Turkish Journal of Engineering

Turkish Journal of Engineering (TUJE)
Vol. 3, Issue 3, pp. 149-156, July 2019
ISSN 2587-1366, Turkey
DOI: 10.31127/tuje.498878
Research Article

USER-ORIENTED FILE RESTORATION FOR OPERATING SYSTEMS

Hüseyin Pehlivan *[1]

[1] Karadeniz Technical University, Engineering Faculty, Department of Computer Engineering, Trabzon, Turkey
ORCID ID 0000-0002-0672-9009
pehlivan@ktu.edu.tr

* Corresponding Author
Received: 18/12/2018 Accepted: 18/02/2019

ABSTRACT
Folders such as recycle bin are a crucial component of wide working environments like operating systems. In current operating systems, such facilities are implemented either in no user-oriented fashion or very poorly. Various intrusion detection mechanisms are developed to prevent any damage, but very few offers the repair of the user's file system as an additional level of protection. This paper presents how to build a recycle bin mechanism for Unix operating systems entirely at the user level. The mechanism involves the control of system resources in a more intelligent way. Programs thus are running under greater control, monitoring and analyzing their resource requests. The idea is based on the interception of a particular class of system calls, using tracing facilities supported by many Unix operating systems. This provides better high level information, and presents efficient techniques to prevent foreign or untrustworthy programs from doing any irreparable damage. A program called trash has been constructed and experimented to investigate potential consequences of the recycle bin mechanism. The experiments highlight possible overheads imposed on the system. The paper also performs a comparative analysis of the trash program with some related approaches and tools

Keywords: *Recycle Bin, Operating Systems, System Calls, Process Tracing, Restoration*

1. INTRODUCTION

Recycle bin mechanisms are usually provided for file recovery purposes and directly employed by users. User-oriented recovery in operating systems has become very least concern among past studies, possibly due to reasons such as the lack of necessary system facilities. The system-wide integrity protection is accomplished by third party programs that, once an anomalous state has been detected, remove the anomaly from the system, restoring the original state. The anomaly removal and restoration capabilities of these commercial programs fail to completely reverse the effects of an anomalous program (Passerini *et al*., 2009).

In programs such as editors, users are presented with recovery commands (undo, redo, etc.) to meet their preservation and restoration requirements. Given an operating system, undo and redo commands, which are rather useful for smaller environments, do not seem very functional to bring back any destructed file seamlessly, which is mainly caused by the multi-tasking nature of operating systems and possible dependencies between user commands. Two forthcoming studies are made on Windows, where untrusted programs are monitored, logging their operations, (Hsu *et al*., 2006; Paleari *et al*., 2010). Using the logs, it can completely remove malware programs and reliably restore the infected data. However, it is a recovery facility that is provided for a single user environment.

In modern operating systems, notably POSIX compliant ones, recycle bin functionality is integrated into a desktop environment and its file manager. Typical examples are Microsoft Windows with Windows Explorer and GNOME with Nautilus. In such environments, an overwritten file does not usually go to the recycle bin. In fact, the recoverability of files deleted by a program depends on its level of integration with a particular desktop environment. Low-level utilities can bypass this layer entirely and delete files immediately. In Windows or Unix operating systems, a file removed through a DOS or terminal window is not placed in the recycle bin and so is not recoverable.

The aim of the paper is to provide a user-oriented restoring facility for Unix operating systems. Unlike the Windows recycle bin mechanism, the restoring facility operates in a more intelligent way, dealing with dynamically deleted files as well. In fact, it basically deals with only file-related damage of either commands issued or applications managed by a user. The damage imposed by the system itself as a result of situations such as a crash, however, have to be handled by the administrator.

The described mechanism, called *trash* (TRAcing SHell), is implemented using C++, completely at the user level without having to modify the existing system internals. It consists of two separate subprograms which monitor user activities and restore destructed files. The monitoring program (*monitor*) is started as a daemon by the administrator, and controls all users' activities. It can handle all existing commands and programs, and any new ones which have been installed recently. The restoring program (*restorer*) runs as an ordinary program, and can be configured for each different user, with their own restoring requirements.

