2-8 Detecting Unknown Computer Viruses — A New Approach —

MORI Akira, SAWADA Toshimi, IZUMIDA Tomonori, and INOUE Tadashi

We give an overview of a tool detect computer viruses without relying on "pattern files" that contain "signatures" of previously captured viruses. The system combines static code analysis with code simulation to identify malicious behaviors commonly found in computer viruses such as mass mailing, file infection, and registry overwrite. These prohibited behaviors are defined separately as security policies at the level of API library function calls in a state-transition like language. The current tool targets at Win32 binary viruses on Intel IA32 architectures and experiments show that they can detect most email viruses that had spread in the wild in recent years.

Keywords
Unknown computer viruses, Static code analysis, Code simulation, Security policy, Virtual runtime environment

1 Introduction

Since today's society is more and more dependent on computer networks—in personal communications, corporate activities, and the overall infrastructure—we are seeing an increasing risk of serious damage from computer viruses. Antivirus programs are now installed in computers as a matter of course; however, conventional antivirus software relies on pattern matching based on a database of known viruses and therefore is ineffective in detecting unknown viruses.

The increasing sophistication of viruses has led to higher risk of infection, and a more frequent emergence of virus variants has increased the efforts in pattern extraction to cope with such variants. Since these infections can cause immediate and serious damage, demand is high for accurate detection technology that can prevent this damage by detecting unknown viruses.

Accordingly, we have been working to develop the technology to detect unknown computer viruses in Win32 executable file format (a standard program format running on Windows platforms) that are activated on an x86 processor. Specifically, we are developing a tool to detect characteristic viral behavior, such as file infection and mass mailing, through analysis of API function calls. The analytical process consists of decryption by code simulation and static code analysis. Defining anticipated virus behavior in the form of policies will allow for the detection of unknown viruses without relying on pattern definitions (Fig.1).

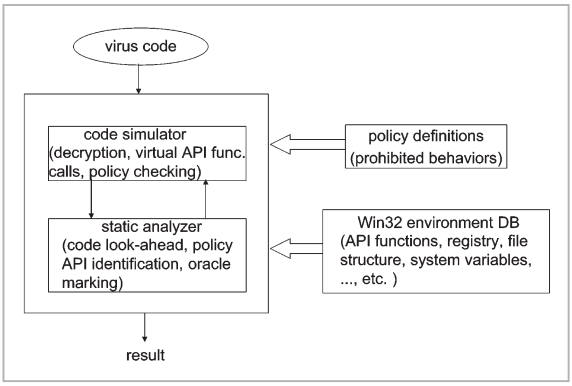

Fig.1 *Scheme to detect unknown viruses*

Through various experiments we have confirmed that this new tool can detect most of the viruses (unknown in terms of this tool) that have proliferated in recent years. We analyzed virus samples, defined and implemented policies based on the results, checked for misidentification of safe programs, and conducted detection experiments. As we repeatedly performed this procedure, we discovered techniques used by many recent viruses to circumvent antivirus software.

In this paper, we will describe how this tool works, typical techniques to circumvent antivirus software, and the ways the tool can respond to these techniques.

In this study, we focused on viruses in Win32 executable file format and activated on an x86 processor, as these are the most common viruses and the most difficult to detect. The results of this research can also be applied to other processors and operating systems.

2 Conventional antivirus technology

Many commercially available antivirus programs apply a detection system based on the "pattern (signature) matching" or "scanner" method. This system extracts certain binary code segments from known viruses, enters them into a database in the form of hexadecimal strings (called "patterns" or "signatures"), and matches files against this database to determine whether they are viruses. Generally, this system has the following disadvantages:

- The system cannot detect unknown viruses whose patterns are not contained in its database.
- It is difficult to create patterns that can uniquely characterize viruses and prevent safe files from being misidentified as viruses.[1]
- Existing patterns are rendered inapplicable to matching simply with partial modificationof the virus code (as seen in numerous virus variants)—in an extreme example, this can be accomplished merely through recompilation of the code with a different compiler.

In addition to matching of simple string patterns, antivirus vendors are now developing more common patterns that can include regular expressions instead of simple character strings, as well as pattern matching using file or program structures. However, these matching methods essentially rely on syntactic information and are thus fundamentally limited.

To detect unknown viruses, some antivirus programs apply the "dynamic protection" process, in which suspicious executable files are run and observed on an isolated computer to determine whether they are indeed viruses. However, this method relies on actual observed program functions and may not be able to reliably detect viruses that do damage only under specific conditions (e.g., on a specific date and at a designated time). The "heuristic scan" method, on the other hand, uses common patterns to detect specific program structures, yet with this method it is considered more likely that useful programs will be misidentified as viruses.

1 According to one antivirus software vendor, more than 20,000 types of viruses, including variants, exist.

3 Self-encrypting and polymorphic viruses

To detect both known and unknown viruses effectively, we must be able to combat viruses that are capable of self-encryption and polymorphism. Self-encrypting and polymorphic viruses were originally devised to circumvent pattern-matching detection by preventing the virus generating a pattern. Unknown viruses applying this technique are even more difficult to detect.

