Behavioural API based Virus Analysis and Detection

Sulaiman Al amro
Software Technology Research Laboratory (STRL)
De Montfort University
Leicester, UK
salamro@dmu.ac.uk

Antonio Cau
Software Technology Research Laboratory (STRL)
De Montfort University
Leicester, UK
acau@dmu.ac.uk

***Abstract*—The growing number of computer viruses and the detection of zero day malware have been the concern for security researchers for a large period of time. Existing antivirus products (AVs) rely on detecting virus signatures which do not provide a full solution to the problems associated with these viruses. The use of logic formulae to model the behaviour of viruses is one of the most encouraging recent developments in virus research, which provides alternatives to classic virus detection methods. To address the limitation of traditional AVs, we proposed a virus detection system based on extracting Application Program Interface (API) calls from virus behaviours. The proposed research uses a temporal logic and behaviour-based detection mechanism to detect viruses at both user and kernel level. Interval Temporal Logic (ITL) will be used for virus specifications, properties and formulae based on the analysis of API calls representing the behaviour of computer viruses.**

Keywords-computer viruses; virus behaviour; API calls; interval temporal logic

I. INTRODUCTION

Since they first appeared, computer viruses have caused disruption to private and public organisations, governments and computer users, as they attempt to remove, modify or steal sensitive data. It is highly recommended that virus researchers should be aware of new trends, which virus writers will exploit whenever they have the opportunity. The success that attackers enjoy demonstrates that there needs to be a novel and robust detection system to prevent attacks. Therefore, a novel system is needed in order to minimise damages caused by these viruses and to defeat the new techniques used by skilful attackers.

Existing antivirus (AV) products provide detection techniques which are based on signatures that have been collected from previous seen viruses and then added to an AV database. Prior to the arriving of a virus to the system, its signature will be compared with those stored in the database and if there is a match, the virus will be detected; otherwise, the system will run normally [1]. Thus, zero day viruses will not be detected by traditional detection systems unless this new virus is received by the antivirus company and the virus signature is stored in its own database. Signature-based detection systems need databases in order to store the signatures. As the number of viruses increases every day, ever larger databases are needed to store all their signatures, so that more storage space will be needed in the near future [3,2, 14]. The large database will also affect the speed of searching for signatures, and, thus, affect the performance of the system. These disadvantages mean that the signature-based detection techniques will soon be inadequate to protect computer systems.

Behaviour-based virus detection systems have been developed recently. They do not rely on a database of signatures, but instead concentrate on the behaviour of the system. They have come to light in order to overcome the problems associated with traditional signature-based detection. The principle behind this approach is first to observe the normal behaviour of the system, after which any deviation from it will be classified as an intrusion [4]. The second is to predefine virus behaviour, so that any process which resembles virus activity can be identified as a potential virus. However, there are difficulties associated with behaviour-based detection, the greatest of which is how to define the behaviours that will detect known and novel viruses without confusing them with normal processes running in the system (known as false positives). In addition, some existing virus behaviour detection techniques rely on detecting subclasses of viruses. In general, behaviour-based detection techniques rely on identifying virus characteristics in order to detect these viruses and other viruses sharing the same characteristics in the future. One of the objectives of this research is to look into the API calls issued by computer viruses in order to specify virus behaviour that will be used in this research.

There is a growing need for behavioural specification to be used in detecting attacks, providing a robust and manageable detection technique [5, 6]. The present research proposes to build a detection technique using temporal logic specifications that have been inferred from the analysis of Windows and native API calls that represent virus behaviours. We believe that using such logical specifications and formulae will minimise the problem of the rapidly growing database of traditional AV products as well as detecting newly released viruses. A logic called Interval Temporal Logic (ITL) has been chosen to be used in this research because this logic is very suitable to describe system traces, i.e., it can be used to describe bad and good behaviours. The Tempura tool will be used to check whether a particular system trace is a good or bad behaviour.

The paper is organised as follows: Section II will provide background and related work to our framework. Section III will explain the behavioural virus analysis including our API extraction mechanism. Section IV will describe how a virus can be detected at both kernel and user level using ITL formulae and Tempura. Section V will present the results.

II. BACKGROUND

It is very important to understand application program interfaces (APIs) and their features, in order to trace the behaviour of programs and to understand hidden features of malicious codes. Therefore, an outline of API calls is provided here in order to enhance understanding of this important system service.

A. Windows Application Program Interface (API) system calls

In 1995, Microsoft released Windows 95 and at the same time introduced a set of system calls known as Win32 API, which represented a 32-bit application program interface [10, 1]. The new APIs had the advantage of higher system speeds because they provided a set of optimised system operations [11]. User applications in the Windows operating system (OS) based on these API function calls are stored in dynamic link libraries (dlls) such as User32.ddl, Kernel32.dll, Advqpi.dll, and Gui32.dll, in order to gain access to system resources involving registry and network information, processes and files. Each Win32 API call has its own memory address place in the import address table (IAT) which every process in the system has and which each process will consult when it makes an API call. A Win32 API call is normally called from a process running at the user level [12], then the called API will be handled by the system and converted to its equivalent function, known as a native API call, which will be understood by the kernel of the OS. A service in the kernel will handle the requested operation and its outcome will return to the original user application that made the call [7].

