Dynamic detection and classification of computer viruses using general behaviour patterns

Baudouin Le Charlier
Abdelaziz Mounji

University of Namur
Institut d'Informatique
{ble, amo}@info.fundp.ac.be

Morton Swimmer

University of Hamburg
Fachbereich Informatik
Virus Test Center
swimmer@acm.org

July 2, 1995

Abstract

The number of files that need processing by the virus labs is growing nearly exponentially. Even though only a small proportion of these files contain new viruses, each file requires examination. The normal method for dealing with these files in the virus labs is still brute force manual analysis. A virus expert runs several tests on a given file and delivers a verdict on whether it is virulent or not. If it is a new virus, it will be necassary to detect it. Some tools have been developed speed up this process. These range from programs that identify previously classified files to programs that generate detection data. Some antiviruses have built in mechanisms based on heuristics that enable the antivirus to detect unknown viruses. Unfortunately all these tools have limitations.

In this paper, we will demonstrate how an emulator is used to monitor system activity of a virtual PC, and how the expert system ASAX is used to analyse the stream of data the emulator produced. We use general rules to generically detect real viruses reliably, and specific rules to extract details of their behaviour. The resulting system is

called VIDES and is a prototype for an automatic analysis system for computer viruses and possibly a prototype anti virus for the emerging 32 bit PC operating systems.

1 Introduction

Virus researchers must cope with many thousands of suspected files each month. But the problem is less the number of new viruses, which number perhaps a few hundred a month but are growing at a nearly exponential rate. Instead it is the number of files the researcher receives and must deal process — the glut — that cause the problems. Out of a hundred files, only one may actually contain a new virus. Unfortunately, there are no short cuts. Every file has to be processed.

The standard method of sorting out such files is still brute force manual analysis that requires specialists. Some tools have been developed to help cope with the problem. These tools range from programs that identify previously classified files and viruses and remove them, to utilities that extract strings from infected files that aid in identifying the viruses. None of the solutions are satisfactory, though. Clearly, more advanced tools are needed.

In this paper, the concept of dynamic analysis is discussed as applied to viruses. This is based on an idea called VIDES (*Virus Intrusion Detection Expert System*) coined at the Virus Test Center [BFHS91]. The system comprises of a PC emulation and an IDES-like expert system. It should be capable of detecting viral behavior using a set of *a priori* rules, as shown in the preliminary work done with Dr. Fischer–Hübner. Furthermore, advanced rules will help in classifying the detected virus.

The present version of VIDES is primarily of interest to virus researchers as it is not designed to be a practical system for the end–user. Its demands on processing power are too high. However it can be used to rapidly identify unknown viruses and provide detection and classification information to the researcher. It also serves as a prototype for the future application of intrusion detection technology in detecting malicious software under emerging and future operating systems, such as OS/2, MS–Windows NT and 95, Linux, Solaris, etc.

The rest of the paper is organized as follows. Section 2 presents the current state of the art in anti-virus technology. Section 3 describes a generic virus detection rule. Section 4 discusses the architecture of the PC auditing

system. In section 5 we show how the expert system ASAX is used to analyze the activity data collected by the PC emulator. Finally, section 6 contains some concluding remarks.

2 Current state of the art

For the purpose of discussion it will be necassary to define the term computer virus.

2.1 Terms

There are still no definitions for computer viruses that can be universally agreed on. What is missing are precise definitions that are still general enough to account for all possible implementations of computer viruses. An attempt was made in [Swi95], which is the result of many years of experience with viruses in the Virus Test Center. The following definition for a computer virus is the result of discussion in comp.virus (Virus–L) derived from [Seb]:

Def 1 *A* Computer Virus *is a routine or a program that can "infect" other programs by modifying them or their environment such that a call to an infected program implies a call to a possibly evolved, functionally similar, copy of the virus.*

A more formal, but less useful definition of a computer virus can be found in [Coh85]. Using the formal definition it was possible to prove the virus property undecidable.

We talk of the infected file as the *host program*. System viruses infect system programs, such as the boot or master boot sector, whereas file viruses infect executable files such as EXE or COM files. For an in–depth discussion of properties of viruses please refer to literature such as: [Hru92], [SK94], [Coh94] or [Fer92].

Today, anti virus technology can be divided into two approaches: the *virus specific* and the *generic* approach. In principle, the former requires knowledge of the viruses before they can be detected. Due to advances in tehnology, this prerequisite is no longer entirely valid in many of the modern anti viruses. This type of technology is known to us as a *scanner*. The latter attempts to detect a virus by observing attributes characteristic of all viruses.

For instance, integrity checkers detect viruses by checking for modifications in executable files — a characteristic of many (although not all) viruses.

