Microservice Fingerprinting and Classification using Machine Learning

Hyunseok Chang Murali Kodialam T.V. Lakshman Sarit Mukherjee
Nokia Bell Labs, USA
{firstname.lastname}@nokia-bell-labs.com

***Abstract*—Application aware data centers promise various benefits for data center management, in terms of resource provisioning, power estimation, network management, security protection, etc. However, the emerging microservices make it challenging for data center operators to accurately identify what applications are deployed by tenants, due to their highly dynamic and heterogeneous nature. In this paper, we address the problem of fingerprinting microservices in a unified, efficient, accurate and non-intrusive fashion. To this end, we characterize the runtime behaviors of microservices using eBPF-based lightweight system call tracing. To accurately fingerprint a diverse set of microservices based on their system call activities, we utilize the machine learning approach which combines Bayesian learning and LSTM autoencoders. We demonstrate that our approach can fingerprint many real-world microservices with 99% accuracy, using 1–2% additional CPU resource, and can detect the presence of previously unseen microservices with near perfect accuracy.**

I. Introduction

Modern cloud-based enterprise applications [1], [2] and emerging web-scale service architectures [3]–[6] are increasingly realized using *microservices* due to their intrinsic benefits such as high developer productivity, deployment agility, improved scalability, flexibility and resiliency [7]. Faced with the increasingly microservice-dominated workloads, today's data center infrastructures are evolving to accommodate large numbers of dynamically created microservices (e.g., Kubernetes, Mesos).

Another trend in today's data center management is the growing *application awareness* in the data center infrastructure. Traditionally, data center management has remained mostly application agnostic, focusing on maintaining the health of the data center without knowing what applications are deployed by tenants. However, application awareness plays an increasingly important role in different aspects of data center operations as follows.

Resource provisioning. Server consolidation in data centers often results in performance interference among co-located applications, caused by incompatible resource usage patterns (e.g., CPU-bound vs. I/O-bound, interactive vs. batch-oriented) [8]. If the data center is application-aware, it can help with consolidating servers with compatible workloads, and forecasting long term growth in hardware resource demand.

Power modeling. Accurate power modeling is important for energy-efficient data center operations. However, the increasing heterogeneity of data center applications makes accurate power modeling and estimation challenging. Compared to application-oblivious usage based power models, application-dependent model calibration is shown to be promising [9].

Network management. Next-generation network management and monitoring products provide application-level traffic engineering, QoS and reporting capabilities based on application identification [10], [11]. Also, exporting application-level knowledge to the network is shown to be useful to improve application's networking performance [12], and to predict networking behaviors (e.g., elephant flow detection [13]).

Security management. Application awareness is particularly important in data center security. Application-level knowledge can for example be utilized to generate risk assessment reports tailored for deployed tenant applications [14], or to automatically generate security policies in dynamic DevOps environments [15] or relevant application signatures in intrusion detection systems [16].

While benefits of application awareness can be realized in different contexts as described above, it is challenging to identify applications reliably and scalably, especially in the presence of the emerging microservices. The microservice ecosystem of modern data centers is extremely heterogeneous, with a mix of open-source software (e.g., Nginx, Redis), custom-built services [17], and proprietary applications. A straightforward approach to fingerprint such diverse sets of microservices is to investigate individual microservices in search for application-specific hints (e.g., executable names, port numbers, config files, container image metadata). Such an application-specific approach is not reliable or complete as those hints are not binding properties of microservices (i.e., can be easily tweaked or hidden), and does not scale with the growing microservice ecosystem. On the other hand, deep packet inspection based application identification (e.g., OpenAppId [18]) is not only computationally inefficient, but also incomplete (e.g., unable to identify CPU or I/O bound non-networking microservices). For practical deployment, any proposed solution needs to be able to identify highly dynamic microservices in a timely fashion with minimal overhead. In terms of accuracy, none of these approaches can provide implementation-level fingerprinting capabilities. For example, MySQL, MariaDB and Percona, which are binary-compatible database implementations, may be exposed to distinct security vulnerabilities, and the ability to correctly detect and differentiate them can provide proper security protection for

978-1-7281-2700-2/19/$31.00 2019 © IEEE

those tenants who deploy them. Finally, increasingly rigorous tenant privacy standards, which prohibit any kind of intrusive profiling or instrumentation of tenant-owned microservices, add to the difficulty in accurate fingerprinting.

Faced with these challenges, we address in this paper the problem of fingerprinting microservices in a *unified, efficient, non-intrusive, and accurate* fashion. To this end, we set out to characterize the run-time behaviors of microservices in an application agnostic fashion. The most generic way to monitor the behavior of a microservice is to examine the system calls issued by the microservice while it interacts with the host operating system (e.g. to access file systems and networks, synchronize threads, etc.). The predominant container-based (e.g., Docker, LXC) microservice deployments enable the monitoring of system calls within the end server's operating system without profiling the microservices themselves. Not only application-agnostic, but also transparent to the microservices, system call level tracing meets the unified and non-intrusive monitoring requirements. For efficient monitoring of such low-level system activities, we turn to the modern kernel tracing technology called the Extended Berkeley Packet Filter (eBPF) [19], which enables flexible tracing of server-wide system calls at marginal CPU overhead.

To achieve accurate fingerprinting, we design a supervised Bayesian learning model called a *fingerprint model* that classifies the sequences of system calls generated by deployed microservices to derive their identities. In this model, we characterize the sequence of system calls as a Markov chain of order-k, where the order-k transition probability captures the history in a sequence for finer classification. To cope with the model's increasing resource demand from a large number of classification categories, which is common in real world environments, we apply the Bayesian-based fingerprinting in a hierarchical fashion.

Still, the drawback of the Bayesian-based fingerprint model is that it can only fingerprint *known* classes of microservices, against which the model is already trained. When a previously unseen class of new microservices (i.e., outliers) are encountered, the model blindly categorizes them into one of known classes with the highest probability, which is obviously incorrect. In real-world data centers, where the universe of microservices is rarely predefined, but constantly evolves with new types of microservices added, the accuracy of Bayesian-based fingerprinting will be significantly degraded by outliers. To address this limitation, we supplement the fingerprint model with deep-learning-based outlier detection that can reliably detect whether or not a given system call sequence is generated by an outlier. The outlier detection is realized with a self-supervised LSTM autoencoder which learns the representative system call sequences generated by known microservices. When fed with a system call sequence from a previously unseen type of microservice, the autoencoder produces a high reconstruction loss, from which we can reliably tell that the sequence is not suitable for classification by the fingerprint model. In this case, we re-train the fingerprint model as well as the autoencoder to incorporate the newly found outlier.

We implement a working prototype consisting of eBPF-based system call tracing, fingerprinting and outlier detection modules, and test the prototype against various real-world microservices. We show that we can differentiate among 30 different types of microservices of mixed similarity using less than 1K system calls with 99% accuracy. In general, a higher-order Bayesian model achieves similar accuracy with shorter system call sequences. For typical microservice deployments, we show that fingerprinting requires only marginal CPU overhead (1–2%) for eBPF-based system call tracing. Finally, we demonstrate that the autoencoder model can detect outliers near perfectly with no false-positive/false-negative error, except when outliers are highly similar to inliers (e.g., binary compatible implementations).

