Reverse Engineering Anti-Virus Emulators through Black-box Analysis

Jeremy Blackthorne
Computer Science Department,
Rensselaer Polytechnic Institute
Troy, NY 12180-3590
Email: whitej12@rpi.edu

Bülent Yener
Computer Science Department,
Rensselaer Polytechnic Institute
Troy, NY 12180-3590
Email: yener@cs.rpi.edu

***Abstract*—Anti-virus (AV) programs have traditionally used signature matching in order to detect malware. Malware authors try to evade signature matching by encrypting and compressing malware, also known as packing. Packed malware will be unintelligible on disk, but will unpack itself at run-time to return to its original form. AV's attempt to exploit this by emulating the malware. AV emulators step through the unpacking portion of code, attaining the unpacked malware, which is again susceptible to signature matching. For malware, a natural counter to emulation is to detect when being emulated, and exit. But to do this, malware authors need to establish detectable differences between running inside an emulator, and running normally. These differences are known as artifacts. Malware authors could easily find artifacts if they were able to extract information while running inside emulators. The difficulty is that the emulators are opaque and do not allow emulated programs to communicate outside the emulator. In addition to that, AV's often have many anti-analysis techniques embedded in their code, so normal reverse engineering techniques like disassembly and debugging are severely impeded. To avoid the difficulties of traditional reverse engineering, we have developed a black-box technique to extract the return values of API's called from within AV emulators. These return values allow for easy detection of emulator artifacts, and thus evasion. In this paper, we describe our technique and its implementation called AV Oracle. We demonstrate its effectiveness against six current AVs. We give practical applications for our approach and then suggest ways to defeat. Finally, we give possible future directions of research that build on our current work.**

I. Introduction

Malware is clearly a large problem and is only getting worse. And from commercial malware to state-sponsored malware used for espionage, AV's are often central to defense against malware. AV's may have begun with simple byte signature matching but have now evolved to include a wide breadth of technologies. These technologies include signature matching, run-time behavioral monitoring, network monitoring, and emulation. Emulation is the newest techologyto be integreated into AV's. Emulation allows AV's to execute suspected malware in a contained environment. This allows the malware to expose itself to other forms of detection like signature matching or behavioral monitoring. It this technology and it's use in AV's that we focus on.

In this paper, we use the terms malware and virus interchangably. We understand that virus is particular class of malware, and in fact our approach works using any type of malware. We choose to use the term virus at times because more often than not, our malware samples were viruses. We chose to use viruses because older viruses were small in size and easy accessible.

A fact that is central to this paper is that AV emulators do not provide any output. This is contrast to other emulators used in malware analysis, i.e. Bochs [1] or QEMU [2]. These emulators allow for total control of a process, but also allow that process to provide output. AV emulators on the other hand, do not allow any output, even if the program being scanned normally has output. This includes visual output, output to memory, and output to disk. This severly limits the ability to get information out of the emulator. AV emulators are designed specifically for emulating malicious executables, so it is in the emulators best interest to not let anything actually change the system. Otherwise, these changes could lead to infection of the system itself.

The paper is organized as follows. Section 2 covers related work. Section 3 describes our approach. Section 4 describes our testing methodlogy. Section 5 describes experimental results. Section 6 describes applications of our approach. Section 7 covers possible countermeasures to our approach. Section 8 covers future work and section 9 describes our conclusions.

II. Related Work

AV emulators are commercial technologies and are closed source, but we can learn some about the internals of these emulators through patents [3] [4] [5]. Other resources describing AV emulator technologies are press releases [6] [7]. Because of the closed source nature of most AV products, there has been much academic work evaluating the effectiveness of AV scanners through black-box analysis. None of the work so far has focused on the emulator technology though.

In [8], Filiol shows how black-box analysis can be used to extract static byte signatures in AVs. By static byte signature, we mean a string of bytes in the malware used to identify it. Filiol demonstrates his technique in a wide-scale study of AVs, and proposes a technique that limits black-box analysis.

In a follow-on work [9], Filiol et al. selectively modify the behavior of a sample, instead of byte sequences. They then record whether the tested AV still detects the code or not. This approach is similar to [8], but is now considering behavioral analysis. They did not consider code emulation due to the current limitations of AVs at the time.

