J Comput Virol (2010) 6:161–180
DOI 10.1007/s11416-008-0101-5

EICAR 2008 EXTENDED VERSION

Identification of file infecting viruses through detection of self-reference replication

Jose Andre Morales · Peter J. Clarke · Yi Deng

Received: 20 January 2008 / Revised: 27 June 2008 / Accepted: 6 July 2008 / Published online: 26 July 2008

Abstract This paper presents an approach to detecting known and unknown file infecting viruses based on their attempt to replicate. The approach does not require any prior knowledge about previously discovered viruses. Detection is accomplished at runtime by monitoring currently executing processes attempting to replicate. Replication is the fundamental characteristic of a virus and is consistently present in all viruses making this approach applicable to viruses belonging to many classes and executing under several conditions. An implementation prototype of our detection approach called SRRAT is created and tested on the Microsoft Windows operating systems focusing on the tracking of user mode Win32 API system calls and Kernel mode system services.

1 Introduction

Current virus detection is primarily based on the use of a signature database. This approach is most effective in detecting previously discovered viruses. Unfortunately, this approach does not work well in detecting newly released unknown viruses. Behavior based detection is a more effective approach in detecting unknown viruses. The principle drawback to behavior based detection is the high production of false positives. Despite this drawback, behavior based detection is the most promising approach to detecting newly released unknown viruses. Several behavior based detection models can be found in the literature [1, Chap. 4], [2, Chap. 11] and [3, Chap. 5]. The detection methodology of each of these models are normally based on identifying a specific set of one or more characteristics found in a previously discovered virus or viruses. These characteristics are present in some but not all viruses. This results in a successful detection capacity that is limited to a specific class of virus or under specific execution conditions. Identifying a characteristic that is consistently present in many viruses can lead to successful virus detection in several classes and under many different execution conditions.

Replication is the fundamental qualifying characteristic of all viruses [2, Chap. 1], [3, p. 7–78] and [4, p. 163]. For a specific malware to be classified as a virus it must have the ability to replicate. This guarantees the replication characteristic is consistently present in all viruses. Replication is therefore an excellent basis for detection algorithms to successfully detect viruses under several conditions and that belong to many different classes [5]. When a virus replicates, it will execute a series of operations that will cause the virus to be written to some other area of the target system. The virus can infect one or more currently existing files and infect the system by copying itself to newly created target files. Both of these infection types require a series of read and write operations to succeed.

Self-reference is an essential property of the read and write operations executed by a virus during replication. A virus must refer to itself in order to replicate itself to some other area of the target system. The term “itself” refers to the static image of the virus file saved on a storage device such as a hard drive. The name of the virus file is the same as the name of the executing virus process. This name is passed between read and write operations as the source or

J. A. Morales (✉) · P. J. Clarke · Y. Deng
School of Computing and Information Sciences,
Florida International University, Miami, FL 33199, USA
e-mail: jmora009@cis.fiu.edu; jose@joemango.com

P. J. Clarke
e-mail: clarkep@cis.fiu.edu

Y. Deng
e-mail: deng@cis.fiu.edu

"from" argument of the replication. We name this property the self-reference property (SR) and replication that occurs using SR we identify as SR-replication.[1] SR is the focus of this research and SR-replication is the centerpiece of our behavior based virus detection approach. We present a detection approach for SR-replication that is based on SR which focuses on the transitive relation between a running virus and a target file. The approach is tested in a real-time scenario with a runtime monitoring implementation prototype called SRRAT which focuses on user mode Win32 API system calls and kernel mode system services. We assume that by detecting SR-replication we can detect both known and unknown file infecting viruses belonging to different virus classes and that execute under several conditions. We further assume SR-replication to be unique to viruses and that it is unlikely for SR-replication to occur in benign processes. We do recognize that not all viruses will replicate using SR-replication and these viruses may not be detected by our approach.

The contributions of this paper are:

(1) A detection technique for file infecting viruses based on SR-replication.
(2) Ability to detect viruses with no prior knowledge of any specific virus allowing for detection of both known and unknown viruses.

Using SR-replication for virus detection does not require any preliminary training or analysis of known viruses, this is an improvement from previous work [5] which required extensive preliminary training before running in a real time system. The focus of the detection being on one specific characteristic allows for thorough formalization that precisely articulates the representation of SR-replication. This leads to more accurate detection by eliminating possible ambiguities during implementation. The elimination of preliminary analysis and knowledge of previously discovered viruses results in a detection algorithm capable of true dynamic behavior detection of viruses with no dependencies and no updates while being capable of detection at the moment the virus executes eliminating further infection and possible system damage.

The remainder of this paper is as follows: Sects. 2 and 3 is background and motivation for this research. Section 4 presents formal definitions for SR and SR-replication and the detection approach. Sections 5, 6, 7, 8, 9, 10, 11 and 12 describe our runtime monitor prototype SRRAT and the testing results. Section 13 is related work and lastly Sect. 14 is conclusion and our future work.

[1] patent pending.

2 Background

The fundamental virus models [4, p. 163] and [6] explicitly define virus replication. Cohen provides the seminal results using Turning Machines to illustrate virus replication as symbols on a tape transferred from one segment of the tape to another segment of the same tape. During the transfer of symbols, the virus refers to itself on read operations one symbol at a time followed by a write operation of the just read symbol which illustrates SR-replication. Adleman defined infection as virus replication using recursive functions. Von Neumann created a self reproducing automata showing that replication can be defined formally with computational models [7]. In the formalism of both Adleman and Von Neumann, SR is present in the read and write operations that are executed during replication.

A file can be considered as an abstract data type that has attributes and operations. The attributes of a file include: name, identifier, type, location, size, protection, and time, date and user identification [8]. The basic operations of a file include: creating, writing, reading, repositioning, deleting and truncating [8,9]. A virus is defined as a program that can infect other programs by modifying them to include possibly evolved version of itself [4, p. 2]. From the point of view of the system, a virus is a file and therefore possesses the attributes and operations of files. We can deduce that if the virus copies itself is must therefore invoke the read and write file operations when it is infecting other programs. Therefore the virus must have the appropriate access privileges in order to perform the copy [10]. In our approach it does not matter if the copy was successful or not since we are just interested in the virus making an attempt to replicate.

In this paper we use the name, identifier and location file attributes to reference the static image of the file on a storage device. The name (identifier—a unique tag) of a file F is represented as $F.name$. The location of F is usually an argument of the write and/or read operations that are used during file replication. Writing F involves making a system call specifying both the name of F and the location where F will be written. To read F, a system call is invoked that states the name of F and where in memory F or a part of F will be placed. In the event that F cannot be written or read in one execution of the operation then a pointer keeps track of the next block to be written or read.

3 Motivation

Static analysis of viruses and benign processes was conducted to establish preliminary support on our assumptions of SR-replication. A test set of 56 viruses was built by downloading live samples from various Internet malware repositories [11,12]. A second test set of benign processes was

Table 1 56 Viruses with replication attempts

Email worms	Replication attempts	Peer to Peer worms	Replication attempts	Network worms	Replication attempts	Win32 viruses	Replication attempts
Baconex	1	Agobot.a	1	Afire.b	3	Apathy.5378	1
Bagle.a	1	Banuris.b	217	Afire.d	1	Arch.a	1
Bagle.j	1	Bereb.a	474	Bobic.k	1	Barcos.a	4
Bagle.k	1	Bereb.b	481	Bozori.b	1	BCB.a	6
Bagle.m	1	Blaxe	6	Bozori.e	1	Bee	2
Bagle.n	1	Cassidy	19	Bozori.j	1	Canbis.a	14
Bagle.o	1	Cocker	61	Cycle.a	1	Civut.a	1
Dumaru.r	3	Compux.a	36	Dabber.c	1	Cornad	1
Eyeveg.m	1	Delf.a	1	Domwoot	1	Jlok	2
Klez.a	3	Gagse	257	Doomjuice.b	1	Parite.a	1
Klez.e	1	Irkaz	2	Doomran	1	Parite.b	1
Klez.i	1	Kanyak.a	1	Incef.b	27	Tenga.a	1
klez.j	2	Kifie.c	2	Kidala.a	1	Watcher.a	1
Mimail.j	1	Mantas	233	Lebreat.a	1	Zori.a	1

Table 2 56 Benign processes with replication attempts

Benign processes	Replication attempts	Benign processes	Replication attempts	Benign processes	Replication attempts	Benign processes	Replication attempts
accevt	0	ckcnv	0	diskperf	0	ipconfig	0
accwiz	0	cleanmgr	0	dllhost	0	ipv6	0
actmovie	0	clipbrd	0	dmremote	0	lodctr	0
ahui	0	cmd	0	doskey	0	lpq	0
append	0	cmdl32	0	eventcreate	0	lsass	0
blastcln	0	common32	0	exe2bin	0	makecab	0
bootcfg	0	control	0	extrac32	0	mem	0
bootok	0	convert	0	fastopen	0	netsetup	0
cacls	0	cscript	0	finger	0	notepad	0
charmap	0	csrss	0	fsutil	0	ntbackup	0
chkdsk	0	ctfmon	0	getmac	0	openfiles	0
chkntfs	0	debug	0	help	0	ping	0
cipher	0	defrag	0	hostname	0	qprocess	0
cisvc	0	diskpart	0	iexpress	0	setup	0

built using 56 executable processes from the Microsoft Windows System32 folder. All the viruses were randomly chosen and belong to the classes of Win32 viruses, network worms, email worms and peer-to-peer worms. The virtual machine software VMware Workstation with Windows XP SP2 was used to execute the the test sets. The programs Api Spy 32 and Process Monitor [13,14] were used to create log files documenting the system calls made by each process in one complete execution. Each log file was examined for SR-replication. This was determined through identification of SR by examining the arguments of read and write system calls for a reference in the "from" argument that was the name of the currently executing process or a temporary memory location where the currently executing process had copied itself earlier in the execution. The results of the testing are in Tables 1 and 2.

The total number of SR-replication for each process listed in Tables 1 and 2 is the count of distinct filenames that each process attempted to infect in one execution. We did not verify if each attempt was a success or a failure. The attempt to perform SR-replication is enough for us to label the process as a possible virus regardless if it is successful or not. The test results showed all 56 viruses attempted SR-replication at least one time to as many as over 400 times in

a single complete execution. None of the benign processes attempted SR-replication. These results provided support of our assumptions and motivation for this research.