II. Machine Learning Models for Microservice Classification

A. Bayesian Model for Microservice Fingerprinting

In the following, we present the supervised Bayesian model for fingerprinting microservices. A microservice usually runs as a stand-alone process or inside a container. We refer to the execution of a microservice in either form as a microservice engine. Each microservice engine invokes a stream of system calls, which we call verbs. Let $\mathcal{V}$ denote the universe of verbs and $\mathcal{E} = \{E_1, E_2, \dots E_n\}$ denote the set of engines. For the purpose of modeling, we assume these verbs to form a random process. Let $V_1^j, V_2^j, \dots$ represent the ordered sequence of verbs invoked when engine E_j is executed. The i^{th} verb invoked when engine E_j is executed is denoted by the random variable V_i^j which takes on values from the set $\mathcal{V}$. The goal of the model is to characterize the underlying probabilities of this random process so that each engine's unique characteristics (i.e., fingerprint) can be expressed through the probabilities.

1) Estimating Verb Probabilities: We generate training data from each engine by executing the engine and collecting the sequence of verbs invoked by the engine, which we call the **training sequence**. Each engine has one training sequence. Assume that we have a training sequence of length n_j verbs from engine E_j. Let $T^j = (T^j(0), T^j(1), \dots T^j(n_j))$ where $T^j(t) \in \mathcal{V}$ denotes the t^{th} verb invoked by engine E_j. We use $\mathbf{v} = (v_0, v_1, \dots, v_{k-1})$ where $v_i \in \mathcal{V}$, to represent k dimensional vector of verbs. We say that $\mathbf{v}$ is at location t for engine E_j if $T^j(t) = v_0, T^j(t-1) = v_1, \dots, T^j(n-k) = v_k$. We use $\mathbf{V}^k$ to denote the set of all combination of verbs of length k. Therefore there are $|\mathcal{V}|^k$ vectors in $\mathbf{V}^k$. We define an indicator variable $I^j(t, \mathbf{v})$ which is set to one if and only if $T^j(t-i) = v_i \quad \forall\ 0 \le i \le k$. In other words, $I^j(t, \mathbf{v})$ is set to one if the sequence $\mathbf{v}$ is at location t for engine E_j. If we want to estimate the probability that sequence $\mathbf{v} \in \mathbf{V}^k$ occurs when engine E_j is invoked, we can use the standard frequency definition of probabilities to say

$$p^j(\mathbf{v}) = \frac{\sum_{t=k}^{n_j} I^j(t)}{n_j}.$$

2) Estimating Conditional Probabilities: We are interested in computing $p^j(v|\mathbf{v})$ which is the probability that a verb $v \in \mathcal{V}$ occurs immediately after the occurrence of the sequence of verbs $\mathbf{v}$ in the invocation of engine E_j. We use the training sequence data to estimate this conditional probability.

Dependency order. Depending on the model that we use, we restrict the length of the conditioning sequence. For the **order-0 model** we assume that verb invocation for engine E_j is a sequence of independent calls. In this case, there is no conditioning and we just want to estimate $p^j(v)$ which is the probability that verb $v \in \mathcal{V}$ is invoked by engine E_j. More generally we can define an **order-k model** where the current verb depends on the last k verbs. In this case, we want to compute the conditional probability $p^j(v|\mathbf{v})$ where $\mathbf{v} \in \mathbf{V}^k$. There are $|\mathcal{V}|^k$ conditioning vectors.

The conditional probabilities $p^j(v|\mathbf{v})$ can be computed using a simple, frequency based interpretation of probability. However, this will result in assigning a probability of zero for all sequences that are not seen in the training sequence, which is a problem especially for higher order dependency models. To avoid this issue, we use a Bayesian model with Dirichlet prior (see Appendix for detail). Once we compute the conditional probabilities $p^j(v|\mathbf{v})$ for all engines E_j in $\mathcal{E}$ from their training sequences, we consider a sequence of verbs called the **test sequence** that are invoked from some unknown engine. We now use these conditional probabilities to estimate the probability that this test sequence is from engine E_j.

3) Estimating the Engine Probabilities: We observe a test sequence of verbs $\mathbf{v} = (v_1, v_2, \ldots, v_m)$. Our objective is to determine the probability that this sequence is generated by the engine E_j. We use the notation $v[i:j]$ to represent the test sub-sequence $(v_i, v_{i-1}, \ldots, v_j)$. We use a Bayesian approach to determining this probability. We denote the probability that the engine is E_j given that we are observing a test sequence $\mathbf{v}$ by $P[E_j|\mathbf{v}]$. We use Bayes theorem to write

$$P[E_j|\mathbf{v}] = \frac{P[\mathbf{v}|E_j]P[E_j]}{P[\mathbf{v}]}.$$

If we do not have any prior information about the engines, we assume that $P[E_j] = \frac{1}{n}$ (where n is the number of engines learnt by the supervised learning) for all engines E_j. Therefore, $P[E_j|\mathbf{v}] \propto P[\mathbf{v}|E_j]$.

Since we typically use order-k models for small k, we ignore the k terms before the product and write for order-k,

$$P[\mathbf{v}|E_j] \approx \prod_{i=k}^{m} p^j(v_i|v[i-1:i-k]).$$

We are interested in picking the engine J with the highest probability, i.e.,

$$J = \arg\max_j P[\mathbf{v}|E_j] = \arg\max_j \log\left(P[\mathbf{v}|E_j]\right)$$

where we used the fact that $\log()$ is an increasing function. This in turn means that

$$J = \arg\max_j \log\left(P[\mathbf{v}|E_j]\right) = \sum_{i=k}^{m} p^j(v_i|v[i-1:i-k])$$

4) Extending to Hierarchical Fingerprinting: The computation overhead of fingerprinting with the aforementioned Bayesian model grows linearly with the cardinality of the microservice universe $\mathcal{E} = \{E_1, E_2, \ldots E_n\}$. This can be problematic for real-time fingerprinting as $|\mathcal{E}|$ can be high in reality. Besides, if a higher-order Bayesian model is desired, computing the order-k conditional probability matrices across all known E_j will result in exponentially long training time, as well as significant memory footprint for the matrices. To address this potential scalability issue, we apply Bayesian-based fingerprinting in a hierarchical fashion. That is, we first cluster all engines E_j in $\mathcal{E}$ into N groups. Different clustering methods are possible [20]. In this case, we define the distance between two engines using the Frobenius distance between their respective order-1 conditional probability matrices, and perform agglomerative clustering based on the distance metric. Once microservice groups are so defined, we apply fingerprinting in two steps; *group-level classification*, followed by microservice-level classification within a chosen group. In order to perform group-level classification (i.e., identify which group a test sequence belongs to), we combine the training sequences from all E_j in each group, and train a group-level Bayesian model with the aggregate per-group training sequences. This hierarchical approach scales better with the size of $\mathcal{E}$, and allows us to limit resource-expensive higher-order Bayesian fingerprinting to particular groups only.

B. Autoencoder Model for Outlier Detection

The Bayesian-learning-based fingerprint model presented before is designed under the assumption that the types of available microservice engines are known a-priori or under the tight control of data center operators (e.g., in telecom data centers). However, in typical multi-tenant public clouds, where the universe of microservice engines constantly evolves with new types of microservices introduced by different tenants, the fingerprint model alone is not sufficient, as it cannot detect "none-of-the-above" types of microservices that the fingerprint model is not trained against.

To solve this so-called *outlier detection problem*, we turn to the deep learning based autoencoder approach, which has been successfully adopted for anomaly detection [21], [22]. During training, autoencoder learns to capture representative features of normal training data into fixed-length feature vectors, which it then uses to reconstruct the original training data. During testing, if the trained autoencoder is fed with abnormal data not seen during training, it yields a relatively high reconstruction loss, from which the abnormality of the test data is detected.