Like all usual recovery implementations, the *trash* mechanism needs to find out what changes each user command makes to the file system. Shells in common use never know which file operations commands pass on to the kernel to execute, since the commands make these requests with system calls. The *trash* daemon must itself monitor file requests of commands. Fortunately, many Unix operating systems provide standard tracing facilities such as the /proc file system which is used by *truss*-like applications and various intrusion detection systems. With these facilities, system calls are intercepted and then resumed after the appropriate information is extracted.

2. RELATED WORK

Approaches of system protection can be broadly divided into two categories; system restoration and intrusion detection. System restoration based approaches maintain some specific checkpoints in the system and roll back a user-selected collection of actions. Approaches of intrusion detection can be further divided into anomaly detection and misuse detection. Anomaly detection based approaches first construct a profile that describes normal behaviours and then detect deviations from this profile. In contrast, misuse detection based approaches define and look for precise sequences of events that damage the system.

System restoration is based on information stored or gathered during normal interaction with a program. Almost all programs are being currently equipped with restoring facilities. In the environments controlled by self-contained programs such as editors and painters, the provision of recovery support is relatively easy due to paucity and known effects of operations. Wide environments presented by operating systems are open to every kind of operations, and thus the effects of system programs must be handled in a more intelligent way. Therefore different environments involve differentiating types of restoration, which is basically associated with the number of users working in them. Small environments use data chunks for restoration, while wide ones do data files for restoration. Restoration in small environments is inherently user-oriented and supplied as a result of recovery commands such as undo and redo. There are many recovery models introduced for such environments where interactions with a single user (Vitter, 1984; Spenke and Beilken 2003; Brown and Patterson, 2003) or multiple users (Choudhary and Dewan, 1992; Berlage and Genau, 1993) are supported. Typical examples of those models are history undo model (Stallman, 1986), selective undo model (Prakash and Knister, 1992) and object-based undo model (Zhou and Imamiya, 1997). Restoration in wide environments is either system-oriented or user-oriented. As in Unix operating systems, system-oriented restoration is carried out via backup tapes accessible by only the system administrator. As in Windows operating systems, user-oriented restoration is done via a filestore maintained in the system.

Anomaly detection techniques address the existence of an intrusion by considering any abnormalities in user or system behaviour as a potential attack (King and Chen, 2003; Qiao *et al*., 2002; Christodorescu and Jha, 2004). In order to learn normal user or system behaviours, most techniques analyze program behaviours. Typical examples of analyzing program behaviours are the N-gram and FSA-based algorithms

(Sun *et al*., 2005; Wu *et al*., 2016; Yu *et al*., 2005), which are usually applied to server programs. Both algorithms characterize normal program behaviours in terms of sequences of system calls. A sequence of system calls that have not been observed under normal operation of programs is treated as anomalous program behaviour. The N-gram algorithm breaks a system call sequence into substrings of a fixed length N, and then stores these substrings (called N-grams) in a table. The FSA-based algorithm maintains state-related information (the program state in the point of each system call) as system calls is made by a process under normal execution, where the system calls correspond to transitions in FSA (finite-state automata). The performance of these algorithms for three popular servers (FTP, HTTP, NFS) can be found in (Sekar *et al*., 2001).

Misuse detection techniques model known attacks using patterns and detect them via pattern-matching (Abed *et al*., 2015; Anandapriya and Lakshmanan, 2015; Chen *et al*., 2016; Creech and Hu, 2014; Jose *et al*., 2018; Liu *et al*., 2018). These techniques rely on a wide variety of observable data such as system-call data, preventing intrusions from either local or remote users as a result of evaluating the legitimacy of their activities. Many solutions to detecting and preventing intrusions are based on the interception of system calls. A recent comprehensive review is conducted to assess the advantages and drawbacks of intrusion detection techniques proposed in the literature (Ramaki *et al*., 2018).