4 Self-reference virus replication

In this section we will present a formal definition for SR and SR-replication. An approach to detect SR-replication is also presented along with an example of its use.

4.1 Definition

An operation o is invoked with arguments $(a_1 \ldots a_n)$ by a currently executing process P where $P.name$ is the name of P. The static file image F saved on a storage device is from where P was created. The name and path of F is held in $F.name$ and $P.name \mapsto F.name$, thus $P.name$ refers both to P and F. The label T is a temporary memory location containing a copy of F. When an operation $o \in O = \{read(s, d), write(s, d)\}$ where the $source$ argument $s = a_i$ and $destination$ argument $d = a_j$ with $1 \leq i, j \leq n$ and $i \neq j$ is invoked by P where $s \in S = \{P.name, T\}$ then o is said to have the *self-reference property* (SR). The argument $d \in D = \{M, I.name\}$ where M is temporary memory location and $I.name$ is the name of the destination static image file I saved on a storage device with $I.name \neq P.name$. The formal definition for SR is given in Fig. 1.

We restrict the set O to only read and write operations. We assume a process only needs to execute a sequence of these two operations to attempt replication. The sets S and D are restricted to static file images and temporary memory locations because we are only detecting replication of one file to one or more files where one or more temporary memory locations are used to complete the process. The basis case for $SR(o) = true$ is with $o.s = P.name$. In this case P refers to F in an attempt to read or write itself to $o.d$. In the case where $o.s = T$, $SR(o) = true$ when $o(T, d)$ was invoked by P at time t, $o(s, M)$ was invoked by P at time t', $t' < t$ and $T = M = F$. In this case P must have previously invoked at least one o with $o.d = M$, placing F into M which results in M converting to T.

By uniquely enumerating all o_m executed by P with $1 \leq m \leq n$, we can define $SR(o_m)$ in terms of $F\mathtt{R}o_m.s$ as shown in Fig. 2. Testing for $SR(o_m)$ is equivalent to establishing a *transitive relation* $\mathtt{R}$ between F and $o_m.s$. When $F\mathtt{R}o_m.s = true \rightarrow F = o_m.s$ through invocation of $o_1 \ldots o_m$ by P.

P invokes a sequence of o_m operations with $1 \leq m \leq n$. If $o_1.s \in S$, $o_m.d = I.name$, $o_m = write(s, d)$, $I.name \neq P.name$ and $SR(F, I) = true$ then P is said to have performed *self-reference replication* (SR-replication). The formal definition of SR-replication in Fig. 3 focuses on detecting processes that read and write their static file image to other newly created or already existing static file images. This can be accomplished in one write operation or in several read and write operations, also many memory locations can be used intermediately from F to I. $SR(F, I)$ is established by testing for SR on every o that leads from $P.name$ to $I.name$, thus SR-replication$(P) = true$ iff a transitive relation $F\mathtt{R}I$ = true. We assume that static file images can only be read from and written to. The definition does not detect a process that overwrites or modifies its own static file image.

4.2 Detection

When P starts execution, the operations o can be traced using a *directed graph* G consisting of *edge* = o_m and *node* = $\{P.name, T, M, I.name\}$. A graph is created for each P in a system and is linked to a specific P by the value of the first node of G which must always be $P.name$. Upon P invoking its first operation o where $o_m.s = P.name$ a new G is created and its *root node* = $P.name$. When a new edge is added it must be of the form $o_m.s \rightarrow o_m.d$ with $s \in S$ and $d \in D$ and the value $o_m.s$ must already be present as a previous $o_m.d$ node in G with exception of cases where $o_m.s = P.name$ which is the root node of G. A sample graph is given in Fig. 4 for a process named $vx1$.

We can see from Fig. 4 each o is enumerated in order of execution by P. The first two operations $read(M_1, M_3)$, $write(M_3, sys.bat)$ are not included in the graph G since

- $O = \{read(s, d), write(s, d)\}$
- $s \in S = \{P.name, T\}$
- $d \in D = \{M, I.name\}$
- $P.name$ = name of currently executing process that is invoking o
- T = temporary memory location containing a copy of the static file image F
- M = temporary memory location
- $I.name$ = name of the destination file
- $o.s$ = the s argument of o
- $o.d$ = the d argument of o

Fig. 1 Formal definition of SR property

Fig. 2 Transitive relation of SR

$$\forall o_m(s, d) \text{ executed by } P \text{ with } 1 \leq m \leq n,\ SR(o_m(s, d)) = true \text{ iff } F\mathtt{R}o_m.s = true$$

Fig. 3 Formal definition of $SR - replication$

$$SR{-}replication(P) = true \text{ iff } \exists o_1 \ldots o_m \text{ with } 1 \leq m \leq n, \text{ where } o = write(s, d) \text{ and } o_m.s \in S, o_m.d = I.name, I.name \neq P.name \text{ and } SR(F, I) = true$$

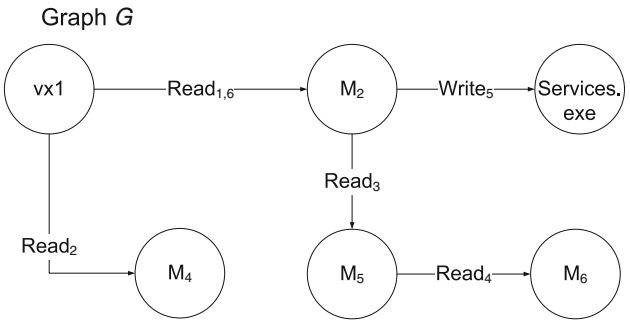

Fig. 4 Sample abstract graph for vx1

neither has $o_m.s = P.name$ which is vx1. The root node of G must always be the first o of P where $o.s = P.name$. We see this in $read_1$ where $read_1.s = vx1$. Notice the operation $read_{1,6}$, the notation shows the operation with the same arguments occurred twice, at the first and sixth invocation. Every operation in G is true for SR and correctly placed in the form $o_m.s \rightarrow o_m d$. A test for SR-replication($vx1$) was done when the operation $write_5(M_2, services.exe)$ was added to G. The path vx1 $rightarrow$ services.exe shows the transitive relation FRI. This path also satisfies our definition of SR-replication in Fig. 3 and therefore SR-replication$(vx1) = true$. When a graph G of a process P contains a path from $P.name \rightarrow I.name$ then $FRI = true$ which results in SR-replication$(P) = true$. Construction of G only has to continue until SR-replication$(P) = true$ at which point P can be flagged as exhibiting virus replication. If P finish execution and SR-replication$(P) = false$ then P is assumed benign.

If P invokes an operation $o_m(s, d)$ where $SR(O) = false$ and $o_m.d$ is already a node of G, then $o_m.d$ must be removed in one of two ways: If $o_m.d$ is a leaf node, it is simply removed and G remains the same. If $o_m.d$ is an internal node in G then $o_m.d$ is removed and G is reorganized by eliminating all incoming edges to $o_m.d$ and repositioning all outgoing edges from $o_m.d$ to each child node to come from each parent node of $o_m.d$ to the child node. Fig. 5 shows graph G from Fig. 4 after removal of node M_2. The incoming edge $Read_{1,6}$ from the parent node $vx1$ was eliminated and the outgoing edges $Read_4$ and $Write_5$ were each reposition to come from the parent node $vx1$ to the child nodes M_6 and $services.exe$.

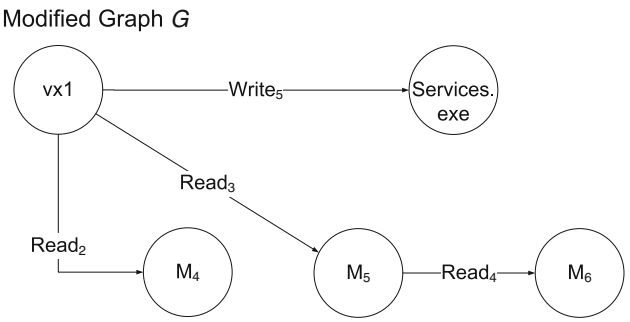

Fig. 5 Reorganized abstract graph for vx1 after removal of node M_2

Table 3 Win32 API calls with equivalent read/write operation

Win32 API	Read/Write operation
`BOOL WINAPI CopyFile( __in LPCTSTR lpExisting FileName, __in LPCTSTR lpNewFile Name, __in BOOL bFailIfExists);`	*write*(lpExisting FileName,lpNewFile Name)
`BOOL WINAPI ReadFile( __in HANDLE hFile, __out LPVOID lpBuffer, __in DWORD nNumberOfBytes ToRead, __out LPDWORD lpNumberOf BytesRead, __in LPOVERLAPPED lpOverlapped);`	*read*(hFile, lpBuffer)
`BOOL WINAPI WriteFileEx( __in HANDLE hFile, __in LPCVOID lpBuffer, __in DWORD nNumberOf BytesToWrite, __in LPOVERLAPPED lpOverlapped, __in LPOVERLAPPED_ COMPLETION _ROUTINE lpCompletion Routine);`	*write*(lpBuffer, hFile)

Our approach is based on general read and write operations. We assume any specific operation that performs a read, write or copy by specifying in the arguments the source and destination can be equivalently written using the general read and write operations used in this research. Table 3 shows some Win32 API calls [15] and their conversion to an equivalent general read or write operation. Note that we are only interested in the source and destination arguments of the operation.

Our approach focuses on detecting SR-replication on a local machine, it currently does not detect SR-replication from one local machine to another across a network, we reserve this for future work. We are aware of the ability of

some viruses to replicate without using SR-replication. This can be accomplished either by replicating from a source that is not P or invoking commands in some other process that results in replicating P. These types of replication we refer to as indirect self-reference replication, (ISR-replication), and is currently not detectable by our current approach. Expanding our approach to include ISR-replication is reserved for future work.