In [10], Borello et al. design a code morphing engine to generate malware samples. These samples were then used to do black-box testing against static AV scanning and scanning at run-time. It was found that AV's employ behavioral monitoring and behavioral blocking at run-time. Again, code emulation was not considered in the AV products tested.

[11] describes another black-box approach to determining static AV signatures. The authors do this by querying the AV with a program, then recording features about the program and whether it was detected or not. From this information they infer what the signatures are.

Two interesting self-published papers [12][13] on reverse engineering AVs were written by Tavis Ormandy. Ormandy is a security engineer at Google. In his papers, he uses white-box and black-box analysis to expose flaws in Sophos AV.

Even though AV emulators have not been specifically analyzed, there has been a large body of work in analyzing malware through the use of emulators.

[14] [15] shows how emulation has been successfully used against packed malware.

Along with using emulators to unpack malware, there are a lot of papers describing techniques to counter emulation. Peter Ferrie, the principle anti-virus researcher at Microsoft, has a great 14 part series of articles on anti-unpacking techniques [16]. Many of these target emulators.

With one exception, papers up til now have explored the related areas of emulator analysis and black-box AV analysis, but none have done both.

[17] stands out as targeting AV emulators . It was written and self-published by a virus author. It describes a way to detect AV emulators through black-box analysis. The author's technique involves calling an API with random parameters, recording the result, calling the API again with different parameters and recording the result. If the results of the two API calls are the same, even with different parameters, then assume the program is being emulated. This work most closely aligns with our own, in that it targets AV emulators through black-box analysis. The major draw back of the technique presented in [17] is that no information can ever be extracted from inside the emulator. Any emulator artifacts found while running inside the emulator are lost when the emulation ends. Our approach solves this problem. We present a way to extract information from inside the emulator for future use.

III. Our Approach

A. Overview

Our approach involves building a mapping of ASCII values to known viruses. The actual viruses are then XOR'ed and appended to an unpacking stub. The stub, with XOR'ed viruses and mapping, selectively unpacks viruses at run-time to expose information about the emulator. This simple process is best illustrated by describing each step in detail.

B. Step 0: Gather Known Viruses

Collect known viruses from places like [18] or [19]. There is actually a torrent of 45GB of viruses from [19] available at [20]. This is the our starting set of binaries B.

C. Step 1: Filter for Exact Signatures

To describe this step, we must first formalize a basic model for most AVs, where A is an AV scanner running on program x:

$$A(x) = \begin{cases} 0, & \text{if } x \text{ is not detected} \\ 1, & \text{if } x \text{ is detected with a general signature} \\ s, & \text{if } x \text{ is detected with a unique signature s} \end{cases}$$

$$s \in S \mid S \text{ is the set of all viruses detected by A(x)}$$

Using this model, we filter our starting set of binaries B, for viruses that return exact signatures when scanned with A. More formally we define V as:

$$V := \{\forall b \in B \mid A(b) = s \in S\}$$

D. Step 2: Encrypt Viruses

Now we encrypt the set V to create a set of encrypted viruses C. The encryption can be trivial like XOR with a static key k. It only has to transform the viruses enough for them to no longer be detected by the AV. We define C as:

$$C := \{\forall c \in C, \forall v \in V \mid c = E(v,k) \text{ and } A(c) = 0\}$$

and we define E's inverse D as:

$$v = D(E(v))$$

E. Step 3: Build Decryption Stub

In this step, we build a decryption stub which reads in the set of encrypted viruses C and selectively decrypts members of C at run-time. C is appended to the stub before run-time, so that it's part of the same binary. AV emulators will often intercept API calls made by an executable and return false information, except when those API calls operate on the executable itself. Presumably, some emulators allows an executable to have truthful access to itself in order to facilitate accurate unpacking.