Following this approach, we build a Long Short-Term Memory (LSTM)-based autoencoder for each engine E_j, which learns the representative sequences of verbs generated by the engine. We denote an LSTM autoencoder trained for engine E_j as AE_j. Let's consider that the reconstruction loss ℓ yielded by AE_j against test sequences of engine E_i forms a random variable L_j^i. Then one can expect that for any engine $j \in \mathcal{E}$, the probability distribution of L_j^i $(i \neq j)$ is shifted to the right of L_j^j's distribution. For example, Fig. 1 shows the probability

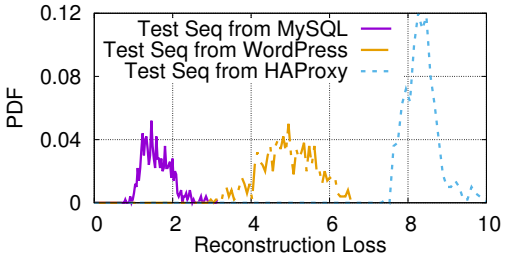


Fig. 1: Loss distributions from MySQL's autoencoder model.

distributions of L_j^i for three real-world microservices (j = MySQL, i = MySQL, WordPress, HAProxy). Intuitively, when a test sequence from a microservice under consideration is evaluated against the engines E_j in $\mathcal{E}$ using their respective AE_j, if none of AE_j in $\mathcal{E}$ yields a reconstruction loss that falls within the confidence interval of L_j^j, it is likely that the test sequence does not come from any of the engines in $\mathcal{E}$. Following this intuition, we use the procedure in Algorithm 1 for detecting an outlier. The loss threshold $\hat{L}$ needs to be carefully chosen such that the procedure minimizes both false-positive and false-negative detection errors (see Section IV-C for more details). While this procedure can be invoked against the entire universe of microservice at once (i.e., $\mathcal{G} = \mathcal{E}$), it is more scalable to apply the procedure to a subset of engines in a hierarchical fashion, similar to the fingerprint model.

Algorithm 1 Procedure for detecting an outlier.

procedure DETECT_OUTLIER(T, $\mathcal{G}$, $\hat{L}$)
input: T, /* test sequence(s) */
$\mathcal{G}$, /* group of engines to test against */
$\hat{L}$ /* reconstruction loss threshold */
output: TRUE or FALSE
  $min\_loss \leftarrow$ MAX_LOSS
  /* find the minimum reconstruction loss in $\mathcal{G}$ */
  for each E_i in $\mathcal{G}$ **do**
    $loss \leftarrow evaluate\_model(AE_i, T)$
    if $min\_loss > loss$ **then**
      $min\_loss \leftarrow loss$
    end if
  end for
  if $min\_loss > \hat{L}$ **then**
    return TRUE /* T is generated by outlier */
  else
    return FALSE /* T is generated by an engine in $\mathcal{G}$ */
  end if
end procedure

III. Implementation

We implement a fully functional prototype of the proposed architecture. The functional diagram of the prototype is presented in Fig. 2. The two main components of the prototype are (i) the system call monitor and (ii) the microservice classifier, each of which we describe respectively in the following.

A. System Call Monitor

The system call based microservice fingerprinting is predicated on efficient system call tracing for practical deployment. The traditional way (e.g., `strace`) of tracing system calls using the `ptrace` system call introduces prohibitively high

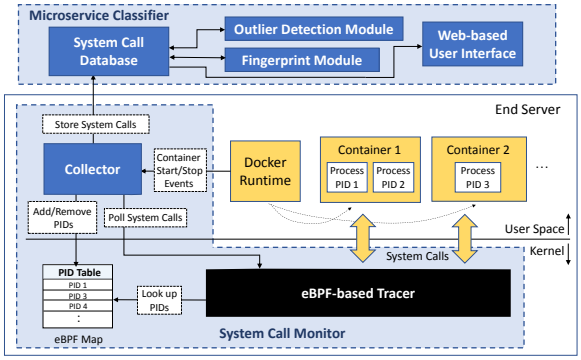


Fig. 2: Microservice classification prototype.

overhead as the process being traced is paused twice for each system call invocation. For example, we found that when simple disk I/O workload is traced with `ptrace`, it results in 200× slowdown in disk I/O rate and 180× more CPU usage compared to the baseline without monitoring. To implement system call monitoring with minimum resource overhead and without intrusive instrumentation, we utilize the in-kernel virtual machine technology called eBPF [19]. Without any kernel customization, eBPF allows user-defined byte code (known as eBPF programs) to be dynamically attached to various kernel hooks to monitor any interesting kernel events. It also provides in-kernel key-value maps for stateful processing and various helper functions for specialized tasks. Modern eBPF toolchains such as `bcc` [23] support just-in-time compilation to allow eBPF programs to run at native speed in the kernel.

For our purpose, we extend the eBPF-based system call tracer called `vltrace` [24]. The monitoring procedure of the original `vltrace` is more heavyweight than necessary. For example, the tool captures both entry and exit of each call, and reports system call arguments as well. More importantly, `vltrace` does not offer any run-time flexibility, and its tracing behavior remains fixed once the eBPF program is compiled and loaded into the kernel.[1] This inflexibility makes it unsuitable if we want to dynamically enable/disable tracing on a per-microservice basis to minimize tracing overhead. We address these limitations as follows. We simplify the eBPF tracer to record only system call ID and timestamp at the entry of each call, but also extend it to collect system call-specific contexts (e.g., type of file descriptors for `open`, `read`, `write`, etc.) for more fine-grained system call monitoring. We dynamically turn on/off tracing for different PIDs without reloading the eBPF tracer itself, but simply by writing to an in-kernel eBPF map (called *PID table*) accessed by the tracer. When a new microservice is launched, the userspace *collector* daemon is notified of its main PID by Docker runtime. It then records the PID as well as all the PIDs in its process tree in the PID table, so that the eBPF tracer starts tracing them. When the collector accumulates enough tracing data for a particular PID, it marks the PID off from the PID table to stop tracing it.

[1]Reloading the `vltrace`'s eBPF byte code running in the kernel for any update takes more than 30 seconds as the byte code needs to be detached from and re-attached to several hundreds of individual system call hooks.

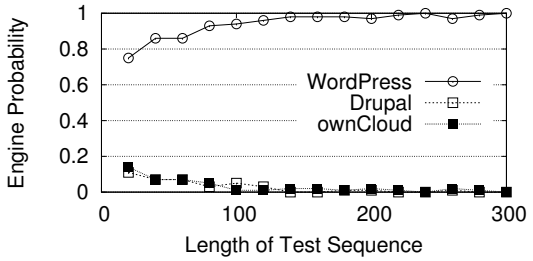


Fig. 3: An example probability distribution over different test sequence length.

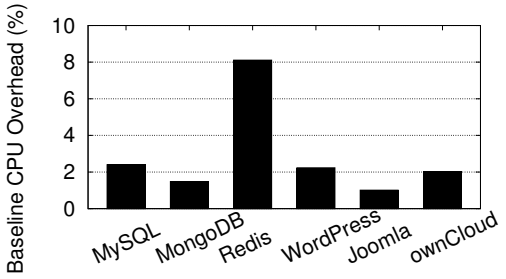


Fig. 4: Baseline CPU overhead of system call tracing.

B. Microservice Classifier

Microservice classifier consists of (i) the central database which stores system call sequences collected from the system call monitor, (ii) the outlier detection module which performs outlier detection with newly added sequence data, and (iii) the fingerprint module which performs fingerprinting based on only those sequence data not originating from outliers.

