	Trend Micro Virus Scan Engine
SEShellExt	Trend Micro Secure Erase Shell Extension DLL
ProToolbarIMRatingActiveX	Trend Micro TrendSecure
patchw64	RTPatch Executable
TmUmEvt	Trend Micro User-Mode Hook Event Module
iau	Trend Micro ActiveUpdate Module
Ssapi64.old	Anti-Spyware Engine
coreUpdateManager	Trend Micro Anti-Malware Solution Platform
utilDebugLog	Trend Micro Anti-Malware Solution Platform
TmopCfg	Trend Micro Osprey Configuration DLL
libcurl.old	libcurl Shared Library
TmopDbg	Trend Micro Osprey Debug Log DLL
plugLicense	PlugLicense
plugEngineTMUFE	Trend Micro Anti-Malware Solution Platform
QuietModeHelper	QuietModeHelper
smv64	smv64
ssleay32.old	OpenSSL Shared Library
tscdll64.old	Trend Micro Damage Cleanup Engine (64-Bit)
tmeedbg	Trend Micro EagleEye Debug Log DLL
libcurl	libcurl Shared Library
TmoppeSsF.old	Trend Micro Safe Search Filter Engine
Tmelapi.old	Trend Micro ELAM Communication Module

Table 24: **VIPRE.** Libraries

Library	Description
Vipre.Models.HistoryModels	History
kbu	kbu Dynamic Link Library
atcuf32	BitDefender Active Threat Control Usermode Filter
Vipre.ObjectModel.Events	Events
Vipre.Infrastructure.Plugins	Plug-In Helper
VIPRE.Common	VIPRE.Common
VIPRE.Consumer.Resources	VIPRE.Consumer.Resources
scan	BitDefender Threat Scanner
log4net	Apache log4net for .NET Framework 4.5
Vipre.Infrastructure.Product	Product
SbFwe	ThreatTrack Security Firewall SDK Firewall Engine Library
XceedZip	Xceed Zip for COM/ActiveX
Vipre.ObjectModel.Services	Controllers
Vipre.ObjectModel.DataModel	Data Model
Vipre.ViewModels	View Models
SerenityRose.Theme	SerenityRose Theme
atcuf64	BitDefender Active Threat Control Usermode Filter
SBArva	Email Antivirus
IncompatiblePrograms	IncompatiblePrograms
mimepp	DLL for Hunny MIME++ Library
Dark.Theme	Dark Theme
asunicode	Bitdefender Antispam Unicode Library
Vipre.Infrastructure.LoggingHelper	Logging Helper
Vipre.Models	Models
Light.Theme	Light Theme
System.Windows.Interactivity	System.Windows.Interactivity
Prism	Prism
spursdownload	Spurs Download Dynamic Link Library
Vipre.Tray.Notifications	Tray Notifications
ArcticWaters.Theme	ArcticWaters Theme
VSGNx64	VIPRE Search Guard for Internet Explorer (64-bit)
mimepack	MIME packer
Microsoft.Practices.Unity.Configuration	Microsoft.Practices.Unity.Configuration
Prism.Wpf	Prism.Wpf
Vipre.Diagnostics	Vipre.Diagnostics
Vipre.SocialWatch.Engine.Interfaces	SocialWatch Engine Interfaces
PI_ecovery	Recovery Monitor Plug-in
Microsoft.Practices.Unity.Interception.Configuration	Microsoft.Practices.Unity.InterceptionExtension Configuration
patchw32	RTPatch Executable
ascore	Bitdefender Antispam Core
Vipre.ObjectModel.Interfaces	Interfaces
SBAMSvcPS	SBAMSvcP Dynamic Link Library
VSGN	VIPRE Search Guard for Internet Explorer (32-bit)
unrar	RAR decompression library
SBAMOutlook	Outlook Antivirus Plugin
vipre	VIPRE Threat detection and remediation system
Vipre.Infrastructure.History	History
Vipre.SocialWatch.Scanner.Providers.Facebook.XmlSerializers	
bdsmartdb	BitDefender SmartDB
Vipre.Models.Interfaces	Models Interfaces
Vipre.Infrastructure.UserInterface	User Interface

Table 24: **VIPRE.** Libraries (continued from previous page)

