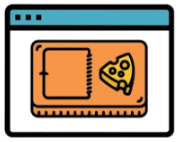
Recommended readings:

1. Ether: Malware Analysis via Hardware Virtualization Extensions. Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. In Proceedings of The 15th ACM Conference on Computer and Communications Security (CCS 2008), Alexandria, VA, October 2008. http://ether.gtisc.gatech.edu/

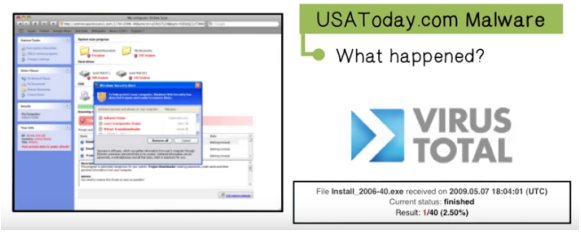
You can find the summary of the paper at the end of the document.

Let's review why malware is such a big security problem. It is very easy for malware to get on a user's computer.

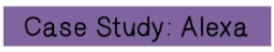


Say a user browses the web, for example, reading the news at USAToday.com. And this may result in a compromise to his computer.

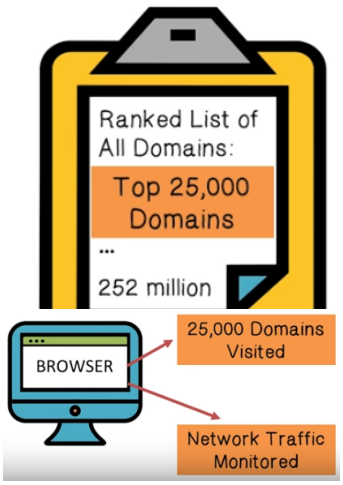
The reason is that USAToday.com's ad network can be compromised, so when a user uses his browser USAToday.com, his browser will be served with malicious JavaScript.



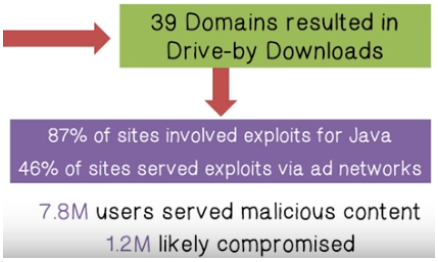
And of course, this script is bundled with ad and this malicious JavaScript will automatically direct the user's browser to a rogue AV website. Rogue AV stands for rogue antivirus software. The end result is that the user is tricked to download a rogue AV, which is actually a malware.



Here's a case study on how many sites may have been compromised and serve malicious contents. The researchers analyzed the Alexa top ranked domains. They allowed 252 million domains worldwide in 2016. A research system was created to examine Alexa top 25,000 domains each day. Essentially the browser within a virtual machine is forced to visit each domain. The network traffic that follows a visit to these websites were analyzed to determine whether drive-by downloads had occurred.

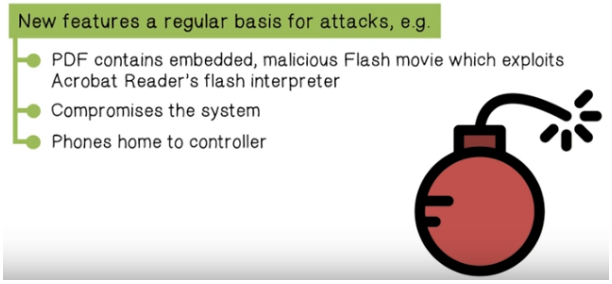


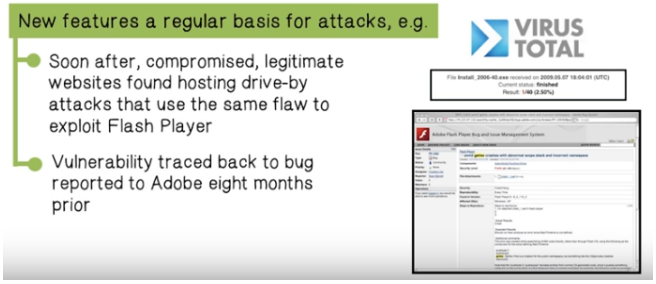
The result showed that 39 domains resulted in drive-by download. And among these, 87% of these sites involved exploits of JavaScripts. And 46% of these sites served the exploits through ad networks.



From Alexa statistics, about 7.5 million users visited these 39 sites. And about 1.2 million user computers are likely compromised because these computers do not have adequate defenses. For example, they have out-of-date anti-malware software.

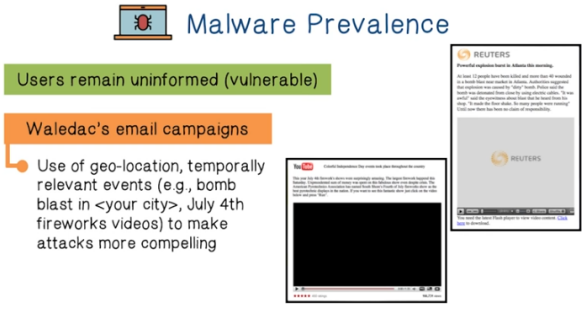
Collectively the attackers always develop new malware and new ways to spread malware widely. For example, an exploit was developed on mobility in Acrobat Reader's Flash interpreter. As defenders we may discover that one of our user's computers was compromised by this exploit. For example, we may be able to observe the phone home traffic from the compromised computer including the command-and-control traffic.

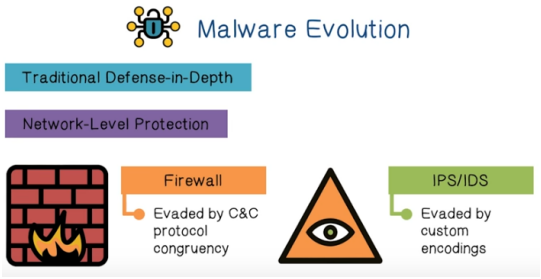




And the chances are that our user is not alone because we may soon discover that many websites are hosting the same kind of exploits. This is indeed the real case.

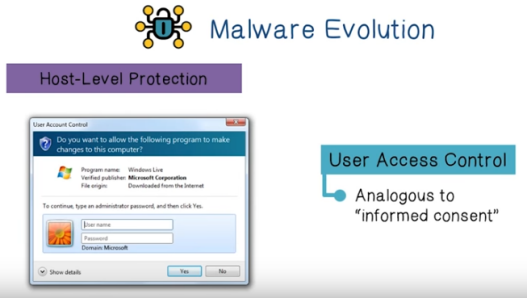
And of course, users remain the weakest link and often they are subject to social engineering attacks. For example, a very compelling email may make a user click on an active content, for example, a video or flash content. That results in a compromise such as this real case.

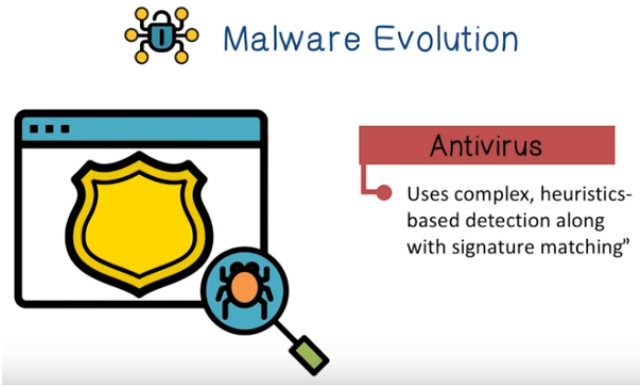




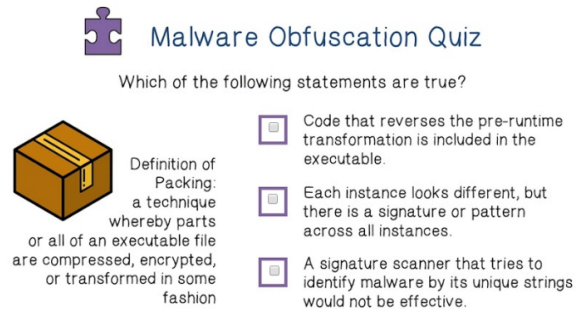
You may wonder, don't we already have defenses against malware? Yes, we do, but malware keeps evolving very fast and some of the traditional mechanisms are not adequate. We should deploy defense-in-depth, and for network protection we have firewalls as the prevention mechanism and IDS as the detection mechanism. But for firewall, command-and-control traffic can look just like normal traffic, such as visiting a webpage. For IDS that analyzes the payload of traffic, the encrypted or specially encoded malicious contents can evade such analysis.