A self-encrypting virus consists of an encrypted payload and code for decryption once in memory (Figure 2 shows how this virus is activated). Since the virus code is encrypted, the behavior of the active virus cannot be determined by program code checking. Moreover, patterns can only be deter-

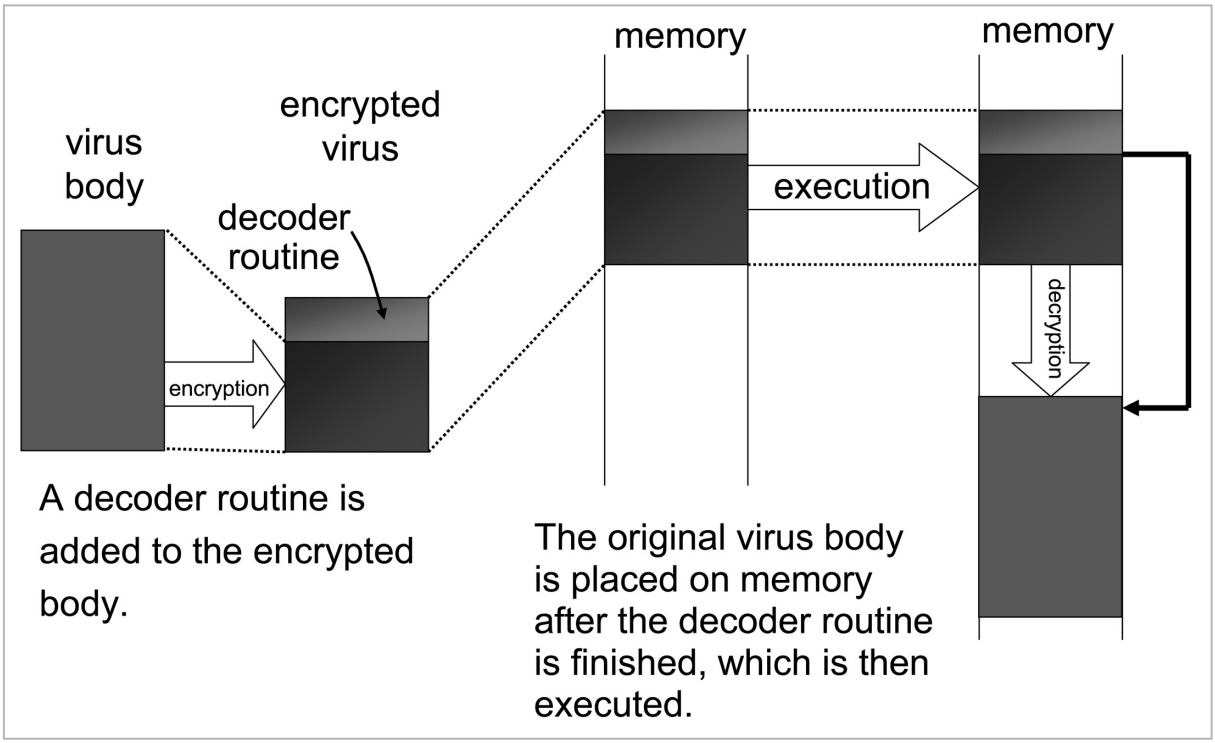

Fig.2 *Self-encrypting virus*

mined from the unencrypted segment (the decryption code), impeding pattern matching even further.

As an enhanced version of the self-encrypting virus, a polymorphic virus is designed to avoid any fixed pattern. This virus attempts to apply a different decryption code each time it is activated, through changes to its encryption method (Fig.3). In practice, however, it is extremely difficult to create a virus with no fixed pattern, as there is a limit to the number of available encryption algorithms. Nevertheless, virus authors attempt to create as many variants as possible through network downloads of encryption algorithms. Detecting polymorphic viruses with conventional pattern matching entails the preparation of a considerable amount of complex patterns, significantly reducing processing efficiency.

4 Methods for detecting unknown viruses

There are two main types of methods to determine whether an unknown executable program is a virus:

- Running the program in an isolated environment to determine whether it does damage (the "dynamic protection" described in Section **2** above)
- Analysis of the program to identify segments that result in malicious behavior (static analysis)

We selected the latter method as the focus of this research. This is due to the technical limitations of the dynamic method as described above, and also to the fact that constant monitoring on every computer is necessary to prevent virus damage.

To develop a detection method based on static analysis, we must overcome a number of challenges, as follows.

(1) Measures against self-encrypting and polymorphic viruses

As described above, most viruses today feature a mechanism for self-encryption to circumvent simple file checking or detection by pattern matching. There are only two ways to detect such viruses when these are unrecognized: by cracking the encrypted code or by allowing the viruses to activate decryption on their own.

