Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 140368, 12 pages
doi:10.1155/2012/140368

Research Article

A Stateful Approach to Generate Synthetic Events from Kernel Traces

Naser Ezzati-Jivan and Michel R. Dagenais

Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, C.P. 6079, Station Downtown, Montreal, Quebec, Canada H3C 3A7

Correspondence should be addressed to Naser Ezzati-Jivan, ezzati@gmail.com

Received 15 December 2011; Accepted 13 April 2012

Academic Editor: Antonella Di Stefano

We propose a generic synthetic event generator from kernel trace events. The proposed method makes use of patterns of system states and environment-independent semantic events rather than platform-specific raw events. This method can be applied to different kernel and user level trace formats. We use a state model to store intermediate states and events. This stateful method supports partial trace abstraction and enables users to seek and navigate through the trace events and to abstract out the desired part. Since it uses the current and previous values of the system states and has more knowledge of the underlying system execution, it can generate a wide range of synthetic events. One of the obvious applications of this method is the identification of system faults and problems that will appear later in this paper. We will discuss the architecture of the method, its implementation, and the performance results.

1. Introduction

Tracing complete systems provides information on several system levels. The use of execution traces as a method to analyze system behavior is increasing among system administrators and analysts. By examining the trace events, experts can detect the system problems and misbehaviors caused by program errors, application misconfigurations, and also attackers. Linux trace toolkit next generation (LTTng), a low-impact and precise Linux tracing tool, provides a detailed execution trace of system calls, operating system operations, and user space applications [1]. The resulting trace files can be used to analyze the traced system at kernel and user space levels. However, these trace files can grow to a large number of events very quickly and make analysis difficult. Moreover, this data contains too many low-level system calls that often complicate the reading and comprehension. Thus, the need arises to somehow reduce the size of huge trace files. In addition, it is better to have relatively abstract and high-level events that are more readable than raw events and at the same time reflect the similar system behavior. Trace abstraction technique reduces the size of original trace by grouping the events and generating high-level compound synthetic events. Since synthetic events reveal more high-level information of the underling system execution, they can be used to easily analyze and discuss the system at higher levels.

To generate such synthetic events, it is required to develop efficient tools and methods to read trace events, detect similar sections and behaviors, and convert them to meaningful coarse-grained events. Most of the trace abstraction tools are based on pattern-matching techniques in which patterns of events are used to detect and group similar events or sequences of related events into compound events. For instance, Fadel [2] uses pattern-matching technique to abstract out the traces gathered by LTTng kernel tracer [1]. They have also created a pattern library that contains patterns of Linux file, socket, and process operations. Waly and Ktari [3] use the same technique to find system faults and anomalies. They have also designed a language for defining fault patterns and attack scenarios.

Although defining patterns over trace events are a useful mechanism for abstracting the trace events and finding the system faults, there are other types of faults and synthetic events that are difficult to find with these techniques and need more information of the system resources. In this way, modeling the state values of a system and using them may help much in finding those complex kinds of system problems. Indeed, without a proper state model, many patterns will simply attempt to recreate and recompute some of that repetitive information in a time- and performance-consuming manner.

This paper mainly describes the architecture of a stateful trace abstractor. Using a state database to store system state values enables us to have more information about the system at any given point. Indeed, after reading the trace files and making the state database, we can jump to a specific point and abstract out the trace at that point. For example, suppose we see there is a high load or a system problem at a certain time. In this case, we can load the system states at that point, reread, abstract out, and analyze only the desired part to discover the main reason of the given problem.

The main goal of this paper is to explain how to use the system state information to generate synthetic events as well as to detect complex system faults and anomalies. In this paper, we first explain converting the raw trace events to semantic events. Secondly, using a predefined mapping table, we describe extracting system metric values from trace and creating a database of the system state values. Finally, we investigate using pattern matching technique over semantic events and system state values to generate synthetic events as well as to detect system faults and misbehaviors.

The next section discusses related work. It is followed by a section explaining the proposed techniques and also the architecture of the model. Subsequently, we discuss our method in detail and provide some illustrative examples to show how it can be adopted to generate a wide range of synthetic events. Finally, our paper concludes by identifying the main features of the proposed method and possibilities for future research.

2. Related Work

The related work can be divided into two main categories: trace abstraction techniques and their usage in intrusion detection systems (IDSs). Trace abstraction combines groups of similar raw events into compound events and by means of eliminating the detailed and unwanted events reduces the complexity of the trace [4]. Furthermore, abstraction provides a high-level model of a program and makes understanding and debugging the source code easier [5]. Several studies have been conducted on analyzing, visualizing, and abstracting out large trace files [2, 3, 6]. Trace visualization is another way to show abstractions of trace events [7]. It uses visual and graphical elements to reveal the trace events. Through their work in [8], Hamou-Lhadj and Lethbridge carry out a survey on the several trace exploration and visualization tools and techniques. Some important tools that make use of these techniques are Jinsight [9], which is an Eclipse-based tool for visually exploring a program's runtime behavior; Program Explorer [10], a trace visualization and abstraction tool that has several views and provides facilities like merging, pruning, and slicing of the trace files; ISV [11], which uses automatic pattern detection to summarize similar trace patterns and thereupon reduces the trace size. Other tools include AVID [12], Jive [13], and Shimba [14].

Besides visualization, pattern matching technique has been widely used for trace abstraction [6, 15]. Most of the aforementioned tools use this technique to detect repeated contiguous parts of trace events and to generate abstract and compound events [8]. Fadel [2], Waly and Ktari [3], and Matni and Dagenais [6] use pattern matching technique to generate abstract events from the LTTng kernel trace events. Pattern matching can also be used in intrusion detection systems [16]. For example, STATL [15] models misuse signatures in the form of state machines, while in [17], signatures are expressed as colored petri nets (CPNs), and in MuSigs [18], directed acyclic graphs (DCA) are used to extract security specifications. Beaucamps et al. [19] present an approach for malware detection using the abstraction of program traces. They detect malwares by comparing abstract events to reference malicious behaviors. Lin et al. [18] and Uppuluri [16] use the same technique to detect system problems and misuses.

Almost all of these pattern-matching techniques have defined their patterns over trace events and did not consider using the system state information. Our work is different as, unlike many of those previous techniques, it considers the system states information and provides a generic abstraction framework. Our proposed method converts raw events to platform-dependent semantic events, extracts the system state value, and sends them as inputs to the pattern-matching algorithms.