On the host, if you ask for the user's consent, most often since the users often do not understand the security implications, they will simply say yes.

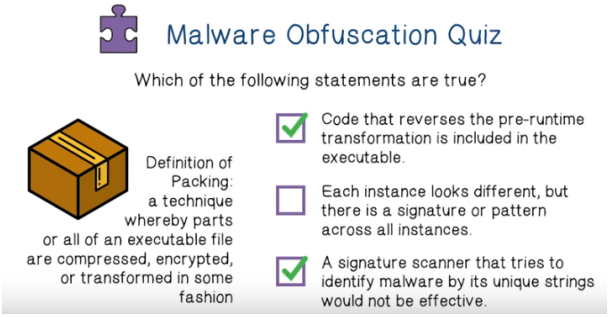




In terms of anti-virus software, the traditional signature matching approaches are not effective where malware uses obfuscation techniques. And so, we have to continue to develop more complex behavior-based analysis approaches.



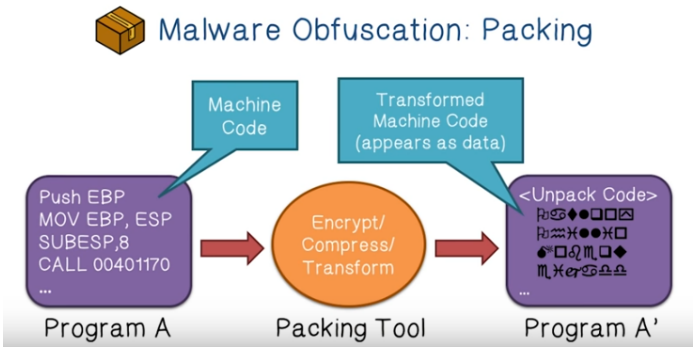
Now let us do a quiz. Based on this definition of packing, which is a typical obfuscation technique? Which of these statements are true?



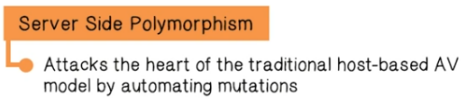
Since the malware contents are encrypted, and look random, a signature-based approach would not work. Therefore, the first statement is true. And, since the malware contents are encrypted, and look different, there's no single signature that matches all the instances. So, the second statement is false. Of course, we need to include code that decrypts the malware in runtime so that the malware can execute. Therefore, the first statement is true. Note that we cannot simply use the code that is part of the compressed or the encrypted malware as a signature because legitimate programs can contain such instructions, for example, for digital rights management.

Malware Obfuscation: Packing

Now let's take a closer look at one of the most widely used obfuscation techniques, packing.

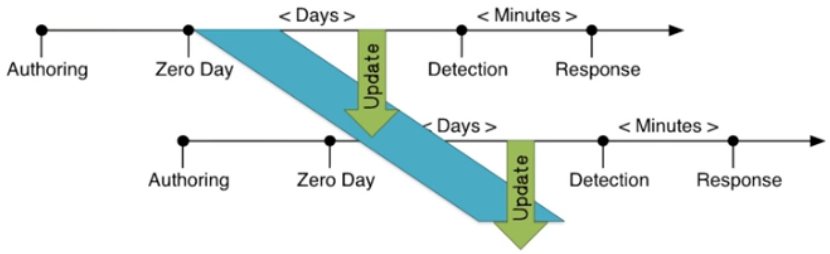


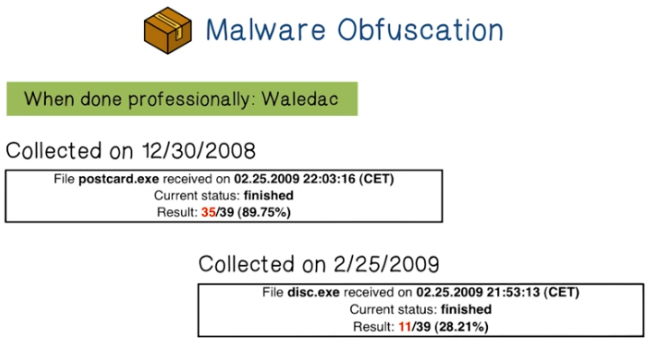
Given the original malware program, the packing tool would transform it, so that the transformed code looks random. Because it is encrypted, we randomly generate a key. And this happens each time the packing program is run on a malware program. That is, even for the same malware program, each packed instance will look different. And therefore, a signature based approach is not effective in detecting the malware. Furthermore, the transformed machine code looks like data. Therefore, a network IDS that looks for executables and email attachment will miss it.



If an anti-virus company obtains the zero-day malware, even though it is obfuscated, eventually the researchers can de-obfuscate the malware and discover its behaviors. But such analysis takes time and the attacker can make such analysis fruitless.

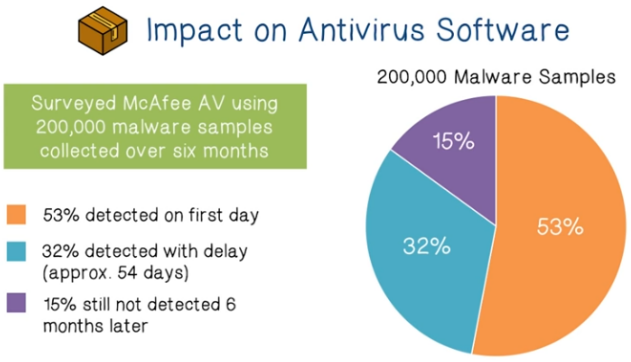
In particular, the attacker's server can continuously send an updated or new malware. That is obfuscated of course, to the compromised computers. Then in effect, the defenders or researchers have to deal with zero-day on new malware constantly. That is, by the time they have successfully analyzed a malware, it has become obsolete.

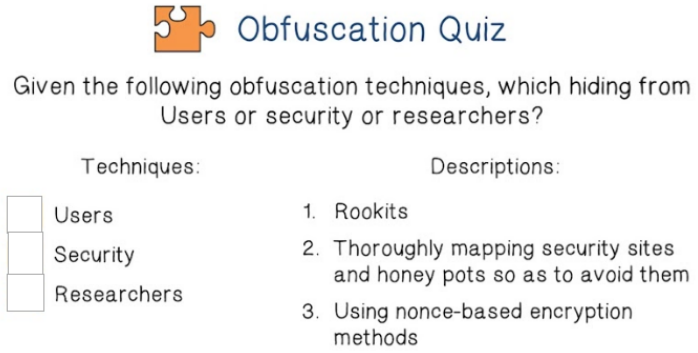




A real example of this server polymorphism is Waledac. This particular version of the malware postcard.exe was released on December 30, 2008 and by February 25, 2009, the majority of the antivirus software can detect it. But a new version of the malware disc.exe was detected by only a very small percentage of the antivirus software on the date it was released.

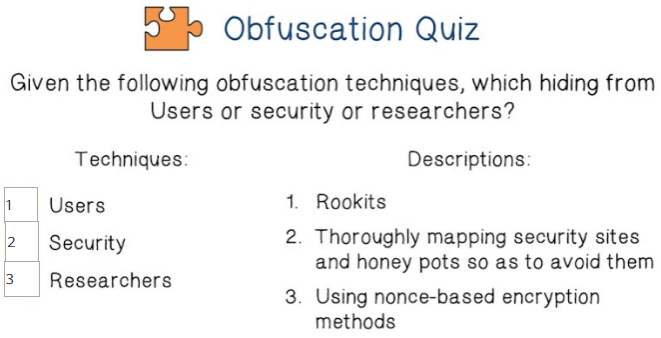
Here is an example showing the challenges the anti-virus industry is facing. The researchers surveyed McAfee anti-virus software using 20,000 malware samples collected over six months. 53% of the malware samples were detected on the day of release. 32% of the malware samples were detected with a delay. And the delay was on average 54 days. And 15% of the malware samples were not detected even six months later.