3. CHARACTERISTICS OF RESTORATION

In this work, a conventional distinction is made between state and file restorations. The restoration of a state occurs as a result of undoing or redoing user commands. The names and contents of files together determine the system state, as well as other components such as directories and access permissions. A change in file names is considered to move the system into another state. So a state recovery means that each component of a state affected of the execution of a command is reinstated. File restoration is usually different from state restoration, which is associated with the contents of files only, excluding file names or permissions.

In practice, this distinction helps provide more useful functionalities from the perspective of a user's requirement. The user would mainly expect a utility to be able to bring the files with the correct contents back. In restoration, since the contents are just important, there is less information stored to carry out file protection and restoration.

On the other hand, there are some difficulties with ensuring the applicability of recovery commands to all situations. One important difficulty is the requirement of controlling concurrency, where programs interleave. Indeed, even if concurrency is controlled desirably, it may leave some programs irrecoverable, restricting the usability of the recovery mechanism. For example, consider the following situation which is led to by concurrent execution of two programs, `P1` and P2, entered in separate command-lines:

```
10> Delete fileA 5 P1
11> Create fileA 11 P1
12> Read fileA 11 P2
13> Create fileB 13 P2
14> Read fileB 13 P1
```

This does not enable both programs to be undone separately. Thus it is impossible to get back the old version of `fileA` (version 5). There is no elegant way to cope with this situation, because the user is currently allowed to specify only one command to undo from the history list at a time. This involves considering each atomic file operation individually.

There is a close relationship between file protection and safe command execution. The best way of protecting a file system is to ensure that every system command executes securely. Provision of secure execution of commands seems to require restricting the environment. There is a lot of work associated with file protection, which have concentrated on safe command execution, as given in Section 2. Many of previous works provide users with restricted environments for safe execution of programs. Any destroyed file cannot be retrieved. However, the *trash* mechanism aims not to restrict the working environment.

4. DESIGN AND IMPLEMENTATION

The *trash* program is designed in two components (subprograms), namely *monitor* and *restorer*, which run as separate processes in tracing and restoration modes respectively. The *monitor* component both controls one user's activities as a result of tracing user-serving programs and stores information required for restoration in a directory named *trash* under the user's home directory. The kind of restoration information stored depends entirely on intercepted system calls, whose effects on the system may vary considerably, and so each system call is handled individually. The *restorer* component handles the restoration requirements, bringing old versions of files back.

The owner of these components is the system administrator, and thus users are not allowed to directly write to the *trash* directory. This restriction is required to protect restoration information against other programs executed by users, intentionally or not. Besides, for more efficient protection, a system group named *restore* is created, which consists of the name of users to employ the *trash* mechanism, and the group ownership of *restorer* is changed to that. In this way restoration is made possible only through the *restorer* program.

In order to control all activities of a particular user, it is not adequate to trace only login shells for recovery purposes. Unix operating systems provide various levels of remote access. For each level, there are many tools which allow users to manage their accounts remotely, as described in Table 1.

Table 1 also gives the names of daemon processes serving the specified tools, which are system-dependent. For example, X servers which allow running Unix desktop environments such as KDE and GNOME, interact with display managers (daemons) such as kdm and gdm, respectively. The file-related effects made through all such tools need monitoring properly.

Table 1. Unix remote access tools and daemons

Tools	Instances	Daemons
Telnet/SSH client programs	AxeSSH, FiSSH, PuTTy	telnetd, sshd
FTP/SFTP programs	AceFTP, SmartFTP, WinSCP3	ftpd, sftpd
X Window System servers	X-Win32, eXceed, WeirdX	kdm, gdm
Web servers	Apache, Java Web Server	httpd, webservd

Unix environments are full of many utilities used for text formatting and program developing purposes that produce numerous temporary or permanent files, which we call "generated files". To hold these specific files, they can use some subdirectories under the system's root directory (e.g., /tmp and /var) or the user's home directory (e.g., ~/.netscape and ~/.ssh). In the current implementation, all files only in home directories are protected by default, except some language and program specific files with the extensions ".aux", ".log", and ".o". However, each user can individually change the default configuration via the menu provided by *restorer* program.

