Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

Secured Networking by Sandboxing LINUX 2.6

Hrushikesha Mohanty*, M. VenkataSwamy*, P. Thilak*, Srini Ramaswamy†
*Department of Computer and Information Sciences,
University of Hyderabad, Hyderabad, India 500 046
hmcs_hcu@yahoo.com
†Dept. of Computer Science
University of Arkansas at Little Rock, Little Rock , USA
srini@acm.org

***Abstract*—From system security point of view, system calls are vulnerable as they operate in kernel space. Hence monitoring of system call patterns performed by an application has been extensively studied for the development of Intrusion Detection Systems(IDS), which have to respond immediately to abnormal behaviors. However these IDSs have limitations in detecting new types of attacks. Policy driven IDSs have the ability to detect novel attacks based on policies written for system activities. In this paper we propose a hybrid architecture for IDSs, that combines the features of both policy driven IDS and system call based IDS and the idea is experimented for sandboxing Linux 2.6.**

***Index Terms*—Intrusion, Linux Kernel, Sandboxing.**

I. Introduction

Any unauthorized attempts to access, manipulate, modify, or destroy information or to render a system unreliable or unusable is defined as intrusion [1]. An intrusion takes place for variety of reasons like accessing system resources using a backdoor - unregulated system entry point. This happens when a system is wrongly configured to allow direct or indirect access to unauthorized users. This may also happen due to execution of a piece of malicious software. A system whose resources are successfully exploited by intruders is considered as 'compromised'. A system may go to compromised state without knowledge of the system users and so the danger involved. And the detection of intrusions has been always a challenging problem. A continuous watch on system performance is necessary to detect an intrusion at right time before it makes an irreparable damage[2]. Such a system is termed as "Intrusion Detection System (*IDS*)". A successful *ids* should provide the following services:[3]

- Monitoring of users activities as well as system performances
- Guarding system configuration
- Maintaining integrity of the system data files
- Identifying attack patterns
- Deriving new attack pattern

As most of the systems are currently networked, intrusion detection system for an installation needs to be vigilant of activities at a system as well as its network. Usually, a firewall around a system keeps it isolated from intrusions through its network. Based on its implementation, we categorize detection systems as *host-based ids* and *network-based ids* [4]. The standard procedures for such a system is to identify a pattern in system / user behavior and match the pattern with known attack patterns or signatures. A detection system may commit errors at pattern identification or matching a pattern to the identified patterns. Due to such an error an identification system may either fail to identify or wrongly identify an intrusion. The research in this field aims to reduce errors in detecting intrusions. Some of the important methods include the following.

- Anomaly Detection: is a method that works in two phases. One deals with identification of *normal* behavior of an authenticated user. This identification process is automated and the techniques for the purpose ranges from statistical methods to machine intelligence including soft computing techniques. Using any of these techniques, an intrusion detection makes a repository of normal behaviors for users. And then, when an user is in session with an system, its intrusion detection system goes into monitor mode to identify behavior patterns for the users. And then the detection system matches a deduced pattern to the stored patterns defined by a statistical profile of normal behavior. A pattern that deviates significantly from the normal profile is considered an attack. [5], [6] Anomaly detection technique defines the baseline of the normal behavior on a system and detects intrusions using deviations from this baseline. The method can detect new attacks against systems by comparing current activities against statistical models of past behavior. However, the system may generate too many false alarms (False Positives) by narrowly defining the baseline of normal behaviors. [1]
- Attack Detection: technique intends to identify attacks that may lead to disruption of services, denial of services or even damage to system e.g. deletion of crucial files, theft of important data. The technique crucially depends on intrusion awareness of a detection system. An IDS system should have a library of a Hack signatures and the system should also be capable of identifying an attack by deriving an attack pattern and matching that pattern with the available repository of attack signatures[7], [8]. Though this technique is widely practiced still has serious limitation that constrains the capability of an IDS to identify attacks that are only listed. Such an IDS turns blind to new variety of attacks.
- Policy-driven Intrusion Detection: technique intends to

