syslrn: Learning What to Monitor for Efficient Anomaly Detection

Davide Sanvito, Giuseppe Siracusano, Sharan Santhanam, Roberto Gonzalez, Roberto Bifulco
NEC Laboratories Europe

Abstract

While monitoring system behavior to detect anomalies and failures is important, existing methods based on log-analysis can only be as good as the information contained in the logs, and other approaches that look at the OS-level software state introduce high overheads. We tackle the problem with `syslrn`, a system that first builds an understanding of a target system offline, and then tailors the online monitoring instrumentation based on the learned identifiers of normal behavior. While our `syslrn` prototype is still preliminary and lacks many features, we show in a case study for the monitoring of OpenStack failures that it can outperform state-of-the-art log-analysis systems with little overhead.

CCS Concepts: • **Computing methodologies** → **Anomaly detection**; • **Computer systems organization** → **Reliability**.

Keywords: monitoring, ebpf, anomaly detection, openstack

ACM Reference Format:
Davide Sanvito, Giuseppe Siracusano, Sharan Santhanam, Roberto Gonzalez, Roberto Bifulco. 2022. syslrn: Learning What to Monitor for Efficient Anomaly Detection. In *2nd European Workshop on Machine Learning and Systems (EuroMLSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3517207.3526979

1 Introduction

Monitoring the behavior of software to detect errors and issues is a critical task in any operational system deployment [4, 7–9]. A common monitoring approach is to use automated tools to continuously collect and analyze the *logs* written by the different software components [24, 25]. Logs contain rich information that can help to reconstruct the software execution flow, thereby enabling the detection of potential issues and errors. For instance, the reconstructed execution flow can be compared with the expected correct flow to identify the occurrence of an issue [2, 13, 23].

Nonetheless, deploying log analysis systems is difficult, and their ability to detect failures is limited by which logging practice was applied during development. In fact, logs parsing, interpretation and correlation are all tasks that require knowledge of an application, and they have to be repeated for each of the monitored applications, and anytime log messages change due to software updates [26]. Alternative monitoring approaches, such as building provenance graphs using Operating System (OS) event monitoring [21], are instead only used in specific cases, due to the increased monitoring overhead. In fact, provenance graphs are a tool used mostly for security-critical services and for offline analysis, since: (i) Kernel-level auditing [16] introduces significant performance overheads; (ii) the cost of updating and maintaining the graph since system boot is high and grows over time; (iii) the analysis of the (large) graph may take long time.

Our goal is to complement these existing approaches, and in some cases replace them, with an alternative that requires little domain knowledge, which is independent from software developers' practices, and which is sufficiently lightweight to be deployed in high performance scenarios. Towards this goal, we design `syslrn`. The `syslrn`'s key idea is to split the monitoring system operations in two phases: during an offline training phase the monitored software behavior is observed in details to identify key *indicators* of *normal* behavior. During the online monitoring phase, only these indicators are continuously monitored and verified, thereby reducing the monitoring overhead.

In this paper, we present a first minimalist implementation of a `syslrn` prototype, and show that it can outperform state-of-the-art log analysis systems when monitoring a complex cloud management system like OpenStack.

In particular, inspired by provenance graphs, in `syslrn` we first track software behaviors using only information available at the interface between OS and User space applications. These interfaces are stable and have a clear semantic associated with them, thereby freeing us from the need to know application-specific semantics. Furthermore, the widespread adoption of microservice architectures makes relevant internal software events visible also at this level. Therefore, we build a complete system behavior *graph* and analyze it during an offline training phase. The analysis is targeted to the identification of relevant *features* that can model the *normal* software behavior. While in a final version of `syslrn`

EuroMLSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9254-9/22/04...$15.00
https://doi.org/10.1145/3517207.3526979

we envision approaches to test multiple analysis techniques in this phase, in our current prototype we introduce a simple heuristic based on *bag-of-components kernels* and *linear regression* algorithms. The *bag-of-components* synthetically captures the structure of the software behavior graph in a vector representation. Using this representation we then build a *linear regression* model to describe the relationship between the processed workload, e.g., number of service requests, and the observed graph structure. While in future syslrn implementations we plan to complement these approaches with several other techniques, we show that this simple method is sufficient to provide a compelling anomaly detection performance in the OpenStack case study.