2.2 Virus specific detection

Virus specific detection is by far the most popular type of virus protection used on PCs. Information from the virus analysis is used in the so–called scanner to detect it. Usually a scanner uses a database of virus identification information so that it can detect all viruses that have been previously analyzed.

The term *scanner* is becoming increasingly incorrectly called so. The term comes from *lexical scanner*, i.e. a pattern matching tool. Traditionally scanners have been just that. The information extracted from the viruses were strings that were representative of that particular virus. This means that the string has to

- differ significantly from all other viruses, and
- differ significantly from strings found in bona fide programs.

Finding such strings was the entire art of anti virus writing until polymorphic viruses appeared on the scene.

Encrypted viruses were the first challenge to the string searching methods. The body of the virus was encrypted, and could not be searched for due to its variable nature. However, the body was prepended by a decryptor–loader, that has to be in plain text (unencrypted) else it would not be executable. This decryptor can still be detected using string seaching methods even if it becomes difficult to differentiate between different viruses.

Polymorphic viruses are the obvious next step in avoiding detection. Here, the decryptor is implemented in a variable manner so that pattern matching becomes impossible or very difficult. Early polymorphic viruses were identified using a set of patterns (strings with variable elements). Moreover, simple virus detection techniques are made unreliable by the appearance of the so-called *Mutation Engines* such as MtE and TPE (Trident Polymorphic Engine). These are object library modules generating variable implementations of the virus decryptor. They can be easily linked with viruses to produce highly polymorphic infectors. Scanning techniques are further complicated

by the fact that the resulting viruses do not have any scan strings in common even if their structure remains constant. When polymorphic technology improved, statistical analysis of the opcodes was used.

Recently, the best of the scanners have shifted course from merely detecting viruses to attempting to identify the virus. This is often done with additional strings, sometimes position dependent, or checksums over the invariant portions of the virus. In support of this, many anti viruses have implemented machine–code emulators so that the viruses own decryptor can be used to decrypt the virus. Using these enhancements, the positive identification of even polymorphic viruses poses no problem.

The next shift many scanners are presently going through is away from known virus detection to detection of unknown viruses. The method of choice is *heuristics*. The heuristics built into the anti virus, attempt to infer whether a file is infected or not. This is most often done by looking for a pattern of certain code fragments that occur most often in viruses and not in bona fide programs.

Heuristics suffer from a moderate to high false positive rate. Of course, a manufacturer of a heuristic scanner will improve the heuristics to both avoid false positives and still find all new viruses, but both goals cannot be entirely achieved. Usually a heuristic scanner will contain a "traditional" pattern matching component, so that viruses can be identified by name.

2.3 Generic virus detection

Computer viruses must replicate to be viruses. This means that a virus must be observable by its mechanism of replication.

Unfortunately it is not as easy to observe the replication as it may seem. DOS, in it various flavors, provides no process isolation or even protection of the operating system from programs. This means that any monitoring program can be circumvented by a virus that has been programmed to do so. There used to be many anti viruses that would try to monitor system activity for viruses, but they were not proof against all viruses. This problem led to the demise of many such programs. Later in the paper we will discuss how we avoided the problem when implementing VIDES.

A more common approach is to detect symptoms of the infection such as file modifications. This type of program is usually called an *integrity checker* or *check–summer*.

When programs are installed on the PC, checksums are calculated over

the entire file, or sometimes portions of the file. These checksums are then used to verify that the programs have not been modified. The shortcoming of this method is that the integrity checker can detect a modification in the file, but cannot determine whether the modification is due to a virus or not. A legitimate modification to, for instance, the data area of a program will cause the same alarm as a virus infection.

Another problem is virus technology aimed specifically against anti viruses. Advances in stealth and tunneling technology have made updates necessary. There have also been direct attacks against particular integrity checkers, rendering them useless. Again, the lack of support from the operating system makes the prevention of such attacks very difficult. As a consequence, acceptance of such products is low.

The non–specific nature of the detection has little appeal for many of the users. Even generic repair facilities in the anti viruses do not help, despite these methods effectively rendering identification unnecessary. The problem is partially understandable. The user is concerned with his data. Merely disinfecting the programs is not enough if data has been manipulated. Only if the virus has been identified and analyzed can the user determine if his data was threatened.

Generic virus detection technology should not be dismissed. It is just as valid as virus specific technology. The problems so far have stemmed from the permissiveness of the underlying operating system, DOS, and from the limits in the programs. Both problems can be addressed.

3 Dynamic detection rules

Before we can attempt to detect a virus using ASAX, we need to model the virus attack strategy. This is then translated into RUSSEL, the rule–based language that ASAX uses to identify the virus attack.