The encryption stub will contain a one-to-one mapping f of the printable ASCII set P to the set of encrypted viruses C.

$$P := \{\forall a \in ASCII \mid int(a) \geq 32, int(a) \leq 126\}$$

$$f : P \rightarrow C$$

This should allow us to perform the following operation in the decryption stub:

```
string = "cat"
for each char in string do
    D( f(char), k )
end for
```

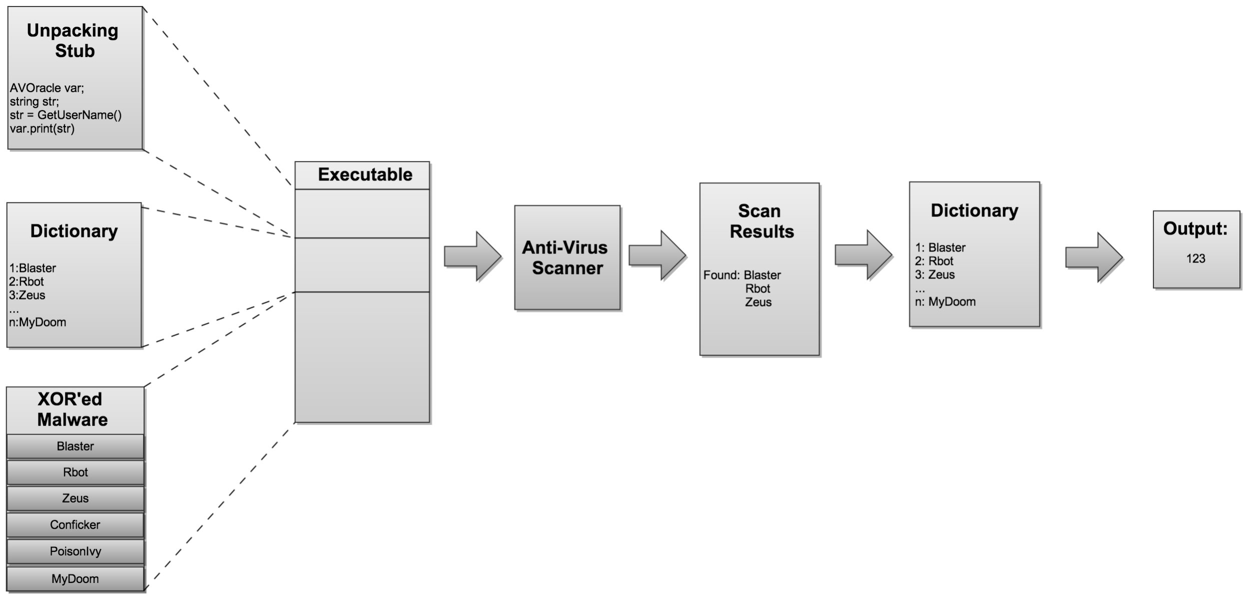

Fig. 1. This is illustrates the process running a sample program through an AV. It begins on the left with composing an executable with an unpacking stub, dictionary, and encrypted malware. This executable is then scanned by an AV. The output from the AV is parsed and cross-referenced with the malware dictionary for the corresponding ASCII characters. These characters are then printed as output.

When the code is emulated, the AV will detect and print viruses with signatures $s_x, s_y, s_z \in S$. Knowing the mapping f beforehand, we parse the output of the AV by running the inverse of f on the detected viruses. This will give us the ASCII characters that are assigned to those viruses:

$$f^{-1}(s_x) = \text{‘}c\text{’}$$
$$f^{-1}(s_y) = \text{‘}a\text{’}$$
$$f^{-1}(s_z) = \text{‘}t\text{’}$$

In a sense, we have printed through the emulator. In this last example, printing something that we know *a priori* is hardly useful. In the next step, we show how we can use this mechanism to extract useful information, i.e. the return values of API calls.

F. Step 4: Integrate Useful Queries with Stub

Now that we have a mechanism to output infromation from within the emulator, we will use it to extract information normally unavailable outside the emulator. A simple modification to our stub is as follows:

```
string = GetUserName()
for each char in string do
    D( f(char), k )
end for
```

This will give us the return value for the API call GetUserName(), as seen from within the emulator. If this value is different than what is returned when running this API call outside the emulator, then we have found an artifact of the emulator. Emulator artifacts can be used by malware for detection and manipulation of the emulator.

IV. Testing Methodology

A. Virus Total

To test our method, we needed AVs that use emulation. We used the 42 AVs on VirusTotal [21] as a sample set. To determine which AVs on VirusTotal contained emulation in their scanners, we first uploaded a common virus. Most AVs detected it with an exact signature, $A(x) = s$. We then XORed the virus and uploaded it again to acheive a zero percent detection rate. We then uploaded the same XORed version of the virus with an unpacking stub, that would XOR the virus back to its original form at run-time. With this version, 7 AVs detected our sample. Out of these 7 AVs, 6 were chosen for testing because their evaluation versions offered full functionality. They are listed above in figure 2.