3. Overview

First, here are some terms that will be referred to throughout this text:

(i) "raw event" is used to identify the event that is directly generated by the operating system tracer. Raw events show various steps of the operating system running, such as a system call entry/exit, an interrupt, and disk block read;

(ii) "semantic events" to show the events resulting from conversion of platform-specific raw events to environment-independent events. As will be discussed later, there is a mapping table between raw events and environment-independent semantic events;

(iii) Also the term "synthetic events" is used to identify events that are the result of trace abstraction and fault identification analysis modules and depict high-level behavior of the system execution. In other words, synthetic events are generated from raw and semantic events to explain the system behavior at various higher levels. "sequential file read," "attempt to write to a closed file," "DNS request," and "half-opened TCP connection" are examples of synthetic events. Figure 1 shows the relations of these three event types.

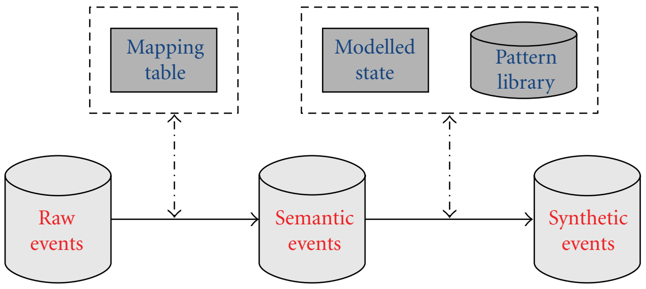


Figure 1: Raw, semantic, and synthetic events and conversion between them.

Table 1: Raw events to semantic events.

Raw events	Semantic events
sys_open event sys_dup event sys_create event	File open
sys_read event sys_pread64 sys_readv event	File read
sys_write event sys_pwrite64 event sys_writev event	File write
sys_kill event sys_tkill event sys_tgkill event	Process kill

In order to support different versions of trace formats, we propose a generic synthetic event generator that uses a set of semantic events rather than versioned raw events. Indeed, having different versions of kernel tracers as well as an evolutionary Linux kernel that leads to different trace formats makes it difficult to have a stable version of an analyzer module. It means that, for each new release of kernel or the tracer and also for any change in the trace events format, the abstraction module will have to be updated. However, by designing a generic tool, independent of kernel versions and trace formats, we can achieve a generic abstraction module that will not be dependent on certain kernel or tracer version. Semantic events help in this way: we define a semantic "open file" event instead of events of both the sys_open and sys_dup Linux system calls. In the Linux kernel, there is often no unique way to implement user-level operations. Thus, by grouping the events of similar and overlapping functionalities, we reach a new set of semantic events that will be used in the synthetic event generator. Examples of such semantic events have been outlined in Table 1.

Later, for the trace formats, one must simply update the mapping table for converting the raw events to semantic events. With this table, the synthetic event generator will work without the need to be updated for the new trace formats, being independent of the specific format and version.

Another technique to make the synthetic event generator more generic and powerful is to use the system state values. In this work, we use system state values besides the trace events for extracting the high-level information. As discussed in the related work, most of the abstraction techniques use patterns over raw and high-level events to generate abstract events. However, generating some complex types of abstract events can be very time consuming and can affect system performance. It is also difficult to generate some complex types of synthetic events that deal with several system resources or under different user identities and at different levels. Thus, using patterns of trace events is somehow not enough, and there is a need to extract and use more system information. To do so, we model state values of important system resources in a database named "modeled state." This database contains current and historic state values of the system resources, keeping track of information about running processes, the execution status of a process (running, blocked, and waiting), file descriptors, disks, memory, locks, and other system metrics. The modeled state can then be used to show users the current system states. For example, each scheduling event which sets the current running process in the modeled state will be readily available for the upcoming events. Similarly, file open events can associate filenames to file descriptors in the modeled state. These values can then be used to retrieve the filename of the given file descriptor in the context of the upcoming file operation events.

4. Architecture

In this section, we explain the architecture of the proposed solution which is shown in Figure 2. It consists of various modules: event mapper, modeled state, and synthetic event generator. In the following, we will explain how each module works.

4.1. Mapper. In the architecture, the event mapper is used to convert the trace raw events to environment-independent semantic events and also to extract the state values. The mapper actually has two steps: converting the raw events to semantic events and converting the semantic events to state changes. For each step, it uses a different mapping table. The first table is used for converting the raw events to the semantic events, and the second one has been used to convert the semantic events to corresponding state changes. It makes use of two mapping tables that contain list of conversion entries for each event type. There is not necessarily a one-to-one relationship between the raw events and corresponding outputs. A raw event or a group of raw events can be converted into one or more semantic events. For example, a Linux_sched_schedule(p1, p2) event may be mapped into two semantic events: process_stop(p1) and process_running(p2). Table 2 lists examples of these mapping entries. The mapper in Figure 2 includes both the mapping Tables 1 and 2. In other words, when events are processed in the mapper, corresponding semantic events are generated, and changes in the modeled state also occur. Most

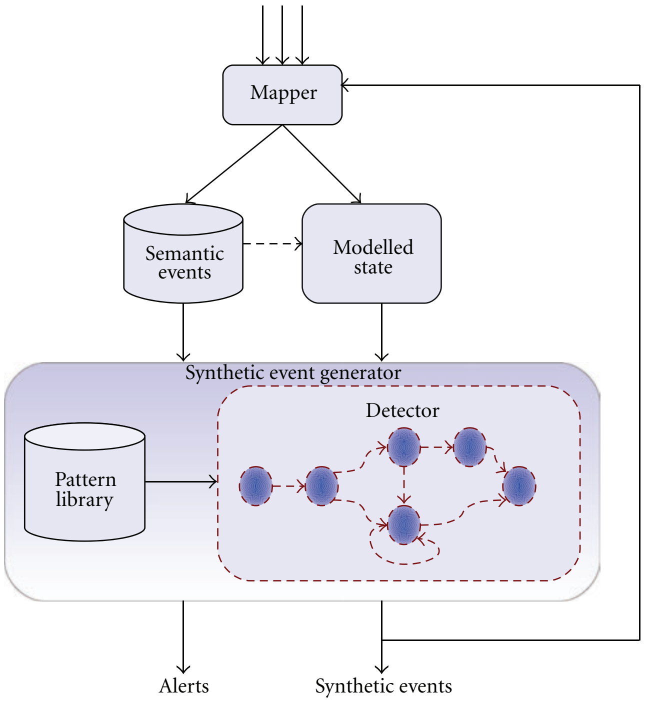


FIGURE 2: Architectural view of the stateful synthetic event generator.

TABLE 2: Semantic events to state changes.

Semantic events	Corresponding state change
File open (fd)	Changes the state of the input fd to opened
File read (fd, count)	Changes the state of the input fd to read
File close (fd)	Changes the state of the input fd to closed
Kill process (p1)	Changes the state of the input p1 to killed

of the changes take place by directly changing a state value. However, some events may have more complex effects on the modeled state. In this case, a set of changes may be queued to be performed on the state values.

