

A Generic Virus Detection Agent on the Internet *

Jieh-Sheng Lee
 Center of Computing Services
 Hua-Fan Institute of Technology
 Shi-Ting, Taipei, TAIW AN
 jasonlee@huafan.hcht.edu.tw

Jieh Hsiang Po-Hao Tsang
 Department of Computer Science
 National Taiwan University
 Taipei, TAIW AN
 {hsiang,chenyi}@csie.ntu.edu.tw

Abstract

The dissemination of software has never been so easy since Internet becomes widely available. This ease of access to free software has also pushed the wide spread of viruses to a new plateau. In this paper we present VICEd, a system for generic virus detection over the Internet.

VICEd is based on a virus detection methodology which is a combination of software emulation and knowledge base. It detects viruses using their behaviour instead of pattern matching. It is thus more effective against unknown or mutated viruses than scanners. This methodology is interesting in its own right.

VICEd is a member of a group of system management agents currently under development at the National Taiwan University.

I. Introduction

Computer viruses have long been a problem for computer users. The emergence of the Internet has increased the significance of the problem since the dissemination of software (along with the viruses) has never been easier. It also changed certain patterns in the spread of viruses. For instance, there were viruses that were running rampant in some part of the world but virtually unknown in the rest of the world. This locality property will no longer be true as Internet becomes widely available around the world.

Techniques for virus detection can be roughly categorized into two types, *scan* and *dynamic traps*. *Scan* is a static method which identifies viruses by matching a certain sequence of instructions with existing virus patterns stored in a data base. A *dynamic trap* is a program which resides in memory and is tightly coupled with the operating system. It monitors programs being executed and looks for suspicious behaviour, such as modifying an executable file illegally.

*Partially supported by a grant from the Trend Micro Devices Inc. Correspondence should be sent to hsiang@csie.ntu.edu.tw.

Scan is the most common method, and the biggest advantage is its speed. But scan is virtually useless against viruses whose patterns are not already in its data base. This problem is particularly keen with the emergence of polymorphic and mutation techniques, where the same virus may have different shapes and forms. Dynamic trapping is more effective against unknown viruses, but it also has its drawbacks. It may affect the performance of the system. If the rules of suspicious behaviour are defined too loosely, the trap may produce many false alarms. The biggest potential danger, however, is that unlike scan which allows easy updates of virus data bases, dynamic trapping is basically a passive device. If a virus is designed specifically for disabling the dynamic trap¹, it would render the system virtually defenseless.

In addition to helping the spread of viruses, Internet poses, to the anti-virus software providers, the additional challenge of how to provide effective anti-virus services over the Internet. Dynamic trapping is obviously not a good solution since it has to reside in a (local) system and runs constantly. Neither is scan an ideal model since, when running a scanner, the user needs to constantly access the data base of virus patterns. One possibility is for the user to download the entire data base of patterns before running the scanner. This defeats the purpose of using the Internet. Another possibility is to streamline the output back to the provider to match the patterns. This might be feasible but it would slow down the speed of scanning considerably. Since the biggest advantage of scan is its speed, this is not an elegant solution because no other advantages are gained.

In this paper we describe a generic virus detection method. Our method employs two main mechanisms, an *emulator* and a *virus analyzer*. The virus analyzer contains a knowledge base of a set of rules, each of which describes the behaviour of a known virus type.

¹Such viruses do exist!

A behaviour rule is expressed as a sequence of *events*. For example, a typical *file-type* virus will first *find* an executable file from the hard disk, *open* it, *write* a portion of its own (already executed) codes into it. Since no normal programs would do such a thing, one may conclude that the target file being inspected is a virus. In other words, the sequence of events *find*, *open*, and *write*, in that order (although not necessarily in procession), defines the behaviour of file-type viruses. We remark that there are over one thousand file-type viruses (and thus over one thousand virus patterns) but only one behaviour rule is needed to characterize all of them.

The events are generated by feeding the target file into the emulator. The emulator emulates the execution of the file and summarizes the execution as a sequence of events, and feed it into the virus analyzer (which contains the knowledge base).

Our method is highly portable and, unlike dynamic trapping, it does not need to reside in a system. Like dynamic trapping, our method detects viruses based on their behaviour rather than relying on pattern matching. Thus it can catch unknown viruses and is effective against mutated or polymorphic viruses. Moreover, since our method emulates the instructions of a program in a simulated environment rather than executing it in the system itself, it can be implemented in any system independent of the operating or file system that the viruses may infect. This provides safeguard against turning the virus loose while trying to identify it. In addition to detecting unknown viruses, it also generates much fewer false alarms than dynamic traps.