4.3 Example

In this section we will use portions of the log file of a virus used in our static analysis to give an example of SR and SR-replication using a graph for testing. The log file was created using API SPY 32 [13] which logs all the Win32 API calls invoked by a process [15,16]. The example in Fig. 6 is of the Cassidy worm, a packed Peer-to-Peer worm [2, p. 332] and [17] that from our static analysis testing results in Table 1 attempted replication 19 times. From the partial log file we see the Cassidy worm attempted to copy itself six times using the API call `CopyFileA` which is the same as the API call `CopyFile` but is used when dealing with the ANSI character set [15]. From Table 3, `CopyFileA` is mapped to *write*(`lpExistingFileName`,`lpNewFileName`). As an example, the fourth `CopyFileA` operation is mapped to:

Partial Log File for Cassidy Worm

```
00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 pindlebot.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 pindlebot.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 maphack.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\diablo 2 maphack.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\playstation2 emulator.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\playstation2 emulator.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\kazaa hack.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\kazaa hack.exe",
                              DWORD:00000104)

00402D6E:CopyFileA(LPSTR:00BA0330:"C:\DOCUME~1\JAM-VX~1\Desktop\CASSIDY.EXE",
                   LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\cable modem utility.exe",
                   BOOL:00000000)
00402D28:GetWindowsDirectoryA(LPSTR:00BA0200:"C:\WINDOWS\Shared Folder\cable modem utility.exe",
                              DWORD:00000104)
```

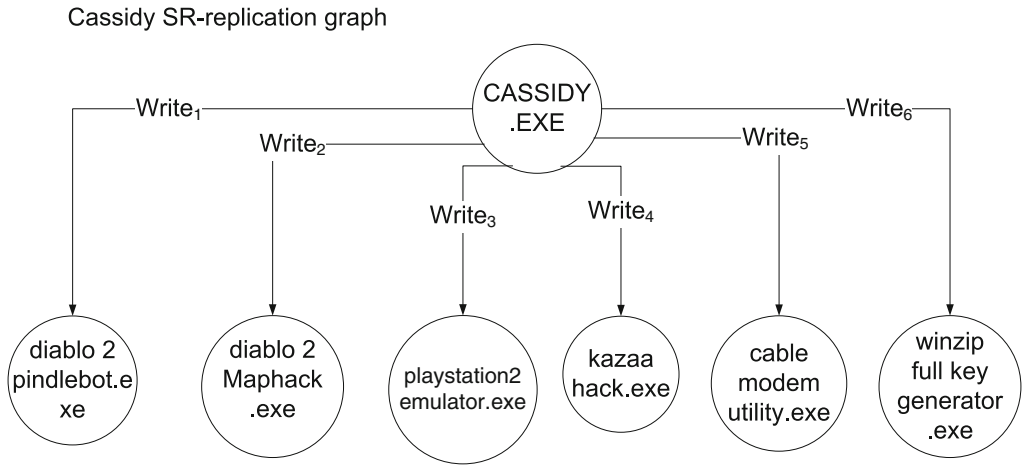

Fig. 6 $SR-replication$ of cassidy peer-to-peer worm

$write$("C:\DOCUME 1\JAM-VX 1\Desktop\CASSIDY.EXE", "C:\WINDOWS\Shared Folder\kazaa hack.exe").

All the other operations are mapped in similar fashion. From the graph we see $rootnode$ = CASSIDY.EXE and $SR(o_m) = true$ for each o_m in the graph. Consider

$write_4$(C:\DOCUME 1\JAM-VX 1\Desktop\CASSIDY.EXE, C:\WINDOWS\Shared Folder\kazaa hack.exe).

We can see:

P = CASSIDY.EXE, $P.name = write_4.s$ = C:\DOCUME 1\JAM-VX 1\Desktop\CASSIDY.EXE and $I.name = write_4.d$ = C:\WINDOWS\Shared Folder\kazaa hack.exe.

Applying these values to the definition of SR in Fig. 1, we see $SR(write_4) = true$ and this result holds for all the other $write_m$ operations as well. When operation $write_1$ was invoked, the graph was updated and a test for $SR - replication$ was conducted since a $write$ operation occurred with $write.d = I.name$ = diablo 2 pindlebot.exe. According to the definition in Fig. 3, SR-replication($CASSIDY.EXE$) = $true$. Had this been a real time detection, the process would have been flagged as exhibiting virus replication behavior. To allow readability, only the filenames were placed in the graph of Fig. 6 when it should be the complete path and filename.

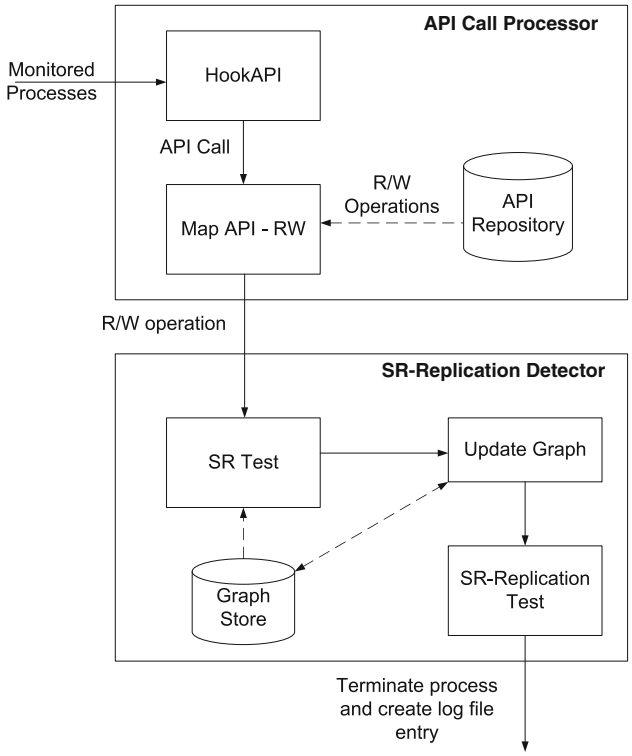

Fig. 7 SRRAT architecture

5 Implementation prototype

To test our SR-replication theory, a runtime monitor implementation prototype named SRRAT (SR-Replication Analysis Tool) was created in two versions. One version runs in user mode and the other in Kernel mode, both prototypes were built to run on the Windows XP platform. The user mode version tracks API function calls and the Kernel version tracks system services used by all currently running processes using a technique known as hooking [18,19]. Each prototype followed the design architecture in Fig. 7. The architecture consists of two main components: API Call Processor and the SR-Replication Detector. The API call processor is composed of: HookAPI, MapAPI-RW and an API Repository. The SR-Replication Detector consists of: SR test, SR-Replication test, Update-Graph and a graph storage.

The overall idea of the prototype is to follow the execution of processes on a system. As the process executes it will inevitably interact with the operating system and this interaction is recorded and analyzed by SRRAT. The method of interaction between all processes including viruses and the operating system is through the invocation of API function calls and Kernel system services [16,18–20]. SRRAT tracks only a subset, principally those that implement file system operations: open, close, read, write, copy and and a few other operations. When one of these is invoked by a process, SRRAT hooks it and analyzes its parameters to determine the presence of SR and if SR-replication has occurred. A hook is a method by which a user can redefine a standard API call and have the operating system redirect invocations of the standard API call to the user defined API call.

It is very powerful in the sense that a user can modify the execution behavior of processes without having direct access to the source code of that process. The following is a description of the purpose and responsibility of each component of SRRAT.

5.1 ACP: API CAll processor

The API Call Processor's (ACP) main purpose is to detect the invocation of an monitored API call and pass its parameters to the SR-replication detector. The ACP is in idle mode waiting for the operating system to send a notification that a monitored API has been invoked by some process. At this point the APC takes control of the invocation and checks to see if the API is a read or write operation according to a predefined mapping. If the API is a read or write operation the APC passes it along with its parameters to the SR-replication detector for further processing. During this time the process that originally invoked the API is in a wait state pending the completion of the API. This serves to stall the execution of possible viruses while they are being analyzed for SR-replication. The APC consists of two subcomponents

called HookAPI and MapAPI-RW plus an API repository which are explained next.

The HookAPI subcomponent of the ACP is responsible for the actual interception of the API calls being monitored by SRRAT. The interception is done using an API hooking mechanism that notifies the ACP of the invocation of a specific API call which HookAPI has hooked. Once the process calls an API that has been hooked, the operating system redirects the call from the standard API to the user defined API where the redirection is part of HookAPI. When HookAPI completes its job the standard API call and its parameters have been redirected to the user defined API call and thus commences the second component of ACP which is MapAPI-RW.

The second subcomponent of the ACP is called MapAPI-RW and it serves the singular purpose of deciding if the API call that has been passed to it is a general read or write operation. If the API is determined to be a read or write operation then the API is labeled as such and it is passed along with the source and destination parameters to the SR-replication detector, which is the second component of SRRAT. The determination of an API being a read or write operation is accomplished by searching for the API name in the API repository and checking if its mapping is to a read or write operation. If it is matched to a read or write operation, the API parameters specified in the repository as the source and destination parameters are parsed from the API call's parameter structure and passed along with the API and its read/write label to the SR-replication detector.

The ACP has an API repository which is a list of all the hooked API functions. The list has the API function name and the parameter names of the source and destination parameter according the the specific API function's documentation [16,20] along with its mapping as a read or write operation. The API repository does not have the name of API functions that are not read or write operations even though they may be hooked by SRRAT for various implementation reasons. Only those API functions that represent a read or write operation require a mapping to a general read or write operation with the appropriate parameters and therefore are the only ones that need to be stored in the repository.

5.2 SRD: SR-replication detector

The second main component of SRRAT is the SR-replication detector (SRD). This component will execute as a result of the ACP passing along to SRD an API function that has been determined to be a read or write operation. The API function is received with the function name, a read or write label and the source and destination parameters. SRD has several responsibilities, the first one is to check for SR, if SR has occurred then a graph has to be created for the process that invoked this API function. If a graph already exists it is updated. The second responsibility is to check for SR-replication, this is done when a graph is updated with a write operation where the destination is the name of a file. The third responsibility is to return a detection confirmed message back to SRRAT so the process can be terminated and flagged as exhibiting possible virus behavior. The read and write operations of a process are stored in a graph using nodes and edges. SR-replication is determined by traversing the graph to establish transitivity between the process name at the root node and a file name located in some leaf node. SRD is composed of three subsections: SR-test, SR-replication test and update graph plus a graph storage which are explained below.

The SR-test subcomponent of the SRD is responsible for testing if a process has attempted to reference itself in the source parameter of an API function that it has invoked. We are most interested in this case when it occurs in read or write operations. The test is performed by comparing the process name with the source parameter of the API function. If the process name is a substring of the source parameter then the process has tested positive for SR. The other form of testing for SR is to search the existing graph of the process for a node that matches the source parameter of the API function. If a match is made on the graph then process has tested positive for SR. When SR occurs the API function with all its parameter information is passed along to update graph for insertion in an existing graph or creation of a new graph.