B. Environment

Our environment consisted of a Windows XP SP3 virtual machine (VM). Each of the six AVs was installed on its own VM. The VM software was VirtualBox[22] with FreeBSD as the host operating system.

V. Experiment Results

A. Initial Experiments

The goal of our initial experiments was to establish the ability to reliably unpack selected malware in all of the AV's. To do this we used the approach we described in section 3. We first established success on the simplest case, which is unpacking virus 1 if true, and virus 2 if false. This is illustrated in the following pseudocode:

AV	Version	Detections per Scan
Bitdefender	2013	1
Emsisoft	7.0.0.21	1
F-Secure	1.77 Build 243	1
GData	24.0.1	1
Kaspersky	13.0.1.4190 (e)	31
nProtect	2012.12.29.001	1

Fig. 2. This is largest distinctions between the each AV and its emulator was the number of detections per scan.

Kaspersky AV Emulator		
Data	Emulated	Bare Metal
HwProfInfo.szHwProfileGuid	{6D2074C9-558B-860D-1A90-1A8BAF609E64}	{e29ac6c0-7037-11de-816d-806e6f6e6963}
HwProfInfo.szHwProfileName	Profile 1	Undock Profile
lpSystemInfo–>dwPageSize	4096	4096
lpSystemInfo–>dwNumberofProcessors	1	2
lpSystemInfo–>dwProcessorType	586	586
lpSystemInfo–>dwAllocationGranularity	65536	65536
lpSystemInfo–>wProcessorLevel	15	15
lpSystemInfo–>wProcessorRevision	519	18434
osvi.dwBuildNumber	2600	7601
osvi.dwMajorVersion	5	6
osvi.dwMinorVersion	1	1
osvi.wServicePackMajor	2	1
osvi.wServicePackMinor	0	0
osvi.wSuiteMask	256	256
userName	Administrator	user
computerName	<randomized>	GATEWAY
uDriveMask	0x200001d	0x7c
NX_ENABLED	NO-NX	1

Fig. 3. This clearly shows the discrepencies between API return values inside Kasperky's emulator and outside of it. One value of interest is the computer name. It was randomally generated each time an executable was scanned.

```
x = rand() mod 2
if x==0 then
    D( f(0), k)
else
    D( f(1), k)
end if
```

Once this was established, we had a channel to bring information out of the emulator. This was successfully applied to all 6 AV's. This is a very narrow channel to communicate though.

One simple way to exapand the amount information we can communicate is by increasing the number of unique virus signatures in our dictionary.

The ASCII table has 127 characters total, of which the 96 printable ones we are concerned with. If we assign each character a different virus, we can leak 1 character. This can be scaled to n choose k, where n is 96, and k is the number of characters in the string we would like to leak. 96 choose 3 is 142880, which is a reasonable number of virus definitions to include in our dictionary. So this system would get us 3 characters of output per scan.

Another way to expand this communication channel is to do multiple scans. For instance, we begin by establishing the length of the string we want to communicate. This could be done with a binary search within the unpacking stub. Once we have the length of a string, we can communicate a single character of that string per run. This is all based on the assumption that the string will be constant over multiple runs. This assumption can be verified by applying this technique over many multi-run experiments. We did encounter randomized API return values in one of the emulators. We will talk more about that in the following section.

And finally, there are always various compression techniques. We chose not to pursue compression for expanding the communication channel because we wanted to prove the concept possible before we began optimizing. We see all of the previous techniques as plausible ways to improve our technique and plan to do implement them in the future.

B. Additional Experiments

One AV that stood out from the rest was Kaspersky. It's biggest distinction was that it detected 31 viruses per scan. This gave a much larger communication channel out of the emulator. This allowed us to run additional experiments without having to implement any of the optimization techniques we mentioned at the end of the last section. It was through Kaspersky AV that the power of our technique is really demonstrated.

With the increased bandwidth, we were able to efficiently extract API return values. Figure 3 lists several API return values, as returned while running inside the emulator and outside of it.

The most interesting value in the table is the hardware GUID string in the first row. This serves as a 38-character unique id for Kaspersky.

Another value that stood out was that of the username. This was different across scans, even on the same executable. We did not explore the entropy of this randomized value, but plan to in follow-on research.

