AVLeak:
Fingerprinting Antivirus Emulators Through Black-Box Testing

Jeremy Blackthorne Alexei Bulazel Andrew Fasano Patrick Biernat Bülent Yener
Rensselaer Polytechnic Institute

Abstract

To fight the ever-increasing proliferation of novel malware, antivirus (AV) vendors have turned to emulation-based automated dynamic malware analysis. Malware authors have responded by creating malware that attempts to evade detection by behaving benignly while running in an emulator. Malware may detect emulation by looking for emulator "fingerprints" such as unique environmental values, timing inconsistencies, or bugs in CPU emulation.

Due to their immense complexity and the expert knowledge required to effectively analyze them, reverse-engineering AV emulators to discover fingerprints is an extremely challenging task. As an alternative, researchers have demonstrated fingerprinting attacks using simple black-box testing, but these techniques are slow, inefficient, and generally awkward to use.

We propose a novel black-box technique to efficiently extract emulator fingerprints *without* reverse-engineering. To demonstrate our technique, we implemented an easy-to-use tool and API called *AVLeak*. We present an evaluation of AVLeak against several current consumer AVs and show emulator fingerprints derived from our experimentation. We also propose a classification of fingerprints as they apply to *consumer AV emulators*. Finally, we discuss the defensive implications of our work, and future directions of research in emulator evasion and exploitation.

1 Introduction

Recent estimates from Symantec claim that almost one million new pieces of malware are created every day [23]. Given this high rate of proliferation, antivirus software (AV) cannot simply rely on static signature-based malware detection. Automated dynamic analysis is necessary in order to identify packed known malware samples and to heuristically detect new malware. In response to the growth of automated dynamic analysis systems, malware authors have created evasive malware which resists automated analysis. A study from Lastline Labs showed that in the second half of 2015, over 80% of malware exhibited evasive behavior [37].

Our research specifically focuses on the *emulator*-based automated analysis systems used in consumer AV products. In order to evade AV emulators, malware authors have limited options. Many write malware that uses generic anti-analysis tricks (e.g., stalling loops, simple timing checks, obscure CPU instructions), though these behaviors can be detected and countered [15, 32, 38].

To more deftly evade emulation, malware authors may use distinguishing emulator "fingerprints" such as hard-coded environmental values, timing inconsistencies, or CPU "red pills." Malware that observes these fingerprints can recognize that it is being emulated and subsequently behave benignly to avoid detection. In order to discover fingerprints, malware authors may pursue challenging reverse-engineering or use black-box testing.

Our Contribution Our work is motivated by the difficulty of fingerprinting AV emulators through reverse-engineering. We propose a novel black-box technique that efficiently extracts fingerprints from emulators *without* requiring reverse-engineering. Our approach significantly advances upon prior black-box approaches, and our survey of fingerprints is more comprehensive than has been presented in prior literature. To demonstrate our attack, we built *AVLeak*, a tool and API for fingerprinting consumer AV emulators. We evaluated AVLeak against Kaspersky, Bitdefender engine (licensed for use in over twenty other AV products [4]), AVG, and VBA. During testing, we discovered hundreds of emulator fingerprints which we classify into six categories: environmental artifacts, OS API inconsistency, network emulation, timing, process introspection, and CPU "red pills."

In addition to the offensive implications of our research, it is also valuable in a defensive context. Emulator fingerprints can be used to discover advanced evasive malware in the wild and create new signatures.

Our work contributes to a growing body of research demonstrating vulnerabilities in AV software and raises awareness that these systems are not a panacea against malware, and may in fact expose users to more risk.

2 Background

Modern consumer AV software is highly complex and uses a number of techniques to identify malware. Our focus is the *emulators* embedded inside these systems, used to examine binaries which cannot be identified as malicious by simpler methods such as hashing, static signaturing, or static heuristic analysis. By running suspicious binaries in isolated virtual environments, emulators may look for known malware signatures in packed binaries or

droppers, or may heuristically classify runtime malicious behavior in novel malware.

We refer readers interested in a more comprehnsive overview of AV software to Koret and Bachaalany's "The Antivirus Hacker's Handbook" [35].

2.1 Consumer AV Emulators vs. Academic and Enterprise Analysis Systems

Consumer AV emulators are highly vulnerable to detection by malware, more so than high-end systems used in academic research and enterprise network protection. Despite their weaknesses, understanding these emulators (and AV software at large) is challenging.

Academic and enterprise malware analysis systems generally seek to record malware behavior in order to create signatures, or as an aid to human malware analysts. These systems return analysis reports that detail specific malware behaviors such as files and registry keys accessed, mutexes created, and network connections opened. Consumer AV emulators only seek to identify malicious binaries, using runtime signaturing or heuristic analysis.

The rich analysis reports provided by high-end systems may be exploited by attackers to exfiltrate fingerprints, e.g., creating a file named with the return value of `GetUserName` [72]. Consumer AVs are opaque in their operation and generally only return the names of detected malware.

Network-connected analysis systems are vulnerable to fingerprinting by malware which exfiltrates fingerprints over the network to attacker controlled servers [25, 36, 72]. In our experience, consumer AV emulators do not provide malware with network access.[1]

High-end analysis systems often run full installs of Windows in emulators (QEMU, Bochs), virtual machines (VMware, VirtualBox), or hypervisors (Xen, Intel VT). Consumer AVs are limited by copyright and performance considerations, preventing them from running a real Windows OS, or virtualizing a full hardware system. Instead, consumer AVs emulate a subset of the Windows API on top of often incomplete CPU emulation. When analyzing unknown binaries, consumer AV emulators must present a realistic execution environment with features such as concurrently executing processes, file systems, GUI and windows subsystems, a mouse, and the system clipboard. All of these facilities are provided by Windows itself for high-end systems.

Even when they are not publicly accessible, attackers may discover ways to evade high-end analysis systems by testing their readily-available underlying host platforms, i.e., QEMU, VMware, etc. Consumer AV emulators, on the other hand, are closed source and built on proprietary software.

Because of these considerations, consumer AV emulators remain incredible vulnerable to detection, but also difficult to analyze.

2.2 Reverse-Engineering AV Emulators

In our assessment, AVs are one of the most challenging types of software to reverse-engineer. AV software is highly complex, closely integrated with the operating system, and often resistant to analysis.

Reverse-engineers face many practical challenges in analyzing AV software. Anti-debugging protections prevent debuggers from attaching to AV processes. Libraries are often stored in custom non-PE packed file formats, hindering the use of standard static analysis tools. Even analyzing AV binaries in a disassembler can be difficult due to their enormous size.

AV emulators are more difficult to analyze than AV software at large.[2] Emulator analysis requires expert reverse-engineering skill given their immense complexity. Reverse-engineers also need deep knowledge of AV design; the x86 instruction set architecture, to analyze CPU emulation; Windows internals, to analyze Windows API emulation; and malware behavior, as the systems are purpose-built to run and analyze malware. Emulator binaries are particularly large with thousands of functions, and often include disassembler-breaking functions with thousands of basic blocks.[3]

As Koret and Bachaalany [35] point out, emulators are frequently updated, so attackers may have to re-analyze them with each new release in order to discover changes.

While we are not aware of prior art utilizing the technique, we note that looking for strings in running AV processes or core dumps may allow attackers to discover simple hardcoded environmental artifacts (e.g., user names, computer names, file system contents, etc.).[4]

Readers interested in learning more about reverse-engineering AVs are referred to "The Antivirus Hacker's Handbook" [35].

2.3 Black-Box Fingerprinting

Prior approaches to black-box fingerprinting have used slow, inefficient, and generally unrefined testing to discover simple, easily found emulator fingerprints.

At Black Hat 2014, Swinnen and Mesbahi [66] presented a novel packer and a scheme for discovering emulator fingerprints in consumer AV emulators leveraging a black-box construction. At B-Sides Las Vegas 2014, Adams [1] presented similar testing against AVG's JavaScript emulator. Sauder's DeepSec 2014 presentation [62] uses the same style of testing to create evasive Metasploit payloads. Nasi's self-published whitepaper [42] uses the black-box model to discover fingerprints in

AV emulators available on VirusTotal. Perhaps the most comprehensive work on offensive research against AVs, Koret and Bachaalany's "The Antivirus Hacker's Handbook" [35], demonstrates the same style of black-box attacks as described above. All of these efforts exploited in-emulator malware detection as a means of extracting the result of a single true or false query about the the emulator. In Figure 1 we show pseudocode demonstrating this "one-bit oracle" style of attack.

```
if EMULATOR_READING equals EXPECTED_VALUE:
   DropMalware()
else:
   Exit()
```

Figure 1: Pseudocode demonstrating one-bit oracle black-box testing as described by [1, 35, 42, 62, 66]. When emulated, this code will either drop malware, or exit without dropping malware. By checking if the emulator detected malware or not, an attacker can extract one bit of information about the internal state of the emulator, revealing if the value `EMULATOR_READING` is equal to `EXPECTED_VALUE`.

