Frequent Pattern Mining for Kernel Trace Data

Christopher LaRosa, Li Xiong, Ken Mandelberg
Department of Mathematics and Computer Science
Emory University, Atlanta, GA 30322
+1 404-727-7580
{clarosa,lxiong,km}@mathcs.emory.edu

ABSTRACT
Operating systems engineers have developed tracing tools that log details about process execution at the kernel level. These tools make it easier to understand the actual execution that takes place on real systems. Unfortunately, uncovering certain types of useful information in kernel trace data is nearly impossible through manual inspection of a trace log. To detect interesting inter-process communication patterns and other recurring runtime execution patterns in operating system trace logs, we employ data mining techniques, in particular, frequent pattern mining. We present a framework for mining kernel trace data, making use of frequent pattern mining in conjunction with special considerations for the temporal characteristics of kernel trace data. We report our findings using our framework to isolate processes responsible for systemic problems on a LINUX system and demonstrate our framework is versatile and efficient.

Categories and Subject Descriptors
D.4.8 [**Operating Systems**]: Performance – *measurements, monitors, operational analysis, stochastic analysis*; D.2.5 [**Software Engineering**]: Testing and Debugging – *tracing*; G.3 [**Probability and Statistics**]: *time series analysis*; H.2.8 [**Database Applications**]: *data mining.*

General Terms
Measurement, design, performance, experimentation.

Keywords
Frequent pattern mining, sequential pattern mining, kernel tracing.

1. INTRODUCTION
The introduction of low-impact kernel-level tracing tools allows for comprehensive and transparent reporting of process and operating system activity. An operating system trace log provides detailed, explicit information about which processes use which system resources at what time. This time series data contains underlying knowledge, such as common execution patterns. This information can assist in many systems-related tasks: application debugging, security enforcement, performance optimization, operating system debugging, and dynamic reconfiguration. However, while kernel trace collection tools have advanced and matured, there remains a lack of trace analysis tools for extracting useful knowledge from raw trace logs.

SAC'08, March 16-20, 2008, Fortaleza, Ceará, Brazil.

Motivation and Goals. Most current trace collection tools, such as the Linux Trace Toolkit (LTT) [22] and Solaris's dTrace [2], provide powerful mechanisms for collecting data. Unfortunately for their users, the tools provide limited or no functionality for analyzing the data collected. Neither LTT nor dTrace provide analysis functionality beyond simple aggregations, which must be specified before a trace can begin. Neither tool provides a mechanism to look for patterns either during a trace or after a trace is completed.

The lack of proper kernel trace analysis tools motivates us to build a framework that applies data mining techniques to analyze kernel trace data. The framework is designed to discover information in trace logs that could not be detected with existing system tools like *top* or *ps* or be found through manual inspection of a kernel trace log. We study the pre-processing and data mining techniques that can be used to identify interesting recurring inter-process patterns in noisy kernel trace data. These execution patterns can potentially help a variety of users including system administrators, application programmers, operating systems engineers, and security analysts.

In the systems area, data mining techniques have been successfully used in the past for profiling and detecting mal-ware [19], optimizing data placement and prefetching for fast retrieval [14], and detecting operating system bugs introduced by copying and pasting of kernel source code [15]. To our knowledge, this is the first effort for a general-purpose solution that mines across multiple kernel subsystems and the first attempt to tackle the task of mining kernel trace logs.

Issues and Challenges. While data mining techniques [5, 10] have been successfully applied to mine time-series data in a variety of applications, mining kernel trace data presents a unique set of challenges.

The *complex and voluminous data* generated by kernel tracing tools create our first mining challenge. A typical kernel tracing tool, the Linux Trace Toolkit (LTT), can report thousands of kernel-event records every second. Each kernel event is a multi-attribute tuple containing, a record of which process caused the event, the sub-system of the operating system involved with executing the event, the event type, the address or descriptor for any resource accessed during the event, and the time at which the event occurred. Sample values for these attributes are shown in **Table 1**.

Further adding to the challenge is our desire to detect *complex*

Table 1. Trace Attributes with Sample Values

Attribute	Sample Values
process	Firefox, staroffice, xFree86
subsystem	file system, memory, syscall, sched
Event	open, alloc, syscall entry
descriptor	bookmarks.html, gettimeofday 2 (file descriptor), 2531 (process ID)

patterns in multiple attribute data. To uncover systemic problems we cannot focus our attention on one subsystem or application. For example, isolating a memory allocation problem would necessitate looking at information in the event attribute that shows when a log entry involves the memory subsystem and the process attribute that indicates which program generated the event.

