On the Comparison of User Space and Kernel Space Traces in Identification of Software Anomalies

[1]Syed Shariyar Murtaza, [2]Afroza Sultana, [2]Abdelwahab Hamou-Lhadj, [3]Mario Couture

[1, 2] Software Behaviour Analysis Lab, Concordia University, Montreal, QC, Canada

[3]System of Systems Section, Software Analysis and Robustness Group, Defence Research and Development Canada, Valcartier, Québec, QC, Canada

[1]smurtaza@encs.concordia.ca, [2]{abdelw,af_sulta}@ece.concordia.ca, [3] mario.couture@drdc-rddc.gc.ca

—Abstract: Corrective software maintenance consumes 30-60% time of software maintenance activities. Automated failure reporting has been introduced to facilitate developers in debugging failures during corrective maintenance. However, reports of software with large user bases overwhelm developers in identification of the origins of faults, and in many cases it is not known whether reports of failures contain information about faults. Prior techniques employ different classification or anomaly detection algorithms on user space traces (e.g., function calls) or kernel space traces (e.g., system calls) to detect anomalies in software behaviour. Each algorithm and type of tracing (user space or kernel space) has its advantages and disadvantages. For example, user space tracing is useful in detailed analysis of anomalous (faulty) behaviour of a program whereas kernel space tracing is useful in identifying system intrusions, program intrusions, or malicious programs even if source program code is different. If one type of tracing or algorithm is infeasible to implement then it is important to know whether we can substitute another type of tracing and algorithm. In this paper, we compare user space and kernel space tracing by employing different types of classification algorithms on the traces of various programs. Our results show that kernel space tracing can be used to identify software anomalies with better accuracy than user space tracing. In fact, the majority of software anomalies (approximately 90%) in a software application can be best identified by using a classification algorithm on kernel space traces.

Keywords-Tracing, classification algorithms, system call traces, function call traces, failures, deployed software.

I. INTRODUCTION

Corrective maintenance, an activity that aims to rectify faults in a program, can soak up to 30-60% [1][2] of software maintenance time. Typically, maintainers collect data (such as execution traces) related to software failures in order to fix faults. Organizations of such applications as Firefox, NetBeans, Microsoft Visual Studio.NET and others often employ automated means to collect and report failure data. This is to reduce the cost of software maintenance, facilitate debugging, and improve software quality.

While such automation makes data collection and reporting practical from numerous sources, it can also overwhelm developers because manually interpreting such reports and identifying origins of faults is resource draining for large systems with huge user bases [3]. Moreover, it is not always known whether a trace collected from the field actually contains a fault or not. This is because the size of a trace buffer is limited, and a failure could manifest itself well after the fault occurs since many faults do not necessarily crash the system. Thus, a trace might not capture the faulty control flow (e.g., faulty function calls, exceptions, etc.).

The importance of classifying a field trace into a "passing" and "failing" trace can help in many software engineering activities such as:

- Software Debugging: Remote analyses and fault localization techniques (e.g., statistical debugging for fault isolation [4], locating faulty functions [5] and faulty paths [6] in field traces, and visualization of field traces [7]) need to know whether a trace has come from a successful or failing execution to facilitate debugging.
- Autonomic Computing: Self managing applications need to know when a system enters an abnormal state [8][9] so that they can automatically reconfigure the system to a normal state.
- Software Intrusion: Anomaly detection systems raise alerts when an execution trace is anomalous (faulty) by matching a trace with past normal traces (e.g., using hidden Markov models [10] and neural networks [11] on system calls to detect anomalies).

Prior techniques employ different machine learning algorithms on user space traces (e.g., function calls) or kernel space traces (e.g., system calls) to detect anomalous software executions. These technique include: (a) decision tree algorithms and Markov models to classify user space traces (e.g. statement, branch or function call traces) as passing or failing [12][8]); (b) pattern extraction algorithms to detect abnormal behaviour such that the user space trace collection for failures could be started at the right time [13]; and (c) Markov models [10], neural networks [11], support vector machines [14], decision trees [14] and k-nearest neighbor [14] algorithms use on the system calls to classify abnormal and normal software behavior from the perspective of software security.

Zahalka et al. [15] identified in their experiments with user space traces that the discriminating strength of failing and passing traces significantly varies from program to program. This means a technique that produces high accuracy on user space traces of one program might yield low accuracy on other programs. Also, each algorithm and type of tracing (e.g., user space or kernel space) has different traits, and what is suitable in one situation might not be suitable in another situation. For example, the user space tracing is useful in detailed analysis (e.g., fault localization) of anomalous (faulty) behaviour of a program, whereas the kernel space tracing is useful in identifying system intrusions, program intrusions, or

malicious programs even if the source program code is different. Similarly, another example is that the kernel space tracing can be used to trace all the applications in a system simultaneously with lesser overhead than the user space tracing, but the kernel space tracing misses control flow information that is not executed through kernel (e.g., function calls executed directly by CPU).

Thus, the literature lacks information on the comparison of kernel space and user space tracing in identification of anomalous software behaviour. This will be useful in understanding if one type of tracing is not feasible to implement, then how different would the results of another type of tracing be? Thus, the main research question of this paper is:

(Q1) Can kernel space tracing be used to classify pass fail traces of a program with the same accuracy as user-space tracing?

We find the answer of this research question by evaluating six classification algorithms on both user space and kernel space traces. The six classification algorithms are C4.5 decision tree, naïve Bayes classifier, neural network, Bayesian network, support vector machine, and hidden Markov model. We evaluate these algorithms by (a) training them on both passing and failing traces and (b) training them only on normal traces. This is because in some situations abnormal behaviour (traces) is not available for training; e.g., in biometric password hardening system which strengthens the login process when password is not type in a correct rhythm [16]. Irrespective of the training method, each type of algorithm has different characteristics; e.g., hidden Markov models are slower to train but they consider temporal relationship of attributes unlike other algorithms. In finding out the answer to our main research question, we identified a secondary novel research question on the comparison of classification algorithms in identification of normal and anomalous software behaviour:

(Q2) Can we substitute one classification algorithm with another without affecting the accuracy of classification normal and anomalous traces?

These questions are important because efficient debugging and anomaly detection systems can be built if we know that a particular type of tracing and algorithm perform better or similar to others. For example, if a particular algorithm and type of tracing can be used to classify passing-failing traces with high accuracy, then due to similar characteristics of traces that algorithm and tracing will presumably be also able to perform better for further analysis of traces, such as fault localization, finding origin of software intrusions, etc.

The rest of the paper is as follows: Section II describes related work; Section III explains the four UNIX utilities (i.e., Flex, Grep, Gzip, and Sed) that we used as subject programs; Section IV explains our approach with working examples from the subject program; Section V describes the evaluation criteria of our approach, Section VI articulates results; Section VII discusses threats to validity; and Section VIII concludes this paper with directions to future work.