By checking if an AV emulator returns a malware detection or decides that the scanned binary is benign, an attacker may extract one bit of information about the emulator's internal state. One-bit testing is best suited for discovering negative results, e.g., OS API function f is not emulated correctly, file x is not present on the emulated file system, CPU instruction c does not work correctly. Specific state information can be slowly extracted through repeated testing (e.g., is byte b 0? Is it 1? Is it 2?...), as shown by Adams [1].

3 Our Approach: AVLeak

Our approach improves upon and generalizes prior testing schemes (as shown in Figure 1), by exploiting *specific* malware detections to leak fingerprints out of AV emulators. Whereas prior approaches answer a simple true or false query about emulator state with each scan, we rapidly extract arbitrary multibyte data. Our technique extracts emulator fingerprints *at least* an order of magnitude faster than previous black-box schemes. Further, our system is engineered for ease of test case construction, and offers a programmatic API to script testing routines.

Figure 2 shows our technique from steps 3 to 7.

Step 1: Filter Malware Samples

We begin by obtaining a large set of malware samples, and scan each with the AV under test. After scanning, the set is filtered, keeping only those samples identified as malware with unique signatures.[5]

Step 2: Map Malware Samples to Bytes

Given a filtered set of malware samples, we construct a mapping of malware signatures to bytes.

In our implementation, each malware signature is assigned to a single unique byte (e.g., `Blaster` to `0x00`, `Nimda` to `0x01`, and so on, up to `0xFF`). Multi-byte mappings could be used to enable more efficient attacks (e.g., individual malware samples to multi-byte values, `Blaster` to `0x0000`, `Nimda` to `0x0001`, etc.).

Step 3: Encrypt and Package Malware Samples

After constructing a mapping, each malware sample is encrypted to prevent static signature-based detection. The set of encrypted binaries is then packaged together into a single file.

Step 4: Write Dropper Code

Given a packaged malware file, we write code to decrypt and "drop" (write to disk) the encrypted malware samples when given their corresponding bytes of data (e.g., drop `Blaster` for `0x00`, `Nimda` for `0x01`).

Step 5: Build Test Case Logic

Next, we write code to find emulator fingerprints, for example, querying `GetUserName`, or analyzing register state after a particular call. Each byte of fingerprint data to be extracted is passed to our dropper code, as shown in Figure 3.

Step 6: Compile and Scan

After compilation, the resulting binary is given to the AV under test for scanning. Unable to statically identify the previously unseen binary, the AV runs the code within its emulator.[6]

During emulation, the binary decrypts and drops malware samples *within the emulator*. When emulation is complete, the AV returns the results of its scanning.

Step 7: Reconstruct Leaked Information

By correlating the malware signatures detected during emulation with the the bytes that they are mapped to, we can exfiltrate information from within the emulator. In Figure 4 we show example output from an antivirus program, and how the the leaked data is reconstructed.

3.1 Implementation

We tested four AV emulators: Kaspersky, Bitdefender (licensed for use in over twenty other AV products, [4] we tested via Emsisoft[7]), AVG, and VBA. The AVs were selected by uploading a dropper for EICAR[8] to

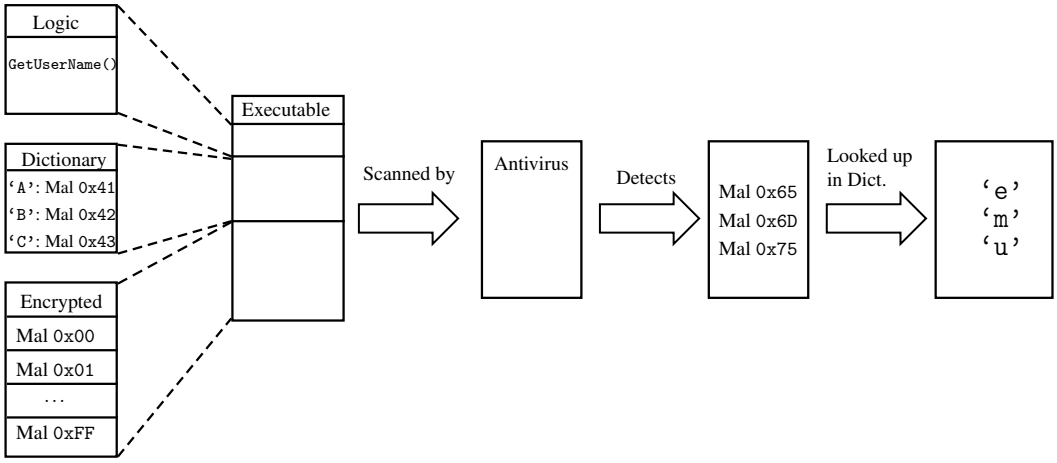

Figure 2: We begin by constructing an executable from a set of malware, a mapping of malware to bytes, and logic to fingerprint the emulator under test. This executable is then emulated by an AV, where it drops malware in response to fingerprint readings. The AV's malware detections are subsequently parsed and cross-referenced with the malware dictionary to leak data about the emulator. In this example, we extract the fingerprint that the user name within the emulator is "`emu`".

```
// EmulatedFunction is an OS API function
// emulated by the AV emulator, and returns
// the hardcoded value "Antivirus"

for each byte in EmulatedFunction():
    WriteToDisk(Decrypt(Malware[byte]))
```

Figure 3: Pseudocode sketch of the malware dropping process. This code will drop the malware samples corresponding to each byte of the value returned by `EmulatedFunction`. Assuming `EmulatedFunction` returned "`Antivirus`", the binary would drop malware `#0x41` ('`A`'), `#0x6e` ('`n`') ... `#0x73` ('`s`').

```
$ antivirus.exe --scan=EmulatedFunction.exe
  EmulatedFunction.exe detected as malicious:
  DROPPED: Poison Ivy    (sample 0x41 = 'A')
  DROPPED: MyDoom        (sample 0x6e = 'n')
  ... (samples for 't','i','v','i','r','u')
  DROPPED: Morris Worm   (sample 0x73 = 's')
```

Figure 4: Example output from an AV and reconstruction of exfiltrated information. Correlating the malware detections with the values they are mapped to, we reconstruct that `EmulatedFunction` returned "`Antivirus`".

VirusTotal and checking for EICAR identifications, indicating that the binary had been emulated. We filtered the set for AVs which returned *uniquely* named EICAR detections, as emulators are often licensed between AV vendors, and use the same detection strings.[9] We further refined the set of AVs based on their availability, general reputation, and ability to invoke scanning from the command line. We focused on extracting a wide variety of fingerprints over demonstrating our technique's obvious applicability to a wider range of AVs. In testing, we targeted 32-bit x86 Windows emulators, by far the most common type of AV emulator in our experience.

3.1.1 Testing Setup

The results presented in this paper were derived from testing Kaspersky Antivirus 15.0.2.480, Emsisoft Commandline Scanner 10.0.0.5366 (specific underlying Bitdefender engine version unclear), AVG 2015.0.6173, and VBA Windows/CL 3.12.26.4. Testing was conducted on a 32-bit Windows 7 SP1 VM running within VMware Fusion on a Mac OS X host system. The VM was allocated 4 GB of RAM and two 2.8 GHz processor cores.

After initial setup, we disconnected the VM from the internet to prevent the AVs from submitting our samples to their cloud servers for further analysis, and to stop them from downloading software or signature updates.

3.1.2 Implementation Specifics

Our first step in building AVLeak was to select sets of uniquely identified malware for each AV. Bitdefender and AVG were able to detect one dropped malware sample per emulator scan, so the AVs were each assigned sets of 256 malware samples to represent the range of bytes `0x00` to `0xFF`. Kaspersky could detect 30 *unique* dropped malware samples per scan, while VBA could detect 8, so they were assigned sets of 7680 (256 *x* 30) and 2048 (256 *x* 8) samples respectively. These sets allowed us to write code for the two AVs that could draw upon a full set of

```
#include "AVLeak.h"

int main(){
  char UserName[UNLEN + 1] = {0};
  DWORD len = UNLEN+1;
  GetUserName(UserName, &len);
  leak(UserName);
}
```

Figure 5: A simple AVLeak test case to extract the user name from an emulator. The `leak` function behaves much like `puts`, but causes malware to be dropped rather than printing characters to the terminal. Preprocessor directives abstracted away from the programmer in header files manage backend specifics for each AV.

256 *unique* malware samples for each byte of data to extract, bytes 0-29 in Kaspersky and 0-7 in VBA.

After selecting sets of malware, we encrypted the samples to prevent static detection, and compiled object files containing the encrypted samples and decryption code.

3.1.3 Software Engineering

AVLeak is implemented in Python, and test cases to fingerprint emulators from within are written in C.