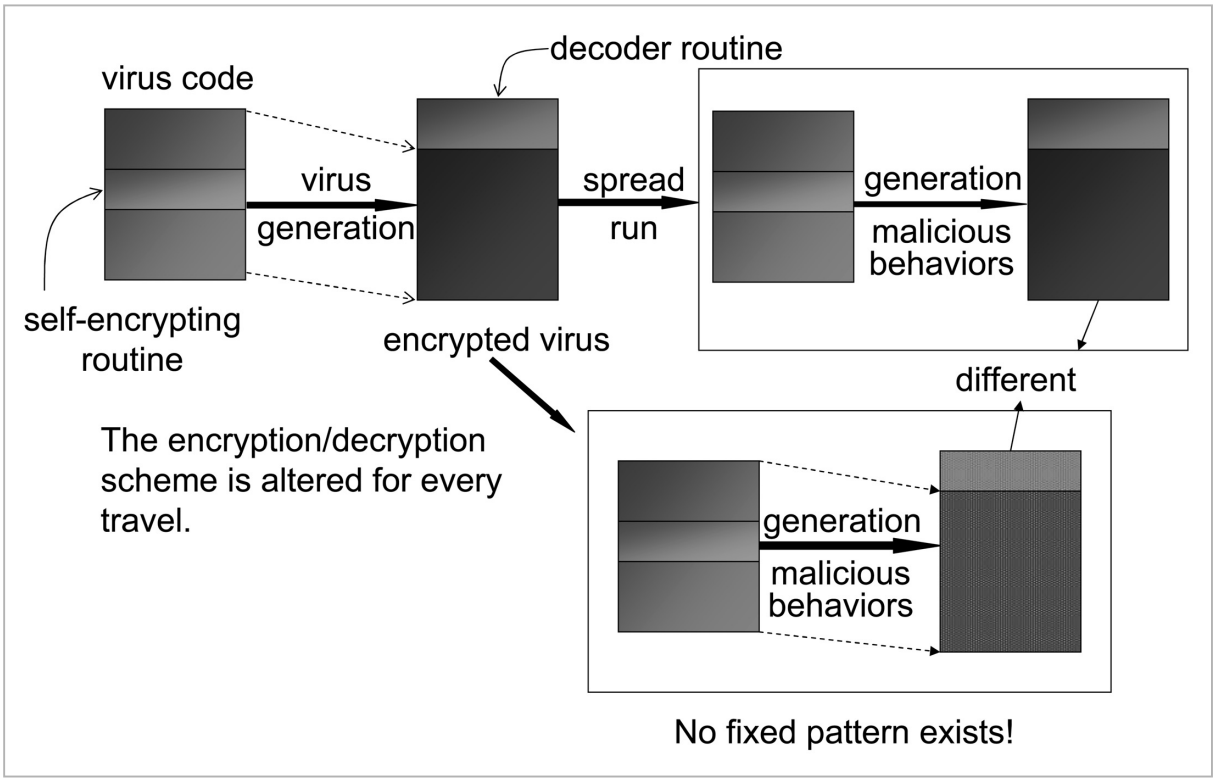

Fig.3 *Polymorphic virus*

(2) Methods of identifying malicious behavior

To determine whether an executable program will exhibit malicious behavior (corruption of files, mailing of confidential documents, etc.), it is necessary to analyze the special function calls (API function calls for Windows; system calls for Unix) requested by the program from the operating system. Operating systems such as Windows and Unix protect files and other resources from direct manipulation by ordinary programs. Even viruses must call functions from the operating system in order to perform any given malicious action (Fig.4). A mechanism is therefore required to identify sets of function calls that are indicative of malicious behavior.

In this study, we used code simulation technology to supplement static code analysis. This enabled us to analyze runtime behavior without executing virus code in a real machine environment. In the case of a self-encrypting virus, for example, the virus can decrypt its code by itself on the simulator; this code can then be used to perform static analysis.

Since each type of viral behavior is related to a specific pattern of API function calls (as shown in Fig.4) we identified these patterns in the process of code analysis. Based on the API function calls, we defined anticipated viral behavior with advanced policies, and used these policies as criteria to determine whether a given program was viral. Conventional pattern matching, on the other hand, relies solely on syntactic information. Providing semantic information on the behavior of viruses, these policy definitions play a key role in increasing detection sensitivity to previously unreachable levels.

In the following sections, we will detail a new detection tool and a method built on the principles outlined above.

4.1 Code simulations

A code simulation precisely imitates changes in the internal structure of an x86-architecture CPU (i.e., changes to registers, memory, flags, etc.) through the execution of machine instructions. The code simulator we used is operable both on Windows and Linux, and features basic debugging functions such

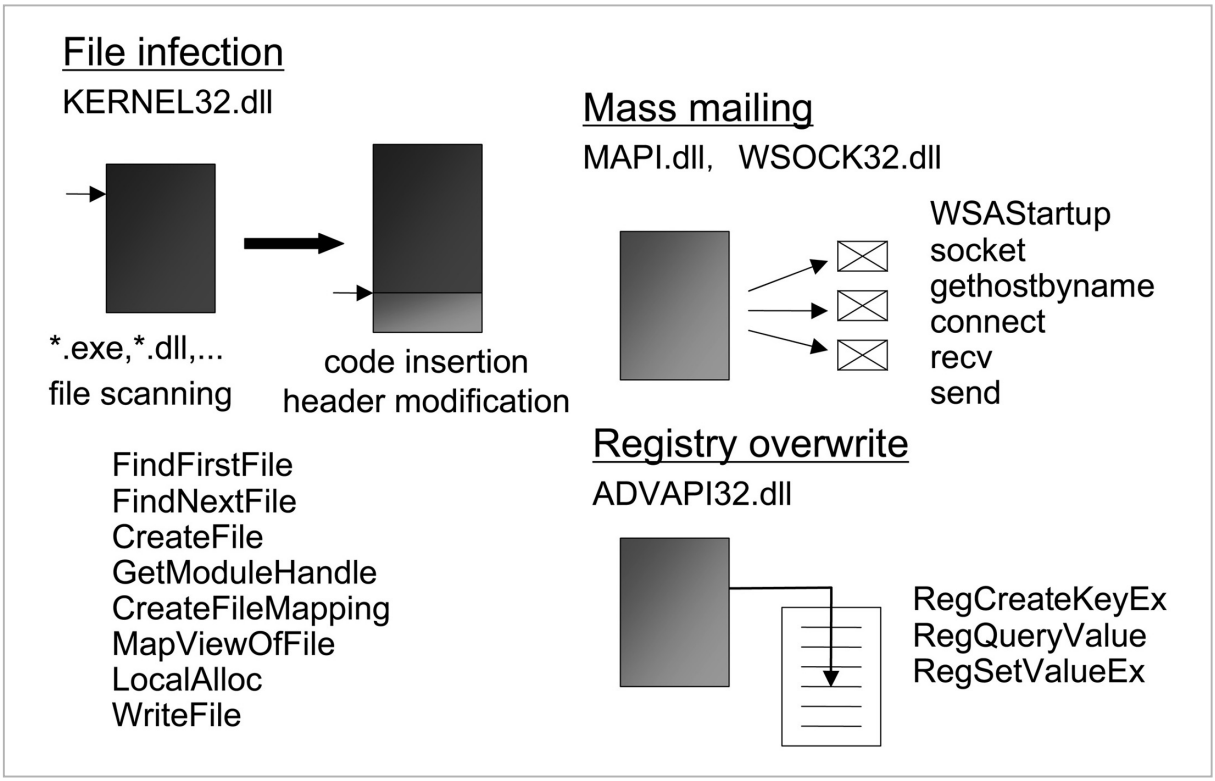

Fig.4 *Identification of virus patterns by analysis of API function calls*

as step execution, breakpoint setting, and memory dump.