The majority of systems services run in the kernel and need privileges in order to access it. Native API calls, which can be directly called by any process at the kernel level, are dealt with in the dynamic link library (ntdll.dll) in order to have the kernel provide the requester service. The complete list of kernel mode functions is stored by memory location addresses in the system service dispatch table (SSDT), which is accessed each time a native API routine is called. The parameters are then passed to the memory location and the function continues with its execution [12, 13].

As explained by [18], Windows API calls play an important role in exploiting the power of Windows, allowing virus writers to use API calls to gain more security privileges and perform malicious actions. Windows APIs issue calls to perform several actions, such as user interfaces, system services and network connections, which can be utilised for good or evil [7]. Because API calls will give a full and complete description of a particular program, the analysis of its API calls will lead directly to the understanding of its behaviour.

B. Related work

Skormin et al [8] have designed an approach that intercepts API calls while a program is running. They detect any attempt of a malware to self-replicate at run-time. Their methodology was to trace the behaviour of normal processes and analyse API calls along with their input, output arguments and the execution results. The replication of a process was modelled by the Gene of Self Replication (GSR) based upon building blocks. Each block in the GSR is considered as a portion of the self-replication process which includes seeking for files and directories, writing to files, reading from files, and closing and opening a file. This approach has detected several viruses from different classes but on the other hand, they used to hook native API calls only in the kernel. As said by [24, 13] native APIs are not fully documented that gives some viruses the ability to use some of these undocumented API to attack the system.

Alazab et al [7] have used a static analysis to track API calls using existing tools. They analyze malware to classify program executable as normal or malicious. They have used the IDA Pro [22] with their own Python program to automatically extract API calls. They had examined six groups of virus steps such as search, copy, delete, read and write. They have found that read and write files were the most API calls used by malwares to infect the program. Lists of Win32 API calls have been extracted at the user level. However, there are some viruses that might not be detected by [7] because they directly call the kernel by using native API calls as mentioned by [12, 18].

Veeramani and Rai [15] have used a statistical analysis for Windows API calls to describe the behaviour of programs. They used an automated framework for analysing and categorising executables rely on their relevant API calls. They try to increase the detection rate by using Document Class wise Frequency feature selection (DCFS) measure by getting the information related to malware from the extracted API calls. They have categorised malware into groups and the relevant APIs were extracted from these categories. DCFS based feature selection measure is used to classify the executable as malicious or benign. Their analysis and detection have been done at the user level leaving the system liable to viruses that can directly contact the kernel.

III. VIRUS BEHAVIOUR ANALYSIS

Figure 1 shows the mechanism used to analyse and extract API calls. Existing software was used to obtain information about the viruses through the following steps:

Step one: Unpack the virus.

Step two: Get the assembly code by disassembling the virus.

Step three: Extract the sequence of API calls that represent the virus behaviour.

A. Build a secure environment

In order to analyse computer viruses, a secure environment is needed to make sure that no virus can escape the system and infect other machines. In addition, some viruses will use the Internet or a local area network (LAN) to spread their malicious effects, allowing them to spread very widely indeed. Therefore, a virtual machine (VM) (Oracle VM VirtualBox) [11] was used in this research in order to secure the system. The Linux Ubuntu operating system was used as host with Windows XP as a guest to ensure that no viruses leaked from the guest to the host, because a virus that infects one OS will not run when a different OS platform is used [1]. In some

Identify applicable sponsor/s here. *(sponsors)*

cases, viruses will use the Internet to connect to anonymous remote hosts. It is preferable not to connect to these unknown hosts, even if the virus is running on a virtual machine, so a way to prevent this is needed. However, the behaviour of viruses is the target and the Internet plays an important role in tracking these behaviours. Therefore, a fake Internet was used, allowing all the network activities in addition to allowing the tracking of virus behaviour in this research. This was achieved without causing any risk to the real Internet by installing NetKit [17], which provides a simulation of the entire Internet. NetKit was therefore installed on the host (Ubuntu) machine and then the virtual machine ran Windows XP using the fake internet.

B. Unpacking the virus

Packers are known as "anti-anti-virus" programs and also can be called "anti-reversing", because they exist to fight against anti-virus software as well as reverse engineering techniques. Packers are mostly used to disguise and/or compress codes. According to [18], packers are just computer programs which have the ability to restore the original executable image of a file from its encrypted and compressed one in a secondary memory location. Hence, the code might appear to do one thing, but it actually does something else, which is likely to confuse researchers.

Nowadays, computer virus writers have the benefit of using these packers to make their viruses run faster, as well as avoiding detection systems. Furthermore, the methods of packing make recognising and understanding viruses very complicated both for detection systems and analysts, because the authors can make small code modifications in order to change a signature and so avoid detection. Packing also makes analysis by researchers less easy, because to extract and understand unpacked code requires a third party tool, beside a deep and strong understanding of assembly language and the kernel, which leads to a better understanding of low level programming [7].