4.2. Modeled State. The modeled state stores the system metrics and different state values of them. The system metrics (e.g., process name, file name, etc.) are stored in a tree-based data structure called "attribute tree" [20] or "metric tree." In this tree, each metric has an address starting from its machine name. The content of the attribute tree is similar to the tree shown in Figure 3, where names starting with $ identify the variables and can have many values. For example, $pid can be process1, process2, ...

The tree shown in Figure 3 acts like a lookup table in which various system resources and attributes are defined in a file system like path. Besides that, there is another tree based interval database to store the different values of those resources and attributes during the system execution. We store all of the extracted state values in this database that enables us to retrieve the state values at any later given time. Montplaisi and Dagenais [20] proposed a Java implementation of the modeled state that is used in this project to store and retrieve the state values.

Different types of data may be stored in the modeled state: the state of a process, the current running process on a CPU, state of a file descriptor, and so on. System resources statistics are another type of information that can be stored in the modeled state. Examples of these statistics include number of bytes read and written for a specific file, for a process or for the whole trace, total CPU usage per process or per trace or for a specific time period, the CPU waiting time, number of TCP connections in the last 2 seconds, the duration for which a disk was busy, and large data transfers. Statistics extracted from trace events are similar to the extracted state values from events. By processing trace events, one can interpret the event contents, gather relevant statistics, and aggregate them by machine, user, process, or CPU for the whole trace or for a time duration. For that, we identify the event types and event arguments used to count, aggregate, and compute the statistics of the system metrics. In other words, when one defines a mapping between events and states, he or she may also want to relate associated trace events to the statistics values.

The statistics can then be used to generate synthetic events to detect system faults and problems. In this project, we use a threshold detection mechanism [15, 21] to detect system problems. In this approach, the occurrences of specific events and statistics values of important system metrics are stored, updated, and compared to predefined threshold values. If the values cross the thresholds, or in the case of a quick rise in a short period, an alarm is raised or a log record is generated [15]. With this approach, some hook methods may be registered and invoked in the case of unusual growth or the reaching of certain threshold. For some predefined semantic events, we store and update the statistics values of the important metrics (e.g., quantity of I/O throughput, number of forks, number of half-opened TCP connections, CPU usage, number of file deletions, etc.) in the modeled state.

It is important to mention that system state is a broad term, and it could be too resource and performance consuming to care of all system resources. For trace abstraction, we only need to store a subset of that information required to represent the synthetic event patterns and associated initial, intermediate, and final states. In other words, the amount of data stored in the modeled state will depend on the patterns and will be extracted from them. Therefore, by having created a pattern library in advance or by importing them during analysis, it would be possible to determine the required set of system attributes and metrics that should be kept track of, in the modeled state.

4.3. Synthetic Event Generator. Synthetic event generator is another module that is used to generate high-level events from trace events. The synthetic event generator may use either the semantic events, modeled state values, or both to generate the synthetic events. It makes use of a pattern library that contains various patterns for reducing the trace size and also for detecting the system faults and attacks.

```
$machine/processes/$pid/state
                          execmode_stack
                          exec_name
                          fds
                            $fd1
                                filename
                                type
                            $fd2
                            . . .
          /cpus/$cpuid/current_process
                         pid
          /disks
          /memory
```

Figure 3: Typical organization of the modeled state elements.

In this paper, we use finite state machine to define the patterns. In this way, we have created a set of state machines based on the semantic events and state values to abstract out the Linux kernel execution events. In addition, it contains patterns to detect Syn flood attack, fork bomb attack, and port scanning. Each state machine represents a set of states and a sequence of actions and transitions. A transition is triggered by reading an associated semantic event or a state value change. Reaching a particular state targets a synthetic event that can reveal either a high-level system behavior or a system fault or a misbehavior. In this model, we use the modeled state to store both the states of system resources as well as the state machines' intermediate states. Based on this idea, common methods are used for storing, retrieving, and exploring the states.

As an example, suppose that we want to list all "write to a closed file" synthetic events. In this case, by comparing the filename of each write event to the open files list kept in the modeled state, we can generate and list all of these synthetic events. Here, we actually have a pattern of "write to file" semantic events and a specific path in the modeled state (open files list). Another example of such a pattern is when we want to detect all sequential file reading operations. In this case, for each read event, we keep track of the last read position as well as whether all previous read operations were sequential or not. Then, upon closing a file, if the state values for all previous read operations were sequential, and the last read operation has read the file to the end, then a "sequential file read" synthetic event can be generated. In the same way, a pattern is defined over "file read" semantic events and relevant modeled state values.

The resulting synthetic events are again passed to the mapper. The mapper may have mapping entries for synthetic events as well, which means that the same way that semantic events change the state values, synthetic events can affect them. For instance, a "TCP connection" synthetic event can change the state of a socket to connecting, established, closing, and closed. Previously generated synthetic events may also be passed again through the synthetic event generator, which means that they can be used to generate complex higher-level events. For example, one can generate "library files read" synthetic event from the consecutive "read */lib/* files" synthetic events. Figure 4 depicts this issue.

Another result is that the stateful approach enables the analyst to seek and go back and forth in the trace, select an area, and abstract out only the selected area. In other words, it supports partial trace abstraction. For instance, suppose we see there is a high load at a specific point, we can jump to the starting point of the selected area, load the stored system information, and run abstraction process to get meaningful events and to achieve a high-level understanding of the system execution.

5. Illustrative Examples

Using patterns of semantic events and system states helps to develop a generic synthetic events generator. This way, we are able to generate more meaningful high-level events and to detect more system faults and problems. Here, we provide a few examples that outline aspects of synthetic events we can generate using the proposed method.

5.1. System Load and Performance. By keeping track of the system load and usage (e.g., CPU usage, I/O throughputs, memory usage, etc.), and aggregating them per process, per user, per machine, and per different time intervals, it becomes easy to check the resource load values against predefined threshold values. Thus, by processing the trace events and having defined standard patterns, we can compute the system load and store in the modeled state. For example, for each file open semantic event, we increment the number of opened files for that process and also for the whole system. Likewise, we decrement the number of opened files for each file close semantic event. In the same way, for each schedule in and out event, we add the time duration to the CPU usage of that process. We perform the same processing for memory usage, disk blocks operations, bandwidth usage, and so on and update the corresponding values in the modeled state.