II. Related Work

Prior empirical studies that classify normal and failing traces have shown that failing execution traces have unusual characteristics than normal execution traces (e.g., classifying normal failing software behaviour using user space traces [13][12][8] and detecting anomalous behaviour using kernel space traces [10][17][18][11][14]). These studies can be divided into two categories: techniques focusing on software maintenance, and techniques focusing on software intrusion.

A. Techniques Focusing on Software Maintenance

Elbaum et al. [13] experiment with three different anomaly detection methods on function call traces of a deployed system. Their objective is to anticipate the occurrence of a failure in a deployed system such that trace collection for the failure could be automatically started at the right time. Bowring et al. [12] and Haran et al. [8] develop techniques based on the Markov model [12] and the decision tree [8] to classify (statement, branch and function level) executions as being passing or failing. Jiang et al. [9] extract varied length n-grams from function call traces of normal behaviour, and build an automaton from the n-grams that represent the generalized state of the normal traces: they use this automaton to detect anomalous traces.

Zahalka et al. [15] determin the factors affecting the differences between passing and failing user space traces. Zahalka et al. [15] identify that the discriminating strength of failing and passing traces is significantly different from program to program. They [15] also identify that the effect of number of faults on the discriminating strength is less than the program itself.

Podgurski et al. [3] form clusters of execution traces of field failures based on common faulty source files. Podgurski et al. [3] first employ logistic regression to classify passing and failing traces, second they select relevant attributes from classification, and third they employ k-medoid clustering to cluster failures. Liu et al. [19] cluster failing runs of deployed systems according to a rank list of assertions (check points) in source code by using a statistical debugging technique. Statistical debugging [19], requires a collection of passing and failing traces and. Liu et al. assume that passing and failing traces are provided. Apart from clustering, there were researchers who use the C4.5 decision tree algorithm [5], statistical utility functions (HOLMES [6]), literal comparison of traces [20][21] to identify fault locations of field failures. These researchers also require a distinction between passing and failing executions for their techniques.

B. Techniques Focusing on Software Intrusion

Our work is related to the type of intrusion detection systems that focuse on detecting anomalous software behaviour by measuring the deviations in system calls of a system from that of normal behaviour of the same system [22]. They are called host based anomaly detection systems [22] and techniques focusing on them are described below.

Forrest et al.[23], Hofmeyr et al. [24] and Warrender et al. [18] extract sequences of system calls from traces of a system and compare them with the historical sequences of normal behaviour. In the case of a mismatched sequence they raise alerts for anomalous behaviour. This is called a sliding window technique.

Warrender et al. [18], Yeung and Ding [25] and Wang et al. [10] also train hidden Markov model (HMM) on system call traces and raise alerts when the probability of a system call in a sequence is below a certain threshold [18] or the probability of whole system call sequence is below a certain threshold [10]. Hoang et al. [26] propose a multiple layer detection approach by using the sliding window technique on the first layer and HMM on the second layer and combining their output using fuzzy inference engine to predict anomalous system call sequences.

Ghosh et al. [11] employ standard multilayer perceptron and Elman [27] recurrent neural network on system calls to detect

anomalous system calls in test data. Their results [11] show better accuracy with recurrent neural networks but at the expense of more time than the standard multilayer perceptron. Yuxin et al. [14] use support vector machines, decision trees, and the k-nearest neighbor algorithm to classify malicious software code (e.g., virus) and normal code. They first extract static system call sequences for a program (i.e., extract system calls without running a program) and then train the algorithms for classification. They identify that their static system-call based technique produce better results than dynamic system-call based techniques. Tandon [28] and Warrender et al. [18] use variations of association rules on system calls[28] and system calls with arguments [28] to identify anomalous rules of system calls.

C. Research Gap

Prior studies have used a variety of algorithms on system call traces [10][14][18][11][25][28][23] and user space traces (e.g., function calls) [8][9][12][13] to identify anomalous software executions. However, none of them compared user space and kernel space traces. This paper aims to fill this void. Comparison of user space and kernel space tracing is important to understand the problem: when it is not feasible to collect one type of tracing (e.g., user space tracing due to source instrumentation) then can other type of tracing (e.g., kernel space tracing) be used with the same efficacy to identify anomalies? This paper also helps in understanding a novel issue: can we substitute classifiers when identifying normal anomalous software behaviour? This is important because some classifiers are faster to train and some are slower to train.

III. SUBJECT PROGRAMS

In this section, we explain the subject programs used in our study and how we collected traces. We present the subject programs before explaining our approach (presented in the next section) because we will be drawing examples from the subject program in the next section. We used open source UNIX utilities [29] Flex, Grep, Gzip and Sed for our experiments, which are known commercial C language applications. The faults in these programs were hand seeded by Do et al. [29] by using a specific procedure to keep them realistic (described in their paper [29]). Do et al. [29] used several releases of every program to insert faults. The important steps of fault insertion procedure were: (a) identification of the changes in source code of different releases; (b) insertion of faults at the changes in the code by multiple programmers working independently; (c) insertion of faults associated with definition, redefinition, deletion, and change of values of variables; (d) insertion of faults associated with control flow, such as deletion of path, addition of new block of code, redefinition of execution condition, modification to external function-calls, etc.; (e) insertion of faults associated with memory, such as erroneous use of pointers, memory not allocated, etc.; and (f) merging of all the faults and removal of overlapping faults such that programs should compile.

Flex, Grep, Gzip, and Sed [29] are made available in several releases by Do et al. For our experiments, we randomly selected one release of every program. The release numbers of each program, used in our experiments, are shown in Table 1. Each subject program comes with a test suite containing many test cases, and source code of the program with a list of faults in a header file—these faults were not active. We compiled each program without activating faults and ran test cases on the faults free programs. Since no faults were activated, the output of the program for each test case considered normal and traces were collected as normal traces. To collect failing/anomalous traces, we activated all the faults provided with the programs by Do et al. , and ran test cases on the fault programs. If the output of a faulty program on a particular test case differs from its fault free program then we collected a trace as a failing trace. Mainly two main types of faults were resulted when test cases were run: (a) crashing faults (e.g., segmentation faults); (b) non crashing faults (e.g., logical errors). The details of each of the programs with number of faults, number of test cases, and number of passing and failing traces that we collected are shown in Table 1. In Table 1, in some cases the number of passing traces is not equal to the total test cases because some of the input files could not be run.

Table 1: Characteristics of the subject programs (UNIX utilities).
LOC excludes blank lines and comments

Releases used: Flex 2.5.1; Grep 2.4; Gzip 1.1.2; Sed 4.0.7.						
Prog.	LOC	# Functions	# Faults	# Test Cases	# Passing Traces	#Failed Traces
Flex	9724	167	20	567	566	545
Grep	9041	149	18	809	799	710
Gzip	4032	88	16	214	214	204
Sed	4735	115	6	370	366	166

We used LTTng [30] to collect both user space (function call traces—see Figure 2 for an example) and kernel-space traces (see Figure 3 for an example). All of our experiments were performed on Ubuntu 11.04. Kernel space tracing was specific to Linux operating system as obtained using the LTTng tool, and user space tracing was independent of the operating system. For user space tracing we used a tool called Etrace[1] [31] to collect function call traces.