In addition to our original environment, we ran Kaspersky AV in several more, including bare-metal/WinXP, bare-metal/Win7, VM/WinXP, and VM/Win7. The values in figure 3 were consistent across all 4 environments.

VI. Applications

Our approach is well suited for circumvention of AVs and also in their testing. We describe specific applications in these two domains in this section.

A. Malware

The biggest weakness of the emulators we tested was returning constant distinct values for APIs called within the emulator. These values could be checked for within malware to detect emulators. Detection of an emulator would then lead the malware to act benignly. Not only is detection of an emulator possible, but detection of a specific brand of emulator. This could lead to targeted exploitation of the emulator, possible leading to infection of the host computer.

B. Anti-Virus

AV companies can use our technique to better evaluate their own exposure of information to the customer. Right now it seems that AVs are relying on the opaqueness of their emulators as a source of security. We think this is a mistake. We have shown that emulators are not just a source of security, but actually increase the attack surface of the AV scanner itself.

VII. Prevention

We do not think there is any security advantage for keeping the return values for API calls constant and distinct across scans. Ideally, values should be indistinguishable from those received when running in an unemulated environment. To achieve this, AVs could generate random return values for API calls that fit within the possibility of normal API return values. AVs have to be careful to choose a space large enough that malware authors cannot enumerate all possible generated return values and embed all of them in their malware. Also, generated values cannot have any qualities of format or content that could be used to distinguish them from normal return values.

VIII. Future Work

We are now in the process of developing a more powerful API built on top of AV Oracle primitives. Currently, we have APIs to dump CPU registers and arbitrary memory address. We plan on doing extensive testing using this API frameowork in future work.

We think that AV Oracle and other frameworks built on top of it could lead to more efficient emulator detection and exploitation.

IX. Conclusion

In this paper, we have described a technique to extract information from AV emulators. These emulators are normally opaque and difficult to reverse engineer. Our approach used only blackbox analysis to avoid the normal difficulties of reverse engineering programs with protection mechanisms. We built a program titled AV Oracle and demonstrated it against six current AVs. All six AVs we tested were succeptible to our technique.

Acknowledgment

The authors would like to thank Patrick Biernat for his contributions in testing and improving the software used in this research.

References

[1] Bochs, http://bochs.sourceforge.net/.
[2] Qemu,http://wiki.qemu.org/.
[3] Sergeh Y. Belov. Method for accelerating hardware emulator used for malware detection and analysis, 2009.
[4] Mikhail A. Pavlyushchik Alexey V. Monastyrsky, Andrey V. Sobko. System and method for detecting multi-component malware, 2009.
[5] Mihai Novitchi. Anti-malware emulation systems and methods, 2012.
[6] Eugene Kaspersky. Emulation: A headache to develop but oh-so worth it. BLOG.
[7] bitdefender. B-have - the road to success: A case study in the successful deployment of new anti-malware technology. Technical report, bitdefender, 2010.
[8] Eric Filiol. Malware pattern scanning schemes secure against black-box analysis. *Journal in Computer Virology*, 2(1):35–50, 2006.
[9] Eric Filiol, Grégoire Jacob, and Mickaël Le Liard. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology*, 3(1):23–37, 2007.
[10] Jean-Marie Borello, Éric Filiol, and Ludovic Mé. From the design of a generic metamorphic engine to a black-box classification of antivirus detection techniques. *Journal in Computer Virology*, 6(3):277–287, 2009.
[11] Kevin W. Hamlen, Vishwath Mohan, Mohammad M. Masud, Latifur Khan, and Bhavani Thuraisingham. Exploiting an antivirus interface. *Comput. Stand. Interfaces*, 31(6):1182–1189, November 2009.
[12] Tavis Ormandy. Sophail: A critical analysis of sophos antivirus. Blackhat, 2011.
[13] Tavis Ormandy. Sophail: Applied attacks against sophos antivirus. Technical report, 2012.
[14] Sbastien Josse. Secure and advanced unpacking using computer emulation. *Journal in Computer Virology*, 3(3):221–236, 2007.
[15] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM workshop on Virtual machine security*, VMSec '09, pages 11–22, New York, NY, USA, 2009. ACM.
[16] Peter ferrie's homepage, http://pferrie.host22.com/.
[17] Second Part To Hell. Dynamic anti-emulation using blackbox analysis. *Valhalla*, 2, October 2011.
[18] Offensive computing, http://www.offensivecomputing.net/.
[19] Vx heavens, http://vxheaven.org/.
[20] Vx heavens database, http://thepiratebay.sx/torrent/7066921/.