Another important consideration for mining a kernel trace is *appropriate treatment of a trace's time-series data*. The timestamp attribute allows us to find more interesting patterns using data mining techniques than we could obtain by computing event aggregations for data in kernel trace logs. In addition, the unique scheduling characteristics of the operating system make the kernel trace mining task not a straightforward application of existing sequence mining algorithms.

Finally we must consider a way to *interpret the results* of our mining system. A main concern is that events related to systemic problems could appear in the logs with relatively low frequencies compared to normal events that appear with very high frequencies. Our system needs to identify not-too-frequent event patterns that indicate problems and report those patterns in its output. In addition, we should provide efficient ways to hide very frequent patterns that do not indicate problems.

Contributions and Organization. Bearing the above issues in mind, we design a framework for effectively and efficiently mining kernel trace logs and implement a suite of trace mining tools to test the design. Our framework makes a number of unique contributions. First, we transform the problem of kernel trace data pattern mining to maximal frequent itemset mining. We provide special treatment for the unique temporal characteristics of kernel trace data and propose a combined approach of window folding and window slicing to group trace events into itemsets using their timestamp as a measure of temporal proximity (Section 2.1). Second, we develop a set of data preprocessing techniques including *bit packing* and *data filtering* that allow efficient and flexible cross-attribute pattern mining (Section 2.2). Finally, we perform experimental studies to detect systemic problems on real systems. We test our tools with a range of algorithmic parameters, showing the feasibility and effectiveness of the approach (Section 3). We conclude the paper with a review of related work (Section 4), a brief summary, and a discussion of futures directions for our research (Section 5).

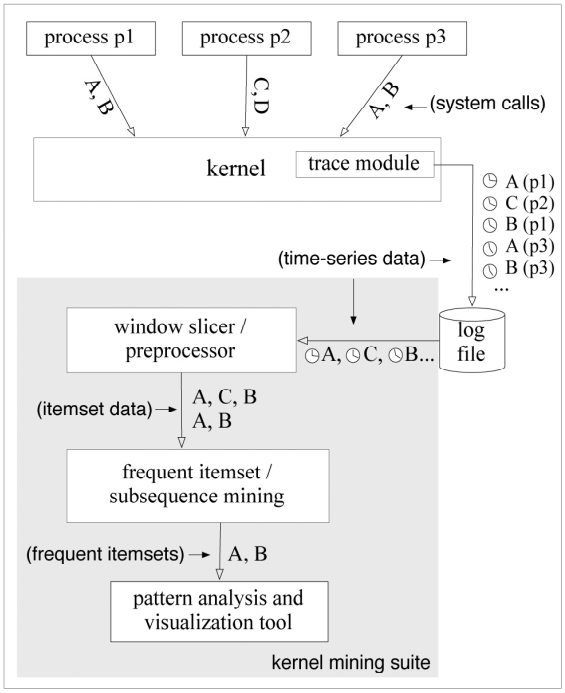

Figure 1. System Architecture

2. KERNEL TRACE MINING FRAMEWORK

Kernel trace logs are massive, ordered records of events that occur inside the operating system. Our goal is to find common execution patterns so that we can better understand the execution that is taking place on a machine. In this section, we present an overview of our framework and show how we model kernel trace mining.

We present a conceptual diagram of our framework for mining kernel data in **Figure 1.** In our system, un-modified processes make requests of the operating system in the form of system calls. A trace module inside the operating system transparently monitors these calls and other internal activity. The trace monitor writes a detailed, time-series record of these events to a log file. The preprocessing utilities in our suite harvest the time-series data from a log and translate it into itemset data for frequent itemset mining. The preprocessor output is passed into a frequent itemset mining tool and the output patterns are passed to a program for display and analysis.

2.1 Frequent Itemset Mining

Finding frequently occurring patterns in ordered or time-series data like our trace logs is a mining task commonly referred to as Sequential Pattern Mining [5]. Given an ordered series of events, *S*, and a minimum support, *min_sup*, mining for frequent sequences involves finding the set of all ordered series of events, *F*, that occur at least *min_sup* times in *S*. For example, with *min_sup* = 2 and *S* = <A, C, B, A, B> we have the frequent sequences *F* = <A, B>, <A>, <B>. These frequent subsequences are referred to as *serial episodes*.