Fingerprint module. The fingerprint module, which implements the supervised Bayesian learning model, operates in two modes: training and testing. In the training mode, the module is fed with training sequences as well as the true identification of the underlying microservice engines responsible for the sequences. Recall that in an order-k Bayesian model, $k+1$ consecutive verbs are checked, and a transition from $v_0, v_1, \ldots, v_k$ to v_{k+1} is assumed to happen. During training, the module examines the input sequence in groups of k, advancing by 1 for the next group, and builds the transition matrices with order $k = 0, 1, \ldots, L$ (L is a system parameter) for the given microservice. If the microservice is already learnt, and therefore such matrices exist, the module augments them with the newly learnt sequence. In order to achieve this efficiently, we keep the raw occurrence counts of the transitions in the matrices and normalize them (i.e., so that row sum is 1) only during testing mode. Note that the matrices are initialized with Dirichlet prior (1 in each cell of the matrices) so that there is always a non-zero probability of a transition.

In the testing mode, a test sequence is fed to the module. The probability of the sequence is computed using the matrix of order-k for each of the trained microservices. This generates an order-k probability distribution for the test sequence to belong to a particular trained microservice. The module selects the highest probability and declares the sequence to belong to the corresponding microservice. For example, Fig. 3 shows how the probability distributions among three example microservices change with increasing test sequence length. If a microservice is classified successfully, the probability for that microservice quickly converges to one as shown in the figure.

Outlier detection module. The outlier detection module is realized as a set of LSTM autoencoders AE_j, trained for each E_j in $\mathcal{E}$. Each LSTM autoencoder implements a sequence-to-sequence autoencoder [25], which reconstructs the original input sequence as output. Before being fed into the autoencoder, the input sequence is first converted into a sequence of one-hot encoded vectors. The autoencoder then uses an LSTM layer to encode this sequence into a 128-dimensional feature vector. The output of the LSTM layer is then repeated L times (L=input sequence length) to construct an intermediate sequence. Finally, this intermediate sequence is decoded into the original one-hot encoded input sequence by another LSTM layer with 128 output units, followed by a time-distributed Dense layer with softmax activation. We train each AE_j with 1K input sequences, each with length of 500.

IV. Evaluation

In this section, we evaluate our prototype implementation to answer the following key questions.

- What is the resource and performance overhead of tracing microservices at the system call level?
- How accurate is the Bayesian based microservice classification?
- Can the supervised microservice classification scale with a growing microservice universe?
- How accurately can we detect outlier microservices?
- How robust is the outlier detection to different deployment environments?

For evaluation we deployed the end-server component of our prototype as well as test microservices in a VM equipped with 16 virtual CPU cores at 2.60GHz and 64GB memory. The VM runs on Ubuntu 17.10, kernel 4.15.0, with `bcc` v0.5.0 and eBPF JIT flag enabled.

A. System Call Monitoring

First, we investigate the resource and performance implications of real-time system call tracing in our prototype.

Baseline CPU overhead. The eBPF-based in-kernel tracer captures the first N system calls for every new process spawned by any unclassified microservice. Any subsequent system calls from the microservice are ignored by the eBPF tracer. While the resource overhead of the eBPF tracer is the most pronounced during the first N system calls, every single subsequent system call still hits the eBPF tracer. We refer to the CPU overhead of the eBPF tracer *after* tracing is disabled as *baseline overhead*. Any microservice which remains running after the first N system calls will incur the baseline overhead. Since real-world microservices typically generate a huge number of system calls, the baseline overhead is the critical measure of eBPF tracing's resource impact.

Fig. 4 reports the baseline overhead for different types of microservices. To measure the baseline overhead, we compare

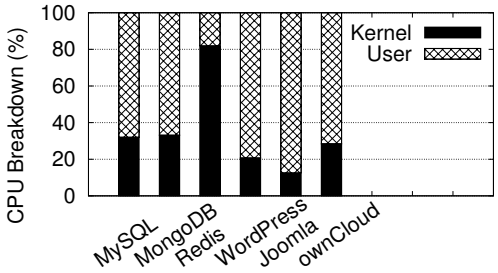


Fig. 5: CPU usage breakdown.

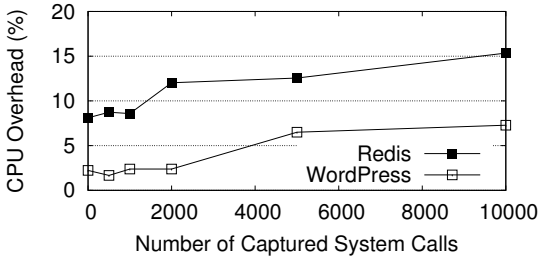


Fig. 6: The effect of the number of captured system calls.

the total CPU usage of a server in two different settings. First, we deploy a microservice on a baremetal server without the eBPF tracer and collector running, and measure the total CPU time (number of CPU seconds) of the server (CPU_1) while synthetic workload is injected into the microservice. Next, we run the eBPF tracer and collector, but disable tracing for the microservice. We then replay the same workload used previously, and measure the total CPU time (CPU_2) of the server. The baseline CPU overhead is then $\frac{CPU_2-CPU_1}{CPU_1} \times 100$.

Fig. 4 shows that the baseline overhead remains low (1–2%) for most microservices except for Redis. In case of Redis, its synthetic workload consists of invoking simple `GET/SET` APIs over TCP, which does not involve much userspace processing other than memory read/write and network I/O. However, even with Redis, we observe that if it is loaded with non-trivial APIs, e.g., iterative operations (`KEYS`) or compound operations (`EVAL`), its baseline overhead is diminished quickly. From this experiment, we conclude that our eBPF tracer introduces reasonably small resource overhead (1–2%) for most real-world microservices handling realistic workloads.

Fig. 5 explains the reported baseline CPU overhead. In this figure, we plot the CPU breakdown (kernel vs. user space) for the same set of microservices, which is measured while they are operated normally without eBPF tracing. It confirms that Redis with higher baseline overhead is indeed operating system intensive, spending 80% time in kernel space.

Effect of test sequence length. In the next experiment, we show the CPU overhead of tracing actual system calls with eBPF. We choose Redis and WordPress as two candidates for testing. As already shown in Fig. 5, these two microservices exhibit widely different CPU usage patterns; Redis is operating system intensive with 80% kernel processing, while WordPress runs only 20% of time in the kernel. In the experiment, we deploy either microservice for five minutes while constant workload is injected (e.g., 10K `SET/GET` requests/sec. for Redis, and 20 `HTTP-GET` requests/sec. for WordPress). During this run, we activate the eBPF tracer, and monitor the first N system calls. We repeat this experiment without the eBPF tracer. The ratio of CPU time difference in these two experiments ($\frac{CPU_2-CPU_1}{CPU_1} \times 100$) indicates the CPU overhead of tracing N system calls. In Fig. 6, we plot this CPU overhead as a function of N. As expected, the CPU overhead increases with N. The overhead at N=0 corresponds to the baseline overhead for Redis and WordPress, which is consistent with Fig. 4. Note that the reported CPU overhead is the *percentage* of eBPF's CPU usage incurred during the five-minute run. Thus, as the microservices remain running longer than five minutes, the overhead curve will flatten and eventually approach the constant baseline overhead, regardless of N.

Benchmark	Impact	Description
`iperf`	-3%	Network throughput
`dd`	-8.7%	Disk I/O rate
`gzip`	-0%	CPU time

TABLE I: Performance overhead.