3.1 Representing infection patterns using state transition diagrams

State transition diagrams are very suitable for representing virus infection scenarios. In this model of representation, we distinguish two basic components: a node in a state transition diagram represents some aspects of the computing system state. Arcs represents actions performed by a program in

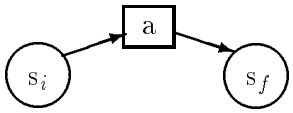

Figure 1: State transition diagram

execution. Given a (current) state s_i, the action a takes the system from the state s_i to the state s_f as shown in Figure 1. The infection process played by a virus can be viewed as a sequence of actions that drives the system form an initial *clean* state to a final *infectious* state where some files are infected. In order to get a complete description of the actual scenario, a state is adorned by a set of *assertions* characterizing the objects as affected by actions.

In practice, we only represent those actions that are relevant to the infection scenario. As a result, many possible actions may occur between adjacent states, but are not accounted for because they do not entail a modification in the current state. In terms of auditing, irrelevant audit records may be present in the sequence of audit records representing the infection signature.

For the sake of simplicity, discussion of the generic detection rules are based on the state transition diagrams described above.

3.2 Building the rules

VIDES uses three types of detection rules: *generic detection rules*, *virus specific rules*, *other rules*. As the name implies, generic rules are used to detect all viruses that use a known attack pattern. For this, models of virus behavior are needed for the target system (in our case MS–DOS). Virus specific rules use information from a previous analysis to detect that specific virus, or direct variants of it. In this way, these rules are similar to virus specific detection programs except that they analyze the dynamic behaviour of the virus instead of its code. Finally there are other rules for gleaning other information from the virus that can be used in its classification.

We will not go into the virus specific rules or the other rules, but instead concentrate on the generic rules.

In developing a generic rule for detecting viruses, we need to have a model for the virus attack. No one model will do, because MS–DOS viruses can choose from many effective strategies. This is compounded by the diversity of executable file types for MS–DOS. Fortunately for us, the majority of viruses

have chosen one particular strategy and infect only two types of executable files. This means we can detect most viruses with very few rules. On the other hand, a virus that uses a unknown attack strategy will not be detected. For this reason, the prototype analysis system contains an auxiliary static analysis component to detect such problems.

In the following we will develop a generic rule that detects file infectors that modify the file directly to gain control over that file. We will concentrate on COM file infectors, but EXE file infectors are detected in an analogous way.

We need to make two assumptions about the behaviour of DOS viruses to help us build the rule.

Assumption 1 *A file infecting virus modifies the host file in such a way that it gains control over the host file when the it is run.*

This is a specific version of the virus definition (Def 1). However, it does not specify when the virus gains control over the host file.

Assumption 2 *The virus in an infected file receives the control over the file before the original host does.*

That is, when the infected file is run, first the virus is run and then the host program.

Discussion: If the virus never gains control over the host file, it would not fulfill the definition of a virus. This observation leads to Assumption 1. However there is no reason (in the definition) why the virus must gain control before the host does.

We make the additional assumption that the virus *does* gain control before the host does. The reason we do this, is to avoid very blatant false positives. However it should be noted that Assumption 2 is not a result of the virus definition and can cause some viruses to be missed. For these cases, other rules are used.

3.3 Finding COM file infections:

With respect to assumptions 1 and 2 we are looking for two possible infection strategies:

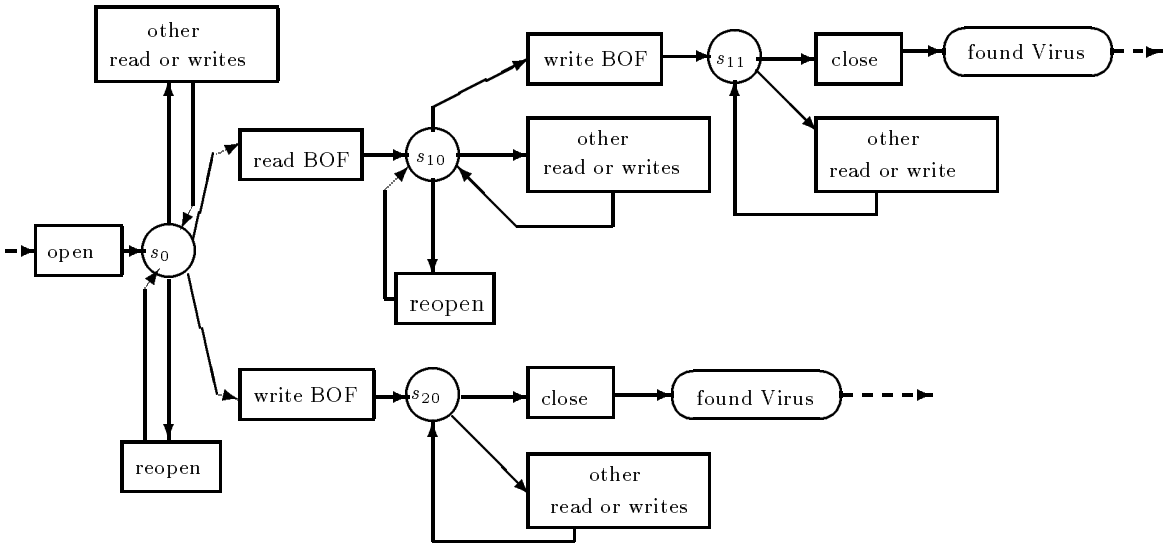

Figure 2: Generic rule for identifying COM file infectors

1. The virus is *overwriting*. Therefore we are looking for a write to the beginning of file (BOF), without a previous read to the same location. Other read and write are permitted.