However, a number of researchers have reported the construction of tools that automatically unpack viruses such as Eureka [19], Ether [20] and Renovo [21]. The present research uses PEiD [22] to unpack the virus samples examined. PEiD is an unpacking tool that detects most common packers, cryptors and compilers for Portable Executable (PE) files. The first step was to use an interactive disassembler, IDA Pro [22], to decide whether a virus was packed or not, after which PEiD was used to indicate which packer (e.g. UPX, Upack, Xpack or PEPack) had been used. As Figure 1 shows, OllyDbg [23] was used to seek the entry point of the virus and to dump the unnecessary code. It would also save the newly unpacked virus in order to conduct a clear investigation of the malware. Our analysis shows that approximately 70% of the viruses analysed had been packed and needed to be unpacked, using the process explained above, while the remaining 30% were directly observed.

C. Extracting the assembly code

IDA Pro can be used to extract the assembly code from both executable (such as PE, ELF, EXE, etc.) and non-executable files and is the most practical disassembly tool [7]. It runs a static analysis and can detect whether a file is packed, as well as disassembling the code, thus providing more details and improving the understanding of the code.

IDA Pro [22] was selected as part of the API extraction mechanism used in this research because it can statically and automatically extract API calls from a file, giving an initial image of what sort of API calls the file might make. Thus, using IDA Pro allows API calls to be statically extracted and gathered, offering an important method of identifying virus behaviour. In order have more evidence about the API calls made by viruses, IDA Pro needs to be used with more than one tool that provide tracking API calls at runtime.

D. Extracting API calls

Viruses are just like normal programs and can be distinguished by tracking their API calls that lead to malicious actions. Therefore, this research concentrates on tracing API calls in order to understand virus behaviour. As shown in Figure 1, more than one tool [22, 25, and 26] was utilised to trace API calls in static and runtime environments. Most researchers [7, 8] rely on just one tool, which runs either statically or dynamically, but this research uses both in order to have a full understanding of what API calls have been made and when. Static analysis misses some API calls when comparing to dynamic analysis. In addition, there some Win32 and native API that appear in [25] but not in [26] and vice versa. Thus, these three tools have been used in this research to track API calls.

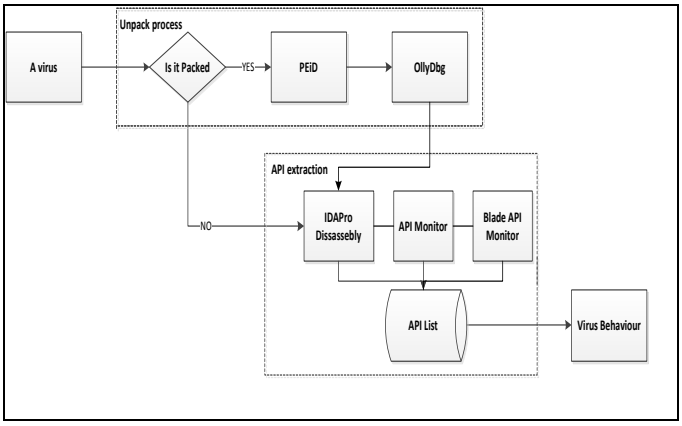

Figure 1. API extraction mechanism

The present research considers API calls to provide a way to determine whether malicious actions have been performed or not, by analysing them to understand their behaviour and to indicate whether a file contains a malicious or benign program. To do this, 283 virus samples downloaded from [29, 30] and 50 Window (XP) normal processes, such as svchost.exe and iexplore.exe, were examined to discover what sort of API calls malicious programs use in order to perform their actions.

The research began with the assumption that a virus must read from and write to a file, as [9, 8, 7] explain, in order to infect a computer, to replicate itself, to infect other files and to spread throughout the world. More precisely, the following five steps are considered to represent virus steps in a behaviour:

1) Find to infect: In order to infect, a virus needs to find a file or to retrieve the contents of a directory in which to write its malicious code. This research has concentrated on three types of computer virus, listed in Table I: those which overwrite existing files, known as overwriting viruses, those which can be attached to existing files, known as parasitic viruses, and those which create a file resembling a known one, known as companion viruses.

After analysing the API calls issued by a group of computer viruses related to the three types explained above, it has been considered that 'find to infect' as the first step in the behaviour, addressing its potential API function calls that relate a search to a particular file or directory.

TABLE I. VIRUS DESCRIPTIONS

Virus type	Description	Behaviour
Overwriting	A virus (V) will replace its content with an existing file (F) by overwriting it.	Read "V.exe" Open "F.exe" Write"V.exe"into "F.exe" Close ""V.exe""
Parasitic	A virus (V) will attach itself to an existing file (F) by injecting its code into F and replace its entry points.	Open "V.exe" Read "V.exe" code Open "F.exe" Inject code into "F.exe" Replace"F.exe"entry point
Companion	A virus (V) will change the name of an existing file (F) with its original name.	Read "F.exe" Rename "F.exe" as "F.ex" Rename "V.exe" as "F.exe"

2) Get information: The second category of steps in a virus behaviour observed in this research was to discover a file's attributes, to retrieve specific information regarding a file, or to retrieve information on a directory, such as path name. A virus needs to have information about a particular file or directory to infect it and to read and write to it.