IV. APPROACH

In order to compare kernel and user space traces using different classification algorithms we proceeded with the approach shown in Figure 1. The steps of our approach were:

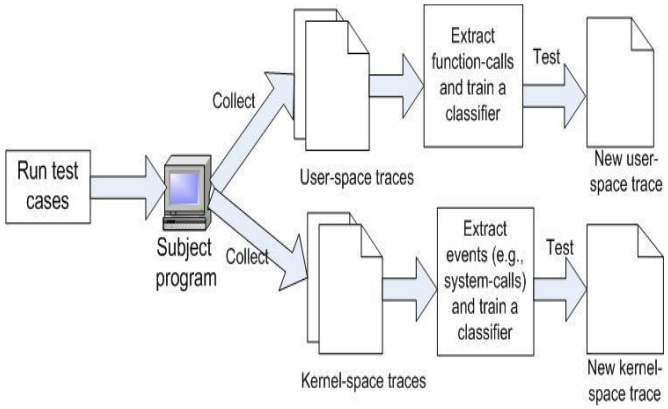

Figure 1: Steps of our approach.

1) First, we collect user space and kernel space traces for a program by running test cases (see Section III). An example of user space traces —i.e., function calls traces—is shown in Figure 2. An example of kernel space traces collected using LTTng [30] is shown in Figure 3. LTTng allows us to

[1] Etrace has a bug which prevents it from capturing traces of the segmentation faults. We fixed it to collect such traces.

collect system calls, IRQs, trap, memory management, softIRQs, scheduling, network management, file system management and other events.

```
23        fooPrevious exit
24         foo1 entry
25         | foo2 entry
26         | | foo3 entry
27         | | | foo4 entry
28         | | | foo4 exit
29         | | foo3 exit
30         | foo2 exit
31         foo1 exit
32        fooLater entry
```

Function entry point shows when control enters a function and function exit show when control exits a function.

Figure 2: Function call trace at user space level.

```
channel:kernel; event:syscall_entry  process:./gzip.exe;  state:
SYSCALL;  markers:ip = 0x22cbad, syscall_id = 6 [sys_close+0x0/
0x100]; pid:2842

channel:fs; event:close  process:./gzip.exe; state:SYSCALL;
markers:fd = 3;  pid:2842

channel:kernel; event:syscall_exit  process:./gzip.exe;
state:USER_MODE;  markers:ret = 0; pid:2842

channel:kernel; event:softirq_exit process:./gzip.exe;
state:USER_MODE;  markers:softirq_id = 4 [blk_done_softirq+0x0/
0x70]

channel:kernel; event:irq_exit process:./gzip.exe; state:USER_MODE;
markers:handled = 1; pid:2842

channel:kernel; event:page_fault_entry process:./gzip.exe;
state:TRAP;  markers:ip = 0x8049aa9, address = 0x805d000, trap_id
= 14, write_access = 1; pid:2842
```

Channels group events (i.e., handlers and methods in Linux OS code) of a particular type; other variables are specific to events.

Figure 3: Trace containing system wide events associated to gzipe.exe at kernel-space level.

2) Secondly, for user space traces, we extract function calls and their likelihood of occurrences in a trace and train classification algorithms on them. Similarly, for kernel-space traces, we extract all the occurring events related to the program under study (e.g., system calls, IRQs, etc.) and measure their likelihood of occurrence in the traces.

3) Third, we train the classification algorithms on the extracted data. In our experiments we have used six classification algorithms to compare the accuracy of classification of user space and kernel space traces. These algorithms are: C4.5 decision tree, Naïve Bayes, Bayesian network, multilayer perceptron (artificial neural network), support vector machine and hidden Markov model. We then use the trained algorithms to classify traces present in a test set. These classification algorithms are well known algorithms and their details can be found in standard text [32]; we do not provide their details to save space. Also, recall from Section I, we trained the classification algorithms from two different perspectives: (a) training and testing on both normal and anomalous traces; and (b) training on only normal traces and testing on both types of traces.

In Section IV.A, we explain in detail our procedure for training the classification algorithms on user space traces. Similarly, in Section IV.B, we explain our approach for training the classification algorithms on kernel space traces. Both Section IV.A and Section IV.B describe classification form the perspective of training on both normal and anomalous traces. In Section IV.C, we describe how classifiers were trained only on normal traces and tested on both types of traces.

A. User Space Tracing

At user space level we collected function call traces, see Figure 2, because prior researchers [13][12][8][15] [5] have mostly used function call traces from the field to classify passing and failing traces. Also, function call traces are the commonly collected traces from deployed software systems as they are easier to collect and incur less overhead than finer grained traces, such as statements.

After collecting passing and failing user space traces, we transformed them into a form on which the classification algorithms could be trained. This is shown in Figure 4. In Figure 4, each row represents a trace, and each column shows the name of a function. The last column in each row denotes whether a trace is a passing trace or a failing trace. Each cell represents the chances of occurrence of a function in a trace and it is measured by the following equation:

$$\begin{bmatrix} Likelihood \\ of\ a \\ function\ f_k \\ in\ a\ trace \end{bmatrix} = \frac{\#\ occurences\ of\ function\ f_k\ in\ a\ trace}{Total\ occurrences\ of\ all\ functions\ \ in\ a\ trace} * 100$$

Equation 1: Equation to measure the chances of a function's occurrence in a trace.

The Equation 1 simply measures the frequency of occurrences of a function 'f' in a trace and divides it by the total number of function calls in the trace.

In Figure 4 we have used only function "exit" events for training; the term "exit" is not shown in Figure 4. However, Figure 2 shows function "entry" and function "exit" events. The reason for using only "exit" events lies in our earlier experiments [5], where we have found that when a classifier is trained on function "entry or exit" and on both function "entry and exit" then there is no significant difference in the accuracy of the classification. This discovery helps in reducing the size and overhead of a trace to half, as the function "entry" or function "exit" events can be removed from traces. Thus, in our experiments in this paper we only use function "exit" events.

	Functions							
Trace id	longest_match	pqdownheap	send_bits	ct_tally	.....	.....	bi_reverse	Type
FT210	26.6	1.13	48.8	21.8			1.39	Fail
FT84	27.1	1.11	48.8	21.7			1.38	Fail
PT136	18.3	1.20	54.9	23.8			1.49	Pass
.....								
.....								.....
PT77	24.4	1.12	50.4	22.4			1.39	Pass

Figure 4: Function calls and their chances of occurrences in a passing and failing traces of the Gzip program.