Given the limited and varying detection capabilities of each AV, leaking multi-byte fingerprints out of their emulators often requires compiling and scanning multiple test binaries. For example, the first binary will leak fingerprint bytes 0 through 7, the second will leak bytes 8 through 15, and so on. To minimize the time spent writing C code in this inefficient paradigm, the specifics of malware dropping are isolated in header files and preprocessor definitions managed by AVLeak. Test case developers can use simple functions to leak data from within emulators, much like writing C to print to standard output. In Figure 5 we show a test case that extracts the username from within an emulator.

AVLeak's test case compilation, AV interaction, and data reconstruction are automated in Python. In addition to a command line tool, AVLeak also offers a Python API that may be used to script complex testing routines or integrate with other applications. In Figure 6, we show a Python function that uses the API to test various `HttpQueryInfo` flags.

AVLeak was designed for portability and ease of use. All Python scripts and C test cases may be written once and run against any AV. Most AV fingerprints can be extracted with only a few seconds of testing. We present several tables evaluating AVLeak's efficiency in the appendix.

We found that it was possible to integrate new AVs with only a few hours of work, the vast majority of which is spent running automated scripts.

```
from AVLeak import *

http_flags = ["HTTP_QUERY_ACCEPT",
              "HTTP_QUERY_ACCEPT_CHARSET",
              "HTTP_QUERY_ACCEPT_ENCODING",
              ... ]

def test_http(av):
  for flag in http_flags:
      result = av.leak(
        testfile = "HttpQueryInfo_flags.c",
        string = flag,
        printmax = 20)
    print flag + ": " + result
```

Figure 6: A simple AVLeak Python testing function which enumerates all HTTP attribute flags that can be queried with `HttpQueryInfo` and extracts the first 20 bytes of each returned attribute.

4 Results

In evaluation, we used AVLeak to find hundreds of emulator fingerprints spanning six categories, as summarized in Table 1: environmental artifacts, OS API inconsistency, network emulation, timing, process introspection, and CPU emulator "red pills."

We chose fingerprints to test by looking to traits of the emulated environments which we believed would be likely to be hardcoded or incorrectly emulated. Our intuition about these fingerprints was guided by study of AV software, evasive malware, and prior research on anti-analysis (see Section 8).

In this section we provide a brief overview of some of the most interesting findings from our research.

4.1 Environmental Artifacts

The simplest class of emulator fingerprint are environmental artifacts: traits of the execution environment itself. Consumer AVs do not run a real Windows installation, so their environments must be created from scratch, leaving them vulnerable to detection from hardcoded or inaccurate values. AV developers must also keep environmental values consistent when they are observed using different OS functions.

We found that Bitdefender, AVG, and VBA used hardcoded names for binaries under analysis, Bitdefender - "`C:\TESTAPP.EXE`"; AVG -

Class	Definition	Examples
Environmental Artifacts	Artifacts of the execution environment itself	Program / user / computer names, MAC addresses, file system, registry entries, running processes, GUI windows
OS API Inconsistency	Inconsistency in OS API implementation	Functions which return incorrect results, incorrectly fail or succeed
Network Emulation	Fingerprints related to network emulation	Hardcoded network responses, responses to invalid requests, protocol emulation inconsistencies
Timing	Timing inconsistencies related to emulation	Timing skews, inconsistencies, and failures
Process Introspection	Artifacts in the memory space of the program	Register states after API calls, runtime process structures, uninitialized memory, heap metadata, DLLs in memory
CPU "Red Pills"	Inconsistencies in CPU instruction emulation	Incorrectly emulated instructions, unique cycle counts, out-of-order execution behavior

Table 1: Table summarizing the six categories of fingerprints we examined with AVLeak.

"`C:\Documents and Settings\Administrator\My Documents\mwsmpl.exe`"; and VBA - "`C:\SELF.EXE`". Kaspersky randomized the name with each scan, returning "`C:\[5-8 random lowercase letters].exe`".[10]

All four AVs used hardcoded computer names: Kaspersky - "`NfZtFbPfH`"; Bitdefender - "`tz`"; AVG - "`ELICZ`"; and VBA "`MAIN`".

Querying system MAC addresses, we found that Kaspersky generated a random MAC per scan, Bitdefender used a hardcoded value, and AVG and VBA returned zeros.

We used AVLeak's API to recursively dump the names of all files on the emulated file systems, and all registry entries in the emulated registries. We discovered numerous uniquely named files,[11] missing system files,[12] fake installs of multiple AV products,[13] file sharing clients,[14] video games,[15] and common consumer software.[16] We believe that many of these files are present to bait unknown binaries into showing malicious behavior. Bitdefender and AVG did not emulate the presence of the "`.`" and "`..`" paths when iterating through directory contents using the `FindFirstFile` / `FindNextFile` functions.

Bitdefender's file system had several files which are clearly "easter eggs" from programmers: "`A_E_O_FANTOMA_DE_FISIER_CARE_VA_SA_ZICA_NU_EXISTA.BAT`"[17], "`TZEAPA_A_LA_BATMAN.EXE`"[18] and "`C:\BATMAN`".

Kaspersky's file system had 33 files in the "`My Documents`" directory with seemingly random file names and common file extensions (`xls`, `doc`, `mp3`, etc.). Close examination of the file names showed that they were likely created by a programmer typing random characters on a QWERTY keyboard.[19] We found PE headers and the string "`<KL Autogenerated>`" (presumably "Kaspersky Lab") in files on Kaspersky's file system (even files without executable extensions).

Taking inspiration from an attack documented by Lindorfer et al. [40], we checked the AVs' emulated registries for Windows product IDs, and found that only AVG's had one.[20]

AV emulators do not virtualize a full operating system, instead they only run a single (possibly multi-threaded) user mode process at a time. We used the `CreateToolhelp32Snapshot` function to extract process listings from each of the AVs, and found that all had hard-coded listings for other processes running on the system. Bitdefender's assigned the process under analysis to PID 8, running before essential Windows processes, and also had six processes sharing PID 12.[21] Kaspersky, Bitdefender, and AVG featured multiple AV processes in their process listings. Many of the fake processes were not backed by files on disk.

We also found environmental artifacts related to environment variables, open windows, hardware configuration, and system settings, among others.

4.2 OS API Inconsistency

Windows API emulation within AV emulators is often incomplete, and emulators can be fingerprinted by their inaccurate behavior. We found that AVLeak was not significantly more useful in finding these fingerprints than one-bit black-box approaches.

API inconsistencies within AV emulators are often manifested as total failure of certain functions rather than subtle inconsistencies in API operation, making them easy to discover with the one-bit testing shown in Figure 1. For more subtle artifacts we found that implementing test cases and analyzing their output for inconsistencies was time consuming and did not greatly benefit from the use of our technique. We also found that many interesting functions that we believed would be inaccurately emulated simply caused analysis to abort, a valuable result,

but not necessarily a fingerprint.[22]

The `FormatMessage` function returned interesting results, and benefited specifically from AVLeak's use. The function can be used to translate error messages from numeric codes to text based descriptions of the error. Kaspersky and VBA returned with failure error codes, though subsequent calls to `GetLastError` indicated that the function had succeeded. In Bitdefender, the function returned "`(from_other)`" for all error codes, and AVG returned a string with the error code in hexadecimal prefixed by "`MID`".

We found numerous subtle fingerprints related to incorrect WinAPI error codes returned by OS functions. We also explored OS API artifacts related to clipboard manipulation, permissions enforcement, input device state, interprocess communication, various file system actions, GUI interaction, and memory management.

4.3 Network Emulation

The AVs that we tested all denied network access to binaries under analysis, though Kaspersky, Bitdefender, and AVG emulated network connectivity. VBA returned failure when we tried to make network connections.

Testing HTTP connectivity was particularly fruitful, allowing us to discover networking related artifacts within Kaspersky, Bitdefender, and AVG. These three AVs returned HTTP success status codes when we attempted to make connections to any URL, including clearly invalid ones.

We extracted downloaded HTTP content from the three AVs, and found that all returned Windows executables in response to requests. Kaspersky returned a 32-bit Windows DLL filled with meaningless random code, and the string "`<Downloaded>`" followed by the URL the binary was "downloaded" from, in place of "`<KL Autogenerated>`" as found in binaries on the file system. Bitdefender returned a malformed MS-DOS (MZ) executable containing the unique string "`SetSuspect`". AVG returned a 32-bit Windows (GUI) PE executable which simply executes "`lock mov ebx, 0xff810598`" when run (perhaps related to AVG's function emulation, as detailed in section 4.5.1).

We also tested HTTP status flags, as shown in Figure 6, and found unique values and erroneously absent headers.

4.4 Timing

Garfinkel et al. [22] demonstrated the futility of creating timing-accurate virtualization systems. Timing attacks against virtualizated automated analyis systems have been demonstrated in prior research [15, 40, 54, 58]. We were unsurprised to find that consumer AVs struggled to accurately emulate timing.

