J Comput Virol (2012) 8:1–13
DOI 10.1007/s11416-011-0157-5

ORIGINAL PAPER

Shadow attacks: automatically evading system-call-behavior based malware detection

Weiqin Ma · Pu Duan · Sanmin Liu · Guofei Gu · Jyh-Charn Liu

Received: 4 June 2011 / Accepted: 20 November 2011 / Published online: 20 December 2011

Abstract Contemporary malware makes extensive use of different techniques such as packing, code obfuscation, polymorphism, and metamorphism, to evade signature-based detection. Traditional signature-based detection technique is hard to catch up with latest malware or unknown malware. Behavior-based detection models are being investigated as a new methodology to defeat malware. This kind of approaches typically relies on system call sequences/graphs to model a malicious specification/pattern. In this paper, we present a new class of attacks, namely "shadow attacks", to evade current behavior-based malware detectors by partitioning one piece of malware into multiple "shadow processes". None of the shadow processes contains a recognizable malicious behavior specification known to single-process-based malware detectors, yet those shadow processes as an ensemble can still fulfill the original malicious functionality. To demonstrate the feasibility of this attack, we have developed a compiler-level prototype tool, AutoShadow, to *automatically* generate shadow-process version of malware given the source code of original malware. Our preliminary result has demonstrated the effectiveness of shadow attacks in evading several behavior-based malware analysis/detection solutions in real world. With the increasing adoption of multi-core computers and multi-process programs, malware writers may exploit more such shadow attacks in the future. We hope our preliminary study can foster more discussion and research to improve current generation of behavior-based malware detectors to address this great potential threat before it becomes a security problem of the epidemic proportions.

W. Ma · P. Duan · S. Liu (✉) · G. Gu · J.-C. Liu
Department of Computer Science and Engineering,
Texas A&M University, College Station,
TX 77843-3112, USA
e-mail: sanminliu@gmail.com

W. Ma
e-mail: weiqinma@cse.tamu.edu

P. Duan
e-mail: dp1979@cse.tamu.edu

G. Gu
e-mail: guofei@cse.tamu.edu

J.-C. Liu
e-mail: liu@cse.tamu.edu

1 Introduction

Malware, such as viruses, worms, trojan, spyware, rootkits, and botnets, are a prevalent and severe threat to Internet security. Malware writers have developed sophisticated techniques to evade existing signature-based detection tools. These evasion techniques include packing, code obfuscation [20], polymorphism, and metamorphism [23]. These techniques generate different variants of a malware program, i.e., every instance looks different (syntactically) but still maintains the same function (semantically). To nullify those evasion techniques defenders began to develop countermeasures [1,3,12,19,24] that aimed to recognize malware based on their behaviors, which are typically characterized by sequences/graphs of system calls since system calls are inevitable interaction interfaces between applications and OS. This behavior based solution detects malicious behaviors of malware families by matching suspicious system calls with existing malicious behavior specifications built on certain system call sequences or graphs [1,3,8,30]. Thus this behavior-based detection solution is more robust and hard to evade by using traditional attacking techniques.

We believe that knowing the limitations of the contemporary behavior-based malware detection research is an important problem. In this paper, we propose a new class of attack, "shadow attack", to counter behavior-based malware analysis by splitting critical system call sequences/graphs of malware and exporting them to separate processes. Specifically, shadow attacks create shadow process communication (SPC) channels between the rewritten malware and its shadow processes to achieve the original malicious functionalities. As of writing of this paper, most behavior-based malware detectors are designed based on malicious specifications in terms of system call sequences/graphs of individual single-process program (or these with simple inheritance/fork relationships). It is worth noting that the practically used system call sequence/graph behaviors are rarely just a single system call because that will have high false positive rates as likely many normal programs could use the same single system call with similar parameters as a malware does. Therefore, these behavior-based malware detectors could hardly detect shadow processes because they only contain small segments (e.g., just one system call) of the malicious behavior of malware.

As behavior-based malware detection becomes more prevalent, understanding its weaknesses and evasion vectors is very important to improve its resilience. We investigate the feasibility of indirect, implicit SPC design so that explicit in-host SPCs can be concealed with mixed implicit chains or even with the help of remote network coordination. We also adopt the technique proposed in [22] to hide local SPC among shadow processes by transforming data dependence to control dependence to evade dynamic information flow and data tainting based detections [25]. Given the myriad collections of process partition and coordination constructs, our study unveils the potential evasion vectors of this attack.

We have developed a compiler-level prototype tool, AutoShadow, for malware writers to automatically make source to source and source to binary transformation of C/C++ based malware codes. We applied AutoShadow to several real-world malware examples and found that our technique can successfully export critical system calls into shadow processes. Our preliminary results show that shadow processes can evade the detection from real-world behavioral detection/analysis tools such as Norman Sandbox [28].

In short, this paper makes the following contributions:

- We present a new, general class of attacks to conceal malware behaviors in multiple shadow processes and provide a systematic and in-depth study. Shadow attacks can be automated for partition and export of critical system calls or other functions into shadow processes. This kind of new attack can help us better understand the limitation of existing behavior-based malware detection techniques.
- We develop a compiler-level prototype system, AutoShadow, to demonstrate the practicality of automatic shadow attacks using several real-world malware samples.
- Experimental results suggest that shadow attacks can effectively conceal behaviors of malware and evade several current behavioral detection solutions. Although we provide some defense insights, the rich functionalities of SPC give malware writers a new ground to protect their properties. We hope this study can foster more discussion and research efforts to address this new class of attacks before they elevate to large scale malware outbreaks.

Table 1 Notations of our model

Description	Set	Instance
Process	$P = (p_1, \ldots, p_n)$	$\forall i \in [1, \ldots, n], p_i \in P$
Process state	$Q = (q_1, \ldots, q_m)$	$\forall i \in [1, \ldots, m], q_i \in Q$
System call	$S = (s_1, \ldots, s_k)$	$\forall i \in [1, \ldots, k], s_i \in S$

Table 2 System call relationship

Symbol	Description
$s_i \wedge s_j$	Both s_i and s_j happen, $s_i, s_j \in S$
$s_i \vee s_j$	Either s_i or s_j happen, $s_i, s_j \in S$
$s_i \rightarrow s_j$	s_i happens before s_j

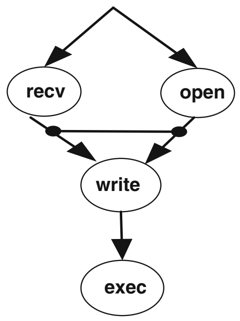


Fig. 1 Malware specification graph of download-execution: recv ∧ open → write → exec

2 Problem statement

2.1 Problem formulation and illustration

Follow what current behavioral detection approaches do, we model program behaviors at the system call level. That is, the behavior of a program is represented by the sequence of system calls, their I/O parameters and data. The behaviors of most malware can be tied to their system call sequences/graphs. However, when a shadow attack is added as a part of the malware, this sequence/graph is broken. Table 1 lists some basic notations that will be used in the rest of the discussion.