As described above, we cannot analyze the behavior of self-encrypting and polymorphic viruses by analyzing program code. To track virus decryption and malicious behavior we must perform sequential simulation of executed instructions.

4.2 PE loader functions

However, static code analysis for virus detection cannot be performed adequately using simulations of machine instructions alone. In addition to executable code, a program stores additional information—for example, externally defined functions and memory addresses (for storage of executable code or initial assignment of execution control)—in various locations within a file. The file must therefore be scanned to extract this information and to load the executable code in the proper memory addresses on the simulator. In this way, the simulator performs the PE (Portable Executable) loader functions normally executed by the operating system. Since we focused on the Win32 environment, we designed the simulator to process PE binary format files.

4.3 Processing of external API function calls

API functions are provided as an external library for application program use. To perform code simulations using API functions, a complete real machine environment must be prepared; however, most API functions are unrelated to virus detection. As a result, the simulator must skip any instruction to call one of these unrelated API functions (instead recording the occurrence of a call), and proceed to a state in which this subroutine call is finished. To accomplish this, the following steps are required:

- Addresses are established that are used to store the addresses of external functions established at runtime. Specifically, the following addresses are established, with their respective stored addresses and the names of the external functions:

  a. An address allocated by the loader to

each program upon loading; and

b. An address allocated by the virus code to itself at runtime, using an API function such as LoadLibrary (used to dynamically load a library) and GetProcAddress (to acquire the address of a function in the loaded library).

- Arguments are removed from the stack.[2]
- Policy checking is executed (state-transition processor [state machine] is driven to detect policy violations, as described below).

In practice, however, there are so many API functions that system extensibility may not be ensured if these steps are included in the simulator's code for each API function. As a solution, we designed the simulator to call a dummy function (called a "stub function"), instead of an API function, to perform the necessary processing, and we prepared a separate library of stub functions corresponding to individual API functions. We have already enabled automatic generation of stub function templates through mechanical processing of Windows system information (using the byte count of the argument for a given API function). It is also possible to handle each function without affecting the remaining functions.

2 In accordance with one rule of the Windows environment, arguments need to be removed from the stack by library functions. However, it is necessary to deal with the cases individually, to which the above rules is not applicable.

4.4 Virtual runtime environment

As described above, preparing the following allows us to analyze programs and identify virus-like behavior—essential for virus detection—based on API function calls without requiring virtual execution of external library code:

- Mechanism to load executable files/ libraries
- Stub functions
  - Removal of arguments from the stack
  - Policy checking (state-transition processor [state machine] is driven)

However, if we are to perform analysis in greater depth we must collect more detailed information on the runtime environment, and we must generate in a virtual environment the side effects associated with program execution. Our newly developed detection tool uses a Windows virtual environment database and stub function values to meet the former and latter needs, respectively.

Specifically, the following must be addressed in a virtual runtime environment:

- Windows virtual environment database
  - Registry
  - Shell environment variables
- Runtime information
  - Heap areas: dynamic work areas in memory allocated by an API function such as "HeapAlloc"
  - Files: directories and files created or opened by API functions such as "CreateFile"

Since the Windows registry itself is a large database and the uses of many of its registry keys are unclear, the virtual environment database only includes registry keys referenced by ordinary programs or likely to be abused by viruses. A standard Windows environment has over a thousand runtime libraries, and a platform SDK includes an enormous number of API functions. As a result it is extremely difficult to set stub functions for all of these libraries by plugging in argument byte counts, return values in successful execution, and the like. Complicating matters further, different return values (for zero values, character counts, and error codes, for example) are used among these API functions. It is currently possible to generate stub function templates automatically for API functions defined in approximately 100 frequently used libraries. Special return values need to be entered manually with reference to various sources.

In terms of runtime information, virtual heap areas are allocated to individual programs so that virtual memory areas are allocated in the virtual heap areas. Although no actual files are created, file structures are generated and managed based on the relevant stub

functions, and the resultant information can be used in subsequent policy checking. As described above, our new tool is designed for accurate simulation of the execution of instructions by the CPU alone. Additional operating system processes such as PE loader functions must be handled separately. A number of other features must also be managed appropriately:

- Memory management including paging
- Exception interrupts
- Thread management

Exception interrupts can be handled by SEH (Structured Exception Handling), a Windows platform-specific mechanism. Paging is indispensable for basic access control. The new tool can handle basic paging as well as exception interrupts caused by the paging operation. In thread management, memory areas must handle thread-specific data known as TEBs (Thread Environment Blocks), and a multithread execution mechanism must also be available on the CPU simulator for synchronous and exclusive control. The new tool supports open information with TEBs and simply analyzes elements of thread code in the order of generation with multithread. This is currently considered sufficient for behavior identification. However, if several threads coordinate to carry out a single malicious task, the tool may not be able to perform policy checks in high detail. One of our next goals is to enable the tool to emulate thread execution in greater detail.

4.5 Static code analysis

When simulating program code simply in a sequential manner, it is only possible to assess behavior that happens to occur during the simulation (as in the case of emulation-based dynamic protection). On the other hand, it is extremely difficult to check every element of code when a virus is capable of self-encryption or polymorphism.

