Fmeter: Extracting Indexable Low-level System Signatures by Counting Kernel Function Calls

Tudor Marian[1], Hakim Weatherspoon[2], Ki-Suh Lee[2], and Abhishek Sagar[3]

[1] Google
[2] Cornell University
[3] Microsoft Corp.

Abstract. System monitoring tools serve to provide operators and developers with an insight into system execution and an understanding of system behavior under a variety of scenarios. Many system abnormalities leave a significant impact on the system execution which may arise out of performance issues, bugs, or errors. Having the ability to quantify and search such behavior in the system execution history can facilitate new ways of looking at problems. For example, operators may use clustering to group and visualize similar system behaviors. We propose a monitoring system that extracts formal, indexable, low-level system signatures using the classical vector space model from the field of information retrieval and text mining. We drive an analogy between the representation of kernel function invocations with terms within text documents. This parallel allows us to automatically index, store, and later retrieve and compare the system signatures. As with information retrieval, the key insight is that we need *not* rely on the semantic information in a document. Instead, we consider only the statistical properties of the terms belonging to the document (and to the corpus), which enables us to provide both an efficient way to extract signatures at runtime and to analyze the signatures using statistical formal methods. We have built a prototype in Linux, Fmeter, which extracts such low-level system signatures by recording all kernel function invocations. We show that the signatures are naturally amenable to formal processing with statistical methods like clustering and supervised machine learning.

Keywords: information retrieval, term-frequency inverse document frequency, indexable system signatures

1 Introduction

System monitoring is key to understanding system behavior. Developers and operators rely on system monitoring to provide information necessary to identify, isolate, and potentially fix performance bottlenecks and hidden bugs. Unfortunately, as computer systems become increasingly complex, understanding their execution behavior to identify such performance bottlenecks and hidden bugs has become more difficult. Furthermore, large scale system deployments, like the present-day datacenters that power cloud services, require increasingly complex automatic system monitoring infrastructures [1–3].

One issue is that existing monitoring solutions have not been designed to enable the extraction of low-overhead, low-level, system signatures that are sufficiently expressive

to be used in automatic analysis by formal methods. For example, instruction level monitoring in software and breakpoint debugging incur prohibitive overheads; system call tracing is both expensive and not expressive enough; hardware counters by themselves provide little amounts of specialized information while hardware counter assisted profiling is not expressive enough since it relies on sampling. By contrast, high-level metrics, like the number of completed transactions per second are overly general and application specific, and are unable to capture with sufficient fidelity low-level system behavior.

Another issue is that few monitoring solutions provide a systematic and formal way to leverage past diagnostics in future problem detection and resolution [4]. Instead, system monitoring has traditionally been performed in an ad-hoc fashion, using anything from `printf/printk` statements, debuggers, operating system process tracers, runtime instrumentation [5], to logging libraries, kernel execution tracing [6], low-level hardware counters [7, 8], generalized runtime statistics [9, 10], and system call monitoring [11] to name a few.

In this paper we introduce Fmeter—a novel monitoring technique that efficiently extracts *indexable* low-level system descriptions, or signatures, which accurately capture the state of a system at a point in time. Every low-level signature is essentially a feature vector where each feature roughly corresponds to the number of times a particular operating system's kernel function was invoked. Fmeter draws inspiration from the field of information retrieval, which showed that counting words in a document is sufficiently powerful to enable formal manipulations of document corpora. Likewise, Fmeter does not rely on any additional contextual information, like call stack traces, function parameters, memory location accesses, and so on.

By construction, embedding kernel function calls into the vector space model [12] yields formally indexable signatures of low-level system behavior. Developers and operators can automatically analyze system behavior using conventional statistical techniques such as clustering, machine learning, and similarity based search against a database of previously labeled signatures. For example, Fmeter enables operators to instrument entire datacenters of production-ready machines with the flip of a switch, and provides a way to automatically diagnose problems. At the very least, Fmeter enables operators to prune out the space of potential problems. By contrast, expending human expertise to perform forensic analysis in such an environment on a large number of individual systems is intractable.

Fmeter occupies a new point in the design space of monitoring systems that yield low-level system signatures. Unlike low level statistical profilers (e.g., Oprofile [7]) which only capture the most frequent events in their event space, Fmeter records every single kernel function invocation, therefore there are no events that fly under the radar—as long as they belong to Fmeter's event space to begin with. This is an important feature of Fmeter since bugs typically reside in cold code. (Section 2.1 formally defines what is the precise contribution of each kernel function invocation count to a signature.) Moreover, Fmeter signatures are insensitive to nondeterminism and are machine independent.

Since Fmeter does not need to collect any detailed contextual information (like entire stack traces), generating and retrieving signatures can be more efficient than general-purpose function tracers. As we demonstrate in Section 4, we leverage this

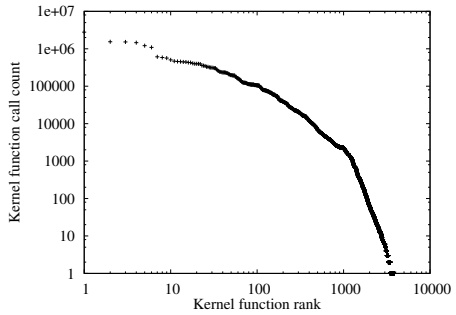


Fig. 1. Kernel function call count during boot-up.

knowledge of the problem domain to render the Fmeter prototype more efficient than the default Ftrace [6] kernel function tracer. Like Ftrace, Fmeter has virtually zero runtime overhead if it is not enabled. However, unlike the Ftrace function tracer, Fmeter does not collect any additional semantic information with each function call. The Fmeter runtime overhead introduced by signature generation is sufficiently low that signature generation can be turned on at production time for long continuous periods of time. Generating and logging signatures over such long continuous time intervals increases the likelihood of success of post-mortem analysis of crashed systems.

Our contributions are as follows:

- We provide a novel method for extracting indexable low-level system signatures by embedding kernel function calls into the vector space model. The signatures are naturally amenable for formal statistical manipulations, like clustering, machine learning / classification, and similarity based search.
- We introduce Fmeter—an efficient prototype implementation of a monitoring system capable of generating and retrieving the low-level system signatures continuously over long periods of time, in real-time, and with little overhead.
- We show that the signatures are sufficiently powerful to capture meaningful low-level system behaviors which can be accurately classified by conventional unsupervised and supervised machine learning techniques. Furthermore, the signatures are also sufficiently precise for automatic classifiers to unambiguously distinguish even between system behaviors that differ in subtle ways.

The rest of the paper is structured as follows. Section 2 discusses our motivation, insight, approach, and challenges for creating an indexable signature via embedding kernel function calls into a classical vector space model. We describe our Fmeter design and implementation in 3. In Section 4, we evaluate Fmeter and our proposed approach. We discuss limitations to our approach and design in Section 5. Finally, we discuss related work and conclude in Sections 7 and 8.

2 Methodology

To extract meaningful, low-level system signatures that are also formally indexable, we turned to the discipline of information retrieval (IR) and text mining for inspiration. The

information retrieval community has had a long and proven track record of developing successful statistical techniques for automatic document indexing and retrieval. In particular, the IR discipline has shown that simple statistics computed over the document's terms are sufficiently powerful to yield information which can be formally analyzed. For example, search engines typically throw away semantic information (e.g., they do not parse sentences and paragraphs) and use term frequencies mechanically for scoring and ranking a document's relevance given an input query.