3) Read and/or copy: Read and write calls are the most important API calls issued by viruses, because they give it the ability to replicate and spread. As explained by [9], there is a very narrow difference between normal and malicious behaviour in the case of system calls. Indeed, although this research has given careful consideration to distinguishing between normal and abnormal activity, there exist some legitimate processes that may look like malicious software but would never be captured by the detector used here, because they will never act exactly the same as the malware, i.e. there is always a difference, however slight. Pervious researches such as [8] and [9] have observed that normal processes will never issue system calls that have the same order as computer viruses. This means that our concept of virus behaviour has to trace system calls from the beginning to the end, having a set of system calls which have to be made in order, because normal processes are supposed never to follow the concept of replication completely.

Therefore, read and write function calls must be made in a certain order, i.e. a virus will read a file first and then write to this or another file. In addition, other previously observed API calls of another category may or may not be called, but when it comes to read and write API calls, they must be called by the file for it to be considered a virus. Copy API calls are considered to be malicious here, because some viruses will copy to or from files when they infect a system [7]. The use of 'and/or' in the category name means that a copy API call may or may not be issued by a virus.

TABLE II. API FUNCTION CALLS FOR CATEGORIES OF STEPS

Step	Virus category	API Function Calls
First	Find to infect	FindFirstStream, FindFirstFileTransacted, FindFirstStreamTransactedW, FindFirstStreamW, FindClose, FindNextFile, FindFirstName, FindFirstFileEx, FindFirstFile, FindFirstNameW, FindNextFileName, FindNextFileNameW, FindFirstNameTransactedW, FindNextStreamW, FindNextStream.
Second	Get information	GetFileAttributesEx, GetFileAttributesTransacted, GetFileAttributes, GetFileInformationByHandle, GetFileBandwidthReservation, GetCompressedFileSizeTransacted, GetFileInformationByHandleEx, GetCompressedFileSize, GetBinaryType, GetFileSizeEx, GetFileSize, GetFileType, GetTempFileName, GetTemppath, GetFinalPathNameByHanle, GetLongPathNameTransacte,d GetFullPthNameTransacted, GetFullPthName, GetLongPathName, GetShortPathName.
Third	Read and/or copy	ReadFile, ReadFileW, OpenFile, OpenFileByld, ReopenFile, CreateHardLinkTransacted, CreateHardLink,CreateSymbolicLink Transacted, CreateSymbolicLink, CopyFileEx , CopyFile, CreateFileW, CreateFile, CopyFileTransacted , CreateFileTransacted.
Fourth	Write and/or delete	ReplaceFile, WriteFile, DeleteFileTransacted, DeleteFileW, DeleteFile, CloseHandle.
Fifth	Set information	SetFileInformationByHandle, SetFileValidData, SetFileBandwidthReservation,, SetFileShortName, SetFileAttributesTransacted, SetFileApisToOEM, SetFileAttributes, SetFileApisToANSI.

4) Write and/or delete: As mentioned in the previous subsection, a file must issue write API calls in order to be classified as a virus. Therefore, every read API call should be followed by a write API call, issued at any time by the same file, to be considered a virus and not to conflict with benign processes. However, as with 'copy', the delete API call is

considered malicious, because some viruses will delete some files when they infect a system, as reported by [7]. It will also be optional, as the phrase and/or appears in the category name; that is, the API delete call may or may not be issued by a virus.

5) Set information: The last category of steps in a virus behaviour observed in the research is the setting of specific information regarding a file, which leads to a change in its attributes. It has been observed that after infecting a file, a virus will need to change some of the file information in order to deal with it in the future.

Therefore, five categories of steps in representative virus behaviours, reiterated in Table II, were observed in this research and were compared with API calls to determine whether a virus was present, as explained in Figure 2.

The previous five categories were found to have used API calls that could be called by a file at the user level, known as Win32 APIs. However, there is an alternative, whereby native API calls perform this function in order to provide the service requested by the kernel. Win32 APIs are converted to native API calls by the ntdll.dll process [7, 9], in order to be understood by the kernel. For example, when a file issues a CreateFile API call, the ntdll.dll will convert it in to its native API call, NtCreateFile, then redirect it to the kernel. However, there exist some files that can call the kernel directly, avoiding the need for user level API calls [18]. These calls were observed in this research.

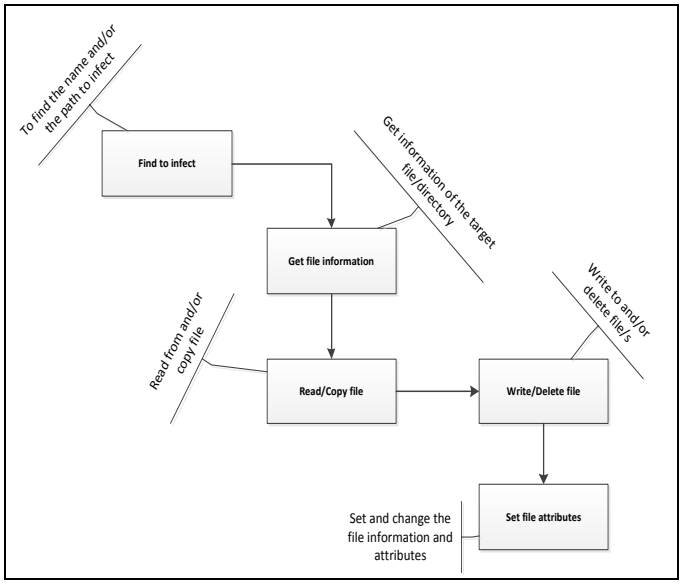

Figure 2. Five Categories

Table III shows the native API calls that can be issued by a file in order to be classified as a virus. However, these native API calls are not fully documented, as the Microsoft does not make them available [24], so API call researchers are struggling to acquire more knowledge about them. Therefore, both native and Win32 API calls need to be observed and taken into account in order for the present research to achieve good results.