The detector module then compares the stored values against predefined thresholds and detects whether an overload exists in the system. In case of overload, a "system

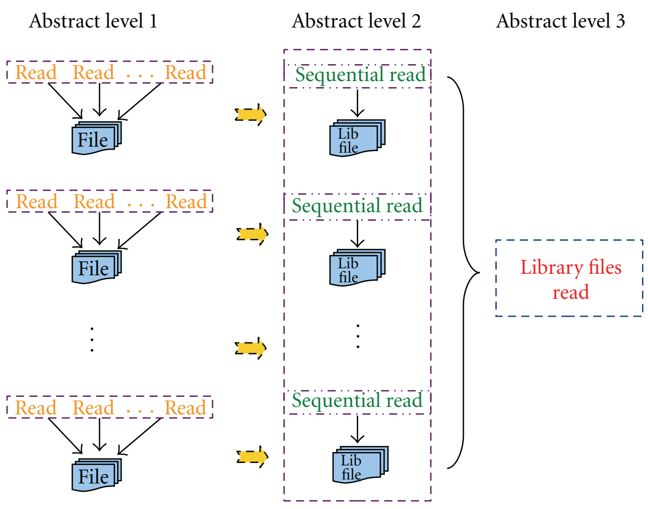


Figure 4: Generating several levels of synthetic events.

overload" synthetic event would be generated, and an alarm would be raised to the system administrator or any predefined monitoring systems. The Administrator or the monitoring system, in turn, responds appropriately to the problem. This solution can be extended to all types of system load and performance problems.

5.2. Denial of Service Attacks. The proposed stateful method can also be used to detect denial of service (DOS) attacks. For instance, a "fork bomb" is a form of denial of service attack which clones a large number of child processes in a short time, in order to saturate the system resources. To detect this attack, one can store the number of fork operations for each process in the modeled state. In this case, upon forking a process, value of a corresponding counter is incremented, and upon killing a process, the value is decremented. Each time the value changes, it is compared to the predetermined threshold value, and in case the threshold value is reached, it will generate a synthetic event and will send an alarm to the system monitoring module. Figure 5 outlines the state machine used for detecting these kinds of attacks.

The same technique may be used for detecting the "Syn flood attack" and for other similar DOS attacks as well. For each connection (even half-opened or established), we keep track of a counter in the modeled state, and the attack (or a possible attack attempt) is detected by comparing the value of the counter to a predefined value. The predefined value can be defined by an expert or by an automated learning technique. These values can be adjusted for different servers and applications.

5.3. Network Scan Detection. Network scanning—especially port scanning—is the process in which a number of packets are sent to a target computer to find weaknesses and open doors to break into that machine. The scanner sends a connection request to targeted ports and, depending on the scanning method, receives a list of open ports, version of operating system, and running services on the other end. Port scanning has legitimate uses in computer networks and is usually considered one of the early steps in most network attacks. Nmap [22] is the most widely used tool for such network scanning. We used Nmap to generate the relevant kernel traces of high-level port scanning. There is actually no way and no reason to stop a user from scanning a computer. However, by detecting port scanning, one can be alerted because of a potential attack.

There are many ways to perform and accordingly detect a port scanning from network packets. A common method, which is implemented in Nmap, involves establishing a typical TCP connection and immediately closing it by sending an RST or an FIN packet and repeating it to different ports at defined time intervals. Another hard-to-detect port scanning method is sending a dead packet (a typical TCP SYN or TCP FIN instead of a regular TCP connection). According to RFC 793 [23], an open port must ignore this packet, and conversely, a closed one must reply with an RST packet. Consequently, any answer from the other end will determine the status of the port: whether it is open, closed, or filtered.

The LTTng kernel tracer traces the packets in both the IP and TCP layers. However, the proposed prototype uses socket-related events (TCP layer) to detect "port scanning" synthetic events. Simplifying the prototype, we consider every TCP connection pattern—even established or half opened—followed by a disconnection, a single port scan. Upon detecting these kinds of events, we update an associated entry in the modeled state. The scanned port number, pattern used, source address, source port, and timestamps of the first packet are stored in that entry. There is a registered method in the detector module which monitors the associated modeled state entry and, in case of detecting a successive port scanning, generates a "ports scanning" synthetic event and raises an alarm to the system-monitoring module. Due to the completeness of the defined rules and patterns, this technique can detect distributed port scanning even with different timings between packets. Figure 6 depicts

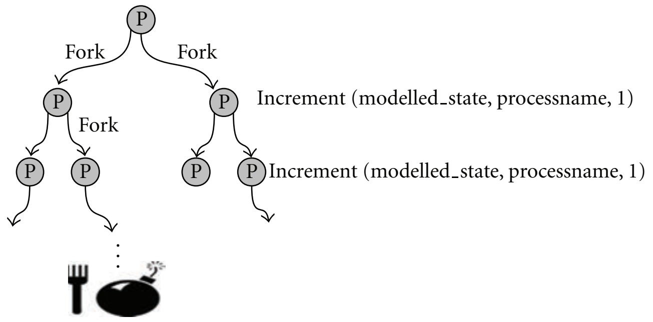

P
Fork
Fork
P
P
Increment (modelled_state, processname, 1)
Fork
P
P
P
P
Increment (modelled_state, processname, 1)

FIGURE 5: State transition for detecting the fork bomb attack.

the state machine used to detect "port scanning" synthetic events.

6. Implementation

We have prototyped a Java tool in Linux that takes the LTTng trace as input and applies the proposed techniques to create the several levels of abstract events. We use the Java library provided by Montplaisi and Dagenais [20] to manage the modeled state and other shared information. Our synthetic event generator stores all the system states as well as the states required for state machines in this tree.

We also created a prototype pattern library that covers common Linux file, network, and process operations. Using this pattern library and the proposed method, we can generate various levels of file operations (e.g., file simple open and close operation, sequential and nonsequential, file read and write, etc.), network connections (e.g., TCP connection, port scanning, etc.), and process operations (fork process, kill process, etc.). The pattern library also includes patterns of some system problems and attacks.

In this implementation, we have defined the patterns in XML format. Also we hardcoded the preactions and postactions for each state transition. Waly and Ktari [3] have developed a language for defining patterns and attack scenarios over LTTng events. Efforts are needed to extend this language to support the patterns of system states. Generally, the required language should be a declarative language with which one can declare the patterns and scenarios and mapping between events and outputs. It can, however, also be a programming-like language with which one can define state variables, pre- and posttransition actions, conditions, output formats, and so forth. Designing such a language is a future work of this project.

We have used a C helper program that simply calls the "wget WWW.YAHOO.COM" in Linux to generate the corresponding kernel traces. Figure 7 shows the highest level of generated synthetic events for this command. The count of generated raw event for this command is 3622. The synthetic event generator has converted these 3600 raw events into less than 10 synthetic events. Here, the synthetic events include reading some library and configuration files, making a TCP connection to WWW.YAHOO.COM, and writing the data read to file index.html.1, following with some other event.

TABLE 3: Number of events in different levels of abstraction.