Like the frequency of words in documents, function invocations appear to follow a power-law like distribution. Figure 1 shows invocation counts of 3815 functions of the Linux kernel version 2.6.28 invoked on a Dell Power Edge R710 four way quad core x86 Nehalem platform from the late boot-up stage until the login prompt was spawned. It shows that some functions are called more frequently than others. This behavior is also consistent with the role of instruction-caches in exploiting temporal locality of code. Such heavy-tailed distributions have been observed often in the real-world. A classic example of such a power-law is the distribution of wealth in the world, the distribution in rankings of U.S. cities by population, and the distribution of document terms in a large corpus of natural language [13]. For example, the word frequency in the whole of Wikipedia [14], reported on November 27, 2006, follows a shape similar to that of Figure 1. Such distributions have been thoroughly analyzed by statisticians, economists, computer scientists and mathematicians alike, and various analytical modes have been proposed—e.g., power-laws can be mathematically modeled by preferential attachment, also referred to as the "rich get richer" effect.

2.1 Low-level System Signatures

Our key insight is that we extract low-level system signatures by mapping the concepts of information retrieval and text mining to system behavior. In our model, the information retrieval concept of a "term" corresponds to a kernel function call, while the concept of a "document" corresponds to a period of low-level system activity, or function calls, over a predetermined period of time. (The kernel function calls should *not* be confused with the system calls exported by the kernel through it's application binary interface.) The "corpus" then corresponds to a collection of low-level system activities. Like in the classical vector space model [12], we disregard the semantic information in a document and consider only the statistical properties of the terms belonging to the document (and to the corpus). In our case, we disregard the sequence of kernel function calls (the "call stack" trace), the function parameters, memory location accesses, or hardware device state manipulation. Instead, we rely solely on counting the kernel function invocations, which is significantly cheaper and introduces less overhead.

We use the term frequency-inverse document frequency (`tf-idf`) model to represent documents, and thus system signatures, as weight vectors. The weight vector for a document j is $\mathbf{v_j} = [w_{1,j}, w_{2,j}, \ldots, w_{N,j}]^T$ where N is the number of "terms," i.e., the total number of kernel functions. Each weight $w_{i,j} = tf_{i,j} \times idf_i$, or the product of the term frequency, and the inverse document frequency. The term frequency is given by: $tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$ where $n_{i,j}$ is the number of times the term (function) i appears (was called) in document (during the monitoring run) j. Essentially the term frequency counts the

number of times a term appears in a document, and normalizes it by the size of the document. Normalization is required to prevent bias towards longer documents (or in our case towards longer runs) which would implicitly have a higher term count by sheer virtue of their length (duration of execution).

The inverse document frequency is used to diminish the weight of terms that occur very frequently in the entire corpus, which is the case for example with prepositions in text documents, or multiplexed functions like the `ioctl, ipc` and `execve` system calls, or virtual memory management internal routines during the boot-up phase (the top ranked kernel functions as seen in Figure 1). The inverse document frequency is computed as: $idf_i = \log \frac{|D|}{1+|\{d:i\in d\}|}$ where $|D|$ is the size of the corpus, or in our case the number of monitored low-level system activities, and the term $|\{d : i \in d\}|$ represents the number of documents containing the term i.

Fmeter collects low-level system *signatures* as weight vectors $\mathbf{v_j}$ (for each signature j), by counting the number of times each kernel function was called during a given time-interval. More precisely, the set of distinct kernel functions induce the orthonormal basis for the weight vectors $\mathbf{v_j}$. Each distinct kernel function corresponds to one of the unit-vectors, i.e., versors, that together span the space in which every system signature is defined to be a point.

Since each signature is represented as a vector belonging to the same vector space, we can express signature similarities as the similarity between the vectors. One such measure is the cosine similarity between two vectors—the cosine of the angle between the two vectors: $\cos\theta = \frac{\mathbf{x}\cdot\mathbf{y}}{||\mathbf{x}||\,||\mathbf{y}||}$; $||\cdot||$ is a vector norm and $\mathbf{x}\cdot\mathbf{y}$ the dot product between the two vectors. Alternatively, one may specify a *distance metric*, like the Minkowski distance induced by the L_p norm: $d_p(\mathbf{x},\mathbf{y}) = (\sum_i |x_i - y_i|^p)^{\frac{1}{p}}$. Unless specified otherwise, throughout this paper we compare vectors using the Euclidean distance, i.e., the distance metric induced by the L_2 norm. Furthermore, certain formal methods require we normalize the vectors, in which case we rely on the L_2 norm as well.

Fmeter retrieves such formal, indexable, low-level system signatures by embedding kernel function invocations into the classical vector space model [12]. Our approach was inspired by the information retrieval and text mining literature. By broadly ignoring the semantics of "documents," we balance the delicate act of constructing effective low-level signatures while incurring low signature retrieval overhead, and in the process we gain the opportunity to manipulate the signatures using conventional statistical tools.

2.2 Statistical Data Analysis

The low-level system signatures collected by Fmeter are indexable, hence they can be manipulated by formal data analysis methods like unsupervised and supervised machine learning, similarity based search, and so on.

Clustering is a typical unsupervised learning technique that groups together vectors (and therefore low-level system signatures) that are naturally close to each other, or similar, based on a given distance metric. When used in conjunction with system signatures, clustering can identify similar low-level behaviors. A typical clustering algorithm also returns the *centroid* of each grouping assignment. The centroid of a cluster of signatures can then be used as a *syndrome* which characterizes a manifestation of a common

behavior, e.g., an undesired behavior. Clustering can therefore be used to detect system behaviors which are similar to past pathological behaviors or previously encountered problems. A key property of clustering is that it allows for unknown behaviors to be classified as similar to some syndrome *S*, even though the unknown behaviors may belong to a distinct class of their own (i.e., clustered together, the unknown signatures yield a centroid which is closest to *S*). Section 4.2 contains our evaluation of Fmeter signature clustering.

Unlike unsupervised learning methods like clustering, supervised learning requires labeled training data to construct a predictive model. The model is subsequently used to make predictions about unlabeled data. For example, if an operator has access to a labeled training data set containing both signatures of buggy / compromised device driver behavior and signatures of normal behavior exercised by a correct device driver, future unlabeled instances of buggy device driver behavior may be identified by a classifier. Section 4.2 contains a detailed evaluation of such machine learning using Fmeter signatures as training, validation, and test data.

We envision an environment in which an operator has access to a database of labeled low-level system signatures describing many instances of normal and abnormal behavior, and perhaps the necessary steps to remedy problems. The signatures are retrieved and stored from systems whose behavior has been forensically identified and labeled. For example, signatures can be retrieved from systems that operate within normal parameters, as well as from systems that have been identified to exert certain bugs, performance issues, and any unwanted behavior (like the system reacting to a denial of service attack or a system being compromised and acting as a spam-bot and so on). Once the root cause of the problem is found for some abnormal behavior, Fmeter can then be used to generate a large number of `tf-idf` signatures with low overhead. These signatures are subsequently labeled appropriately, and stored in the database for future training references by classifiers. Likewise, signatures can be clustered to obtain syndrome centroids. By labeling similar vectors and syndrome centroids with semantic meaning, an operator may later determine automatically whether a system has some property or is behaving in an undesired fashion.