The *trash* program stores restoration information in two files named *.trash* and *.conf* under ~/.trash. The first keeps the record of file deletions, while the latter contains the names of files and directories that are not to be protected by *trash*. File names can be specified explicitly or using some extensions. Other entries in ~/.trash are copies of files stored as a whole with a number extension at the end which is incremented sequentially.

Typical scenario of storing information might be as follows. Whenever *trash* intercepts the *open* system call, it checks the first argument to see whether it points to a pathname under the user's home directory. If the file is owned by other users or belongs to a standard shared library, for example, the *open* is allowed to go ahead, in which case nothing is stored. Otherwise, the second argument is checked to see which operation *open* is to perform on the file. In the case of a *deletion* operation, for example, a complete copy of the file is taken. (For *open*, a *deletion* means modification of the file as a result of flags such as `O_WRONLY`, `O_RDWR` and `O_TRUNC`.) Then the system call is released to resume as normal. The other system calls, such as *creat*, *rename*, *unlink*, *symlink* and *link*, can also perform deletion operations.

The text-based versions of the *monitor* and *restorer* programs are currently implemented. During tracing, *monitor* uses each user's own configuration to gather and store information. There is no graceful way to individually monitor user tasks via separate processes due to the fact that a single process can serve more than one user. Therefore, only one instance of it is running in the system, without spawning one monitoring child process for each user. On the other hand, each user executes her/his own instance of the *restorer* program from the command line which accesses the .trash directory to obtain restoring information.

All these preparations make the restoration of files quite simple. The effect of the *restorer* program consists only of the movement or exchange of file versions between the current state and the *.trash* directory of previous versions. For the restoration of a particular file that has been previously deleted by a user task, the *restorer* program firstly checks to see if an overwrite occurs. If there is no overwrite operation, it brings the deleted file back to the current state and removes the related line of the *.trash* file. Otherwise it warns the user.

5. CONCURRENCY CONTROL

An important issue to handle file restoration is concurrency. Subject to their use of system calls, processes execute independently of each other and share system resources arbitrarily, which can cause non-determinism. In order to reduce the usual non-determinism problems caused by uncontrolled sharing of resources to a minimum, *trash* has to handle concurrency in a more sophisticated way than the operating system or ordinary shells. Fortunately, the /proc interface provides a means of monitoring all user processes and their descendants via only one tracing process. The *monitor* program behaves like a tracing process, using the *poll* call to listen to process events. To efficiently monitor running processes, a process table named `procT` is defined by the following C++ structure:

```
int nprocs;
int current_time;
struct pollfd Pollfds[MAXPROCS];
struct processTable {
    int         time;
    pid_t       pid;
    int         procfd;
    prstatus_t *pstatus;
} procT[MAXPROCS];
```

where the structure Pollfds are used by *poll* to perform efficient control of process events concurrently. Each new process, which is represented by *pid*, is inserted into the table with a unique *time* entry, using the value *current_time*. Note that parent-child relations do not require maintenance in the table. All processes are supposed to be at the same level. To detect and record their concurrent file accesses, a second table `fileT`) is maintained as follows:

```
static int nfiles;
struct fileTable {
    int nprocfds;
    int procfd[MAXPROCS];
    char *path;
} fileT[MAXFILES];
```

where the attempt of opening a file adds a new entry to the table. All processes that hold a single file open are kept together within the same entry.

The time during which a file remains open is usually determined by *open*-like calls and *close*. However, on seeing a *close* call, We cannot simply assume that the process is finished with the related file. One reason is the duplications of file descriptors. In this way, a process can have some files open even after *close*, as a result of performing system calls such as *dup* and *fcntl*.

The monitoring of system calls that copy existing

descriptors is not enough to detect all files a process holds open. There is another situation where files might remain open. It is associated with inheritance structures of processes. When the `FD_CLOEXEC` flag of a file descriptor created by a parent process is clear, the file remains open across *fork* and *exec* calls, which means that its child processes can inherit those file descriptors. This complicates the issue of keeping the track of file descriptors for each process.