Size (MB)	Number of events		
	Original trace	Abstract level 1	Abstract level 2
25	2279766	335362	4954
75	5420727	710052	5466
150	8872888	976672	86697
500	37328387	7668926	178426
1000	68961889	20186771	192788
2000	140507496	33924846	328430
5000	328868336	130293720	2099836
10000	621132167	159023500	2247225

7. Performance

For the performance testing, the Linux kernel version 2.6.38.6 is instrumented using LTTng and the tests are performed on a 2.8 GHz with 6 GB ram machine. We will show the results from different points of view. As discussed in the Implementation Section, we have generated two other levels of abstraction in addition to the semantic events. For the first abstract level, we have used 120 patterns, and for the second level, we have used 30 patterns. Following tables and figures show the results for traces with different sizes from 25 MB to 10000 MB. To generate these trace files, we have used "wget -r -l0 URL," "ls –R," and also "grep x . -r" commands consecutively.

7.1. Reduction Ratio.

7.1. Reduction Ratio. As discussed earlier, one goal of the abstraction technique is reducing the trace size. We have measured the reduction ratio of the proposed method. Figure 8 and Table 3 show the reduction of the number of events in the different levels of abstraction.

Table 3 shows the numbers of events at each abstract level. For instance, by applying the synthetic event generator over a 10 GB trace file, we could reduce it to a trace with 25% of the original size but still with mostly the same meaning. It is important to note that the content of trace files at different levels should yield the same interpretation of the system execution. In the Implementation Section, we showed that the same meaning can be extracted from different levels of abstraction. Using the same pattern-matching technique, the

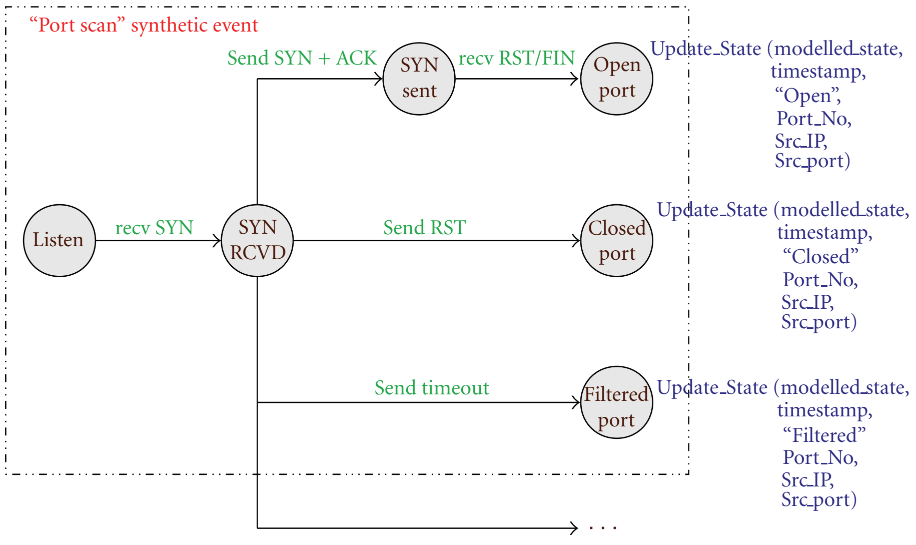
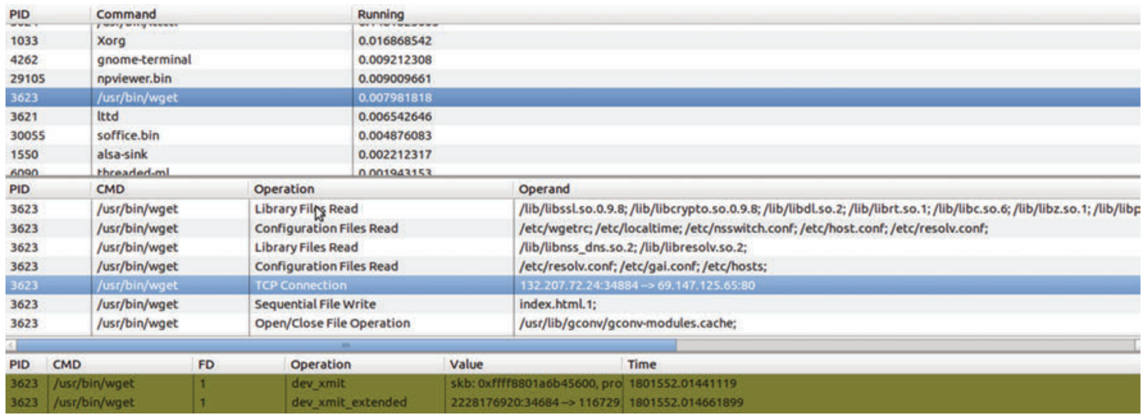

"Port scan" synthetic event
Listen
recv SYN
SYN RCVD
Send SYN + ACK
SYN sent
recv RST/FIN
Open port
Send RST
Closed port
Send timeout
Filtered port
. . .
Update_State (modelled_state, timestamp, "Open", Port_No, Src_IP, Src_port)
Update_State (modelled_state, timestamp, "Closed" Port_No, Src_IP, Src_port)
Update_State (modelled_state, timestamp, "Filtered" Port_No, Src_IP, Src_port)

Figure 6: State transition for detecting the port scanning.



PID	Command	Running
1033	Xorg	0.016868542
4262	gnome-terminal	0.009212308
29105	npviewer.bin	0.009009661
3623	/usr/bin/wget	0.007981818
3621	lttd	0.006542646
30055	soffice.bin	0.004876083
1550	alsa-sink	0.002212317
6090	threaded-ml	0.001943153

PID	CMD	Operation	Operand
3623	/usr/bin/wget	Library Files Read	/lib/libssl.so.0.9.8; /lib/libcrypto.so.0.9.8; /lib/libdl.so.2; /lib/librt.so.1; /lib/libc.so.6; /lib/libz.so.1; /lib/libp
3623	/usr/bin/wget	Configuration Files Read	/etc/wgetrc; /etc/localtime; /etc/nsswitch.conf; /etc/host.conf; /etc/resolv.conf;
3623	/usr/bin/wget	Library Files Read	/lib/libnss_dns.so.2; /lib/libresolv.so.2;
3623	/usr/bin/wget	Configuration Files Read	/etc/resolv.conf; /etc/gai.conf; /etc/hosts;
3623	/usr/bin/wget	TCP Connection	132.207.72.24:34884 –> 69.147.125.65:80
3623	/usr/bin/wget	Sequential File Write	index.html.1;
3623	/usr/bin/wget	Open/Close File Operation	/usr/lib/gconv/gconv-modules.cache;

PID	CMD	FD	Operation	Value	Time
3623	/usr/bin/wget	1	dev_xmit	skb: 0xffff8801a6b45600, pro	1801552.01441119
3623	/usr/bin/wget	1	dev_xmit_extended	2228176920:34684 –> 116729	1801552.014661899

Figure 7: A view of the implemented stateful synthetic event generator.