Additional constraints may be placed on the sequential mining problem. Constraining the interval *i* puts an upper limit on the maximum time that can have elapsed between any two consecutive items in a serial episode. On an integral timescale an interval of 0 requires all serial events to have no intervening events. With our original *S, min_sup*=2, *i*=0, only the single event series <A> and <B> would be reported as frequent series; there would be no multi-itemsets in our results for this example.

Kernel trace logs have unique temporal characteristics due to the partially ordered execution resulting from OS scheduling. We discuss below why we need a special treatment of the kernel trace data compared to typical time-series data. We propose two important techniques for the treatment, namely, window folding and window slicing. Window folding creates a series of parallel events and window slicing creates a sequence database. Together these steps create a database of parallel events. As a result, we treat the parallel events as unordered and reduce the problem of sequential pattern mining to frequent itemset mining.

Window Folding. For the purposes of examining system trace data we need to maintain a sufficiently large interval to account for sequence gaps and re-orderings introduced by the operating system's scheduler. On any modern OS the scheduler will regularly suspend the execution of a process so that other processes may execute. The maximum period for which a process may execute uninterrupted by the scheduler is referred to as the timeslice. Intervals used by our mining algorithms need to be

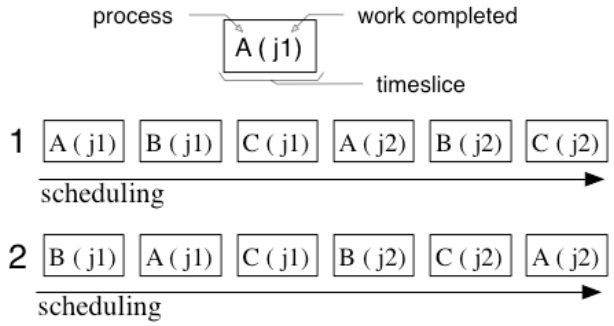

Figure 2. Alternative Scheduling of a Single Load

sufficiently long to detect patterns that exist across multiple timeslices. Certainly if we hope to find inter-application patterns we need to consider more than one process at a time. Furthermore, because the scheduler does not guarantee any ordering of process execution to processes, detecting common process interactions requires relaxing the definition of sequence mining further to allow for changes in ordering.

To illustrate this challenge, suppose jobs j1 and j2 require inter-process communication among processes A, B, and C. A and B's execution can happen in any order. Furthermore, C is dependent on B having completed its work. The two similar jobs j1, j2 could be completed according to either of the execution schedules shown in **Figure 2**. Both sequences illustrate the same pattern of collaborative work being done by the same processes but in different orders of execution. Unless we relax the strict ordering constraint enforced in traditional sequence mining, we will fail to generate any sequences demonstrating the interesting interaction between processes A, B, and C.

Relaxing the ordering constraint means that we can loose potentially useful information, such as event ordering information for frequently occurring deadlock situations. However, relaxing ordering allows us to discover problem interactions both where ordering is an issue and where ordering is not a factor. Our system could be adapted to recover lost ordering information by employing an approach presented in [16], which first identifies frequent parallel episodes and then recovers ordering information for the parallel episodes to establish frequent serial episodes.

In order to expose all potentially interesting inter-process interactions on timeshared systems we relax the ordering requirement to consider events occurring within a certain temporal distance to be parallel events. The temporal distance is referred to as a *folding window*.

Window Slicing. Our task now becomes finding frequent *parallel episodes*—frequently occurring sets of temporally proximal events. Unfortunately, representing a list of all groups of parallel events with a folding window size *w* for a sequence with *N* items leads to a growth of data O(*w***N*). Although this growth is constant, for the *w* values used in our experiments storing every parallel event window would require several hundred to several thousand times more storage and processing than the original trace representation. To avoid expanding our dataset we adopt a technique called *window slicing*.