Once traces were transformed into a form shown in Figure 4, we then trained the classification algorithms on them. We actually first divided the original dataset into two parts: training (approximately 65%) and testing (approximately 35%). The classification algorithms were trained on this 65% of the original training data and tested on the remaining 35% traces. An example of the C4.5 decision tree (a classification algorithm) trained on the transformed traces of Gzip is shown in Figure 5. This tree was obtained by applying the J48 algorithm in the data mining tool Weka [32] which was an implementation of the C4.5 decision tree algorithm.

```
fill_window <= 0.004698
|   clear_bufs <= 3.448276
|   |   do_stat <= 0.005896: pass
|   |   do_stat > 0.005896
|   |   |   getopt_long <= 14.285714: pass
|   |   |   getopt_long > 14.285714: fail
|   clear_bufs > 3.448276
|   |   treat_stdin <= 2.439024: fail
|   |   treat_stdin > 2.439024: pass
fill_window > 0.004698: fail
```

Figure 5: C4.5 decision tree on function call traces of the Gzip program.

Each line in Figure 5 contains a function name, its likelihood, and a name of passing and failing trace after a colon sign if any. The discovery of a faulty function was done by traversing this trained tree (like If-then-else statements) according to the likelihood values of functions. For example the decision tree of Figure 5 shows that if in a trace, the likelihood value of a function "fill_window" is less than or equal to "0.004698" and the likelihood value of "clear_bufs" is less than or equal to "3.448", and the value of "do_stat" is less than equal to "0.005896" then the trace is a passing trace.

After building the classifier like the C4.5 decision tree, we classified every trace in a test set as a passing or failing trace, and recorded the accuracy of classification. Finally, we repeated the above process two more times (three in all) every time with a different 35% test set and 65% train set. The accuracy on the test set was then averaged. This is called three fold cross validation. Similarly, we repeated this procedure for all other classification algorithms by using three fold cross validation and the results are discussed in Section VI.

In Figure 4, we showed that we extracted only single function calls and their likelihood of occurrences to train a classifier. This does not preserve the temporal order of function calls as they occur in a trace. There are two possible solutions to keep the temporal order of sequences in the model of a classifier: (a) use hidden Markov model (HMM); and (b) extract temporal sequences of function calls and train any classifier. First method is the use of HMM as a classifier. HMM preserves the temporal order of function calls, as they occur in a trace, in its model. Our results on HMM are shown in Section VI.

Another method is to train every classifier on the patterns of function calls. For example, consider an example of a pattern of length three function calls: "adddef→elemdef→waitcont". This pattern is read as "adddef" precedes "elemdef" and "elemdef" precedes "waitcont" in traces. If all such function-call patterns [5] of different lengths are extracted from the failed traces and used with the classifier to identify faulty functions, then our earlier experiments show that results are not better than the use of single function calls with the classifier, such as decision tree [5]. Thus, we considered using only single function calls for classification in this paper.

B. Kernel Space Tracing

Table 2: Extracted attributes for channels and events and example of elements.

Channel	Event	Attribute	Example Element*
kernel	syscall_entry	syscall_id	kernel:syscall_entry:10
kernel	syscall_exit	*ret	kernel:syscall_exit:10_zero
kernel	irq_entry	irq_id	kernel:irq_entry:20
kernel	softirq_entry	softirq_id	kernel:softirq_entry:1
kernel	softirq_raise	softirq_id	kernel_softirq_raise:1
task_state	process_state	*status	Task_state:process_state:0
*fs	*all	*fd	fs:open:3
*block	*all	*rw, not_uptodate, error	block:rq_insert_fs:1

"fs" stands for file system,"block" stands for block IO devices,*"all" means all the events and channels, *"ret" stands for return value, * "status" is for state of the process in system,*"fd" is file descriptor id, *"rw" stands for read/write. *In element a number at the end indicates a value of attribute.

In a similar manner to user space traces, we also collected kernel space traces. An example of a kernel space trace is already shown in Figure 3. Recall from Section III that we used LTTng to collect kernel space traces. Kernel space traces are specific to an operating system as they record functions of operating system's source code. An LTTng trace actually groups different events (e.g., functions, handlers, etc.) executed by the Linux operating system using a channel name. For example, in Figure 3, events related to a file system are grouped under the channel "fs", and events related to core kernel functions (e.g., system calls, page_fault_entry, soft_irq) are grouped under the channel "kernel".

An LTTng trace also groups several associated attributes with an event. For example: (a) whenever control enters a particular function (system call) in operating system's code then the system call "name" and "id" is recorded (see the event "syscall_entry in Figure 3); (b) when a control exits a function (system-call) in operating system's code then its return value is recorded with the "exit" event (see the event "syscall_exit" in Figure 3); and (c) whenever there is an event of a page fault then the specific trap which caused the page fault along with the read-write status of the page fault is recorded (see the event "page_fault_entry" in Figure 3). In order to train a classifier on such kernel-space level traces, we followed the following steps:

- First, we extracted only those channels, events and attributes that were associated with the subject program under investigation. For example, in Figure 3, all the events and channels are associated with the process "gzip.exe", and we filtered out all other events.
- Second, we extracted the channel name (e.g., kernel), the event name (e.g., syscall_entry) and the relevant attributes

if found any (e.g., syscall_id=6). We combined these to make one single "element". This is shown in Table 2. We extracted all the channels and events associated to a program. We also extracted attributes and for some channels and events that are relevant for classification. For example, for the event "syscall_entry" we extracted "syscall_id" since it provides the unique id of a system call. Similarly, for "syscall_exit", we extracted the return ("ret") value of a system-call. Moreover, each system call returns a different value, such as error codes, hardware specific locations (e.g., memory address, file location, etc.). We categorized the return values into three types: positive (for normal return values), negative (for errors), and zero (for normal or no value). Table 2 shows the list of attributes that we extracted for different channels and events. For all other cases, we only extracted channels and events (e.g., for events "irq_exit", "softirq_exit" in channel "kernel" we only extracted their names as there was no relevant attribute associated to them). We discarded all the page faults[2] and memory management events because they would vary from machine to machine. Similarly, we discarded attributes such as process-id, memory address, and thread-id as they were dependent on a machine.

- Third, we transformed the extracted elements into a similar form as in Figure 4. For each element we measured its likelihood of occurrences in a trace by using Equation 2, which is similar to Equation 1 (see Section IV.A).

$$\begin{bmatrix} Likelihood\ of \\ an\ element\ \ E_k \\ in\ a\ \ trace \end{bmatrix} = \frac{\#\ \text{occurrences of an elemenet } E_k \text{ in a trace}}{\text{Total occurences of all elements in a trace}} * 100$$

Equation 2: Equation to measure the chances of an element's occurrence in a trace.

Finally, in the same manner to user space traces, we trained the six classifiers (including hidden Markov model) on the transformed traces and evaluated them using three fold cross validation. The results are shown in Section VI.

