Software Behaviour Correlation in a Redundant and Diverse Environment Using the Concept of Trace Abstraction

Abdelwahab Hamou-Lhadj, Syed Shariyar Murtaza, Waseem Fadel, Ali Mehrabian
Software Behaviour Analysis (SBA) Research Lab, Concordia University, Montréal, QC, Canada
{abdelw, s_eskand, w_fadel, al_meh}@ece.concordia.ca

Mario Couture, Raphael Khoury
System of Systems Section, Software Analysis and Robustness Group, Defence Research and Development Canada, Valcartier, Québec, Canada
{mario.couture, raphael.khoury}@drdc-rddc.gc.ca

ABSTRACT
Redundancy and diversity has been shown to be an effective approach for ensuring service continuity (an important requirement for autonomic systems) despite the presence of anomalies due to attacks or faults. In this paper, we focus on operating system (OS) diversity, which is useful in helping a system survive kernel-level anomalies. We propose an approach for detecting anomalies in the presence of OS diversity. We achieve this by comparing kernel-level traces generated from instances of the same application deployed on different OS. Our trace correlation process relies on the concept of trace abstraction, in which low-level system events are transformed into higher-level concepts, freeing the trace from OS-related events. We show the effectiveness of our approach through a case study, in which we selected Linux and FreeBSD as target OS. We also report on lessons learned, setting the ground for future research.

Categories and Subject Descriptors
D.4 **[Operating Systems]**: Security and Protection – *information flow controls, invasive software, security kernels.*

General Terms
Security, Reliability, Algorithms.

Keywords
Redundancy and diversity, Anomaly Detection, Dynamic Analysis, Trace abstraction, Autonomic systems.

1. INTRODUCTION
Redundancy – the process of having multiple instances of the same application run on redundant nodes, is a key component of system resilience in the presence of an security breach. If one node is down (due to an attack for example), a backup (and presumably healthy) instance takes over the load and provides services. Monitoring of the divergence between the behaviors of each instance has also been shown to be an effective method of intrusion detection. Redundancy alone, however, has been shown to be ineffective since an attack can propagate to other nodes and compromise the whole system. To address this issue, the nodes should support some sort of diverse design. Studies have shown that it is difficult for an attacker to compromise multiple diverse nodes with the same attack [19].

There are different ways in which diversity can be introduced in a computing infrastructure including the use of system architectures [12, 40], automatic diversity through randomization [30], design diversity using N-version programming [15], and so on. A thorough survey of redundancy and diversity techniques for security is presented in [20]. To detect anomalies, most of these techniques rely on comparing the output generated by the diverse instances providing the same input. This design, as noted by Giffin et al. in [21], makes these methods vulnerable to attacks that mimic the original system behaviour by returning the correct service response. To overcome this issue, Gao et al. proposed to compare the control flow (represented as execution traces) of diverse processes running the same input using a behavioural distance [17] and Hidden Markov Models [18]. Despite the authors' efforts, their proposed techniques do not overcome the inherit complexity associated with the semantic variations of traces coming from different platforms. In many ways the problem can be thought of as analogous to that of comparing two sentences from different languages.

In this paper, we propose a new approach for anomaly detection in a diverse environment. Our approach relies on the concept of trace abstraction, which is the process of transforming a trace of low-level events into higher-level concepts by abstracting out details pertaining to the computing platform. In other words, the resulting abstract trace contains operations that are agnostic to the platform from which the trace is generated. For example, the content of an event-based trace generated from reading a file on disk can vary significantly from one operating system to another. The aim of trace abstraction is to transform these low-level events into a higher concept, such as 'read file', making it possible to compare the traces despite the environment in which they have been generated.

The focus of this study is on operating system (OS) diversity. OS diversity is an effective way to improve the overall resilience of the system in the presence of kernel-level attack threats. For example, if an attack is designed to exploit Linux vulnerability, it will most likely fail to compromise a Windows system since both systems exhibit different flaws. In this study, we limit ourselves to two nodes for simplicity reasons (although the concepts presented

RACS'13, October 1–4, 2013, Montreal, QC, Canada.

in this paper can easily be extended to multiple OS). The selected operating systems are Linux Ubuntu[1] and FreeBSD[2]. Naturally, other operating systems can be used. Our choice is motivated by the following criteria:

- Although Linux and FreeBSD differ internally, they both derive from Unix. Similar conventions have been used to develop both systems. This permits the reuse of expertise.
- Both systems are open source and free. This is very important in the context of security since more advanced security mechanisms might require investigating the source code or even modifying it. This would not be possible if a proprietary system (such as Windows) is used.
- Both systems enjoy a large online community support with extensive documentation. We used online documentation to understand the system call mechanisms of both systems and be able to compare traces generated from their kernels.
- Both systems have built-in tracing capabilities. We used LTTng[3] to trace the Linux kernel and DTrace[4] to trace FreeBSD.

This article makes two key contributions to the scientific literature on intrusion detection. First it proposes a new trace abstraction algorithm that allows the translation of low-level system specific traces into abstract traces that capture the behaviour of the target system in a semantics-based and system agnostic representation. Second, the paper shows how this representation can be used for intrusion detection by correlating the simultaneous executions of two diverse systems. This correlation leverages the fact that it is difficult to simultaneously attack two different systems in order to build more secure systems.

The remainder of this paper is structured as follows. The next section develops the methodology we adopt in this study. This is followed in section 3 by a case study which highlights the strengths and limitations of the approach. Section 4 presents an overview of the relevant literature. Section 5 draws the conclusion to the study and outlines directions for future research.