Interestingly, clustering may also be applied recursively. Applying meta-clustering on the retrieved cluster centroids, we can determine which entire classes, not just individual signatures of behaviors, are similar to one another. If two classes of system behaviors are similar with respect to their `tf-idf` signatures, it means they are similar in the way they invoke the kernel's functions. We can therefore schedule concurrently executing tasks that rely on the same kernel code-paths (and implicitly the same in-kernel data-structures) on cores that share a cache domain (e.g., the L3 cache for an Intel Nehalem microarchitecture). For a monolithic kernel (the only kind we instrument with Fmeter) such an assignment boosts performance due to improved cache locality while executing in kernel-mode [15]. For example, Fmeter logging over large time intervals would enable such a cache-aware task assignment feedback loop; as shown in Section 4.1, Fmeter signature retrieval and logging is sufficiently cheap to render such logging feasible (and can be switched on and off at runtime).

3 Extracting Signatures

Instrumenting every existing application to count all possible function calls is unrealistic. Instead, we only instrument the operating system kernel, since all applications depend on it to varying degrees. User-mode applications typically request services from the kernel through a well defined application binary interface (ABI). Fmeter reduces the size of the possible feature-space by limiting its dimensionality to a subset that is both manageable and contains significant low-level information. However, unlike statistical (kernel and otherwise) profilers, Fmeter triggers upon every kernel function call. Fmeter keeps track of how many times each kernel function is called, and exports this information to user-space through the debugfs [16] file system interface.

Since function names are not sufficient as unambiguous identifiers (e.g., a kernel may have duplicate static functions), we identify kernel functions by their start address. Absolute addresses work since unlike relocatable code, the kernel symbols are loaded at the same address across reboots, however, using addresses means that the signatures are not valid across different kernel versions. We consider this limitation to be minor given the target Fmeter environment, namely that of compute clouds which run a small number of managed virtual machine / bare-metal kernels. Function symbols that reside in runtime loadable modules introduce further complications since modules are relocated at load time. Initially Fmeter identified functions in modules using a tuple comprising of the module name, version, and function offset within the module. However, we observed that different version drivers may contain mostly the same code (confirmed after we compared disassembled modules one function at a time) but adding even the slightest modifications at some point in the module changes all subsequent offsets. Therefore we decided that Fmeter does not instrument functions that live within runtime loadable kernel modules, and signatures will only capture the behavior of modules by virtue of the calls the modules make into the core-kernel. This means that Fmeter effectively reduces the dimensionality of the feature space, a technique that is commonly used throughout machine learning (e.g., to select only the most meaningful features based on principal component analysis, and prune out the otherwise low-impact features).

The Linux kernel already provides several facilities to intercept and execute ad-hoc handlers when kernel functions start or finish executing. For example, the Kernel Dynamic Probes (Kprobes) [5] subsystem may be used to graft breakpoint instructions at runtime, and call into implanted handler routines (these routines may live in runtime loadable modules as well, hence new ones can be coded as needed). Unlike Kprobes, which incur the runtime overhead of inserting a breakpoint, executing the handler, and single-stepping through the breakpointed instruction, the Ftrace [6] infrastructure shifts most of the overhead at kernel compile time and during the kernel boot-up phase. In particular, when compiled with `gcc`'s `-pg` flag, all kernel functions are injected with a call to a special `mcount` routine, a technique similar to the way in which the ATOM [17] platform converted a program into its own profiler. The `mcount` routine must be implemented in assembly because the call does not follow the conventional C–ABI. During kernel boot-time, the `mcount` call sites are iterated over and recorded in a list, and are subsequently converted into noops. The saved list can later be used at runtime to dynamically and selectively convert any of the call-sites back into trace calls.

Currently, Ftrace implements several tracers in this manner, e.g., a function call tracer to trace all kernel functions, a function graph tracer that probes functions both upon entry and exit hence providing the ability to infer call-graphs, a tracer of context switches and wake-ups between tasks, and so on. Since the Ftrace subsystem supports a large variety of tracers, it encompasses a general purpose machinery that generically logs retrieved data to user-space through the debugfs interface. More precisely, Ftrace relies on large fixed size circular buffers to store traced information, and individually recorded information has variable size (e.g., function traces and call-graphs). Moreover, the circular buffer management is fairly complex since it has to be accessed in an SMP-safe fashion to protect against concurrent updates since the kernel executes concurrently on all available processors. Although the Ftrace circular buffer available in the kernel version we started with (version 2.6.28) was deemed to be somewhat lock-heavy [18] with impact on performance, there have since been various attempts to replace it with a wait-free alternative [18, 19]. Wait-free FIFO buffers [20, 21] are difficult to prove correct and are prone to subtle race-conditions and errors, which is why their adoption into the mainline Linux kernel has been slow.

Since Ftrace is not extensible, i.e., new tracers cannot be added in a non-invasive way, we implemented the Fmeter tracing to rely only on the `mcount` kernel functionality and did not make use of the conventional ring-buffers. Instead, we constructed an efficient data structure which takes advantage of the structure of the monitored data to further reduce overheads. Conceptually, Fmeter requires only a small, fixed size array that maps kernel function address to an integer value denoting invocation count. Fmeter creates this mapping at boot-time, right after the kernel introspects itself and records the `mcount` sites for all traced kernel functions. To access and update the mapping during normal operation, we provide a specialized `mcount` routine.

The function-to-invocation map is slightly more involved. Fmeter actually maintains a set of per-CPU indices, each index mapping a kernel function to a cache aligned 8 byte integer value. The integer value is incremented each time the corresponding function is invoked while running on the current CPU. Each per-CPU index is allocated as a series of free pages, and each page contains an array of "slots." Before a kernel function executes for the first time, the `mcount` routine is invoked. Our specialized `mcount` routine replaces the call site that triggered its call with a call to a custom-built stub for the original kernel function whose preamble invoked `mcount` in the first place. There will be one such stub dynamically created by the specialized `mcount` routine for every instrumented function. All subsequent calls to the instrumented kernel function will execute the custom, personalized stub from then on.

The custom stub for each kernel function is generated by embedding two indices into the stub code itself. The first index identifies the page in the page list which constitutes the per-CPU data buffer. The second index identifies the corresponding slot on the selected page corresponding to the invoked function. The indices are generated at boot-time, when the mappings between function addresses and invocation counts are allocated. When invoked, each individual stub disables preemption to prevent the current task from being scheduled out and potentially moved on a different CPU, follows the mapping by way of the two embedded indices, increments the corresponding invocation count, and re-enables preemption before returning.

Enabling and disabling preemption is a cheap operation that amounts to integer arithmetic on a value in the current task's process control block. It is cheaper than atomic operations like the `lock;inc` instructions used by the Linux kernel spinlocks and cheaper than compare-and-swap instructions used, for example, by wait-free circular buffers. Note that lock-free constructs do not absolve such atomic operations from generating expensive cache-coherency traffic over the cross-core interconnect.

A user-space daemon periodically reads the function invocation counts from debugfs and logs them to disk. The normalizing step during the `tf-idf` score computation ensures that the collection period does not have a major influence on the signatures; though it can be configured. The logging daemon reads all kernel function invocation counts twice (before and after the time interval) and computes the difference which is later transformed into `tf-idf` scores, once an entire corpus is generated.