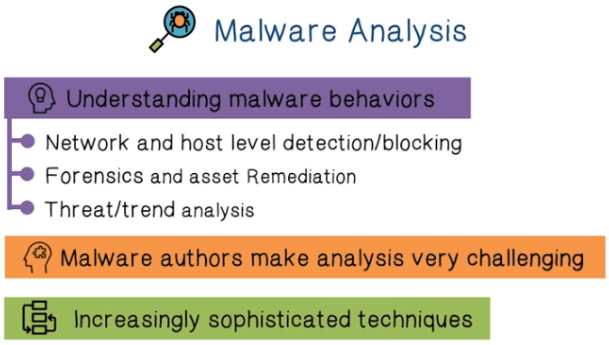




Now let's do a quiz. Obfuscation techniques are commonly used for one of three purposes, to hide from the users, to hide from security mechanisms, or to hide from researchers. Given these techniques, which one is hiding from the users or hiding from the security mechanisms or hiding from the researchers?

Rookits is used to hide the malware from the users. Mapping the security sites and honey pots are used to avoid security mechanisms, such as detection mechanisms. Using nonce-based encryption schemes would make cryptanalysis more difficult, and this is to hide from the security researchers.

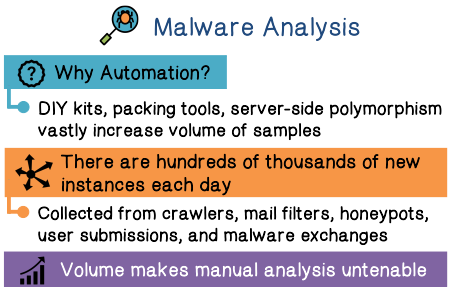




Now let us discuss malware analysis. What are the benefits of malware analysis?

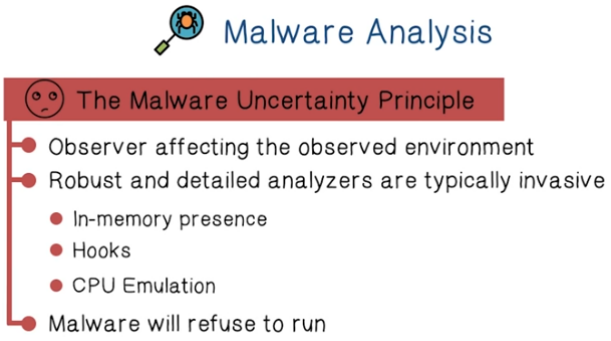
- If we understand malware's network behaviors, we can create rules to detect and block malware traffic.
- If we understand malware's on-host behaviors, we can repair the compromised hosts.
- If we have knowledge of the malware's support infrastructure on the Internet and its evolution history, we can also analyze its trend and threat scope, such as how widespread the malware is and the likely next target, etc.

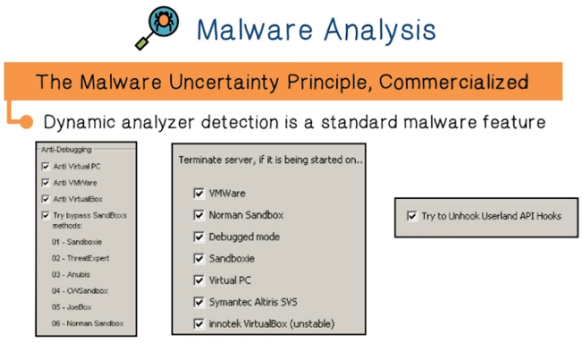
Of course, the attackers try to defeat malware analysis and are using the most sophisticated techniques available.



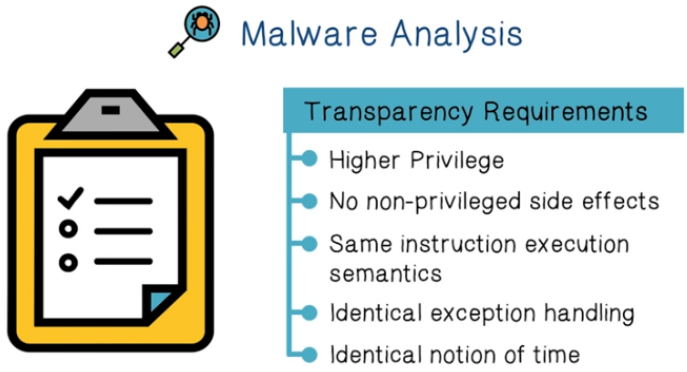
Another challenge in malware analysis is the volume of malware is really huge. Available and automated tools make the job of creating obfuscated malware very easy of course. And in fact, there are hundreds of thousands of new samples every day. Since the malware creation process is already automated, it is imperative that malware analysis has to be automated as well.

In addition to automation, malware analysis also needs to be transparent so that the malware does not know it is being analyzed. Otherwise the malware may refuse to run, or purposely alter to its behaviors to fool the analysis. This is the so-called malware uncertainty principle, but this is very challenging. In physics, the Heisenberg principle says that an observer will inevitably affect the observed environment. In malware analysis, our tools may be so invasive that the malware can detect them if we are not careful.

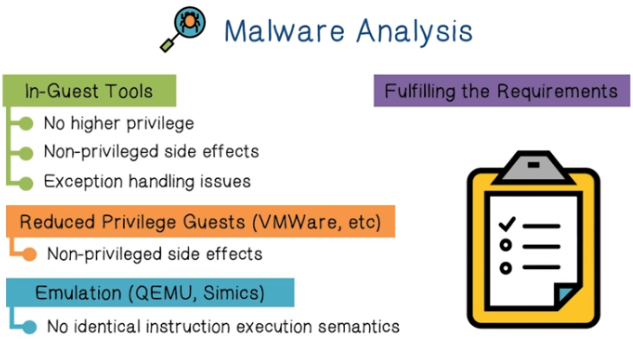




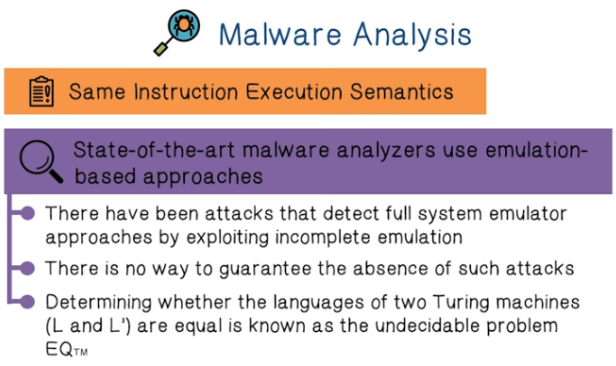
In fact, malware authors already tried to actively detect malware analysis tools. In malware creation tool kits, there are standard point-and-click options to add logics to detect various malware analysis methods, such as these examples.



So, we need malware analysis to be transparent to malware. That is, malware should not be able to detect that it is being analyzed. But how do we fulfill such transparency requirements? A malware analyzer should be at a higher privilege level than a malware, so that the malware cannot directly access, or know about, the analyzer. In addition, the malware may use operations to try to find the presence of the analyzer. For example, it may try to find a side effect of analysis. These operations should be privileged, so that the malware cannot directly get the answers back. And, since the analyzer is at a higher privilege level, it can actually lie the answers back to the malware. That is, the malware gets the wrong answer and does not know about the side effects. And obviously, the malware should get the same correct result of instruction execution as if the analyzer is not present. Likewise, the malware should see identical signals and instruction handlings, as well as time durations.

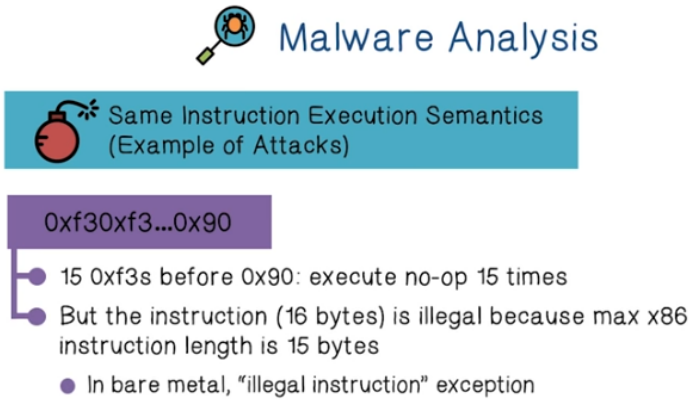


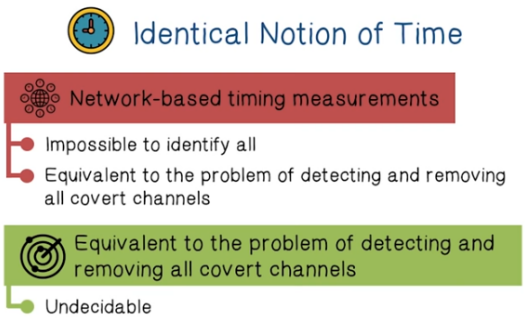
In terms of fulfilling the transparency requirements, most interesting tools fall short. For example, even if an analyzer is in guest this means that the analyzer runs on the same machine and has the same privilege as the malware, and then the analyzer does not have high privilege. Some of the analysis' side effects can be discovered by a malware without privilege operations. And the exceptions trapped by the analyzer may be discovered by the malware. If the analyzer runs in a virtual machine that is based on software virtualization such as VMWare then there could be side effects that can be discovered by a malware without privileged operations. If the analyzer runs in emulation environment such as QEMU the execution semantics of instructions maybe different from real hardware.