2. APPROACH

Our approach is shown in Figure 1. The first instance of the application runs under normal conditions and is intended to capture healthy behaviour. The second instance is deliberately infected by a simulated attack. Traces generated from these instances during operation are first abstracted out using our trace abstraction process and then compared. As mentioned earlier, the abstraction process turns a raw trace into a more descriptive and meaningful sequence of operations rather than a sequence of low-level events.

An alternative approach would be to design mapping rules to map system calls in Linux to their corresponding ones in BSD and use these rules as a reference model to guide the trace correlation process. The challenge, however, is to build an adequate set of mapping rules that takes into account the many variations that exist in the system call mechanisms of different operating systems. Trace abstraction eliminates the need for a mapping model. It is also useful in itself since it reduces the size of traces, which is necessary for the effective use of several trace analysis techniques including trace correlation (see [24, 25] for more discussion on the use of trace of abstraction techniques for trace comprehension). However, unlike a rule-based model, trace abstraction causes loss of information. This information might be needed to detect some attacks. Future work should focus on determining the level of details that abstract traces should contain to reduce the effect of lost information.

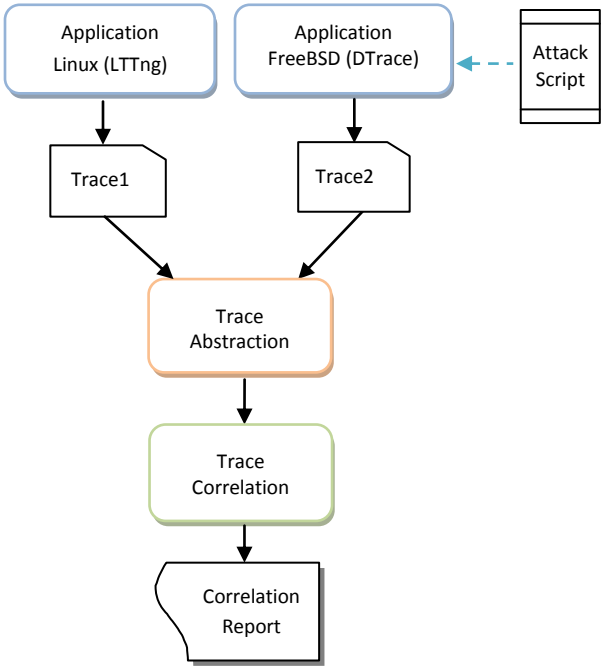

Figure 1. Correlating traces using trace abstraction

2.1. Trace Generation

The traces used in this study are system call traces, which depict the interactions between user applications and the kernel. We chose system call traces because they have been used extensively in the area of intrusion detection (e.g., [15, 16, 27, 34, 35]). The common approach is to build a reference model (using various machine learning techniques) from traces of system calls during normal execution of the system (usually in a lab environment). A fault detection technique can then be developed by observing, using monitoring capabilities, any deviations of the deployed system from the baseline model. These studies, however, do not take into account diversity.

We used two tracers, namely LTTng and Dtrace, which generate traces for Linux and BSD respectively. LTTng is a tracer that was developed to extract information from the Linux kernel, user space libraries, and user applications by running a recompiled instrumented version of the kernel [7]. In this study, we installed LTTng version 2.6 on Linux Ubuntu 10 with kernel version 2.6.34. LTTng traces can be generated by directly running LTTng through the command line, or by using a tool called LTTV (LTTng Viewer) through its graphical user interface [8].

An LTTng trace contains information related to the process being executed including the trace file, event name, time in seconds, time in nano seconds, trace file path, process ID, process name, parent ID, process group ID, execution mode, and other parameters related to the event being executed. However this

[1] http://www.ubuntu.com/
[2] http://www.freebsd.org/
[3] http://lttng.org/
[4] http://wiki.freebsd.org/DTrace

information could vary depending on the selected trace points, and hence, there may be differences in the traces from one version to another and from one testing platform to another. A typical LTTng trace has the following format:

```
TraceFile.Event Time(s).Time(ns)
(Path_To_Trace_File), PID, PGID, ProcessName,
PPID, MODE {PARAMS}
```

An example of a trace using the above format is shown below. This trace represents the system calls executed when a file is opened, data written to it, and then closed.

```
kernel.syscall_entry: 442192.435342606
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id
= 5 [sys_open+0x0/0x40] }
fs.open: 442192.435348299 (/tmp/trace10/fs_1),
22438, 22438, ./Files, , 29184, 0x0, SYSCALL { fd
= 3, filename = "output.txt" }
kernel.syscall_exit: 442192.435348407
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, USER_MODE { ret = 3 }
kernel.syscall_entry: 442192.435350985
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id
= 4 [sys_write+0x0/0xc0] }
fs.write: 442192.435351307 (/tmp/trace10/fs_1),
22438, 22438, ./Files, , 29184, 0x0, SYSCALL {
count = 72, fd = 3 }
kernel.syscall_exit: 442192.435351415
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, USER_MODE { ret = 72 }
kernel.syscall_entry: 442192.435351522
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, SYSCALL { ip = 0xb7fac430, syscall_id
= 6 [sys_close+0x0/0x100] }
fs.close: 442192.435351629 (/tmp/trace10/fs_1),
22438, 22438, ./Files, , 29184, 0x0, SYSCALL { fd
= 3 }
kernel.syscall_exit: 442192.435351844
(/tmp/trace10/kernel_1), 22438, 22438, ./Files, ,
29184, 0x0, USER_MODE { ret = 0 }
```

Figure 2. An example of an LTTng trace

DTrace is a tracing tool that was originally developed by Sun Microsystems for Solaris to provide users with ways to understand and troubleshoot applications and the operating system. DTrace has been ported to several other Unix-like systems including FreeBSD. In order to install DTrace on FreeBSD, the DTrace package must be embedded in the BSD Kernel. DTrace should be provided with a script file that describes the desired output format of the trace. Figure 3 shows an example of the output file of DTrace. The CPU column identifies the system that is used, the ID shows the process id and the FUNCTION:NAME is the result of the script used to generate the trace (the figure shows system calls).