Window slicing is introduced in [14] as an effective technique to perform sequence mining of an extended sequence. Window slicing converts an extended sequence into a sequence database, a collection of relatively shorter sequences. The non-overlapping version of *window slicing* technique divides time-series data using a window period. Events that take place in the same window are

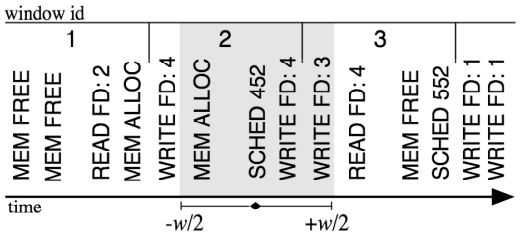

Figure 3. Windowing Slicing Error for Event SCHED 452

treated as a single sequence record in the event database. We adapt this form of window slicing to our data while simultaneously performing window folding using the same period. This slice-and-fold step gives us an unordered database of parallel events.

Without loss of generality, we consider LTT logs where timestamps are recorded as the number of microseconds elapsed since the UNIX epoch. We define a parameter *w*, the fold-slice-window period in microseconds. Using *w* we calculate a window ID for each log entry, L_x, using the formula:

$$L_x.window\_id = (L_x.timestamp - L_1.timestamp) / w$$

Any two log entries that share a window ID are treated as parallel events. The window ID serves as an itemset ID for our frequent itemset mining.

Clearly, for a window of size *w*, some event sequences of size *w* will not be included in our sequence database using this window slicing method (**Figure 3**). However by using a window long enough to capture frequent patterns at least once in most windows, we minimize the disruptive effects of window slicing. Our experimentation confirms previous work in [14] that shows window slicing will not adversely affect the effectiveness of our mining for sufficiently large windows.

Frequent Itemset Mining. By combining the window folding and window slicing methods, we can treat the parallel events as unordered and so our task of mining trace data has been reduced to frequent itemset mining.

Specifically we look for maximal frequent itemsets, the elements of which comprise common maximal execution patterns in the original trace. We use maximal frequent itemset mining (MFI) in favor of frequent closed itemset mining (FCI) and frequent itemset (FI) mining because the volume of MFI output is much lower than that produced by FCI and FI mines. This is important because we want our output to be human-readable. Furthermore frequent closed itemset mining introduces noise into our results because slightly different supports for items in a maximal frequent itemset, which exists as a result of window-slicing and scheduling, cause multiple frequent closed itemsets to be reported—none of which provide any more insight than the single maximal frequent itemset.

2.2 Data Preprocessing

Our kernel trace mining system includes a series of C++ and perl data preprocessing tools that transform time-series trace logs into a series of integers for frequent itemset mining.

Our preprocessing tools work together to perform the following tasks in the listed order: resolve operating system descriptors to file and executable names, create uniform dictionaries for attribute representation, convert log entries to normalized integer representations. We also have a tool that filters data using a pass-through filter that discards records that do not

match any rule in a user-specified list. Our rules are a series of formatted strings adhering to our attribute format *proc-subsystem-call-descriptor-size*. We allow wildcard matching for an attribute with the asterisk character. For example, the rule `*,MEM,*,*,*` allows any call to the memory subsystem to pass into the dataset we use for mining.

3. EXPERIMENTATION

We performed a set of experiments evaluating the feasibility, effectiveness and cost of our proposed framework. Our goal is to answer the following important questions: 1) Can kernel trace mining help system users in system-related tasks such as performance tuning and systems debugging? 2) How do different algorithmic parameters in our framework affect the mining results?

In this section, we report findings from a case study with our system. We use our framework to detect a known problem program, the GNOME applet *gtik*. We describe the problem and our experiment setup for the GNOME applet, present the mining results, and illustrate the effects of different algorithmic components and parameters of the mining framework. We also implemented our framework on Solaris using dTrace and conducted a separate case study that confirmed our findings from the LINUX and LTT case study. Interested readers can find details about our implementation and experience mining kernel traces on Solaris in [9].

3.1 Problem Scenario

The GNOME stock ticker applet *gtik* version 2.0 is a process now known to induce systemic problems because of the number of high-impact X programming calls it generates. However, identifying this process as a heavy consumer of system resources presents a challenge.

Because much of the work being created by the *gtik* applet takes place inside the X server, detecting the applet as the source of system overhead using traditional tools such as *top* is impossible. A recent study [2] used dTrace to identify the applet on a multi-processor Solaris system. The study used the output of *mpstat*, a tool for monitoring processor activity, as a starting point for writing a series of dScripts. dScripts provide control and analysis functionality for Solaris' dTrace kernel tracing mechanism. In all, five ad-hoc dScripts were necessary to trace the suspicious activity reported by *mpstat* back to *gtik*. These scripts required not only knowledge of X programming calls, but also implementation knowledge of an X server. Furthermore this solution using dTrace and custom dScripts was only possible because of the clues provided at the onset by *mpstat*'s cross call report, a feature that is not available on single processor machines.