We tested seven methods of reading time: `GetSystemTime`, `GetSystemTimeAsFileTime`, `GetTickCount`, `QueryPerformanceCounter`, `NtQuerySystemTime`, and the assembly instructions `rdtsc` and `rdtscp`. We found that there was no need for complex attacks on timing emulation as used in academic research.

Start times for analysis were hardcoded in all four emulators, e.g., Kaspersky's emulator always started the time at 11:01:19 (+/- a few ms, likely due to natural variation in the time for the function call to complete), July 13, 2012; while VBA returned 1,234,560,000 in response to calls to `QueryPerformanceCounter`, a unique value we assume was hardcoded by a programmer.

While Kaspersky and AVG's emulators attempted to accurately emulate timing, Bitdefender and VBA's were completely dysfunctional.

Taking timing readings with `GetTickCount` before and after a call to `Sleep(1000)` (sleep for 1,000 ms, or one second), Bitdefender showed average tick count differences of approximately 150,000,000 ms; 150,000 times larger than the expected 1,000. [23] VBA featured similar disproportionately large time deltas approximately 500 times larger than would be expected. Bitdefender aborted analysis after calls to `NtQuerySystemTime`, and did not modify the `SYSTEMTIME` structure given in to a call to `GetSystemTime`.

VBA identified the date and time as 1:31:12.123, 11/3/2013 in calls to `GetSystemTime` but returned 7:30:01.110, 7/17/2009 in calls to `GetSystemTimeAsFileTime`. VBA's implementation `FileTimeToSystemTime` always returned a system time of 0:0:0.0, 0/0/2000, no matter what `FILETIME` was passed in, and `NtQuerySystemTime` did nothing.

Kaspersky and AVG's higher fidelity timing emulation was detectable through "hyperreality", wherein the passage of time was emulated *too* accurately without consideration for the inherent overheard and variability of a real computer. Taking timing readings over executions of `Sleep`, we found that the emulators failed to account for the time it would take for `Sleep` to call into the kernel, and for the process to be put to sleep and subsequently woken. AVG monotonically incremented the time by the number of ms requested for sleep, showing 0 ms of overhead for the operation. Kaspersky showed some variability in timing, with roughly half of sub-15 second sleeps incurring a 15-16 ms overhead, while those above 15 seconds incurred 0 ms of overhead. For reference, our test VM incurred 14-16 ms of overhead for all `Sleep` calls.

While conducting networking testing, we timed the four emulators making HTTP requests to "`www.google.`

com", and found that all reported that requests took 0 ms. In our isolated test VM, each failed request took approximately 300 ms.

4.5 Process Introspection

Process introspection artifacts are fingerprints related to the observable state of code and data within a given binary's process space. Examples include heap metadata, periodicity of heap allocation, contents of uninitialized memory, Windows runtime data structures such as the PEB and TEB, data left on the stack or in registers between function calls, and DLLs loaded in memory.

We built AVLeak test cases to extract process introspection artifacts, but found that they are often very subtle and require deep knowledge of undocumented Windows internals. These fingerprints are not as obvious as others such as hardcoded environmental strings.

4.5.1 Library Code Artifacts

We analyzed the code of Windows system DLLs loaded in the process space of our test binaries, and found common patterns in three of the four emulators. We extracted code by dumping the raw bytes at addresses returned by calls to `GetProcAddress`, and found that obscure or excepting operations were used as a means of signaling the need for function emulation.

To trigger function emulation, AVG uses the obscure "`lock mov ebx`" instruction (shown in Figure 7), while Bitdefender calls or returns to an invalid address, and VBA moves the number `0x406` to address `0xFFF1[two byte ordinal number of invoked function]`. Kaspersky was unique in attempting to prevent detection by generating random bytes on per-run basis after the standard "`mov edi, edi`" and a "`push`" of the current function's address.

```
mov  edi, edi       ; WinAPI hot patch point
push ebp            ; function prologue
mov  ebp, esp       ; function prologue
nop
lock mov ebx, 0xff[1b lib #][2b func #]
pop  ebp            ; function epilogue
ret  [size of args] ; stack cleanup
nop...              ; nops between functions
```

Figure 7: Example of code extracted from AVG's `kernel32.dll` in memory. The second byte of the `mov` instruction argument denotes the library, while the third and fourth bytes denote a specific function. AVG's CPU emulator presumably intercepts the obscure "`lock mov ebx`", and invokes code to emulate the function.

In addition to the unique signatures within functions, we also found inaccurate use of the WinAPI hot patch point appearing on functions that do not have it in real Windows systems. Padding between functions was inaccurate in Bitdefender ("`int 3`") and VBA ("`hlt`").

4.6 CPU "Red Pills"

CPU "red pills" are instructions which behave differently on a CPU emulator than they do on a real CPU [53]. Efficient red pill discovery requires complex testing frameworks [53, 64]. We considered building such a framework to be beyond the scope of our current research. However, we were able to build tools for CPU state serialization and extraction with AVLeak, and preliminary experimentation appears promising. We hope to further document our red pill testing in future publications.

We were able to discover some red pills by creating custom testing scripts for particularly unique instructions unlikely to be correctly emulated.

Testing the `CPUID` instruction revealed that all four emulators under test identified themselves using Intel's vendor ID string "`GenuineIntel`", but implementation of other `CPUID` functions were inconsistent. When checking the processor brand string for the emulated CPUs, Kaspersky returned "`Intel(R) Pentium(R) 4 CPU 2.40GHz`"; while VBA identified as "`Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz`"; and Bitdefender did not return anything. AVG identified its processor as "`x86 Family 15 Model 4 Stepping 3, AuthenticAMD`", contradicting its vendor identification of "`GenuineIntel`". Further, AVG did not produce the "`IT'S HAMMER TIME`" string for `CPUID` function `0x8FFFFFFF`, a feature in AMD processors [19].

We found interesting preliminary results from testing the `RDTSC` instruction to retrieve CPU cycle counts. We believe that it may be possible to fingerprint specific CPU emulators by the number of (emulated) CPU cycles which specific instructions take. Further, `RDTSC`-derived cycle counts may be used to detect the lack of concurrently executing processes and an real operating system.

While conducting timing testing, we found that the `RDTSCP` instruction caused analysis to abort within Kaspersky and VBA's emulators.

4.7 Evasion

We briefly evaluated AVLeak's viability for operational use by creating twenty evasive malware droppers using AV fingerprints we discovered. We achieved 100% evasion, all of the binaries were not detected as malicious during emulation, but successfully dropped malware on a real system.

5 Malware Discovery

Through Google searches of strings found within the emulators, we discovered numerous malware samples profiled on automated malware analysis sites such as `totalhash.com` and `malwr.com`.[24] We were limited by our ability to only search for simple text based patterns in public malware reports.

AVLeak-derived fingerprints can be used defensively to create new static malware signatures for use in network protection systems, or to search through existing malware databases. Knowledge of specific emulation detection methods may be used to build mitigations against future detection, or to illicit previously unseen behavior in evasive malware.

5.1 Thai Malware

When searched on Google in March 2015, the AVG product ID[20] we discovered through registry dumping returned a single result, a file hosted on a Thai middle school's website.[25,26] The file has since been removed from the site. We extracted a user mode executable[27] and a kernel driver[28] from the file. Uploading the user-mode binary to VirusTotal showed that we were the third to upload the file, with the first upload on 11/12/2012. The kernel driver had never been uploaded before.

We found a third PE header in the file, but it did not correspond with any easily extractable executable. In the rest of the file, we found numerous environmental strings related to AVG's emulator as well as over 1,300 code snippets using AVG's function emulation triggering instructions, as discussed in section 4.5.1.

While their ultimate intentions remain unknown, the creators of the malware undoubtedly possessed intimate knowledge of AVG's emulator internals.[29]

5.2 EvilBunny

Our research also enabled better understanding of "EvilBunny", a highly advanced malware platform associated with the "Animal Farm" APT, first discovered by Marion Marschalek of Cyphort [41]. Before unpacking its multithreaded Lua scripting engine, EvilBunny's dropper[30] checks if its name contains "`TESTAPP`", the name used for binaries under analysis in Bitdefender's emulator. If named "`TESTAPP`", the malware aborts execution to avoid detection. EvilBunny also checks that its name is not "`afyjevmv.exe`" (among several other anti-analysis checks), which we believe may be a randomly generated name from Kaspersky's emulator.[31] EvilBunny's dropped payload[32] exhibits the same anti-analysis behavior, but checks for the string "`testapp.exe`". Prior to our research, it was not known that that the "`TESTAPP`" string was related to Bitdefender evasion.[33]

6 Future Work

Directions for future work include improvements to testing and AV integration, as well as the development of novel emulation detection attacks, and vulnerability research targeting emulators for breakout. We plan to construct more pre-built test cases and integrate more AV products with AVLeak.