The second subcomponent of the SRD is called SR-replication test (SRT) and its principle responsibility is to check if SR-replication has been attempted by a specific process. This test occurs every time a process's graph has been updated with a write operation where the destination is a file. The graph is traversed backwards from the just inserted node, which contains the destination parameter of the API function which is a file name and path, back to the root node of the graph. If a path exists between these two points then the transitivity property holds true between the process and another file and therefore SR-replication has been attempted and SRT returns true to SRRAT.

The third subcomponent of SRD is Update-Graph which is in charge of adding new nodes to the graph as they are passed in from the SR-test. When an API function with its source and destination parameter are passed in, one of several actions can be taken. If there is no existing graph for the process that invoked this API function and the source parameter is the file name and path of the process, then a new graph is created with the source parameter as the root. If a graph already exists for the process, the graph is traversed to find a node that matches the API functions source parameter. When a match is made if the node has no outgoing edges or none of its outgoing edges point to a node that matches the destination parameter then a new edge is created from the existing node to a new node with the file name and path stored in the

destination parameter. If an edge already exists with the same source and destination parameters but a different API function name on the edge, the new edge is created. If an edge already exists with the same source and destination parameters and the same API function name on the edge then only its enumeration is modified to show the order of execution for multiple attempts of the same API function by the same process with the same source and destination parameters. Once the update to the graph is done, a notification is sent to SRT if and only if the just inserted edge contains an API function that is a write operation. The last operation done by this subcomponent before exiting is to save the graph in the Graph Store.

The SRD has a Graph Store which is a temporary memory storage of all the graphs currently being used by SRRAT to track processes. Each graph is accessed by the root node which holds the name of the process currently running on the system. When a process with a graph in the store finishes execution or is terminated by SRRAT, its graph is destroyed to release memory and reduce resource usage on the system.

6 User mode prototype

The first version of SRRAT was implemented as a user mode process running in Windows XP. In this version, SRRAT traced the Win32 API function calls invoked by all currently running user mode processes. The prototype was a terminate and stay resident runtime monitor, meaning it would be quietly running in the background monitoring the execution behavior of all user mode processes currently running on the system while conveniently placed as an icon in the windows task bar for simple start and stop functionality. The API functions that were traced for read and write operations are listed in Figs. 8 and 9 along with their read/write mapping and the source and destination parameters. All the API functions that were monitored by SRRAT are located in the kernel32.dll dynamic link library. SRRAT implemented API hooking on the functions that were being monitored. To successfully perform hooking SRRAT was implemented as a dynamic link library. Aside from the API functions monitored in Figs. 8 and 9 for their read and write operations necessary to establish SR and SR-replication, there were other API functions, listed in Fig. 10 that had to be hooked and monitored to correctly implement this version of SRRAT. The following is a description of the implementation of the components of SRRAT in user mode.

6.1 Implementation

The HookAPI was implemented using hooking techniques for Win32 user mode API function calls. This is accomplished with the invocation by HookAPI of an API function

Win32 API	Read operation
`void CopyMemory( PVOID Destination, const VOID* Source, SIZE_T Length);`	*read*(Source, Destination)
`BOOL WINAPI ReadFile( HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);`	*read*(hFile, lpBuffer)
`BOOL WINAPI ReadFileEx( HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead, LPOVERLAPPED lpOverlapped, LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);`	*read*(hFile, lpBuffer)

Fig. 8 Mapping of read Win32 API calls in user version SRRAT

Win32 API	Write operation
`BOOL WINAPI CopyFile( LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName);`	*write*(lpExistingFileName, lpNewFileName)
`BOOL WINAPI CopyFileA( LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName);`	*map*(lpExistingFileName, lpNewFileName)
`static BOOL WINAPI CopyFileW( LPCWSTR lpExistingFileName, LPCWSTR lpNewFileName);`	*map*(lpExistingFileName, lpNewFileName)
`BOOL WINAPI ReplaceFile( LPCTSTR lpReplacedFileName, LPCTSTR lpReplacementFileName);`	*write*(lpReplacementFileName, lpReplacedFileName)
`BOOL WINAPI WriteFile( HANDLE hFile, LPCVOID lpBuffer);`	*write*(lpBuffer‘hFile)
`BOOL WINAPI WriteFileEx( HANDLE hFile, LPCVOID lpBuffer);`	*write*(lpBuffer‘hFile)

Fig. 9 Mapping of write Win32 API calls used in user version SRRAT

called `SetWindowsHookEX` [16,20]. Invoking this function allowed SRRAT to hook API functions by rewriting the IAT of all currently running processes. SRRAT reads from the API repository all the API function names that needed to be monitored which are listed in Figs. 8 and 10. These names are loaded into memory and HookAPI invokes `SetWindowsHookEX`. For each API functions that needs to be hooked, SRRAT has a new version of the API which implemented our SR-replication detection code. Each running process has in its IAT table the memory address of all the Win32 API functions that it may invoke during its execution. When the hooks are placed for SRRAT, Windows overwrites these memory locations with memory locations of our redefined API functions. This change of memory addresses allowed Windows to redirect invocations of the monitored API functions from the standard API function to our version of the function. Hooking the API functions in HookAPI was the critical step needed for SRRAT to function.

The MapAPI-RW subcomponent was automated as a result of API hooking techniques. Our redefined API functions are executed only when the API is invoked by some

Win32 API	Read/Write operation
`void CreateFileW( LPCWSTR lpFileName);`	update list with new file/handle: `lpFileName` and `HANDLE`
`HANDLE WINAPI CreateFileA( LPCSTR lpFileName);`	update list with new file/handle: `lpFileName` and `HANDLE`
`HFILE WINAPI OpenFile( LPCSTR lpFileName);`	update list with new file/handle: `lpFileName` and `HFILE`
`BOOL WINAPI CloseHandle( HANDLE hObject);`	*read*(hFile, lpBuffer)
`BOOL WINAPI DeleteFile( LPCTSTR lpFileName);`	remove from list existing file/handle: `lpFileName`

Fig. 10 List of Win32 API calls needed to implement user version of SRRAT

process. As a result when the body of our API function executed we already predetermined the function to be a read or write. In the body of the function we inserted code to read the correct parameters that represented the source and destination and continued to the SRD component of SRRAT.

One key piece of information needed for the SRD to work was acquiring the name of the process invoking the API function currently being processed through SRRAT. In Windows, process names are represented in two forms, the first is as a string containing the full process name and path, the second is with a process id (PID). When a process starts execution, Windows assigns it an integer value which is the process's PID, this value is used for various tasks throughout the life cycle of the process, especially when it interacts with the Windows operating system. When a process invokes a hooked API, Windows provided SRRAT a field of the hooked API structure containing the full process name and path as a string. This was used to implement the SRD.

A second key piece of information used by SRD is names of files that are read or written by a process. In similar fashion to processes, files in Windows are represented in two forms, the first is a string with the file name and path, the second is with a file handle. The file handle is an integer value assigned to a file when it is first opened by a process and destroyed when the file is closed. Some API functions such as `ReadFile` and `WriteFile`, use handles to represent the file that is being accessed but other API functions such as `OpenFile` use a string to represent the filename. Thus one file can have two ways of representation and this required the SRD to keep a list of file names and their associated handle. Each time the API functions `OpenFile` and `CreateFile` was invoked, SRD would make a new entry in the list of the newly opened or created file and the associated file handle. This entry would later be removed from the list when `CloseFile` was invoked. Each time `ReadFile`, `WriteFile` functions were invoked SRD would look up the file handle and return the associated file name and path.

Once all the needed information became accessible to the SRD, performing SR tests was straightforward, every time a read operation occurred the filename and path in the source parameter would be compared to the process name. If the process name was a substring of the source parameter then SR was indeed present. Each time SR was found the graph for the process would be updated in Update-Graph. The graph store was a set of graphs each one with a different process name as the root node which identified the graph uniquely. A test for SR-replication occurred each time the `WriteFile` or `CopyFile` API function was hooked. The graph for the process was retrieved from the storage and traversed backwards from the newly added destination node from the `WriteFile` or `CopyFile` API to the root node checking for transitivity. When SR-replication was established for a specific process, that process was terminated by SRD using the `TerminateProcess` API function [16,20].

6.2 Limitations

Several viruses do not interact with the operating system at the user mode level. Instead, they deal directly with the kernel and its function calls thus completely avoiding SRRAT. These viruses cannot be detected by the user mode version of SRRAT. Also this version ran in user mode and can be infected by the very same virus it is trying to detect if the virus injured before infecting, thus rendering SRRAT useless. Some viruses purposely encrypt the parameters when calling an API, SRRAT would process these parameters correctly but can produce a false negative since the encrypted parameters do not match the actual files being manipulated by the virus. Other viruses call the API functions directly by loading the dynamic link library and acquiring the memory addresses of the API functions needed for the virus to run. Later in execution the virus passes the parameters to the memory location while completely preventing detection by avoiding the hooks placed by SRRAT.

7 Kernel mode prototype

The second version of SRRAT was created to run in Windows Kernel mode. This version traced the Zwxx system services provided by the ntdll.dll dynamic link library which are exported from the Kernel process named ntoskrnl.exe. These are all Kernel system services and can be called directly by a Kernel process or indirectly by a user mode process. When a user mode process invokes a user mode API such as `OpenFile`the API function sends the request from user mode to a system service in Kernel mode, in this case `ZwOpenFile`. Tracing Kernel mode system services has three main advantages over tracing user mode API function calls:

- This version allows high probability of identifying SR-replication that may have been missed by the user

mode version. This results from the higher level of difficulty any process, including viruses, faces in trying to execute and avoid using Kernel mode system services. It is very difficult to avoid interacting with the Kernel in some form, thus the probability of detecting SR-replication.

- SRRAT itself has a higher level of protection from virus infection by running in the Kernel. The Kernel is considered to be privileged access and not every process including other Kernel processes can have direct access to this privileged space. This give SRRAT a higher rate of survivability from virus attack and therefore increases the chances of running longer and detecting SR-replication.
- The form in which Zwxx system services are structured requires two parameters to be included that represent the file name with path and the file handle. This requirements eliminates the need to hook and open or close system services and also eliminates the need of SRRAT to keep a list of file names and handles. This reduction in processing allows SRRAT to run faster and with less consumption of processing time.