One of the key differences between LTTng and DTrace traces consists of the amount of information contained in the trace as well as the way the information is structured. We found that LTTng is much more expressive than DTrace. A typical LTTng event contains the system call entry or exit information, the timestamp, the CPU number, the process id of the current process as well as that of its parent, the mode (system or user mode), and the number of the bytes that need to be read from the specified file descriptor. A typical DTrace event consists of only the system call and the file descriptor. The generation of additional information is possible but requires additional scripting instructions, which is not practical, especially for operations that require on-demand probing where this information must be provided on the fly.

```
CPU   ID     FUNCTION:NAME
1     39240  ioctl:return dtrace
1     39239  ioctl:entry SYSCALL:ioctl, dtrace
1     39240  ioctl:return dtrace
1     39239  ioctl:entry SYSCALL:ioctl, dtrace
1     39240  ioctl:return dtrace
1     39535  _sysctl:entry SYSCALL:_sysctl, dtrace
1     39536  _sysctl:return dtrace
1     39535  _sysctl:entry SYSCALL:_sysctl, dtrace
1     39536  _sysctl:return dtrace
```

Figure 3. An example of a DTrace trace

2.2. Trace Abstraction

The trace abstraction process takes an LTTng or FreeBSD system call trace as input and returns a trace composed of a sequence of high-level operations as output. Each operation is built from an aggregation of several low-level events. The abstraction process relies on a pattern library (i.e., a knowledge base) that we have developed to characterize the main operations of the Linux and FreeBSD kernel.

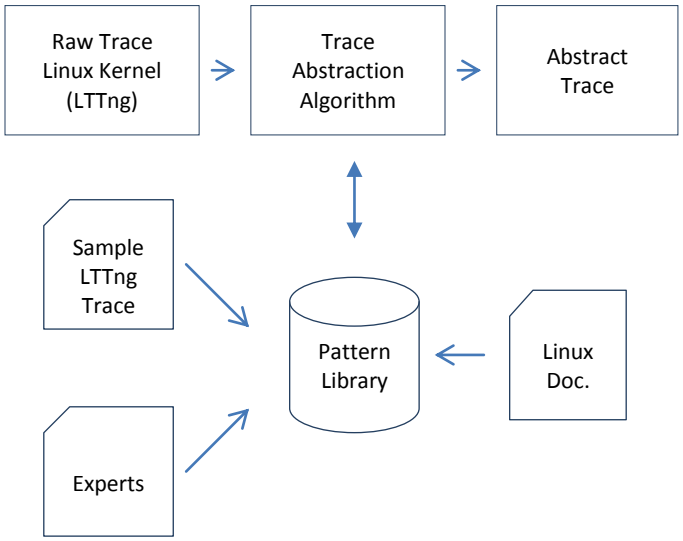

Figure 4. Trace abstraction approach

There exists several trace abstraction techniques developed for the purpose of program comprehension. A good survey of trace abstraction techniques for program comprehension can be found in [5]. The objective of these techniques is to reduce the size of traces by keeping as much of their essence as possible. For example, Hamou-Lhadj et al. [24, 26] showed that the simple removal of utilities from raw traces can result in traces that can be understandable by humans. The content of such abstract traces still consists of events found in the original trace. In this paper, by abstraction, we mean changing the level of granularity of the information contained in the trace by grouping system call streams into operations.

To build the pattern library, we studied both the Linux and FreeBSD kernels and their system calls mechanisms. We also

executed a number of applications with different operations and examined the traces thus generated to understand how the kernel functions (as shown in Figure 4 in the case of Linux). In the case of Linux, we also had access to experts in the area, namely, the designers of the LTTng tool.

The pattern library for the Linux kernel system calls models the most common operations of the Linux kernel. These operations include: File Management (Open, Read, Write, Close. Access, Stat), Socket Management for TCP and UDP (Create, Bind, Connect, Listen, Accept, Send, Receive, Close), Process Management (Clone, Execute, Exit), Memory Management and Page Faults. In total, we created eighty patterns modeled and implemented as finite state machines. These patterns are documented in details in [13]. Examples of file operations patterns are shown in Figure 5.

Figure 5 shows two patterns: File Read and File Write. In this example, the 'file read' pattern involves entering the sys_read system call, executing the read function with the appropriate parameters to read data from the opened file, and finally exiting the sys_read system call. Similarly, the File Write pattern, which represents the action of writing data to an open file, involves entering the sys_write system call, executing the write function with the appropriate parameters to write data to the open file, and finally exiting the sys_write system call. An analogous process has been followed in the development of patterns for other Linux operations such as socket and process management operations.