Alternative Platforms AV emulators for ELF binaries, x86_64, ARM, .NET bytecode, JavaScript, and ActionScript have been documented [1, 35]. We hope to experiment with fingerprinting these emulators using AVLeak.

Autonomous Fingerprinting Our technique requires advance access to AV software in order to discover emulator fingerprints before creating evasive malware. Second Part To Hell's emulator detection technique [63] does not require prior access, and may be autonomously deployed in malware. Future researchers in this field may look into discovering other approaches to heuristically-enabled autonomous evasion.

Environmental Artifacts Future research in discovering environmental artifacts may look more deeply at file system and registry artifacts, emulated hardware devices, and other environmental traits. We are particularly interested in exploring fingerprints related to statistical discrepancies in file properties and metadata in emulated file systems versus real file systems.

OS API Inconsistency Improvements to OS API testing may benefit from the Wine project [70], which provides a compatibility layer allowing Windows binaries to run on POSIX systems. Unit tests for Wine's implementation of the Windows API could be repurposed for API testing with AVLeak.

Network Emulation Our exploration of network emulation primarily looked at HTTP traffic, but we believe valuable insights may come from exploration of other methods potentially used for exfiltration (FTP, raw sockets), command and control (IRC, email, DNS), DNS implementation, and HTTPS cryptographic negotiation. Preliminary results from manual reverse-engineering during initial experimentation showed us that some emulators may also support email protocols in order to analyze spambots.

Timing Swinnen and Mesbahi [66] intentionally introduced race conditions into multi-threaded code in order to detect incorrect timing emulation. We implemented tests for multi-threading-related fingerprints, but found that this style of testing did not benefit from the use of AVLeak over prior black-box schemes. Future research could develop more advanced threading-based detection attacks which benefit from AVLeak's use.

Process Introspection Further work in discovering process introspection fingerprints may look to detection attacks against dynamic binary instrumentation frameworks such as Intel Pin and DynamoRIO [65].

CPU Red Pills Advancements in CPU red pill discovery may draw upon prior work in the field, using open test suites such as Shi et al.'s [64] against AV CPU emulators.

6.1 New Reverse-Engineering Approaches

Future fingerprinting research could combine light reverse-engineering with black-box testing.

As emulators use functions to emulate the Windows API, an attacker could hook emulation functions and examine their arguments. For example, hooking an emulator's implementation of `WriteFile`, and saving off all data being written to the emulated file system within the AV. Fingerprinting binaries would then write observed fingerprints to a "file" with `WriteFile` to leak data.

In addition to tapping function calls within emulators for hooking, attackers could leak sensitive data though passive observation of an emulator process. If predictable or static memory locations are used as part of emulation, attackers could create emulator profiling binaries which place fingerprint observations in these locations for collection. Collection could be facilitated by an injected library, observing process, or external instrumentation. If memory locations are randomized, fingerprinting binaries could format observations with patterns so that they can be easily found in process memory.

6.2 Vulnerability Research

AVLeak may also be useful to researchers attacking emulators to discover vulnerabilities allowing breakout and in the case of many AVs, simultaneous privilege escalation.[34] AV exploitation has received recent attention from Joxean Koret [33, 34, 35], and Tavis Ormandy. Ormandy's Google Project Zero blog post "Analysis and Exploitation of an ESET Vulnerability" [46] demonstrates an ESET emulator breakout exploit. Various Project Zero bug reports from Ormandy have shown vulnerabilities in other emulators [47, 48, 49, 50, 51, 52]. While exploit development requires intensive reverse-engineering, AVLeak may provide vulnerability researchers with a good starting point for their analyses.

7 Mitigations

Our technique exploits AVs' essential ability to detect malware as a means of undermining them. Without fundamental changes to the structure of AV software, black-box attacks will continue to be effective against them.

We were surprised by the number of hardcoded environmental artifacts we discovered. Simply randomizing data where appropriate could make emulators more difficult to fingerprint. Kaspersky randomized program names, MAC addresses, and in-memory DLL code, but we did not observe any other randomization during experimentation.

Future developments in AV software may look to heuristically detecting anti-emulation behavior in malware. AV vendors can draw upon a wealth of academic research on the topic from the past decade [5, 26, 30, 31, 32, 39, 40]. Consumer AVs have an advantage over academic systems in that they only need to block malware from infecting endpoint computers, whereas academic systems seek to automate malware analysis. If an AV heuristically detects malicious or anti-emulation behavior, blocking the binary and sending it back to the AV's vendor for further analysis is a useful action.

Kolbitsch et al.'s work [32] uses five traits of system call invocation over time in order to detect stalling behavior in analysis-resistant binaries. Similar, but less computationally complex techniques may be useful for detecting anti-emulation in consumer AVs. The behaviors exhibited by AV emulation resistant malware are likely quantifiably different those seen in benign programs.

Program analysis techniques such as taint analysis, symbolic execution, and forced path exploration could eliminate the need for traditional emulation, or at least frustrate fingerprinting efforts. However, the use of these techniques in malware analysis remains an open area of research, and they are unlikely to be employed in consumer AV software anytime in the near future.

8 Related Work

Our research builds upon prior work from academic, industry, and independent security researchers.

Our literature survey focuses on attacks against Windows-based automated malware analysis systems and AVs, as well as publicly available materials on emulators from AV vendors themselves. We note that similar attacks have also been documented against alternative virtualization systems, including those for mobile devices and web browsers.

8.1 Academic Work

Prior academic research has used black-box analysis against commercial AV products, and has explored anti-analysis behavior seen in the wild. Academic research into virtualization detection and counter-detection mitigations is too vast to sufficiently document here, we refer interested readers to Egele et al.'s [18] survey of automated malware analysis system designs, Pék et al.'s [55]

survey of security issues in hardware virtualization, and Raffetseder et al.'s [58] demonstration of low-level detection attacks against system emulators.

Independently conducted work from Filiol et al. [20] and Hamlen et al. [24] demonstrates how black-box analysis may be used to extract static signatures from AVs by repeatedly scanning modified malware samples. Further work from Filiol et al. [21] uses a similar black-box scheme to extract information about run-time behavioral signatures. Borello et al. [11] use black-box analysis against sixteen AV products to evaluate their ability to detect metamorphic malware.

Yoshioka et al.'s work [72] is particularly relevant to our research, demonstrating the use of analysis reports as a means of exfiltrating fingerprints from within automated analysis systems.

Chen et al. [15] propose a taxonomy of anti-virtualization and anti-debugging techniques, and analyze the prevalence of anti-analysis behavior in malware. Bayer et al. [38], present statistics on malware behavior observed in the wild in the Anubis sandbox, and discuss detection attacks used against it.

Oberheide et al.'s PolyPack [44] provides a cloud service which packs malware binaries using multiple packers and evaluates their detection by consumer AVs.

In experimentation, we sought to use attacks on virtualization and emulation as documented in prior work, but found that consumer AVs were vulnerable to far simpler attacks. Our research is unique in attacking *consumer AV emulators*, we are not aware of prior academic treatment of the topic.

8.2 Industry and Non-Academic Work

Emulation detection and black-box testing of AV emulators has also been explored in whitepapers and conference talks from industry and non-academic researchers. We note that other non-academic research has also attacked high-end virtualization systems, and similar attacks have also been mounted against dynamic binary instrumentation frameworks [65].

As noted in Section 2.3, previous conference talks and written work have used one-bit black-box constructions to fingerprint AV emulators [1, 35, 42, 62, 66].

In presentations at REcon and Black Hat 2010, Georg Wicherski of Kaspersky Lab's Global Research and Analysis Team (GReAT) discussed emulator evasion techniques while presenting "dirtbox", an x86 Windows emulator for malware analysis motivated by shortcomings in AV emulators [68, 69].

In 2009, Kleissner mounted an attack on several AV companies, distributing a binary which contacted his servers to leak sensitive system fingerprints when executed in a network-connected analysis system [25, 36]. Kleissner's attack is the same as Yoshioka et al.'s [72], and was carried out shortly after the publication of the first technical report on the technique [71]. The fingerprints were subsequently made available on the now defunct `avtracker.info`.

In a blog post, Rolles [60] discusses an attack against Renovo, an academic automated analysis system, which exploits its ability to detect and return unpacked code in order to exfiltrate data about the host system.

Austrian virus author Second Part To Hell (SPTH) [63], proposes a method of using black-box analysis to detect AV emulators by examining undocumented register states left after the invocation of Windows API functions. SPTH's technique is particularly notable as it may be deployed autonomously by malware running in an unknown emulator, and does not require exfiltration of data from the emulator to enable evasion.

In a self-published paper, Ormandy [45] discusses the design of Sophos' emulator with insights presumably gleaned from manual reverse-engineering.