TABLE III. NATIVE API FUNCTION CALLS FOR CATEGORIES OF STEPS

Step	Virus category	Native API calls
First	Find to infect	NtQueryDirectoryFile.
Second	Get information	NtQueryAttributesFile, NtQueryInformationFile.
Third	Read and/or copy	NtOpenFile, NtReadFile, NtCreateFile.
Fourth	Write and/or delete	NtWriteFile, NtDeleteFile, NtClose.
Fifth	Set information	NtSetInformationFile.

IV. VIRUS DETECTION

The observation of steps in a virus behaviour used in our system is based on the API hooking method at runtime. Hooking APIs provides the ability to intercept a set of API calls and redirect them to other functions [27, 9]. The benefit of doing so is to examine these calls in order to decide whether a virus is present or not. API hooking is done in either user or kernel mode.

A. User mode API

User mode API hooking, based on the technique of altering the IAT, redirects API calls to another place. However, Tempura will receive the API calls in order to decide whether a virus is present, as shown in Figure 3. If a virus is detected, the system will not allow it to continue making API calls and the file is terminated.

When a prototype was run in user mode only, the virus detection rate was low, because viruses are designed to evade the detection used at the user level [11]. Therefore, if no virus was detected, it was directed to the second approach and the kernel native API calls were examined.

B. Kernel mode API

The majority of computer viruses try to run at kernel level in order to gain more security levels and control of the system, which cannot be gained at the user level. At the kernel level, native API hooking does not differ from the user level, at which the SSDT can be overwritten and redefined. Therefore, any native API calls will be received at a runtime by Tempura, where they are examined for a virus.

If the user level fails to detect any suspicious API calls issued by a file, it is directed to the kernel level for further examination. If the native API calls indicate that it is suspicious (i.e. a virus), it will be terminated, while if no suspicious steps in a behaviour are detected, both API and native API calls are returned to their original file.

The disadvantage of examining API calls only in the user level is that some processes will directly contact the kernel and avoid using Win32 APIs [9, 12], allowing them to remain undetected. On the other hand, the drawback with using kernel level by itself is that unlike system calls, native APIs are not completely documented and are almost entirely hidden from view, with only handful of their functions documented in generally accessible publications [18]. This drawback makes the use of native APIs incomplete and liable to both false negatives and false positives, so that the system is not fully protected.

Therefore, it can be hypothesised that the use of a combined user and kernel level approach provides a better detection system and minimises the rates of false negatives and false positives. Such a system is able to examine API calls issued in the user mode and if a file is detected as a virus, no further examination is needed. If, however, it is not considered to be a virus, the detection system will examine it at the kernel level by observing its native API calls.

In order to apply this approach, a parallel execution tool is needed to run user and kernel level detection simultaneously. ITL can do this, handling both sequential and parallel composition [16] and offering user and kernel level detection at the same time. We can also make the native API calls used at the kernel level adaptable by using ITL formulae and this allows us to add more native API calls in the future.

Figure 3 shows how the system works. At the user level, API calls are extracted and then sent to AnaTempura, which examines them to see if they match the five categories of steps in a virus behaviour.

However, if the five categories are not detected in the user level API calls, Tempura examines the native API calls coming from the kernel level. This comparison is similar to the above, but concerns only those categories which have not been detected. For example, if three of the five are discovered in the first comparison, the second one considers only the undiscovered categories. Then, if kernel observation completes the set of five categories, Tempura decides that a virus has been detected and the file will not complete execution.

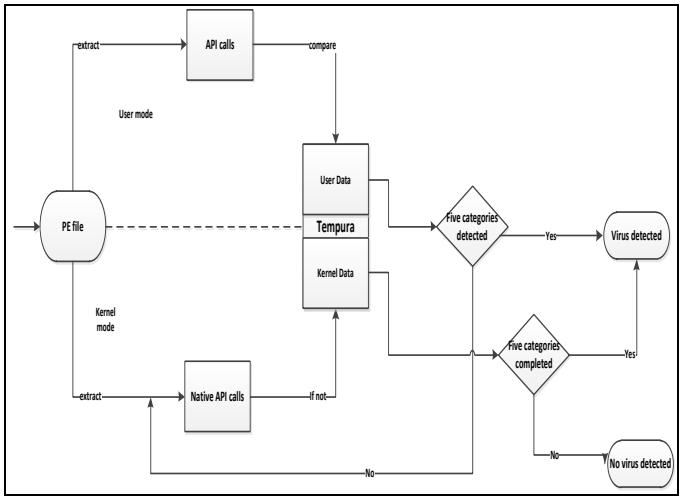

Figure 3. Virus Detection Flowchart

C. Interval Temporal Logic

ITL will be used in this research and our choice of this logic is inspired by the existence of Tempura, an executable subset of ITL that enables our virus behaviour specification to be checkable [16]. In addition, ITL is very suitable to describe system traces, i.e., it can be used to describe bad and good behaviours.