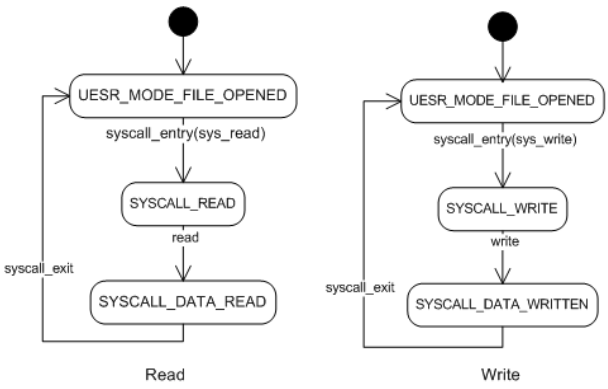

Figure 5. Example of two patterns 'read and write' file Linux operations

We followed the same process to create patterns that represent FreeBSD kernel-level operations. For this study, we have only focused on patterns that model file and process management operations. We did not attempt to model all patterns in FreeBSD since we only experimented with a simple attack in which the file operations patterns were involved. Future research should focus on improving the pattern library for FreeBSD and Linux in order to generalize the abstraction process and support additional operations.

Once the pattern libraries for both OS are in place, the abstraction process takes an LTTng or FreeBSD trace as input and starts by parsing the trace from the first line, comparing each event with the event patterns that exist in the corresponding pattern library until a match is found. Subsequently, the pattern containing the event is shifted from its old state to a newer state waiting for the next event to be read. Then, a new line is read by the algorithm, and the events are compared. When an event causes a pattern to be shifted to a final state, a new high-level construct representing that pattern is created and pushed into a stack of high-level constructs. When the algorithm has finished processing the entire trace, the patterns of events will be replaced with higher-level constructs, ordered in a stack, that reflect the system behaviour in a more compact and readable format.

Repeats Detection Algorithm

Inputs:

- N = The minimum length of a repeat
- S = Stream of data (i.e., the trace)Output
- A hash table T that contains the repeats, their location in the stream of data, and their frequency

1. Set T to the empty table
2. For all i from 0 to the length of S
  2.1 Find an N-gram sequence of size N starting from S[i]
  2.2 If T does not contain this N-gram
    2.2.1 Place it and its location in the stream (i.e., i), and its frequency in the table (in this case the frequency is 1 since this is the first occurrence of this N-gram)
    2.2.2 Update T by inserting the N-gram, its location, and its frequency
  2.3 If T contains N-gram
    2.3.1 For each previous occurrence of the N-gram check if, by extending the current N-gram and comparing it to an extension of the previous occurrence, we can obtain a longer sequence. If yes then replace the previously stored occurrence with the new N-gram with the extension. If not then update the number of occurrences of the stored N-gram
3. Output T.

Figure 6. Repeats detection algorithm

2.3. Trace Correlation Algorithm

The next step is to compare the abstract traces to enable the detection of potential deviations. One way of doing this is by performing an event-to-event matching. This approach, however, has several limitations. To start with, it does not account for the number of repetitions that occur in a trace that can vary from one system to another. By studying examples of LTTng and Dtrace traces generated from the same system, we found that it is common to have significant differences in the number of operations that appear in both traces. For example, write and read operations may appear at different frequencies due to the way the buffer is set for each operating system. In many cases, we also found that the execution order of some operations also varies (due, perhaps, to parallelism or optimizations done by the compiler). Comparing traces using an event-to-event matching is simply too restrictive.

In this paper, we propose to compare traces based on their main behaviour. Previous studies conducted by Hamou-Lhadj et al. and Idris et al. [22, 28], show that the main behaviour embedded in a trace often takes the form of a pattern, defined here as a sequence of data which occurs non-contiguously in a trace at least twice. We use the term *repeats* to distinguish the patterns in this context

with the patterns used for abstraction. These repeats form the basis of our behavioral comparison of the event traces: the more common repeats two traces contain, the more similar they are.

Our repeat detection algorithm is described in Figure 6 and is based on n-gram extraction techniques, a well-known approach used in text mining. The algorithm takes as input, N, the minimum number of elements in a repeat and the trace (which is seen here as just a data stream). It then goes through the data stream and finds all longest sequences of minimum size N. It uses a hash table to save the repeats, their locations in the trace, and their frequency. N is specified by the user and could vary from one application to another. It is therefore important for the tool that implements this approach to allow enough flexibility to experiment with different values of N. An alternative would be to find the longest patterns that exist in the data, but this might lead to very large repeats containing large portions of the abstract trace defeating the purpose of dividing a trace into repeats in the first place.

Let ST1 and ST2 be two sets of repeats, (extracted from the trace), we have developed a distance metric that measures how "far" ST1 is from a set ST2. This is accomplished by measuring the difference between ST1 and ST2 using the edit distance [36]. The edit distance measures the cost of substitution, insertion and deletion operations needed to transform one set into another. We define three sets that are used to measure the distance between two traces:

$$M = \{ r \mid r \in ST_1 \wedge r \in ST_2 \}$$
$$I = \{ r \mid r \in ST_1 \wedge r \notin ST_2 \}$$
$$J = \{ r \mid r \notin ST_1 \wedge r \in ST_2 \}$$

Here *M* is a mapping set that contains all the repeats that are in ST1 and ST2; *I* is a set of repeats that are in ST1 but not in ST2 and finally, J is the set of repeats that are in ST2 but not in ST1. The distance between ST1 and ST2 using the edit distance is computed as follows:

$$dist(ST_1, ST_2) = p|M| + q|I| + r|J|$$

where *p* is the cost of substitution, *q* is the cost of deletion, and *r* is the cost of insertion.