AV emulator evasion has received attention among penetration testers. SideStep [16] and peCloak [17] pack Metasploit payloads to evade AV emulators, while Veil [67] enables AV evasion at large.

Our work takes inspiration from prior black-box testing, but we use an expanded exfiltration bandwidth, and conduct a more through survey of consumer AV emulator fingerprints than previously presented.

8.3 Promotional Material and Patents

AV vendors generally do not discuss the internals of their software, though some have described their emulator technology in promotional material and patents.

Kaspersky Lab CEO Eugene Kaspersky has discussed Kaspersky's emulator in two blog posts [27, 28]. A case study from Bitdefender discusses high level features in the company's "B-HAVE" engine [9].

Kaspersky Lab has been particularly thorough in patenting technologies associated with their emulator [3, 6, 7, 8, 56, 57, 73]. A patent from Bitdefender [43] discusses a method for identifying known code sequences during emulation.

9 Conclusion

We have presented a novel technique for extracting artifacts of emulation from AV emulators. We constructed a tool, AVLeak, that successfully demonstrates our technique against several popular commercial AV products. Previously, discovering emulator fingerprints required difficult reverse-engineering or inefficient black-box testing, while our work makes the problem tractable to even novice programmers. AVLeak is efficient, generic, and

easy to use, making it a viable alternative to reverse-engineering. AVLeak can be integrated with new AVs in a matter of hours, and its API and abstracted design allows attackers to write fingerprinting code that works against any AV.

Our work contributes to a growing corpus of research attacking consumer AV software, exposing vulnerabilities in its operation previously protected by obscurity. Our findings show that advanced attackers are already aware of many consumer AV fingerprints, and have been using them to evade detection in the wild for years.

Emulation is a vital tool in stopping modern malware, and the battle between analysis-resistant malware and automated analysis systems remains highly active. In academic research, the most state-of-the-art automated analysis systems rely on instrumentation to physical hardware (avoiding emulation entirely) [29, 31, 61] or complex program analysis techniques and large scale test systems [2], both infeasible for deployment in consumer AV software.

We have just scratched the surface in fingerprinting AV emulators, and we hope to continue this research in order to enable more advanced attacks and greater AV coverage.

References

[1] ADAMS, K. Evading Code Emulation: Writing Ridiculously Obvious Malware That Bypasses AV, 2014. Talk at BSides Las Vegas 2014, Las Vegas, Nevada.

[2] ALWABEL, A., SHI, H., BARTLETTE, G., AND MIRKOVIC, J. Safe and Automated Live Malware Experimentation on Public Testbeds. In *7th Workshop on Cyber Security Experimentation and Test (CSET 14)* (2014).

[3] ANTUKH, A., AND MALANOV, A. System and method for generating emulation-based scenarios for error handling, Dec. 31 2013. US Patent 8,621,279.

[4] AV COMPARATIVES. List of AV Vendors (PC). `http://www.av-comparatives.org/av-vendors`, 2014.

[5] BALZAROTTI, D., COVA, M., KARLBERGER, C., AND KIRDA, E. Efficient Detection of Split Personalities in Malware. In *NDSS 2010, 17th Annual Network and Distributed System Security Symposium* (2010).

[6] BELOV, S. Method for accelerating hardware emulator used for malware detection and analysis, Feb. 21 2012. US Patent 8,122,509.

[7] BELOV, S. System and method for countering detection of emulation by malware, Dec. 9 2014. US Patent 8,910,286.

[8] BELOV, S. System and method for improving the efficiency of application emulation acceleration, Jan. 27 2015. US Patent 8,943,596.

[9] BITDEFENDER. B–HAVE –The Road To Success. Tech. rep., 2010.

[10] BLACKTHORNE, J., AND YENER, B. Reverse Engineering Anti-Virus Emulators through Black-box Analysis. Tech. rep., Computer Science Department, Rensselaer Polytechnic Institute, 2015.

[11] BORELLO, J. M., FILIOL, E., AND MÉ, L. From the design of a generic metamorphic engine to a black-box classification of antivirus detection techniques. *Journal in Computer Virology 6* (2010).

[12] BULAZEL, A. AVLeak: Fingerprinting Antivirus Emulators For Advanced Malware Evasion, 2016. Talk at Black Hat 2016, Las Vegas, NV.

[13] BULAZEL, A. AVLeak: Turning Antivirus Emulators Inside Out, 2016. Talk at ShmooCon 2016, Washington, DC.

[14] BULAZEL, A., AND YENER, B. AVLeak: Profiling Commercial Anti-Virus Emulators Through Black Box Testing. Master's thesis, Rensselaer Polytechnic Institute, Troy, NY, 2015.

[15] CHEN, X., ANDERSEN, J., MAO, Z. M., BAILEY, M., AND NAZARIO, J. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)* (2008), IEEE.

[16] CODEWATCH. SideStep: Another AV Evasion Tool. `https://www.codewatch.org/blog/?p=414`, 2015.

[17] CZUMAK, M. peCloak.py An Experiment in AV Evasion. `http://www.securitysift.com/pecloak-py-an-experiment-in-av-evasion`, 2015.

[18] EGELE, M., THEODOOR, S., KIRDA, E., AND KRUEGEL, C. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. *ACM Computing Surveys 44*, 2 (2012).

[19] FERRIE, P. Attacks on Virtual Machine Emulators. Tech. rep., Symantec Advanced Threat Research, 2006.

[20] FILIOL, E. Malware pattern scanning schemes secure against black-box analysis. *Journal in Computer Virology 2* (2006).

[21] FILIOL, E., JACOB, G., AND LIARD, M. L. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology 3* (2007).

[22] GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *HOTOS'07 Proceedings of the 11th USENIX workshop on Hot topics in operating systems* (2007).

[23] HALEY, K. 2015 Internet Security Threat Report: Attackers are bigger, bolder, and faster. `http://www.symantec.com/connect/blogs/2015-internet-security-threat-report-attackers-are-bigger-bolder-and-faster`, 2015.

[24] HAMLEN, K. W., MOHAN, V., MASUD, M. M., KHAN, L., AND THURAISINGHAM, B. Exploiting an Antivirus Interface. *Computer Standards & Interfaces 31* (2009).

[25] KAMLUK, V. A black hat loses control. `https://securelist.com/blog/incidents/30575/a-black-hat-loses-control`, 2009.

[26] KANG, M. G., YIN, H., HANNA, S., MCCAMANT, S., AND SONG, D. Emulating Emulation-Resistant Malware. In *VMSec '09 Proceedings of the 1st ACM workshop on Virtual machine security* (2009).

[27] KASPERSKY, E. Emulation: A headache to develop – but oh-so worth it. `http://eugene.kaspersky.com/2012/03/07/emulation-a-headache-to-develop-but-oh-so-worth-it`, 2012.

[28] KASPERSKY, E. Emulate to exterminate. `http://eugene.kaspersky.com/2013/07/02/emulate-to-exterminate`, 2013.

[29] KIRAT, D., GIOVANNI, V., AND KRUEGEL, C. BareBox: Efficient Malware Analysis on Bare-Metal. In *ACSAC '11 Proceedings of the 27th Annual Computer Security Applications Conference* (2011).

[30] KIRAT, D., AND VIGNA, G. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *CCS '15 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[31] KIRAT, D., VIGNA, G., AND KRUEGEL, C. BareCloud: Bare-metal Analysis-based Evasive Malware Detection. In *SEC'14 Proceedings of the 23rd USENIX conference on Security Symposium* (2014).

[32] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code. In *CCS '11 Proceedings of the 18th ACM conference on Computer and communications security* (2011).

[33] KORET, J. Breaking Antivirus Software, 2014. Talk at SYSCAN 2014, Singapore, Singapore.

[34] KORET, J. AV: Additional Vulnerabilities, 2016. Talk at Hack & Beers Bilbao 2016, Bilbao, Spain.

[35] KORET, J., AND BACHAALANY, E. *The Antivirus Hacker's Handbook*. Wiley, Indianapolis, Indiana, 2015.

[36] KREBS, B. Former Anti-virus Researcher Turns Tables On Industry. `http://voices.washingtonpost.com/securityfix/2009/10/former_anti-virus_researcher_t.html`, 2009.

[37] KRUEGEL, C. Three interesting changes in malware activity over the past year. `http://labs.lastline.com/three-interesting-changes-in-malware-activity-over-the-past-year`, 2016.

[38] KRUEGEL, C., KIRDA, E., COMPARETTI, P. M., BAYER, U., AND HLAUSCHEK, C. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009)* (2009).

[39] LAU, B., AND SVAJCER, V. Measuring virtual machine detection in malware using DSD tracer. *Journal in Computer Virology 6*, 3 (Aug. 2008).

[40] LINDORFER, M., KOLBITSCH, C., AND COMPARETTI, P. M. Detecting Environment-Sensitive Malware. In *RAID'11 Proceedings of the 14th international conference on Recent Advances in Intrusion Detection* (2011).