ITL is a temporal logic whose key feature is its intervals, each of which must be a non-empty, finite or infinite sequence of states σ1, σ2… A state is mapping from a set of variables *Var* to a set of values *Val*. ITL is known as a linear-time temporal logic for both infinite and finite intervals with a discrete model of time. In the finite interval it has a finite number of states and the length $|\sigma|$ of an interval is the number of these states minus one. However, a one-state interval, which is known as an empty interval, has the length zero. The sequences of states from a given system can be represented as the behaviour of this system. All behaviours of a system can be represented as the specifications of this system that can be demoted by ITL formulae using the ITL syntax and semantics.

The syntax of ITL is described in Table IV, in which μ is an integer value, a is a static variable that does not change within an interval, A is a state variable that can change within an interval, v is a static or state variable, g is a function symbol and p is a predicate symbol.

TABLE IV. THE SYNTAX OF ITL

Expressions
$e ::= \quad \mu \mid a \mid A \mid g(exp_1, \ldots, exp_n) \mid \textit{ıa}\colon f$
Formulae
$f ::= \quad p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \ \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$

The constant μ is a function without a parameter which has a fixed value, such as true, false, 1, 5. A static variable is one whose value remains unchanged in all cases within an interval (known as a global variable). On the other hand, a state variable is one that can change within an interval (known as a local variable). A function symbol can be one of several operators such as, +, -, and * (multiplication), etc. An expression of the form $\textit{ıa}\colon f$ is called a temporal expression. Relation symbols such as = and ≤ are used to construct atomic formulae, which will then be composed with first order connectives such as ¬, ∀ and ∃ and with skip, chop, and chopstar, which are known as temporal modalities.

D. *Informal* semantics

The beginning of an interval evaluates expressions and formulae in ITL. If there are no temporal operators in a formula, it is called a state formula. A state formula within an interval is required to hold only at the initial state of that interval. The informal semantics of the most interesting temporal constructs are defined as follows [28]:

- *skip*: is a unit interval that has a length equal to 1.
- $$skip: \ \bullet \sigma 0 \bullet \sigma 1$$

Here is a two-state Interval that has the length of 1.

- The formula $f_1; f_2$ is known to be true within an interval if it can be decomposed (chopped) into two parts, a prefix and suffix interval, such that $f1$ holds for the former and f_2 for the latter, or if the interval is infinite and f_1 holds for that interval.

Figure 4. Chop

- The formula f^* which holds for an interval is true over this interval if it can be decomposed to a finite number of intervals and the subformula is true in each of these chopped intervals.

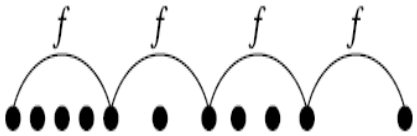

Figure 5. Chopstar

E. Derived constructs

The following constructs will be used frequently. Non-temporal derived constructs are listed in Table V and temporal derived constructs in Table VI.

TABLE V. NON-TEMPORAL DERIVED CONSTRUCTS

true	$\triangleq$ $0 = 0$	true value
false	$\triangleq$ $\neg$ true	false value
$f1 \vee f2$	$\triangleq$ $\neg(\neg f1 \wedge \neg f2)$	or
$f1 \supset f2$	$\triangleq$ $\neg f1 \vee f2$	implies
$f1 \equiv f2$	$\triangleq$ $(f1 \supset f2) \wedge (f2 \supset f1)$	is equivalent to
$\exists v \,.\, f$	$\triangleq$ $\neg\forall v \,.\, \neg f$	exists

TABLE VI. TEMPORAL DERIVED CONSTRUCTS

O f	$\triangleq$ skip ; f	next
more	$\triangleq$ Otrue	non-empty interval
empty	$\triangleq$ $\neg$ more	empty interval
inf	$\triangleq$ true ; false	infinite interval
isinf (f)	$\triangleq$ inf $\wedge$ f	is infinite
finite	$\triangleq$ $\neg$ inf	finite interval
isfin (f)	$\triangleq$ finite $\wedge$ f	is finite
fmore	$\triangleq$ more $\wedge$ finite	non-empty finite interval
$\Diamond f$	$\triangleq$ finite ; f	sometimes
$\Box f$	$\triangleq$ $\neg \Diamond \neg f$	always
$\textcircled{w} f$	$\triangleq$ $\neg$ O $\neg f$	weak next
$\langle\!\rangle f$	$\triangleq$ f ; true	some initial subinterval
$\boxed{i} f$	$\triangleq$ $\neg(\langle\!\rangle \neg f)$	all initial subinterval
$\textcircled{a} f$	$\triangleq$ finite; f ; true	some subinterval
$\boxed{a} f$	$\triangleq$ $\neg(\textcircled{a} \neg f)$	all subinterval

For more information on ITL syntax, semantics, derived constructs, Tempura and examples, we refer the reader to our previous paper [14], [28] and [16].