In the early stages of development, we considered using a backtracking method for each branch instruction. However, taking snapshots of CPU usage and memory proves inefficient, and we found it extremely difficult to handle loop structures. The current control system first performs static code analysis to read instructions for API function calls that must be checked, and then leads the simulator, on a priority basis, to execution paths that include these API function calls (see Fig.5).

Static code analysis essentially consists of steps performed by a disassembler:

(1) One byte of data is extracted from each address in memory starting from a given address.
(2) Reference is made to a machine instruction established to determine the instruction type(opcode) and argument (operand) for the extracted data.
(3) The appropriate number of bytes is skipped, according to the instructions.

The static code analyzer repeats these steps. We devised a technique to mark each element of code (separated by a branch instruction and a jump address) to indicate the completion of analysis. This technique allows the simulator to call the analyzer again for a subsequent (unmarked) element of code newly generated by self-modification. The tool can therefore efficiently check elements of code that modify themselves dynamically.

4.6 Policy check system

As described above, policies anticipate the unique, destructive behavior of virus. Specifically, they define scenarios covering mass mailing, file infection, and so on with the same degree of detail applied in API function calls. Based on these policy definitions, the new tool checks the results of code simulations and static code analysis to determine whether defined types of behavior can be identified within the code. In addition to an intended type of behavior (mainly represented by certain API function calls), each policy defines a state-transition processor (state-transition machine) that uses additional information (argument strings in the case of API calls) as input parameters. For example, a certain type of mass mailing behavior is defined as follows: when a “send” function is called from

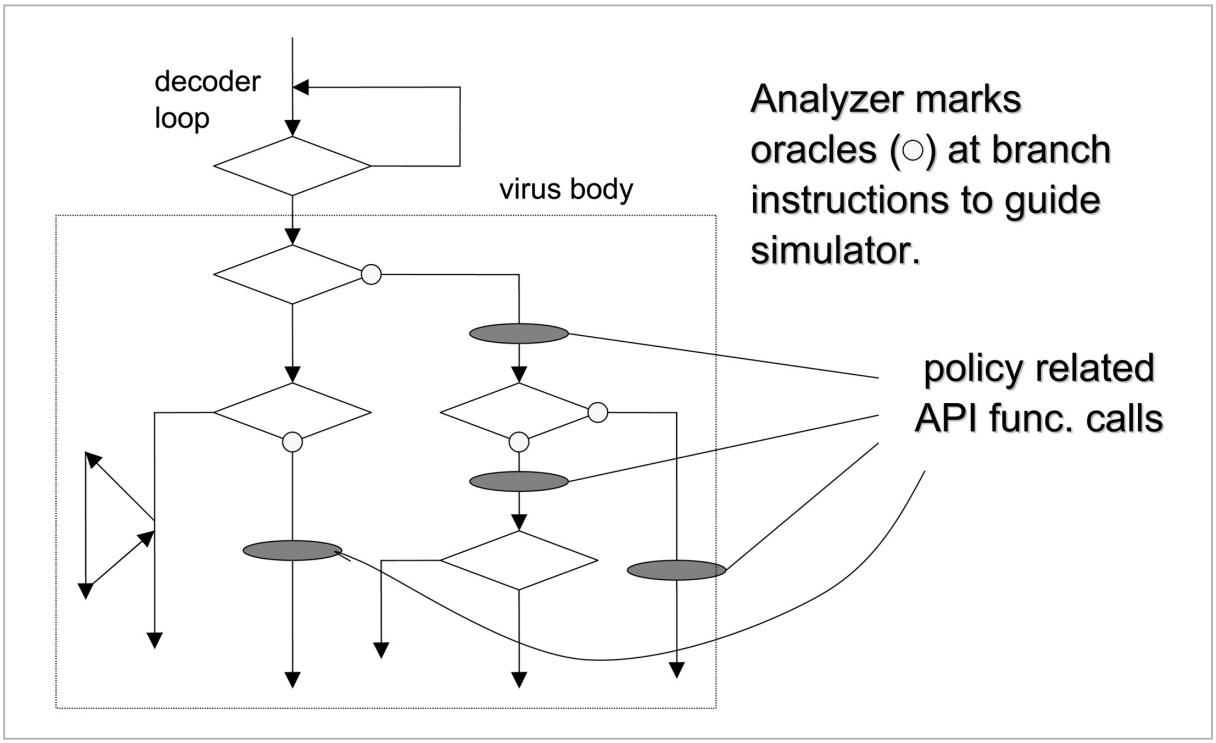

Fig.5 *Combined use of simulation and static analysis*

the WSOCK32.dll library, a state transition will occur from "socket connection completed" to "mass mailing has taken place" if the "To" attribute (destination) of the function's second argument (message) is randomly obtained from a certain file and the first argument (socket) is connected to a default mail server address obtained from a certain registry.

Based on the information obtained by simulations and static analysis, the policy checking system drives the state-transitioning as defined in the policies to determine whether the "accept" state is reached (see Fig.6). To perform policy checking using the state-transition processor (state-transition machine) in this manner, the tool is designed to activate the policy checking system through stub functions under the control of the simulator. If the simulator is forced to jump based on the static analysis results, it is led on a priority basis to the execution paths that include the API function calls to be checked. As a result, policy checking may become inconsistent with policy definitions. However, we have checked over 200 virus types so far and have yet to encounter such a problem.

5 Techniques to circumvent antivirus software and corresponding countermeasures

This section will give an overview of the various viral techniques currently used to circumvent antivirus software and will describe how our new detection tool counters these techniques.

5.1 Concealment of API function calls

The new tool is designed to identify virus behavior based on an API function called by programs from the operating system. If the tool fails to check API function calls, detection will be unsuccessful. Since it is possible to estimate roughly how programs behave by determining which API functions they call, an increasing number of viruses now use various techniques to conceal their API function calls.