Given this problem scenario, we study how the proposed kernel trace mining system can help in detecting the problematic process through pattern discovery in kernel traces.

3.2 Experiment Setup

Using the LINUX Trace Toolkit we collected traces of kernel level activity, including system calls. We mined the traces collected using our suite of data preprocessing tools and MFI mining. For MFI mining we adopt MAFIA [1], which uses a highly efficient algorithm for outputting maximal frequent patterns. We examined the output of our mining to look for meaningful patterns of activity.

Table 2. Experiment Traces

Trace	Gtik	Interactive App.
ltt_gtik_20_isolated	Buggy	No
ltt_gtik_20_noisy	Buggy	Yes
ltt_gtik_26_isolated	Fixed	No
ltt_gtik_26_noisy	Fixed	Yes

We collected four 1 minute traces, summarized in **Table 2**, from a machine running version 2.5.7 of the Linux Kernel, patched with version 0.9.5 of the Linux Trace Toolkit. Two traces were conducted while the *gtik* applet version 2.0, the version known to induce systemic problems, was running: The first trace, *ltt_gtik_20_isolated*, contained one minute of trace activity with no interactive applications running and no user activity. The second trace, *ltt_gtik_20_noisy*, was conducted while a user surfed the web using FireFox and edited a document in OpenOffice. The next two traces, *ltt_gtik_26_isolated* and *ltt_gtik_26_noisy*, were collected under similar circumstances, but with an improved version of the *gtik* applet running, version 2.6. The newer version reduced the number of high-impact X programming calls, but the program still communicated with the X server very frequently—a phenomenon we saw in our mining results.

3.3 Mining Results

We first present a set of results showing clear patterns that can be used to identify the problematic *gtik* process. The frequent itemset output from our system, along with its corresponding experimental configuration, is illustrated in **Figure 4**. The output was exceptionally clear. It consisted of only 2 frequent itemsets, with *itemset 2* directly pointing to the systemic problem on the trace machine. In the itemset the pairing of *gtik*'s writes and *XFree86*'s reads, and the two processes' complementary allocs and frees of memory suggest that *gtik* is responsible for much of the *XFree86's* work. *itemset 1* reveals no non-obvious information—we expect the *XFree86* server, which is responsible for all graphic display on the system, will be making almost constant system calls.

We also conducted experiments across all attributes with no data filtering and in each of our traces we were able to detect frequent itemsets pointing to the *gtik* and X server interaction. Even with an interactive load that consumed 40 percent of CPU, meaning the presence of considerable noise for our data mining algorithm to contend with, the *gtik* and *X11* interaction was clearly visible in our output.

Experimental Configuration
input: ltt_gtik_20_noisy; *minimum support*: .5
fold-slice-window period: 25,000 (microseconds)
filters: *,FS,OPEN,*,*; *,FS,CLOSE,*,*;
,FS,READ,,*; *,FS,WRITE,*,* ;
,MEM,,*,*; *,SCHED,*,*,*;

Output

itemset 1:	*itemset 2:*
XFree86, FS, WRITE	gtik2_applet_2, MEM, ALLOC
XFree86, FS, READ	gtik2_applet_2, MEM, FREE
XFree86, MEM, FREE	gtik2_applet_2, FS, WRITE
Xfree86, MEM, ALLOC	XFree86, FS, READ
	XFree86, MEM, FREE
	XFree86, MEM, ALLOC

Figure 4. Experimentation Input/Output

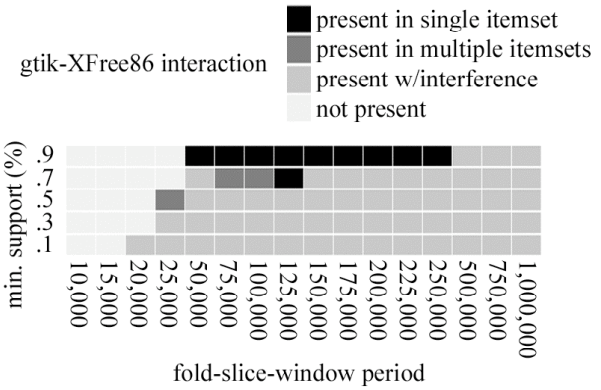

Figure 5. Variations in Mine Quality

3.4 Parameter Effects on Mining Quality

To test the durability of our system we also performed an extensive set of experiments over a range of mining parameters. We investigated the effects of two important mining parameters, fold-slice-window and minimum support. We detail results for the *ltt_gtik_20_noisy* trace in this section because it contains the noisiest data and accordingly presents the greatest challenge to our mining system.