To run a process in Kernel mode it must be executed by loading it as a system service. Two separate programs were created for this purpose, one to load and the other to unload the service from the operating system. Normally, services do not have user mode start and stop functions, these were added for convenience. Also SRRAT in this version was purposely created as a rootkit [18,19] to include some techniques allowing SRRAT to hide from the system and therefore avoid being attacked or infected by a virus. The two techniques used for this purpose was: (1) the system service was hidden from the operating system, the name of the service would not show as currently running by any application in Windows and (2) the configuration file used by the service was redirected to a different part of the operating system thus hiding it as well. These are only basic hiding techniques but in fact are very useful. They allow SRRAT to run on the system as an invisible process which puts SRRAT on the same level playing field as some advance viruses which is necessary for any virus detector to run effectively and successfully.

Hooking system services has a different implementation than hooking Win32 API function calls but the underlying theory is the same. In the Kernel all the system services including the Zwxx services are exported by their memory location to a table called the System Service Dispatch Table (SSDT). When a request for a service comes in from user mode, the specific request is located in the SSDT and the operating system carries out the service. Similarly to hooking Win32 API function calls, this version of SRRAT had a list of redefined system services, when SRRAT was loaded it would overwrite in the SSDT the memory location of the standard system services with our redefined versions and thus the requests would be redirected to SRRAT and its redefined versions, this is how SRRAT hooked the needed Zwxx system services, which are listed in Fig. 11. The following is a description of the implementation of the components of SRRAT in Kernel mode.

7.1 Implementation

Overall implementation in this version of SRRAT was much easier than the user mode version for both the ACP and the SRD. HookAPI was implemented by modifying the memory address of standard system services with our redefined versions in the SSDT. Since there is only one SSDT for the

Fig. 11 Mapping of system services used in Kernel version SRRAT

Kernel System Service	Read/Write operation

`NTSTATUS ZwCreateSection()`
`OUT PHANDLE SectionHandle,`
`IN ACCESS_MASK DesiredAccess,`
`IN HANDLE FileHandle OPTIONAL);`
	read(`FileHandle, SectionHandle`)

`NTSTATUS ZwReadFile(`
`IN HANDLE FileHandle,`
`OUT PVOID Buffer,`
`IN ULONG Length,`
`IN PULONG Key OPTIONAL);`
	read(`FileHandle, Buffer`)

`NTSTATUS ZwMapViewOfSection()`
`IN HANDLE SectionHandle,`
`IN HANDLE ProcessHandle,`
`IN OUT PVOID BaseAddress,`
`IN ULONG AllocationType,`
`IN ULONG Win32Protect);`
	write(`SectionHandle, BaseAddress`)

`NTSTATUS ZwWriteFile(`
`IN HANDLE FileHandle,`
`IN PVOID Buffer,`
`IN ULONG Length,`
`IN PULONG Key OPTIONAL);`
	write(`Buffer, FileHandle`)

entire operating system, the actual hooking only had to occur by SRRAT once as opposed to the user mode version where hooking occurred once for each process. The number of system services hooked were less than those in the user mode version, they primarily were only the Zwxx system services that represented a read or write operation. In Kernel mode system services decompose user mode file operations to basic read and write operations. Most notably the Win32 API function call `CopyFile` is translated to a call to `ZwReadFile` and `ZwWriteFile` in Kernel mode. This decomposition to simplified services greatly reduced the implementation of HookAPI.

As was the case in the user mode version of SRRAT, the MapAPI-RW subcomponent was automated as a result of system services hooking techniques. Our redefined system services are executed only when the Kernel receives a request for a specific system service. As a result when the body of our system service executed we had already predetermined the service to be a read or write. The services's body had code inserted to read the necessary parameters that represented the source and destination parameters of the service and SRRAT continued to the SRD component of SRRAT. In this version of the ACP, the API repository had a much smaller list of system service functions needed to be hooked. This was a key advantage allowing for slightly less memory usage when SRRAT was operating.

The SRD was implemented in principally the same fashion as in the user mode of SRRAT. Both the test for SR and the SRT subcomponent has the same basic code as their user mode version. One difference was the removal of the list of file names and handles which was needed in the user mode version for the SRD to work properly. In Kernel mode the system services already provide all the file information that had to be found by SRD in user mode. This slight reduction in code creates a faster implementation which is key when dealing with aggressive fast spreading viruses. The graph store was kept in Kernel memory and the individual graphs were created, destroyed and accessed in the same manner as the user mode version. When a process was found to have exhibited SR-replication it was terminated using the Kernel system service `ZwTerminateProcess`.

7.2 Limitations

The main limitation with this approach was in its implementation. Kernel mode programming is a very complex form of programming and there is little documentation available to aide the programmer. Many of the types, structures, functions found in the Kernel are not documented and using them with some confidence is only based on feedback from other programmers that have walked the path before. Implementing most of the code took some researching before being successful. Only with experience can a programmer become skilled in working with the Kernel. Most of the implementation was built using already established code heavily modified and questions posted on various forums provided some answers and partial solutions, the rest was done through trial and error. Many of the undocumented code used in this version had to be modified or rewritten to please the build environment into successfully compiling and building the executable version of SRRAT.

Two main limitations encountered during implementation was: (1) obtaining the name of the process requesting the system service, which is critical to test for SR, graph creation and identification and (2) Acquiring enough memory for SRRAT to effectively run. It was very difficult at first to obtain working code that would produce a string representing the name of the process. After 4 days of trial and error the name was finally obtained. The Windows Kernel seemingly runs within a limited memory space called pools and all kernel processes use this memory pool for their specific purposes. Allocating and using Kernel memory is a difficult science to understand and implement. Several setbacks were suffered by SRRAT using too much memory causing Windows to display a blue error screen, also known as the Blue Screen of Death (BSOD), which led to the system crashing and requiring a restart. The main memory problems came with building the graphs in the SRD which works by implementing a linked list, each pointer was created with a chunk of kernel memory, it seemed this process repeated several times caused the system to produce the BSOD. The memory problem was not solved and this resulted in the implementation creating log files displaying all of the needed information to detect SR-replication in a given process.

8 Tests and results

A suite of experiments were created to test the theory of SR-replication and the user and Kernel mode implementation prototypes of SRRAT. The tests were conducted with viruses drawn from a collection of 445 virus samples. The collection was built from malware repositories on the Internet [11,12]. The viruses in the collection were chosen to be representative of all the major categories of virus types. The amount of virus samples for each category is listed in Fig. 12. All the sample viruses in the collection were scanned using Kaspersky Anti-Virus software [21] to validate their authenticity, name and classification. The focus of these tests was to count the total number of correct identifications of viruses plus the total amount of false negatives and false positives produced by the prototypes. All the tests were conducted on a desktop computer running Microsoft Windows XP with no anti-virus software installed. The testing involving viruses were done using Vmware Virtual Workstation [22], which allows for

Virus Types	Total Samples
Email Worms	**110**
Network Worms	**99**
Peer-to-Peer Worms	**79**
Instant Messaging Worms	**6**
Win32 Viruses	**151**

Fig. 12 Virus classification with total samples amount

safe isolation of the viruses from infecting an actual machine while providing a rich real computer emulation environment.

Testing the theory of SR-replication entails inquiring if this is a characteristic that occurs in multiple viruses across several different virus classifications and can further be identified in some manner. More importantly, it is pivotal to establish if SR-replication is a characteristic that does not occur in benign processes, this is one of our assumptions. Establishing these two points will indicate if SR-replication can be used to distinguish between viruses and benign processes and at the same time produce little or no false negatives and false positives.

Our approach to test the theory of SR-replication was to execute several viruses and commonly used applications and operating system processes and have their Win32 API function calls and Kernel system service requests with source and destination arguments recorded and analyzed. The program used for this was API SPY 32 [13] which records Win32 API function calls and Process Monitor [14] which records Kernel system service requests. The benign processes used for testing were chosen by logging all processes running on two computers on a 5 day span, the processes executed the most were chosen for testing. The viruses chosen for this test were randomly selected from the collection assuring that each category was represented in this test set.

In testing the user mode implementation prototype of SRRAT three criteria need to be analyzed, they are: false positive production, false negative production and usability as a real time monitor and detector. To test for false negative production a test set of viruses were executed one by one in the virtual machine software with SRRAT running. False positive production was tested together with usability as a real time monitor and detector by running SRRAT on two actual computer desktops for three days under normal computer use. Both computers had full Internet access and carry heavy use of several popular desktop applications plus Internet programs. Installations of new software and updates to already existing software were purposely done during the testing period as well. Anti-virus software was present and running on both computers during testing. The viruses were chosen by using those that showed use of Win32 API function calls during their execution as recorded by the API SPY 32 log files. This resulted in a set of 66 viruses.

The Kernel Mode Prototype of SRRAT was tested using the same three criteria as that used for the user mode prototype: false positive production, false negative production and usability as a real time monitor and detector. Testing false positive production was conducted jointly with usability as a real time monitor and detector by executing SRRAT on two actual computer desktops for three days under normal computer use. The two computers had full Internet access and experience heavy daily use of many popular Internet and desktop applications. New software installations and updates to already existing software were purposely done during the testing period as well. Anti-virus software was running on both computers during testing. Testing for false negative production was done by executing a test set of viruses one at a time in the virtual machine software with SRRAT running. The viruses were chosen by using those that showed use of Kernel system services during their execution as recorded by the Process Monitor log files. This resulted in a set of 367 viruses.

Performing the tests was a long and strenuous process. The nature of virus testing requires several re-installations of the host computer to ensure a clean virus free environment for the next test. To ensure that each virus was executed in a virus free environment, the Vmware workstation virtual machine was restored to a clean state after concluding each test. Assuring a virus free environment for each test was needed to ensure that a virus was not kept form executing normally as a result of a previous virus's infection on the virtual machine. What follows is an analysis and evaluation of all the test results along with observations and experiences from conducting the tests.

9 Theory validation test results

Conducting this test took approximately 4 days to complete. The benign process testing was completed in one day and the balance of days was taken by the virus testing. The test results for the benign processes are presented in Fig. 13. The first and fourth columns are the names of each benign process tested, the second and fifth columns are the results of testing for SR, the third and sixth columns are the test results for SR-replication with Y meaning yes and N meaning No. When testing commenced we decided to also record any occurrence of SR. Our reasoning for this was if a benign process was an SR process and did not attempt SR-replication during testing, the possibility of attempting SR-replication could still occur under different execution conditions. Therefore we considered an SR benign process as being a potential false positive assuming the correct execution conditions were in place for SR-replication to occur. The test results show that all 62 benign process not only did not attempt SR-replication but none even attempted SR. The result is the whole test set can be classified as non-SR benign processes based on the test results. Not finding any SR-replication did not surprise us as

this characteristic not being found in benign processes is one of our main assumptions in this research. We were surprised though that none of the processes attempted SR. As each process was executed we interacted with them in as many typical user ways as possible to afford maximum possibility to the process to exhibit different forms of behavior. Finding none of these processes attempted SR-replication and SR supports our assumption that SR can be used to distinguish between viral and non-viral processes. Furthermore the lack of SR reenforces our assumption by showing that not only do benign processes not attempt SR-replication but they may not even read themselves, thus not be an SR process, in any way during their execution. This further distinguishes benign from viral based on SR-replication characteristic and reduces the chances of false positive production.