In this study, we consider p = 0 and q = r = 1 (see [36] for more details on using the edit distance). In other words, we replace substitution with insertion and deletions. Other weights could be given to the cost of substitution, insertion, and deletion if justified with thorough experimentation. In this study, we limit ourselves to equal weights between the insertion and deletion operations. Once the distance between two traces is measured, the similarity metric can be computed as follows:

$$sim(ST1, ST2) = 1 - \frac{dist}{|ST1| + |ST2|}$$

where |ST1| and |ST2| denote the number of repeats in ST1 and ST2 respectively.

This metric yields a result between 0 and 1. If the result is zero then the two traces are completely different. If the result converges to one then the two traces contain many similarities. A threshold should be used to determine the extent to which two traces are considered similar (or dissimilar). We anticipate that this threshold will vary from one application to another.

3. CASE STUDY

The objective of the case study is to study the effectiveness of our approach by testing it with a real attack on two instances of the same system running on two different OS, namely Linux FreeBSD. We stress that while we experimented with these two OS, the principles exposed in this paper are general enough to be applied in different contexts, such as a different choice of OS, or even in the case where diversity is introduce at another layer of the system, such as hardware or user-level application.

3.1. Target System and Experiment Setting

We focus in this study on the Easy Editor (ee) ver. 1.4 system[5] as the target application. The ee editor is a simple screen oriented text editor for Linux and FreeBSD. It supports most common operations found in today's text editors (inserting text, editing, searching for text, etc.).

The experimental setting consists of two Intel Core Duo 1.86 GHz computers running Linux and FreeBSD separately. We installed LTTng on the Linux machine and DTrace on the FreeBSD machine. Finally, we installed the ee application on both systems. We run the healthy application on Linux and the attacked version on FreeBSD. The attack that was simulated on FreeBSD is described in the next section. It should be noted that, despite our efforts, it was a challenge to find attacks that exploit vulnerabilities in systems that run on both Linux and FreeBSD. Most attacks reported on known attack repositories such as CVE[6] (Common Vulnerabilities and Exposures) come without implementation, which complicates the process of simulating and generating real attacks.

3.2. Simulation of the Attack

In FreeBSD, we open two terminals, one with root privileges as a real user and the other one, also with root privileges, as the attacker. Root runs the ee editor and creates test.txt (a sample text file), in which the text "Hello" is written in the editor. We then apply the method ispell() to check the spelling. FreeBSD creates a temporary file in the /tmp/ folder called ee followed by the process id (that we refer to here as ee.processID). On the other terminal, the attacker creates a symbolic link (we wrote a script to create the link automatically) to the file "ee.processID" inside the folder /tmp/. This will provide the attacker the opportunity to have a link with the root privilege and it is now possible for the attacker to overwrite the temporary file that was created (for example by inserting malicious code). A subsequent usage of this file can cause the malicious code to run.

To prevent such attacks, race guards are normally implemented in the kernel to protect it against vulnerabilities that result from race conditions during temporary file creation — a typical TOCTTOU (Time of Check To Time Of Use) problem.

It appears that the ispell_op function used by ee while executing spell check operations employs an insecure method of temporary file generation. This method produces predictable file names based on the process ID and fails to confirm which path will be overwritten by the user.

These predictable temporary file names seem to be problematic because they allow an attacker to take advantage of a race condition in order to execute a symlink attack, which could allow

[5] http://www.ipnom.com/FreeBSD-Man-Pages/ee.1.html
[6] http://cve.mitre.org/

him to overwrite files on the system in the context of the user running the ee editor. This vulnerability was reported in 2006 (See CVE bug report entry below) and fixed in subsequent versions of FreeBSD.

Bugtraq ID: 16207
CVE Name: CVE-2006-0055
URL:http://www.securityfocus.com/bid/16207/info
FreeBSD Advisory: SA-06:02.ee

On the Linux machine, we simply run the same scenario without the attack (i.e., opening ee and writing to it the Hello message). We generated two traces from the execution of both the healthy and the unhealthy instances of the ee application. The sizes of the raw traces of LTTng and FreeBSD are 142 MB and 152 MB respectively. These are considered relatively small traces and are used in this paper as a proof of concept. Future work should focus on large scale experimentation using more attacks with the potential of generating significantly larger traces.

3.3. Correlation Results

After applying the abstraction algorithm, the correlation process resulted in 54% similarity (with N = 2), which indicates that the two traces differ substantially. We further examined the content of the healthy and unhealthy traces semi-automatically using SEAT (Software Exploration and Analysis Tool), which is a tool that permits the analysis and exploration of large traces [23]. This exploration required both the examination of the abstract and original traces. We tried to find keywords in the original traces that could reveal the attack. For example, we knew that the attack uses the command *ln* to create the symbolic link. This command could only be found just in the unhealthy trace and not the healthy one.