The separation of the emulator and the virus analyzer also makes our methodology a good candidate for virus detection over the Internet. The user may download the emulator, run it locally, and send the events back for analysis. The knowledge base, being maintained by the anti-virus provider, can be updated easily by virus experts on the provider's side. If the provider is a commercial vendor, it is also a good working model under which the vendor can charge the users by their usage.

The rest of the paper is organized as follows: In Section II we give a brief overview of computer viruses and existing anti-virus approaches. We present our methodology in Section III, and some experiments are described in Section IV. In Section V we discuss an implementation intended for Internet usage. Finally we give some discussion and future work in Section VI.

II. Overview of computer viruses and anti-virus techniques

For the benefit of the readers who may not be familiar with computer viruses, we give a brief survey of the

field.

A. What is a Computer Virus

There are different definitions of what a computer virus is. We adopt one which is more consistent with the definition of biological viruses [2]: A *computer virus* is a program which can spread itself by attaching its execution codes to other programs or redirect the control of execution to itself first and then pass the control to other companion programs. Under this definition, the prime directives of a virus are to *spread* and to *survive*. A "good" computer virus will try to keep itself from being detected and, in the meantime, try to spread as much as possible. There are other types of malicious programs, such as trojan horses, which can be extremely damaging to the user infected, but they are not considered as viruses (or good viruses) if they are not capable of spreading. The main objective of an anti-virus software is usually aimed at detecting "good" viruses. Therefore anti-virus programs are usually not very effective in dealing with other types of malicious programs.

B. Categorization of Viruses

Until about one year ago, all known computer viruses are executable files, which will not infect unless they are executed. About a year ago, the so-called "macro viruses" appeared. A macro virus hides itself in the user-defined macros of a data file. When such a file is loaded into the intended application software (such as a text editor), the virus macros will overwrite the predefined macros and, subsequently, infect all data files loaded henceforth. As of today, there are only a few dozens of macro viruses exist, most of which are targeted at the popular software Microsoft Word. Since data files are quite different from executable files, the emergence of macro viruses has caught the anti-virus software community rather off guard. And as far as we know, there have not been any effective, general methods for dealing with them. However, since over 99% of computer viruses are still executable-type viruses, In the rest of the paper, we shall focus only on executable-type viruses.

We divide (executable-type) viruses into two groups, the *basic* viruses and the *polymorphic* viruses. The basic viruses can be further categorized roughly into three groups, *file-type*, *memory-type* and *boot-type*, although there are variations.

File-type Viruses

A *file-type* virus is a virus that is able to infect other files when and only when it is executed. When one executes a program that carries a file-type virus, the virus will first obtain control of the execution. It will

then search through target files, try to infect them, then finally recover the host file and pass the control to it. Afterward, the host file will execute normally as before.

Memory-type Viruses

A *memory-type* virus is a virus that resides in memory. Once a user executes a file infected by a memory-type virus, the virus will reside in the system and wait for the chance to infect other files or attack the system. Usually memory-type viruses are more troublesome than file-type ones, since they can stay in memory and infect all files that are executed. After a virus resides in memory, it is possible that even anti-virus tools will fail to detect the virus because it may filter normal file access and provide fake data.

Boot-type Viruses

A *boot-type* virus infects the boot sectors of a disk and activates itself when the disk is used to boot the system. During the booting process, a boot-type virus will get the control first. When the illegal operations of the viruses are completed, the virus restores the original data of the boot sectors and transfers the control of execution to the entry point of the booting procedures. In most cases, this type of viruses are also memory-type viruses. They will try to reside in memory during the booting process and wait to infect other target files later. Detecting this kind of viruses means analyzing the boot sectors of a disk.

Polymorphic Viruses

A few years ago virus writers invented a clever way to mass-produce viruses and to defeat the increasingly sophisticated scan tools. The basic idea is to use simple encryption to encrypt the body of the viruses so that when infecting, the code of the virus will look different each time. This renders the pattern-matching driven scan tools virtually useless since now a virus may appear in thousands of different forms. This kind of viruses are called *polymorphic* viruses. Moreover, the encryption routine used in different host files may vary. Sophisticated polymorphic viruses can even change the sequence of independent instructions and insert meaningless instructions, such as NOP, into the codes before spreading. One of the most famous program is the MtE "Mutation Engine" written by a Bulgarian virus writer code-named "Dark Avenger". He distributed the object codes of his program which is ready for other virus writers to use. By simply including his engine, virus writers can turn their simple viruses into polymorphic ones. Currently there are quite a few widely distributed mutation engines in the world.

C. Anti-Virus Techniques

We now briefly introduce some of the most popular anti-virus techniques.