We assume that a system consists of n processes $P = (p_1, \ldots, p_n)$. Each process can be in any state of Q, which represents the state of process resources such as memory, CPU, file, and network. $\forall i \in [1, \ldots, l], s_i \in S$ denotes a system call. Table 2 lists different relationships between two

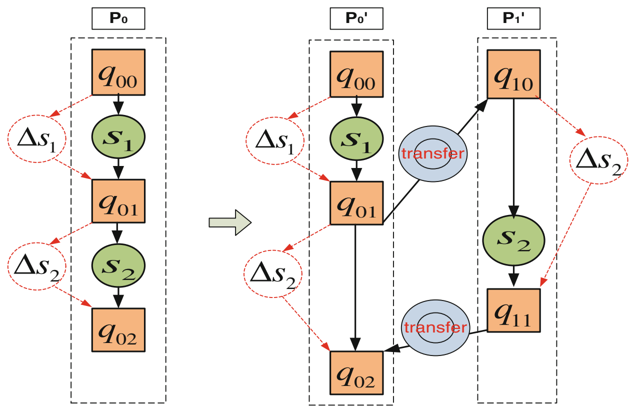


Fig. 2 Illustration of a shadow attack

system calls. γ denotes a set of relation operators between two system calls, i.e., $\gamma = \{\wedge, \vee, \rightarrow\}$. We define a *malware specification* as M = $s_1\gamma s_2\gamma s_{3\ldots}$, i.e., a set of sequence and parameter/data dependence relations of system calls.

Figure 1 shows a typical system call sequence of a malware specification which attempts to download an executable file from the Internet and then execute it [1], such as "egg" downloading and infection.

If an execution trace of a program p is denoted by T_p, $T_p = s_1 \quad \rightarrow s_2 \quad \rightarrow \cdots \rightarrow s_n$. We define $sub(T_p)$ as any possible sub-sequence of T_p. Then for a behavior-based malware detector, we define a detection function

$$\text{Detect(p)} = \begin{cases} \text{True, } \mathrm{sub_i(T_p)} \in \text{M}, \exists i \in [1 \ldots n] \\ \text{False, } \mathrm{sub_i(T_p)} \notin \text{M}, \forall i \in [1 \ldots n] \end{cases}$$

Definition A *shadow attack* can be regarded as a program transformation function SA_p : given a program p and some malware specifications M as inputs, SA_p will generate a multiple-process program p' with two properties: (i) *Detect(p')=False* while *Detect(p)=True*, (ii) p' has the same functionality as p.

Figure 2 illustrates how the functionality of a program remains the same when its system call sequence is exported to shadow processes. That said, the ensemble of shadow processes could achieve the same state as the original process had when same input parameters, output parameter values and return values are transferred appropriately between shadow processes. Here, the left side, P_0, is the original program, while the right side, P_0' and P_1', is the transformed program. q_{ij} denotes the state of process P_i for $\forall i \in [0, 1] and j \in [0, 1, 2]$, s_i denotes system call. We further define Δs_j as the impact of s_j over the environment including the changes to output parameters, return values and changes to system resources. Through transferring the same input parameter from P_0' to P_1', and transferring the output parameter value and return value from P_1' back to P_0', the ensemble of P_0' and P_1' will achieve the same state as P_0 had.

2.2 Bootstrapping shadow attacks

As illustrated above, a shadow attack is essentially a multi-process malware program. Executing multi-process shadow attack takes more elaborated steps than a single process attack. An important question is then how to get this multi-process malware executed on a single victim machine (i.e., the bootstrapping procedure)? Actually there are many ways to accomplish this. Here we provide two example scenarios to demonstrate it.

- The partitioned binaries can be spawned from one auxiliary process. For example, the partitioned binaries can be compressed into a self-extraction binary using compression tools. The advantage of this method is that malware can be easily and efficiently spread. The disadvantage is that the processes can be grouped or correlated relatively easily.
- Web-based malware infection is one of the most popular malware infection vectors nowadays. The shadow processes can be (drive-by) downloaded separately into the target machine and then executed separately. This approach is practical. Nowadays many Internet browsers, such as Google Chrome, Microsoft Internet Explorer, are all implemented in multiple processes. In such an environment, malware can be downloaded by different processes via the same (or different) malicious URLs and executed separately. Correlation of multiple shadow processes will likely be harder.

Although we give two example scenarios above, we note that with the increasing popular use of multi-process programs, and dynamic, complex and variant existing infection vectors, multi-process malware is very feasible with many possibilities to arrive at end users.

3 Shadow attack design

In principle, a shadow attack can export any critical system call in a malware specification to different shadow processes so that any such specification-based malware detector will be hard to detect it. One of the key questions in designing such a shadow attack is how to coordinate these shadow processes so that they can still accomplish the original functionality as a whole system. In this section, we first show the general architecture of shadow attack malware. We discuss how we can design shadow process coordination/communication (SPC) with different levels of sophistication, flexibilities and stealthiness. In particular, we show the design of indirect, implicit local SPC using remote network coordination. We also discuss to hide local SPC from taint-based data dependence tracking among processes. We leave detailed

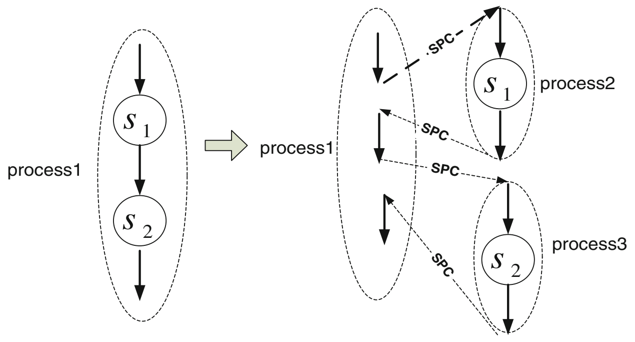


Fig. 3 Architecture of shadow attacks

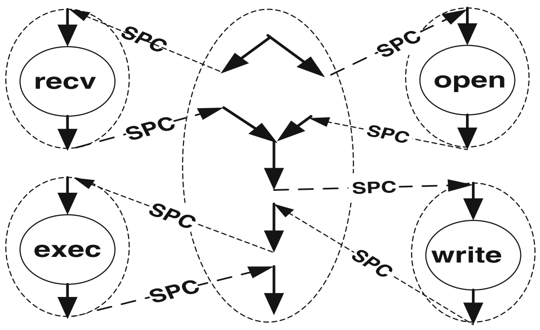


Fig. 4 Shadow attack version of example shown in Fig. 1

discussion on how to automatically partition a given malware code to its shadow-process version to next section.