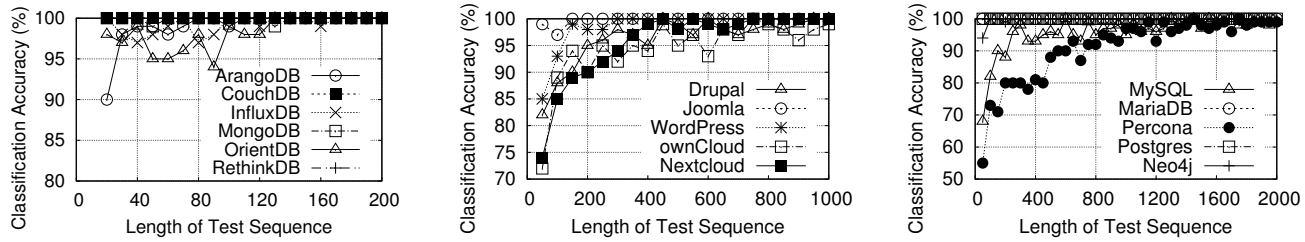
Performance overhead. The previous experiments evaluate the resource overhead of system call tracing. Now we evaluate the implication of system call tracing on the performance of the microservices being traced. To emulate network, I/O, and CPU-bound microservices, we run three benchmark programs, `iperf`, `dd`, and `gzip`, with and without system call tracing. Table I suggests that eBPF tracing introduces insignificant performance overhead for different types of workloads.

B. Microservice Classification

Next, we evaluate the classification capabilities of our Bayesian model by deploying a variety of real-world microservices on our prototype.

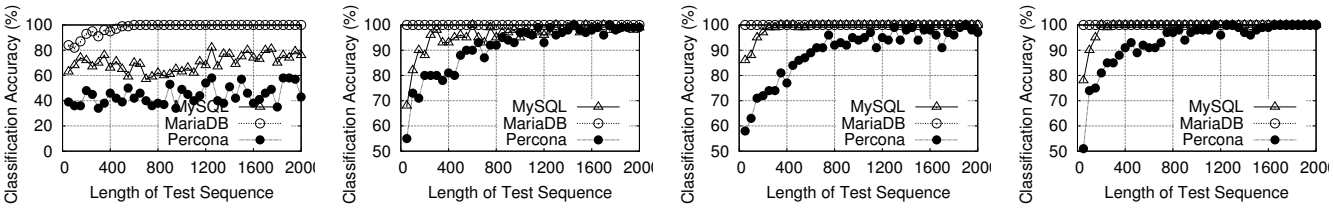
System call sequence collection. For prototype evaluation, we collect system call sequences from a total of 30 different types of real-world microservices (Table II). These are among the most popular containers in terms of downloads [26], and cover a reasonably diverse spectrum of functionalities. Some of them are more closely related than others. For example, MariaDB and Percona are binary compatible with MySQL. Similarly, Nextcloud is forked from ownCloud, inheriting many of its features. Such derivative implementations are not rare in open-source communities, but may complicate reliable fingerprinting. Other microservices such as WordPress, Drupal, Joomla, ownCloud and Nextcloud are all PHP-based web applications running on Apache HTTP server. Thus their implementations share the same set of PHP APIs and request processing mechanics of Apache HTTP server.

To collect system call sequences from these microservices, we generate synthetic workloads either by using public domain (microservice-specific) workload generators or by manually accessing the engines. When a given microservice launches multiple processes, we collect separate system call sequences from individual PIDs, and concatenate the individual per-PID system call sequences one after another to form a single long



(a) NoSQL database engine. (b) Web-based microservice engines. (c) SQL database engines.

Fig. 7: Microservice engine detection.



(a) Order-0 (b) Order-1 (c) Order-2 (d) Order-3

Fig. 8: The effect of Bayesian order dependency.

sequence per microservice. We observe that some microservices issue a particular system call consecutively many times even during idle time (e.g., `futex`, `sched_yield`). Since such workload-independent duplicate verbs can create a bias in conditional probability matrices, we restrict the maximum number of consecutive duplicates in a system call sequence to 10, and trim any subsequent duplicates. The length of the resulting sequences ranges from 30K to 600K across different microservices. We subdivide the obtained sequence for each microservice into training and test sequences. The former is fed into the fingerprint model for training, while the latter is used to evaluate the accuracy of classification performed by the model. We utilize these data sets in the rest of evaluation.

Type	Microservices
NoSQL database	ArangoDB, CouchDB, InfluxDB, MongoDB, OrientDB, RethinkDB
SQL database	MariaDB, MySQL, Neo4j, Percona, Postgres
Analytics	ElasticSearch, Telegraf
Content management	Drupal, Ghost, Joomla, WordPress, Xwiki
Key-value store	Kafka, Memcached, Redis, ZooKeeper
Proxy	HAProxy, Squid3
Storage	MinIO, Nextcloud, ownCloud, Samba
Remote desktop	Xrdp, Xvnc

TABLE II: Deployed microservices.

Classification accuracy. In the first experiment, we choose three different categories of microservices (web-based microservices, SQL databases, NoSQL databases), and investigate whether our fingerprint model can accurately identify their different implementations. In all subsequent experiments, the model used is order-1 Bayesian unless stated otherwise.

Fig. 7 plots the model's classification accuracy as a function of test sequence length. To generate many test sequences of varying length, we extract random subsequences of length N from the collected original test sequence. For each length N, we run classification using 100 test sequence samples, and compute average accuracy. One can see that our model is able to correctly classify all engines with increasing test sequence length. NoSQL database engines require less than 200 verbs for accurate classification, while classifying web-based microservices and SQL databases turns out to be more difficult, requiring longer sequences (1K to 2K). This behavior is to be expected considering the varied heterogeneity of their implementations. Among the three groups, NoSQL databases are the most diverse implementations developed in different languages (C++, Erlang, Go, Java). The web-based microservices are all written in PHP and powered by Apache HTTP server, resulting in moderate similarity. All three SQL databases are binary compatible implementations with possibly the most similar run-time behaviors. Note that the required sequence length ($\sim$1K) for web-based engines is still short enough to add insignificant CPU overhead from system call tracing (2% for WordPress; see Fig. 6).

Effect of Bayesian order dependency. As shown above, it is more difficult to classify microservices if they are functionally similar or originate from the same source code base. We now investigate whether higher-order Bayesian models are any better at detecting differences in such cases. In Fig. 8, we evaluate MySQL and its two variants, MariaDB and Percona. It shows that higher-order Bayesian models are able to classify those closely related microservices with shorter test sequences. For example, the order-1 Bayesian can detect Percona reliably with 2K verbs, while the order-3 Bayesian can achieve the same accuracy with 1200 verbs. In case of MySQL, the required number of verbs for accurate fingerprinting differs more (1K verbs for order-1 and 200 verbs for order-3). Note that a higher-order model does not affect the completion time of fingerprinting, but only increases the model's training time and memory footprint for conditional probability matrices.

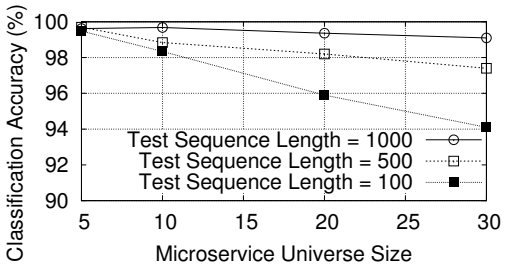


Fig. 9: Accuracy vs. microservice universe size.