Checksum

Checksum is one of the first and readily available methods to detect viruses. The checksum program checks whether a file has been modified. Since viruses must modify files, checksum has also been used to detect viruses. An obvious problem of checksum is that it may produce an excessive amount of false alarms [5], since obviously not all modified files contain viruses. Later improvements such as checking only the critical part of a program make this methodology work better, but virus writers invented the so-called *stealth* viruses to defeat checksum programs. A stealth virus redirects certain DOS² interrupt service routines (mainly file I/O routines) to routines of its own. Then when a checksum program tries to check it, the virus fools the program by temporarily restoring the original data [1], [6].

Vaccine

Vaccine was at one time a popular practice against viruses. Vaccine is actually a metamorphosis of computer viruses. It tries to gain control at the first moment of execution to check if there is anything wrong with the host file, such as the file size. If a virus tries to append itself to the host file like the vaccine has done, then the injected vaccine will take control and be able to detect the virus. The vaccine can also restore the host file and truncate the virus body with the aid of critical information previously saved in vaccine. By this way a vaccine can also clean the virus automatically. However, similar to checksum, vaccine is not effective against stealth viruses. Furthermore, most users may not wish to inject thousands of files with a vaccine.

Scan (Pattern Matching)

Pattern matching technique is the most commonly used among existing anti-virus tools. It is featured in all available commercial anti-virus software. A *pattern* (or *signature*) of a virus is a sequence of consecutive binary codes, which can be used to identify a virus and, sometimes, its variations. Since the location of the matched pattern may not be a constant value with respect to the beginning of the host file, a scan tool needs to decide the entrance of a virus body. Once a scan tool knows where the virus body starts, it can apply pattern matching to discern different viruses effectively. Scan is not effective for detecting mutated and polymorphic viruses. The instructions at the starting point of the body of such a virus cannot be decided because they are decrypting routines with different permutation and combination of instructions. Some anti-virus companies claim they are developing generic decryptors to

²Since most of viruses are written for IBM-compatible PC, in the rest of the paper we only talk about viruses for PC, although the viruses and the anti-virus methods we describe all apply to other systems.

decrypt the encrypted codes to get to the original virus body, then perform pattern matching. Indeed, there are scan tools which seem effective in deciding *some* mutated viruses. However, the methods behind them are carefully guarded trade secrets, and it is hard to find documents describing them. Furthermore, such generic decryptors usually slow down the scan tools considerably.

Dynamic Trap

A trap tool is usually a program that resides in memory. It usually requires the user to execute the trap tool when bootstrapping the computer. The user can execute it under DOS prompt or let it start in the DOS config.sys file, working as a device driver. To give the user a higher level of security, one should wake up a trap tool as early as possible, such as at the booting phase of a computer. Once a trap tool is hooked into the system, it monitors any suspicious activities related to virus behaviour, such as writing boot sectors illegally, appending to an executable file, etc. If the trap tool is designed properly, it should be able to protect the system at the critical moment when viruses attack. However, there have been viruses designed specifically with the purpose of intruding and disabling trap tools. This flaw of trap tools is mainly due to the lack of system protection of DOS. In order not to bypass potential viruses, trap tools may also produce false alarms when suspicious operations happen.

Heuristic Scan

Heuristic scans are a new technique aimed at compensating the shortcomings of the traditional scan tools in detecting unknown viruses. Heuristic scan detects viruses by their operations, not by their patterns. The idea is to specify certain consecutive codes, which represent suspicious operations, as meaningful alarms. For example, the binary code "E8 00 00" is a call instruction which calls into the following instruction. This seemingly innocuous instruction has been used by virus writers as a common trick to let the virus locate itself in order to deal with the different sizes of the host files. Normal programs will neither contain nor execute such an instruction. Since instructions as such differ a virus from a normal program, a heuristic scanner can take the presence of such an instruction as a sure sign that the file contains a virus. Thus, the developers of heuristic scanners study viruses to identify such instructions and their combinations to include them in their scanners.

An immediate problem is that the fine line between a normal and an abnormal program is often not so clear cut if one only looks at individual instructions. In order to reduce the number of false alarms, heuristic scan tools often need to incorporate an additional table to exclude some popular softwares in order not to cause

too many false alarms.

Another problem is the rigidity of looking for specific instructions. For instance, a heuristic scan may interpret the sequence

```
MOV AH, 40
```

```
INT 21
```

(write interrupt service routine under DOS) as an alarm for a virus. However, if one disguises the same introductions as

```
MOV AH, 3F
```

```
ADD AH, 1
```

```
INT 21
```

then the heuristic scan may be fooled.

III. The Virus Instruction Code Emulation (VICE) Methodology

A. Motivation

The combination of scan and trap tools is the most common weapon against viruses among casual computer users. However, there are drawbacks. A scan tool cannot detect viruses whose patterns are not in its data base. A trap tool gives out an alarm when some behaviour that it perceives as abnormal occurs. When this happens, a user usually invokes her scan tools to check whether it is a virus or not. If the scan tools fail to confirm, a casual user may decide that it is yet another false alarm and ignores the warning. Since dynamic traps are prone to false alarms, this scenario is not uncommon. However, if the alarm was indeed caused by a virus whose pattern is not in the data bases of the scanners, then the integrity of the system is compromised.