C. Training on only Normal Traces

In identification of anomalous software behaviour, often there are situations when only normal traces are available for training, for example, in software intrusion detection and in self-healing systems (autonomic computing). Such type of classification is called anomaly detection or outlier detection. For outlier detection, we have also trained classifiers only on normal traces by using one class classifier [16] available in Weka [32]. The one-class classifier works by transforming data of one class (normal traces) into two-class data (normal-anomalous traces) for a two (or more) class classifier (e.g., the C4.5 decision tree). The one-class classifier works by taking data of a target class–the class on which to train a classifier (normal traces in our case)—estimates the probability density function of the target class, and generates artificial data using the density function for an outlier class—in our case outlier class corresponds to anomalous traces. This provides a two-class (normal and anomalous trace) data to the two-class classifier (e.g., C4.5) for training without the knowledge of data of actual anomalous traces. Once the two-class classifier is trained on such data then it is tested on the test set that contain all the anomalous traces and a proportion of normal traces that are not used for training. In this case we also used three-fold cross validation in which every time a different set of normal traces were used for training but the anomalous traces were always kept in the test set.

We were able to apply one-class classifier on the C4.5 decision tree, naïve Bayes, Bayesian network, multilayer perceptron and support vector machine. In the case of HMM, one-class classifier did not work because of the sequential data format. Our results do not contain results of HMM for outlier detection. However, it should be noted that the main purpose of this paper is not the comparison of classification algorithms but the comparison of user space and kernel space traces, hence excluding HMM does not affect the main objective.

V. Evaluation Criteria

In this section we explain different parameter settings we used in Weka [32] for the six classification algorithms and what measures we used to evaluate their results.

The Bayesian belief network (BBN) was implemented by using the K2 algorithm in Weka [32] to learn the network structure from the data. We started the K2 algorithm by initializing the class node as a parent node and all other attributes as a child node. We set the maximum number of parents to two, the ordering of nodes to random, and the calculated conditional probabilities by using simple estimator in Weka. The goodness of fit of the network structure was measured by the Bayes score [32] .

The C4.5 decision tree [32] was implemented by using Weka J48 [32] algorithm. To avoid over-fitting the tree, we used sub tree raising, 25% confidence interval to prune the tree, and used MDL correction for finding splits on numeric attributes leaves.

The naïve Bayes (NB) algorithm estimated prior and conditional probabilities for each attribute by using Gaussian distribution, and Bayes rule was used to measure probabilities of test traces [32].

We implemented the artificial neural network (ANN) by using back propagation feed forward multilayer perceptron (MP) in Weka [32]. We selected only two hidden layers of neurons as they often yield optimum results [32], the weights are update at a learning rate of 0.3 and a momentum of 0.2, and the number of epochs were 500.

The support vector machine (SVM) was implemented using sequential minimal optimization algorithm (SMO) in Weka [32]. We also used polynomial kernel in SMO to train SVM, and applied a filter "standardized training data" for data preprocessing.

The hidden Markov model (HMM) was implemented by using a third party plugin[3] available for Weka. We used 6 states to train HMM, built an Ergodic model in which every state was connected to every other state, and the state transition probabilities were initialized by k-means clustering.

After applying the classification algorithms, the next task was how to compare their results correctly. In our experiments, we had only two decisions to make; that is, classify a trace in a test set as normal or anomalous. In machine learning [32], usually the performance of the classifiers is evaluated using true positives (TP) and false positives (FP). TP occurs when a normal trace in a test set is classified as normal. FP occurs when an anomalous

[2] A page fault occurs when there is a request to swap a memory page from disk to OS.

[3] http://www.doc.gold.ac.uk/~mas02mg/software/hmmweka/

traces in a test set is classified as normal. We measured TP rate and FP rate by following two equations:

$$TP\ rate = \frac{Number\ of\ normal\ traces\ detected\ as\ normal}{Total\ number\ of\ normal\ traces}$$

Equation 3: True positive rate.

$$FP\ rate = \frac{Number\ of\ anomalous\ traces\ detected\ as\ normal}{Total\ number\ of\ anomalous\ traces}$$

Equation 4: False positive rate.

Another important characteristic to evaluate the results of machine learning algorithms is the Receiver Operating Characteristics (ROC) curve. The ROC curve depicts the performance of a classifier without regard to class distribution or error cost [32] by plotting the TP rate against the FP rate. However, often there is no clear better ROC curves of classifiers in the entire range of FP rate and TP rate. In such situations, the area under the curve of ROC (simply AUC) provides a single number summary for the performance of a classifier [33]. Ling et al. [33] also formally proved that AUC is a better measure than accuracy, and the classifiers which produce better AUC also yield better accuracy. AUC is measured as [33]:

$$AUC = \frac{Sp - N(N+1)/2}{N * A}$$

Where N is the number of normal traces in the test set, A is the number of anomalous traces in the test set, Sp = r_i and r_i is the rank of i_{th} positive example.

Equation 5: AUC: Area Under the ROC Curve.

In short, the larger the area under the curve (AUC), the better would be the classifier's model [32]. The AUC is actually the measure of the quality of ranking and can be interpreted as the probability that the classifier ranks a randomly chosen positive instance above a randomly chosen negative instance. In our experiments TP rate, FP rate and AUC were measured using Weka.

kernel space traces by training the classifiers on both normal and anomalous traces. In Section VI.B, we show the results by training classifiers only on the normal traces. Finally, Section VI.C summarizes the results.

A. Results of Classifiers When Trained on Both Normal and Anomalous Traces

In Table 4, we show the TP rate, FP rate and AUC of six classification algorithms (C4.5 decision tree, Naïve Bayes, Bayesian network, multilayer perceptron, support vector machine, and hidden Markov model) on the function call level traces of the Flex, Grep, Gzip and Sed programs. The results are obtained by using three fold cross validation (see Section IV.A).

For example, when the C4.5 decision tree was applied on the user space traces of the program Flex, the TP rate was 0.924, FP rate was 0.099 and AUC was 0.925. This means TP rate, FP rate and AUC were 92.4%, 9.9% and 92.5% respectively when C4.5 was used on Flex. These results were obtained when the C4.5 decision tree was trained on two-third (2/3) of traces of the Flex program and tested on the remaining one-third (1/3) of traces of the Flex program: the process was repeated three times with different proportion of traces for training and testing, and finally the results were averaged. Similarly the results can be interpreted for other algorithms and programs in Table 4. A special case occurred in the case of HMM when most of the traces, from 54% to 88%, remained unclassified for the subject programs. This resulted in low false positive rate as Weka [32] counted FP and TP of those traces which were assigned a normal or anomalous labels; unclassified traces were not included by Weka. However, this resulted into lower accuracy and lower AUC values.

Table 3: Results of the classification algorithms on user space (function call) traces of the subject programs.