978-1-4244-2794-9/09/$25.00 ©2009 IEEE

remove such limitation by providing facility to system administrator for providing new policies that are useful at present scenario to defend a computing system. Usually such approach is being used for access control. Users and even processes are characterized by their roles and for each role policy(ies) is(are) specified. Based on these policies users/processes are allowed to access resources of a computing system[12]. Again this technique is constrained to the efficiency of policies. Another policy driven IDS is reported in [9]. This method makes use of Linux resource access structures and looks for fields in structures to test the permissibility of user. Thus, the policy lacks clarity being very close to system syntax. Also, policies do not specify different scenarios evolving due to states of computation. Policies are successfully used to specify scenarios enabling users to access system i.e. access control system. Such uses are not found in IDS. However, the policies for user accesses are formulated considering organizational practices and those for a process are formulated considering its states. For both the cases the higher level abstractions are required in formulating policies. But, this kind of abstractions is prone to lapses as certain aspects (with respect to organization / system states) can be overlooked.

Considering the limitation we propose to define policies at each resource level i.e. at the lowest level. Each resource is an atomic entity and it passes through certain defined states while the entity is being used during computation. Considering its dynamic behavior, we propose to define policies and while the entity is in use these policies are evaluated and in case of any violation alarm is raised indicating a possible intrusion. This approach is popularly known as sandboxing and the concept is discussed in the next section.

II. SANDBOXING

The idea behind sandboxing is to provide an all-time safeguard to computing resources like files, memory etc. by defining policies for detecting their misuses. This is a conservative approach as it processes each request before entertaining any access to a resource. While other approaches, detects an intrusion after it happens thus making the computing system vulnerable. Thus, the proposed approach is almost successful in detecting attacks that are harmful to system.The basic operations on any resource includes access, read and write. A state diagram depicting life cycle of a resource usage is presented by the following state diagram (figure:1).

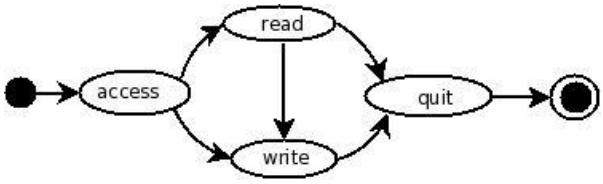


Fig. 1. Resource Usage State Diagram

A process intending to use a resource, access the resource and then either may read or/and write before quitting the resource. Then the usage of the resource comes to an end. In order to provide security we can specify policies of each state for legal usage of the resource. For an example, a process in supervisory state can only access the password file for read and write. An process may have permission only to read the file but not to write. A process may be allowed to acquire a specified amount of memory. These issues can be written as policies and attached to a resource state. Just before a resource reaches a state the corresponding policies are invoked for enforcement. This basic principle is followed in this paper to safeguard a resource.
A policy abstractly can be specified as a rule in the form of if-then statement as listed in Listing:1

Listing 1
POLICY FORMAT

```
Policy :: if {< condition >} then {< action >}
condition :: < operand > < op > [< operand >]
action :: < procedure > | < operand = operand >
```

A policy has two components viz. condition and action, the first component is composed of predicates and the second has procedures primarily for restoration of system status and issuing warning message to alert system administrators.
Policies are incorporated at operating system level as plug-in components to Linux Kernel 2.6. Many policy based security systems enforce polices at application level and some at middle ware level[10]. The first approach we feel not useful to provide a secured system as there could be lapses at application designers. Offloading responsibility to users is not a wise decision. Putting responsibility on middle ware interfacing applications and operating system is operationally sometimes not feasible for accessing attributes of system resources e.g. task control block, system call routines and associated data structures. so, we have decided to enforce policies at operating system kernel level. We have developed a software that translates a policy to a 'C' program. Such a 'C' program is added to Linux 2.6 kernel. As we have told a policy is meant for enforcement at certain states of resource usage, so the 'C' program corresponding to the policy is to be executed at associated states. Usually making a system call for system resource usage, the corresponding interrupt service routine is executed.