Another intuition behind the invention of traps is the following: The most obvious way to decide whether a file contains a virus is to execute it and see if it infects some files or destroy the system. Since this is obviously a dangerous thing to do, a trap passively waits for a virus to activate itself and tries to detect it before it does any damage. This approach works as long as either the user is patient and knowledgeable enough to carefully analyze all alarms or the virus is not designed to evade (or even disable) the trap.

In this section we present a new methodology, *Virus Instruction Code Emulation* (VICE), for generic virus detection based on the behaviour of viruses. VICE was originally motivated by the challenge to come up with a more effective methodology than scan to decide if an alarm reported by a dynamic trap is indeed caused by a virus. It can be used as a stand-alone anti-virus software although it can also be combined with a scan and a trap.

Our method proposes an alternative to executing a potential virus on the system by using an emulator.

That is, we emulate the execution of the target file in a virtual system environment. There is, therefore, no chance for a virus to infiltrate the system like it may happen with trap tools. The emulator summarizes the emulated instructions into a sequence of *events*, which are fed into a *virus analyzer*. The virus analyzer has a knowledge base of rules, each of which describes a known virus behaviour. The virus analyzer analyzes the sequence of events that it receives from the emulator, and decides whether it constitutes the behaviour of a known virus.

The idea of using a knowledge base of virus behaviour comes from the experience of observing how virus experts analyze viruses and learn new behaviour of viruses. A virus expert traces the instructions of the codes, collects critical system information, and decides whether what she has seen is a virus without actually executing the codes. In other words, a virus expert detects viruses from their behaviour. When she encounters a new virus behaviour for the first time, it may take her a fair amount of time to trace and test the target file. But once the virus expert deduces a behaviour pattern from the sequence of operations, she can detect the same type of viruses rather easily next time. She can also share the new knowledge with others by describing the behaviour of the new virus. All these observations lead us to the incorporation of VICE with a knowledge-based structure in which one can incorporate expert knowledge of virus behaviour. This knowledge-base approach makes VICE more flexible than scan or heuristic scan tools since one does not need to maintain a comprehensive data base of patterns.

B. Components of VICE

One can view VICE as two parts: an emulator and an analyzer, or more specifically speaking, a *multi-path emulator* and a *knowledge-based virus analyzer*. The emulator emulates the execution of a target file in a virtual system environment, and send events that it collects to the analyzer. The analyzer is responsible for collecting incoming events and, using the knowledge base, deciding whether the events exhibit virus behaviour. If the sequence of received events satisfies a rule in the knowledge base, the analyzer explains why the target file is a virus and what type of virus it is. In the meantime it tells the emulator to stop generating events. Otherwise the emulator will proceed until the time limit is reached. In order to detect latent viruses (viruses that are dormant most of the time), the emulator may need to explore other possible execution paths. This explains the need of a multi-path emulator, which will be explained in more detail later.

In the following sections, we explain the individual

components and their internal structures in more detail.

C. Software Emulator

The software emulator of VICE creates a virtual system environment and emulates the instructions of the target file in that environment. In the virtual system, VICE acts like a CPU. After the target file is loaded into virtual memory, VICE first fetches some bytes from the target file. It then decodes an instruction and tries to execute it. All operations are performed in the virtual system environment, where there are virtual registers, virtual memory space, virtual interrupt service routines, etc. When the instructions from the target file are emulated, the values of the internal registers of VICE will be the same as the values of the physical registers as if the codes were actually executed. If an instruction tries to access data at some memory location, VICE will also manage its virtual memory to access or put data. If the introduction issues an interrupt service routine, VICE will provide one of its own. However, the emulated interrupt service routines do not execute any real system interrupt operations; they merely return appropriate register values.

C.1 Multi-Path Emulation

Straightforward emulation alone is not sufficient for analyzing viruses. The reason is that some viruses are designed to spread only when certain conditions, such as a specific date or OS version, are met. We call them *latent viruses*. An emulator may not detect a latent virus if the parameters of the system environment are not identical to the requirements of the virus. We take, as an example, a variation of a well-known latent virus ATOM.COM. ATOMIC.COM will spread only on the first day of each month. If one executes a file that carries ATOMIC.COM on any other day of the month, the virus will not activate itself to infect other files. One may envision a naive solution, which simply alters the values of the variables or registers so that to force the CPU to execute another path. For instance, one can set the date to be 1 June to force ATOMIC.COM to activate. This approach is not feasible because it is not easy to find out the exact data dependency relation in a virus.