Library	Description
ThemeManager	VIPRE
atccore	BitDefender Active Threat Control Communications Library
SBTIS	ThreatTrack Security Firewall SDK Transport Inspection System Library
Facebook	Facebook
Microsoft.WindowsAPICodePack.Shell	Microsoft.WindowsAPICodePack.Shell
SBRES_AS_n-US	VIPRE English Language Resources
gfiarkup	gfiarkup
Vipre.ViewModels.Infrastructure	View Models
Microsoft.Practices.ServiceLocation	Microsoft.Practices.ServiceLocation
Vipre.Infrastructure.Services	Services
Vipre.SocialWatch.Plugins.Facebook	Social Watch Facebook Plug In
VIPRE.Consumer.Schemas	VIPRE.Consumer.Schemas
SBCA	Custom Actions for the Installer
Vipre.Tray.Notifier	Notifier
Vipre.Infrastructure.Services.Interfaces	Services.Interfaces
Vipre.SocialWatch.Scanner.Interfaces	Social Network Scanner
Microsoft.WindowsAPICodePack	Microsoft.WindowsAPICodePack
Vipre.ObjectModel.ControllerEventAggregator	Controller Event Aggregator
Vipre.Tray.NotificationService	Notification Service
PI_atchMonitor	Patch Monitor Plug-in
SBAMScanShellExt	SBAM Scan Shell Extension
Vipre.SocialWatch.Scanner.Serialization	SocialWatch Serialization
AntiSpamThin	Bitdefender Anti-Spam SDK Cloud
Vipre.ViewModels.Interfaces	ViewModels.Interfaces
Vipre.SocialWatch.Configuration.Facebook	Social Watch Configuration Provider For Facebook
Vipre.Commands.Infrastructure	Controller Commands
asmcocr	BitDefender Antispam Image Processing Multi-character Optical character recognition Library
SbHips	ThreatTrack Security Firewall SDK Host Intrusion Prevention System Library
Vipre.ObjectModel.Interop.SBAMSvc	
gfiark	gfiark
remediation	VIPRE remediation library
bdcore	Bitdefender Core
Vipre.CommandHandlers.Infrastructure	CommandHandlers.Infrastructure
DotNetZip	Ionic's Zip Library
Vipre.SocialWatch.Authentication.Facebook.XmlSerializers	
sbap	Active Protection Library
Microsoft.Expression.Interactions	Microsoft.Expression.Interactions
Microsoft.Practices.Unity.Interception	Microsoft.Practices.Unity.InterceptionExtension
Vipre.Commands	Commands
asregex	Bitdefender Regular Expression Module
BDUpdateServiceCom	UpdateService
CartSdk	CART SDK
bdnc	Bitdefender Nimbus Client
Vipre.Views	Views
vcore	VIPRE Threat detection and remediation system
Vipre.SocialWatch.Scanner.Providers.Facebook	Facebook Provider
DarkHorse.Theme	DarkHorse Theme
SBTE	Threat Engine Library
Vipre.Commanding	Commanding
EndlessSierra.Theme	EndlessSierra Theme
Vipre.SocialWatch.Configuration.Interfaces	Social Watch Configuration

Table 24: **VIPRE.** Libraries (continued from previous page)

Library	Description
CityNights.Theme	CityNights Theme
SBFE	Secure File Eraser Shell Extension
Microsoft.Practices.Unity	
Vipre.Infrastructure	Infrastructure
Controls	Controls
Vipre.SocialWatch.Authentication.Interfaces	Authentication
SbWebFilter	ThreatTrack Security Firewall SDK WebFilter Library
gfiarksh	gfiarksh
Vipre.SocialWatch.Authentication.Facebook	Facebook Authentication Provider
updater	VIPRE Threat detection and remediation system

B Userland Hooks

Table 25: **Avast.** Userland Hooks.

Library	Function
ntdll	LdrLoadDll
	RtlQueryEnvironmentVariable
	ZwQueryInformationProcess
	NtMapViewOfSection
	ZwWriteVirtualMemory
	NtOpenEvent
	NtCreateEvent
	NtProtectVirtualMemory
	NtResumeThread
	ZwCreateMutant
	NtCreateSemaphore
	ZwCreateUserProcess
	ZwOpenMutant
	ZwOpenSemaphore
	RtlDecompressBuffer
USER32	SetWindowsHookExW
	SetWindowsHookExA

Table 26: **Bitdefender.** Userland Hooks.