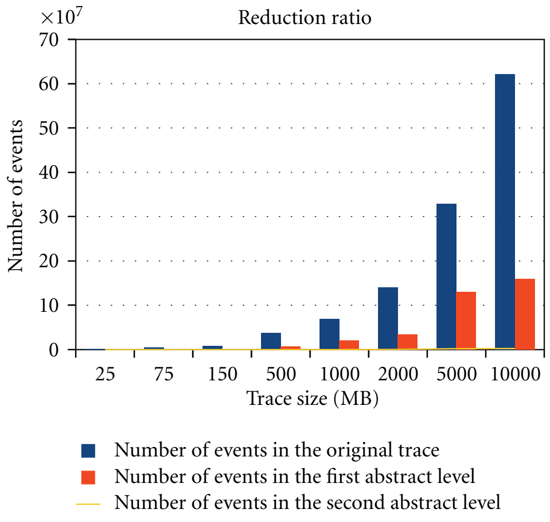


Figure 8: Reduction ratio.

reduction ratio is related to the number of patterns and also the number of containing events in each pattern. For that, the reduction ratio of the different examined traces is not the same.

7.2. Patterns and Containing Events. Table 4 shows the number of selected synthetic events of various trace sizes for the first abstract level. In the proposed pattern library, a set of patterns have been defined for each of these synthetic events, so that by applying them over trace events and by retrieving the current system state, the engine is able to detect those events. We have defined 120 patterns for abstracting out the trace events and generating the first level of abstraction. These patterns cover important aspects of file, socket, process, and also system wide operations. The patterns are stored in an XML file, and the system is defined in such a way that administrators can easily add new patterns to the system.

Table 4: Count of different event types in first level of abstraction.

Events count	Size (MB)	Number of synthetic events in first level of abstraction								
		File operations				Network operations				
		File open	File read	File write	File close	Socket create	Socket connect	Socket receive	Socket send	Socket close
2279766	25	2727	8401	12913	2327	150	112	4583	7911	150
5420727	75	2474	18563	30251	2122	370	718	21523	26570	611
8872888	150	86780	59108	20913	86536	154	106	7574	10767	161
37328387	500	87484	158484	1025703	88143	673	979	60880	70651	1070
68961889	1000	98583	218789	6507052	96226	159	73	57965	62003	168
140507496	2000	161458	562239	5980178	161577	2140	2500	166420	245718	2638
328868336	5000	1045710	612821	23001032	1044562	4592	5154	137733	296566	5356
621132167	10000	755647	3541833	23214444	720016	27676	20741	735271	1288181	29059

As the second step, Table 5 shows some important synthetic events for the second abstract level. To generate these events, a set of patterns were defined over the events of the first level of abstraction and also the relevant modeled state values. For instance, "check file" synthetic events contain a sequence of an open file, check file status or check file permission, and a close file. Also "sequentially file read (write)" refers to a set of subsequent file operations that consist of a file open, read from (write to) a file sequentially, and finally close it. An "HTTP (DNS) connection" is detected through a pattern of socket create, socket connect to a 80/53 port with transport protocol equal to TCP (UDP), zero or more send/receive data, and finally socket close events. However, a "network connection" synthetic event means that there is a complete connection (sequence of socket create-socket connect, send/receive data, and close socket), but that the engine could not find the type of connection protocol or the destination port. This happens because of probable missing events. It may also occur if the connection was already established at the starting time of the trace. At this level, we have defined 30 different patterns for detecting such kind of synthetic events.

It would be interesting to know the number of events participating to form a synthetic event on one upper level. Table 6 shows the average number of events from one level below contained in each synthetic event.

As shown in Table 6, "check file" synthetic events always contain three events: open a file, check a file attribute, and close that file. It is important to note that the numbers here only show the containing events from one level below. On the other hand, in this example, each of these three events can in turn contain several events from a lower level. As another example, the average number of 48 events obtained from a 5 GB trace means that each "sequentially file write" synthetic event contains 48 events on average: one for open file, one for close file, and the rest for write events. The null value for the DNS connection is because there is no such connection for those traces.

7.3. Execution Time. As discussed earlier, one of the important features of the proposed method is its execution efficiency. In this section, execution times for different trace sizes will be shown. Table 7 shows the time spent to read the relevant events, to look for patterns, and to generate the first abstraction level.

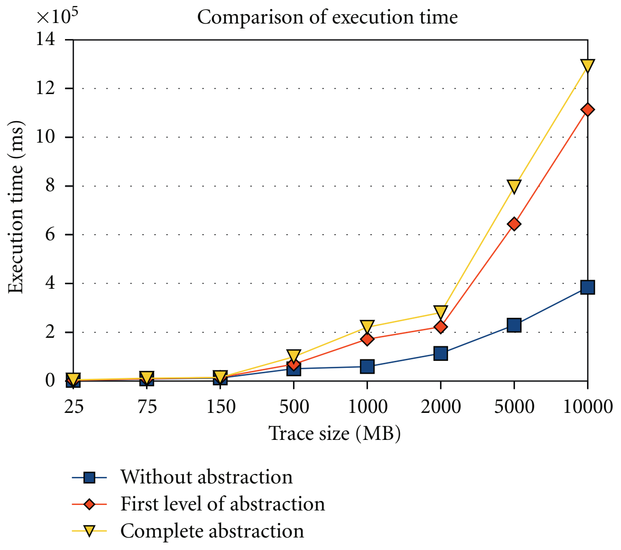


Figure 9: Execution time comparisons.

The numbers in Table 7 show the time spent for reading the events, looking for the patterns, and generating the abstract events. The execution time for checking each pattern has a relatively minor impact when checking all patterns simultaneously. Reading the trace events and finding relevant events for each pattern takes most of the analyzing time.

Figure 9 shows the differences between execution times spent for different levels of abstraction.

In Figure 9, the blue line shows the execution time needed for reading the trace events. For this step, no pattern was specified, and the diagram only shows the time needed for reading all trace events. In the same way, the orange line shows the execution time for generating the first level of abstract events. The execution time consists of reading events, looking up the patterns, and generating the abstract events. The yellow line depicts the time needed for analyzing

Table 5: Count of different event types in second level of abstraction.

Events count	Size (MB)	Number of synthetic events in second level of abstraction						
		File operations				Network operations		
		Check file	Sequentially file read	Sequentially file write	Read write file	Network connection	HTTP connection	DNS connection
2279766	25	928	673	496	230	43	103	4
5420727	75	1117	742	253	10	372	193	46
8872888	150	66141	19742	440	213	58	103	0
37328387	500	53371	32093	1791	888	688	274	108
68961889	1000	81607	14343	207	69	139	29	0
140507496	2000	61647	96921	2505	504	1486	867	285
328868336	5000	948436	87095	8047	984	1921	3116	319
621132167	10000	395067	303452	20267	1230	11740	16668	651

Table 6: Average number of containing events for generating the upper level synthetic events.