To solve the problem of latent behaviour of viruses, we have used a multi-path emulation approach. During the execution of a program, the CPU may encounter many branching points where conditional jump instructions are executed. Under a normal condition, if the flag conditions satisfy, the CPU will execute the instruction whose location is assigned in the conditional jump as the next instruction. If the flags are not satis-

fied, the CPU will execute the instruction that follows the conditional jump. A multi-path emulator is capable of emulating all executable paths, even if the paths may not be executed under current system conditions. This way we can detect latent viruses such as ATOM.COM.

In our multi-path emulation, we record all "critical" branching points during an execution. If a branching point is deemed important, after one destination is explored and no virus behaviour found, the program will be rerun and be forced to jump to the other destination. We remark that we do not use backtracking when traversing to the other destination. This is because emulation keeps record of all of its registers, and using backtracking involves too much overhead and does not seem worth the effort.

To make multi-path emulation run faster, we assign different priorities to different branching points according to their closest interrupt service routines. For instance, if a branching point follows the GET DATE service routine, it will have higher priority than one that follows the GET DOS VERSION service routine. Then when the next fresh run starts, the unexplored path according to the first branching point will be emulated first. The decisions of priorities are made according to past experience with the behaviour of viruses.

Although multi-path emulation is a powerful technique, it may have drastic impact on efficiency if not used carefully. This is because there may be many branching points when emulating the execution of a file. If every branching point is marked and traversed, then the amount of time needed to test all branches may be staggering, in particular if the target file is not a virus. To overcome this problem, we incorporated two time-out parameters for VICE to terminate, in addition to providing various measures for selecting branching points. They are *total time-out* and *one-path time-out*. Total time-out specifies the total time that VICE can consume, and one-path time-out specifies how much time VICE can spend on a single executable path. We shall discuss in Section III-F why the time-consumption of multi-path emulation is not a serious problem for the purpose of virus detection.

D. Virus Analyzer

As mentioned before, our virus analyzer is inspired by how virus experts analyze files and detect viruses. An expert does not always trace every detail of the virus codes before deciding that it is indeed a virus. She *knows*, from experience, the difference of behaviour or operations between a normal program and virus. Take a file-type virus as an example. A file-type virus will try to find a file, open the file, and write the file with its code, all on its own. A normal program will never

perform these operations without prompting a message to the user to ask first. We can therefore take such behaviour as a sign for a virus. For memory-type viruses, the line between a normal program and a virus is even more clear-cut. Since a memory-type virus will try to write a target file to spread, there will be a write routine in its resident memory, and it will try to reside in memory by hooking itself to some DOS interrupt service routine. On the other hand, it is highly irregular for a normal TSR³ program to use file I/O routines without asking the user first. Thus, a virus expert knows that the target file is a memory-type virus if she finds some file I/O routine in the hooked memory.

The virus analyzer in VICE is based on the same idea. It first receives events that come from the emulation of the target program. It then uses an *event filter* to judge if a received event is critical or not. If the event is irrelevant, such as screen output routines, the event filter will simply discard it. If the event is a critical one, the event filter will send it to the component of *rule matcher*. The rule matcher collects received events and matches them with the build-in virus behaviour rules in the knowledge base. Upon successfully matching a rule with an incoming event, it will notify the successful detection of a virus and send a stop signal back to the emulator. Then VICE will terminate.

With the architecture of the knowledge base and rule matching, VICE resembles a human virus expert: VICE has a knowledge base of rules describing different behaviour of viruses, and it analyzes viruses by emulating their instructions, collecting execution information (the *events*), and judge viruses by their behaviour. A virus expert, in comparison, usually traces a virus with some debugging tool, watches the change of registers, reasons about the intention of instructions and interrupt service routines, links various actions into meaningful behaviour, then finally decides whether it is a virus by its behaviour.

E. The Knowledge Base

In this section we explain the structure of the knowledge base in detail. A knowledge base is composed of rules, which are sequences of (not necessarily consecutive) events. An *event* is a call to an emulated interrupt service routine or a system state defined in VICE.

The following are some examples of events, where $E(n)$ is the number, the statement in quotes is an explanation of its actions, and what in between is the activity that the event captures.