F. ITL formulae

We have declare Cat1, Cat2, Cat3, Cat4, and Cat5 which respectively represent the lists of all API function calls for Find to infect, Get Information, Read and/or Copy, Write and/or Delete, and Set Information, as listed in Table II.

We suppose that X represents all API calls which received at runtime by Tempura. X will be received as a text representing all API calls issued by a certain PE file.

The ITL formulae for Cat1 will be as follow

$$Ucat1(X) =$$

$$in\ Usermode\ (X) \wedge in\ Cat1(X). \quad (1)$$

This formula indicates that if one or more APIs calls denoted by X issued by a file in the user level, is in the list of Cat1.

The previous formula will be applicable for all the categories in the user level except Cat3 and Cat4 that represent the read and write categories respectively.

However, several rules and conditions should be considered in this research. Firstly, in order to write to an existing or new file, a virus will read and write in order, i.e. will read first and then write to the infected file. Secondly, one of the API calls (*ReadFile, ReadFileW, OpenFile, OpenFileByld, and ReopenFile*) must be called in the third category and *WriteFile* API call must be called in the fourth category.

Therefore the formula of Cat3 will be as follow

$$Ucat3(X) =$$

$$in\ Usermode(X) \wedge in\ Cat3(X) \wedge in\ Read(X). \quad (2)$$

Where Read = *(*ReadFile, ReadFileW, OpenFile, OpenFileByld, ReopenFile).

The formula for Cat4 will be

$$Ucat4(X) =$$

$$in\ Usermode(X) \wedge in\ Cat4(X) \wedge X = \text{“}WriteFile\text{”}. \quad (3)$$

Because the order of read and write is very important in this research, the next formula will be applicable.

$$\Diamond Ucat3(X)\ ;\ \Diamond Ucat4(X). \quad (4)$$

It shows that the write calls must be issued sometimes ($\Diamond$) after a read call.

However, if one or more categories are not detected in the user level, then their native API calls coming from the kernel will be examined. Therefore the next formula will be used:

$$\Box(\text{- } Ucat1(X) \vee$$
$$\text{- } Ucat2(X) \vee$$
$$\text{- } Ucat3(X) \vee$$
$$\text{- } Ucat4(X) \vee$$
$$\text{- } Ucat5(X) \equiv In\ kernel) \quad (5)$$

The previous formula indicates that if one or more categories have not been detected in the user level, they will have more examination to the kernel, in order to see if there is a call belongs to the undetected category that has been directly

issued to the kernel. The same mechanism will be used to examine the native API calls coming from the kernel. Therefore, kernel level formulae will be the same as the user ones but with different predicates and variables.

V. RESULTS

30 viruses of different classes namely, Overwriting, Parasitic and Companion viruses have been observed in this experiment. We found that 29 of them can be detected by our approach. 23 of them can be detected at the user level, where 6 viruses can be detected in the kernel mode and 1 remain undetected, as shown in Table VII.

TABLE VII. DETECTED VIRUSES

Detected status	Number of viruses
User Level	(23) viruses
Kernel Level	(6) viruses
Not Detected	(1) virus

The results obtained by our research indicate that the rate of virus detection can be increased by 20%, if the two levels are examined. It also indicates that examining only the user level will leave 23% of viruses as undetected. Therefore, using a hybrid system of kernel and user level will increase the detection rate to 97%, as shown in Figure 6.

Unknown viruses will be detected by this system, if they have the similar sequence of steps of the mentioned categories. Due to the fact that this system has no database, it will not consume memory as traditional AVs. Therefore, detecting previously unseen viruses as well as minimising the memory consumption been obtained by this research.

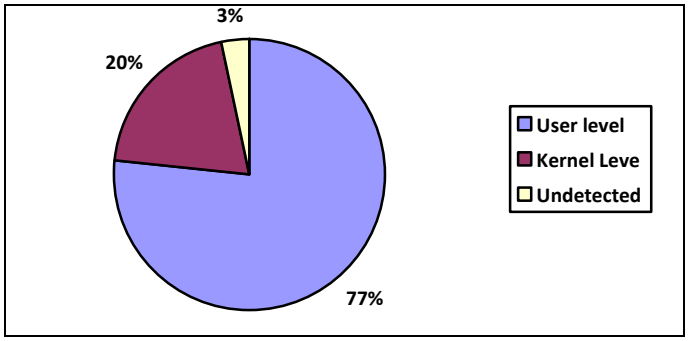

Figure 6. Virus Detection Percentage

VI. CONCLUSION AND FUTURE WORK

Most approaches that use API calls to detect computer viruses operate at either user level (Win32 APIs) or kernel level (native APIs). The problem with the former is that some applications can directly call the kernel and avoid using Win32 APIs, allowing them to remain undetected, i.e. this approach tends to give false negatives. We presented a method of detecting computer viruses in both user and kernel level by using Interval Temporal Logic. The two approaches have been used to increase the detection rate of viruses. The results indicate that the rate of virus detection can be increased by 20%, if both levels have been used. In addition to detect zero day viruses, this paper has provided a faster system by minimising the memory consumption because no database has been used in this research.