Library	Function
ntdll	RtlAllocateHeap
	ZwSetInformationThread
	ZwClose
	NtOpenProcess
	NtMapViewOfSection
	NtTerminateProcess
	ZwWriteVirtualMemory
	NtDuplicateObject
	NtReadVirtualMemory
	ZwAdjustPrivilegesToken
	ZwQueueApcThread
	ZwCreateProcessEx
	ZwCreateThread
	ZwCreateProcess
	ZwCreateThreadEx
	ZwCreateUserProcess
	ZwRaiseHardError
	NtSetContextThread
	ZwWow64WriteVirtualMemory64
	RtlReportException
KERNEL32	Process32NextW
	CreateToolhelp32Snapshot
	MoveFileExA
	MoveFileWithProgressA
	DefineDosDeviceA
KERNELBASE	GetProcAddress
	CreateRemoteThreadEx
	LoadLibraryW
	OpenThread
	DeleteFileW
	LoadLibraryA
	CloseHandle
	CreateProcessW
	CreateProcessInternalW
	GetModuleInformation
	K32GetModuleFileNameExW
	EnumProcessModules
	GetFullPathNameW
	MoveFileExW
	SetEnvironmentVariableW
	GetApplicationRecoveryCallback
	GetApplicationRestartSettings
	K32EnumProcessMoExEx

Table 26: **Bitdefender.** Userland Hooks (continued from previous page)

Library	Function
	K32GetModuleBaseNameW
	PeekConsoleInputA
	PeekConsoleInputW
	ReadConsoleInputA
	ReadConsoleInputW
	GenerateConsoleCtrlEvent
	ReadConsoleA
	ReadConsoleW
	CreateRemoteThread
	CreateProcessA
	CreateProcessInternalA
	DefineDosDeviceW
	SetEnvironmentVariableA
SspiCli	DeleteSecurityPackageW+0x100
	EnumerateSecurityPackagesW
	EnumerateSecurityPackagesA
GDI32	BitBlt
	CreateCompatibleDC
	CreateCompatibleBitmap
	CreateBitmap
	Gdi32DllInitialize
	CreateDCA
	CreateDCW
ADVAPI32	PerfRegQueryValue+0x5490
	CryptGetHashParam
	CryptCreateHash
	CryptImportKey
	CryptHashData
	CryptExportKey
	CryptAcquireContextW
	CryptAcquireContextA
	CreateProcessAsUserW
	CryptGenKey
	EncryptFileW
	FlushEfsCache
	SetUserFileEncryptionKey
	CreateProcessAsUserA
	CreateServiceA
	CreateServiceW
	CryptDeriveKey
	LsaQueryTrustedDomainInfo
	LsaQueryTrustedDomainInfoByName
	CreateProcessWithTokenW
USER32	SendMessageW
	GetDesktopWindow
	SetWindowLongW
	UserClientDllInitialize
	PeekMessageW
	GetKeyState
	SystemParametersInfoW
	PostMessageW
	CallNextHookEx
	GetMessageW
	GetDC
	SetPropW
	SendNotifyMessageW
	SetWindowsHookExW
	UnhookWindowsHookEx
	PeekMessageA
	SendMessageA
	PostMessageA
	GetMessageA
	GetAsyncKeyState
	SystemParametersInfoA
	SetWindowLongA
	GetClipboardData
	SetClipboardData
	SetPropA
	SetWindowsHookExA
	FindWindowExW
	GetDCEx
	GetKeyboardState

Table 26: **Bitdefender.** Userland Hooks (continued from previous page)

Library	Function
	GetRawInputData GetWindowDC RegisterRawInputDevices FindWindowExA SendNotifyMessageA
shell32	Shell_otifyIconW RegenerateUserEnvironment+0x19A0
cryptsp	CryptExportKey CryptImportKey CryptHashData CryptCreateHash CryptGetHashParam CryptAcquireContextW CryptAcquireContextA CryptReleaseContext+0xC40
ole32	PropVariantCopy+0x390
combase	CoGetClassObject

C Kernel Monitoring

Table 27: **Vipre.** Userland Hooks.

Library	Function
ntdll	RtlAllocateHeap
	ZwClose
	NtOpenProcess
	NtMapViewOfSection
	NtTerminateProcess
	ZwWriteVirtualMemory
	NtDuplicateObject
	ZwAdjustPrivilegesToken
	ZwQueueApcThread
	ZwCreateProcessEx
	ZwCreateThread
	ZwCreateProcess
	ZwCreateThreadEx
	ZwCreateUserProcess
	ZwRaiseHardError
	NtSetContextThread
	RtlReportException
KERNEL32	Process32NextW
	CreateToolhelp32Snapshot
	MoveFileExA
	MoveFileWithProgressA
	DefineDosDeviceA
KERNELBASE	GetProcAddress
	CreateRemoteThreadEx
	LoadLibraryW
	OpenThread
	DeleteFileW
	LoadLibteSyst
	CloseHandle
	CreateProcessW
	InitializeContext2+0xFFFFFFFFFFAD740
	MoveFileWithProgressW
	MoveFileExW
	SetEnvironmentVariableW
	PeekConsoleInputA
	PeekConsoleInputW
	ReadConsoleInputA
	ReadConsoleInputW
	ReadConsoleA
	ReadConsoleW
	CreateRemoteThread
	CreateProcessA
	CreateProcessInternalA
	DefineDosDeviceW
	SetEnvironmentVariableA

Table 28: **FSecure.** Userland Hooks.