$E(1)$: Hooked(21) , "INT 21 hooked." ;

³Stands for *Terminate-and-Stay-Resident*, a common term for resident programs in DOS.

```
E(3)  : 21 , AH = 40 , "Write a file" ;
E(5)  : 21 , AH = 3D , "Open a file" ;
E(19) : 21 , AH = 4E , "Find first" ;
```

Here are some typical rules:

```
R(1) : E(19) , E(5) , E(3) ,
      "This program tried to find a file,
      "open the file, and write the file."
      "      It is a file-type virus."
R(2) : E(1) , E(3) ,
      "This program hooked INT 21 and
      "tried to write a file."
      "      It is a memory-type virus."
```

Let us start from the first rule. R(1) specifies that a sequence of event 19, event 5 and event 3 constitutes a file-type virus. Informally, event 19 is a find-first-file function in DOS interrupt service routines. Its interrupt number is 21 (in hex) and the parameter register AH is 4E (hex). Similar to the interrupt service routines of DOS, one can define these register information for events in the format listed above. Event 5 defines an open function and event 3 defines a write function. Any file performing find-file, open-file, write-file functions in a default range at the beginning of execution codes, without any prompt, is a most likely a virus.

The first event of R(2) is "INT 21 hooked". This indicates that all the following events will occur in the hooked memory of INT 21. The second event, E(3), indicates that a write function is executed in hooked INT 21, and leads to the conclusion that the file that hooks INT 21 is a memory-type virus.

While a pattern (signature) of a specific virus is usually a sequence of codes incomprehensible to human, a behaviour rule is represented just as we described, which is easy to understand and to modify. Indeed, our knowledge base is a plain text file in which events and behaviour rules can be added and deleted easily.

F. Why VICE Works

Software emulation is not a new idea. Indeed, it is a well-known technique in software engineering for reverse engineering at the code level. However, software emulation has not been a useful technology for most of the software engineering applications. This is mainly due to the enormous number of branching points that comes with the execution of a typical program. The complexity needed for emulating all possible execution paths caused by the branching points made software emulation virtually impossible. Virus detection, interestingly, turns out to be an ideal area of application for software emulation. The reasons are twofold. When a program is executed the first priority of a virus is to

gain control of the execution. Thus, one does not need to look beyond the first few thousand instructions for virus behaviour. A virus is also small. This is necessary for the virus to spread effectively and without notice.

These two properties means that one almost never need to look beyond the first few thousand instructions to decide if a file contains a virus. This greatly reduces the complexity as far as branching is concerned, and make effective emulation possible. By taking advantage of these features, software emulation also provides the possible additional benefit of virus pattern generation and virus cleaning. We briefly describe the reasons and method here. A virus usually transfers the control of the execution back to the original program after it accomplishes its goal. Since the emulator executes every relevant instruction, it can always detect the "return control" point. Thus, it can replace the portion of infected program with the correct binary image in memory, and truncate the attached malicious codes, as long as the virus is not destructive⁴. For essentially the same reason, the emulator can extract the sequence of instructions which forms the signature of the virus. The signature can then be added to the virus pattern data base of the accompanied scanner.

G. Unique Features of VICE

VICE has a few unique features which are not common among an anti-virus methods.

Detect Unknown Viruses

Since VICE detects viruses based on their behaviour instead of patterns, it is much more effective in detecting unknown viruses than other anti-virus approaches. Theoretical speaking, VICE can catch all known and unknown viruses as long as their behaviour is documented.

Detect Polymorphic Viruses

Polymorphic viruses are no more than existing viruses transformed via a mutation engine. Since VICE emulates the instructions of the virus, it will decrypt the virus the same way the virus would decrypt itself. Thus, as long as the original (non-encrypted) version can be detected by VICE, so can all of its polymorphs.

Generate New Patterns

Since VICE knows, through emulation, the entry point to the body of the virus, it can extract some execution codes as the signature for the virus. This information can be utilized by scanners for future scans. We should mention that since each scanner defines its own convention and algorithm for extracting and representing signatures, the same virus may have different

⁴A destructive virus *overwrites* the host code by itself. In this case, it is virtually impossible to recover.

signatures in different scanners' data bases. Thus, for VICE to generate signatures for a specific scanner, it has to know the algorithm used by that scanner.

Clean Viruses

A *destructive* virus is one that overwrites part of its host file. A non-destructive virus, on the other hand, would normally attach itself to the end of a normal program. An infected file can be cleansed of its parasitic virus only if the virus is not destructive. Since VICE knows the entry point to the virus body, it can truncate the virus code and clean the host file of its parasite if the virus is non-destructive. As for destructive viruses, they usually cannot travel too far since they would interrupt the execution of a normal program and are, therefore, easier to detect.

IV. Experimental results

There are two implementations of VICE, one on PC and one on Unix, although both are for detecting viruses of PC. While the PC version is mainly written in C with about 5% in assembly language, the version for Unix is written entirely in C. Currently there are 24 events and 18 virus behaviour rules in the knowledge base.

VICE has been used extensively in real world situations, and has been successful in detecting unknown viruses in industrial applications. In the following we describe some experiments which we have conducted. The tests are done on an 80486 DX2-66. The setting for the multi-path emulation are 12 seconds for one-path and 60 seconds for all paths.