We believe that some false positives and negatives might appear when examining more and more viruses because of the exact time of switching from the user to kernel level. Therefore, our future work is to develop this research by randomly checking both levels at the same time.

REFERENCES

[1] P. Szor, The Art of Computer Virus Research and Defense, Addison-Wesley, 2005.

[2] W. Britt, S. Gopalaswamy, J. Hamilton, J. Dozier, and S. Tenaglia, "Computer Defense Using Artificial Intelligence, " Proc. The 2007 spring simulation multiconference, Vol. III, ACM, pp. 378-386, June 2007.

[3] P. Harmer, P.Williams, G. Gunsch, and G. Lamont, "An Artificial Immune System Architecture for Computer Security Applications, " IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, Vol. 66, IEEE Press, pp. 252-280, June 2002.

[4] M. Davis, S. Bodmer, and A. LeMasters, Hacking Exposed: Malware & Rootkits, McGraw-Hill, 2010.

[5] E. Nowicka, and M. Zawada, "Modeling Temporal Properties of Multi-event Attack Signatures in Interval Temporal Logic, " Proc. IEEE / IST Workshop on Monitoring, Attack Detection and Mitigation, CiteseerX, , pp. 378-386, Sep. 2006.

[6] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, " Detecting Malicious Code by Model Checking, " Proc. International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05), vol. 3548, Springer, pp. 515-527, July 2005.

[7] M. Alazab, S. Venkataraman, and P. Watters, "Towards understanding malware behaviour by the extraction of API calls", IEEE 2nd Cybercrime and Trustworthy Computing Workshop (CTC-2010), pp. 52-59, July 2010.

[8] A. Skormin, D. Volynkin, I. Summerville, and J. Moronski, "Run-Time Detection of Malicious Self-Replication in Binary Executables" Journal of Computer Security, vol. 15, no. 3, pp. 273-301, 2007.

[9] C. Seifert, R. Steenson, I. Welch, P. Komisarczuk, and B. Endicott-Popovsky, "Capture - A Behavioral Analysis Tool for Applications and Documents" 7th Annual Digital Forensic Research Workshop (DFRWS), pp. 23-30, 2007.

[10] Windows API reference. http://msdn2.microsoft.com/en-us/library/aa383749.aspx., 2011.

[11] J. Morales, "A Behavior Based Approach to Virus Detection". PhD thesis, Florida International University.

[12] G. Hoglund and J. Butler. Rootkits: subverting the Windows Kernel. Addison Wesley Professional, 2005.

[13] R. Vieler. Professional Rootkits. Wrox Press, 2007.

[14] S. Al amro, A. Cau, "Behaviour-based virus detection system using Interval Temporal Logic", Proceedings of the 6th IEEE International Conference on Risks and Security of Internet and Systems, pp.1-6, Sept. 2011

[15] R. Veeramani, and N. Rai, Windows API based Malware Detection and Framework Analysis. In: International Journal of Scientific & Engineering Research (IJSER), vol. 3, no. 3, March 2012

[16] B. Moszkowski. "Some very compositional temporal properties. In E.-R. Olderog, editor, Programming Concepts, Methods and Calculi, volume A-56 of IFIP Transactions, IFIP, Elsevier Science B.V. (North–Holland), pp.307–326, 1994.

[17] NetKit. http://wiki.netkit.org, 2011.

[18] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis", Computer Security -

ESORICS, Lecture Notes in Computer Science LNCS, Springer, pp. 481-500, 2008.

[19] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. "Ether: Malwareanalysis via hardware virtualization extensions", Proceedings of the15th ACM conference on Computer and communications security, pp.51-62, 2008.

[20] M. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables", Workshop on Rapid MalcodeWORM'07 Proceedings of the 2007 ACM workshop on Recurring malcode, pp.46-53, 2007.

[21] PEiD. http://www.peid.info/, 2011.

[22] IDA Pro Dissasember .DataRescue, An Advanced Interactive Multi-processor Disassembler, http://www.datarescue.com, 2011.

[23] O. Yuschuk, OllyDbg v1.1: 32-bit assembler level analysing debugger for Microsoft Windows. http://www.ollydbg.de. 2004.

[24] M. Russinovich. Inside the Native API. http://www.sysinternals.com/Information/NativeApi.html, 2005.

[25] API Monitor. http://www.rohitab.com/apimonitor, 2011.

[26] Blade API Monitor. http://www.bladeapimonitor.com/, 2011.

[27] J. Morales, P. Clarke, and Y. Deng, "Identification of file infecting viruses through detection of self-reference replication" Journal of Computer Virology, vol. 6, no.2, Springer. pp. 161-180, 2010.

[28] A. Cau, B. Moszkowski, and H. Zedan, "Interval Temporal Logic", Software Technology Research Laboratory, De Montfort University, http://www.cse.dmu.ac.uk/STRL/ITL, 2007.

[29] VX Heavens (http://vx.netlux.org), 2011.

[30] Offensive Computing. http://www.offensivecomputing.net, 2011.

AUTHORS PROFILE

Authors Profile …