2. The virus is *non–overwriting*. We expect to see a read to BOF and then a write to BOF. Before, inbetween, and after these two events other reads and writes are permitted.

The assumption in both cases is that the write to BOF causes the virus to gain control on execution.

In the case of a non–overwriting virus, we assume that the virus first reads the original code at BOF and then replaces it with its own code, usually a *jump* to the virus body. In most cases the number of bytes read will be the same as the number of bytes written, but we cannot assume this. In the case of an overwriting virus, the code is not read (and saved somewhere) but overwritten.

Other read and writes are not actually relevant to the detection of the virus. They can be logged and used in generating virus specific rules.

The rule is initiated by the opening of a file (in this case, a COM file). The rule is terminated by a close of the file, where this does not have to be

done by the virus itself. Inbetween these two events we expect the actual infection to occur. We look for the *read BOF* followed by the *write BOF* or the *write BOF* without the read. Other administrative operations, like tracking the file position, are also done by the rule. This is shown in the state transition diagram of Figure 2.

Some viruses cause problems for the rule by closing the file after a given set of operations. This is handled by a *reopen* mechanism that waits for a possible open event on the same file from the virus. So that this rule does not stay active indefinitely and clog up the rule memory, there are a number of terminating conditions. In fig. 2, *reopen* is abstracted as a transition, whereas its implementation is as a separate rule.

MS–DOS provides two methods of accessing files. The most common method is by using file handles. Access using *file control blocks* (FCB) was provided for compatibility to CP/M and is rarely used, even by viruses. However because it is used, we need a seperate rules to handle this method. The basic rule stays the same, but internal handling of the data is different.

We could avoid this problem by abstracting the audit data to give us a generic view of the system events. This way we could reduce the number of audit records to only relevant higher–level records by using a filter. After that, the processing becomes simpler as the problems of reopens and handle/FCB use disappear. This method also allows us to apply the rules on non–MS–DOS systems that provide similar file handling.

As a matter of fact, ASAX itself is the logical choice to act as the filter. The first ASAX system reads the raw audit trail, converts it into generic data and pipes it's output for further processing (see 5). Using ASAX as a filter allows us to reduce the complexity of maintaining such a system, not sacrificing any power.

4 PC auditing

The prerequisite for using an Intrusion Detection (ID) system like ASAX, is an audit system that securely collects system activity data. In addition, integrity of the ID system itself must not be compromized which means that the audit data retrieval, analysis and archiving must be secured against corruption by viruses. Moreover, The ID system must not be prevented from reporting (raising alarms updating virus infromation databases) the results of such analysis. DOS neither provides such a service nor makes the

implementation of such a service easy. Its total lack of security mechanisms means that the collection of data can be subverted. Even if the collection can be secured, the data is open to manipulation if it stored on the same machine.

For the prototype of VIDES we were not bound to a real world implementation, so we explored various alternative possibilities. The experience gained by the use of such a system will not benefit DOS users, but should be applicable to users of various emerging 32 bit operating systems that offer DOS support.

We have made several attempts to build asatisfactory audit system that are described hereafter.

4.1 DOS interrupts

All DOS services are provided to application programs via interrupts, which can be described as indexed inter–segment calls. Primarily, interrupt 0x21 is used. The requested service is entered into the AH register and its parameters are entered into the other registers. When the service is finished, it returns control to the calling program and provides its results in registers or in buffers.

The very first implementation of an auditing system was a filter that was placed before the DOS–Services and registered all calls to DOS functions. This was done very early on together with Dr. Fischer–Hübner to prove the feasibility of the VIDES concept. It also demonstrated the limits that DOS imposes on the implementation of such an auditing system: it did not run reliably and could be subverted by tunneling viruses.

This implementation was soon scrapped, but it did prove that the premises were correct: viruses could be found using ID technology. This was perhaps the first such a trial that had been done [BFHS91].

4.2 Virtual 8086 machine

The Intel iAPX 386 introduced the so–called virtual 8086 machine mode. A protected mode operating system can create many virtual 8086 machine in which tasks can run completely isolated from each other and from the operating system. Each task "sees" only its own environment. Operating systems like OS/2 use these constructs to provide a full DOS environment for DOS programs. All calls to the machine (via the BIOS interface or direct

port access) and DOS are redirected to the host operating system (OS/2 in this case) for processing.