4 Evaluation

We begin our evaluation by measuring the overhead introduced by Fmeter. To quantify the overhead, we perform a set of micro- and macro-benchmarks. We then proceed to show the efficacy of statistical data analysis methods. We employ unsupervised (clustering) and supervised (classification) machine learning techniques to retrieve information and to monitor system behavior.

Throughout our experiments we use a Dell PowerEdge R710 server equipped with a dual socket 2.93GHz Xeon X5570 (Nehalem) CPU. Each CPU has four cores and 8MB of shared L3 cache, and is connected through its private on-chip memory controller to 6GB of RAM, for a total of 12 GB of cache-coherent NUMA system memory. The Nehalem CPUs support hardware threads, or hyperthreads, hence the operating system manages a total of 16 processors. The R710 machine is equipped with a Serial Attached SCSI disk and two Myri-10G NICs, one CX4 10G-PCIE-8B-C+E NIC and one 10G-PCIE-8B-S+E NIC with a 10G-SFP-LR transceiver; the server is connected back to back to an identical twin R710 server (the twin server is only used during experiments involving network traffic). The R710 server runs a vanilla Linux kernel version 2.6.28 in three configurations: with the Ftrace subsystem disabled, with the Ftrace function tracer turned on, and patched with Fmeter instead of Ftrace respectively.

4.1 Micro- and Macro-benchmarks

This section demonstrates the overhead of using Fmeter while deployed to monitor systems in-production. We compare against a vanilla kernel with Linux Ftrace function tracer turned both on and off. When Ftrace is turned off the overhead is zero, whereas if it is turned on, recording every kernel function call incurs additional overhead. Kernel functions are behind all system calls which applications use, they are responsible for handling events, like interrupts, and they are also directly called by kernel threads. Fmeter implements its own technique of utilizing the `mcount` call to record data in dedicated per-CPU data slots while incurring low overhead. By contrast, the Ftrace collection mechanism is more involved, since more information is recorded, e.g. function call-traces, and passed to user-space.

Test	Baseline μs	Ftrace μs	Fmeter μs	Slowdown Ftrace	Fmeter	Ratio
AF_UNIX sock stream latency	4.828 ± 0.585	27.749 ± 2.649	7.393 ± 0.867	5.748	1.531	3.753
Fcntl lock latency	1.219 ± 0.209	6.639 ± 0.039	3.024 ± 0.649	5.446	2.481	2.195
Memory map linux.tar.bz2	206.750 ± 0.590	1800.520 ± 4.486	317.125 ± 1.368	8.709	1.534	5.678
Pagefaults on linux.tar.bz2	0.677 ± 0.008	3.678 ± 0.008	0.866 ± 0.009	5.433	1.279	4.249
Pipe latency	2.492 ± 0.010	12.421 ± 0.042	3.201 ± 0.081	4.985	1.285	3.881
Process fork+/bin/sh -c	1446.800 ± 18.678	6421.000 ± 11.124	1831.590 ± 7.546	4.438	1.266	3.506
Process fork+execve	672.266 ± 6.663	3094.380 ± 14.093	847.289 ± 3.227	4.603	1.260	3.652
Process fork+exit	208.914 ± 6.951	1116.800 ± 10.880	268.275 ± 1.910	5.346	1.284	4.163
Protection fault	0.185 ± 0.009	0.607 ± 0.011	0.286 ± 0.006	3.280	1.544	2.125
Select on 10 fd's	0.231 ± 0.001	1.410 ± 0.001	0.277 ± 0.001	6.110	1.199	5.096
Select on 10 tcp fd's	0.261 ± 0.001	1.798 ± 0.004	0.326 ± 0.001	6.897	1.251	5.512
Select on 100 fd's	0.897 ± 0.002	9.809 ± 0.001	1.321 ± 0.008	10.941	1.474	7.424
Select on 100 tcp fd's	2.189 ± 0.002	26.616 ± 0.242	3.308 ± 0.023	12.160	1.511	8.046
Semaphore latency	2.890 ± 0.072	6.117 ± 0.236	2.084 ± 0.062	2.117	0.721	2.936
Signal handler installation	0.113 ± 0.000	0.280 ± 0.000	0.127 ± 0.001	2.473	1.119	2.209
Signal handler overhead	0.909 ± 0.010	3.124 ± 0.009	1.072 ± 0.005	3.435	1.179	2.914
Simple fstat	0.100 ± 0.001	0.852 ± 0.006	0.145 ± 0.002	8.550	1.458	5.864
Simple open/close	1.193 ± 0.004	11.222 ± 0.019	1.873 ± 0.014	9.410	1.571	5.991
Simple read	0.101 ± 0.000	1.196 ± 0.007	0.171 ± 0.000	11.893	1.701	6.990
Simple stat	0.721 ± 0.002	7.008 ± 0.021	1.067 ± 0.012	9.720	1.480	6.567
Simple syscall	0.041 ± 0.000	0.210 ± 0.000	0.053 ± 0.000	5.156	1.303	3.958
Simple write	0.086 ± 0.000	1.012 ± 0.004	0.130 ± 0.001	11.723	1.511	7.759
UNIX connection cost	15.328 ± 0.057	81.380 ± 0.260	21.919 ± 1.339	5.309	1.430	3.713

Table 1. LMbench: Linux kernel in vanilla configuration, with Ftrace function tracer on, and with Fmeter on.

Table 1 shows the overhead incurred by Ftrace and Fmeter with respect to a vanilla un-instrumented kernel during the lmbench [22] micro-benchmark (the results represent average latencies in μs along with standard error of the mean). Overall, Fmeter incurs significantly less overhead than Ftrace. At best, Ftrace is as little as 2.125 times slower than Fmeter, whereas in the worst case Ftrace it is as high as 8.046 times slower than Fmeter. On average, Fmeter is 1.4 times slower than a vanilla kernel, whereas Ftrace is about 6.69 times slower than the un-instrumented kernel. It is important to note that lmbench tests exert unusual stress on very specific kernel operations by executing them in a busy-loop which is uncommon and typically considered an anomaly in real-world production-ready environments.

Configuration	Requests per second	Slowdown
vanilla	14215.2 ± 69.6931	0.00 %
fmeter	10793.3 ± 77.7275	24.07 %
ftrace	5524.93 ± 33.4601	61.13 %

(a)

	Unmodified	Ftrace	Fmeter
real	57m8.961s	89m56.821s	56m43.264s
user	47m50.175s	49m5.492s	46m24.890s
sys	7m59.642s	41m31.300s	9m45.817s

(b)

Table 2. (a) `apachebench` scores, vanilla (un-instrumented) kernel, Ftrace kernel function tracer on, and with Fmeter on; (b) Linux kernel compile time.

Table 2(a) displays the results of a HTTP server macro-benchmark. We used the standard `apachebench` tool, which was configured to send 512 concurrent connections (1000 times in closed-loop for a total of 512000 requests) and we used a single 1400 byte HTML file as the target served by the apache httpd web server. The apache HTTP server and the `apachebench` client ran both on the same machine to eliminate any network-induced artifacts. All tests were conducted 16 times for each configura-

tion, and we report the average along with the standard error of the mean. The Table shows a 24% slowdown in the number of requests completed per second for Fmeter and a 61% slowdown for Ftrace. As with lmbench, the test stresses the system to magnify overheads by issuing a large number of concurrent connections.