We conducted each of our experiments using fold-slice-window periods varying from 10,000 microseconds to 1 second at discrete intervals. At each of these intervals we looked for maximal frequent itemsets with minimum supports between .1 and .9. We report 4 possible outcomes for each of these experiments: *Present in single itemset*—only 1 frequent itemset was reported and it contained the problematic interaction, *Present in multiple itemsets*—the pattern appeared in at least one frequent of the itemsets reported and no itemset contained events generated by processes other than *gtik* or *XFree86, Present with interference*—the pattern was present in an itemset that also contained noise from processes not involved in the systemic problem, *Not present*—the problematic inter-process pattern was not reported in any itemset.

Figure 5 presents the mining outcomes with varying fold-slice-window periods and varying minimum supports; darker cells contain more useful information or knowledge. For all windows ranging from 25,000 microseconds to 250,000 microseconds we could detect the problematic inter-process interaction. Minimum support played a significant role in determining result quality where the signal was strong. Where the fold-slice-window period was long—and the problem interaction between *gtik* and *XFree86* was nearly guaranteed to be reported in every window—the high minimum support value pruned noise from other processes out of the output itemsets.

At the opposite end of the spectrum, where the problem interaction was present in only a few of the short windows, a high minimum support value pruned away the interaction from the results.

We performed additional experiments to determine the effects of attribute filtering on our system. We found filtering along certain attributes resulted in more readable output. Results from these experiments are omitted because of space limitations; interested readers can refer to [9] for details.

4. RELATED WORK

Pattern mining in time series data has been an emerging technique applied in a variety of applications [5, 21]. We briefly review in this section the most relevant works that apply data mining for operating systems.

In developing a kernel-wide data-mining system, we consider research outcomes for systems targeting each of an operating system's constituent subsystems. Research has been conducted to develop better data pre-fetching at the disk and network level [3, 4]. [3] exploited an existing sequence mining algorithm for data placement optimizations. As systems become increasingly distributed and complex, mining will play an increasingly important role in evaluation and optimization. Already, traditional file-system benchmarking applications are inadequate for meaningful performance evaluation in large-scale storage area networks (SANs) [18, 20]. Mining has also been shown to be useful when optimizing data placement and prefetching from disks [13]. The tools we developed for mining kernel trace data could be adapted to analyze multiple time-sequence logs from different components of a SAN or other distributed system.

Tracking operating system activity for intrusion detection is a mature area of research [7, 8, 11, 13]. This work usually focuses on tracking individual users and processes. Mining-based approaches for system security are emerging as responsive and resilient strategies for system security [11, 12]. Mining techniques have also been successfully used for profiling and detecting mal-ware [19, 6], and for detecting operating system bugs introduced by copying and pasting of kernel source code [14].

Our work differs from above in that it is the first general-purpose solution that allows for mining for patterns across multiple subsystems in kernel traces to detect systemic problems.

Finally, directly related to the performance of our system is work on maximal frequent itemset mining [1] and sequence mining [16, 17, 23]. For future adaptations of our systems, stream mining of frequent itemsets is a key area of interest [10].

5. CONCLUSIONS AND FUTURE WORK

We developed a framework for mining kernel trace data. We translate the task of mining kernel trace data into frequent itemset mining. We experimentally show that our system detects excessive inter-process communication. Our system detects these patterns for a range of parameters and in noisy data. Furthermore, our mining system detects problem inter-processes interactions through a single analysis step—something that is impossible using existing system analysis tools.

While our work is a convincing proof-of-concept there are several components of our system we would like to measure more precisely. First, we would like to determine how much, if any, loss is introduced by our window-slicing approach versus an approach that uses overlapping windows. We would also like to measure how much performance improvement is achieved through our data filtering method.