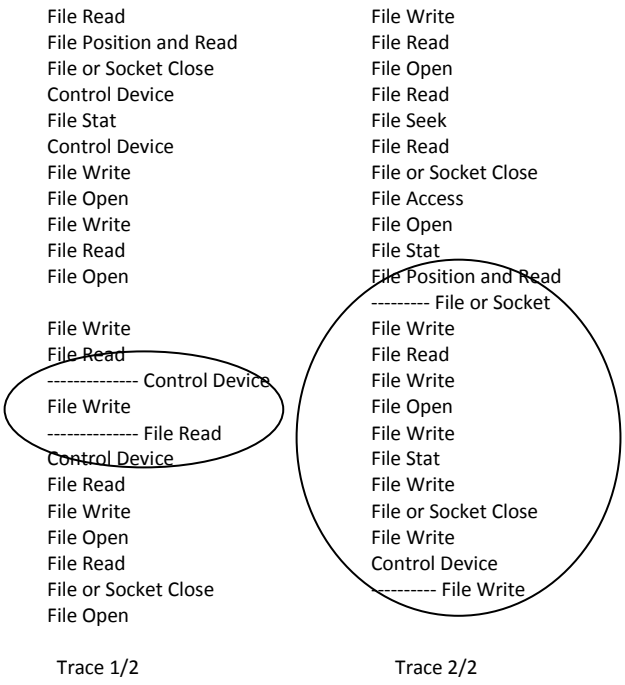

Figure 7. Part of the FreeBSD unhealthy trace with the attack

After thorough investigation of both traces (LTTng and DTrace traces), we were able to detect places in the FreeBSD trace where the effect of the attack appeared (see Figure 7). In the first part of the figure (circled area), it seems that the system was saving new information which consisted in the new link to the temporary file. In the second part of the deviation, after investigating the file descriptors, we found that this part represents an interaction between the root terminal and the admin terminal that could represent the linking to temporary file. These interactions did not appear in the LTTng traces, which clearly indicates the presence of a deviation with respect to the normal behaviour.

This analysis required some manual work to detect whether or not the attack is indeed what caused the behavioural deviation to occur. We recognize that deeper and further analysis is needed to have automatic ways to pinpoint to the attacks in the trace. On the other hand, we believe that this forensic analysis can be very beneficial in situations where automatic detection fails. Also, the abstraction process enabled this forensic analysis, the same way it helped performing a fair correlation between the two different traces.

4. RELATED WORK

Trace correlation as a practical tool of security analysis draws on the theoretical background of employing redundancy and diversity to ensure the security of systems. Indeed, redundancy and diversity has been the subject of many studies with a particular emphasis on building fault-tolerant systems and improving overall system reliability. In this regard, a rich literature dating back to the late 70s exists around the N-version programming development paradigm [1], which consists in using multiple redundant software components to increase the reliability of key systems. This literature reports a number of experiments conducted in academic settings to evaluate the feasibility and efficiency of the approach as well as theoretical enquiries aimed at modeling and reasoning about the behaviour of N-version systems.

The question of using N-version programming for security, rather than for reliability, was raised in a number of studies. Littlewood et al. examined the question in [32]. Bessani et al. argued in favour of using diversity for security on the basis of the recorded distribution of vulnerabilities in several operating systems [3]. The various layers where diversity can be inserted are examined from the perspective of maximizing security [30], whereas the utilization of design diversity to protect against computer viruses was examined in [28].

The use of diversity for security rather than for reliability, termed automated diversity, was first suggested by Forrest et al. in [14]. The authors observed that the homogeneity of computer systems, makes the whole infrastructure vulnerable. Drawing on an analogy to biological systems, Forrest et al. argued that the robustness of systems could be improved if the program instance used by each user differed slightly from that of every other user. In this context, a large body of work exists that examines the way diversity should be introduced in a computing infrastructure (e.g., [3, 31, 40]). However, these studies do not deal with the problem of correlating the behaviour of the applications, which is the main object of this paper. Other studies that propose using diversity towards the goal of securing systems include [4, 6].

Given the above background, this study focuses on trace abstraction and correlation as a tool for intrusion detection. In this regard, trace abstraction represents a useful strategy to circumvent the considerable size of the output of low-level tracers. The resulting abstracted trace can then serve as basis for different types of analyses. Most trace abstraction techniques are based on

pattern matching: low-level events are grouped together to form a single abstract event according to a library of known patterns, which maps the former and the latter. This technique was applied to the traces generated by the Lttng tracer by [11, 12, 13, 33, 39]. Matni and Dagenais propose in [33] an automata-based approach that detects the occurrence of patterns of suspect behaviour in kernel traces. Fadel employs user-defined patterns to extract compact abstract traces from larger low-level system call traces [13]. Waly and Ktari use a similar technique to and show how these abstract traces can be used for anomaly detection [39]. In a recent study, Ezzati-Jivan and Degenais [12], use information about the current system state to provide a more precise abstraction and also show that this finer abstraction can be used for intrusion detection. A number of other tools, such as STATL [11], also use pattern matching for intrusion detection.

Perhaps the closest work to our study is that of Gao et al. in [17, 18]. The authors propose two approaches for measuring the control flow of applications deployed in a diverse environment: Behavioural distance and Hidden Markov Models. The behavioural distance is used to measure the extent to which traces are similar. The authors, however, have to sacrifice the order of sequences to overcome the problem of comparing raw traces. In subsequent work [18], the authors use Hidden Markov Models (HMM) to build a reference model that is used to guide the correlation process.

Finally, we should mention the fact that there are many studies that use system call traces for anomaly detection (e.g., [15, 16, 27, 34, 35]). These studies, however, do not take into account redundancy and diversity, which is the focus of the work presented in this paper.