3.1 Architecture of shadow attack malware

The design space of Shadow Process Coordination/Communication (SPC) includes covert channel communication [14] like covert cache [31] and branch predictor [10], and other traditional communication approaches like Inter-Process Communication (IPC) [32], environment variables, files and registries. In our shadow attacks, we mainly target to export critical system calls from their original process/code to new generated (shadow) processes.

For example, in Fig. 3, two critical system calls are initially in *process1*. We export s_1 and s_2 to two processes *process2* and *process3*, respectively. Then, *process1* communicates with *process2* and s_1 is executed in *process2*. *process2* then communicates with *process1* to return the results of s_1. The same procedure is used to execute s_2. As a result, the functionality of *process1* is maintained since both s_1 and s_2 are executed in the original order.

Using this shadow attack, the previous presented download-then-execute malware example (in Fig. 1) can be transformed to the shadow process version illustrated in Fig. 4.

Our shadow attack uses *marshalling* [33] to transfer objects between two separate processes. Basically, there are two ways to transfer a file descriptor in Unix: Unix Domain Socket and Stream Pipe [32]. In addition, some general in-host communication mechanism can transfer file descriptors between processes as proposed in [16]. In Windows, at least two ways can be used to transfer socket handlers: WSADuplicateSocket()—a function in the Windows Socket 2 library in the context of the source process which created the socket; Win32 DuplicateHandle() function.

3.2 Hiding local SPC through remote network coordination

To make local shadow processes' communication harder to be noticed, we also design indirect/implicit SPC through remote network coordination, i.e., we coordinate local shadow processes via outside stepping nodes. As shown in Fig. 5, we export critical system calls s_1 and s_2 from *process1* to *process2* and *process3*, respectively. *process1* communicates with an outside host A through network channels. This host A then communicates with another host B which then communicates with *process2* and *process2* executes s_1. After the execution, *process2* communicates with *process1* through other outside hosts (A and B) to get results of s_1. Similar procedure is taken when s_2 is executed in *process3*. The advantage is that it is difficult to detect local SPC links between local processes because there is no direct observable connection. These local processes also do not have network-level correlation because they talk to different remote machines. In this way, it is a very challenging task to find their relationship of different local processes that coordinated through different networks.

3.3 Further discussion on SPC design space

A plurality of in-host and network based coordination and communication approaches exist on both Linux/Unix and Windows operation systems. On Linux/Unix, communication methods include message queues, semaphore sets, Unix domain sockets, and shared memory [32]. On Windows, communication methods include Clipboard, COM, DDE, File Mapping, Mailslots, Pipes, RPC, shared memory Windows Sockets and web services. Our shadow attack utilizes communication methods for data/parameter communication and synchronization between processes. Next we use the following examples to illustrate functionality, advantages, and disadvantages of some communication methods:

(1) *Unix domain sockets*: Sockets transfer data between processes using buffers in the kernel memory. A process can exchange a file descriptor to another process using sendmsg() and recvmsg(). The file descriptor is related to process migration and socket migration. For example, the technique used in MSOCK [14] can be used for socket migration between two processes.

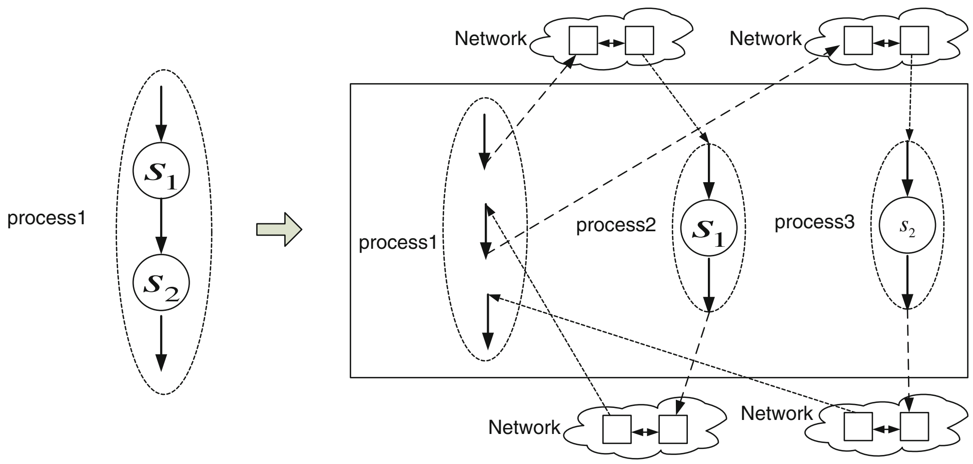


Fig. 5 Hiding SPC through remote network coordination

(2) *Shared memory*: When there is not much data type conversion in the shared data between processes, shared memory methods have better performance compared with other SPC methods (e.g., pipe and sockets). We can utilize certain techniques, e.g., mapping physical pages to two distinct virtual addresses, to complicate the detection of the use of shared memory. On the other hand, shared memory method can only be used in a single machine, while other SPC methods may be used on different machines in a network.

(3) *SOAP*: *Simple Object Access Protocol (SOAP)* can be used for data communication or calling methods in web services through the Internet. In SOAP, data is transmitted in XML files. The advantage of this approach is that it can easily pass through many firewalls since XML files can be transmitted through standard HTTP requests. The disadvantage is that XML files in SOAP use more bandwidth and memory as compared with direct data access methods like shared memory.

In addition to these well-known SPC mechanisms, one can easily use other more advanced SPC approaches (e.g., covert channels), especially mixing of different SPC mechanisms, to complicate the detection of SPC. Various covert channels have been proposed before. In [31] the authors demonstrate that shared access to memory caches can provide an easily used high bandwidth covert channel between threads. While in [10] the authors introduce a Simple Branch Prediction Analysis (SBPA) attack which analyzes the CPU's Branch Predictor states through spying on a single quasi-parallel computation process.