Where NB = Naïve Bayes; BBN= Bayesian Belief network; MP= Multilayer Perceptron; SVM=Support Vector Machine; HMM= Hidden Markov Model

	C4.5			NB			BBN			MP			SVM			HMM		
Prog.	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC
Flex	0.924	0.099	0.925	0.159	0.053	0.609	0.371	0.145	0.675	0.981	0.804	0.646	0.721	0.323	0.699	0.706	0	0.416
Grep	0.96	0.044	0.992	0.921	0.227	0.919	0.96	0.044	0.992	0.95	0.113	0.972	0.935	0.075	0.93	0.122	0	0.428
Gzip	0.967	0.02	0.974	0.698	0.594	0.663	0.939	0.035	0.989	0.528	0.144	0.819	0.962	0.015	0.974	0.727	0.258	0.471
Sed	0.849	0.693	0.714	0.321	0.151	0.630	0.61	0.331	0.694	0.819	0.711	0.72	0.808	0.759	0.524	0.964	0.917	0.651

In a similar manner to the user space traces, we have also evaluated the classifiers on the kernel space traces. The results are shown in Table 3. In Table 3, the results are obtained by training the classifiers on extracted "elements" from LTTng traces as mentioned in Section IV.B. In the case of HMM, similar to user space traces, some of the traces in the test set remained unclassified (from 0.4% to 2%).

Table 4: Results of the classification algorithms on kernel space traces of the subject programs.

Where NB = Naïve Bayes; BBN= Bayesian Belief network; MP= Multilayer Perceptron; SVM=Support Vector Machine; HMM= Hidden Markov Model

	C4.5			NB			BBN			MP			SVM			HMM		
Prog.	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC
Flex	1.00	0.002	0.998	1.00	0.002	0.999	0.993	0.004	0.999	0.998	0.002	1.00	0.998	0.002	0.998	1.00	0.002	0.996
Grep	0.977	0.027	0.981	1.00	0.114	0.942	0.994	0.082	0.996	0.961	0.075	0.992	0.955	0.037	0.959	0.87	0.008	0.913
Gzip	0.953	0.034	0.953	0.682	0.338	0.717	0.883	0.059	0.950	0.963	0.049	0.972	0.963	0.039	0.962	0.256	0.079	0.528
Sed	0.918	0.337	0.813	0.492	0.133	0.727	0.91	0.313	0.872	0.943	0.277	0.863	0.929	0.289	0.82	0.424	0.303	0.548

VI. RESULTS

In this section, we show the results for identification of anomalous and normal software behaviour of Gzip, Grep, Flex and Sed programs using user space and kernel space traces. (See Section III for a refresher on the subject programs and trace collection.) In Section VI.A, we show the results on user space and

It can be observed both from Table 4 and Table 3 that the results obtained using kernel space traces are better than user space traces. In order to ascertain this, we performed a Wilcoxon singed rank test [34] between AUC values of six classifiers on four subject programs of kernel space and user space traces (i.e., 24 observations for both type of traces). We chose AUC values because in the case of HMM, TP and FP values were not

completely representative of all the traces in the test set. We chose the Wilcoxon signed rank test because the dataset was small and it was not known if the data belonged to a normal distribution. We state null hypothesis as "there is no significant difference between the AUC values obtained for user space traces and kernel space traces at the significance level (alpha) 0.05."

A Wilcoxon [34] signed rank test on 24 AUC observations of classifiers between kernel space and user space tracing resulted into Z=3.371 and two sided p=0.001 < 0.05. This means the null hypothesis is rejected as p < 0.05 and identification of normal and anomalous behaviour can be done accurately with kernel space tracing than user space tracing. Kernel space tracing is better when classifiers are trained on both normal and anomalous traces.

We also measured the standardize effect size of AUC between kernel space tracing and user space tracing. Nakagawa [35] mentioned that the advantage of the use of standardized effect size is that they are comparable across different studies even with different sample sizes [35]. In our case, we calculated the effect size using the Cohen's d measure [36] and the effect size is 0.866 between kernel space tracing and user space tracing. It can be interpreted as the average AUC values obtained using kernel space tracing will be 0.866 standard deviations above than the average AUC values of user space tracing. In terms of percentile standing [36], the results are interpreted as the average result obtained using the kernel space tracing would be better than more than 79% results of user space tracing when identifying normal-anomalous software behaviour using classification algorithms.

In a similar manner to the comparison of user space and kernel space tracing we also compared classification algorithms using the Wilcoxon [34] signed rank test. We selected the AUC values of both kernel space and user space tracing for every classifier. This resulted into eight observations for each classifier. The p values obtained using Wilcoxon signed rank test for every pair of algorithms are shown in Table 5.

Table 5: P values obtained using Wilcoxon signed rank test for pair wise comparison of classifiers.

	HMM	SVM	MP	BBN	NB
C4.5	0.012	0.116	0.779	1.00	0.017
NB	0.025	0.123	0.012	0.018	
BBN	0.012	0.093	0.401		
MP	0.012	0.484			
SVM	0.036				

In statistics, if more than one comparison is performed than statistical significance is often measured using Bonferroni correction. In Bonferroni correction, the significance level alpha is divided by n (alpha/n) then p value is tested at that significance level. The reason is that if 'n' tests are performed then there is a random chance that at least one of them out of 'n' will be significant. In our case 'n' is five as there are five comparisons, which means the new alpha level is 0.01. At this alpha level no significant difference exists between the AUC values of all the classifiers in Table 5 , as p > 0.01.

However, Nakagawa [35] argued that using Bonferroni correction reduces statistical power and increases the chances of Type II error (false negative) to unacceptable level. Therefore they [35] suggested that the emphasis should be placed more on (standardized) effect sizes and practical significance related to the field of study. We can also observe from Table 5 (and from Table 3 and Table 4 too) that HMM did not perform better than other algorithms at alpha=0.05. Thus, we measured the effect sizes between all the classifiers and found out that C4.5 and Bayesian belief network (BBN) yield closer results and C4.5 results are better than other classifiers. Due to the lack of space we have avoided showing effect sizes between all the algorithms; however, the effect size between C4.5 and BBN is 0.18, C4.5 and NB is 1.09, C4.5 and MP is 0.38, C4.5 and SVM is 0.43, and C4.5 and HMM is 1.74.

Thus, based on significance test none of the classifiers performed better than each other, but the measurement of effect size reveals that C4.5 should be preferred over other classifiers.

B. Results of Classifiers When Trained on Normal Traces

Recall from Section IV.C, we used one-class classifier [16] to train all the two-class classifiers we studied (except HMM) on normal traces. The test sets contained both normal and anomalous traces. The TP rate, FP rate and AUC for one-class classification are shown in Table 6 for user space traces and in Table 7 for kernel spaces traces. These results are also obtained using three fold cross validation. A well known problem in one-class classification (called anomaly detection) is high rate of false positives. This can also be observed in both Table 6 and Table 7. If you take a close look then the results obtained using kernel space traces are better than user space traces, and multilayer perceptron (artificial neural network) outperforms other algorithms as it has a high AUC than other classifiers—in some cases close to 1.00.