The most basic technique is to use an API function (e.g., GetProcAddress) to acquire API function addresses at runtime. In the case of an ordinary executable program, the runtime libraries and API functions to use are declared beforehand, and the function address-

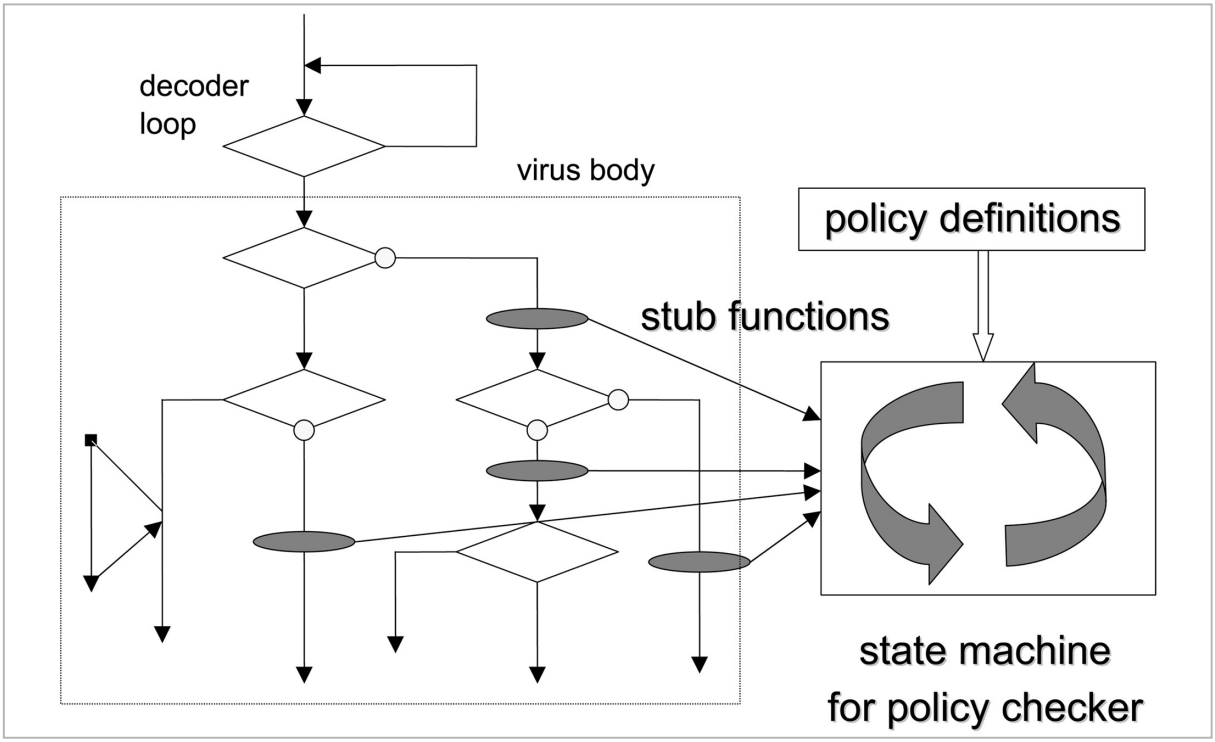

Fig.6 *Policy checking mechanism*

es are established when the loader loads this program in memory. In contrast, if a program acquires API function addresses at runtime, it is only possible to determine which API functions are called through execution of the program.

To counter this technique, we implemented a mechanism to allocate virtual addresses for use by both the loader and by GetProcAddress, to enable subsequent determination of the functions that were called. Currently, however, many viruses can acquire API addresses by scanning runtime libraries loaded in memory, without needing to use GetProcAddress. Merely by determining the start address of a library, it is possible to acquire all runtime addresses from the memory image of the loaded library in PE format (executable file format of Windows platform). Moreover, the following technique is also commonly used:

(1) A library is loaded (e.g., "foo.dll") that includes the desired API function. This library is declared in advance, or loaded by the LoadLibrary API function at runtime.
(2) An address is acquired for an API function[3] (e.g., "Func") other than the desired function defined in the loaded library. The acquired function is declared in advance, or acquired by the GetProcAddress function at runtime.
(3) The library's memory image is scanned on the 16-bit boundary in the forward direction from the address of Func; the magic word "MZ" is located in its position at the beginning of the PE-format image, providing the library's initial address[4].

Through these complicated steps, many viruses make it difficult to identify which libraries they are scanning. Actual library files must be prepared to identify viral behavior in a virtual environment. It appears that these steps are intended to prevent identification of virus behavior by simple emulation. This type of special technique is common to many viruses, a phenomenon attributable to the active exchange of information among virus authors and their extensive imitation of techniques published in clandestine online magazines generally referred to as "e-zines".

Even if a virus applies the technique described above to acquire the addresses of API functions, the new tool can identify the

specific API functions called through emulation of the PE loader which loads library files. Instead of actual Windows platform library files, this tool uses PE-format pseudo-library files. Although these files do not include the actual code of the API functions[5], they hold enough information to simulate viral memory scanning behavior. Since such scanning behavior is considered simply unnecessary with ordinary programs, this behavior can be incorporated in a policy to detect viruses (see "IAC policy" in Section **6**).

3 In terms of viruses, this is preferably a common function unrelated to virus detection.

4 In the case of a Windows platform, executable files are loaded on the 16-bit boundary.

5 These hold only open library information such as names and addresses.

5.2 Techniques against debuggers/emulators

To detect unknown viruses, many antivirus software vendors use a heuristic scan method that mainly checks for suspicious program behavior in an isolated runtime environment. To perform heuristic scanning it is necessary to emulate a complete and real machine environment. Using an OS-level debugger, however, may not allow for such complete emulation.