All 284 viruses were tested one by one in the virtual machine for the attempt of SR-replication. The virtual machine was reset to a clean virus free state before each test was conducted. A summary of the virus results are in Fig. 14. Analyzing the results it becomes clear that a majority of the viruses did in fact show SR-replication with the exception of the Win32 Virus class. For that class the majority, 58 viruses, did not show SR-replication. The viruses that did not show SR-replication could be the result of advanced anti-detection techniques. Some viruses have the capacity to detect running processes that may be used to terminate or erase them, if they detect such a process they will behave as a benign process and do nothing exhibiting virus like behavior, this of course includes replication. Another reason for these viruses not showing SR-replication is they may have not found the right conditions to replicate. Win32 viruses

	Email Worms	Peer-to-Peer Worms	Network Worms	Win32 Viruses
SR-replication	43	47	45	13
No SRreplication	28	24	26	58

Fig. 14 Summary results theory validation virus test

Fig. 13 Theory validation test results benign processes

Benign Process	SR	SRR	Benign Process	SR	SRR
AcroRd32.exe	N	N	netbeans.exe	N	N
AcroRd32Info.exe	N	N	OUTLOOK.EXE	N	N
Ad-Aware.exe	N	N	pa.exe	N	N
AlbumDB2.exe	N	N	palaunch.exe	N	N
AsusProb.exe	N	N	pastatus.exe	N	N
bibtex.exe	N	N	pdflatex.exe	N	N
CFD.exe	N	N	PHOTOED.EXE	N	N
csrss.exe	N	N	POWERPNT.EXE	N	N
Deskup.exe	N	N	procexp.exe	N	N
devenv.exe	N	N	Procmon.exe	N	N
emule.exe	N	N	rundll32.exe	N	N
ErrorKiller.exe	N	N	services.exe	N	N
EXCEL.EXE	N	N	Skype.exe	N	N
Explorer.EXE	N	N	sol.exe	N	N
firefox.exe	N	N	sqlservr.exe	N	N
FrameworkService.exe	N	N	svchost.exe	N	N
gbk2uni.exe	N	N	symlcsvc.exe	N	N
GoogleEarth.exe	N	N	SyncBackSE.exe	N	N
HWN.exe	N	N	System	N	N
IEXPLORE.EXE	N	N	TEXCNTR.EXE	N	N
iexplore.exe	N	N	TexFriend.exe	N	N
java.exe	N	N	tomcat5.exe	N	N
LimeWire.exe	N	N	verclsid.exe	N	N
MATLAB.exe	N	N	WCESCOMM.EXE	N	N
Mcshield.exe	N	N	WinEdt.exe	N	N
MemoryManagement.vshost.exe	N	N	winlogon.exe	N	N
MSACCESS.EXE	N	N	winmine.exe	N	N
mscorsvw.exe	N	N	WinRAR.exe	N	N
msnmsgr.exe	N	N	WINWORD.EXE	N	N
naPrdMgr.exe	N	N	wmiprvse.exe	N	N
nbexec.exe	N	N	wuauclt.exe	N	N

```
NODE: C:\Documents and Settings\JAM-VX-
MACHINE\Desktop\gallo.exe -1
  NODE--Out--> ReadFile  1  read  NULL 1564680
=====================================
NODE: NULL 1564680
  NODE--Out--> WriteFile  2  write  C:\WINDOWS\
system32\kerneldll32.api -1
=====================================
NODE: C:\WINDOWS\system32\kerneldll32.api -1
=====================================
+ENDofGRAPH+
```

Fig. 15 EW-Win32.Alanis *SR*-replication graph

tend to infect files that are of a specific format, most notably the Portable Execution (PE) format. It is possible these viruses searched for victim files and simply did not find any and thus could not replicated. A second interesting observation from the results is where the *SR*-replication occurred. Of the viruses that did replicate, the overwhelming majority did so in Kernel mode and a smaller amount replicated in user mode. Only three viruses replicated in both user and Kernel mode. The implication of the majority of these viruses replicating in Kernel is they do this purposely to attempt detection avoidance. By executing in Kernel mode they have the capacity to run below or at the same level as virus detectors thus allowing them more leeway to hide and avoid detection. Just considering only the static analysis, the viruses that did not show *SR*-replication are false negatives. It is however difficult to say if they really could be false negatives for the reasons stated here, it is possible they could be detected with the proper virus detector in place.

Overall the theory validation testing results strongly support our assumption that *SR*-replication can distinguish between viruses and benign processes. The key to this conclusion is the fact that no false positives occurred and several true positives occurred. If false positives had occurred then one can conclude that *SR*-replication is a characteristic generally occurring in any process. The lack of *SR*-replication and *SR* itself in the benign processes suggests the opposite, that *SR*-replication may in fact be a characteristic unique to viruses and not occurring in benign processes.

10 User mode prototype test results

Testing the user implementation of SRRAT against the 66 test viruses was conducted in less than a day. The detection of *SR*-replication for the viruses is listed in Fig. 16. Out of 66 viruses in the test set 18 were terminated and flagged as attempting to execute *SR*-replication. When each of these viruses were terminated by SRRAT, the virus's *SR*-replication graph was created and saved to a text file. The *SR*-replication graph for the Alanis email worm is presented in Fig. 15. The graph shows the Alanis worm attempted *SR*-replication by first invoking the `Readfile` API function with itself as the source parameter, the function returned the memory address 1568460 pointing to the buffer containing the read portion of the virus, this function call makes Alanis an *SR* process for invoking a read general operation using itself as the source of the read, thus Alanis is reading itself. The virus then called the `Writefile` Api function using the memory address 1568460 as the source of the write and the destination was the file kerneldll32.api. When this function was called SRRAT established transitivity between kerneldll32.api and gallo.exe which is the virus file itself. This positive test for transitivity showed Alanis to be attempting *SR*-replication and was terminated. SRRAT always terminated these processes before the actual `Writefile` function was invoked, this prevented the *SR*-replication from completing. Furthermore the graph show the read operation was the first operation to occur dealing with *SR*, this is noted by the 1 next to the function name, the number 2 next to the write operation function name indicates this operation was then the second that occurred dealing with *SR*. The significance of this numbering is that SRRAT not only terminated Alanis for attempting *SR*-replication but it terminated Alanis on it's first attempt of *SR*-replication.

We classified the viruses that were not terminated into two groups: those viruses not hooked by SSRAT listed in Fig. 17 and those viruses that did not attempt *SR*-replication during testing which are listed in Fig. 18. Of the remaining 48 viruses that were not terminated, 15 of them executed and did not attempt *SR*-replication by the use of API function calls in a way that was detectable by SRRAT. Some of these viruses perform *SR*-replication in Kernel mode and others will only replicate when certain conditions are met and quite possibly these conditions were not present in the virtual machine. Interestingly, 5 of these viruses: `watcher.a`, `weakas`, `rega.a`, `delf.a` and `ezio.a` had previously attempted *SR*-replication during the theory validation testing. During that testing the *SR*-replication had been identified by the log files of API SPY 32. We later concluded that these 5 viruses that should have been detected were not as a result of the implementation of SRRAT missing some key functionality which prevented detection from occurring.

Of the 48 viruses not terminated by SRRAT, 33 were not hooked by SRRAT when execution commenced. SRRAT notifies us through a log file of it's activities while it runs. When it hooks a process the action is noted in the log file. When each of the 33 viruses listed in Fig. 17 were executed one by one, the SRRAT log file did not contain any entry documenting a successful hook of the executing virus. These viruses executed fully on the system with no monitoring of them being conducted by SRRAT. Some of these viruses actually run in Kernel mode and are able to bypass user mode

Fig. 16 Virus test results user implementation of SRRAT

Virus Name	SRR Detected	Virus Name	SRR Detected
Email-Worm.Win32.Alanis	Y	Net-Worm.Win32.Webdav.a	N
Email-Worm.Win32.Android	N	Net-Worm.Win32.Zusha.a	N
Email-Worm.Win32.Anpir.a	N	Net-Worm.Win32.Zusha.b	N
Email-Worm.Win32.Antiax	N	Net-Worm.Win32.Zusha.c	N
Email-Worm.Win32.Apost	Y	Net-Worm.Win32.Zusha.e	Y
Email-Worm.Win32.Asid.a	N	Net-Worm.Win32.Zusha.f	Y
Email-Worm.Win32.Bandet.a	N	P2P-Worm.Win32.Agobot.a	Y
Email-Worm.Win32.Bater.a	N	P2P-Worm.Win32.Agobot.b	Y
Email-Worm.Win32.Benny	N	P2P-Worm.Win32.Agobot.c	Y
Email-Worm.Win32.Bimoco.a	N	P2P-Worm.Win32.Agobot.d	Y
Email-Worm.Win32.Bormex	N	P2P-Worm.Win32.Aplich	N
Email-Worm.Win32.Borzella	Y	P2P-Worm.Win32.Blaxe	Y
Email-Worm.Win32.Botter.a	N	P2P-Worm.Win32.Cassidy	Y
Email-Worm.Win32.Burnox	Y	P2P-Worm.Win32.Compux.a	N
Email-Worm.Win32.Calposa	Y	P2P-Worm.Win32.Delf.a	N
Email-Worm.Win32.Canbis.a	N	P2P-Worm.Win32.Erdam	N
Email-Worm.Win32.Happy	N	P2P-Worm.Win32.Flocker.01	Y
Email-Worm.Win32.Klez.b	N	P2P-Worm.Win32.Gagse	Y
Email-Worm.Win32.Klez.c	N	P2P-Worm.Win32.Gedza.c	N
Email-Worm.Win32.Klez.d	N	P2P-Worm.Win32.Irkaz	N
Email-Worm.Win32.Klez.e	N	P2P-Worm.Win32.Kanyak.a	N
Email-Worm.Win32.Klez.f	N	P2P-Worm.Win32.Kapucen.b	Y
Email-Worm.Win32.Klez.g	N	P2P-Worm.Win32.Weakas	N
Email-Worm.Win32.Klez.i	N	Virus.Win32.Arch.a	N
Email-Worm.Win32.Klez.j	N	Virus.Win32.BCB.a	Y
Email-Worm.Win32.Sircam.d	N	Virus.Win32.Bee	N
Net-Worm.Win32.Doomran	N	Virus.Win32.Canbis.a	N
Net-Worm.Win32.Ezio.a	N	Virus.Win32.Jlok	N
Net-Worm.Win32.Maslan.b	N	Virus.Win32.Redemption	Y
Net-Worm.Win32.Nimda	N	Virus.Win32.Small.c	N
Net-Worm.Win32.Rega.a	N	Virus.Win32.Spreder	N
Net-Worm.Win32.Sasser.b	N	Virus.Win32.Watcher.a	N
Net-Worm.Win32.Syner.a	N	Virus.Win32.Zori.a	N

detectors such as SSRAT. But others do show usage of API function calls in user mode. These were not detected due to lack of functionality in the user mode implementation of SRRAT.