5. CONCLUSION

We presented an approach for anomaly detection in the presence of OS diversity. We successfully compared traces generated from two OS, namely Linux and FreeBSD. For the trace correlation to make sense, the traces were first abstracted out using a pattern library that we had built to capture the main operations of Linux and FreeBSD. For our approach to be generalized, we need to conduct further studies involving different OS and more attacks.

6. ACKNOWLEDGMENTS

This work is partly supported by the NSERC and DRDC (Defence R&D Canada), Valcartier, QC, Canada.

7. REFERENCES

[1]. Avizienis, A. 1985. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering, 11(12)* , 1491-1501.

[2]. Barrantes, E., Forrest S. 2006. Increasing communications security through protocol parameter diversity. In *Proceedings of the 32nd Latin-American Conference on Informatics* (Santiago, Chile, August 25-26, 2006). CLEI'06.

[3]. Bessani, N., Obelheiro, R. R., Sousa, P., Gashi, I. 2008. On the effects of diversity on intrusion tolerance. *Technical Report, Department of Informatics, University of Lisbon, DI/FCUL TR 08–30.*

[4]. Bruschi, D., Cavallaro, L., and Lanzi, A. 2007. Diversified Process Replicae for Defeating Memory Error Exploits. In *Proceedings of the 3rd International Workshop on Information Assurance* (New Orleans, Louisiana, USA, April 11-13, 2007). WIA'07. IEEE Computer Society, 434 – 441. DOI= 10.1109/PCCC.2007.358924

[5]. Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R. 2009. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering (TSE), 35(5)*, IEEE Computer Society, 684-702. DOI= 10.1109/TSE.2009.28

[6]. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J., 2006. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Proceedings of the 15th USENIX Security Symposium* (Vancouver, B.C., Canada, July 31-August 4, 2006). USENIX Association, Article No 9.

[7]. Desnoyers, M., and Dagenais, M. R. 2006. The LTTng tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux. In *Proceedings of the Ottawa Linux Symposium,* 2006.

[8]. Desnoyers, M. 2009. Low-impact operating system tracing. *Ph.D. Dissertation, École Polytechnique de Montréal, Montréal, QC, Canada.*

[9]. Deswarte, Y., Powell, D. 2004. Intrusion tolerance for Internet applications. In *Proceedings of the IFIP 18th World Computer Congress on Building the Information Society (*Toulouse, France, August 22–27, 2004). 241–256. DOI= 10.1007/978-1-4020-8157-6_22

[10]. Deswarte, Y., Kanoun, K., Laprie, J.-C. 1998. Diversity against accidental and deliberate faults. In *Computer Security, Dependability, and Assurance: From Needs to Solutions* (York, UK; Williamsburg, VA, July 7-9, 1998), 171–181. DOI= 10.1109/CSDA.1998.798364

[11]. Eckmann, S., Vigna, G., Kemmerer, R. 2002. STATL: An attack language for state based intrusion detection system. *Journal of Computer Security 10(1-2),* IOS Press, 71-103.

[12]. Ezzati-Jivan, N., Dagenais, M. R. 2012. Stateful Synthetic Event Generator from Kernel Trace Events. *Hindawi Journal on Advances in Software Engineering*, Volume 2012 (2012), Article ID 14036. DOI= http://dx.doi.org/10.1155/2012/140368

[13]. Fadel W. 2010. Techniques for the Abstraction of System Call Traces to Facilitate the Understanding of the Behavioural Aspects of the Linux Kernel. *Masters thesis, Concordia University, Montreal, QC, Canada. Available online, URL: http://spectrum.library.concordia.ca/7075/*

[14]. Forrest, S. Somayaji, A., and Ackley, D. H. 1997. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems* (Cape Cod, MA, USA, May 5-6, 1997), 67–72. DOI= 10.1109/HOTOS.1997.595185

[15]. Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T.A. 1996. A Sense of Self for Unix Processes. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, CA, USA, May 6-8, 1996), 120-128. DOI= 10.1109/SECPRI.1996.502675

[16]. Frossi, A., Maggi, F., Rizzo, G. L., and Zanero, S. 2009. Selecting and Improving System Call Models for Anomaly through System Call Sequence and Argument Analysis. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Milan, Italy, July 9-10, 2009), 206-223. DOI= 10.1007/978-3-642-02918-9_13

[17]. D. Gao, M. K. Reiter, and D. X. Song, "Behavioral distance measurement using hidden markov models," *In Proc. of the 9th International Symposium on Recent Advances in Intrusion Detection, Lecture Notes in Computer Science,* pp. 19–40, 2006.

[18]. Gao, D., Reiter, M. K., and Song, D. X. 2009. Beyond Output Voting: Detecting Compromised Replicas Using HMM-Based Behavioral Distance. *IEEE Transactions on Dependable and Secure Computing,* 6(2), 96–110. DOI= 10.1109/TDSC.2008.39

[19]. Garcia, M., Bessani, A., Gashi, I., Neves, N. and Obelheiro, R. 2011. OS Diversity for Intrusion Tolerance: Myth or Reality? In *Proceedings of the International Conference on Dependable Systems and Networks* (Hong Kong, China, June 27-30, 2011), 383 – 394. DOI= 10.1109/DSN.2011.5958251

[20]. Gherbi, A., Charpentier, R., and Couture, M. 2010. Redundancy with diversity based software architectures for the detection and tolerance of cyber-attacks. *DRDC Valcartier, Technical Report TM 2010-287*.