5.2.1 Abuse of SEH (Structured Exception Handling)

Below we will describe a typical example of SEH (Structured Exception Handling) on a Windows platform.

SEH is intended to handle exceptions in a unified and efficient manner, and is processed by the operating system at a high CPU interrupt level.

Specifically, the segment register FS stores a pointer to the exception handler structure for the current thread at the address "0". This structure usually consists of two dwords (four bytes in total) and is stored in a stack area. The first dword is a pointer to a next exception handler structure, and the other is the address of the handler code. Connecting all exception handler structures through pointers in this way, SEH enables dynamic selection of the exception handler for use in execution.

Debugging usually presents no problem at the OS level. However, we are seeing an increasing number of viruses that force exception interrupts by attempting intentional invalid access to certain memory addresses or by executing division by zero. In such cases, a self-decryption routine or payload (indicating malicious behavior) execution code is entered as an SEH handler against these intentionally caused exceptions. SEH is processed by Windows OS. Therefore, if a debugger is implemented through the operating system, SEH processing will be taken over by the OS, and exception handling will be performed not in the thread for the program to be analyzed but rather in the thread for a different debugger application. As a result, the programmed behavior will not be exhibited and virus detection will be unsuccessful.

When control flow changes due to conditional branching, analysis may be performed by following every execution path. However, when even jump addresses are dynamically set by a special mechanism such as SEH, it is impossible to perform analysis without the processing capability required by the virtual execution mechanism. The new tool is designed to accurately process SEH as an essential function in coordination with exception interrupt handling by the code simulator. However, SEH processing also involves OS-side processing, the details of which are uncertain (i.e., not open to the public), such as the various types of data on the stack, convolutions of the stack when one handler is activated after another, and use of the terminal handler for cleanup processing in a try-finally construct. It may thus be wise to envision future measures against the emergence of new techniques to abuse SEH.

5.2.2 Other techniques against debuggers/emulators

The following techniques are currently used to check for the presence of a debugger/emulator:

• Techniques to check for the presence of a

debugger when debugging functions are implemented through the operating system:

a. Use of an "IsDebuggerPresent" function included in the KERNEL32.DLL library
b. Use of the flag value stored at the address "0x20" of the segment register FS
c. Exploitation of the significant change in values in the segment register FS in the presence of a debugger, relative to an ordinary Windows runtime environment.

- Techniques to check whether programs are executed in an emulated environment make use of the following:

a. Attempted access to an address with very large memory size (up to 4 GB) and verification of success or failure
b. The presence of specific definition files in the system directory, or of specific registry keys
c. The obvious abnormality of system information obtained from BIOS in an emulated environment
d. Instruction specifications not included in the CPU specifications—the AAM instruction, for example. Since this instruction takes any 1-byte operand in division, it is possible to use numbers other than the ten numbers included in the specifications[6].

Through the adoption of memory management by paging, code simulation, and a virtual runtime environment, the new detection tool can prevent programs from recognizing that they are being executed in a virtual machine environment. However, as in the case of SEH above, it would be wise to remain vigilant against the emergence of new techniques enabling such recognition.

It is also possible to define the behavior described above within policies for virus detection, based on the perception that ordinary programs never behave in this manner. For example, under normal conditions, a program that uses the IsDebuggerPresent function can clearly be considered a virus (see the anti-debugger/emulation policy in Section **6**).

6 However, this has yet to be incorporated into Intel's manuals.

6 Policies

Using the new tool and method described above, it is possible to analyze nearly all existing viruses and useful programs. The problem lies in determining which programs qualify as malicious; here it is essential to define the appropriate policies. Many policies are now implemented within this tool; the main ones are described below:

- Mass email policy: Checks for the scenario in which each program acquires an SMTP server address and random destination addresses based on registry information in order to issue a message. Simulation is performed using the SMTP protocol to determine whether mass emailing actually occurs. Libraries are used in accordance with several scenarios; for example, with different sockets: (WSOCK32.DLL, WS2_32.DLL), WININET.DLL, or MAPI.DLL.
- Registry modification policy: Checks for any modifications to system settings, especially registry keys in which autostart programs are listed. Defines a separate list of prefixes for the registry keys to be protected, and matches this list against API function stubs related to the registry settings.
- File modification policy: Checks for any modifications to write-protect directories/files.Uses a prefix list of the directories/files as in the case of the registry modification policy.
- File infection policy: Checks to determine whether each program writes itself into files. Defines a scenario in which each program acquires a file handler by directory scanning, modifies it in memory, and writes it back to the file. Although many file infection scenarios

are possible, currently only typical scenarios are checked.

- Process scan policy: Checks for a scenario in which each program acquires process IDs from a list of processes under execution. It seems necessary to include process infections in this scenario. This is defined as a policy because ordinary programs are unlikely to behave in this manner.
- Self-code modification policy: Checks to determine whether each program modifies headers, especially of import tables, in its memory image. This is defined as a specific pattern because it is difficult to distinguish between an ordinary code modification by a self-extracting routine and a jump table modification by an ordinary program.
- Anti-debugger/emulation policy: checks for special behavior attempting to detect a debugger or emulator, as described in Section **5.2**.
- Out of bounds execution (OBE) policy: Checks to determine whether each program activates its code at the address of a header section in which the program has declared specifications. This is a quite common policy, and is implemented separately, as programs created by ordinary compilers or assemblers never behave in this manner. Files created by certain compressed executable file creation tools may be detected as viruses.
- External execution policy: Checks to determine whether each program starts an external program with a CreateProcess or ShellExecute function call.
- Illegal address call policy (IAC): Checks for any calls in which memory addresses were obtained by direct scanning of the memory image of library functions, as described in Section **5.1**.
- Self-duplication policy: Checks to determine whether each program copies itself during execution. This is implemented as a separate policy, as this behavior is common to many viruses.
- Network connection policy: Checks to determine whether each program performs FTP or HTTP communications. Here addresses are not checked as in the case of the mass email policy. Emulation of FTP and HTTP protocols is required.