[41] MARSCHALEK, M. EvilBunny: Malware Instrumented By Lua. `http://www.cyphort.com/evilbunny-malware-instrumented-lua`, 2014.

[42] NASI, E. Bypass Antivirus Dynamic Analysis: Limitations of the AV model and how to exploit them. Tech. rep., Self-published, 2014.

[43] NOVITCHI, M. Anti-malware emulation systems and methods, Mar. 26 2013. US Patent 8,407,797.

[44] OBERHEIDE, J., BAILEY, M., AND JAHANIAN, F. PolyPack: An Automated Online Packing Service for Optimal Antivirus Evasion. In *WOOT'09 Proceedings of the 3rd USENIX conference on Offensive technologies* (2009).

[45] ORMANDY, T. Sophail: Applied attacks against Sophos Antivirus. Tech. rep., Self-published, 2012.

[46] ORMANDY, T. Analysis and Exploitation of an ESET Vulnerability. `http://googleprojectzero.blogspot.com/2015/06/analysis-and-exploitation-of-eset.html`, 2015.

[47] ORMANDY, T. Comodo Antivirus: Emulator Stack Buffer Overflow handling PSUBUSB (Packed Subtract Unsigned with Saturation). `https://bugs.chromium.org/p/project-zero/issues/detail?id=753`, 2016.

[48] ORMANDY, T. Comodo: Comodo Antivirus Forwards Emulated API calls to the Real API during scans. `https://bugs.chromium.org/p/project-zero/issues/detail?id=769`, 2016.

[49] ORMANDY, T. Comodo: Integer Overflow leading to Heap Overflow in Win32 emulation. `https://bugs.chromium.org/p/project-zero/issues/detail?id=738`, 2016.

[50] ORMANDY, T. ESET Emulation Vulnerability. `https://bugs.chromium.org/p/project-zero/issues/detail?id=456`, 2016.

[51] ORMANDY, T. ESET NOD32 emulator fails if you modify .idata after imports. `https://bugs.chromium.org/p/project-zero/issues/detail?id=470`, 2016.

[52] ORMANDY, T. ESET NOD32 Heap overflow unpacking EPOC installation files. `https://bugs.chromium.org/p/project-zero/issues/detail?id=466`, 2016.

[53] PALEARI, R., MARTIGNONI, L., ROGLIA, G. F., AND BRUSCHI, D. A fistful of red-pills : How to automatically generate procedures to detect CPU emulators. In *WOOT'09 Proceedings of the 3rd USENIX conference on Offensive technologies* (2009).

[54] PÉK, G., BENCSÁTH, B., AND BUTTYÁN, L. nEther : In-guest Detection of Out-of-the-guest Malware Analyzers. In *EUROSEC '11 Proceedings of the Fourth European Workshop on System Security* (2011).

[55] PÉK, G., BUTTYÁN, L., AND BENCSÁTH, B. A Survey of Security Issues in Hardware Virtualization. *ACM Computing Surveys (CSUR) 45*, 3 (2013).

[56] PINTIYSKY, V., AND BELOV, S. System and method for preserving and subsequently restoring emulator state, Aug. 18 2015. US Patent 9,111,096.

[57] PINTIYSKY, V., KIRSANOV, D., AND ANIKIN, D. System and method of transfer of control between memory locations, Aug. 25 2015. US Patent 9,116,621.

[58] RAFFETSEDER, T., KRUEGEL, C., AND KIRDA, E. Detecting System Emulators. In *ISC'07 Proceedings of the 10th International Conference on Information Security* (2007).

[59] ROLLES, R. Detecting an emulator using the windows api. `http://reverseengineering.stackexchange.com/questions/2805/detecting-an-emulator-using-the-windows-api`, 2013.

[60] ROLLES, R. Memory Lane: Hacking Renovo. `http://www.msreverseengineering.com/blog/2015/7/16/hacking-renovo`, 2015.

[61] ROYAL, P. Entrapment: Tricking Malware with Transparent, Scalable Malware Analysis. Talk at Black Hat 2012, Las Vegas, Nevada.

[62] SAUDER, D. Why Antivirus Software Fails, 2014. Talk at DeepSec 2014, Vienna, Austria.

[63] SECOND PART TO HELL. Dynamic Anti-Emulation using Blackbox Analysis. `http://vxheaven.org/lib/vsp42.html`, 2011.

[64] SHI, H., ALWABEL, A., AND MIRKOVIC, J. Cardinal Pill Testing of System Virtual Machines. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014).

[65] SUN, K., LI, X., AND OU, Y. Break Out of The Truman Show: Active Detection and Escape of Dynamic Binary Instrumentation, 2016. Talk at Black Hat Asia 2016, Singapore, Singapore.

[66] SWINNEN, A., AND MESBAHI, A. One Packer to Rule Them All: Empirical Identification, Comparison and Circumvention of Current Antivirus Detection Techniques, 2014. Talk at Black Hat 2014, Las Vegas, Nevada.

[67] VEIL FRAMEWORK. Veil-Evasion. `https://www.veil-framework.com/framework/veil-evasion/`, 2016.

[68] WICHERSKI, G. dirtbox, an x86/Windows Emulator, 2010. Talk at Black Hat 2010, Las Vegas, Nevada.

[69] WICHERSKI, G. dirtbox, an x86/Windows Emulator, 2010. Talk at REcon 2010, Montreal, Canada.

[70] WINE PROJECT. WineHQ. https://www.winehq.org/, 2016.

[71] YOSHIOKA, K., HOSOBUCHI, Y., ORII, T., AND MATSUMOTO, T. Vulnerability in Public Malware Sandbox Analysis Systems. In *Tenth Annual International Symposium on Applications and the Internet, SAINT 2010* (2010).

[72] YOSHIOKA, K., HOSOBUCHI, Y., ORII, T., AND MATSUMOTO, T. Your Sandbox is Blinded: Impact of Decoy Injection to Public Malware Analysis Systems. *Journal of Information Processing 19* (2011).

[73] ZAITSEV, O. System and method for detection of malware using behavior model scripts of security rating rules, Mar. 10 2015. US Patent 8,978,142.

Notes

[1]In a particularly odd bug report, Google Project Zero's Tavis Ormandy demonstrates how Comodo's emulator unwittingly provides malware with network connectivity, and constructs a keylogger for the *host* system that runs from within the emulator while exfiltrating keystrokes to an attacker controlled server [48].

[2]Even locating specific modules within AV software dedicated to emulation can be challenging. Koret and Bachaalany discuss challenges related to locating emulator libraries and propose some simple heuristic methods for finding them in "The Antivirus Hacker's Handbook" [35], pages 304-306.

[3]Following advice from Rolf Rolles in response to a Stack Exchange post [59] we made early in our development of AVLeak, we analyzed several AV emulators by searching for large switch cases in IDA Pro. After increasing IDA's per-function basic block limit to 10,000, we attempted to look at these large functions in IDA's graph view, which froze the window during rendering, was slow to navigate, and crashed the program when non-trivial annotations were made.

[4]We used this technique to find fingerprints during initial experimentation and subsequently confirmed our findings with AVLeak.

[5]Specifically, we used a large set of DOS viruses (the VX Heaven Virus Collection, a 45 GB collection available on numerous torrents online), selected for their small size. Checking for unique identification is important, as many AVs use broad signatures that cover multiple variants of individual malware samples (i.e., identifying two distinct binaries known to us as "`Trojan.DOS.KillCMOS.7`" and "`Trojan.DOS.KillCMOS.A`" both as "`Trojan.DOS.KillCMOS`").

[6]Making sure that code is not heuristically identified as malicious and is emulated with every single run is non-trivial, and required us to specially craft our test cases and experiment with various scanning parameters of the AVs.

[7]We began testing Emsisoft and F-Secure, which use Bitdefender's emulator, but found the same fingerprints in both AVs, and subsequently moved to just testing Emsisoft, as its command line scanner is considerably faster than F-Secure's.

[8]A file developed as a standard test case for AVs by the European Institute for Computer Antivirus Research (EICAR). We used this file due to its nearly universal detection by AV products.

[9]e.g., "`EICAR.Test.File`" vs "`EICAR-Test-File (not a virus)`" vs "`EICAR.TEST.NOT-A-VIRUS`", etc.

[10]e.g., `C:\[lstcvix, tudib, izmdmv, ubgncn, jidgdsp, evabgzib, qzqjafyt, cnyporqb, gfydwrkt].exe`.

[11]The Bitdefender file system included "`COMMAND.COM`", "`NOTHING.BAT`", "`NOTHING.COM`", "`FILE001.EXE`", 100 files named "`EMPTY[2 digit number 00-99].INI`", and several files called "`TRAP`" with various extensions.

VBA had "`STD_OUTxe`", "`Dummy.exebat`" and "`welcome.exe`".