To build a mixed and indirect SPC Chain, as shown in Fig. 6, we take out the critical system call s_1 from *process1* and place it in *process2*. To provide the communication between *process1* and *process2*, we let *process1* communicate with the SPC method *file* at the point right before the execution of s_1. Then *file* communicates with *shared memory*, *shared memory* communicates with *socket* and so on, until we reach *process2*. After the execution of s_1, we return to *process1* at the point right after the execution of s_1, based on the communication provided by the same SPC mechanisms. The advantage of this mixed implementation of SPC mechanisms is that it is more difficult to detect the hybrid communication between *process1* and *process2*. In this communication, because many SPC objects, files, or resources are also used by other regular programs, it is challenging for detectors to differentiate the partitioned malwares from other regular programs.

Another aspect of shadow attack is that it may significantly increase the resource requirements for tracing and detection of multi-process applications. Because interleaving of multiple processes could lead to path explosion, the detection of a specific behavior would become significantly more difficult.

3.4 Hiding SPC from information flow tracking

In our attack, the system calls in multiple processes usually carry the same parameter data (e.g., file name) to fulfill the malware's functionality. A possible technique to detect related processes is to correlate them by tracking the information flow, e.g., taint the system call parameter (e.g., file name) and track the data dependence [25]. However, these techniques are mainly used in offline analysis instead of real-time detection because of their high overhead. In addition, we can systematically transfer data dependence into control dependence [22] through an automated source code rewrite procedure. This kind of technique is particularly well suited for evading taint-analysis-based detection.

We first use two simple examples to introduce control flow dependence, which can be categorized by two types: explicit control flow dependence and implicit control flow dependence.

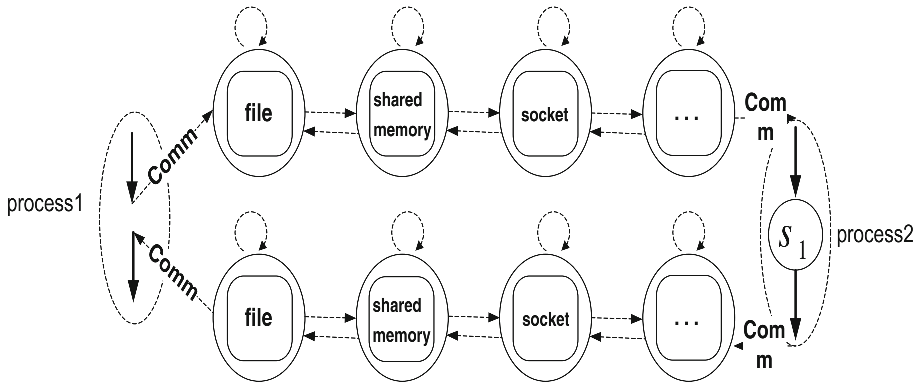


Fig. 6 Mixed, indirect, implicit SPC chain

```
Explicit control flow:

if  x = = a

     y = a

Implicit control flow:

for (int i = 0; i<255; i ++) {

        tmp = 1;

       if (x! = j)

           tmp = 0;

       if (tmp = = 1)

           y = j;

}
```

Then we show how to use the idea of control flow to hide the data dependence of different types of data in system calls, e.g., *char, string, int, struct*. We use an example to illustrate how to transmit the value of a *char* type parameter *x* to another *char* type parameter *y* without revealing their data dependence relationship.

```
  char Convertchar(char x)

          char y;

          for (int i = 0; i<255; i ++)

          {

                  if (x = = i)

                  {

                           y = i;

                           break;

                    }

           }

return  y;
```

We can use the same idea to transmit the value of a *string* parameter *x* to another *string* parameter *y* by applying control flow to each character in the string.

```
char * ConvertString (char * x, int xlength)

{

          char * y = new char[xlength];

          for (int i = 0; i < xlength; i ++)

          {

             y[ i ] = ConvertChar(x[ i ]);

           }

}
```

For numeric types, e.g., *int* or *float* in C or C++, we can first convert them into a *string* type. Then we use the string-based control flow transformation as shown above and convert the string back into a numeric type after transformation. For *struct* type parameters, we can follow a similar procedure because they are constructed with *int* and *string* parameters.

4 Automating shadow attack

We have implemented a prototype tool *AutoShadow* as a proof of concept to automate shadow attacks, i.e., it can automatically generate a shadow-process version of malware given an original malware source code. AutoShadow is built on the intermediate representation (IR) bytecode using compiler frameworks such as LLVM [2] and Phoenix [21]. We selected LLVM because of its mature features in Linux. AutoShadow analyzes source code and locates critical system calls to launch the shadow attack. For the Phoenix based example, some manual patching was done to demonstrate the feasibility.

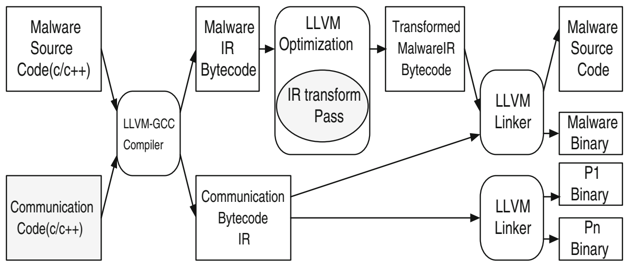


Fig. 7 AutoShadow design architecture

4.1 AutoShadow design architecture

Figure 7 shows the design architecture of AutoShadow. It takes malware source code as input and generates the IR bytecode through the LLVM-GCC compiler, which includes both malware bytecode and communication bytecode. The malware bytecode can be further optimized and transformed based on our pass/plug-in. Then, the transformed bytecode, together with the communication bytecode, is passed to the LLVM linker to generate ELF executable format binaries. These binaries can then be distributed to execute real attacks. Bytecodes can be converted to the source code form so that they can be further morphed into new shapes and structures to fulfill the same malicious functionalities.

4.2 Code analysis and transformation

As shown in Fig. 7, our IR transform pass performs transformation on the IR bytecode of malware and is loaded by LLVM as a shared library during the optimization of the IR code. Our transformation technique converts the IR code into new malware IR bytecode. We replace the critical system calls with an external function to communicate with another process. The new transformed bytecode will be executed in multiple processes, which achieves the process partitioning.