Our research continues along several directions. While our experimental results gave guidelines in selecting parameters for yielding the most interesting patterns, we are exploring the idea of learning and building a library of normal execution (house keeping) patterns on a normal running system. These patterns

would be used for filtering mining results to generate less-noisy, interesting patterns. In addition, tighter coupling between our tools and the kernel's tracing facilities would improve performance and make for a more seamless user experience. Closer integration also would eliminate the need for much of the post-processing we perform on kernel trace logs after trace collection, before mining. Tighter integration with the operating system would make real-time, stream-based analysis of a system possible. Finally, we would like to apply other mining algorithms to kernel data to see if they can be used to discover additional meaningful patterns or deliver results faster.

6. REFERENCES

[1] Burdick, D., Calimlim, M., Flannick, J., and Yiu, T. 2005. MAFIA: A Maximal Frequent Itemset Algorithm. *IEEE Transactions on Knowledge and Data Engineering* 17, 11, 1490–1504.

[2] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. 2004. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference 2004*, 15–28.

[3] Cao, P., Felten, E. W., Karlin, A. R., and Li, K. 1995. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95/PERFORMANCE '95)*, 188–197.

[4] Griffioen, J. and Appleton, R. 1994. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference*, 197–207.

[5] Han, J. and Kamber, M. 2006. *Data Mining: Concepts and Techniques, 2nd ed.*. Morgan Kaufmann Publishers, San Francisco, CA.

[6] Kolter, J. Z. and Maloof, M. A. 2004. Learning to detect malicious executables in the wild. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '04)*, 470–478.

[7] Lane, T. and Brodley, C. E. 1997. Sequence matching and learning in anomaly detection for computer security. In *AAAI Workshop: AI approaches to Fraud Detection and Risk Management*, 43–49.

[8] Lane, T. and Brodley, C. E. 1998. Temporal sequence learning and data reduction for anomaly detection. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS '98)*, 150–158.

[9] LaRosa, C., Xiong, L., and Mandelberg, K. 2007. *Frequent Pattern Mining for Kernel Trace Data*. Technical Report TR-2007-022, Department of Mathematics and Computer Science, Emory University, Atlanta, GA.

[10] Lee, D. and Lee, W. 2005. Finding Maximal Frequent Itemsets over Online Data Streams Adaptively. In *Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM '05)*, 266–273.

[11] Lee, W. Stolfo, S., and Chan, P.K. 1997. Learning Patterns from Unix Process Execution Traces for Intrusion Detection. In *AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, 50–56.

[12] Lee, W. Stolfo, S., and Mok, K. 1999. A Data Mining Framework for Building Intrusion Detection Models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 120–132.

[13] Lee, W., Stolfo, S. J., and Mok. K. W. 2000. *Artificial Intelligence Review* 14, 6, Kluwer Academic Publishers, 533–567.

[14] Li, Z., Chen, Z., Srinivasan, S. M., and Zhou, Y. 2004. C-Miner: Mining Block Correlations in Storage Systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '07)*, 173–186.

[15] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. 2004. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI '04)*, 289–302.

[16] Mannila, H., Toivonen, H., and Inkeri Verkamo, A. 1997. Discovery of Frequent Episodes in Event Sequences, *Data Mining and Knowledge Discovery* 1, 3, 259–289.

[17] Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., and Hsu, M-C. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In *Proceedings of the 17th International Conference on Data Engineering* (*ICDE '01*), 215–224.

[18] Ruwart, T. M. 2001. File System Benchmarks, Then, Now, and Tomorrow. In *Proceedings of the Eighteenth IEEE Symposium on Mass Storage Systems and Technologies (MSS '01)*, 117.

[19] Schultz, M. G., Eskin, E., Zadok, E., and Stolfo, S. J. Data Mining Methods for Detection of New Malicious Executables. 2001. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy (SP '01)*, 178–184.

[20] Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R. 2006. Stardust: tracking activity in a distributed storage system. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06)*, 3–14.

[21] Wang, W. and Yang, J. 2005. *Mining Sequential Patterns from Large Data Sets (The Kluwer International Series on Advances in Database Systems)*. Springer-Verlag, New York, Inc.

[22] Yaghmour, K. and Dagenais, M. R. 2000. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*, 13–26.

[23] Yan, X., Han, J., and Afshar, R. 2003. CloSpan: Mining Closed Sequential Patterns in Large Datasets, In *Proceedings of the 2003 SIAM International Conference on Data Mining (SDM '03)*.