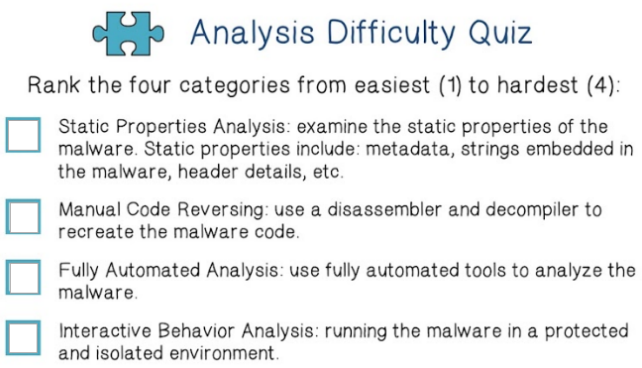
Emulation based malware analysis tools are the most widely used. But they have major shortcomings. The main issue here is that these emulation environments do not fully emulate the hardware. There are corner cases where a set of instructions give different results on emulation environments versus hardware. And there have been attacks based on these corner cases to detect the emulation environments. But the bigger problem is that there is no way to eliminate all these corner cases. In fact, in theory, there is no way to guarantee the absence of such attacks. And the reason is that the so-called EQTM is not decidable, which means that when you view an emulator, you cannot determine that it behaves exactly the same as the real machine. In other words, you cannot rule out the possibility that there are situations that your emulator and the real machine behave differently.

Here is a simple example of the discrepancies between emulator and real hardware. That is, the real hardware will give an illegal instruction exception, but an emulator will happily execute the instruction.



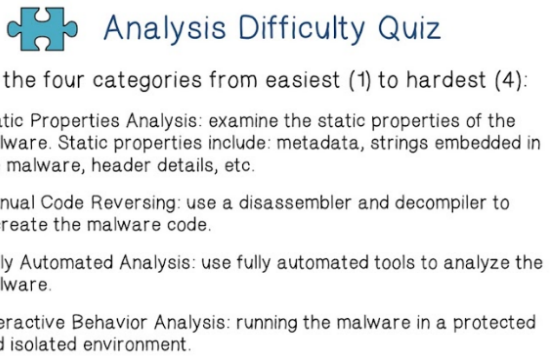


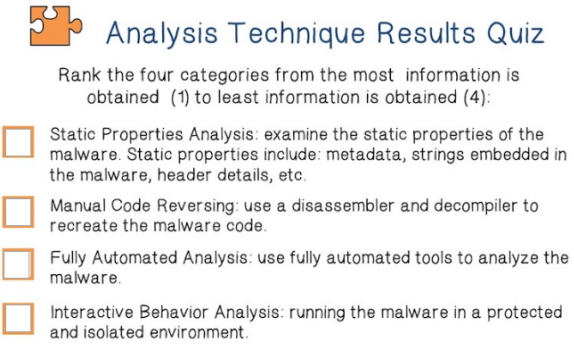
The most challenging transparency requirement is the identical notion of time, in particular, if malware uses network timing measurements to infer the presence of analyzer. For example, the analyzer causes delay, or the website that the malware connects to is not a real one. There are many direct or indirect measurements that a malware can perform and it is impossible to identify all direct or indirect measurement techniques. In fact, the problem of identifying all network-based timing measurements is equivalent to the problem of detecting and removing all covert channels. This problem has been proved to be undecidable.



Now let us do a quiz. There are four basic types of malware analysis that can be performed. Please rank these techniques from the easiest to the hardest.

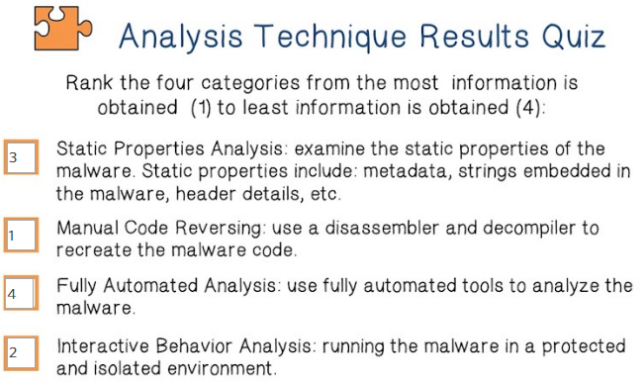
Fully automatic tools should be first performed, they are typically the easiest. Static analysis should be tried next. Interactive analysis requires more skills and time to set up. The most difficult one is manual reverse-engineering. It requires a lot of skills and time.

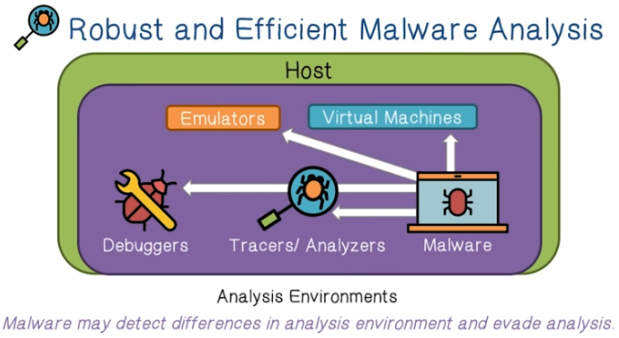




Let us do another related quiz. Rank these analysis techniques by how much information each technique can reveal.

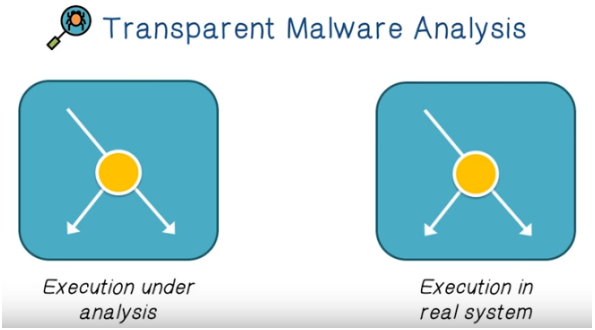
It is not surprising that the harder an analyst's technique is, the more information it can yield.

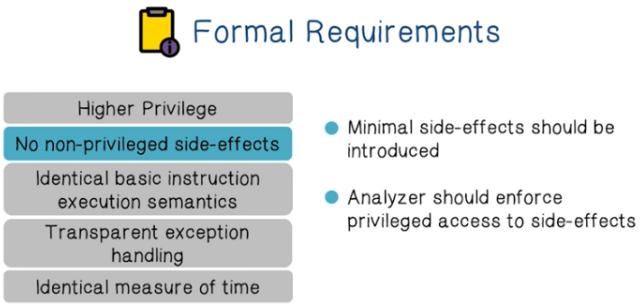




Now let us discuss how to build a robust and efficient malware analysis system. By robust, we mean it is transparent and is not easily detected and evaded by malware. By efficient, we mean it is automated and fast. We focus on host-based analysis. That is, learning malware on a machine and analyze its behaviors. As a recap, the malware will try to detect the presence or the side effect of the analyzer.

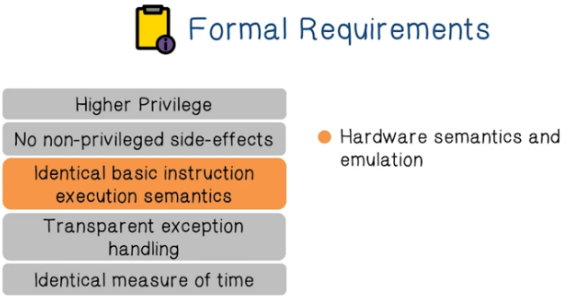
And our goal is to achieve the same result of malware execution, regardless of the presence of the analyzer.