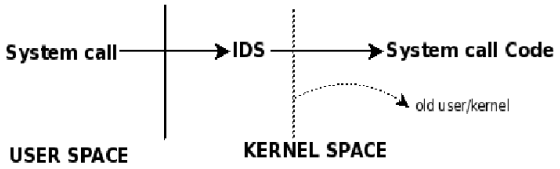


Fig. 2. Positioning IDS

A system call is generated at user space but the call is processed at the kernel level. The proposed IDS essentially executes 'C' routines implementing policies corresponding to system calls. The routines are located at kernel space and interfaced to respective system calls and the corresponding call service codes as shown in figure:2.

Thus IDS has become a part of the kernel. The steps leading to policy execution is shown in the following sequence diagram (figure:3)

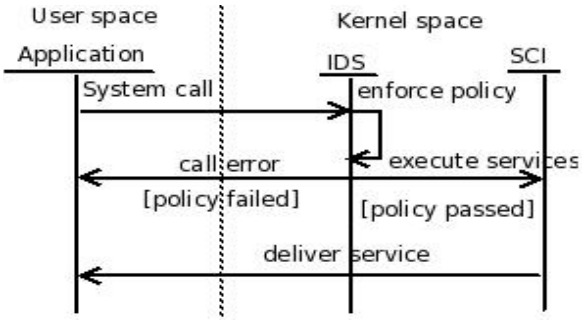


Fig. 3. Policy enforcement process

First, a system call issued by an application is caught by IDS and the policy in 'C' corresponding to the call is executed. If the policy succeeds then the system is allowed to avail the resource and subsequently the call service is delivered to the application. In case the policy fails then the call is not allowed for service and the execution control returns to the application. Thus on deploying IDS, the sand boxing of an operating system is achieved. Though, the concept is simple, its implementation is tricky for that needs tinkering of internals of operating system. In the next section we present a method of sandboxing of Linux Kernel 2.6.

III. Sandboxing Linux

Like any other operating system, GNU/Linux has multi-layered architecture as shown in figure:4 comprising major subsystems including process management, memory management, virtual filesystem, network management, device drivers etc. At the top, SCI (System Call Interface) provides a way for applications to gain access to kernel for accessing system resources through kernel services.

An application on making a system call invokes SCI using Interrupt Service Routine (ISR) at 0x80. The routine decodes a call requirement i.e. the service and looks for the routine that can provide the service. The array sys_call_table[] keeps the address of kernel service routines. For a service requested by a system call, the address of the call processing routine is found and then the routine is invokes. In Listing:2, we present a set of system calls, structures of sys_call_table and call processing routine.

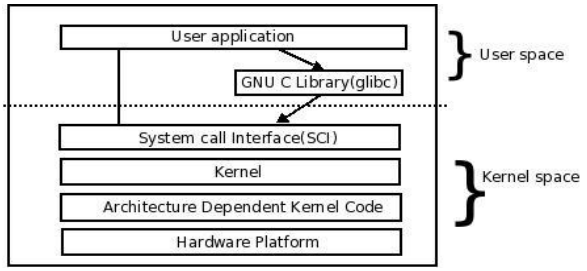


Fig. 4. GNU/Linux Architecture

Linux v2.6 provides a set of system calls approximately counting more than 321. These calls are mainly for open, close, read, and write operations on each resource like files, sockets, etc.

Listing 2
SOME SYSTEM CALLS

```
long sys_restart_syscall(void);
long sys_exit(int error_code);
int sys_fork();
ssize_t sys_read(unsigned int fd, char __user *
    buf, size_t count);
    ..
```

These system calls are specified in "/include/linux/syscalls.h". Listing:3 presents a structure specifying sys_call_table.