This mechanism can also be used to monitor the activity in DOS session. Because all interrupts are being redirected to the native operating system, the native operating system can securely and unobtrusively record the activity.

Care has to be taken how the virtual 8086 machine is implemented. The DOS windows in OS/2 have been shown in tests at the Virus Test Center to be too permissive. In the course of a comprehensive test including the entire collection of file viruses, many of the viruses running under a DOS window have managed to harm vital parts of the system. One problem was that OS/2 files were directly manipulatable from within the DOS session. However, this did not explain the corruption of the running operating system.

Even though using a virtual 8086 machine was originally the method of choice, such experiments showed that the complexity of building a safe implementation would be difficult. A more secure method was sought for the prototype.

4.3 Hardware support

Hardware debugging systems such as the Periscope IV can be used to monitor system events nearly in real–time. This is achieved by a card that is fitted between the CPU and the motherboard and can set break–points on various types of events on the PC's bus. The card is connected to a receiving card in a second PC that is used to control the debugging session.

Monitoring system behavior on a DOS machine can be accomplished by capturing the interrupt 0x21 directly or setting a break point in the resident DOS kernel. Special memory areas can be monitored by setting a break condition on access to those areas.

The monitoring is completely unobtrusive, i.e. the program will not notice a difference between running with or without the debugger. When an event is triggered, the PC is stopped while the controlling PC is processing the data. If the controlling PC is fast enough, the time delay should be nearly negligible.

A hardware solution using the Periscope IV is complicated by the problem of automating the processes necessary to test large numbers of viruses on different operating systems. With such a solution, it would offer the possibility of testing viruses on other PC operating systems, that require full iAPX 386 compatibility.

4.4 8086 emulation

The solution that was finally used was the software emulation of the 8086 processor. An emulation is a program that accepts the entire instruction set of a processor as input and interprets the binary code as the original processor would. All other elements of the machine must be implemented or emulated, e.g. the various ports. To simplify and quicken the emulation, the BIOS Code (Basic Input Output System — the interface between the operating system and the hardware) can be replaced with special emulation hooks so that the complicated machine access can be skipped as long as all access to those services are routed via the BIOS. In the case of the graphics adapter, the entire hardware must be emulated, whereas disk access can be handled with hooks in the BIOS.

Using an emulation gives us all the advantages of the hardware solution plus the possibility of handling everything in pseudo real–time with respect to the program running in the emulation. Because even the time–giving functions of the emulation are being steered by the emulation, when it gets interrupted to process an event, the time in the emulation can be stopped too.

The emulation is "safe" because the running virus has no access to the host machine at all. This is because the target machine's memory is being controlled entirely by the emulation and file accesses are directed to a virtual disk, stored as an disk image file.

The major problem of using an emulation is its slowness. Even on fast platforms, the running speed is only marginally faster than an original PC/XT.

4.5 Activity data format

Audit records representing the program behaviour in general and virus activity in particular have a pattern that is borrowed from the Dorothy Denning's model of Intrusion Detection [Den87] (<*Subject, Action, Object, Exception-Condition, Ressource-Usage, Time-Stamp*>). However, due to the way processes are handled in DOS, this pattern is slightly modified to collect available attributes that are useful. For instance, the code segment of a process is chosen instead of the common process identifier in most existing multi-user operating systems.