[21] http://www.virustotal.com.
[22] https://www.virtualbox.org/.

APPENDIX

Virus Total Results - April 23, 2013			
Anti-Virus	Detected Virus	Detected XOR'ed Virus	Detected Unpacked Virus
Agnitum	CLEAN	CLEAN	CLEAN
AhnLab-V3	CLEAN	CLEAN	CLEAN
AntiVir	BAT/Asscom	CLEAN	CLEAN
Antiy-AVL	CLEAN	CLEAN	CLEAN
Avast	Asscom-164	CLEAN	CLEAN
AVG	CLEAN	CLEAN	CLEAN
BitDefender	BehavesLike:Bat.Shimmer.Gen	CLEAN	Dropped:BAT.Asscom.A
ByteHero	CLEAN	CLEAN	CLEAN
CAT-QuickHeal	CLEAN	CLEAN	CLEAN
ClamAV	Bat.Asscom.164	CLEAN	CLEAN
Commtouch	Virus_Dropper!7ebf	CLEAN	CLEAN
Comodo	Virus.Bat.Asscom.A	CLEAN	CLEAN
DrWeb	Bat.Asscom	CLEAN	CLEAN
Emsisoft	BehavesLike:Bat.Shimmer.Gen (B)	CLEAN	Dropped.BAT.Asscom.A (B)
eSafe	Win32.Trojan	CLEAN	CLEAN
ESET-NOD32	BAT/SS.b	CLEAN	CLEAN
F-Prot	Virus_Dropper!7ebf	CLEAN	CLEAN
F-Secure	BehavesLike:Bat.Shimmer.Gen	CLEAN	Dropped.BAT.Asscom.A
Fortinet	BAT/Asscom.B	CLEAN	CLEAN
GData	BehavesLike:Bat.Shimmer.Gen	CLEAN	Dropped.BAT.Asscom.A
Ikarus	Virus.BAT.SS	CLEAN	CLEAN
Jiangmin	BAT/Asscom.164	CLEAN	CLEAN
K7AntiVirus	Virus	CLEAN	CLEAN
K7GW	CLEAN	CLEAN	CLEAN
Kaspersky	Virus.Vat.Asscom.164	CLEAN	Virus.BAT.Asscom.164
Kingsoft	CLEAN	CLEAN	CLEAN
Malwarebytes	CLEAN	CLEAN	CLEAN
McAfee	Bat/ass	CLEAN	CLEAN
McAfee-GW-Edition	Bat/ass	CLEAN	CLEAN
Microsoft	Virus:BAT/Asscom	CLEAN	CLEAN
MicroWorld-eScan	BehavesLike:Bat.Shimmer.Gen	CLEAN	CLEAN
NANO-Antivirus	Trojan.Script.Asscom.gkqa	CLEAN	CLEAN
Norman	CLEAN	CLEAN	CLEAN
nProtect	CLEAN	CLEAN	Dropped:BAT.Asscom.A
Panda	Bat/Asscom	CLEAN	CLEAN
PCTools	Asscom (bat)	CLEAN	CLEAN
Sophos	Asscom-164	CLEAN	CLEAN
SUPERAntiSpyware	CLEAN	CLEAN	CLEAN
Symantec	Asscom (bat)	CLEAN	CLEAN
TheHacker	CLEAN	CLEAN	CLEAN
TotalDefense	CLEAN	CLEAN	CLEAN
TrendMicro	BAT_ASSCOM.A	CLEAN	CLEAN
TrendMicro-HouseCall	BAT_ASSCOM.A	CLEAN	CLEAN
VBA32	Virus.Bat.Asscom.164	CLEAN	CLEAN
VIPRE	CLEAN	CLEAN	CLEAN
ViRobot	BAT.Asscom	CLEAN	CLEAN

TABLE I
THIS IS THE TABLE OF VIRUS TOTAL RESULTS. CLEAN INDICATES THAT THE AV DID NOT IDENTIFY ANYTHING MALICIOUS.