We again conducted a Wilcxon signed rank test on AUC values for both kernel and user space traces at alpha =0.05. A Wilcoxon sign test with 24 observations resulted into Z= 2.073 and (two sided) p=0.03 < 0.05. This implies there is a significant difference between the results obtained using kernel space traces and user space traces when classifiers were trained on only normal class—kernel space tracing yield better results with an effect size [36] of 0.51 (69% in percentile standing) over user space traces.

A Wilcoxon signed rank test, in a similar manner to Section VI.A with Bonferroni correction, resulted into no significant difference among classifiers. Due to lack of space we omitted the details of the tests. However the effect size reveals us that multilayer perceptron fares better than other classifiers when trained only on normal traces. The standardized effect size using Cohen's d measure [36] between multi layer perceptron (MP) and C4.5 is 0.557, MP and NB is 0.57, MP and BBN is 1.18, MP and SVM is 1.17. Thus based on the effect size we can say that multi layer perceptron is better than other classification algorithms studied in this section; however, again based on the significance test that there is no significant difference among classifiers in general.

C. Summary of Results

In this paper we raise two research questions (Q1) and (Q2) (see Section I) and our results answer them as:

- Kernel space tracing can detect anomalous (failing) software behaviour better than user space tracing. The average result obtained using the kernel space tracing would be better than more than 69%-79% results of user space tracing (see Section VI.A and VI.B) when identifying anomalous software behaviour. This answers (Q1). Moreover our data shows that, after selecting relevant information for

evaluation, the file sizes of kernel space traces of a program were 40-60% smaller than user space traces of the same program. Similarly the training time of different algorithms for kernel space tracing was 30-70% lesser than user space tracing. Thus kernel space tracing is efficient and accurate.

Table 6: Results of the classification algorithms when trained on normal user space traces only.

	C4.5			NB			BBN			MP (ANN)			SVM		
Prog.	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC
Flex	0.996	0.996	0.500	0.917	0.765	0.577	0.998	0.998	0.500	0.924	0.767	0.578	1.000	1.000	0.500
Grep	1.000	0.999	0.501	0.611	0.496	0.584	1.000	1.000	0.500	0.912	0.635	0.751	1.000	1.000	0.500
Gzip	0.976	0.995	0.491	0.444	0.418	0.500	1.000	1.000	0.500	0.877	0.842	0.515	1.000	1.000	0.500
Sed	0.997	1.000	0.499	0.621	0.608	0.515	0.997	1.000	0.499	0.896	0.886	0.509	1.000	1.000	0.500

Table 7: Results of the classification algorithms when trained on normal kernel space traces only.

	C4.5			NB			BBN			MP (ANN)			SVM		
Prog.	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC	TP	FP	AUC
Flex	1.000	0.002	0.999	0.954	0.332	0.811	1.000	1.000	0.500	0.926	0.000	0.972	1.000	1.000	0.500
Grep	1.000	1.000	0.500	1.000	0.807	0.597	1.000	1.000	0.500	0.882	0.110	0.953	1.000	1.000	0.500
Gzip	1.000	1.000	0.496	1.000	1.000	0.497	1.000	1.000	0.491	1.000	1.000	0.510	1.000	1.000	0.505
Sed	0.000	0.000	0.500	0.932	0.958	0.516	1.000	1.000	0.500	0.863	0.837	0.540	1.000	1.000	0.500

Where NB = Naïve Bayes; BBN= Bayesian Belief network; MP= Multilayer Perceptron; SVM=Support Vector Machine; HMM= Hidden Markov Model

- According to statistical significance test, six classification algorithms yield same results and any one can be substituted with another (see Section VI.A and VI.B) when identifying anomalous software behaviour. However, in terms of effect size, the C4.5 decision tree produces better results when normal-anomalous traces are available for training and the multilayer perceptron produces better results when only normal traces are available for training. This answers (Q2).

VII. Threats to Validity

In this section we describe threats to validity of our experiments according to four categories: conclusion validity, internal validity, construct validity, and external validity[34].

A threat to the validity of our conclusions exists because we use the traces of multiple faults for classification, rather than the traces of single faults. Accuracy could be different between traces of single faults and passing traces. However, this threat is mitigated by evidence in the literature that variability of execution profiles increase with multiple faults which actually decreases the discriminating strength of passing-failing traces[15]. This means with traces of single faults the accuracy of classification will be higher.

A threat to internal validity exists in the collection of traces and implementation of programs. This is because we automated this procedure by writing shell scripts and Java code. We have minimized this threat by manually investigating the outputs.

A threat to construct validity exists in how we measured normal behaviour and anomalous behaviour. We collected passing traces when there were no faults in a program and failing traces when the programs were seeded with faults. Another approach is to collect passing traces from the program containing faults when test cases did not fail and failing traces from the same faulty program when test cases failed. This threat is mitigated by the fact that in the later approach different test cases fail and pass on a program resulting in discriminating execution flow, and hence will result in higher accuracy than the former approach. In the former approach same test cases pass and fail and the discriminating strength could be less in some situations.

A threat to external validity exists in generalizing the results of this study as we have only experimented on medium size commercial programs, and evaluation on large industrial applications is yet to be done.

VIII. Conclusion and Future Work

Time spent in corrective maintenance (30-60%)[1][2] often exceeds the time spent in other activities of software maintenance. Prior researchers [3][13][12][8][19][6] [5] have proposed variety of techniques using different type of algorithms on user space tracing (e.g., function calls) to reduce this time. Prior researchers focusing on software security [10][17][18][11][14] have also used different machine learning algorithms on kernel space tracing to classify normal-anomalous software traces—these techniques can be used to reduce the time spent in corrective maintenance. However, a comparison of user space and kernel space tracing does not exist in the literature. The primary research question of this paper is: (1) Can kernel space tracing be used to classify pass-fail traces of a program with the same accuracy as user space tracing? We evaluated six different classifiers on kernel space and user space traces to find out that kernel space tracing yields better accuracy than user space tracing (see Section VI). In fact the average result obtained using the kernel space tracing would be better than more than 69%-79% results of user space tracing (see Section VI.A and VI.B). We also identified a secondary research question during our experiments: (2) Can we substitute one classification algorithm with another without affecting the accuracy of classification of normal-anomalous traces? Our results show that (see Section VI) no classifier yields better results than other classifiers according to significance test; however, according to effect size, the C4.5 algorithm yields better results than other algorithms in two-class classification (see Section VI.A) and the multilayer perceptron produces better results (see Section VI.B) than other algorithms in one-class classification.

Our findings are limited to evaluation on medium size programs. In future, we would like to extend the scope of these findings to large industrial scale applications.

IX. References

[1] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt, "Determining the Distribution of Maintenance Categories: Survey versus Measurement," *Journal of Empirical Soft. Engg.*, vol. 8, no. 4, pp. 351-365, Dec. 2003.