Our pass inherits the existing *CallGraphSCCPass* [2] provided in LLVM, traverses all system calls and finds the candidate critical system calls. *CallGraphSCCPass* is used to traverse the call graph of a program. The candidate system call will be replaced by the external function in all instructions that invoked the candidate system call. In our pass, the alias analysis of LLVM is used to update the change of call graph. Algorithm 1 shows the transformation procedure of in-host SPC based shadow attacks.

Algorithm 1: in-host SPC

```
1. Traverse all the system calls
2. Determine whether a system call s is a candidate system call
3 Create a new external function s' using the arguments and return types of s
4. Replace s with s' in all instructions that called s
5. Update the call graph using alias analysis
```

4.3 Communication code generation

Parameter data serialization For system call partitioning, the most important and challenging task is to decide whether a system call can be exported from its original process to another process to cross process boundaries. Two aspects need to be considered: the type of the system call parameters/data and the semantics of the system call.

For the first aspect, the parameters of a candidate critical system should satisfy two conditions: the parameter communication part can be abstracted as a layer of object serialization and the parameters can be put into the medium like shared memory, message queue, or file. The communication data between processes can be categorized into two types: standard variable type like *int*, *char* and*struct* and *void**, system resource object like *file descriptor* and *socket descriptor*.

• *Standard variable type*: for *int* type parameters, we can easily convert *int* to *string* and then transfer *string* type parameters through SPC to other processes. For *void**, type parameters, it is very difficult to decide which actual types of pointed parameters need to be transferred. However, the number of system calls is limited and we only focus on those critical system calls that are attributable to the description of malware behavior. Therefore, we can know the specific semantics of these system calls and which type of data should be transferred for the specific system call by manually checking its definition and usage. Then, we can transfer *void** to *string* or *struct* or others. For example, we can convert the *void** into *char** or some *struct* type based on semantic of the system call. Also we can know that there is a variable to tell the length of the *char**. For other complex types of objects, we can use boost serialization [29].

• *System resource object*: this type of data includes *file descriptors* and *socket descriptors*. In Linux, we used Unix domain socket and stream pipe to transfer file descriptors. In Windows, we use *DuplicateHandle* and *WSADuplicateSocket* to clone socket descriptor. *WSADuplicateSocket* fills up a *WSAPROTOCOL_INFO* structure with information of existing socket connection. *WSAPROTOCOL_INFO* structure can be transferred by using common communication mechanism like pipe, shared memory, and socket. This could be used for transferring handles in network-coordinated SPC described in Sect. 3.2.

Since we can transfer the file descriptor in addition to directly copying the filename, it is far more expensive and difficult than tracking filename duplications in traditional approaches. Since a file descriptor is a quintessential example of capability object, data transferring could be generalized to an object capability system. More generally speaking, the system call partitioning could be easily extended to function and module level.

We also need to consider several system calls with special semantics. For example, fork() cannot be exported because

Table 3 Example system calls supported in AutoShadow (more system calls could be easily added)

Function category	System call
File I/O operation	open, read, write
Network	socket, connect, recv, send, read, write
Process management	exec,execl

it inherits context of current process. Another example is getpid() because its execution result is different in different processes. In windows, similar function like fork() can be used [19].

Our implementation focused on critical system calls that are heavily used in malware and can be exported to another process. Currently, AutoShadow supports a list of system calls as shown in Table 3. The system calls are frequently used to identify malicious behaviors. For example, they are used to manage processes (e.g., execute an updated binary), operate on file and I/O (e.g., access confidential file, infect existing executable, modify registry), access network (e.g., egg downloading, botnet C&C, spamming and proxy). Some simple examples of malware malicious behaviors and their associated system calls are shown in Table 4.

4.4 Prototype implementation

In current version of AutoShadow, we have implemented in-host shadow attacks. The implementation of remote-network-coordinated shadow attacks follows similar principle and is not included in current version.

For in-host SPC, we implemented pipe and socket as example representative mechanisms. As shown in Fig. 8, malware.c source code can be transformed into three files to extract system call write(). After transformation, the system call write() in malware.c was replaced with an external function called SPC_write() in SPCwriteClient.c which serves as a client to send required data for write() in SPCwriteServer.c. SPC_writeSrv() function will receive data and convert data into needed arguments for write(). Then the critical system call write() is executed. After its execution, output parameters and return values are sent to SPC_write() in SPCwriteClient.c which is called by malware.c. Note that SPC_write() could be further obfuscated to avoid to being statically fingerprinted.

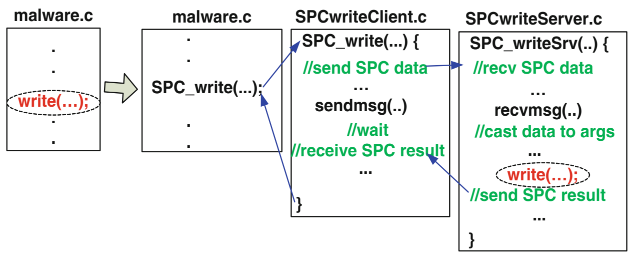


Fig. 8 Code transformation example

4.5 Windows compiler tool implementation

In Windows, we use Phoenix to make the transformation. We generated binary from source codes and used the PEReader to construct IR from the binary. We imported external functions from a dll to replace target system calls or functions. Finally, we use PEWriter to write IR to new binary files.

5 Evaluation

We evaluated the effectiveness and performance of our tool. Since we target behavior-based malware detection, we first filter out tradition signature-based AV tools. Some AV tools claim to combine both signature-based detection and behavior-based detection. They are not suitable for our evaluation because it is difficult for us to differentiate whether a malware sample was detected by its signature or its behavior. Many research prototype systems such as the tool in [1] are unfortunately not publicly available for testing. As a result, we submitted the binary to two online malware behavioral analysis/detection tools: CWSandbox [26] and Norman Sandbox (we also implemented a simple detection tool similar to [1] as described in Sect. 5.4). We have also tried Anubis [13]. However, Anubis could not properly recognize and analyze our multi-process-based binaries inside the self-extracting archive form. Thus, we exclude it in our evaluation.