A. Testing virus data bases

We have tested VICE on a virus bank taken from the CD-title "The Collection - Outlaws from America's Wild West", issued by American Eagle Publications, Inc. This CD-title is a rich source of materials created by virus writers. The virus bank contains 3562 viruses and, according to the readme file in the directory, all are live viruses (capable of spreading).

The test results are given in Table 1. We remark that although VICE did not detect all viruses, it caught almost 75% with only 28 behaviour rules. In order to detect more viruses, we need to extract more knowledge from virus experts and transform them into new rules for VICE. Some of the existing scan tools may be able to detect more viruses from the same data bank, because the viruses in the virus bank are some of the most popular ones (and therefore are targets of commercial anti-virus vendors). However, a scan tool can catch only as many viruses as its data base has patterns. A minor variation of any one of the viruses will not be caught since the pattern will be different. In

order to sustain the same hit ratio, a scan tool needs to constantly update its data base. VICE, on the other hand, can catch any variation of the viruses that it can already catch, without the need of modifying its knowledge base.

B. Testing Normal Programs

In addition to virus files, we have also used "normal" files to test the amount of false alarms, which would also place an indication on the accuracy of our behaviour rules. The test files we used include executable files in DOS and Windows3.1 as well as randomly chosen files (indicated as "User A") from the hard disk of a PC.

The test results are given in Table 2. Note that there was almost no false alarm, which indicates that our behaviour rules are very accurate. Furthermore, we do not need to use any additional tables to exclude common software as in heuristic scan tools.

V. VICEd: Integrating VICE into the Internet

In this section, we describe how to integrate VICE into the Internet.

An inherent limitation of emulation is that it can never be as fast as a purely syntactic method such as scan. For a casual user, who cares mainly about how much time it takes to scan her hard disk, VICE is not suitable as the first line of defense against virus. For this reason we envision VICE, for an individual user, be used in two ways. First it can be used in combination with a scan and a trap. That is, a user uses her scan tool to detect known viruses. Then invokes the trap to reside in memory to detect any potential problems. When an alarm occurs, she sends the suspected file to VICE for detailed analysis. A second mode of operation is to use VICE as an "off-line" tool. That is, run VICE through the file system during off hours (similar to Mr. Clean of BSD Unix) to see if any files which are new or modified have been infected.

Extending the second idea slightly further, we can immediately see how VICE can be used in the client/server computing model. To be more precise, one can have VICE running on the server, which checks the file systems of its clients off-line. The fact that VICE runs on any machines that support the necessary emulation means that the server can even be a UNIX machine, since VICE can indeed run on UNIX. Looking for DOS viruses on a UNIX machine obviously provides maximum assurance as far as breaching system integrity is concerned.

We apply similar concept in adapting VICE to the Internet. The main difference between VICEd, the Internet version of VICE, and VICE is a physical separation between the emulator and the virus analyzer when

Virus Bank					
number of files	result	percentage	average time (sec)	max time (sec)	min time (sec)
3562					
2654	detected	74.51%	3.2	57	0.1
908	undetected	25.49%	26.2	61	0.1

TABLE I
TESTING VIRUS DATA BANK

Normal Programs					
source	number of files	falsely reported as virus	average time (sec)	max time (sec)	min time (sec)
DOS	65	0	23.4	61	0.1
Windows3.1	69	0	6.6	61	0.1
User A	2241	1	37.3	61	0.1

TABLE II
TESTING NORMAL PROGRAMS

VICEd is executed. Suppose VICEd resides at the site of the *provider* and the person who wishes to use VICEd is the *user*. Then the user, when using VICEd, only retrieves the emulator part from the provider over the Internet. Then she runs the emulator on the files that she wishes to be analyzed. The sequence of events produced by the emulator are then sent back to the virus analyzer at the provider's end for automatic analysis. The results of the analysis, together with possible solutions, are sent back to the user.

The physical separation of the emulator and the virus analyzer has benefit for both the provider and the user. From the provider's side, it gives her control over the usage of the software as well as being more effective in updating the knowledge base. If the provider decides to charge a fee for usage, it can be done since the user needs the information from the knowledge in order to make sense of out the outcome of the emulation.

There are also advantages to the user's side. First of all, the user always receives, implicitly, most updated service from the anti-virus community. Every user of anti-virus software knows the headache of the need to constantly update the pattern data base of the scanner. Since the knowledge base of VICEd is maintained by the provider and is accessed each time VICEd is used, the user is always using the most recent rules. Secondly, in the business world it is not uncommon for a corporation to suspect whether their important software has been compromised. However, since the software may involve company secret, the company may not want to

send it to an anti-virus provider to be analyzed. With VICEd, the company only needs to release events from the first few thousand instructions which are usually quite innocuous as far as confidential information is concerned.