Events count	Size (MB)	Average number of containing events						
		Check file	Sequentially file read	Sequentially file write	Read write file	Network connection	HTTP connection	DNS connection
2279766	25	3	3	10	9	1	38	5
5420727	75	3	3	25	87	3	37	5
8872888	150	3	4	11	9	2	38	0
37328387	500	3	4	10	26	2	44	5
68961889	1000	3	3	20	67	4	82	0
140507496	2000	3	4	61	16	5	92	5
328868336	5000	3	6	48	62	3	34	5
621132167	10000	3	4	34	120	4	42	5

and generating both levels of abstract events. Differences between the orange and yellow lines with the blue line show that the time needed for analyzing and generating the first level of abstraction is more than the time needed for the second level. This is explained by the fact that the number of patterns in the first level is much greater than the number of patterns in the second level. The results show that the execution time is relatively linear with the trace size and also the number of relevant events. However, there exists other factors that affect the results. The complexity of patterns and also the number of containing events for each pattern may lead to different execution times for different synthetic events. For example, in Table 7, we see different values for different synthetic events for the same trace file, and the reason is that they have different scenarios with different complexities and different numbers of containing events.

Another important factor is the number of coexisting patterns. As shown in Figure 9, the number of patterns and the number of coexisting patterns affect the execution times for the two different abstraction levels. Since the first level deals with file and socket operations, they need to be called for each file/socket access, thus having a large impact on the performance and execution time of the analyzer. By contrast, in the second abstraction level, the analyzer works with fewer coexisting patterns, less often called during a process lifetime; thus, less time is needed for analyzing and generating the second level of abstract events. The execution time is therefore related to the size of the trace files, the number of relevant events, the number of coexisting patterns, and also the complexity of patterns.

8. Conclusion and Further Work

In this paper, we proposed an abstraction method that uses a set of patterns over semantic trace events and system state values to generate synthetic events. Using the notion of semantic events (the events of a generic type that replace platform-dependent events) can help decouple the synthetic event generator from the system and tracer-dependent events. Using semantic events as well as system state values makes the abstraction process more generic to support different versions of trace formats and operating system kernels and also to support both the kernel and user level tracing. As shown in the Illustrative Examples Section, using proposed techniques, we efficiently generate a wide range of synthetic events that reveal more of the system behaviors and can also be used to detect a larger range of system problems.

Although most of the synthetic events can be defined by patterns and state machines, we do not support the synthetic events that are not representable using state transitions. For

Table 7: Execution time for generating the first abstraction level.

			Time spent (ms) for analysing and generating the first level of abstract events								
Events count	Size (MB)	All (ms)	File operations				Network operations				
			File open	File read	File write	File close	Socket create	Socket connect	Socket receive	Socket send	Socket close
2279766	25	4748	3963	4024	3995	4038	4151	3952	3871	3937	3777
5420727	75	9176	5900	6195	6394	6218	6394	6257	5884	6014	5957
8872888	150	12735	10403	10308	10183	10282	9975	10025	97125	10125	10424
37328387	500	58477	50251	50855	51928	50424	49366	47896	48962	49163	48104
68961889	1000	148284	128252	129157	138988	128186	127912	128312	127191	126915	128061
140507496	2000	226569	177913	180017	187366	178640	177389	174385	179372	178509	179097
328868336	5000	650228	513230	514012	546782	512659	510139	507873	511201	510139	512771
621132167	10000	1105124	902245	916292	941668	909893	907177	901127	899976	904227	874227

example, a dependency analysis between different processes and resources, leading to the computation of the critical path for a request, cannot be defined as a simple pattern. Another possible limitation is related to the output completeness of the tracers. Because not every state modification is logged with a tracer, this may somehow limit the proposed technique.

Using the proposed method and having defined a complete pattern library can also lead to an efficient host-based intrusion detection system. In our project, expanding the rule base and pattern library is an important future work. We have implemented a prototype pattern library that works over semantic events and system states. However, more work is needed to complete it and to support more aspects of system behavior. Examples of the patterns we should extend include memory usage and interprocess communications.

As mentioned in the Implementation Section, there is a need to further develop the supporting language. This language can be declarative or similar to a programing language and should support the requirements needed to implement the proposed method. It could be used for defining the mapping table between raw events and semantic events, as well as the mapping between events and state changes. We will focus on extending the language defined in [3] in a future investigation.

References

[1] M. Desnoyers and M. R. Dagenais, "The LTTng tracer: a low impact performance and behavior monitor for GNU/Linux," in *Proceedings of the Ottawa Linux Symposium*, 2006.

[2] W. Fadel, *Techniques for the abstraction of system call traces [M.Sc.A. dissertation]*, Concordia University, 2010.

[3] H. Waly and B. Ktari, "A complete framework for kernel trace analysis," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE '11)*, pp. 001426–001430, Niagara Falls, Canada, May 2011.

[4] J. P. Black, M. H. Coffin, D. J. Taylor, T. Kunz, and T. Basten, "Linking specification, abstraction, and debugging," CCNG Technical Report E-232, Computer Communications and Networks Group, University of Waterloo, 1993.

[5] M. Auguston, A. Gates, and M. Lujan, "Defining a program behavior model for dynamic analyzers," in *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering (SEKE '97)*, pp. 257–262, Madrid, Spain, June 1997.

[6] G. Matni and M. Dagenais, "Automata-based approach for kernel trace analysis," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE '09)*, pp. 970–973, May 2009.

[7] L. Fu, *Exploration and visualization of large execution traces [M.Sc.A. dissertation]*, University of Ottawa, 2005.

[8] A. Hamou-Lhadj and T. Lethbridge, "Survey of trace exploration tools and techniques," in *Proceedings of the 14th IBM Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 42–55, IBM Press, 2004.

[9] W. D. Pauw, R. Helm, D. Kimelman, and J. M. Vlissides, "Visualizing the behavior of object-oriented systems," in *Proceedings of the 8th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pp. 326–337, ACM, 1993.

[10] D. B. Lange and Y. Nakamura, "Object-oriented program tracing and visualization," *Computer*, vol. 30, no. 5, pp. 63–70, 1997.

[11] D. F. Jerding, J. T. Stasko, and T. Ball, "Visualizing interactions in program executions," in *Proceedings of the 19th IEEE International Conference on Software Engineering*, pp. 360–370, May 1997.

[12] A. Chan, R. Holmes, G. C. Murphy, and A. T. T. Ying, "Scaling an Object-oriented system execution visualizer through sampling," in *Proceedings of the IEEE International Workshop on Program Comprehension (ICPC '03)*, 2003.