[21]. Giffin, J. T., Jha, S., and Miller, B. P. 2006. Automated Discovery of Mimicry Attacks. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection, Vol. 4219 of Springer Lecture Notes in Computer Science* (Hamburg, Germany, September 20-22, 2006), 41–60. DOI=10.1007/11856214_3

[22]. Hamou-Lhadj, A. and Lethbridge, T. 2003. An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls. In *Proceedings of the 1st ICSE International Workshop on Dynamic Analysis (WODA),* Available online at http://homes.cs.washington.edu/~mernst/pubs/woda2003-proceedings.pdf#page=33

[23]. Hamou-Lhadj, A. And Lethbridge, T. 2005. SEAT: A Usable Trace Analysis Tool. In *Proceedings of the 13th International Workshop on Program Comprehension (*St. Louis, Missouri, USA, May 15-16, 2005), 157-160. DOI=10.1109/WPC.2005.30

[24]. Hamou-Lhadj, A. 2006. Techniques to Simplify the Analysis of Execution Traces for Program Comprehension. *Ph.D. Dissertation, School of Information Technology and Engineering (SITE),* University of Ottawa, Ottawa, ON, Canada.

[25]. Hamou-Lhadj, A., and Lethbridge, T. 2005. Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools. In *Proceedings of the 10th International Conference on Engineering of Complex Computer Systems* (Shanghai, China, June 16-20, 2005), 559-568. DOI= 10.1109/ICECCS.2005.57

[26]. Hamou-Lhadj, A., and Lethbridge, T. 2004. Reasoning About the Concept of Utilities. *ECOOP International Workshop on Practical Problems of Programming in the Large, Oslo, Norway, Lecture Notes in Computer Science (LNCS), Vol 3344,* Springer-Verlag, 10-22.

[27]. Hoang, X. D., Hu, J., and Bertok, P. 2009. A program-based anomaly intrusion detection scheme using multiple detection engines and fuzzy inference. *Journal Network Computing and Application, 32(6),* 1219-1228. DOI= 10.1016/j.jnca.2009.05.004.

[28]. Idris, M., Mehrabian, A., Hamou-Lhadj, A., Khoury, R. 2012. Pattern-Based Trace Correlation Technique for Software Evolution. In *Proceedings of the 3rd International Conference on Autonomous and Intelligent Systems, Springer Lecture Notes in Artificial Intelligence Series* (Aveiro, Portugal, June 25-27, 2012), 159-156. DOI= 10.1007/978-3-642-31368-4_19

[29]. Joseph M. K. and Avizienis, A. 1988. A fault tolerance approach to computer viruses. In *Proceedings of the International Conference on Security and privacy* (Oakland, CA, USA, Apr 18-21, 1988), 52–58. DOI= 10.1109/SECPRI.1988.8097

[30]. Just, J. E., and Cornwell, M. R. 2004. Review and analysis of synthetic diversity for breaking monocultures. In *Proceedings of the ACM Workshop on Rapid Malcode,* 23–32. DOI= 10.1145/1029618.1029623

[31]. Keromytis, A. D. 2009. Randomized instruction sets and runtime environments past research and future directions. *IEEE Security and Privacy, 7(1),* 18–25, DOI= 10.1109/MSP.2009.15

[32]. Littlewood, B., Strigini, L. 2004. Redundancy and Diversity in Security. In *Proceedings of the 9th European Symposium on Research Computer Security* (Sophia Antipolis, France, September 13 - 15, 2004), 423-438. DOI= 10.1007/978-3-540-30108-0_26

[33]. Matni, G., Dagenais, M. 2009. Automata-based approach for kernel trace analysis. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering* (St. John's, NL, May 3-6, 2009), 970 – 973, DOI= 10.1109/CCECE.2009.5090273

[34]. F. Maggi, F., Matteucci, M., Zanero, S. 2010. Detecting Intrusions through System Call Sequence and Argument Analysis. *IEEE Transactions on Dependable and Secure Computing, 7(4),* 381-395. DOI- 10.1109/TDSC.2008.69.

[35]. Mutz, D., Valeur, F., Kruegel, C., Vigna, G. 2009. Anomalous System Call Detection. *ACM Transactions on Information and System Security, 9(1),* 61-93. DOI= 10.1145/1127345.1127348.

[36]. Valiente, G. 2001. Simple and efficient tree comparison. *Technical Report LSI-01-1-R, Technical University of Catalonia, Department of Software,* 2001.

[37]. Yang, W. 1991. Identifying syntactic differences between two programs. *Software, Practice and Experience Journal,* 21(7), 739–755. DOI= 10.1002/spe.4380210706.

[38]. Xu, J., Kalbarczyk, Z., Iyer, R. K. 2003. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems* (Florence, Italy, Oct.6-18, 2003), 260 – 269. DOI= 10.1109/RELDIS.2003.1238076.

[39]. Waly, H., Ktari, B. 2011. A Complete Framework for Kernel Trace Analysis. *IEEE Canadian Conference on Electrical and Computer Engineering (*Niagara Falls, ON, May 8-11, 2011), 1426 – 1430. DOI= 10.1109/CCECE.2011.6030698.

[40]. Wang, F., Jou, F., Gong, F., Sargor, C., Goseva-Popstojanova, K., Trivedi, K. 2003. SITAR: A Scalable Intrusion-Tolerant Architecture for Distributed Services. In *Foundations of Intrusion Tolerant Systems,* 153-155. DOI= http://doi.ieeecomputersociety.org/10.1109/FITS.2003.126494.