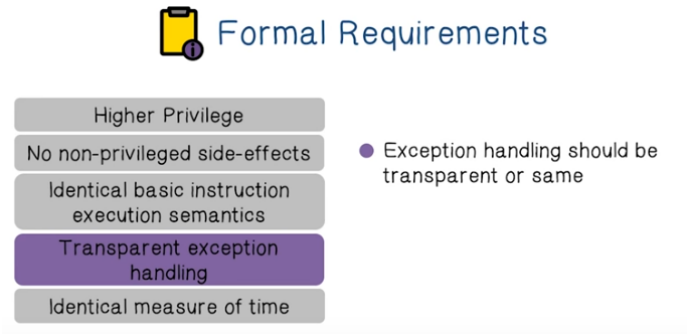




So let us analyze the transparency requirements again and see how they can be fulfilled. For higher privilege, let us put analyzer in hardware or a virtual machine. For the requirement of no non-privileged side effects, we need to make sure that no side effects can be detected by non-privileged instructions, that is, those that the malware can execute and get results. Of course, the analyzer will inevitable introduce side effects. But if the malware has to use privileged instructions to learn about these side effects, these instructions have to be executed at a higher privilege level than the malware, and then the answers are provided back to the malware. The analyzer, since it is at a high privilege level, can lie the answers back to the malware.

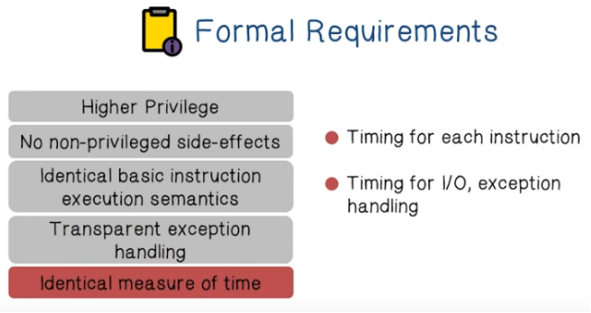
For the requirement of identical basic instruction execution semantics, we need to use the real hardware.

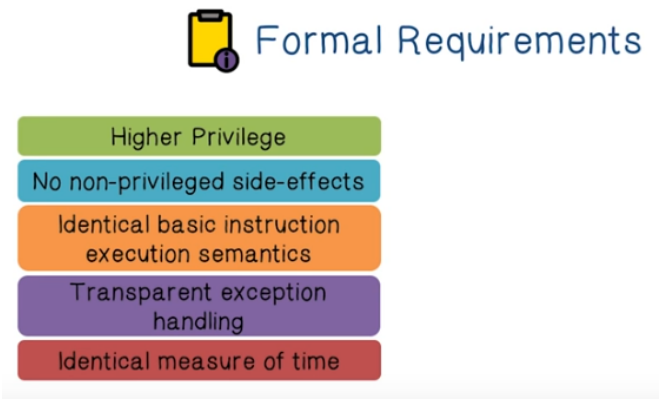




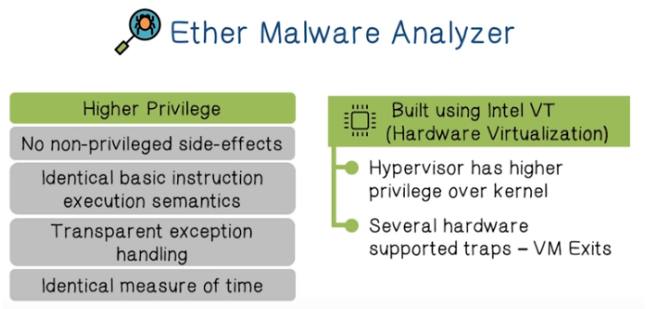
Likewise, we need to make sure that exception handling will be the same as if on real hardware.

For timing, we need to know the timing of each instruction I/O, and exception handling on the real hardware, and make sure that the malware can only use privileged instructions to get timing measurements on the host so that the analyzer can lie the answers back.

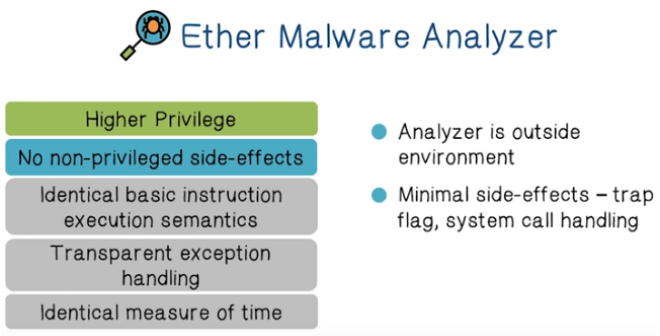




Just to recap, in order to achieve transparency, a malware analyzer needs to be on a higher privilege, and has no non-privileged side effects, identical basic instruction execution semantics, the same transparent exception handling, and identical measurement of time.

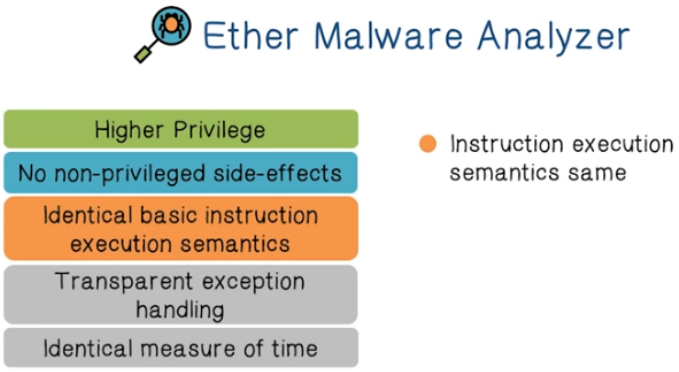


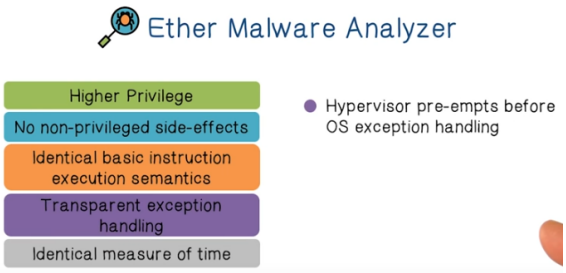
Here is an example of a transparent malware analyzer that they built at Georgia Tech called Ether. By the way, it is open source. Let us briefly describe how Ether fulfills the transparency requirements. Ether is built using Intel VT for hardware virtualization. The Hypervisor has higher privilege over the OS kernel. Therefore, Ether has higher privilege than malware. Some of these hardware supported traps further guarantee this higher privilege.



Ether runs outside of the virtual machine where the malware runs. Therefore, there is minimal side-effects that malware can directly observe.

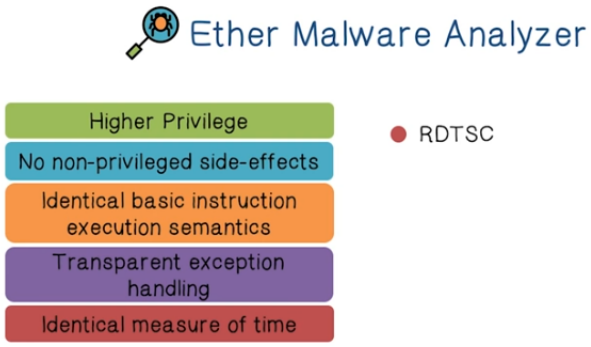
Ether uses hardware-based virtualization and therefore, the instruction execution semantics are the same as on hardware.

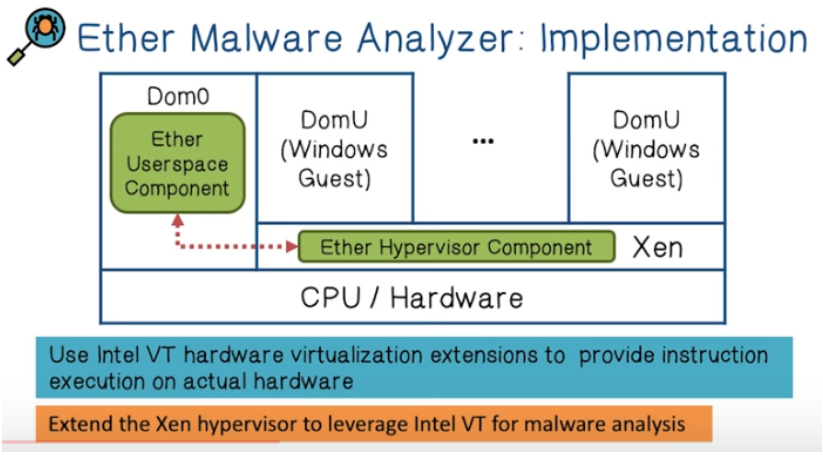




For exception handling, again, hardware-based virtualization allows the same semantics as on hardware.