The audit record attributes of records as collected by the PC emulator

have the following meaning: *code segment* is the address in memory of the executable image of the program. *function number* is the number of the DOS function requested by the program. *arg(...)* is a list of register/memory values used in the call to a DOS function. *ret(...)* is a list of register/memory values as returned by the function call. *RecType* is the type of the record. *StartTime* and *EndTime* are the time stamp of action start and end respectively. The final format for an MS–DOS audit record is as follows: <*code segment, RecType, StartTime, EndTime, function number, arg(...), ret(...)*>. An example of an audit trail is given in fig. 3.

```
                                        ⋮
<CS=3911 Type=0 Fn=30 arg() ret( AX=5)>
<CS=3911 Type=0 Fn=29 arg() ret( BX=128 ES=3911)>
<CS=3911 Type=0 Fn=64 arg( AL=61 CL=3 str1=*.COM) ret( AL=0 CF=0)>
<CS=3911 Type=0 Fn=51 arg( AL=0 str1=COMMAND.COM) ret( AL=0 CX=32 CF=0)>
<CS=3911 Type=0 Fn=51 arg( AL=1 str1=COMMAND.COM) ret( AL=0 CX=32 CF=0)>
<CS=3911 Type=0 Fn=45 arg( AL=2 CL=32 str1=COMMAND.COM) ret( AL=0 AX=5 CF=0)>
<CS=3911 Type=0 Fn=73 arg( BX=5) ret( CX=10241 DX=6206 CF=0)>
<CS=3911 Type=0 Fn=27 arg() ret( CX=5121 DX=8032)>
<CS=3911 Type=0 Fn=47 arg( BX=5 CX=3 DX=828 DS=3911) ret( AX=3 CF=0)>
<CS=3911 Type=0 Fn=50 arg( AL=2 BX=5 CX=0 DX=0) ret( AL=0 AX=50031 DX=0 CF=0)>
<CS=3911 Type=0 Fn=48 arg( BX=5 CX=648 DX=313 DS=3911) ret( AX=648 CF=0)>
<CS=3911 Type=0 Fn=50 arg( AL=0 BX=5 CX=0 DX=0) ret( AL=0 AX=0 DX=0 CF=0)>
<CS=3911 Type=0 Fn=48 arg( BX=5 CX=3 DX=831 DS=3911) ret( AX=3 CF=0)>
<CS=3911 Type=0 Fn=74 arg( BX=5 CX=10271 DX=6206) ret( CF=0)>
<CS=3911 Type=0 Fn=46 arg( BX=5) ret( CF=0)>
<CS=3911 Type=0 Fn=51 arg( AL=1 str1=COMMAND.COM) ret( AL=0 CX=32 CF=0)>
                                        ⋮
```

Figure 3: Excerpt from an audit trail for the Vienna virus

4.6 Activity data collection

The audit system was integrated into an existing PC emulation by placing hooks into the module for processing all opcodes which correspond to the events (see fig. 4). These are primarily calls to the DOS functions. This

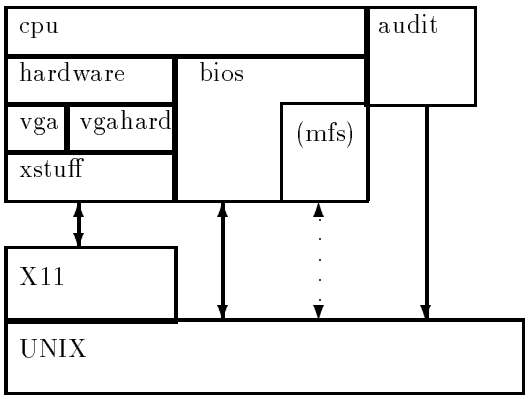

Figure 4: Modules in Pandora

was implemented in such a way, that stealth and tunneling viruses could not circumvent the mechanism. A separate module receives notification of the event and pushes all parameters on to a stack. When the DOS call returns, the parameters are popped from the stack and sent to the audit trail with the return values.

Internally the audit trail complies to a canonical format, which is also ASAX's native format.

An example of an audit trail is printed in Figure 3. This is a human readable representation of the binary NADF file. The example is from an audit trail of the Vienna virus. The text representation does not comply exactly with the binary version. Some of the less important fields are missing so that the audit record becomes clearer and shorter.

In the next section, we show how the activity data produced by the emulator is analyzed using ASAX.

4.7 Using RUSSEL to detect infection scenarios

In this section, we show how the RUSSEL language can be effectively used to detect an infection scenario. We first model the infection as a state transition diagram and then briefly show how this diagram can be translated in RUSSEL rules.

Each state in the diagram is represented by a rule describing not only the current state, but also the sequence of previous states leading to it. The actual parameters of the current rule encode all the relevant information col-

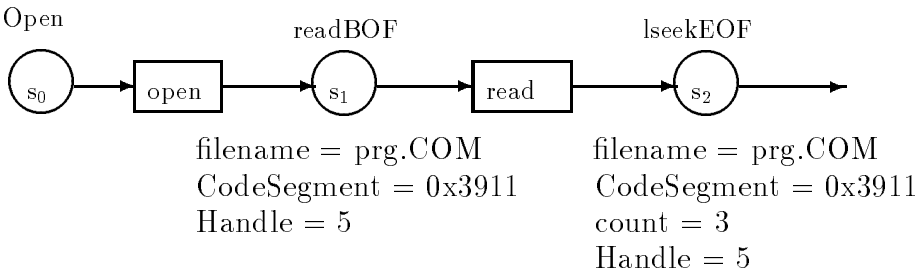

Figure 5: Example of a state transition diagram

lected in previously visited states. A transition in the diagram is represented by the rule triggering mechanism of the RUSSEL language as described in section 5. The actual parameters of the current rule are computed from the data items conveyed by the current audit record and from the parameters of the current rule. Once triggered, the new rule represents the new current state in the transition diagram.