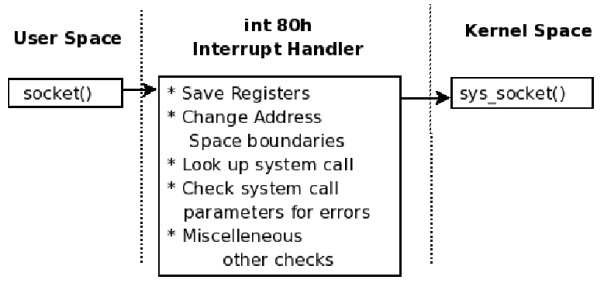
Listing 3
SYS_CALL_TABLE STRUCTURE

```
  // for each system call one entry in table
ENTRY(sys_call_table)
  .long sys_restart_syscall
  .long sys_exit
  .long sys_fork
  .long sys_read
  .long sys_write
  .long sys_open
    ..
```

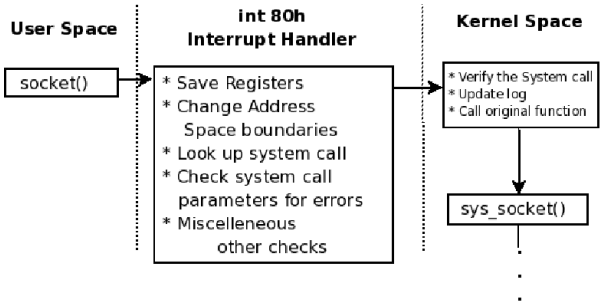
Once a system call is made (usually at user space) the interrupt handler at 0x80 before invoking the intended system call, it saves all registers before system enters to the kernel space. It also checks whether the made call is a valid call. That is if the requested system call id (identification) number is not in the defined range (available at /arch/x86/kernel/sys_call_table_32.S) then the call is aborted. Else *sys_call_table(id) is called to invoke the target system call on picking up its address from sys_call_table. It may be recalled here that the table is an array of long integers and each of those has an address to a particular system call.

Sandboxing of a system call is performed just before the routine *sys_call_table() is invoked by the system call interrupt (SCI) handler. We position sandboxing routines in between

SCI and *sys_call_table(). A sandboxing routine implements a policy that detects a system call posing a potential threat for the resource it uses. A policy specified in XML is translated to assembly language for Linux v2.6 on particular platform. In an another paper we propose a method to translate policies in XML form to an executable form. For sandboxing, sys_call_table datastructure is modified. Each of its nodes now stores address of an executable policy. As before nodes in the table are mapped to policies meant to verify system calls. On modification of the table, the sequence of activities for interrupt handling changes. On initiation of a system call, its interrupt handler access the table to call the routine that executes the policy corresponding to the call. If the call is vested by the policy then the routine that processes the call is invoked. The original sequence of activities and the modified are shown in figure:5(a),5(b) respectively.



(a) Original sequence



(b) Modified sequence

Fig. 5. Sequence of control flow

In order to intercept system calls following the methods as discussed in the previous section, Linux kernel is to be modified. The kernel needs to export sys_call_table for further use. Besides this, it should make the table array writable data section. So to export the table and to make it writable the following patch (Listing:4)is added to the kernel and it also needs to be recompiled.

After recompiling the kernel and booting the machine from the new kernel, we are all set to have host based intrusion detection system. The updated Linux kernel allows intrusion detection system (i.e. that executes policies) to run in kernel space thus avoids context switching for policy execution in contrast to the recent one reported in [13]. In next section we will present an application of our method to monitor network system calls.