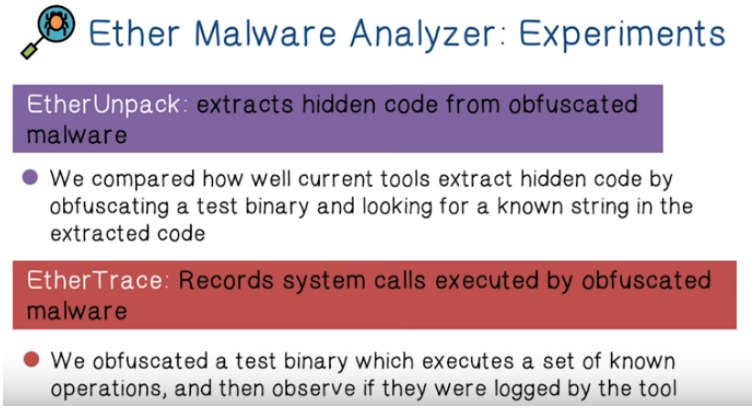
For on-host time measurement, the operations come down to the use of read time-stamp counter instruction. This is a privileged instruction and therefore, and the Hypervisor and ether can control the return results to the malware when the malware tries to get a measurement of time.

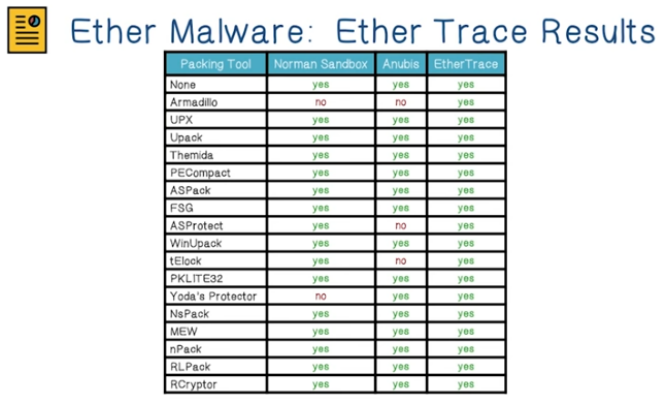




Here is the architecture of Ether. Ether has a component within Xen, the Hypervisor. And the rest of Ether is in Dom0, a separate, privileged virtual machine. The malware runs on a separate user-level virtual machine called DomU. Ether provides a fine-grained instruction by instruction examination of malware, and also a coarse grained system call by system call examination.

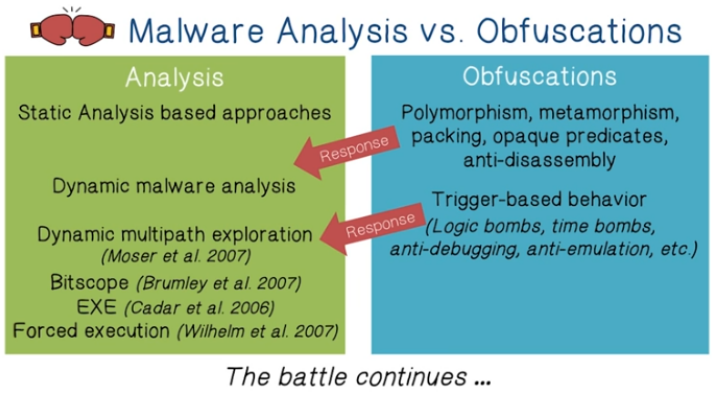
We created two tools to evaluate Ether. The fist is EtherUnpack. It extracts hidden code from obfuscated malware. The second one is EtherTrace, it records system calls executed by obfuscated malware. We then compare both of these tools to the current academic and industry approaches. For EtherUnpack, we compared how well current tools extract hidden code by obfuscating a test binary and looking for a known string in the extracted code. For EtherTrace, we obfuscated a test binary, which executes a set of known operations and then observe if they were logged by the tool.





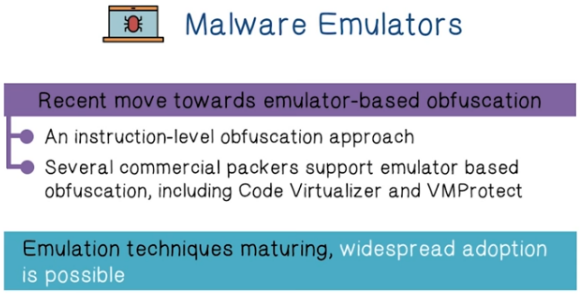
Packing Tool	Norman Sandbox	Anubis	EtherTrace
None	yes	yes	yes
Armadillo	no	no	yes
UPX	yes	yes	yes
Upack	yes	yes	yes
Themida	yes	yes	yes
PECompact	yes	yes	yes
ASPack	yes	yes	yes
FSG	yes	yes	yes
ASProtect	yes	no	yes
WinUpack	yes	yes	yes
tElock	yes	no	yes
PKLITE32	yes	yes	yes
Yoda's Protector	no	yes	yes
NsPack	yes	yes	yes
MEW	yes	yes	yes
nPack	yes	yes	yes
RLPack	yes	yes	yes
RCryptor	yes	yes	yes

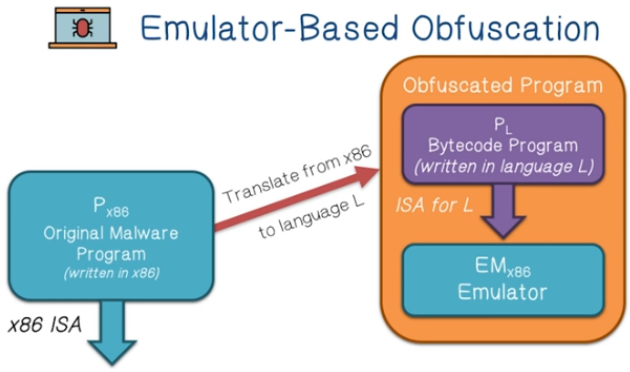
For EtherTrace again, the results show that Ether has much better transparency than other tools.



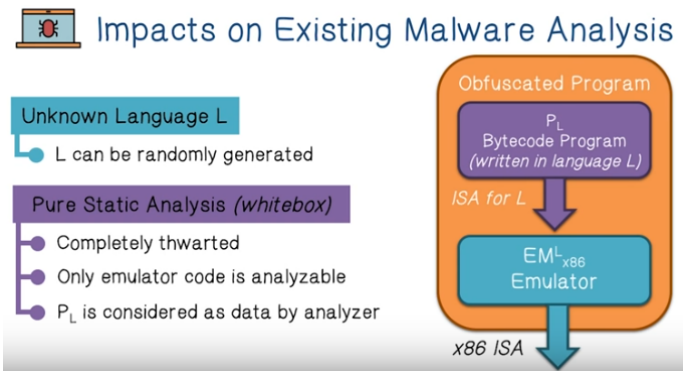
Now, Ether has limitations and it is only one of the tools in the continued battle between defenders or malware analysis and the attackers or obfuscations. Let us take a look at this model. Here, for each major categories of analysis approaches, the attackers come up with obfuscation techniques to defeat such analysis. For example, for static analysis, including scanning the binaries looking for fixed strings, the attacker can obfuscate the malware. For example, as a result of packing, the binary contents will look different from one instance to the next. We also use dynamic analysis, that is, running the malware. The corresponding obfuscation technique is trigger-based behavior. For example, the malware will not run until the time is right. Or it detects the fact that it is being analyzed in an emulation environment and so it stops. And since simple dynamic analysis now becomes inadequate, researchers have come up with various ways to force execution of this malware. And of course, this battle between analysis and obfuscation will continue.

A recent trend is that malware authors are starting to use emulation-based obfuscation techniques. And this is an instruction level approach. And there are several commercial tools out there and they can be used for legitimate purposes such as digital rights management.