Virus Name	Virus Name
Email-Worm.Win32.Android	Email-Worm.Win32.Anpir.a
Email-Worm.Win32.Antiax	Email-Worm.Win32.Asid.a
Email-Worm.Win32.Bandet.a	Email-Worm.Win32.Bater.a
Email-Worm.Win32.Benny	Email-Worm.Win32.Bimoco.a
Email-Worm.Win32.Bormex	Email-Worm.Win32.Canbis.a
Email-Worm.Win32.Klez.b	Email-Worm.Win32.Klez.c
Email-Worm.Win32.Klez.d	Email-Worm.Win32.Klez.e
Email-Worm.Win32.Klez.f	Email-Worm.Win32.Klez.g
Email-Worm.Win32.Klez.i	Email-Worm.Win32.Klez.j
Email-Worm.Win32.Sircam.d	Net-Worm.Win32.Maslan.b
Net-Worm.Win32.Nimda	Net-Worm.Win32.Sasser.b
Net-Worm.Win32.Syner.a	Net-Worm.Win32.Webdav.a
P2P-Worm.Win32.Compux.a	P2P-Worm.Win32.Erdam
P2P-Worm.Win32.Gedza.c	P2P-Worm.Win32.Irkaz
P2P-Worm.Win32.Kanyak.a	Virus.Win32.Bee
Virus.Win32.Jlok	Virus.Win32.Small.c
Virus.Win32.Zori.a	

Fig. 17 Viruses not hooked by user implementation of SRRAT

Testing for false positives occurred together with usability as a real time monitor by running SRRAT on two desktop computers for three days. During this time the two computers were used under normal conditions plus some installation programs were purposely run in an attempt to cause SRRAT to produce a false positive. At the end of the three days SRRAT did not report a single process as having attempted SR-replication, no processes were terminated as a result of exhibiting possible virus behavior which ultimately means that no false positives were produced. The testing also showed the user mode implementation of SRRAT not to be a very practical real time monitor and detector. On five occasions, one of the computers had to be rebooted due to very slow operation resulting from SRRAT consuming high amounts of resources thus starving all the other processes running on the computer. On several occasions, SRRAT would crash when attempting to hook a process that was running at the time SRRAT was started. On a few occasions when SRRAT was terminated it still kept running and the process had to be terminated directly and ungracefully using Windows system tools. These problems were all implementation related and

Fig. 18 Viruses not exhibiting SR-replication in user mode SRRAT Testing

Virus Name	Virus Name	Virus Name
Virus.Win32.Watcher.a	Virus.Win32.Spreder	Virus.Win32.Canbis.a
Virus.Win32.Arch.a	P2P-Worm.Win32.Weakas	P2P-Worm.Win32.Delf.a
P2P-Worm.Win32.Aplich	Net-Worm.Win32.Zusha.c	Net-Worm.Win32.Zusha.b
Net-Worm.Win32.Zusha.a	Net-Worm.Win32.Rega.a	Net-Worm.Win32.Ezio.a
Net-Worm.Win32.Doomran	Email-Worm.Win32.Happy	Email-Worm.Win32.Botter.a

Fig. 19 Summary results Kernel implementation SRRAT Virus Test

	Email Worms	Peer-to-Peer Worms	Network Worms	IM Worms	Win32 Worms	Win32 Viruses	Total Amount
SR-replication	67	50	45	2	1	19	184
No SR-replication	28	28	38	0	0	89	183
True Positve	70%	64%	54%	100%	100%	18%	50%
False Negative	30%	36%	46%	0%	0%	82%	50%

despite them no false positives occur and virus detection had been successful in some cases.

Overall, we feel the testing of the user mode implementation of SRRAT had mixed results. On the one hand detecting a subset of the test viruses shows that detection of SR-replication in user mode is possible. The non-production of false positives further reenforces the idea that SR-replication is a characteristic unique to viruses. On the other hand, implementation issues due to lack of programming knowledge within the Windows environment may have led to some false negative production and a resource intensive implementation causing many problems that made it not to be the best choice as a practical tool for real time monitoring and detection of SR-replication in currently running processes. Only with increased programming experience in this area can a leaner, more robust and effective implementation tool be built.

11 Kernel mode prototype test results

A total of 14 days was need to test the Kernel mode implementation of SRRAT against the 367 test viruses and false positive testing. To test each virus required 8 days with the balance of days being used for false positive and usability testing. A summary of the test results is listed in Fig. 19. As we can see from the summary the overall testing result showed half of the test viruses to exhibit SR-replication behavior with the other have not exhibiting this behavior.

Recall the memory problems encountered during creation of the implementation were not overcome and these results were built from analysis of the log files produced by the Kernel Mode implementation of SRRAT. Besides the four main categories of viruses we also added one and two samples of two new categories which were Instant Messaging viruses and Win32 Worms. These are not major categories of our test set and they were added just to have at least one sample to make the test set representative of other virus categories.

```
00013263    NewZwReadFile ProcessName: ew-win32-Amus-a.exe
00013264    DestMemory: 1581032
00013265    Size: 65024
00013269    NewZwReadFile called!.
00013270    SelfReference Detected
00013271      File Path: \Documents and Settings\JAM-VX-
MACHINE\Desktop\ew-win32-Amus-a.exe
00013272      Bytes read: 65024
00013273    NewZwWriteFile ProcessName: ew-win32-Amus-a.exe
00013274    SourceMemory: 1581032
00013275    DestFile: \Documents and Settings\JAM-VX-
MACHINE\Desktop\ew-win32-Amus-a.exe
00013276    Size: 51782
00013283    NewZwWriteFile called!.
00013284      File Path: \WINDOWS\KdzEregli.exe
00013285      Bytes written: 51782
```

Fig. 20 SRRAT Kernel mode log file amus virus

Viewing the results by virus category is is clear that SR-replication occurred in the majority of viruses in the categories of: email worms, network worms, peer-to-peer worms, instant messaging worms and Win32 worms. The main cause of the 50/50 split in the overall results is directly related to the very high false negative rate produced by the Win32 viruses category.

The viruses showing SR replication did so in one of two basic forms. The first form was a simple read and write general operations. This form was not the dominant one in the log file analysis of the virus executions. A sample of this form is in Fig. 20. The second form and by far the most dominant was a sequence of operations that began with reading a file into memory followed by another reading of that memory to a new memory location and finally writing the memory to a new file. A sample of this form is in Fig. 21. The log files clearly showed multiple attempts to perform SR-replication by several of the test viruses.

The Win32 viruses which produced the highest number of false negatives, were for the most part the same viruses used in the testing of the theory validation. In that testing these viruses showed no attempts whatsoever of SR-replication. In testing these viruses again with the Kernel implementation of SRRAT those results were confirmed by the log file analysis. As it turns out by studying the log files, these viruses

```
00004801    NewZwCreateFile ProcessName: ew-win32-Borzella.exe
00004803    Filepath: \??\C:\Documents and Settings\JAM-VX-
MACHINE\Desktop\ew-win32-Borzella.exe
00004804    Desired Access:
00004805       GENERIC_READ
00004814    FileHandle: 56
00005031    NewZwCreateSection ProcessName: ew-win32-Borzella.exe
00005032    NewZwCreateSection FileHandle: 56
00005033    NewZwCreateSection OUT SectionHandle: 64
00005036    NewZwMapViewOfSection ProcessName: ew-win32-Borzella.exe
00005038    NewZwMapViewOfSection SectionHandle: 64
00005039    NewZwMapViewOfSection ProcessHandle: -1
00005046    After call ----------------------
00005047    NewZwMapViewOfSection BaseAddress: 13565952
00005053    NewZwWriteFile ProcessName: ew-win32-Borzella.exe
00005054    SourceMemory: 13565952
00005055    DestFile:
00005128       File Path: \WINDOWS\dllmgr.exe
```

Fig. 21 SRRAT Kernel mode log file Borzella virus

either: (1) make a copy of the virus itself into memory one or more times. In many cases this copy into memory is into the memory space of a currently running process. or (2) did not attempt to replicate in any fashion at all. This can be the result of the failure to find a suitable environment or victim file to replicate. Given that these viruses performed poorly during the theory validation testing it is not at all surprising those findings would be confirmed here as well.

Excluding the Win32 viruses category, the rest of the false negatives produced in the other categories result from none of their log files showing any attempt to executed SR-replication. In several cases these viruses did copy the virus itself into the memory of currently running processes. Interestingly there were a few viruses that never attempted replication at all. These viruses we consider false negatives as well because their lack of replication can be from the absence of a suitable environment needed to replicate. These viruses may in fact replicate and may even perform SR-replication given the environment facilitating this for each virus.

False positive testing along with testing for usability as a real time monitor of the Kernel implementation of SRRAT was conducted across 4 days. The log files produced by SRRAT were saved once per hour and were analyzed when the testing was completed. Analyzing the log files showed no attempts SR-replication by any of the processes recorded. Furthermore no SR operations were conducted either by any process. This gives further support to our assumption of SR-replication being a characteristic unique to viruses. From a usability standpoint this version is very robust not causing and crashing or slowdown of the system at any point during testing. Furthermore it was never disabled or terminated by any virus during testing.