In particular, the very first active rule at the beginning of the detection process has no actual parameters since no information is contained in the initial state. (One can argue that the initial state contains the assertion: system is clean which is then represented by an empty list of parameters.) As an example, the states s_0, s_1 and s_2 of fig. 5, are represented by the rules Open, readBOF(...) and lseekEOF(...) respectively. Figure 4.7 depicts this set of rules in the RUSSEL language. In this figure, RUSSEL keywords are noted in bold face characters, words in italic style identify fields in the current audit record, while actual parameters are noted in roman style words.

Finally, the transition leading to the final state does not trigger further rules but instead initiates a procedure which raises an alarm message describing the infection and using the data items accumulated along the path.

5 ASAX Features

The present section outlines the main features of the ASAX (*Advanced Security audit trail Analysis on uniX*) tool. It is used in VIDES as an expert system for intelligent analysis of virus activity data collected by the PC emulator. For a more detailed description of ASAX's architecture, the reader is referred to [HLCMM92a]. A comprehensive description of ASAX and its implementation is presented in [HLCMM92b, HLCM92].

```
rule Open;
if
        strToInt(Function) = 45 /* open file */
    and match('.*COM$', arg_str1 = 1
    ->
            trigger off for_next
                readBOF(ret_AX, CS, arg_str1)
    true
    ->
            trigger off for_next
                Open
fi;

rule readBOF(handle, codeSeg, filename: string);
if
        strToInt(Function) = 47 /* read file */
    and CS = codeSeg
    and ret_BX = handle
    ->
            trigger off for_next
                lseekEOF(handle, codeSeg, arg_CX, filename)

                                                    ⋮
fi;

rule lseekEOF(handle, codeSeg, count, filename: string);
if
                                                    ⋮
```

Figure 6: The FileOpen rule

ASAX has proved very powerful for efficient intrusion detection on UNIX platforms. It uses *a priori* rules for detecting malicious behaviors. Two versions of the ASAX system are currently available. The single audit trail analysis version is only applicable to a single audit trail. The other version allows a distributed analysis of multiple audit trails produced at various machines on a network. In the latter version, ASAX filters audit data at each monitored node and analyzes the filtered data gathered at a central host (see [MLCHZ95]). In the following, we briefly describe the main features of ASAX.

5.1 Universality

ASAX is theoretically able to analyze arbitrary sequential files. No semantic restrictions are imposed on the file being analyzed. For instance, analyzed

files could be trace data generated by a process controller or audit data collected in a multi-user environment. In the context of this paper, the sequential file is the activity data records produced by the PC emulator. The universality is attained by translating native files to a generic format which is the only one supported by the evaluator. This format is simple and flexible enough to allow a straightforward conversion of most file formats. This generic format is referred to as the *Normalized Audit Data Format* (NADF).

An NADF file is a sequential file of records in NADF format. A NADF record consists of the following:

- a four bytes integer representing the length (in bytes) of the whole NADF record (including the length field);
- a certain number of contiguous audit data fields. Each audit data field contains the three following contiguous items:

  identifier: an unsigned short (16 bit) integer which is the identifier of the audit data. This item must be aligned on a 2-bytes boundaries;

  length: an unsigned short integer which is the length of the audit data value;

  value: the actual audit data value.

In addition, audit data identifiers appearing in a NADF record must be sorted in a strict ascending order. This is important for ASAX to preprocess efficiently audit records before analysis. A user guide for constructing NADF files is presented in [Mou95].

5.2 Power: The RUSSEL language

RUSSEL (RUle-baSed Sequence Evaluation Language) is a novel language specifically tailored to the problem of searching arbitrary patterns of records in sequential files. The built-in mechanism of rule triggering allows a single pass analysis of the sequential file *from left to right*.

The language provides common control structures such as conditional, repetitive, and compound actions. Primitive actions include assignment, external routine call and rule triggering. A RUSSEL program simply consists of a set of rule declarations which are made of a rule name, a list of formal

parameters and local variables and an action part. RUSSEL also supports modules sharing global variables and exported rule declarations.
The operational semantics of RUSSEL can be sketched as follows:

- records are analyzed sequentially. The analysis of the current record consists in executing all active rules. The execution of an active rule may trigger off new rules, raise alarms, write report messages or alter global variables, etc;

- rule triggering is a special mechanism by which a rule is made active either for the current or the next record. In general, a rule is active for the current record because a prefix of a particular sequence of audit records has been detected. (The rest of this sequence has still to be possibly found in the rest of the file.) Actual parameters in the set of active rules represent knowledge about the already found subsequence and is useful for selecting further records in the sequence;

- when all the rules active for the current record have been executed, the next record is read and the rules triggered for it in the previous step are executed in turn;

- to initialize the process, a set of so-called *init* rules are made active for the first record.

User–defined and built–in C–routines can be called from a rule body. A simple and clearly specified interface with the C language allows to extend the RUSSEL language with any desirable feature. This includes simulation of complex data structures, sending an alarm message to the security officer, locking an account in case of outright security violation, etc.