5.1 Effectiveness test via online analysis

We compare detection results of a single-process based malware with that of the transformed multiple processes. We extracted relevant critical system calls, i.e., connect, send, recv, CreateFile, WriteFile, from the source code of

Table 4 Example malware behavior and system calls (notations are the same as in Table 2)

Malicious behavior	Key system call sequence
Download and execute	(recv^write)^(open→write)^(recv→write)^(write→exec)
Proxy	recv(socket, buffer) →send(destsocket, buffer)
Modify registry	(RegCreateKeyA→DeleteValueA) ∨ (RegCreateKeyA→RegCloseKeyA)

```
DoTcpConnectExecute.exe:INFECTEDwith
W32/Downloader (Signature: NO_VIRUS)
[ DetectionInfo ]
*Filename:C:\analyzer\scan\DoTcpConnectExecute.exe.
* Sandbox name: W32/Downloader.
[ Network services ]
* Connects to "students.cs.*.edu" on port 80.
*OpensURL: students.cs.*.edu/*/notepad.exe.
[ Security issues ]
*Starting downloaded file-potential security problem.
```

Fig. 9 Norman report of single process

```
self.exe : Not detected by Sandbox
(Signature: NO_VIRUS)
[ DetectionInfo ]
* Filename: C:\analyzer\scan\self4.exe.
* Sandbox name: NO_MALWARE
[ Network services ]
* Connects to "students.cs.*.edu" on port 80.
```

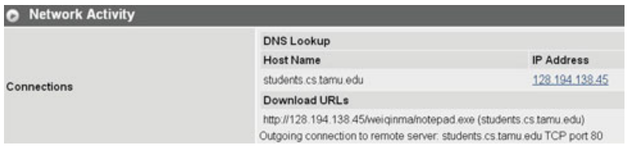
Fig. 10 Norman report of shadow processes

Agobot [4], which implemented the malicious behavior of downloading a file from a URL and then executing the file. We compiled the code into a binary, and then submitted the binary to two online malware behavioral analysis tools: CWSandbox and Norman Sandbox.

We evaluated the shadow attack based on the extracted code from Agobot, which consisted of three different binary executable files representing connect process, download process and execute process, respectively. The three binary files were combined into one self-extracted binary using winrar (as the simplest scenario described in Sect. 2.2) because these online services only accept single binary submission. We then used winrar to unfold and execute the binaries to verify the original functionality as downloading and executing.

We now describe the detection outcomes of two cases in the Norman Sandbox. For the single-process based binary, the Norman sandbox detected it as "Win32/Downloader" as shown in Fig. 9. For the shadow-attack based binary, the Norman sandbox reported it as not infected as shown in Fig. 10. The Norman Sandbox's Report showed the opened URL in the single process detection result. While for the shadow processed binary it only showed the connection.

Then, we submitted the multi-process-based binary (.exe) file to CWSandbox for analysis. For single-process based binary, CWSandbox clearly reported the network activity via a download URL as shown in Fig. 11. However, for the shadow processes based binary, it only reported the outgoing connection without indicating the download URL, as shown in Fig. 12. These two examples demonstrate the effectiveness of shadow attack.



Network Activity

Connections

DNS Lookup

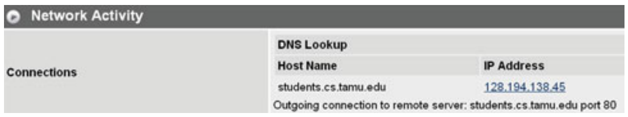
Host Name	IP Address
students.cs.tamu.edu	128.194.138.45

Download URLs

http://128.194.138.45/weiqinma/notepad.exe (students.cs.tamu.edu)

Outgoing connection to remote server: students.cs.tamu.edu TCP port 80

Fig. 11 CWSandbox report of single process



Network Activity

Connections

DNS Lookup

Host Name	IP Address
students.cs.tamu.edu	128.194.138.45

Outgoing connection to remote server: students.cs.tamu.edu port 80

Fig. 12 CWSandbox report of shadow processes

5.2 Evaluation using real-world Malware

We further evaluated the efficiency and feasibility of AutoShadow on several real-world malware programs. Our evaluation was performed on a Linux machine with an Intel core 2 Duo 2.40GHz processor, 6MB cache, and 2GB of memory.

The results in Table 5 show that AutoShadow can automatically extract critical system calls in well-known malwares like Q8bot (a bot program), Apachworm, Computer_dunno, and Kaiten. After partition, Q8bot becomes a new malware with four shadow processes. Obviously, by splitting, creating, sending, and receiving behavior into multiple processes, behavior-based detection tools that attempt to capture the C&C (command & control) pattern (e.g., [25]) in one process are likely to fail. The apache worm has a mailing behavior. This mailing behavior can be modeled using system call sequence as "connect() ˆ write() ˆ write("Hello") ˆ write("RCPT")", which can be readily detected by existing behavior-based detectors (e.g., [1,3]). This behavior specification is exported to multiple shadow processes, and therefore they do not exist in any single process. Corompuţer_dunno and Kaiten are two examples of bot programs that are designed to launch DDoS attacks. They have C&C patterns that could be potentially detected by detection approaches like [25]. Similar to the Q8bot, AutoShadow successfully partitions its critical system calls and thus make it undetectable. We also report the transformation time. For example, Apach-worm has a total of 2,255 lines of code. It costs AutoShadow only 0.05 s for the partition. As shown in Table 5, other three malware programs take less than 0.05 s for the transformation. These tests on the four real-world malware examples demonstrate the high efficiency of AutoShadow.

5.3 Performance

We also evaluated the process communication cost by using the malware example in Fig. 1.

Table 5 Evaluation results on malware examples

Malware name	Code line	Transform time (s)	File related system call	Network related system call	Process related system call
Apach-worm	2,255	0.05	open write 9 read 4	recv 1	execl 1
Q8Bot	926	0.02		socket 7	
				recv 5	
				write 1	
Corompuṭer_dunno	254	0.005	write 6	socket 1	system 1
				connect 1	
				send 6	
				recv 6	
Kaiten	971	0.025	write 1	socket 7	
				write 1	
				recv 5	

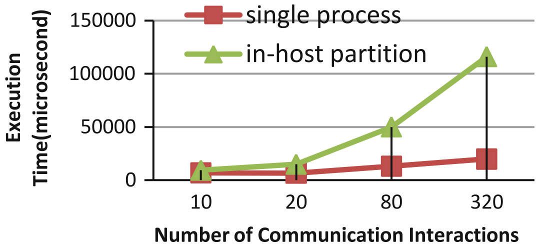


Fig. 13 Performance time

Since the main overhead occurred when two shadow processes communicate, we changed the number of such communication interactions by using different buffer size of write(). Figure 13 shows the total execution time of the malware in two cases: before partition (single process) and after partition using in-host communication. We can see that when there are less than 10 interactions between shadow processes, the attack have similar execution time as the original malware. With the increase of interactions, the shadow attacks cost more time on the communication. It is worth noting that although multi-process malware is slower than the original single-process malware, we believe most malware writers might still favor more evasive multi-process malware if the task is not time-critical. In addition, malware writers can always optimize communications between shadow processes in reality, e.g., choosing selective critical system calls and limiting the number of shadow processes.