Finally, Table 2(b) depicts the time elapsed while compiling the Linux kernel, as reported by the `time` utility (not the bash `time` command), atop various configurations. As expected, the time spent in user-mode (under the row labeled `user`) is roughly the same irrespective if a vanilla kernel is used, or whether one of the Ftrace function tracer or the Fmeter subsystems are enabled instead. However, unlike user-mode code which is not instrumented, the kernel code is, and the numbers shown in the Table (under the row labeled `sys`) reveal that while Fmeter slows down the kernel compilation by about 22%, Ftrace slows it down by no less than 420%, i.e. it is 5.2 times slower. The numbers are consistent with the Fmeter and Ftrace design which only rely on the instrumentation of the kernel code-paths. In general, applications that rely little on the operating system's kernel functionality, e.g., those applications that issue few system calls (like the scientific programs that crunch numbers), would show a lower overhead. However it also implies that there are less opportunities for meaningful system signatures to be collected by Fmeter when such applications are running, thereby reducing the efficacy of our system profiling methodology altogether.

4.2 Clustering and Supervised Machine Learning

Next we show the amenability of signatures retrieved with Fmeter towards statistical data analysis techniques. We extract the signatures while performing workloads in a controlled environment. First, we show that supervised machine learning can be applied to distinguish with high accuracy amongst the signatures extracted while performing three different workloads. Second, we evaluate the efficacy of the same machine learning classifiers in distinguishing between highly similar behaviors, as induced by subtle modifications in the code of a network interface device driver. The device driver resides in an un-instrumented kernel module, hence the signatures retrieved only account for the core-kernel functions the driver calls *into* (i.e., none of the functions of either driver are instrumented). Our assumption is that such subtle device driver modifications are characteristic of compromised or buggy systems which are otherwise exceedingly hard to forensically analyze.

And third, we show that signatures retrieved during the same workloads can be automatically clustered together and accurately distinguished from signatures belonging to different workloads. We employ the same set of signatures used to previously evaluate the supervised machine learning. Since clustering is an unsupervised learning method, system operators may rely on it to identify specific behaviors without having access to labeled signatures. Operators may categorize whether a particular behavior of interest is similar to a previously observed syndrome by comparing the behavior's signatures with syndrome signatures.

Throughout the evaluation we employ conventional, though state-of-the-art, machine learning algorithms and information retrieval measurement techniques.

Supervised Machine Learning First we show how supervised machine learning can distinguish with high accuracy between Fmeter signatures corresponding to different system behaviors. We then proceed to evaluate the efficacy of machine learning classifiers in distinguishing between highly similar system behaviors—as induced by subtle modifications in the code of a network interface controller's device driver which resides in an un-instrumented kernel module. For the former experiment, we collected a set of signatures from three different tasks in a controlled fashion. The tasks in question were:

- kernel compile (`kcompile`)
- secure copy of files over the network (`scp`)
- dbench disk throughput benchmark (`dbench`)

All three tasks ran on the same system—our Dell PowerEdge R710 server—without interference from each-other. The Fmeter logging daemon collected the signatures every 10 seconds. For every workload type we retrieved roughly 250 distinct signatures, which we subjected to our machine learning methods.

There are many available types of supervised classifiers one can use, e.g. decision trees, Neural Networks, Support Vector Machines (SVMs), Gaussian mixture models, and naïve Bayes, not to mention ensemble techniques that combine one or more classifiers of the same (e.g., bagging and boosting of decision trees) or different type to perform classification. Based on our previous experience, we chose to use the SVM^{light} [23, 24] classifier, which is an implementation of Vapnik's Support Vector Machine [25]. We are considering experimenting with a hand-crafted C4.5 decision tree package that supports high dimension vectors and is capable of performing boosting and bagging.

In a nutshell, SVMs construct a hyperplane that separates the vectors in the training set such that the separation margin is maximized (i.e., the hyperplane is chosen such that it has the largest distance to the nearest training data points of any class). Since the vectors in the training example may not be linearly separable by a hyperplane in the vector-space defined by the features, SVMs rely on kernel-functions (not to be confused with the operating system's in-kernel functions traced by Fmeter) to construct the hyperplane in a higher dimensional space. Classifying is performed in a straightforward manner, simply by determining on which "side" of the hyperplane an example point/vector resides.

A common practice for evaluating the performance of a machine learning algorithm when one does not have a large data set is to use a technique called K-fold cross validation. As we only collected signatures for 30 or 60 minutes every 10 seconds, we did not create a very large data set, therefore we performed K-fold cross validation. We split the positive and negative signatures into K sets of equal sizes (modulo K). We merge the positive signatures of set i with the negative signatures of set i, $\forall i \in \{0, K-1\}$, thus creating K folds. For each fold i, we set it aside and mark it as the *test data*. Fold $((i+1) \bmod K)$ is marked as the *validation data*, and the remaining folds are concatenated together and marked as the *training data*. Then we proceed to repeatedly train the SVM^{light} classifier on the training data while using the validation data to incrementally tune the parameters of the classifier, if any. Once the classifier parameters are chosen based on the performance on the validation data (e.g., choosing the parameters that maximize accuracy), the classifier is evaluated a single time on the test data. (Note that

Signature grouping	Baseline	Test set (average± std. dev., over all folds)		
	Accuracy (%)	Accuracy (%)	Precision (%)	Recall (%)
dbench(+1), kcompile(−1)	51.797	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
scp(+1), kcompile(−1)	51.177	99.39 ± 0.99	99.28 ± 1.54	99.56 ± 1.38
scp(+1), dbench(−1)	50.619	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
dbench(+1), kcompile ∪ scp (−1)	65.589	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
scp (+1), kcompile ∪ dbench (−1)	66.432	99.57 ± 0.69	99.17 ± 1.76	99.56 ± 1.38
kcompile (+1), scp ∪ dbench (−1)	67.977	99.57 ± 0.69	99.56 ± 1.38	99.09 ± 1.92

Table 3. Clustering: SVM^{light} averaged accuracy, precision, and recall over all 10-folds.

Signature comparison	Baseline	Test set (average± std. dev., over all folds)		
	Accuracy (%)	Accuracy (%)	Precision (%)	Recall (%)
myri10ge 1.4.3 (+1), 1.5.1(−1)	50.765	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
myri10ge 1.5.1 (+1), 1.5.1 LRO off(−1)	50.25	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00
myri10ge 1.4.3 (+1), 1.5.1 LRO off(−1)	51.015	100.00 ± 0.00	100.00 ± 0.00	100.00 ± 0.00

Table 4. myri10ge: SVM^{light} averaged accuracy, precision, and recall over all 8-folds.

to ensure correctness, the test set should be used only once, to assess the performance of a fully trained classifier.) We report the average metrics obtained by evaluating the classifier on the test data for each of the K folds—without further training the model.

We did not spend significant time searching the parameter space for either of the experiments. Instead, we simply set the SVM's kernel parameter to the default polynomial function, and we searched the parameter space of the trade-off between training error and margin, also known as the C parameter. Note that the signature vectors were scaled into the unit-ball using the L_2 norm—a common SVM classification practice.