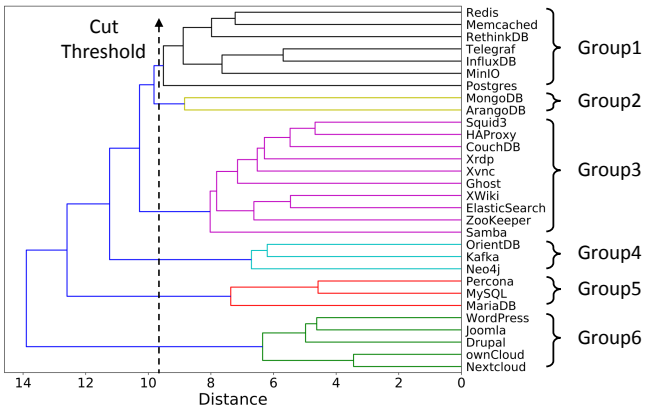


Fig. 10: Hierarchical clustering dendrogram for microservices.

Effect of microservice universe size. So far we evaluate our model using very limited types of microservices. To evaluate the model in larger-scale settings, we examine classification accuracy as a function of microservice universe size (up to 30) and plot the result in Fig. 9. For a given universe size M, we randomly choose M microservices out of 30, perform classification among them 1K times, and compute an average accuracy. The figure shows that with a bigger universe, accuracy suffers visibly more with shorter test sequences. With test sequences of length 1K, our model can fingerprint all 30 microservices of mixed similarity with 99% accuracy.

Hierarchical fingerprinting. The previous result empirically demonstrates high classification accuracy for a universe size of up to 30, but it is still difficult to extrapolate the fingerprint model's accuracy beyond that. In the next experiment, we evaluate the feasibility of hierarchical fingerprinting described in Section II-A4, as a solution to deal with an even larger-scale universe. As the first step, we perform agglomerative clustering on the 30 microservices we have evaluated so far. Fig. 10 plots the dendrogram of hierarchical clustering with Ward's minimum variance linkage. When the default cut threshold (70% of the maximum linkage) is used, hierarchical clustering results in six groups (Group 1–6) as labeled in the figure. One can see that the grouping reflects the functional similarity among different microservices to a certain extent. For example, all five web-based microservices fall into Group 6, and three binary-compatible SQL databases form Group 5.

Once microservice groups are ready, we apply a group-level fingerprint model for group identification (i.e., finding out which group a given test sequence belongs to). Fig. 11 shows

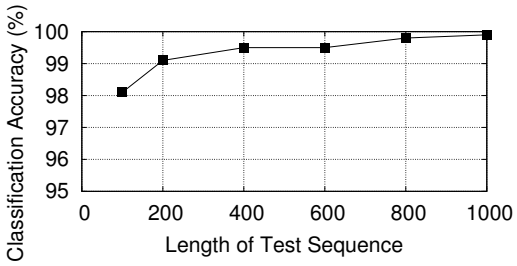


Fig. 11: Accuracy of group-level classification.

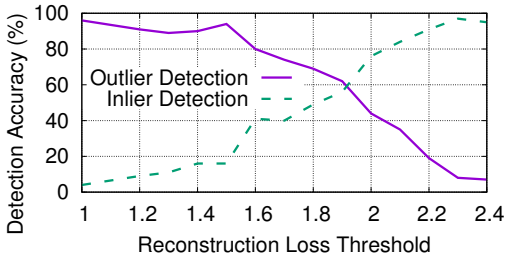


Fig. 12: Outlier/inlier detection of an aggregate autoencoder.

the accuracy of group identification. With test sequences of length 1K, group-level fingerprinting can achieve near perfect identification. Once a group is determined, one can restrict the microservice universe to those engines in that group, and perform microservice-level classification, for which we already established the accuracy. Depending on the heterogeneity of engines in the group, one can selectively apply a higher-order model for more fine-grained classification as shown in Fig. 8.

C. Outlier Detection

Next, we evaluate the accuracy of our autoencoder-based outlier detection. The requirement for an ideal outlier detection model is two-fold. Given a test sequence from an actual outlier microservice, the detection model should correctly predict that the sequence does not belong to any existing microservice (i.e., no false negative). Conversely, given a test sequence from any known microservice (i.e., inlier), the model should not erroneously conclude that the sequence comes from an outlier (i.e., no false positive). Since our outlier detection procedure in Algorithm 1 relies on the reconstruction loss produced by autoencoders, the success of outlier detection is predicated on finding a reasonable loss threshold $\hat{L}$ that minimizes *both* false-positive and false-negative errors. In the following, we evaluate this possibility using the system call data sets from the 30 microservices used previously.

Outlier detection accuracy. Before evaluating Algorithm 1, we first consider a simpler outlier detection approach as a comparison, which builds a *single* autoencoder for the entire microservice universe $\mathcal{E}$. This autoencoder, which we call an *aggregate autoencoder*, learns all possible legitimate sequences of verbs generated by existing engines E_j in $\mathcal{E}$. Then given a test sequence, it declares an outlier if a reconstruction loss returned by the aggregate autoencoder is higher than $\hat{L}$. Otherwise, the sequence is considered an inlier sample. Outlier detection is simpler than Algorithm 1, but it needs to be retrained to incorporate every newly detected outlier.

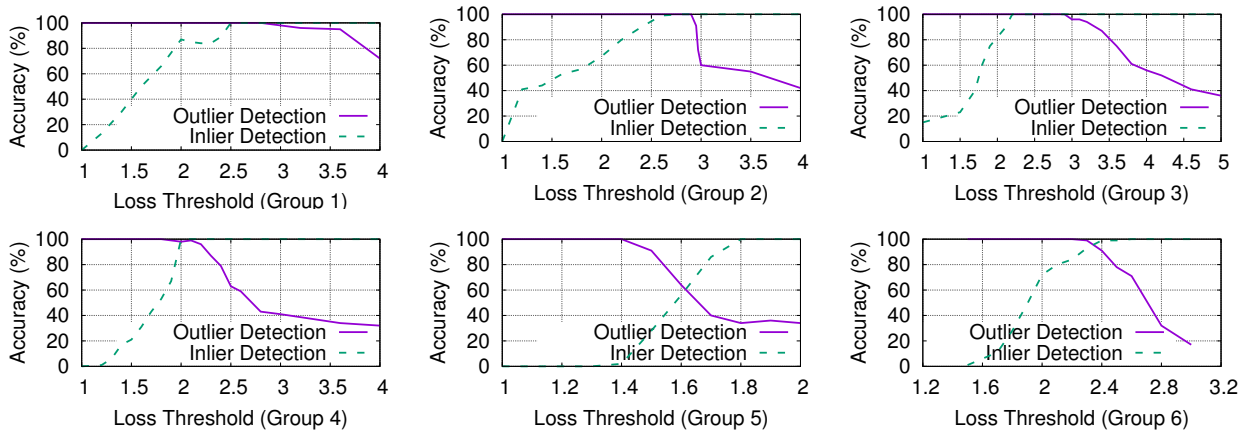


Fig. 13: Outlier/inlier detection of microservice-level autoencoders across different groups.