[2] M. G. Lee and T. L. Jefferson, "An Empirical Study of Software Maintenance of a Web-based Java Application," in *Proc. of Int'l Conf. on Soft. Maint. (ICSM)*, B2005, pp. 571-576.

[3] A. Podgurski et al., "Automated Support for Classifying

Software Failure Reports," in *Proc. Intl. Conf. on Software Eng.*, Portland, US, May 2003, pp. 465-475.

[4] M. Naik, A. X. Zheng, A. Aiken, M. I. Jordan B. Liblit, "Scalable statistical bug isolation," in *Proc. of Conf. on Programming Language Design and Implementation*, Chicago, USA, June 2005, pp. 15-26.

[5] S. S. Murtaza, M. Gittens, and Z., Madhavji, N. H. Li, "F007: Finding Rediscovered Faults from the Field using Function-level Failed Traces of Software in the Field," in *Proc. of CASCON 2010*, Toronto, Canada, Oct. 2010, pp. 57-71.

[6] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K Vaswani, "HOLMES: Effective Statistical Debugging via Efficient Path Profiling," in *Proc. of 31st Intl. Conf. on Soft. Eng.*, Vancouver, Canada, May 2009, pp. 34-44.

[7] J. A. Jones, M. J. Harrold A. Orso, "Visualization of program-execution data for deployed software," in *Proc. of the ACM symposium on Soft. Visualization*, San Diego, USA, June 2003, pp. 67-76.

[8] M. Haran et al., "Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks," *IEEE Trans. on Soft. Engg.*, vol. 33, no. 5, pp. 287-304, May 2007.

[9] G. Jiang and C. Ungureanu, and K.i Yoshihira H. Chen, "Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata," in *Proc. 2nd Intl. Conf. on Automatic Comp.*, Seattle, USA, June 2005, pp. 111-122.

[10] W. Wang, X. H. Guan, and X. L. Zhang, "Modeling program behaviors by hidden Markov models for intrusion detection," in *Proc. of Intl. Conf. on Machine Learning and Cybernetics*, Shanghai, China, Aug. 2004, pp. 2830-2835.

[11] A. K. Ghosh, C. Michael, M. Schatz, and l, "A Real-Time Intrusion Detection System Based on Learning Program Behavior," in *Proc. of the third Intl. Workshop on Recent Advances in Intrusion Detection*, Toulouse, France, Oct. 2000, pp. 93-109.

[12] J.F. Bowring, J.M. Rehg, and M.J Harrold., "Active Learning for Automatic Classification of Software Behavior," *SIGSOFT Soft. Eng. Notes*, vol. 29, no. 4, pp. 195-204, July 2004.

[13] S. Elbaum, S. Kanduri, and A. Andrews, "Trace anomalies as precursors of field failures: an empirical study," *Journal of Empirical Soft. Engg.*, vol. 12, no. 5, pp. 447-469, Oct. 12.

[14] D. Yuxin, Y. Xuebing, Z. Di, D. Li, and A. Zhanchao, "Feature representation and selection in malicious code detection methods based on static system calls," *Computers & Security*, pp. in-press, 2011.

[15] A. Zahalka, K. Goševa-Popstojanova, and J. Zemerick, ""Empirical Evaluation of Factors Affecting Distinction between Failing and Passing Executions," in *Intl. Symp. on Soft. Reliability Engg.*, San Jose, USA, Nov. 2010, pp. 259-268.

[16] K. Hempstalk and I. H. Witten E. Frank, "One-class Classification by Combining Density and Class Probability Estimation.," in *Proc.12th European Conf. on Principles and Practice of Knowledge Discovery in Databases*, Berlin, 2008, pp. 505-519.

[17] D. Y. Yeung and D. Yuxin, "Host-based intrusion detection using dynamic and static behavioral models," *Pattern Recognition*, vol. 36, no. 1, pp. 229-243, Jan. 2003.

[18] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: alternative data models," in *Proc. of 1999 IEEE Symposium on Security and Privacy*, Oakland, USA, May 1999, pp. 133-145.

[19] C. Liu and J. Han, "Failure proximity: a fault localization-based approach," in *Proc. of 14th SIGSOFT Sym. on Foundations of Software Engineering*, Portland, USA, Nov. 2006, pp. 45-56.

[20] M. Brodie et al., "Quickly Finding Known Software Problems via Automated Symptom Matching," in *Proc. of Second Int'l Conf. on Autonomic Computing*, Seattle, USA, June 2005, pp. 101-110.

[21] X. Ding, H. Huang, Y. Ruan, A. Shaikh, and X. Zhang, "Automatic Software Fault Diagnosis by Exploiting Application Signatures," in *Proc. 22nd Conf. on Large Installation System Admin.*, San Diego, USA, Nov. 2008, pp. 23-39.

[22] A. Patcha and J.,M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer Networks*, vol. 51, no. 12, pp. 3448-3470, Aug. 2007.

[23] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A sense of self for Unix processes," in *Proc. of the 1996 IEEE Symp. on Security and Privacy*, Washington, DC, USA, May 1996, pp. 120-128.

[24] S. A. Hofmeyr, S. Forrest, and and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Security*, vol. 6, no. 3, pp. 151-180, Aug. 1998.

[25] D. Y. Yeung and Y Ding., "Host-based intrusion detection using dynamic and static behavioral models," *Pattern Recognition*, vol. 36, no. 1, pp. 229-243, Jan. 2003.

[26] X. D. Hoang, J. Hu, and and P. Bertok., "A program-based anomaly intrusion detection scheme using multiple detection engines and fuzzy inference," *J. Netw. Comput. Appl*, vol. 32, no. 6, pp. 1219-1228, Nov. 2009.

[27] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179-211, April 1990.

[28] Tandon, G., "Machine Learning for Host-based Anomaly Detection," Florida Institue of Technology, Melbourne, Florida, USA, Ph.D. thesis 2008.

[29] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Journal of Empirical Soft. Eng.*, vol. 10, no. 4, pp. 405-435, Oct. 2005.

[30] LTTng. (2011) http://lttng.org/.

[31] N. Devillard and V. Chudnovsky. (2004, March) Etrace--Runtime Tracing Tool. [Online]. http://ndevilla.free.fr/etrace/ [March,2008]

[32] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. USA: Morgan Kaufmann Publisher, 2005.

[33] C. X. Ling, J. Huang, and H. Zhang, "AUC: a statistically consistent and more discriminating measure than accuracy," in *Proc. 18th Intl. Conf. on Artificial Intelligence*, 2003, 519-524.

[34] C. Wohlin et al., *Experimentation in Software Engineering: An Introduction*. Norwell, USA: Kluwer Academic Pub., 2000.

[35] S. Nakagawa, "A farewell to Bonferroni: the problems of low statistical power and publication bias," *Behavioral Ecology*, vol. 15, no. 6, pp. 1044-1045, 2004.

[36] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. NJ, USA: Lawrence Earlbaum Associates, 1988.