[13] S. P. Reiss, "Visualizing Java in action," in *Proceedings of the ACM Symposium on Software Visualization (SoftVis '03)*, pp. 57–65, ACM, June 2003.

[14] T. Systä, K. Koskimies, and H. Müller, "Shimba—an environment for reverse engineering Java software systems," *Software—Practice and Experience*, vol. 31, no. 4, pp. 371–394, 2001.

[15] S. T. Eckmann, G. Vigna, and R. A. Kemmerer, "STATL: an attack language for state-based intrusion detection," *Journal of Computer Security*, vol. 10, no. 1-2, pp. 71–103, 2002.

[16] P. Uppuluri, *Intrusion detection/prevention using behavior specifications [Ph.D. dissertation]*, State University of New York at Stony Brook, New York, NY, USA, 2003.

[17] S. Kumar, *Classification and detection of computer intrusions [Ph.D. thesis]*, CERIAS lab, Purdue University, 1995.

[18] J. L. Lin, X. S. Wang, and S. Jajodia, "Abstraction-based misuse detection: high-level specifications and adaptable strategies," in *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSFW '98)*, pp. 190–201, Rockport, Mass, USA, June 1998.

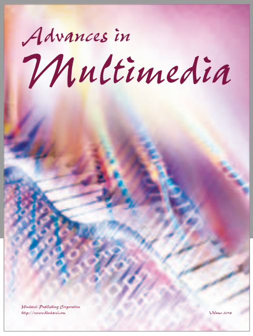
[19] P. Beaucamps, I. Gnaedig, and J. Y. Marion, "Behavior abstraction in malware analysis," in *Proceedings of the Runtime Verification Conference (RV '10)*, pp. 168–182, 2010.

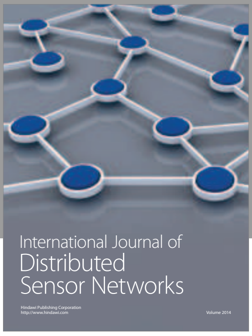
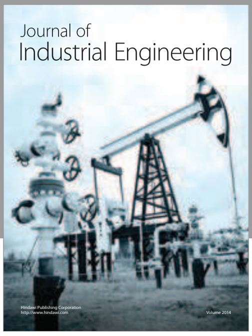
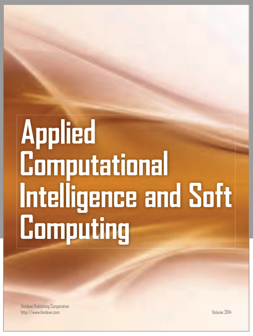
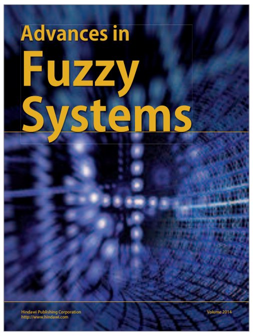
[20] A. Montplaisi and M. R. Dagenais, *Stockage sur disque pour accs rapide dattributs avec intervalles de temps [M.Sc.A. dissertation]*, Dorsal lab, Ecole Polytechnique de Montreal, Montreal, Canada, 2011.

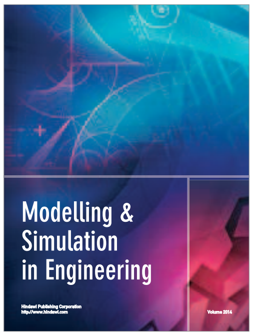
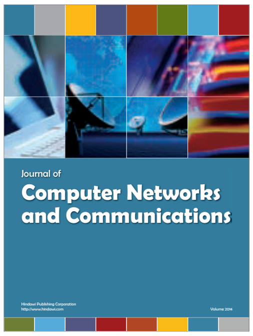
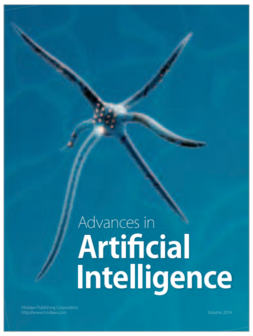
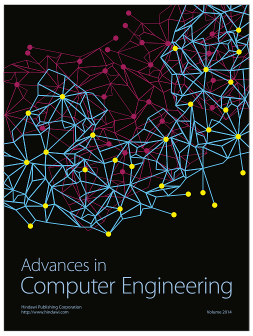
[21] M. M. Sebring, E. Shellhouse, M. Hanna, and R. A. Whitehurst, "Expert systems in intrusion detection: a case study," in *Proceedings of the National Computer Security Conference*, pp. 74–81, 1988.

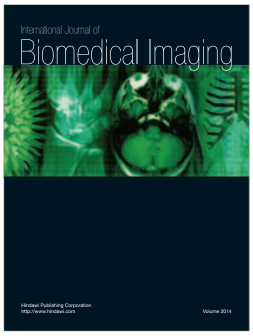
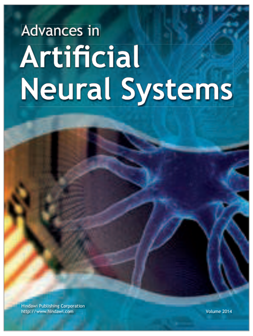
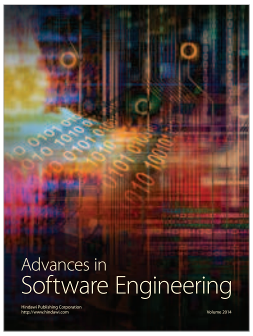
[22] 2011, http://www.nmap.org/.

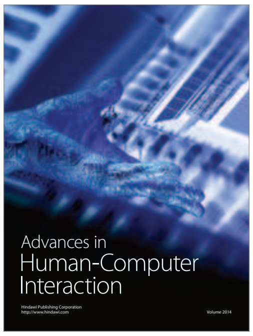
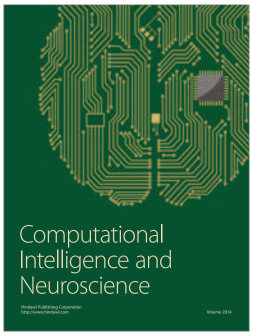
[23] RFC 793: Transmission Control Protocol, 2011, http://www.faqs.org/rfcs/rfc793.html.


Advances in
Multimedia

The Scientific World Journal

International Journal of
Distributed Sensor Networks

Journal of
Industrial Engineering

Applied Computational Intelligence and Soft Computing

Advances in
Fuzzy Systems


Modelling & Simulation in Engineering

Journal of
Computer Networks and Communications

Advances in
Artificial Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games Technology

International Journal of
Biomedical Imaging

Advances in
Artificial Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer Interaction

Computational Intelligence and Neuroscience

International Journal of
Reconfigurable Computing

Journal of
Electrical and Computer Engineering