We have implemented VICEd on SUN workstations (running SUNOS 4.1.x or Solaris 2.x) with an 8088 emulator written entirely in C. Our implementation follows the client/server model, where the client is the emulator at the user's end, and the server is the virus analyzer of the provider. Each time a client starts to execute, it initiates a peer-to-peer communication with the server by sending a request. We use the TCP protocol to handle packet loss and out-of-order delivery problems. We have also implemented an encryption algorithm to encrypt the events generated from the emulator before sending them to the server. This adds additional security and prevents the events sent from being compromised.

The working scenario goes as follows: The user retrieves the emulator by clicking an icon on the browser of the homepage of the provider, downloads the emulator, then runs the emulator on either a file or a directory of files. The emulator initiates communication with the server, waits for the go-ahead and starts emulating. It emulates for a few (preset) seconds before sending the first batch of (encrypted) sequence of events to the server. The analyzer at the server's end decrypts the events and checks to see if the received events match one of the virus behaviour rules. If they

do, then it sends a report to the client to report the virus and tell the client to stop emulating. Otherwise it asks the client to send more events. This continues until the client exceeds its emulation time-limit.

VI. Discussion and future work

VICEd is a member of a group of system management agents currently being developed at the National Taiwan University. The purpose of this group of agents is to provide users with security, convenience in operations and useful on-line help. VICEd is designed to be part of the security agents, and plays the role of safeguard against viruses. The security agents, when fully implemented, will serve as a "personal software firewall" against different types of intrusion and malicious attacks.

For an individual or a group of users sharing the same file system, we envision VICEd to traverse through the file system during the off hours looking for viruses. After this is done for the first time, the follow-up traversals will take much less time since very few users would alter her files *en masse* in one day. (In fact, if a large number of executable files have been altered in the same day, it is a clear sign that a file-type virus might have invaded the system. So an initial simple traversal of checking whether a lot of files of the same user have been modified is a simple way of deciding whether a virus has been running rampant.)

There are a number of directions that we are planning to investigate. The first is the possibility of implementing the emulator in a mobile language such as Java. There are several advantages for doing this. First, it provides additional portability since VICEd can then run on any machine which supports the (mobile) language. Second, the user need not download the emulator at all, since it can now be executed from the browser directly.

The main obstacle in carrying out this program using Java is that currently Java does not allow its applets to access local hard disk. This will, no doubt, limit the usability of a Java-based VICEd considerably. We think that eventually Java has to allow some flexibility in in this regard, since otherwise it will defeat the purpose of having a mobile language.

Another interesting issue is the question of whether one can develop Java viruses. The Java virtual machine (JVM) is a simple stack machine with a simple instruction set and a few registers. The architecture is elegant and the file format of executable applets is not complicated. If it is possible to append some code to an executable applet, there is a high possibility that a pure Java virus may emerge soon. Since applets travel far and wide through the Internet, such viruses may

cause tremendous damage to the computing community. Since a Java virus does not need to violate the rules of the instruction set in a JVM, and that it does not need to overwrite any system library, the security mechanism of JVM will not be able to trap such a virus.

The VICE methodology would be the most ideal defense weapon in such a case. First of all, since the JVM is already available, one doesn't even need to write the emulator. All that need to be done are to translate the emulation results from JVM into events, and define Java virus behaviour rules, then a *VICE-for-Java* will be ready to work.

At this point we should point out that the VICE technology is a general virus detection methodology which works on any operating system or machine instruction sets as long as the suitable emulator is implemented.

The last question which we wish to address is the question of *learning*. It would be interesting if there is some mechanism to automatically deduce new virus behaviour rules. We are currently investigating potentially applicable methods.

References

- [1] Vesselin Bontchev, "Possible Virus Attacks Against Integrity Programs And How To Prevent Them", Proc. 2nd Int. Virus Bulletin Conference, September 1992, pp. 131-141.
- [2] Frederick B. Cohen and Sanjay Mishra, "Some Initial Results From the QUT Virus Research Network", Proc. 2nd Int. Virus Bulletin Conference, September 1992, pp. 15-25.
- [3] Frederick B. Cohen, *Protection and Security on the Information Superhighway*, John Wiley and Sons, 1995.
- [4] David Ferbrache, "A Pathology of Computer Viruses", Springer-Verlag, 1991
- [5] Alan Solomon, "False Alarms", Virus News International, February 1993, pp. 50-52.
- [6] Alan Solomon, "Mechanisms of Stealth", Proc. 5th Int. Comp. Virus and Sec. Conference, New York, March 1992, pp. 232-238.
- [7] VIRUS-L FAQ (Frequently Asked Questions) list, March 1996
- [8] News Group : comp.virus