In fact it is not easy and practical to monitor processes for file descriptors information, because it imposes numerous dynamic checks on run-time environments of processes. We do not deal with file descriptors much. For simplicity, it is assumed that a process creating or inheriting a file descriptor holds the related file open until it terminates.

File open activities of processes provide the most valuable information for *trash*. Each opening operation does not correspond to a backup. For the first process that opens a file for writing for the first time, the file is copied to ~/.trash. As far as the file remains open, all later accesses to it by different processes would have no effect on the restoring mechanism (that is, no new copies are made). Only after all the related processes end, a new process that tries to open the same file for writing would cause a backup.

6. OVERHEADS AND PERFORMANCE

In order to measure the whole overheads caused by the *trash* program, the overhead is examined in terms of runtime and space usage. Runtime overheads occur with both system call interception and execution of detection/storing code. Space overheads are caused by files stored in ~/.trash.

Table 2. CPU time and disk space overheads of the *trash* program

Program	Not Monitored	Monitored	Overhead
cp	35ms	37ms	5%
latex	160ms	182ms	13%
xterm	390ms	439ms	12%
emacs	483ms	528ms	9%
mozilla	2476ms	2552ms	3%
eclipse	9342ms	9986ms	7%
smc	12350ms	13015ms	5%

With root privileges we monitor three system daemons (e.g. sshd, sftp-server and dtlogin), actually grandchild processes through them. The webservd daemon could not be monitored satisfactorily, due to the absence of the web content that interacts with the file system intensively. Table 2 shows the results for certain programs which operate on Sun Solaris Sparc machine with 2 processors with 1.28GHz each, 4GB of RAM and 4 Ultra160 SCSI hard disks with 73GB each. The machine is daily connected by about 34 users remotely through the tools given in Table 1. To get around the effects of network environment, the connection times are not measured. The CPU time is only given for some system programs. For the long-running programs given in Table 2, the time measurement is restricted by the point the user input is asked (*smc* is the Solaris management console program).

Compared to techniques for interception of system calls within the kernel, user-level mechanisms tend to incur significantly higher overheads. This is unavoidable because, for each system call, there are additional context switches between a process that is handling a particular user task and another process that is intercepting its system calls. The number of context switches depends on how many system calls of interest a program issues. Given the file system operations, each user program usually need to perform a small number of system calls of interest to *trash*, causing a few context switches. For instance, the Unix command "rm *" performs one *unlink* call for each file in the current directory. The fact that file system activities of most Unix commands are very few introduces trivial overheads in the short terms.

Furthermore, to find out the name and parameters of a system call intercepted, the monitoring process is required to access the monitored process memory, incurring the overhead of system call interception. After detecting and decoding a system call, possible storing operations are executed, incurring the overhead of execution of detection/storing code.

Disk space overheads are also explored on the machine with the specifications described above, for the period of one month. For a particular user, the size of disk space is increased with 19% of currently occupied one. For another user, the increase is only 12%. These overheads are imposed by file storing operations, which may lead to more space overheads. As a solution to keeping space usage down, all stored files might be compressed, but this possibly leads to an increase in storing time, which is beyond the subject of this paper.

On the other hand, compared to traditional recovery mechanisms, the restoration technique needs less amount of space in the command-independent fashion. To illustrate, consider the following situation, assuming that `fileB` and `fileC` does not exist in the current state:

```
P1: rename(fileA, fileB)
P2: rename(fileB, fileC)
```

where the *rename* calls are executed through two separate user programs. For purposes of recovery, fileA and `fileB` have to be saved so that they can be separately reinstated later. For restoration, there is no need to save any file, because the contents of `fileA` remain unchanged, even if it is eventually renamed as `fileC`.