5.4 Estimating possible detection cost of multi-process Malware

In theory, one could defend against shadow attack with adequate monitoring of process activities to determine their relationship, data flows, etc. That being said, it is important to gain a realistic sense of the performance in these countermeasures. We use the Linux system calls to gain a glance of the design complexity and run-time costs.

Firstly, we implemented a very simple single process behavior-based malware detector, similar to the idea of [1]. In order to monitor system calls, we can employ either Ptrace or kernel modifying. We choose using Ptrace for simplicity. We use shadow attack to transform Agobot, which contains five malicious behaviors including: (a) download and execute; (b) Remote-Initiated Network Download; (c) Remote-Initiated Send Email; (d) Remote-Initiated Sendto; (e) TCP Proxy. Not surprisingly, our result showed that Agobot successfully evades the single-process detector after shadow attack (while it can be detected before shadow attack).

In a generic defense system, one needs to group *correlative* processes which have direct or indirect communications with each other. After grouping correlated processes, one could implement a system call dependency graph of multiple processes using relevant state information as its inputs. To monitor the System-V IPC, we need to monitor the system call shmget()/msgget()/semget(), and group processes by comparing parameters of these system calls. Besides, to monitor in-host SPC method of pipe, socket, and disk file, the *inode* value of each file descriptor opened by all processes (called IPC object hereafter), needs to be analyzed. In doing so, we can tell which processes have ever accessed the same IPC object to communicate with each other. We employ a user mode approach, that is whenever a pipe/socket/disk-file related system call is triggered, the change of file descriptors in /proc/fd needs to be logged, and then analyzed. Figure 14 displays the architecture of our example multi-process malware detector.

For each correlated process group, we can recover a global system call sequence according to the specification. Without considering any specific malware, we conducted a series of experiments to gain a sense of the monitoring cost. We use a benchmark program which makes five system calls:

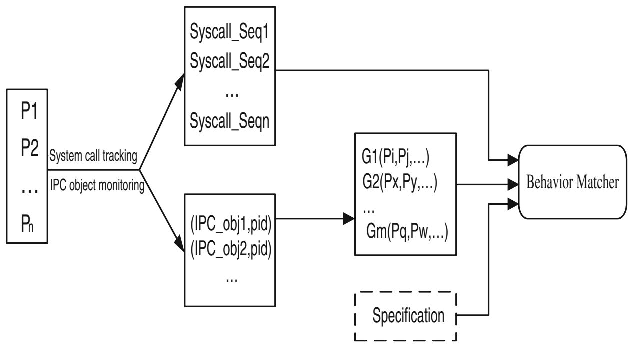


Fig. 14 Example architecture of a multi-process malware detector

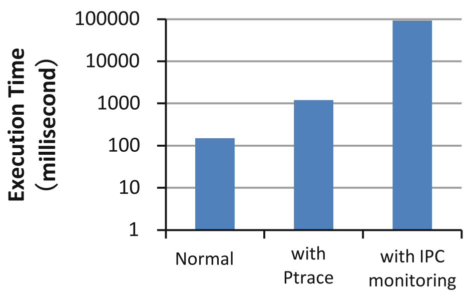


Fig. 15 Time cost induced by multi-process monitoring/correlation

fopen->read->write->fclose->execv, with 1,000 iterations. And measure the running time of this program, normally, with ptrace monitoring and with /proc/fd monitoring. Figure 15 shows that the running time of this program increased from 150 to 1,195 and 93,121 ms, respectively, when Ptrace and /proc/fd monitoring were activated. The checking of /proc/fd directory is trigged whenever fopen/fclose are called. Thus it is time consuming.

We note that our above naïve example detector is merely for the purpose of estimating *one possible* detection cost for multi-process malware. It is just an example, and by no means a best, realistic, nor optimal design. One may easily argue that there can be much better design in terms of both effectiveness and efficiency. However, the message we want to convey from this naïve example is that fully real-time correlating processes (through explicit or even implicit SPC) can be much more expensive than single-process based monitoring. To our knowledge very few tools support multi-process analysis due to the high cost of blind enumeration on grouping of processes to one or more malwares. In addition, inter-process coordination can be made indirect and hidden using many ways (even remote-network-coordination) to make it more difficult for multi-process malware detection. We need further research to study more efficient real-time solution for correlating processes communicating using different SPC.

We also need to improve the effectiveness of accurately correlating multi-process malware that uses indirect/implicit SPC. A possible way is to perform fine-grained kernel-level global processes and system object tracking and correlation analysis. In [18], BackTracker, a VM-based monitoring method, is proposed for such a fine-grained tracking. However, this approach is very expensive and used only for off-line analysis instead of real-time detection purpose. Thus, we believe how to build an effective and efficient shadow attack detector is still an open problem.

6 Discussion

Our approach could easily be combined with current signature-evading techniques such as metamorphic viruses [34]. By doing so, AutoShadow could be able to evade both signature-based and behavior-based malware detector.

To defend against SPC based shadow attacks, all possible in-host SPC activities might need to be monitored to screen for correlated processes. Some major obstacles must be overcome by defenders. First, a detection approach may have high false positive rate because normal processes also use in-host communications. In particular, with the increasingly used multi-core CPUs and increasingly adopted normal multi-process programs, this issue could become worse. Second, with conversion of data flows to control flows, it is not easy to efficiently track the parameter data dependency between processes. Finally, the use of information flow and data tainting are relatively expensive in practice because of their high operational overhead. The remote-network-coordinated shadow attacks could further eliminate the direct observable communications and makes the detection even harder.

We believe that it will be useful to consider correlations and dependencies between processes based on *system objects/resources*. For example, the correlation between two processes may be established when they are found to operate on a same file. However, this approach must also address the false positive issue and the high overhead (as shown in BackTracker [18]).