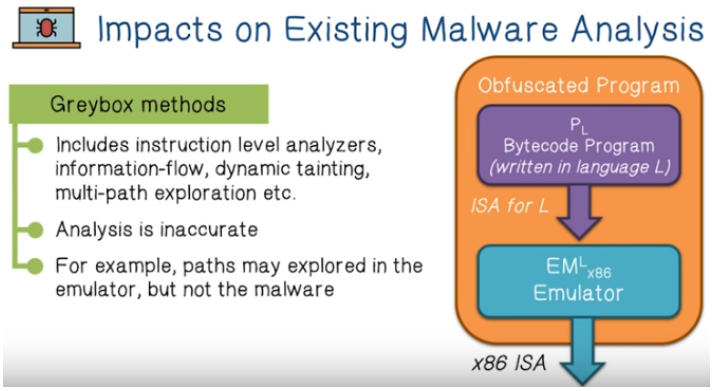


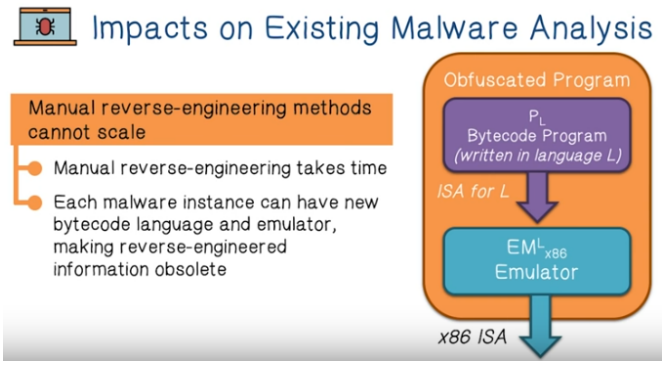
So how does emulator-based obfuscation work? Suppose we have the original malware and it is for x86 instruction set architecture, for example, your Intel based machines. The malware is then transformed into bytecode program of an arbitrary language, L. And then, this emulator, based on L, will emulate this microprogram on x86. That is, the obfuscated malware includes both the bytecode program and its emulator. And when the obfuscated malware runs on x86, the emulator will emulate this bytecode program and execute the malware logic.



So, what are the impact of emulation-based obfuscation on malware analysis? First of all, the malware program now is a bytecode of arbitrary language L, which is not known. In fact, the language L can be randomly generated and since we do not know the language L we cannot perform pure static analysis on malware bytecode program written in L. We can perform analysis on the emulator. The emulator actually is not specific to the malware bytecode program. It is only specific to the language L. In fact, the malware bytecode program is only one of the possible inputs to the emulator.

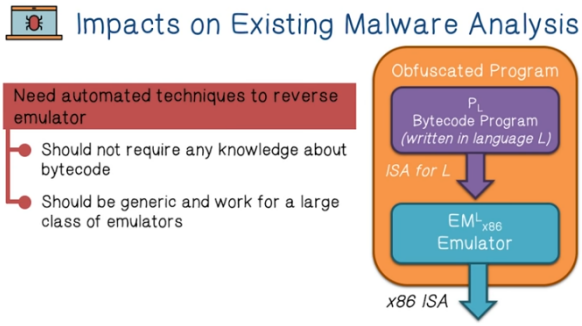
We can perform dynamic analysis including some code analysis. We call this greybox methods. But such analysis is actually performed on the emulator, not the malware code directly. The reason is that the executable is the emulator. The malware bytecode is the input. Therefore, the analysis results that we get are from the emulator, not directly from the malware bytecode.

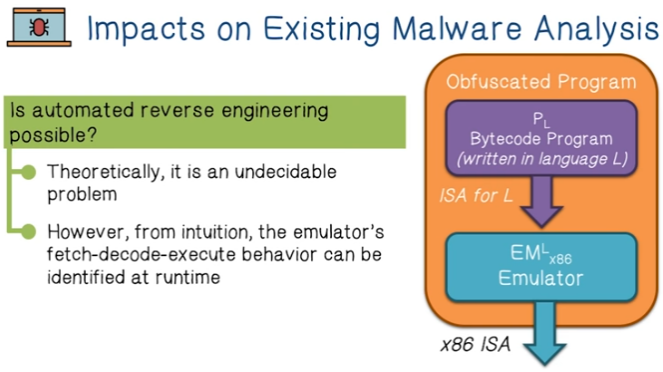




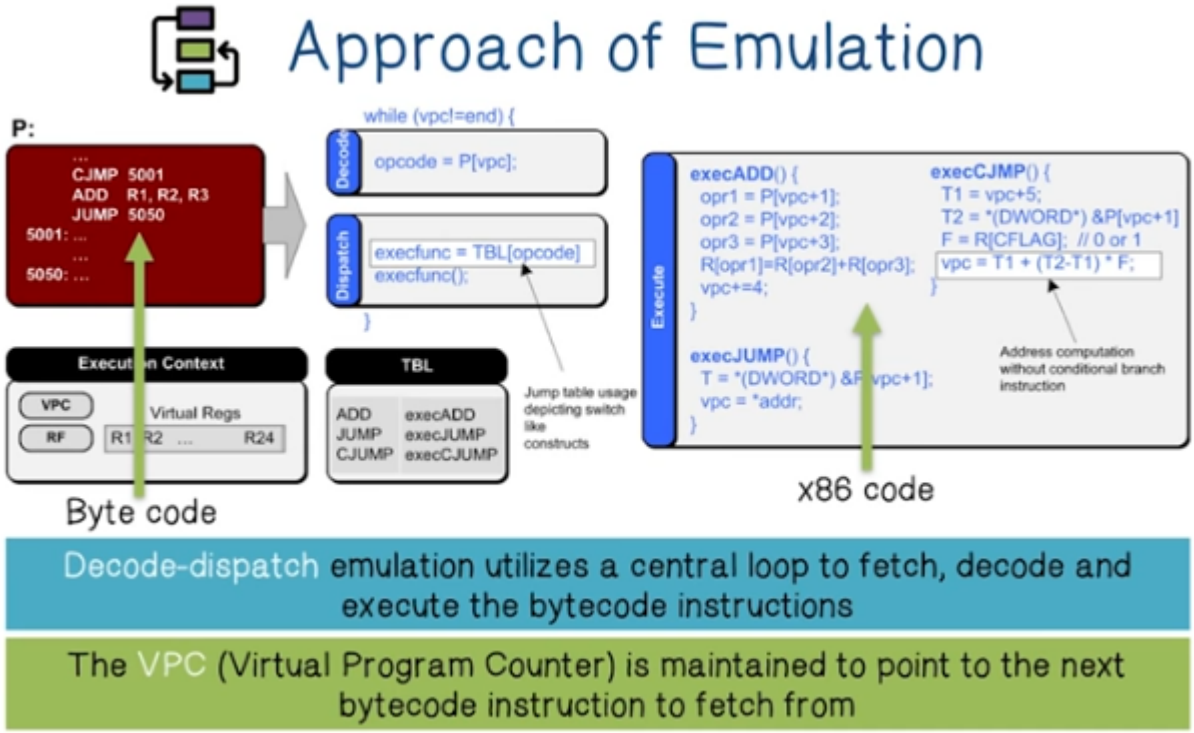
Manual reverse-engineering cannot scale, because each instance can have a different language L, and a different emulator.

And since the process of creating such confiscated malware is automated, we also need an automated approach to reverse engineer these emulators. By this, we mean that we should not require knowledge of the language L. And our approach should be general enough that will work on a large class of emulators.

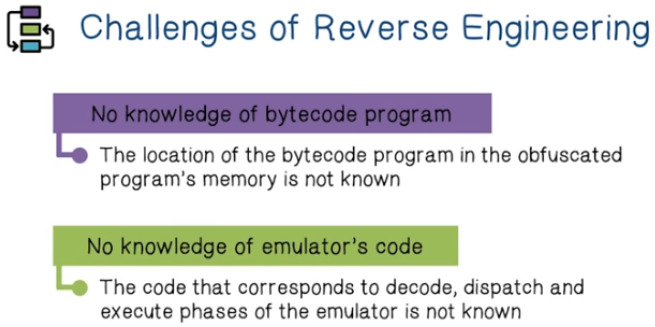




But, is an automated approach possible? In theory, it is not. On the other hand, most of the emulators have a fetch-decode-execute behavior that can be identified at runtime and can be the starting point of our reverse-engineering.