Library	Function
KERNEL32	OpenMutexA
KERNELBASE	CreateRemoteThreadEx
	CreateDirectoryW
	CreateMutexW
	OpenMutexW
	CreateMutexExW
	GetFileSize
	GetFileSizeEx
	WriteProcessMemory
	CopyFileExW
	CreateDirectoryExW
	TerminateThread
sechost	ControlService
	OpenServiceW
	CloseServiceHandle
	OpenServiceA
USER32	SetWindowsHookExW
	SetWindowsHookExA

D AV's Databases

```
[{19EA8BF0-A12F-1AF0-FB25-293AD7155932}]
Comment=*@1009
DefaultTask=1
Job=Scan
Label=*@1008
Priority=1
ScanAreas=
ScanFullFiles=1
ScanPackers=All
ScanPUP=1
ScanType=Content
ScanTypes=AllFiles
SpecialTask=0
TaskImage=chest
TaskSensitivity=100
UseCodeEmulation=1
```

Code 3: Avast Configuration File

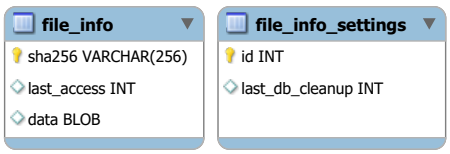

Figure 14: Avast File Database.

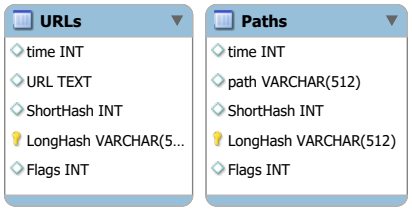

Figure 15: Avast URL Database.

Table 29: **Avast.** Kernel Drivers.

Driver	Description	Imports
aswArDisk.sys	Anti Rootkit Filter	IoAttachDeviceToDeviceStack
aswArPot.sys	Anti Rootkit	PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine KeStackAttachProcess ExRegisterCallback
aswbidsdriver.sys	IDS Activity Monitor	FltRegisterFilter PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine
aswbidsh.sys	IDS Helper	IoRegisterShutdownNotification PsSetCreateProcessNotifyRoutine
aswbuniv.sys	Universal Driver	PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutine EtwRegister
aswHdsKe.sys	Network Security	PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine
aswKbd.sys	Keyboard Filter	IoAttachDeviceToDeviceStackSafe
aswMonFlt.sys	Filesystem minifilter	FltRegisterFilter PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine PsSetLoadImageNotifyRoutine
aswRdr2.sys	WFP Redirect	PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine FwpmCalloutAdd0
aswRvrt.sys	Avast Revert	PsSetCreateProcessNotifyRoutine
aswSnx.sys	Virtualization	FltStartFiltering PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine IoRegisterPlugPlayNotification KeStackAttachProcess
aswSP.sys	Self Protection	IoAttachDevice PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine FltStartFiltering
aswStm.sys	Stream Filter	PsSetCreateProcessNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetLoadImageNotifyRoutine FwpsCalloutRegister1
aswVmm.sys	VMMonitor	PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutine

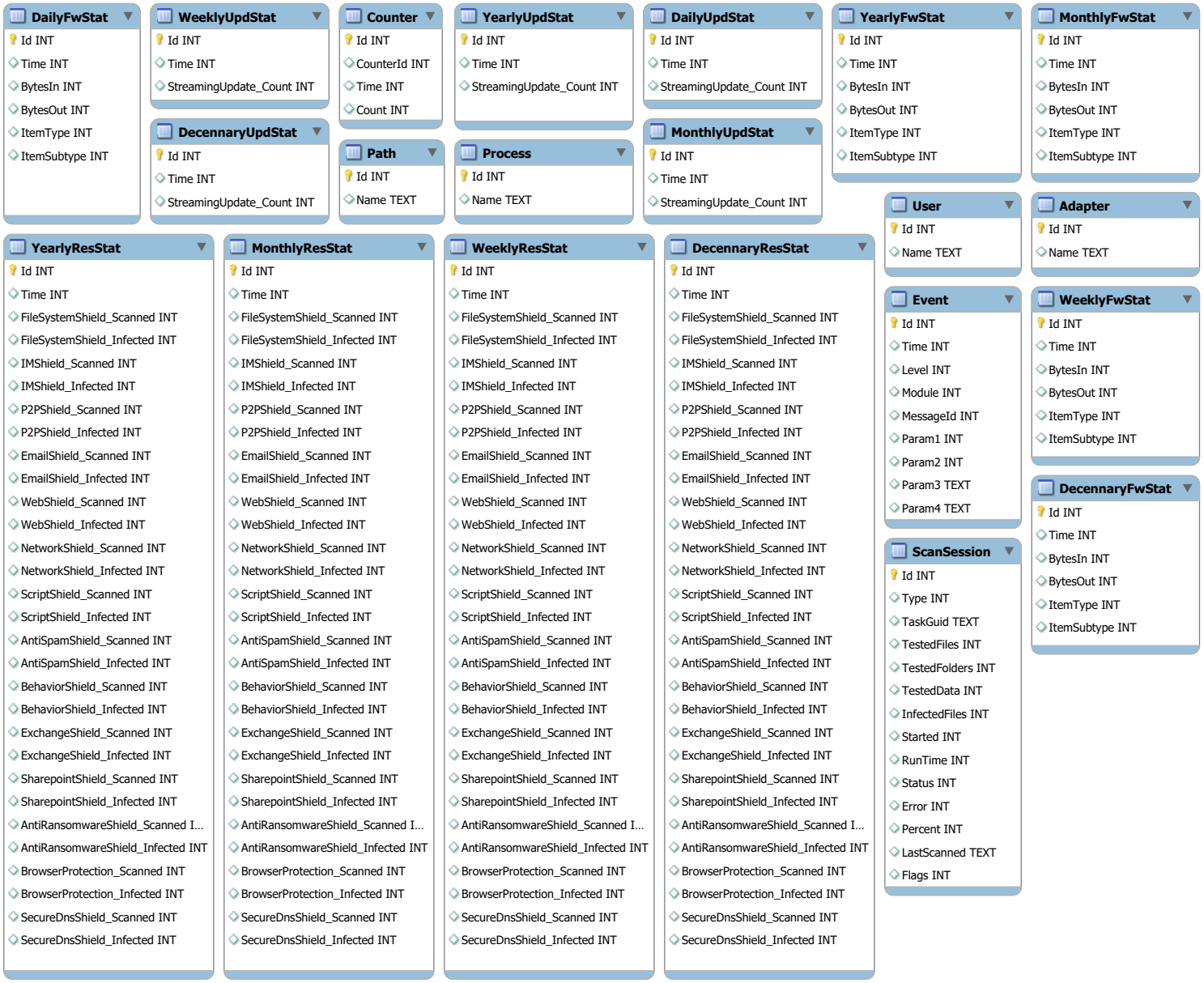

Figure 16: Avast Log Database.

Table 30: **BitDefender.** Kernel Drivers.

Driver	Description	Imports
atc.sys	Active Threat Control	FltRegisterFilter KeStackAttachProcess PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutineEx PsSetCreateThreadNotifyRoutine
bddci.sys	DCI filter driver	FwpmCalloutAdd0 PsSetCreateProcessNotifyRoutineEx
gemma.sys	Generic Exploit Mitigation	FltStartFiltering KeStackAttachProcess PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutineEx
gzflt.sys	Gonzales Filtesystem filter	PsSetCreateProcessNotifyRoutine FltStartFiltering
trufos.sys	Trufos Module	PsSetCreateProcessNotifyRoutine PsSetCreateThreadNotifyRoutine KeAttachProcess FltStartFiltering

Table 31: **FSecure.** Kernel Drivers.