5.3 Implementation

Efficiency is a critical requirement for the analysis of large sequential files, especially when on–line monitoring is involved. The very principle of the rule–based language RUSSEL allows to process each record only once whatever complex is the analysis. RUSSEL is efficient thanks to its operational semantics which exhibits a bottom–up approach in constructing the searched record patterns. Furthermore, optimization issues are carefully addressed in the implementation of the language: for instance, the internal code generated by the compiler ensures a fast evaluation of boolean expressions and the

current record is pre-processed before evaluation by all the current rules, in order to provide a direct access to its fields.

All reports and conference papers related to the RUSSEL language as well as the whole ASAX package are available from the anonymous ftp site ftp.info.fundp.ac.be/pub/projects/asax.

6 Conclusion

As with all virus detection systems, it is not possible to state that all future viruses will be detected by the system. However, whereas scanner technology requires previous knowledge of the actual virus, VIDES requires only knowledge of the infection strategy. Whereas the number of new viruses averages a few hundred every month, the number of new infection strategies that are significantly different from the point of view of the detection rules average less that one a year.

This is demonstrated by the generic detection rule that was developed using some of the first viruses. The rule, when used on a collection of all known viruses, scored 95% of all viruses that ran in the emulation. This indicates that significant departures from the mainstream infection strategy are rare.

The advanced rules generate enough information to give a rough idea, what type of virus is being analyzed. With further development of the audit system, we hope to get more details of the virus. In particular, indications to the viruses damage routine would be important.

VIDES could conceivably be used outside the virus lab to do detect viruses in a real environment. One possibility is to use it as a type of firewall for programs entering a protected network. Another possibility is the detection of viruses in the DOS–sessions of emerging 32 bit desktop operating systems.

For such a system to be accepted, it must not cause false positives. A concept for this is currently under developement. Such a system must also be unnoticable unless a virus is found. As a virtual 8086 machine will be the basis for this, the only extra overhead will come from the audit system and ASAX. The audit system can be tuned to only provide the data necassary, which eliminates some overhead. ASAX has proven itself very fast, so the only the rules must be tuned for speed.

The prototype VIDES system shows that an automatic analysis system for computer viruses is possible and useful. At the same time, it is a prototype

for the use of intrusion detection technology for desktop systems.

References

[BFHS91] Klaus Brunnstein, Simone Fischer-Hübner, and Morton Swimmer. Concepts of an expert system for computer virus detection. In *Proceedings of the 7th International IFIP TC–11 Conference on Information Security, SEC '91*, May 1991.

[Coh85] Frederick Cohen. *Computer Viruses*. PhD thesis, University of Southern California, December 1985.

[Coh94] Dr. Frederick B. Cohen. *A Short Course on Computer Viruses*. John Wiley & Sons, Inc., 1994. ISBN 0-471-00769-2.

[Den87] Dorothy E. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, 13-2:222, feb 1987.

[Fer92] David Ferbrache. *A Pathology of Computer Viruses*. Springer Verlag, 1992. ISBN 3-540-19610-2.

[HLCM92] N. Habra, B. Le Charlier, and A. Mounji. ASAX: Implementation design of the NADF evaluator. Technical report, Institut d'Informatique, university of Namur, Namur, Belgium, March 1992.

[HLCMM92a] N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX: Software Architecture and Rule-based language for Universal audit trail analysis. In *Proceedings of the third European Symposium on Research in Security (ESORICS'92)*, Lecture Notes in Computer Science, Toulouse, November 1992. Springer-Verlag.

[HLCMM92b] N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu. Preliminary report on ASAX. Technical report, Institut d'Informatique, university of Namur, Namur, Belgium, January 1992.

[Hru92] Jan Hruska. *Computer Viruses and Anti-Virus Warfare*. Ellis Howard Ltd., 2nd edition, 1992. ISBN 0-13-036377-4.

[MLCHZ95] A. Mounji, B. Le Charlier, N. Habra, and D. Zampuniéris. Distributed Audit Trail Analysis. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security (ISOC'95)*, San Diego, California, February 1995. IEEE.

[Mou95] A. Mounji. User Guide for Implementing NADF Adaptors. Technical report, Institut d'Informatique, University of Namur, Namur, Belgium, January 1995.

[Seb] Brian Seborg. Upcoming comp.virus FAQ.

[SK94] Alan Solomon and Tim Kay. *Dr. Solomon's PC Anti–Virus Book*. Newtech, 1994. ISBN 0-7506-16148.

[Swi95] Morton Swimmer. Fortschrittliche Virus–Analyse — Die Benutzung von statischer und dynamischer Programm–Analyse zur Bestimmung von Virus–Charakteristika. Diplomarbeit, University of Hamburg, Germany, Fachbereich Informatik, Arbeitsbereich AGN, 1995.