Listing 4
KERNEL PATCH

```
  arch/i386/kernel/i386_ksyms.c
+extern void * sys_call_table;
+EXPORT_SYMBOL ( sys_call_table );
  arch/i386/kernel/entry.S
-.section .rodata, "a"
+.section .data, "aw"
```

IV. SANDBOXING NETWORK SYSTEM CALLS

In this section we discuss on implementation of the proposed scheme for sandboxing system network calls. Linux uses one sys_socketcall for all network operations like open, close, bind network ports. Usually the call takes an integer as an input parameter and a switch statement operates on integer to invoke the intended network system call operation. This can be viewed from the following skeletal code (Listing:5) for sys_socketcall (at /net/socket.c)

Listing 5
ORIGINAL SYS_SOCKETCALL CODE

```
asmlinkage long sys_socketcall(int call,
    unsigned long __user *args)
{
  if( call <1 || call > SYS_PACCEPT )
    return -EINVAL;
  switch(call) {
    case SYS_SOCKET:
      sys_socket(a[0], a[1], &a[2])
        .
        .
      break;
    case SYS_BIND:
        .
        .
      break;
    ..
  }
}    // end of sys_socketcall
```

Then for each of the cases corresponding routines like asmlinkage long sys_socket(int family, int type, int protocol) etc. are defined.

For sandboxing network calls we have modified the 102th entry of sys_call_table by putting the address of a routine say policy_sys_socketcall. This routine implements the policy for network call verification and then the routine calls the original sys_socketcall shown in Listing:5. In Listing:6, we present a framework of the sandboxing routine policy_sys_socketcall.

In the above routine (Listing:6) for SYS_SOCKET case the policy for not allowing more than hundred sockets to open, is implemented. Upon confirming the compliance of the policy

Listing 6
POLICY_SYS_SOCKETCALL CODE

```
asmlinkage long policy_sys_socketcall(int call,
      unsigned long __user *args)
{
  switch(call) {
    case SYS_SOCKET:
      no_of_sockets_opened++; // does not allow
           more than 100 sockets to open}
      if( no_of_sockets_opened > 100)
        return -1;
      else
        return sys_socketcall(call, args);
        .
        .
      break;
    case SYS_BIND:
        ..
      break;
    ..
  }
}  // end of policy_sys_socketcall
```

the original routine sys_socketcall is invoked. In case of non compliance of the policy the processing of the network call is aborted. We have performed an experiment to demonstrate the efficacy of our method of sandboxing at kernel level. Next we present the experiment.

A. Experiment

We have designed an experiment for sandboxing of network system calls. The objectives of the experiment are to 1. check the possibility of intercepting a system call and in prior to execution of the call invoking a policy to judge that the call uses permissible resources 2. measure the overhead incurred due to policy evaluation. As described in the previous sections, we begin with the patching of Linux 2.6 kernel with the patch given in (Listing:4); then compiling and installing the kernel. The newly installed kernel allows us to modify the sys_call_table array. A system call viz. sys_socketcall is responsible for all the network operations via the socket interface. So we write a kernel module sandbox_linux which on initialization changes the address at offset __NR_socketcall of sys_call_table array. The initialization code listed in Listing:7

Listing 7
SANDBOX_LINUX MODULE INITIALIZATION CODE

```
static int sandbox_linux_init (void)
{
  old_sys_socketcall = sys_call_table[
      __NR_socketcall];
  sys_call_table[__NR_socketcall] =
      policy_sys_socketcall;
  return 0;
}
```

At offset __NR_socketcall of sys_call_table, a pointer points to sys_socketcall. Then, for sandboxing, we redirect the pointer to a procedure that implements the policy for safe execution of network calls. On execution of the procedure the control of execution moves for execution of network system calls. In order to achieve the said changes we have proposed two add-ons to the kernel given in Listing:7 and Listing:8. The Listing:7 reorients pointer to policy_sys_socketcall and the Listing:8 sets the pointer back to sys_socketcall.
As and when Linux kernel receives a call for network operation, the add-on policy_sys_socketcall (Listing:6) is executed. This procedure is expected to implement all the checks required to be done to safe guard system resources as defined by security policy specification. For an example, we have defined a policy to ensure that at any time not more than hundred connections to be opened. In case this situation occurs, the system would take action as it is directed to do in policy specification. In this case, it may be seen that in case of violation an error condition is returned.