Driver	Description	Imports
fsbts.sys	Boot Time Scanner	
fshs.sys	DG Module	PsSetCreateProcessNotifyRoutineEx PsSetLoadImageNotifyRoutine
fsni64.sys	Network Interceptor	FwpsCalloutRegister1
fsulgk.sys	GateKeeper	FltStartFiltering

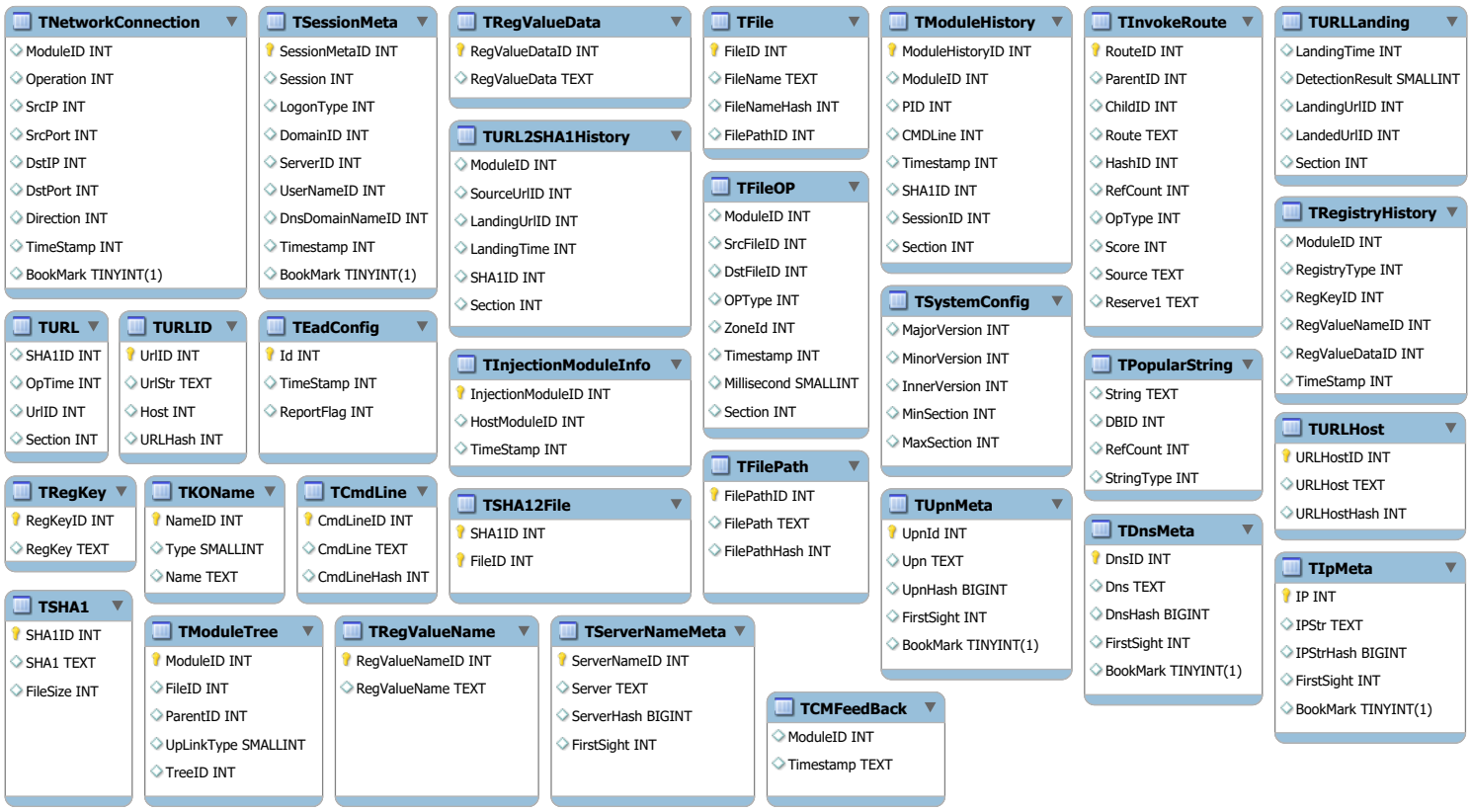

Figure 17: Trend Micro MBG database.

Table 32: **Kaspersky.** Kernel Drivers.