Fig. 12 shows outlier/inlier detection accuracy of the aggregate autoencoder as a function of reconstruction loss threshold $\hat{L}$. To estimate outlier detection accuracy, we remove one microservice out of 30, and train an aggregate autoencoder for the remaining 29 microservices. The removed microservice is then considered an outlier with respect to the trained autoencoder. We prepare 30 such aggregate autoencoders, each of which detects a missing microservice as an outlier. To evaluate inlier detection accuracy, we also train an aggregate autoencoder for all 30 microservices. Then, for a particular loss threshold $\hat{L}$, we randomly pick one microservice out of 30, treat it as either inlier or outlier, and perform inlier/outlier detection using prepared autoencoders. For each value of $\hat{L}$, we repeat this 100 times, and compute average inlier/outlier detection accuracy. As expected, if we set $\hat{L}$ too high, more outlier sequences are mistakenly classified as inliers (i.e., false-negatives). If we set $\hat{L}$ too low, more inlier sequences are erroneously detected as outliers (i.e., false-positives). More importantly, the figure clearly shows that the aggregate autoencoder is unable to minimize false-positives and false-negatives simultaneously, with the best achievable accuracy of 60% only.

The above experiment motivates training a separate autoencoder for each microservice E_j, and building a composite outlier detection procedure as described in Algorithm 1. In this case, one can leverage hierarchical two-level outlier detection, similar to fingerprinting. Using the same six groups of 30 microservices presented earlier, we apply the outlier detection procedure to each group. Similar to the above experiment, we examine the outlier/inlier detection accuracy for each group, and plot the results in Fig. 13. Unlike the aggregate autoencoder case, in Groups 1–4 and 6, one can find a range of loss thresholds for which the outlier detection procedure can achieve zero false-positive and false-negative error. Group 5 is an exceptional case though, where outlier detection performs poorly. It turns out that this is because Group 5 contains two highly similar SQL databases; MySQL and Percona. When either one is an outlier, there is high chance the detection procedure misses it because of the presence of the other in the group. We re-test Group 5 after removing either Percona or MySQL from the group. In that case, the plot becomes similar to the other five group cases, achieving 100% accuracy in a wide range of $\hat{L}$. To conclude, our experiments demonstrate the feasibility of highly accurate outlier detection for a majority of microservices tested. When outlier detection fails in some corner cases (e.g., MySQL vs. Percona), one can employ application-specific heuristics (e.g., container image tags) to disambiguate the detection.

Effect of deployment environments. The demonstrated accuracy of our autoencoder-based outlier detection is based on the data sets where training and testing sequences are collected from the same microservice deployment. This raises a question on how robust the result will be if training and test sequences are collected from *different* environments, which is a more likely scenario. To assess to what extent the autoencoder is influenced by differing deployment environments, we conduct the following experiment. We first train an autoencoder for MySQL using its original training sequences, which were collected from a standalone MySQL microservice loaded with an off-the-shelf MySQL load generator. We then re-deploy a MySQL microservice as a *backend* of actual web applications (Drupal, Joomla, WordPress, ownCloud and Nextcloud). While running, these web applications perform more realistic MySQL access in application-specific fashions. From each web application deployment, we collect separate system call data for MySQL. As a result, we obtain five different sets of MySQL sequence data.

In Fig. 14(b), we compare the cumulative probability distribution of the reconstruction loss from a MySQL's autoencoder when it is tested with five different MySQL data sets. The figure shows visibly different loss distribution curves across five different deployment environments. However, when compared to Fig. 14(a), where reconstruction losses are generated by *non-MySQL* test sequences, these differences are insignificant. In Fig. 14(c), we re-train MySQL's autoencoder with combined MySQL data sets from Drupal and Nextcloud deployments (as these two data sets yield the biggest differences in their reconstruction losses according to Fig. 14(b)). One can see that, after retraining, all five loss distributions are now more

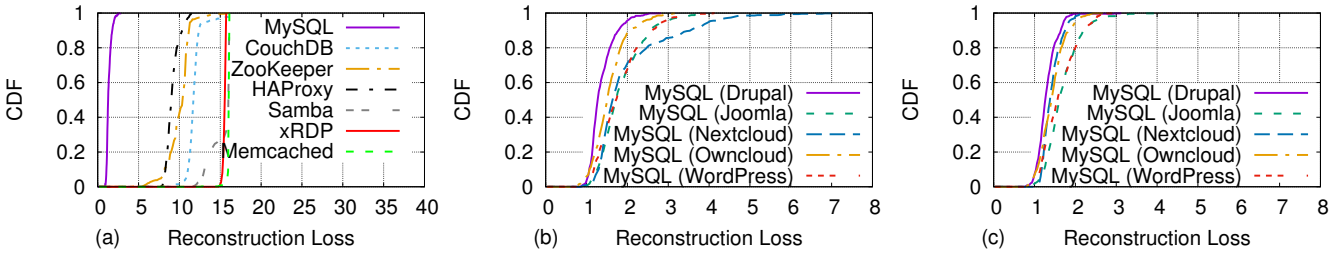


Fig. 14: Robustness of MySQL's autoencoder.

tightly matched. In short, by comparing Figs. 14(a) and (b), we can say that the current autoencoders are reasonably robust to different data collection environments. But Figs. 14(b) and (c) re-affirm the well-established importance of diversity in training data to build more robust models.

V. Related Work

System call monitoring has a long history [27] mainly in the context of anomaly-based intrusion and malware detection. A number of anomaly detection models are proposed based on system call monitoring, such as subsequence analysis [27], behavioral Markov models [28], finite state automata [29], dynamic Bayesian networks [30], and deep neural networks [31]. All these works focus on identifying any anomalous deviation from normal system behaviors, whereas our model is designed to *distinguish among multiple normal and legitimate behaviors* for fingerprinting purposes. None of these works investigates the practical aspects of their proposed models (e.g,. overhead of real-time system call monitoring) like ours does.

In [32], the author proposes an approach to enforce security policies at the system call level. In this approach, auto-generated security policies, which capture normal application behaviors, realize simple admission control on a per system call basis, without considering any dependencies across system call sequences. There are several works [33], [34] on application identification, that are based on network traffic fingerprinting. By design, the applicability of these approaches is limited to network applications, or even to a particular network protocol [33], whereas our approach is universally applicable to all types of application workload.

Addressing the outlier detection problem in the context of supervised classification has been attempted for network traffic classification [35], [36]. These works propose semi-supervised outlier detection, where a mix of labeled samples and unlabeled samples potentially from outliers are clustered together with k-means algorithm to identify an outlier cluster. These works are evaluated in a much smaller scale than ours, with less than ten classification categories.

VI. Concluding Remarks

In this paper, we explore the possibility of classifying real-world microservices using their system call level behaviors. The hierarchical Bayesian models combined with LSTM autoencoder-based outlier detection show promising results in terms of accuracy and scalability. We conclude the paper by discussing two open research problems.

Bayesian vs. LSTM models. Our approach builds on two complementary models (Bayesian and LSTM models) to achieve accurate fingerprinting and outlier detection. One valid question is whether either model can be improved to subsume the role of the other. On one hand, since LSTM models are supposed to learn complex sequential dependencies over arbitrarily long sequences, they, in theory, may be able to learn local dependencies like order-k Bayesian models. Alternatively, the LSTM-based outlier detection procedure can be redesigned based on the Bayesian approach by treating engine probabilities $P[\mathbf{v}|E_j]$ (or $log\,(P[\mathbf{v}|E_j])$) as a loss metric, and detecting outliers similar to Algorithm 1 using this metric. However, our experience is that either model alone does not achieve the same level of accuracy for classification and outlier detection as the combined models. With LSTM, we are so far unable to differentiate highly similar, binary-compatible microservices, which was already suggested in Fig. 13. With Bayesian-based outlier detection, we achieve around 90–95% accuracy for outlier detection with Groups 1–4 and 6, which is not as good as our LSTM-based approach.