Overall we were quite happy with the testing results of the Kernel implementation of SRRAT. The number of true positives was much higher than those produced by the user implementation of SRRAT and no false positives occurred. The one disappointment though not surprising was the high false negative amount of the Win32 viruses category. The Kernel implementation of SRRAT proved to be superior to the user mode in many aspects. It ran leaner, more robust, never crashed or slowed down the system at any time and proved capable of detecting far more viruses exhibiting SR-replication attempts then its user mode counterpart. This implementation detected 50% 127/259 viruses used for the theory validation testing and 87% 57/65 of the viruses used for the user mode implementation testing. Further analysis of the false negatives showed these viruses never attempted SR-replication into other files therefore they are not valid for the testing. The rest of the viruses causing the false negatives simply did not execute correctly in the testing environment and there cannot also be used as valid tests. The result was all the correctly functioning file infecting viruses that did attempt SR-replication were detected and thus from this point of view no false negatives were produced. Given this version is more capable of true positive detection then the user implementation version along with an overall 50% false negative production indicates to us this approach may be best used in conjunction with other known approaches to compensate their detection abilities with the false negatives produced by this implementation.

12 Discussion

Analyzing the results of all the testing two conclusions can be made about SR-replication. First it seems clear that this form of replication is unique to viruses and not to benign processes. It may therefore be suitable as a characteristic to differentiate between the viruses and benign processes. Second, implementing this theory is better suited at the lowest possible level of a system to maximize detection capabilities. This is evident from the much larger number of true positives produced by the kernel mode of SRRAT then the user mode of SRRAT.

The false negative production can be from one of two observations each with its own unique solution: First the viruses replicate at different levels from those in our implementations or they are able to avoid detection, this would require better programming techniques which is realizable. Second, these viruses may in fact replicate and our implementations simply lacks some functionality to detect these viruses and this functionality is not implementable. In this case, the best solution would be to compliment this approach with other known approaches with the assumption that the combination will reduce the false negatives while at the same time maintain or increase the true positives.

13 Related work

This research is an extension of the work presented by Morales et al. [23]. This research enhances the findings by

adding a new test prototype specifically to trace kernel mode system services. This additional testing provided excellent results with minimal false positives. This additional testing further shows SR-replication can be implemented at different levels of an operating system. More importantly it illustrates a better ability to detect more file infecting viruses then the user mode prototype indicating that implementation at a lower level can detect more viruses and simultaneously consume less system resources.

Analysis of system call arguments to detect malicious attacks is found in [24]. Several models are presented to characterize system call arguments. These characterizations are used to detect anomalous behavior. The research states two assumptions: (1) malicious attacks appear in system call arguments. (2) system call arguments used in malicious attacks substantially differ from arguments used during normal application execution. The models detect anomalies in the arguments such as unreasonably long string length, unusual characters and illegitimate values. The analysis of the arguments are used to create a score that determines if the system call is part of an attack. The models were trained with sequences of system calls giving no regard to the sequence but focusing only on the arguments. The testing results showed the models to be effective in detecting malicious attacks with low false positives. Our research also analyzes system call arguments without considering the sequence in which the system calls are made. The difference in our approach is we only consider write and read system calls used during replication of a virus. We do not detect anomalies in the actual system call arguments, instead we use the arguments to show relationships between read and write system calls. Our approach also requires no training, detection is done solely based on the appearance of read and write operations containing SR. These differences facilitate our approach to detect unknown viruses as opposed to Mutz et al. [24] where a false negative can occur if an attack not seen in the training session appears in a system call argument.

Skormin et al. present an approach to detect replication in self contained propagating malware [25]. Their detection is done by monitoring at run-time the execution of normal code under regular conditions. They monitor the behavior of each process and analyze the system calls, input and output arguments and the execution results. The Gene of Self Replication models the replication of a process using building blocks. Each block is a portion of the self replication process including opening, closing, reading, writing and searching for files and directories. The approach detected several viruses across many classes with little or no false positives. Our detection method focuses only on read and write operations that have SR. This is a simplification of the Skormin et al. approach which consider additional operations such as search, open, create as essential parts of a replication process. Our simplified approach reduces the overhead time and analysis needed to detect virus replication resulting in faster detection.

Christodorescu et al. present an algorithm to discover malicious behavior through differences between malware execution traces and benign execution traces [26]. The algorithm works by executing a known malware binary and a set of benign binaries. These executions are analyzed to form dependency graphs. These graphs show a trace of all system calls made during execution along with their dependencies from the passed arguments. The malware graph is minimized by comparing it to graphs of benign traces and trimming off the commonalities. This results in a specific trace graph of a known malware that is usable to detect variants of the same malware family. The trace is considered malicious because it was not found in the benign traces and is assumed not to occur in other benign traces. Our approach is similar to this one in that we also have system call graphs which are dependency related based on their arguments. These graphs are used to determine the presence of malicious behavior. A key difference is our work focuses on the specific behavior of SR-replication as opposed to Christodorescu et al. which attempt to detect several different malicious behaviors. SR-replication is present in all viruses and can be used to detect viruses belonging to several families and not just variants of one family. A second key difference is our approach does not require preliminary analysis of known viruses. Instead dynamic analysis of executing processes is conducted to identify SR-replication which has the advantage of detecting just released unknown viruses.

Jacob et al. formulate a formal framework for describing malicious behaviors with a language named Malicious Behavior Language (MBL) [27]. This language is used to describe virus self replication behavior as one of three forms: duplication, infection and propagation. In our research, direct graphs are used to trace the read and write operations of an executing process to attempt SR-replication detection. These graphs represent a subset of the replication descriptions given by Jacob et al. Our graphs allow for detailed tracing of how the replication was actually performed by a process, a feature not included in the MBL language. It is in fact possible to use the MBL representations of replication as a foundation for our graphs but not possible to use them directly as tracings of specific replication instances occurring in an executing process which our graphs do allow.

14 Conclusion and future work

This research has presented an approach to detecting file infecting virus behavior by identifying their attempt to replicate. This behavior is characterized by the SR property, which is a transitive relation existent when a process refers to itself in $read$ or $write$ operations during a replication

attempt. Self-reference replication ($SR-replication$) is the focus of our detection approach. One of the key strengths of our approach is the ability to detect both known and unknown file infecting viruses without prior knowledge. The detection approach is independent of the virus implementation, compilation, programming techniques and functionality. The approach can be implemented at various operating system levels to detect virus behavior which allows for fast detection with reduced overhead.

The results showed SR-replication to be occurring in most file infecting viruses and in none of the tested benign processes. Two implementations of the approach were created and tested. In both cases no false positives were produced. The overall conclusions of this research is twofold: First, SR-replication can be used as a characteristic to differentiate between viruses and benign processes. Furthermore SR-replication can detect known and unknown file infecting viruses when they execute without any a priori knowledge. This ability makes SR-replication well suited to detect newly released unknown file infecting viruses upon their initial attempts to execute SR-replication on a system. Second, a real time process monitor and virus detector on a system can be implemented and is usable using SR-replication as long as the implementation is at a low level of the computer system, for example the in Kernel mode. Future work includes expanding the approach of SR-replication to detect replication of viruses into memory and not just files. Another key aspect is to extend this approach to detect SR-replication across a network.

Acknowledgments This was supported in part by the National Science Foundation under Grant No. HRD-0317692. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied by the above agencies.

References

1. Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C. (eds.): Malware Detection. Springer, Heidelberg (2007). iSBN 0-387-32720-7
2. Szor, P.: The Art of Computer Virus Research and Defense. Symantec Press/Addison-Wesley, /Reading (2005). iSBN 9-780321-304544
3. Filiol, E.: Computer viruses: from theory to applications. IRIS International Series. Springer, Heidelberg (2005). iSBN 2-287-23939-1
4. Cohen, F.: A Short Course on Computer Viruses. Wiley Professional Computing, London (1994). iSBN 0-471-00769-2
5. Morales, J., Clarke, P., Deng, Y., Kibria, G.: Characterization of virus replication. J. Comput. Virology Special Issue on Theory of Computer Viruses Workshop (2008)
6. Adleman, L.: An abstract theory of computer viruses. In: CRYPTO '88: Advances in Cryptology, pp. 354–374. Springer, Heidelberg (1988)
7. von Neumann, J.: Theory of self-reproducing automata. University of Illinois, Tech. Rep. (1966)
8. Silberschatz, A., Galvin, P., Gagne, G.: Operating System Concepts. Wiley, New York (2001)
9. Golden, D., Pechura, M.: The structure of microcomputer file systems. Commun. ACM **29**(3), 222–230 (1986)
10. Linden, T.: Operating system structures to support security and reliable software. ACM Comput. Surv. **8**(4), 409–445 (1976)
11. Vx heavens. http://vx.netlux.org/. Acessed November 2007
12. Offensive computing malware repository. http://www.offensivecomputing.net. Acessed October 2007
13. Api spy 32. [Online]. Available: http://www.matcode.com/apis32.htm. Acessed November 2007
14. Microsoft windows sysinternals software. http://www.microsoft.com/technet/sysinternals/. Acessed November 2007
15. Windows api reference. [Online]. Available: http://msdn2.microsoft.com/en-us/library/aa383749.aspx
16. Nebbett, G.: Windows NT/2000 Native API Reference. Macmillan Technical Publishing, New York (2000). iSBN 1578701996
17. Symantec antivirus research center. http://securityresponse.symantec.com/. Acessed November 2007
18. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison Wesley Professional, Reading (2005). iSBN 0321294319
19. Vieler, R.: Professional Rootkits. Wrox Press, (2007). iSBN 0470101547
20. Windows api reference. http://msdn2.microsoft.com/en-us/library/aa383749.aspx
21. Kaspersky anti-virus. http://www.kaspersky.com
22. Vmware virtual workstation. http://www.vmware.com
23. Morales, J., Clarke, P., Deng, Y.: Detecting self-reference virus replication. In: EICAR 2008: Proceedings of the 17th Annual European Institute for Computer Anti-Virus Research Conference, 2008
24. Mutz, D., Valeur, F., Vigna, G., Kruegel, C.: Anomalous system call detection. ACM Trans. Inf. Syst. Secur. **9**(1), 61–93 (2006)
25. Skormin, V., Volynkin, A., Summerville, D., Moronski, J.: Prevention of information attacks by run-time detection of self-replication in computer codes. J Comput. Secur. **15**(2), 273–302 (2007)
26. C. M., J. S., K. C.: Mining specifications of malicious behavior. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE2007) (2007)
27. Jacob, G., Debar, H., Filiol, E.: Malwares as interactive machines: a new framework for behavior modeling. In: 2nd International Workshop on the Theory of Computer Viruses (2008)