7 Related work

Program/Malware partition: One related work to AutoShadow is k-ary theoretic malware model [6] that provided a theoretical analysis of the hardness of detection. Source code fragmented malware has been proven to be NP-Complete in general [6]. Detection of shadow attack is thus also NP-complete since shadow attack is a form of fragmented malware at behavior level. Our approach is different from [6] because we fragment malware into different executable binaries at behavior level and we provide a practical, automated compiler-level solution. Several systems automatically generated distributed programs by applying static code analysis [2]. In theory, program partition can also be abstracted as a NP-hard problem [11]. In addition, *Mimicry attack* [5,27] obfuscated the order of system calls to evade system call based IDS. Another similar multi-process based work [7] proposed a code injection technique, which writes some malicious code to other processes for detection evasion. It has the weakness that the behavior of writing data to the memory space of other process is uncommon and suspicious, which could cause attention of detection tools.

Behavior-based malware detection: A semantic-aware detector using code template of instructions was developed in [3]. In [12] specification language was mined for malicious behavior. Layered behavior graph using system calls was proposed in [1]. These detectors focus on one single process rather than multiple processes. Jiang et al. [15] detected the break-in point of worms by assigning different colors to different processes derived from different services. It cannot detect covert communication channels. Panorama system [17] was proposed to detect privacy breaking malwares by tracing the information flow of access and processing sensitive information in an offline fashion. BackTracker [18] traced back to vulnerable points by monitoring and logging system calls, files and processes. It is still an offline tool.

It was recently proposed in [9] that one could counter the behavior-based detections by using Time-Lock Puzzle (TLP). It utilizes cryptographic techniques to ensure that a message is never revealed until certain right moment. Thus, TLP prevents the Anti-Virus (AV) engine from discovering the encryption key and thus hide suspect codes without being identified [9]. Another technique, system call obfuscation [29], evades malware detectors by replacing certain system calls with general control system calls. Our work is different from these in that we propose a new class of attacks on behavior-based detectors.

8 Conclusion

In this paper we show the models and techniques of shadow attack. The essence of this kind of attack is to export malicious behavior specifications from a malware program to multiple shadow processes. We implemented a compiler-level prototype tool to demonstrate its feasibility. Our preliminary results show that transformed malware could evade or counter existing behavioral analysis tools. Several research problems still remain open. For example, from attack point of view, how to launch optimal shadow attach in terms of minimal number of processes, resource consumption, and communication cost. More importantly, from defense point of view, how to efficiently and effectively defend against this new threat still requires further research.

References

1. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J. C.: A Layered Architecture for Detecting Malicious Behaviors. In: Proceedings of the 11th international Symposium on Recent Advances in intrusion Detection (RAID'08) (2008)
2. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04) (2004)
3. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-Aware Malware Detection. In: Proceedings of IEEE Symposium on Security and Privacy (2005)
4. Barford, P., Yagneswaran, V.: An Inside Look at Botnets. In: Advances in Information Security. Springer, Berlin (2006)
5. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM conference on Computer and communications security (CCS'02) (2002)
6. Filiol, E.: Formalisation and implementation aspects of k-ary (malicious) codes. J. Comput. Virol. **3**(3), 75–86 (2007) (EICAR 2007 Best Academic Papers)
7. Harbour, N.: Stealth Secrets of the Malware Ninjas. https://www.blackhat.com/presentations/bh-usa-07/Harbour/Presentation/bh-usa-07-harbour.pdf.
8. Kolbitsch, C., Comparetti, P.M., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: Effective and Efficient Malware Detection at the End Host. In: Proceedings of 18th USENIX Security Symposium (2009)
9. Nomenumbra: Counter Behavior Based Malware Analysis, Hacking at Random. HAR (2009)
10. Aciiçmez, O., Koç, Ç.K., Seifert, J.: On the power of simple branch prediction analysis. In: Proceedings of the 2nd ACM Symposium on information, Computer and Communications Security (ASIACCS'07) (2007)
11. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partition graphs. Bell Syst. Tech. J. **49**, 291–307 (1970)
12. Christodorescu, M., Jha, S., Kruegel, 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 (2007)
13. Anubis. http://anubis.iseclab.org/
14. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
15. Jiang, X., Walters, A., Buchholz, F., Xu, D., Wang, Y.M., Spafford, E.H.: Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. In: Proceedings of 26th IEEE Int'l Conf. Distributed Computing Systems (ICDCS'06) (2006)
16. Fletcher, T.: Sharing a File Descriptor Between Processes. http://www.qnx.com/developers/articles/article_913_1.html
17. Yin, H., Song, D., Manuel, E., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conferences on Computer and Communication Security (2007)
18. King, S.T., Chen, P.M.: Backtracking Intrusions. In: Proceedings of the 2003 Symposium on Operating Systems Principles, pp. 223–236 (2003)
19. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based Spyware Detection. In: Proceedings of the USENIX Security Symposium (2006)
20. Cohen, F.: Computer viruses: theory and experiments. Comput. Secur. **6**(1), 22–35 (1987)
21. Phoenix. https://connect.microsoft.com/Phoenix
22. Cavallaro, L., Saxena, P., Sekar, R.: On the limits of information flow techniques for malware analysis and containment. In: Proceedings of 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (2008)
23. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley Professional, Reading (2005)

24. Forrest, S., Perelson, A.S., Allen, L., Cherukuri, R., Self-Nonself Discrimination in a Computer. In: Proceedings of IEEE Symposium on Security & Privacy (1994)
25. Stinson, E., Mitchell, J.C.: Characterizing Bots' Remote Control Behavior. In: Detection of Intrusions & Malware, and Vulnerability Assessment (2007)
26. Willems, C., Holz, T., Freiling, F.: Toward Automated Dynamic Malware Analysis Using CWSandbox. In: Proceedings of IEEE Security and Privacy (2007)
27. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. In: Proceedings of the 14th conference on USENIX Security Symposium (2005)
28. Norman Sandbox Whitepaper. http://www.norman.com
29. Srivastava, A., Lanzi, A., Giffin, J.: System Call API Obfuscation. In: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (2008)
30. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P.: Learning and Classification of Malware Behavior. In: Proceedings of Detection of Intrusions and Malware, and Vulnerability Assessment (2008)
31. Percival, C.: Cache missing for fun and profit. BSDCan (2005). http://www.daemonology.net/hyperthreading-considered-harmful/
32. Stevens, R.: UNIX Network Programming, 2nd edn. Interprocess Communications, vol. 2. Prentice Hall, Englewood Cliffs (1999)
33. Dyshlevoi, K.V., Kamensky, V.E., Solovskaya, L.B.: Marshalling In Distributed Systems: Two Approaches (1997). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.9781
34. Borello, J., Mé, L.: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**, 211–220 (2008). doi:10.1007/s11416-008-0084-2