We also explored that some programs (e.g. an editor or painter) can overwrite a file many times during execution. In this case, the program carries out many deletions of possibly the same file, causing intermediate copies of the file to be stored. Most of these deletions have no effect on restore operations. So it is unnecessary for *trash* to keep track of them all. Restoration information needs only contain at most one copy for a single file managed by a program, which corresponds to the contents of the file at the time when the program is called.

7. COMPARATIVE ANALYSIS

Most literature studies that analyze the trace of

system calls focus on intrusion detection systems. Although system call data is an instrumental artefact of the kernel, it can be modelled to support decision making activities of these systems at program level. Thus, the anomaly detectors use various modelling techniques for assessing the behaviour of processes via the sequence of system calls and their arguments. The techniques are typically based on sequential features (SF), frequency-based features (FF), argument-based features (AF) and hidden-markov models (HM). Table 3 compares our study with some previous work in terms of the modelling techniques and the type of intrusion detection, which can be supervised or not.

Table 3. Comparison of system call-based modeling techniques

Reference	Detec.	Tech.	Supr.
Anandapriya *et al*., 2015	Anomaly	SF	Yes
Creech *et al*., 2014	Anomaly	SF	Yes
Gupta *et al*., 2015	Misuse	SF	No
Xie *et al*., 2014	Anomaly	FF	No
Haider *et al*., 2015	Anomaly	FF	No
Hoang *et al*., 2009	Hybrid	HM	Yes
Hu *et al*., 2009	Anomaly	HM	No
Sekar, 2001	Anomaly	AF	No
Mutz, 2006	Anomaly	AF	No
Our approach	Hybrid	AF	Yes

Another comparison is conducted based on different performance criteria such as scalability (Scal.), space complexity (Space), time complexity (Time) and detection robustness (Rbst.). Table 4 shows the analysis results for various detection systems.

Table 4. Comparison of various detection systems based on some performance criteria

Reference	Scal.	Space	Time	Rbst.
Fuse *et al*., 2017	Low	High	Low	Low
Yolacan *et al*., 2014	High	High	High	High
Hu *et al*., 2009	High	Low	Low	Low
Zhou *et al*., 2008	High	High	High	High
Zhang *et al*., 2006	High	Low	High	High
Hoang *et al*., 2009	Low	Low	High	High
Our approach	High	Low	Low	High

The final comparison of our approach is made with three other restoration approaches and three commercial malware detectors including Nod32 Anti-Virus, Panda Anti-Virus, and Kaspersky Anti-Virus, which is evaluated in (Passerini, 2009). The restoration operations are performed on three system resources of files, registry keys and/or processes. The performance of the approaches and tools is classified as good, average and poor categories. The results are shown in Table 5.

Table 5. Comparison of some restoration approaches and tools

Approach/Tool	OS	File	Reg.	Proc.
Nod32	Windows	Good	Good	Poor
Panda	Windows	Avg.	Avg.	Avg.
Kaspersky	Windows	Avg.	Good	Poor
Hsu *et al*., 2006	Windows	Good	Good	NA
Paleari *et al*., 2010	Windows	Good	Good	Good
Webster *et al*., 2018	Linux	Good	NA	NA
Our approach	Unix	Good	NA	NA

8. CONCLUSION AND FUTURE WORK

We have described a recycle bin mechanism on Unix for repairing the file system damages in a user-oriented fashion. The mechanism deals with all situations which threaten the integrity and security of the file system, giving a chance the user to restore unintentional deletions of files. It achieves these goals by monitoring individual user activities, storing destructed files, and using the restoration information to eliminate their effects. We examined its overheads for possible file-related operations of programs, concluding that the CPU and storage overhead caused by the utility is acceptable. This conclusion is also supported by the results of the comparative analysis made with some other approaches and tools.

The main subject of future work is on maintaining the recycle bin efficiently. To achieve this goal, one issue is to determine how long file copies are preserved in the user's disk area. Some users can require file preservation to be in effect during multiple login sessions. To avoid long-term file preservations, an optimum duration must be determined for most utilization of the facility. This raises another issue, which is the disk space usage of stored files. Using the disk quota allowed for each user as an additional parameter, we are currently working on the recycle bin to make it occupy less disk space and to keep it covering some particular sessions.