Listing 8
SANDBOX_LINUX MODULE EXIT CODE

```
static void sandbox_linux_exit (void)
{
    /* Return the system call back to normal */
  sys_call_table[__NR_socketcall] =
      old_sys_socketcall;
}
```

The routine policy_sys_socketcall shown in Listing:6, initialization and exit module codes presented in Listing:7 and Listing:8 are added to Linux 2.6 by making use of *insmod* utility , thereby making the sandbox_linux ready for deployment. On inserting the module into kernel, all the network system calls are bypassed through the policy implemented routine(policy_sys_socketcall). Thus the kernel is augmented for sandboxing network system calls.

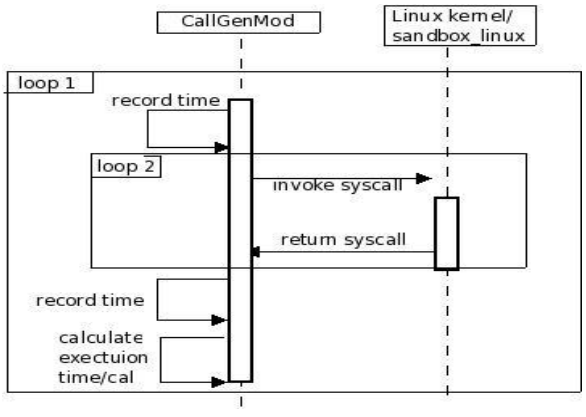


Fig. 6. Sequence diagram for experiment

Now, we are ready to evaluate the augmented kernel with respect to the desired objectives. Experiment includes *CallGenMod* a call generation module and a Linux module.

The experiment is carried out in two phases. First phase uses Linux2.6 module and the second phase uses sandbox_linux. A sequence diagram in Figure:6 shows the dynamic behavior of the modules in the second phase. Each phase, has several checkpoints (loop-1 in Figure:6) and at each point, the Linux kernel is subjected to several network system calls (loop-2 in Figure:6) in burst mode ranging from zero to 30,000 calls. Thus, at a given time the system is made to respond to a multiple of thousand network system calls. And for every completion of loop-2,system response time is read.

The time of execution and overhead in sandbox_linux against number of sys_socketcall is plotted and resulting graph is shown in figure:7.

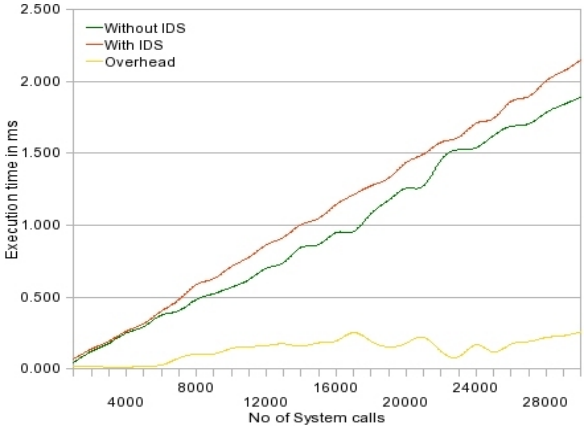


Fig. 7. Time of execution vs no of system calls

As we found from the above results (figure:7), the overhead time in intercepting a system call is about 0.01msec (approx). We also observed that the test program resulting error condition while it is violating the policy of limiting number of opened sockets. From these errors, we concluded that the system functions reliably depends on stated policies.