We begin by evaluating the performance of the SVM classifier while distinguishing between the same three distinct workloads. Our classifier expects only two distinct classes labeled $+1$ and -1 respectively, therefore, since we have a total of three workloads we perform the following experiments. First, we apply the SVM classifier to datasets containing signatures from all possible combinations of two distinct classes, which yields the following groupings: `scp` $(+1)$ vs. `kcompile` (-1), `scp` $(+1)$ vs. `dbench` (-1), and `kcompile` $(+1)$ vs. `dbench` (-1). Next, we apply the SVM classifier to groupings in which we label the signatures from one of the workloads to be of class $+1$ and the remaining signatures from the other two workloads to be of class -1. We repeat the groupings for every workload, yielding three possible combinations (e.g., the first one being `scp` of class $+1$ and `kcompile` ∪ `dbench` of class -1).

Table 3 depicts the SVM performance in terms of accuracy, precision, and recall on the test set, averaged over all 10-folds. The SVM has been previously calibrated on the validation set. We also report the accuracy baseline, which is computed by reporting on the accuracy of a pseudo-classifier that always chooses the class with the label of the majority signatures. For example, if a dataset contains 100 data points of class $+1$ and 150 data points of class -1, then the baseline accuracy would be $\frac{150}{250} = 0.6$ (or 60%). Table 3 shows the SVM classifier to perform remarkably well. In particular, it is able to perfectly distinguish the workloads in three of the signature groupings, and performs almost as good for the remaining groupings. (To get a better intuition of the classifier's performance it is important to compare the reported accuracy with the baseline accuracy.)

Next, we evaluate how well can machine learning tell apart signatures generated by systems that only differ in subtle ways. For this experiment, the core kernel remains the

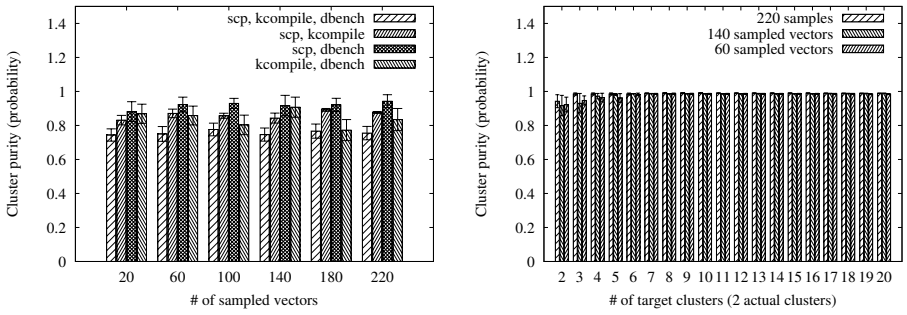
same, and we only alter the myri10ge device driver for the Myri10G NIC. Further, the device driver resides in a runtime loadable module, which Fmeter does not instrument, therefore the possible set of kernel functions that are being counted by Fmeter does not change. Instead, Fmeter records the signatures that contain the driver's behavior by virtue of the core-kernel symbols (i.e., functions) the driver calls *into*.

We chose the following three scenarios for the monitored system: **(i)** running with the myri10ge driver version 1.5.1 and default load-time parameters, **(ii)** running with the myri10ge driver version 1.4.3 and default load-time parameters, and **(iii)** running with the myri10ge driver version 1.5.1 but with the load-time parameter set to disable the large receive offload (LRO) capability. The first scenario provides a baseline for "normal" mode of operation, while the second and third scenarios provide various degrees of diverging modes of operation. For example, the scenario in which the LRO is disabled may correspond to a compromised system that maliciously loaded a runtime module/extension which increases the propensity of the machine to DDOS attacks. Likewise, the scenario in which we use an older version of the driver may be indicative of a buggy or a compromised vital subsystem. As a matter of fact, we disassembled the two driver versions (with `objdump`) and compared the un-relocated binary representation of the functions code. With respect to the older version of the driver, 24 functions were altered in the newer version, one function (`myri10ge_get_frag_header`) was removed, and 11 new functions were added. Of the newly added functions, only one was ever called during our workloads, namely `myri10ge_select_queue`. (Recall that none of these functions, or any other functions defined within the loadable drivers for that matter, belong to the Fmeter vector space.)

We ran Netperf [26] TCP stream tests between the two twin servers with the receiver machine running the Fmeter instrumented kernel and the three myri10ge driver variants. During the Netperf runs, we were able to achieve 10Gbps line rate. By contrast, if the conventional Ftrace kernel function tracer is on, we were able to only achieve a throughput of little more than half the line rate, which indicates that the overall overhead introduced by Fmeter was acceptable. Table 4 shows the results of the SVM classifier on all folds of the test set (we used eight-fold cross validation), after the C parameter was calibrated on the validation set. Our classifier achieves perfect accuracy, prediction, and recall in all cases. (The case in which we compared the version 1.4.3 of the driver against version 1.5.1 with LRO disabled was supposed to be a baseline indicator that is easier to classify than the other two.)

Signature Clustering Next we subject the Fmeter signatures to an unsupervised learning method such as clustering. We use the same three workloads we already evaluated our supervised machine learning against in Section 4.2, namely `scp`, `kcompile` and `dbench`. This choice of workload also allows us to directly compare how the unsupervised clustering stacks against the supervised machine learning.

We implemented two standard well-known clustering algorithms, namely agglomerative hierarchical clustering, and K-means respectively. Both clustering algorithms use the Euclidean distance (as induced by the L_2 norm), while the agglomerative hierarchical clustering is of the complete-, single-, and average-linkage flavors. We only



(a) (b)

Fig. 2. (a) *K*-means cluster purity (probability) given the number of (equally) sampled vectors from each class; (b) *K*-means cluster purity for `scp` and `dbench` signatures with respect to different number of target clusters (the K parameter).

report on the single-linkage variant throughout the paper since the results for complete- and average-linkage are similar.

Although the hierarchical clustering algorithm is more precise than the *K*-means algorithm, it is computationally more expensive, and it requires a notoriously hard to choose "height-cut" for automatic evaluation given more than two distinct classes. By contrast, the *K*-means algorithm converges significantly faster, and since the target number of resulting/expected clusters (i.e., the *K* parameter) is already given as an input parameter, it is straightforward to automatically evaluate the quality of the clustering result. We chose to use the *K*-means algorithm as our primary clustering unsupervised learning mechanism.

There are various metrics for evaluating the quality of clustering, like *purity*, *normalized mutual information*, *Rand index*, or the *F-measure*. We chose to use *purity*, since it is both simple and transparent. In particular, to compute the purity of a clustering, each resulting cluster is assigned to its most frequent class, and the accuracy of the assignment is measured by counting the number of correctly assigned signatures divided by the total number of signatures.

Figure 2(a) shows the cluster purity between all four permutations of the three workloads on the y-axis. We used the *K*-means algorithm, with the *K* parameter set for the actual number of clusters, i.e., $K = 3$ for the clustering of `scp`, `kcompile`, and `dbench`, and $K = 2$ otherwise. On the x-axis, the Figure depicts the number of signatures randomly selected, without replacement, from each workload class (the same number of signatures were selected from the `kcompile` workload as were selected from the `scp`, and `dbench` workloads). The results are averaged over 12 runs, with the error bars denoting standard error of the mean. There are three observations. First, the purity scores are high, denoting good clustering. Second, the clustering performance increases only slightly as the number of signatures increases, hence a small number of signatures are sufficient to properly determine each cluster's *centroid*. And third, the quality of the clusters for $K = 3$ and vectors sampled from each of the workloads available is lower than the quality of clusters yielded by *K*-means with $K = 2$ and vectors sampled only from two separate workloads, irrespective of which two workloads were sampled. This

means that clustering effectiveness appears to decrease as more classes (corresponding to different workloads) are considered.