These policies are written in C++ language as a program and called by stub functions from the library to drive the state-transition processor (state machine). We are now developing a description language to define more sophisticated policies and working to create a processing system for these policies. The most challenging task we face, which we are approaching in a step-by-step fashion, lies in the creation of an automatically convertible vocabulary list for policy definitions, abstracted from the dynamically changing states of target programs during virtual execution.

7 Experimental results

Using the policies described in Section **6**, we conducted detection experiments on approximately 200 types of viruses that have proliferated in recent years. Detection was successful with all of these viruses. We discovered that it is possible to detect not less than 95% of these viruses using the IAC, OBE, and self-duplication policies. Even without the IAC policy, a detection rate of over 85% is possible using the OBE and registry/file modification policies instead.

More than 80% of the virus samples had not been analyzed prior to the experiments; these viruses were thus unknown to the newly developed tool. The tool was able to detect unknown viruses we have seen proliferate in recent years, such as MyDoom, Bagle, Netsky, and all of their variants. These results confirm the effectiveness of this tool in actual environments.

We must also address the misidentification of normal programs as viruses, a problem arising with conventional detection methods. The new detection method, however, consists of a number of deterministic algorithms, and if a certain behavior defined in policy is not pre-

sent, it will not be judged as present. In this sense, the new method will generate no false positives due to misidentification; such generation is always related to the detection sensitivity of policy definitions. For example, with a common policy such as OBE, some safe programs are classified as viruses. This is because the detection sensitivity of OBE is high, not because the policy is incorrect. On the other hand, the new method allows for exact and detailed specification of detection criteria at runtime. This plays a pivotal role in the success of detection under this method and distinguishes the new method from existing heuristic detection approaches based on superficial program code structures.

The newly developed tool can be used as an independent program verification tool or as a virus filter, installed on a mail server or on individual mail clients to protect users.

8 Conclusions and goals for the future

Designed such that each of its functions can be extended independently, this tool is now nearly ready to proceed to the practical level. Test operation, addition of functions, and adjustments can be performed with ease, although a check program is required for each policy using C++.

On the other hand, there remains room for improvement in processing speed and scalability. Before putting this tool to practical use, operational experiments will be required in an actual environment to improve processing speed, detection performance, resistance to load, and operability. Specific improvements are required as follows.

- Refinement of data structures and algorithms. Thread management and other simply implemented segments must be redefined.
- It is difficult for general users to customize security policies that are based on C++ programs. Design and implementation of a policy description language and processing system are required.
- Detection processing sometimes cannot be performed properly due to the omission of datafor the virtual runtime environment, such as registry values, file structures, and dynamic libraries required to identify program behavior. A great deal of time and effort must be expended to collect data when defects are found in Windows platforms.
- Performance analysis must be executed under heavy load conditions and processing speed must be increased based on analysis results.
- The current policies are defined based on the analysis of viruses that have actually proliferated. Virus authors will devise new tricks, and it will become necessary to define general policies against each of them, even though this tool is designed to detect unknown viruses. It is essential that we define a set of comprehensive policies that will leave room for future response.

One of our goals is to establish an unknown-virus detection tool that can run on a mail server with several hundred users and detect not less than 95% of unknown viruses without affecting system throughput. The actual detection rate will depend on the type of viruses that emerge. As we work toward greater system sophistication, we are keeping an eye on current virus activity in our formulation of standard policies. The following are our goals for the coming year:

- To write each policy within an average length of approximately 20 lines using a newly designed policy description language; to prepare a virtual runtime environment database that can adequately handle viruses that have emerged to date
- To define a general policy set consisting of approximately 20 policies considered effective against anticipated types of viruses; to conduct detection experiments on unknown viruses on a mail server with several hundred users
- To detect not less than 95% of unknown viruses while maintaining system

throughput loss below user-noticeable levels; to prepare system operation documents and make the tool widely available for practical use

Acknowledgements

We carried out this study from fiscal 2001 to 2003 under a commission from the Telecommunications Advancement Organization of Japan (currently NICT). Our success in these efforts is due in large part to their support and guidance.

References

Very few papers and articles are available on unknown computer virus detection. We only list here a couple of widely circulated books on malicious mobile codes[1] and virus creation methods[2].

1 Roger Grimes, "Malicious Mobile Code: Virus Protection for Windows", O'Reilly & Associates Inc., 2002.
2 Mark Ludwig, "The Giant Black Book of Computer Viruses", Second Edition, Lexington & Concord Partners, Ltd., 2000.

The followings are Symantec's US patents concerning unknown virus detection methods.

3 USPTO disclosure, patent number 6357008, Dynamic heuristic method for detecting computer viruses using decryption exploration and evaluation phases.
4 USPTO disclosure, patent number 5696822, Polymorphic virus detection module.

***MORI Akira**, Dr. Eng.*
Group leader, National Institute of Advanced Industrial Science and Technology
Formal Method, Ubiquitous Computing, Security

SAWADA Toshimi
Senior Researcher, SRA Key Technology Laboratory, Inc.
Formal Method, Security

IZUMIDA Tomonori
Technical Staff, National Institute of Advanced Industrial Science and Technology
Computer Science

INOUE Tadashi
Chief Researcher, SRA Key Technology Laboratory, Inc.
Computer Virus Detection, Intrusion Detection System