V. Conclusion

System call based IDS is found efficient as it works at kernel space and so avoids context switching that is found in intrusion detection systems operating at user level.However, this kind of IDS monitors each system call in isolation. It only checks the correctness of a system call considering it as an atomic operation. The correctness of complex computing scenarios emerging due to computations can not be verified by checking individual system calls only. Policies are found useful for abstraction of such complex scenario. The research on policy based approach for access control of complex systems is being currently pursued[14]. Policy based IDSs are proposed in [9],[13]. But, these systems have limitations like shallow use of policy[9] and execution of policies in user space[13]. In this work, we have proposed a hybrid method that combines both system call based IDS and policy based IDS techniques. And also, our approach overcomes the limitations observed in [9], [13]. In our scheme policies can be defined at higher level in *if-then-else* scheme that allows a policy to comprehend and specify a security scenario at higher level. We describe policies as combination of condition(s) and action(s) and implement it in C. For sandboxing a system resource, policies associated to its different states are executed in kernel mode. In an experiment for sandboxing network calls in Linux2.6, we have shown that on an average, overhead for the purpose is limited to 0.01msec for a system call. So, the proposed system is time efficient as it is running in kernel space instead of user space. The approach allows a high degree of flexibility for a security administrator to form and deploy context-specific policies on permissible usability of system resources. Currently, we are developing a tool for security administrator to generate executable C code from XML based policy specification and to plug these codes to Linux 2.6 kernel on-the-fly.

Acknowledgment

The work is supported by University Grants Commission,INDIA for the project titled ”Investigation on Model Based Intrusion Detection System for University Resources” as major research project.

References

[1] Anup K. Ghosh, Aaron Schwartzbard: A study in IDS using neural networks for anomaly and misuse detection. In proceedings of the 8th USENIX security symposium, Washington D.C. USA, August 23-26 (1999)
[2] Introduction to Intrusion Detection -ISCA Publications, Prepared by RebekoBag.
[3] A.G.Gonek, T.A.Corbi: The Dawning of the autonomic computing Era. IBM Systems Journal, Vol.42 No.1 (2003)
[4] ”IDS Today and Tomorrow”, SANS Reading room by Thomas Goeldenitz January 22, 2002
[5] Hal Jatz and Alfouso Valdes. The NIDES statistical component description and justification, Technical report, Computer Science Laboratory, SRI international, Menlo Park, California, USA, March 1994
[6] Debra Anderson, Teresa F. Lunt, Harold Janitz, Ann Tamorn and Alfonso Valdes: SAFEGUARD FINAL REPORT: Detecting Unusual Program Behavior using the NIDS statistical component. Technical report, Computer Science Laboratory, SRI international, Menlo Park, California, USA, Dec 1993
[7] Kathlecn A. Jackson. INTRUSION DETECTION SYSTEM Product review IBM International confidential document, IBM Research Division, Zurich Research Labs, April 1999
[8] VanPaxson Bro: A System for Detecting Network intruders in Real time. In the 7th USENIX Security Symposium, 1998
[9] Suresh N. Chari, Pau-Chen Cheng: BlueBox: A policy-driven Host-Based IDS, ACM Transactions on Information and System Security (TISSEC), Vol. 6, .2 (2003)
[10] Harne Debar, Marc Dacier, Mehdi Nassehi, and Andreas Wespi: Fixed vs. Variable-length Patterns for detecting suspicious process behavior, Research Report, No RZ 3012, IBM Research Division, Zurich Research Lab, April 1998
[11] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. LongStaff: A sense of self for UNIX process. In IEEE symposium on security and privacy, 1996
[12] Sihn-Hye Park, WonilKim, and Dong-KyooKim: Autonomic Protection System Using Adoptive Security Policy
[13] ”On Run-Time Enforcement of Policies” by Harshit Shah and R. K. Shyamasundar, Advances in Computer Science ASIAN 2007
[14] ”Infrastructural Support for Enforcing and Managing Distributed ApplicationLevel Policies by Tom Goovaerts,Bart De Win,Wouter Joosen, in Electronic Notes in Theoritical Computer Science 197(2008)31-43