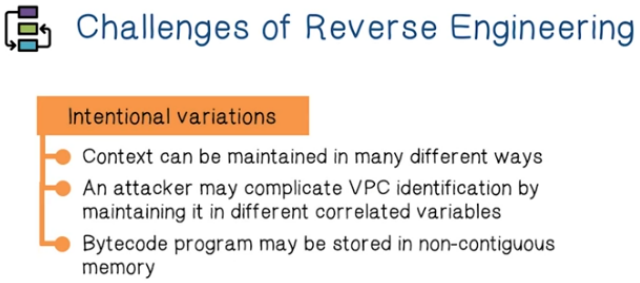


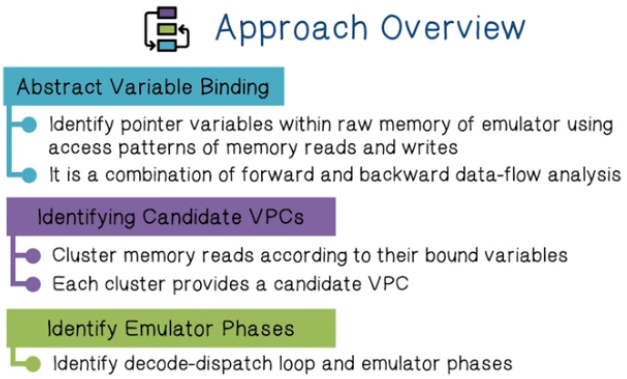
Now, let us understand how emulation works in a bit more detail. In fetch, decode, and execute, the emulator fetches the next bytecode, decodes to get its opcode, and looks up the execution routine for this opcode and then executes it. Notice that the execute routine executes real x86 machine code. The virtual program counter, VPC, is a maintained pointer to the next bytecode to fetch from.



Now, let us discuss briefly how we can reverse engineer emulator-based obfuscation. There are quite a few challenges. First of all, we do not even know where the bytecode program resides in memory. The emulator code responsible for fetch, decode, and execute, is also not known.

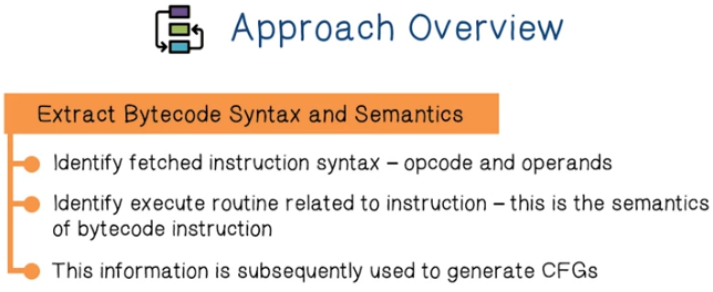
The malware author can certainly make reverse-engineering more difficult by changing how the emulation works.

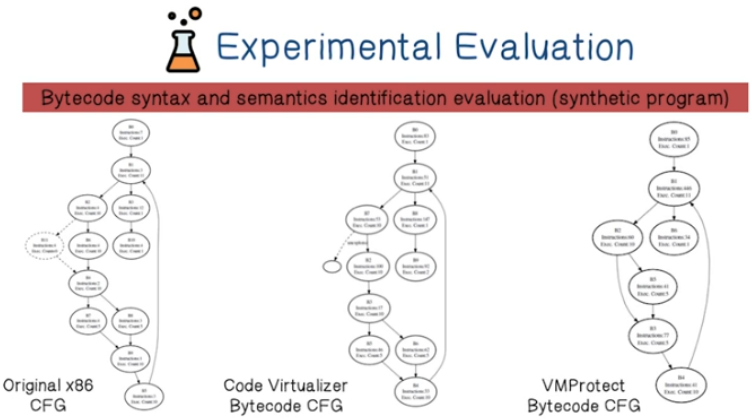




We develop a tool to automatically reverse engineer emulator-based malware, and here is a very high-level overview. The first step is to identify abstract variables, in particular, pointers. And of course, one of the most important pointers we need to identify is VPC. A VPC points to the next bytecode to fetch, decode, and execute. Therefore, once we identify the VPCs, we can identify the fetch, decode, and execute behavior in the emulator.

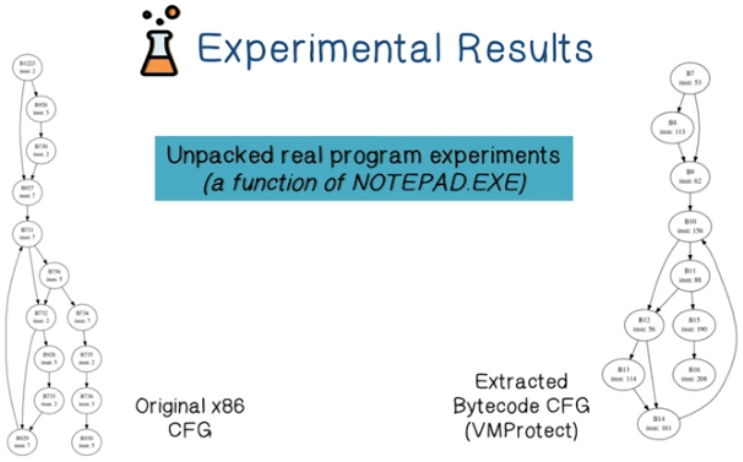
From the fetch, decode, and execute operations in the emulator, we can obtain the opcode and operands, as well as the execution routine of the malware bytecode. We can then construct a control flow graph of the original malware. And this will tell us the behaviors of the malware.





Here are some results of our experiments. We created a synthetic program. We then applied two emulation-based obfuscation techniques. One is called VMProtect, the other one is called Code Virtualizer. We then applied our tool to the obfuscated program. We then compared a control flow graph of the original program and the control flow graphs from our reverse engineering. Here is the control flow graph of the original program, and a graph from reverse engineering Code Virtualizer, and a graph from reverse engineering VMProtect. As you can see, our tool is successful in terms of extracting the main properties of the original control flow graph.

Here are the results from the experiments on a real program, NOTEPAD.EXE. Again, here is a CFG of the original program. And here is the result from reversing VMProtect. Again, our tool is able to obtain the main information and properties of the original CFG.



Ether: Malware Analysis via Hardware Virtualization Extensions

This paper covers a new malware analysis framework based around virtualization introspection named Ether. The paper is split into three major parts:
1. A formal framework for describing program execution and analyzing the requirements for transparent malware analysis. (page 2 – page 5);
2. Implementation of Ether. (page 5 – page 9);
3. Broad-scale evaluation and comparison of Ether with other approaches. (page 9 – page 11).

The researchers initially present background information on anti-VM malware detection and previous methods for avoiding this detection, it also includes why these methods have failed. A formal notation is provided to prove the transparency of Ether along with stating the 5 basic requirements that must be met for analysis transparency. The five basic requirements are as follows (page 4):
a. Higher privilege - it requires that the analyzer has a higher privilege than the maximum privilege that a malware sample can attain.
b. No non-privileged side effects – if the analyzer process induces side-effects, these must be handled using exception handlers that have higher privileges.
c. Identical Basic Instruction Execution Semantics – we need to guarantee that the same instruction has the same execution.
d. Transparent Exception Handling – there are two separate cases considered while looking into the exception handling of the instructions.
e. Identical Measurement of time – this requirement states that the timing received by the analyzer is the same as the one received by the underlying environment.

Ether is an extension of the Xen virtualization system and can run any guest operating system without any sort of modification. It can also run with the newer processors that support Intel VT (see section 4.3). It also talks about how Ether uses VT extensions to analyze malware and how it uses this to maintain transparency. The paper presents different methods to monitor system calls and also individual instructions without modifying the guest operating system to an extent where it can be detectable.

This is followed by the explanation of the reasoning and implementation of the Ether framework. The paper also provides two applications of the framework discussed. The first application is called the EtherUnpack (section 5.1) and this is used for fine-grained tracing of individual instructions. The results from Ether shows that it was able to run undetected by all malware samples while other unpacking tools are successful only with 50%-90% of the malware samples. The second application is the EtherTrace (section 5.2), it was built to perform coarse-grained tracing of system calls in the guest operating system. Ether once again ran undetected on all of their benchmark programs (which are not malware) while other comparable solutions were detected by popular packers such as Armadillo.