At this point it is important to note that high purity is easy to achieve by simply increasing the number of expected clusters; in the case of K-means by increasing the value of the parameter K. In particular, if there are as many clusters as there are vectors (signatures), then the purity evaluates to 1.0. We proceed to leverage this property to show the quality of the clustering results. Figure 2(b) shows the purity of clustering signatures from the `scp` and `dbench` workloads, by increasing the number of target / expected clusters (we simply varied the parameter K of the K-means algorithm). As the Figure shows, the purity scores converge rapidly to the maximum value of 1.0 while the standard error of the mean decreases at the same time. The intuition is that there are very few (1, 2, or 3) additional clusters that capture the clustering "mistakes" made by the ideal clustering (where K is set to the actual number of classes, $K = 2$ in this case). The additional separate clusters group together these incorrectly classified signatures.

Compared to supervised machine learning, clustering on the same sets of signatures performs worse. Nevertheless, clustering is still a useful statistical analysis method, since it can naturally group signatures belonging to many classes. Furthermore, we can apply meta-clustering on the retrieved cluster centroids to determine which classes of behaviors, and hence not just individual signatures which are instances of behaviors, are closer to one another. Determining which system behaviors are similar in the way they use the operating system kernel functions can then be leveraged for low-level optimizations (e.g., improve cache locality).

5 Limitations

Fmeter uses the Ftrace infrastructure, as such, it only traces kernel function calls. We recognize that the kernel makes extensive use of function inlining and pre-processor macros (e.g., common list, hash-table, and even page table traversals) which we are unable to capture with our current methodology. Likewise, processes that require very little kernel intervention, like scientific applications, are likely to be all assigned similar signatures that are very close to the null/zero vector, which makes them harder to distinguish from one another, irrespective of the learning machinery.

Moreover, we recognize that the process of performing a measurement introduces uncertainty itself by interfering with the collected data. For example, the user-space daemon that logs signatures to disk interferes with the monitored system by virtue of using the kernel's pseudo file system and the kernel's proper file and storage subsystem (buffer cache, VFS, ext3, block layer, and so on). However, all retrieved signatures are perturbed uniformly by the logging.

6 Future Work

Currently, the overhead introduced by Fmeter is much higher than the overhead of statistical profiling tools like `oprofile`. Nevertheless, the Fmeter overhead is also significantly lower than that of the precise profiling tools like the ones relying on the conventional Ftrace kernel function tracer. Since the kernel function invocations follow

a power-law distribution (see Figure 1), a straightforward optimization to the Fmeter counting infrastructure would be to maintain a fast cache that holds the call counts for the top N hottest functions. Using a sufficiently small cache to account for the most popular kernel functions could lower the overheads, e.g., by decreasing the cache pollution incurred while following the Fmeter stubs. The value of N can be experimentally chosen based on the size of the processor caches.

We also plan to explore using Fmeter signatures to perform meta-clustering on already retrieved cluster centroids. Being able to apply clustering methods in such a recursive fashion would allow us to determine which entire classes, not just instances of behavior, are similar in the way they invoke the kernel functionality. We can thus leverage this information to better schedule concurrently executing tasks that rely on the same kernel code-paths (and implicitly the same in-kernel data-structures) on processor cores that share a cache domain (e.g., the L3 cache for an Intel Nehalem microarchitecture). Such assignments have the potential to boost the overall performance of monolithic kernels due to improved cache locality while executing in kernel-mode.

7 Related Work

System Monitoring Based on Indexable Signatures There have been several prior approaches that monitored system calls [11, 27, 28] to build some model which can be used to detect deviations from normal behavior. Furthermore, recent work [4, 29] has shown how indexable signatures can be used to capture essential system characteristics in a form that facilitates automated clustering and similarity based retrieval. Formal methods, like K-means clustering and the L_2 norm are then used to compare similarities among system states. Our work uses the statistical vector space model [12] to represent the system execution in a given time frame. Fmeter demonstrates how indexable signatures in low-level system monitoring (based on *all* kernel function calls, as opposed to just the system calls) can be generated with low overhead and used in a running high performance system. Like prior work, we too use existing information retrieval techniques to facilitate formal manipulation of Fmeter's signatures.

System Monitoring using Performance Counters The most commonly used monitoring tools record system variables for performance tuning and failure diagnostics. Oprofile [7] and DCPI [30] use hardware performance counters, and ProfileMe [31] uses instruction-level counters to periodically collect long-term system usage information. Such powerful post-processing utilities aid in visualizing and identifying potential performance bottlenecks. With such statistics, it is possible, for example, to understand and analyze the behavior of Java applications [32]. Since these tools focus on a small and limited set of predefined performance counters, it becomes impossible to look up arbitrary system behavior of interest in the logs. Fmeter differs from these tools by allowing execution sequences (low-level system signatures) to be indexed and later retrieved.

Chopstix [8] expands the use of individual counters by monitoring a diverse set of system information. These "vital signs" provide a wider picture of system execution at a given point in time. Along the same lines are tools such as CyDAT, Ganglia, CoMoN and Artemis [9, 10, 33, 34] which focus on monitoring distributed systems and cater to

the fast growing cloud computing environments. The visualization methods for such tools are important for understanding interactions amongst the nodes in a cluster due to the large volumes of logs and heterogeneity in platforms.

System Monitoring Based on Logging System logging is used in another area of system monitoring. System operators, developers and automatic trainers can extract error conditions in the logs and use machine learning techniques to predict indicated errors [35–38]. Alternatively, system state signatures can be recorded and searched for automatic diagnosis [39]. There is also a dedicated set of tracers which allows isolating non-deterministic system behavior and heisenbugs [40, 41] and replaying execution from the logs [42] to reproduce error conditions or perform fault correction on the fly [43, 44]. In addition, statistical induction techniques exist for automated performance diagnosis and management at the server application level [45]. Fmeter differs from these tools since it is able to generate indexable low-level signatures in a running system with low overhead (see Section 4.2).

System Monitoring Based on Indexing Logs Signature based system monitoring has also inspired methodologies which focus on post-processing of logs to generate useful inferences. This class of methods attempts to generate inferences based either on identifying some signatures in the log data or finding anomaly-based aberrations [46, 47]. Our method is a generalization of such analysis which can be used for both signature-based retrieval and anomaly detection. Alternatively, use of fine-grained control flow graphs as signatures has also been proposed as a useful malware detection strategy [48]. Moreover, similarity based measures working at the application level on a diverse set of system attributes have shown to be successful [49]. Latest work shows a novel path of combining source code analysis and runtime feature creation into console log mining for anomaly detection [50, 51]. Our approach explores a similar way of applying machine learning and information retrieval techniques, yet using a different class of low-level signatures (and an efficient, specific signature extraction method).

8 Conclusion

We present Fmeter, a monitoring infrastructure that extracts formal, indexable, low-level system signatures by embedding kernel function calls into the classical vector space model. Fmeter represents system signatures as `tf-idf` weight vectors by disregarding the semantic information in a document and consider only the statistical properties of the terms belonging to the document (and to the corpus). In our case, we disregard the sequence of kernel function calls (the "call stack" trace), the function parameters, memory location accesses, hardware device state manipulation and so on. Instead, we rely on as little information as possible, namely counting the kernel function calls. This approach is sufficient to provide meaningful and effective system signatures, while incurring low system overhead. Further, the signatures are naturally amenable for statistical information retrieval manipulations, like clustering, machine learning, and information retrieval. We demonstrate the efficacy of Fmeter by yielding near-perfect results during clustering and supervised classification of various system behaviors.