REFERENCES

Abed, A. S., Clancy, C. and Levy, D. S. (2015). "Intrusion Detection System for Applications Using Linux Containers", *International Workshop on Security and Trust Management*, pp. 123-135.

Anandapriya, M. and Lakshmanan, B. (2015). "Anomaly Based Host Intrusion Detection System using semantic based system call patterns", Proc., *9th International Conference on Intelligent Systems and Control (ISCO)*, pp. 1-4.

Berlage, T. and Genau, A. (1993). "From undo to multi-user applications", *Proc., Vienna Conference on Human–Computer Interaction*, Vienna, Austria, Sept 20-22, pp. 213-224.

Brown, A. B. and Patterson, D. A. (2003). "Undo for operators: Building an undoable e-mail store" *Proc., 2003 USENIX Annual Technical Conference*, pp. 1-14.

Chen, C. M, Guan, D. J. and Huang, Y. Z and Ou, Y. H. (2016). "Anomaly network intrusion detection using Hidden Markov Model", *International Journal of Innovative Computing, Information and Control*, Vol. 12, No. 2, pp. 569-580.

Choudhary, R. and Dewan, P. (1992). "Multi-user undo/redo", *Technical Report TR125P*, Computer Science Department, Purdue University.

Christodorescu, M. and Jha, S. (2004). "Testing malware detectors", *Proc., 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, New York, NY, USA, pp. 34-44.

Creech, G. and Hu, J. (2014). "A Semantic Approach to Host-Based Intrusion Detection Systems Using Contiguous and Discontiguous System Call Patterns", *IEEE Transactions on Computers*, Vol. 63, pp. 807-819.

Fuse, T. and Kamiya, K. (2017). "Statistical anomaly detection in human dynamics monitoring using a hierarchical dirichlet process Hidden Markov Model", *IEEE Transactions on Intelligent Transportation Systems*, Vol. 18, No: 11, pp. 3083–3092.

Gupta, S. and Kumar. P. (2015). "An immediate system call sequence based approach for detecting malicious program executions in cloud environment", *Wireless Personal Communications*, Vol. 81, No. 1, pp. 405–425.

Haider, W., Hu, J. and Xie. M. (2015). "Towards reliable data feature retrieval and decision engine in hostbased anomaly detection systems", *IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*, Auckland, New Zealand, pp. 513–517.

Hoang, X. D., Hu, J. and Bertok. P. (2009). "A program-based anomaly intrusion detection scheme using multiple detection engines and fuzzy inference", *Journal of Network and Computer Applications*, Vol. 32, No. 6, pp.1219–1228.

Hsu, F., Chen, H., Ristenpart, T., Li, J., and Su, Z. (2006). "Back to the future: A framework for automatic malware removal and system repair", *Proc., 22nd Annual Computer Security Applications Conference*, ACSAC '06, IEEE Computer Society, Washington, DC, pp. 257–268.

Hu, J., Yu, X., Qiu, D. and Chen, H.-H. (2009). "A simple and efficient hidden markov model scheme for host-based anomaly intrusion detection", *IEEE network*, Vol. 23, No. 1, pp. 42–47.

Jose, S., Malathi, D. Reddy, B. Jayaseeli, D. (2018). "Anomaly Based Host Intrusion Detection System Using Analysis of System Calls", *International Journal of Pure and Applied Mathematics*, Vol. 118, No. 22, pp. 225-232.

King, S. and Chen, P. M. (2003). "Backtracking intrusions", *Proc., 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 223-236.

Liu, M., Xue, Z. Xu, X., Zhong, C. and Chen. J. (2018). "Host-Based Intrusion Detection System with System Calls: Review and Future Trends", *ACM Computing Surveys*, Vol. 51, No. 5, pp 1-36.

Mutz, D., Valeur, F., Vigna, G. and Kruegel, C. (2006). "Anomalous system call detection". *ACM Transactions on Information and System Security (TISSEC)*, Vol. 9, No. 1, pp. 61–93.