Driver	Description	Imports
klbackupdisk.sys	Backup Disk Filter	IoAttachDeviceToDeviceStackSafe
klbackupflt.sys	Backup File Filter	FltRegisterFilter
kldisk.sys	Virtual Disk	PsSetCreateProcessNotifyRoutine
klelam.sys	Early Launch Anti Malware	IoRegisterBootDriverCallback
klflt.sys	Filter Core	PsSetCreateProcessNotifyRoutine PsSetCreateThreadNotifyRoutine IoRegisterPlugPlayNotification IoRegisterBootDriverReinitialization
klhk.sys	???	PsSetLoadImageNotifyRoutine IoRegisterShutdownNotification KeStackAttachProcess KeAddSystemServiceTable
klim6.sys	Packet Filter	NdisRegisterDeviceEx
klkbdflt.sys	Keyboard Filter	IoAttachDeviceToDeviceStackSafe
klmouflt.sys	Mouse Filter	IoAttachDeviceToDeviceStackSafe
klpd.sys	Format Recognizer	
klpnpflt.sys	PnP Filter	
kltap.sys	OpenVPN Adapter	NdisRegisterDeviceEx
klupd_klif_arkmon.sys	Anti Rootkit Monitor	PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine
klupd_klif_kimul.sys	Kernel Heuristics Engine	
klupd_klif_klark	Anti Rootkit	IoRegisterPlugPlayNotification IoAttachDeviceToDeviceStack
klupd_klif_klbg.sys	Boot Guard Driver	PsSetCreateProcessNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetLoadImageNotifyRoutine
klupd_klif_mark.sys	Anti Rootkit Memory Driver	IoAttachDeviceToDeviceStack
klwfp.sys	Network Filter	FwpsCalloutRegister0
klwtp.sys	Network Connection Filter	FwpsCalloutRegister0
kneps.sys	Network Processor	

Table 33: **Malware Bytes.** Kernel Drivers.

Driver	Description	Imports
farflt.sys	Anti Ransomware	FltStartFiltering PsSetCreateThreadNotifyRoutine PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutineEx KeStackAttachProcess
mbae64.sys	Anti Exploit	PsSetCreateProcessNotifyRoutine PsSetLoadImageNotifyRoutine KeStackAttachProcess
mbamchameleon.sys	Chameleon	KeStackAttachProcess PsSetCreateProcessNotifyRoutineEx PsSetCreateThreadNotifyRoutine PsSetLoadImageNotifyRoutine
mbamelam.sys	Early Launch	
mbamswissarmy.sys	Swiss Army	PsSetCreateProcessNotifyRoutineEx KeStackAttachProcess
mbam.sys	Real Time Protection	KeStackAttachProcess PsSetCreateProcessNotifyRoutineEx PsSetLoadImageNotifyRoutine
mwac.sys	Web Protection	FwpmCalloutAdd0 PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutineEx

Table 34: **Norton.** Kernel Drivers.

Driver	Description	Imports
BHDrvx64.sys	BASH Driver	KeStackAttachProcess
ccSetx64.sys	Common Client Settings	
IDSvia64.sys	IDS Core	KeStackAttachProcess FwpmCalloutAdd0 NotifyUnicastIpAddressChange
IRONx64.sys	IRON Driver	
srtsp64.sys	AutoProtect	KeStackAttachProcess
srtspx64.sys	AutoProtect	
SymEFASI64.sys	Extended File Attributes	
SymELAM.sys	ELAM	
symnets.sys	Network Security	FwpmCalloutAdd0 NotifyUnicastIpAddressChange NotifyIpInterfaceChange
wpCtrlDrv.sys	Webcam Protection	IoAttachDeviceToDeviceStackSafe

Table 35: **Trend Micro.** Kernel Drivers.

Driver	Description	Imports
tmactmon.sys	Activity Monitor	KeStackAttachProcess
tmcomm.sys	Common Module	KeStackAttachProcess PsSetCreateProcessNotifyRoutine ZwNotifyChangeKey
tmebc64.sys	Early Boot Driver	PsSetCreateProcessNotifyRoutine
tmeevw.sys	Eagle Eye	KeStackAttachProcess FwpmCalloutAdd0
tmel.sys	ELAM	
tmevtmgr.sys	Event Management	PsSetCreateProcessNotifyRoutine
tmnciesc.sys	NCIE Scanner	PsSetCreateProcessNotifyRoutine
tm.sys	Transaction Manager	
tmumh.sys	UMH Driver	PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine PsSetLoadImageNotifyRoutine KeStackAttachProcess
tmusa.sys	Osprey Scanner	PsSetCreateProcessNotifyRoutine

Table 36: **VIPRE.** Kernel Drivers.