High fidelity data collection. In any machine learning research, the quality of data is crucial to properly train proposed models. In this paper, we make our best efforts to collect realistic training data from different microservices by using microservice-specific workload generators or by actually operating them based on their typical workflows. Even such efforts may not ensure the representativeness of the collected data unless we deploy the microservices in the wild. To improve on the scale of data, we have also explored more systematic ways to generate synthetic sequence data using two types of generative models; Generative Adversarial Networks (GAN) and Variational Autoencoders (VAE). Unfortunately, we discovered that the accuracy of fingerprinting with such synthetic data is highly dependent on the level of randomization we introduce within these models (e.g., variance of latent variables in VAE). As an illustration, it is quite easy to achieve perfect classification using sequence data with moderately randomized sets of verbs. However, without knowing the true heterogeneity of the realistic microservice universe, we cannot claim any level of accuracy based on synthetic data. After all, a more practical approach to improve the models would be data-driven, where the models are continuously re-trained by incorporating additional inlier/outlier sequence data as they become available, so that the models' accuracy and robustness are retained. We already demonstrated the potential of this direction in Fig. 14.

APPENDIX

Probability computation with Dirichlet priors. In general, for an order-k model, the conditional probabilities for engine E_j can viewed as $|\mathcal{V}|^k \times |\mathcal{V}|$ matrix M^j where $M^j_{\mathbf{a}b} = p^j(b|\mathbf{a})$. Note that $\sum_{b\in\mathcal{V}} M^j_{\mathbf{a}b} = 1$ for all $\mathbf{a} \in \mathbf{V}^k$. The number of entries in the conditional probability matrix M^j for the order-k model is $|\mathcal{V}|^{k+1}$. Each row of the matrix M^j can be viewed as a multinomial distribution with a Dirichlet prior [37] with parameter vector comprising of $|\mathcal{V}|$ ones. In this case, we can use E_j's training sequence to compute the posterior probability estimate of $M^j_{\mathbf{a}b} = p^j(b|\mathbf{a})$. The proof of the following result is standard and is omitted.

Theorem A.1: Assuming a Dirichlet prior with parameter vector comprising of $|\mathcal{V}|$ ones for row a of M^j, the maximum likelihood estimate of the posterior distribution given the trace for an order-k dependence model, is Dirichlet with parameter

$$M^j_{\mathbf{a}b} = \frac{1 + \sum_{t=k+1}^{n_j-1} I^j(t, \mathbf{a}b)}{|\mathcal{V}| + \sum_{t=k}^{n_j-1} I^j(t, \mathbf{a})}.$$

REFERENCES

[1] "Office 365 Enterprise Solutions," https://products.office.com/en-us/business/enterprise-cloud.

[2] "Google G Suite - Enterprise Collaboration and Productivity," https://gsuite.google.com/solutions/enterprise/.

[3] "Adopting Microservices at Netflix: Lessons for Team and Process Design," https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/.

[4] C. Munns, "Microservices at Amazon," in *Proc. I Love APIs Conference 2015*, https://www.slideshare.net/apigee/i-love-apis-2015-microservices-at-amazon-54487258.

[5] M. Cebula, "Airbnb, From Monolith to Microservices: How to Scale Your Architecture," FutureStack17, 2017.

[6] M. Ranney, "What I Wish I Had Known before Scaling Uber to 1,000," in *Proc. GOTO Conference Chicago*, 2016.

[7] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," *IEEE Software*, vol. 35, no. 3, 2018.

[8] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Workload-Aware Provisioning in Public Clouds," *IEEE Internet Computing*, vol. 18, no. 4, 2014.

[9] R. Koller, A. Verma, and A. Neogi, "WattApp: An Application Aware Power Meter for Shared Data Centers," in *Proc. IEEE International Conference on Autonomic Computing*, 2010.

[10] "Cisco Application Visibility and Control (AVC)," https://www.cisco.com/c/en/us/products/routers/avc-control.html.

[11] "Junos Application Aware," https://www.juniper.net/us/en/local/pdf/datasheets/1000498-en.pdf.

[12] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia, "SDN-Based Application-Aware Networking on the Example of YouTube Video Streaming," in *Proc. the Second European Workshop on Software Defined Networks*, 2013.

[13] B. Wang, J. Su, L. Chen, J. Deng, and L. Zheng, "EffiEye: Application-aware Large Flow Detection in Data Center," in *Proc. IEEE/ACM CCGrid*, 2017.

[14] "Microsoft Cloud Discovery," https://docs.microsoft.com/en-us/cloud-app-security/set-up-cloud-discovery.

[15] "Automating Security with DevOps," https://www.illumio.com/resource-center/white-paper-automating-security-with-devops, 2018.

[16] B.-C. Park, Y. J. Won, M.-S. Kim, and J. W. Hong, "Towards Automated Application Signature Generation for Traffic Identification," in *Proc. IEEE Network Operations and Management Symposium*, 2008.

[17] "Spring Framework," https://spring.io/projects/spring-framework.

[18] "OpenAppID," https://www.snort.org/documents/openappid-detection-webinar.

[19] "A thorough introduction to eBPF," https://lwn.net/Articles/740157/.

[20] F. Murtagh, "A Survey of Recent Advances in Hierarchical Clustering Algorithms," *The Computer Journal*, vol. 26, no. 4, 1983.

[21] Y. Zhao, B. Deng, C. Shen, Y. Liu, H. Lu, and X.-S. Hua, "Spatio-Temporal AutoEncoder for Video Anomaly Detection," in *Proc. ACM Multimedia*, 2017.

[22] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi, "A Deep Learning Approach to Network Intrusion Detection," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 1, 2018.

[23] "BPF Compiler Collection," https://www.iovisor.org/technology/bcc.

[24] "vltrace," https://github.com/pmem/vltrace.

[25] A. M. Dai and Q. V. Le, "Semi-supervised Sequence Learning," in *Advances in Neural Information Processing Systems*, 2015.

[26] "Docker Hub," https://hub.docker.com.

[27] S. Forrest, S. Hofmeyr, and A. Somayaji, "The Evolution of System-Call Monitoring," in *Proc. Annual Computer Security Applications Conference*, 2008.

[28] F. Maggi, M. Matteucci, and S. Zanero, "Detecting Intrusions through System Call Sequence and Argument Analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, 2008.

[29] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," in *Proc. IEEE Symposium on Security and Privacy*, 2001.

[30] L. Fenga, X. Guana, S. Guoa, Y. Gaoa, and P. Liua, "Predicting the Intrusion Intentions by Observing System Call Sequences," *Computers & Security*, vol. 23, no. 3, 2004.

[31] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep Learning for Classification of Malware System Call Sequences," in *Proc. Australasian Joint Conference on Artificial Intelligence*, 2016.

[32] N. Provos, "Improving Host Security with System Call Policies," in *Proc. USENIX Security*, 2003.

[33] K. S. Laurent Bernaille, Renata Teixeira, "Early Application Identification," in *Proc. ACM CoNEXT*, 2006.

[34] W. Li, M. Canini, A. W. Moore, and R. Bolla, "Efficient Application Identification and the Temporal and Spatial Stability of Classification Schema," in *Computer Networks*, vol. 6, no. 53, 2009.

[35] J. Ermana, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson, "Offline/Realtime Traffic Classification Using Semi-Supervised Learning," *Performance Evaluation*, vol. 64, no. 9, 2007.

[36] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu, "Robust Network Traffic Classification," *IEEE/ACM Transactions on Networking*, vol. 23, no. 4, 2015.

[37] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*, 1st ed. The MIT Press, 2012.