Paleari, R., Martignoni, L., Passerini, E., Davidson, D., Fredrikson, M., Giffin, J., and Jha, S. (2010). "Automatic generation of remediation procedures for malware infections", *Proc., 19th USENIX Security Symposium*, August 11-13, Washington, DC, pp. 419-434.

Passerini, E., Paleari, R. and Martignoni, L. (2009). "How Good Are Malware Detectors at Remediating Infected Systems?", *Proc., 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, July 09-10, Como, Italy, pp. 21-37.

Prakash, A. and Knister, M. J. (1992). "A framework for undoing actions in collaborative systems", *Technical Report CSE-TR-125-92*, Computer Science and Engineering Division, The University of Michigan, Ann Arbor.

Qiao, Y., Xin, X. W., Bin, Y. and Ge, S. (2002) "Anomaly intrusion detection method based on HMM", *Electronics Letters*, Vol. 38, No. 13, 2002, pp. 663-664

Ramaki, A. A., Rasoolzadegan, A. and Jafari, A. J. (2018) "A Systematic Review on Intrusion Detection based on the Hidden Markov Models", *Statistical Analysis and Data Mining*, Vol. 11, No. 3, pp. 111-134

Sekar, R., Bendre, M., Dhurjati, D. And Bollineni, P. (2001). "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors", Proc., *IEEE Symposium on Security and Privacy*, pp. 144-155.

Spenke, M. and Beilken, C. (1991). "An overview of GINA—the generic interactive application", *Proc., User Interface Management and Design*, D. A. Duce *et al.*, Eds. Springer-Verlag, New York, NY, USA, pp. 273-293.

Stallman, R. (1986). "GNU Emacs manual, Version 17", *Free software foundation. Inc*.

Sun, W., Liang, Z., Venkatakrishnan, V. N. and Sekar,R. (2005). "One-way isolation: An effective approach for realizing safe execution environments", *Proc., Network and Distributed Systems Symposium (NDSS)*, pp. 265-278.

Webster, A., Eckenrod, R. and Purtilo, J. (2018). "Fast and Service-preserving Recovery from Malware Infections Using CRIU", *Proc., 27th USENIX Security Symposium*, August 15-17, Baltimore, MD, USA, pp. 1199-1211.

Vitter, J. Z. (1984). "US&R: A new framework for redoing", *IEEE Software*, Vol. 1, No. 4, pp. 39-52.

Wu, F., Wu, D. and Yang, Y. (2016). "A Network Intrusion Detection Algorithm Based on FSA Model", *4th International Conference on Machinery, Materials and Computing Technology*, pp. 615-621.

Xie, M., Hu, J., Yu, X. and Chang. E. (2014). "Evaluating host-based anomaly detection systems: Application of the frequency-based algorithms to adfa-ld", *International Conference on Network and System Security*, Berlin, Heidelberg, Springer, pp. 542–549.

Yolacan, E. N., Dy, J. G. and Kaeli, D. R. (2014). "System call anomaly detection using multi-HMMs, Software Security and Reliability-Companion (SERE-C)", *IEEE 8th International Conference on Software Security and Reliability-Companion*, San Francisco, CA, pp. 25–30.

Yu, F., Xu, C., Shen, Y., An, J., and Zhang, L. (2005). "Intrusion detection based on system call finite-state automation machine". *IEEE International Conference on Industrial Technology*, pp. 63-68.

Zhang, J., Liu, Y. and Liu, X. (2006). "Anomalous detection based on adaboost-HMM", *IEEE 6th World Congress on Intelligent Control and Automation (WCICA)*, pp. 4360–4363.

Zhou, C. and Imamiya, A. (1997). "Object-based nonlinear undo model", *Proc., 21th International Computer Software and Applications Conference*, COMPSAC, pp. 50-55.

Zhou, X., Peng, Q. K. and Wang, J. B. (2008). "Intrusion detection method based on two-layer HMM", *Application Research of Computers*, Vol. 3, No. 1, 75.