Driver	Description	Imports
atc.sys	Active Threat Control (BitDefender)	KeStackAttachProcess PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutineEx PsSetCreateThreadNotifyRoutine
sbapifs.sys	Active Protection (Threat Track)	PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutine KeStackAttachProcess
sbwfw.sys	VIPRE Firewall	KeStackAttachProcess FwpmCalloutAdd0 NotifyUnicastIpAddressChange NotifyIpInterfaceChange
sbwtis.sys	Threat Track Firewall	FwpmCalloutAdd0 PsSetCreateThreadNotifyRoutine KeStackAttachProcess
webexaminer64.sys	Threat Track WFP	FwpmCalloutAdd0

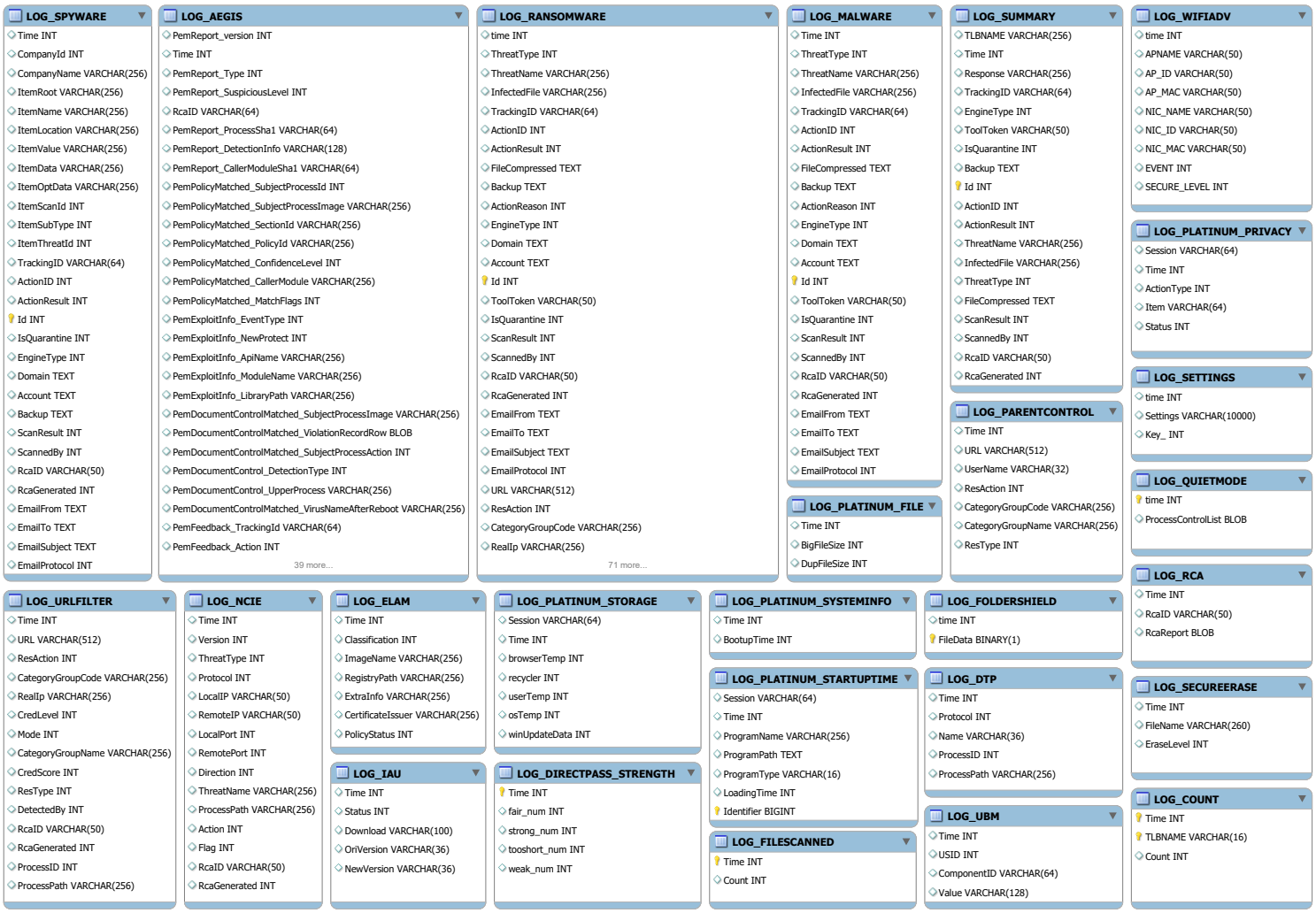

Figure 18: Trend Micro EventLog database.

```
alert tcp $NETWORK $PORT -> $NETWORK any (SBRuleId:XXX; msg:"[CVE-2018-12826]..."; dsize:<
    XXX; content:"HTTP/1.1␣200"; offset:XXX; depth:XXX; content:"Vector|0b|__AS3__.vec|06␣
    53␣...."; distance:XXX; within:XXX; classtype:trojan-activity; SBRiskLevel:2;
    SBCategory:"trojan-activity";)
```

Code 4: VIPRE's Snort Rules.