Kaspersky's file system erroneously contained an empty "`Arquivos de programas`" (Spanish for "Program Files") directory.

[12]AVG's file system was particularly sparse, the `C:\WINDOWS\system32` directory contained just three files: "`victim.exe`" (a file system fingerprint for AVG itself), "`ntdll.exe`", "`kernel32.dll`", and several directories "`Drivers\etc`", "`dllcache`", and "`wbem`".

VBA's `system32` only contained "`calc.exe`", "`KERNEL32.DLL`", "`WSOCK32.DLL`", and a "`Drivers\etc`" directory which in turn contained a "`hosts`" file.

[13]Kaspersky's file system, the largest and most comprehensive of the file systems we examined, contained directories and fake executables for twenty AV products in the `C:\Program Files` directory (Agnitum, AntiVir PersonalEdition Classic, eMule, Eset, F-Secure Internet Security, Kaspersky Lab, KAV6, McAfee, mcafee.com, Messenger, Network Associates, Norton AntiVirus, Norton Internet Security, QIP, Rising, Sygate, Symantec, Symantec AntiVirus, Tencent, Trillian).

Bitdefender similarly featured directories for Anti Virus, Bitdefender (and versions 8, 9, and 10), Complus Applications, F-PROT95, Grisoft, Inoculate, Kaspersky Lab, McAfee, Network Associates, Norton Antivirus, Panda Software, Softwin, Symantec, TBAV, Trend Micro, and Zone Labs.

[14]Kaspersky had directories for CuteFTP, eDonkey2000, and Kazaa. Bitdefender also had a Kazaa directory.

[15]Kaspersky's registry contained keys related to World of Warcraft. Bitdefender included files for Windows default install games such as Pinball.

[16]VBA's file system contained "`C:\Program Files\Far\Far.exe`", the name of Far Manager's executable (a file manager popular in Eastern European countries, such as Belarus where VBA is based). Kaspersky (based in Russia) had references to Far Manager in its emulated registry.

[17]This roughly translates from Romanian (Bitdefender is based in Romania) to "`this is a ghost file which will tell you [that] it doesn't exist.bat`". The `C:\WINDOWS` directory where we found the file also contained "`A_E_O_FANTOMA_DE_FISIER_CARE_VA_SA_ZICA_NU_EXISTA.EXE`", "`Z_E_O_FANTOMA_DE_FISIER_CARE_VA_SA_ZICA_NU_EXISTA.BAT`', and "`Z_E_O_FANTOMA_DE_FISIER_CARE_VA_SA_ZICA_NU_EXISTA.EXE`".

[18]Google Translate was not able to produce a clear translation, but removing "TZ", the computer name for Bitdefender, returned "screed of the Batman". Files with this name appeared in many places throughout the Bitdefender file system, including in directories for various AV products in `C:\PROGRAM FILES` (e.g., `C:\PROGRAM FILES\NORTON ANTIVIRUS\TZEAPA_A_LA_BATMAN.EXE`) .

[19]The file names featured keys that are close together on a QWERTY keyboard, such as "`koio.mpg`" (k, i, and o are all all directly adjacent), "`muuo.mp3`" (three characters on the right hand side of the keyboard), "`wcwe.jpg`" (three characters on the left hand side of the keyboard).

[20]`76588-371-4839594-51979`

[21]`explorer.exe`, `iexplore.exe`, `winlogon.exe`, `lsass.exe`, `smss.exe`, `msnmsgr.exe` all had PID 12.

[22]In evaluation we reimplemented a number of fingerprinting tests demonstrated by prior researchers, and found that many actually caused emulators to prematurely abort analysis, rather than being actual fingerprints. To researchers simply looking to see if malware was or was not detected during emulation, this behavior could make it appear that they had discovered a fingerprint.

[23]The tick count was evidently stored in an unsigned 32-bit integer, as it would overflow over the course of multiple timing tests. We observed it roll over from $3,305,476,124$ to $510,508,871$ after a call to `Sleep(10000)`, indicating that it was incremented by $1,500,000,043 \approx 150,000 \times 10,000$.

[24]Readers are encouraged to try searching for the unique strings mentioned in this paper to verify these findings.

[25]`http://kpp.nfe.go.th/`

[26]MD5: `7a1a62b7fd6a631ebe7bcbbf704b754a`

[27]MD5: `6eb177dedc858b55daadc9a3b1bb4d07`

[28]MD5: `6698f547e3a3dc0cfd21ef6f757e6c73`

[29]Analysts with any insight into these binaries, or an interest in analyzing them are encouraged to reach out to the authors. We conducted a cursory analysis of the files, but their overall purpose is unclear.

The usermode executable had five functions. The program accesses the `GdiHandleBuffer` entry in the `PEB` and enters a switch case based on the last element of the buffer. In switch cases four and five respectively, the program then loads a DLL with a name specified in the command line arguments of the program and calls a command line-specified function in it. In cases three, six, eleven, twelve, and thirteen, the program exits with `ExitProcess(1)`. The remaining default cases loops calling a function while the value in register `eax` is non-zero, and breaks to the `ExitProcess` call when the value is zero. The function called in the loop simply executes the instruction "`lock mov ebx, 0xff810598`" and returns. We were not able to identify the function within AVG which this operation corresponds to, but it follows AVG's form of using "`lock mov ebx`" to trigger function emulation as shown in Figure 7.

The kernel driver was built with debug information in a file called "`UNISYS.pdb`", we were not able to find any information about the name from Google searches. The driver disassembled to a single empty Windows driver entry.

[30]MD5: `c40e3ee23cf95d992b7cd0b7c01b8599`

[31]It is possible that this name was extracted from an earlier version of Kaspersky that does not do per-run name randomization, or that the programmers simply extracted the name from a single run.

[32]MD5: `3bbb59afdf9bda4ffdc644d9d51c53e7`

[33]We shared this information with Marion Marschalek when we discovered it, she has since presented it in talks at SYSCAN, REcon, and VirusBulletin, among others.

[34]Ormandy's Google Project Zero blog post "Analysis and Exploitation of an ESET Vulnerability" [46] shows how a vulnerability in emulator code shared across ESET software for Windows, Mac, and Linux can be exploited to achieve `NT AUTHORITY\SYSTEM` privilege on Windows and root privilege on Mac and Linux.

Appendix

AV	Binaries	Time	Bytes per Sec.
Kaspersky	4	14.86	8.07
Bitdefender	120	157.80	0.76
AVG	120	137.52	0.87
VBA	15	46.73	2.57

Table 2: Table showing AVLeak's efficiency in leaking a 120 byte static string from the emulators (average time in seconds over five trials). Extracting emulator fingerprints often only takes a few seconds, and rarely exceeds a minute. Note that Bitdefender's speed may be slowed by pre-emulation processing in the Emsisoft scanner that invokes it.

AV	Binaries	Bytes	Time
Kaspersky	1	14 (avg)	2.29
Bitdefender	14	14	19.10
AVG	63	63	83.95
VBA	2	11	6.11

Table 3: Table showing AVLeak's efficiency in leaking a `argv[0]` (program name, as described in Section 4.1) from the emulators (average time in seconds over five trials).

AV	Files	Binaries	Bytes	Time
Kaspersky	590	2317	6381	1:06:58
Bitdefender	518	6580	5491	2:23:20
AVG	52	687	518	14:57
VBA	23	104	207	5:14

Table 4: Table showing AVLeak's efficiency in leaking the names of every file on the emulated file system from each of the emulators (time in hours, minutes, and seconds). Our calculation of number of files is the number of directories and files on the file system `C:\` drive, not counting the "`.`" and "`..`" paths which come up when iterating through files in directories (these paths were extracted with AVLeak, and are included in the calculation of number of bytes leaked). Our process for dumping file system entries involves recursively exploring all directories starting at the `C:\` drive. The process is somewhat inefficient as in we did not have an easy way to convey how many files are in a given directory, or how long each file name is, so we recursed through the directories until no more files were present, and dumped file names until a null byte was encountered. We believe that we could expedite this process by a factor of at least two with some optimizations to our design. The process is fully automated by a ~35 line Python script using the AVLeak API, and ~30 lines of C code for the binaries that are scanned to leak information.

Acknowledgements

The authors would like to thank the following individuals for their support: Marion Marschalek, Rolf Rolles, Alex Ionescu, Dr. Sergey Bratus, Bruce Dang, Dr. Gregory Hughes, and RPISEC.

Notes on Prior Presentation

Blackthorne and Yener created a preliminary version of AVLeak under the name *AV Oracle*. Their findings were published in a technical report from Rensselaer Polytechnic Institute Computer Science Department [10]. Bulazel extended this work in his Master's thesis at Rensselaer Polytechnic Institute [14] and presented on AVLeak at ShmooCon 2016 [13] and Black Hat 2016 [12].