Availability

The Fmeter source code is published under BSD license and is freely available at `http://fireless.cs.cornell.edu/fmeter`.

References

1. Hellerstein, J.L.: Engineering autonomic systems. In: ICAC '09
2. Schroeder, B., Pinheiro, E., Weber, W.D.: DRAM errors in the wild: a large-scale field study. In: SIGMETRICS '09
3. Dean, J.: Designs, Lessons and Advice from Building Large Distributed Systems. Keynote talk: LADIS '09
4. Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., Fox, A.: Capturing, indexing, clustering, and retrieving system history. In: SOSP '05
5. Mavinakayanahalli, A., Panchamukhi, P., Keniston, J., Keshavamurthy, A., Hiramatsu, M.: Probing the guts of kprobes. In: Linux Symposium '06
6. : Ftrace - Function Tracer. `http://lwn.net/Articles/322666/`
7. : Oprofile. `http://oprofile.sourceforge.net`
8. Bhatia, S., Kumar, A., Fiuczynski, M.E., Peterson, L.: Lightweight, high-resolution monitoring for troubleshooting production systems. In: OSDI '08
9. Cretu-Ciocarlie, G.F., Budiu, M., Goldszmidt, M.: Hunting for problems with artemis. In: Proceedings of WASL. (2008)
10. Massie, M.L., Chun, B.N., Culler, D.E.: The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. In: Proceedings of Parallel Computing. (2004)
11. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy (SP). (2001) 144–155
12. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Communications of the ACM **18**(11) (1975) 613–620
13. Booth, A.D.: A "law" of occurrences for words of low frequency. Information and Control **10**(4) (1967) 386–393
14. Grishchenko. `http://wikipedia.org/wiki/File:Wikipedia-n-zipf.png`
15. Boyd-Wickizer, S., Morris, R., Kaashoek, M.F.: Reinventing scheduling for multicore systems. In: HotOS'09
16. : Debugfs. `http://lwn.net/Articles/115405/`
17. Srivastava, A., Eustace, A.: ATOM - A System for Building Customized Program Analysis Tools. In: PLDI '94
18. Edge, J.: A lockless ring-buffer. `http://lwn.net/Articles/340400/`
19. Edge, J.: One ring buffer to rule them all? `http://lwn.net/Articles/388978/`
20. Brandenburg, B.B., Anderson, J.H.: Feather-trace: A light-weight event tracing toolkit. In: OSPERT '07
21. Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R.W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., Uhlig, V.: K42: building a complete operating system. In: EuroSys. (2006)
22. Staelin, C.: lmbench: Portable Tools for Performance Analysis. In: USENIX ATC '96
23. Joachims, T.: Svm^{light} `http://svmlight.joachims.org/`.
24. Joachims, T.: Learning to Classify Text Using Support Vector Machines. Dissertation. Springer (2002)
25. Vapnik, V.N.: The Nature of Statistical Learning Theory. Springer (1995)

26. Netperf. http://netperf.org/
27. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. In: IEEE Symposium on Security and Privacy. (1996)
28. Li, P., Gao, D., Reiter, M.K.: Automatically adapting a trained anomaly detector to software patches. In: RAID '09
29. Bodik, P., Goldszmidt, M., Fox, A., Woodard, D.B., Andersen, H.: Fingerprinting the datacenter: automated classification of performance crises. In: EuroSys '10
30. Anderson, J.M., Berc, L.M., Dean, J., Ghemawat, S., Henzinger, M.R., Leung, S.T.A., Sites, R.L., Vandevoorde, M.T., Waldspurger, C.A., Weihl, W.E.: Continuous profiling: where have all the cycles gone? In: SOSP '97
31. Dean, J., Hicks, J.E., Waldspurger, C.A., Weihl, W.E., Chrysos, G.: Profileme: hardware support for instruction-level profiling on out-of-order processors. In: MICRO '97
32. Sweeney, P.F., Hauswirth, M., Cahoon, B., Cheng, P., Diwan, A., Grove, D., Hind, M.: Using hardware performance monitors to understand the behavior of java applications. In: Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM). (2004)
33. DiFatta, C., amd Daniel V. Klein, M.P.: Carnegie mellon's cydat: Harnessing a wide array of telemetry data to enhance distributed system diagnostics. In: Proceedings of WASL. (2008)
34. Park, K., Pai, V.S.: Comon: a mostly-scalable monitoring system for planetlab. SIGOPS Oper. Syst. Rev. **40**(1) (2006) 65–74
35. Salfner, F., Tschirpke, S.: Error log processing for accurate failure prediction. In: WASL '08
36. Sandeep, S.R., Swapna, M., Niranjan, T., Susarla, S., Nandi, S.: Cluebox: A performance log analyzer for automated troubleshooting. In: WASL '08
37. Fulp, E.W., Fink, G.A., Haack, J.N.: Predicting computer system failures using support vector machines. In: Proceedings of WASL. (2008)
38. Hauswirth, M., Sweeney, P.F., Diwan, A., Hind, M.: Vertical profiling: understanding the behavior of object-priented applications. In: OOPSLA '04
39. Redstone, J., Swift, M.M., Bershad, B.N.: Using computers to diagnose computer problems. In: Proceedings of HotOS. (2003) 91–86
40. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI '08
41. Ronsse, M., Christiaens, M., Bosschere, K.D.: Cyclic debugging using execution replay. In: Proceedings of the International Conference on Computational Science-Part II '01
42. Guo, Z., Wang, X., Tang, J., Liu, X., Xu, Z., Wu, M., Kaashoek, M.F., Zhang, Z.: R2: An application-level kernel for record and replay. In: OSDI '08
43. Tucek, J., Lu, S., Huang, C., Xanthos, S., Zhou, Y.: Triage: diagnosing production run failures at the user's site. In: SOSP '07
44. Qin, F., Tucek, J., Zhou, Y., Sundaresan, J.: Rx: Treating bugs as allergies - a safe method to survive software failures. ACM Trans. Comput. Syst. **25** (2007) 7
45. Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., Chase, J.S.: Correlating instrumentation data to system states: a building block for automated diagnosis and control. In: OSDI '04
46. Sequeira, K., Zaki, M.: Admit: anomaly-based data mining for intrusions. In: KDD '02
47. Ghosh, A.K., Schwartzbard, A.: A study in using neural networks for anomaly and misuse detection. In: Proceedings of the 8th conference on USENIX Security Symposium '99
48. Bonfante, G., Kaczmarek, M., Marion, J.Y.: Control flow graphs as malware signatures. In: Proceedings of the International Workshop on the Theory of Computer Viruses. (2007)
49. Lane, T., Brodley, C.E.: Temporal sequence learning and data reduction for anomaly detection. ACM Trans. Inf. Syst. Secur. **2**(3) (1999) 295–331
50. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: SOSP '09
51. Lou, J.G., Fu, Q., Yang, S., Xu, Y., Li, J.: Mining invariants from console logs for system problem detection. In: USENIX ATC '10