The analysis performed during the training phase identifies the features that characterize the software behavior, thus, during online monitoring syslrn only collects such features, thereby tailoring the monitoring to the strictly required events that identify the system behavior. Here. we collect OS-level features relying on the recently introduced Linux's eBPF technology. eBPF allows us to inject small programs at relevant Kernel *hooks*, which extract only the minimal information syslrn requires to *learn* the identified features and then perform online anomaly detection. As we will see, eBPF is efficient, which makes syslrn's monitoring overhead 10x lower than the regular logging tasks overhead.

We test our syslrn prototype monitoring anomalies in OpenStack [14], and comparing it with DeepLog [6], a state-of-the-art automated log analysis system. We perform over 900 experiments to generate a realistic dataset instrumenting a testbed to perform common OpenStack operations, such as Virtual Machine creation, storage and network provisioning. We use the fault injection framework developed by [3] to create failure scenarios, and in the process we collect both logs and the information required by syslrn. Finally, we measure the ability of our partial syslrn implementation to detect failures, and compare it with DeepLog. Our results show that the syslrn prototype, while still limited, can outperform DeepLog in this case study. It generates a significantly lower number of false positives (Selectivity 99%) than DeepLog (Selectivity 83%), furthermore, syslrn can identify a higher number of failures (Recall 98% vs DeepLog's Recall 86%). In fact, unlike DeepLog and log analysis systems in general, syslrn does not depend on what software logs [25], and it is therefore better able to profile the software behavior.

These results are encouraging, and motivate us to invest further on the development of syslrn. Given the significant effort required to generate meaningful datasets to perform research in this area, we make available our dataset of OpenStack monitoring events. The dataset includes over 900 experiments, with per-experiment duration up to 30m, in different scenarios, and the related logs and OS-level monitoring data.

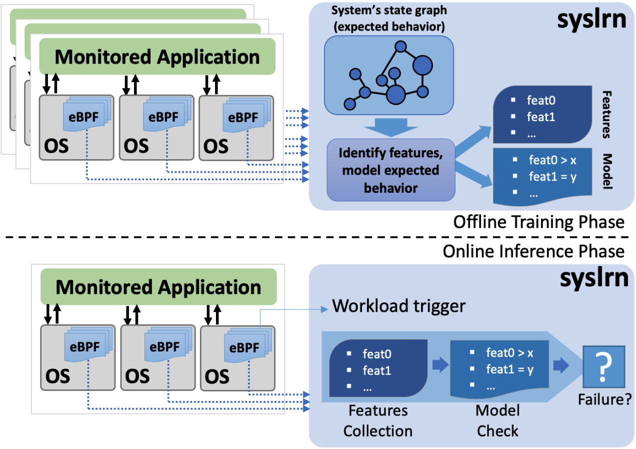

Figure 1. syslrn overview

2 Concept and Case Study

Previous research on security monitoring shows that modeling the application behavior using OS-level abstractions is a powerful tool to uncover potential issues. In fact, a complex software application[1] is typically divided into processes, with each process responsible for a subset of the application's tasks. Processes interact among them and with other processes run by the OS, or other applications, to finally achieve their goals. The type of processes, their numbers and interactions disclose relevant information about application's state and behaviour. However, during regular operations, systems have a potentially large number of processes, which makes their continuous system-wide monitoring expensive and inefficient. The size of the resulting monitoring data, and its complexity, is what limits security tools based on provenance graphs to offline uses.

Here, we observe that, unlike security monitoring solutions that need to track the entire system state evolution over time, failure monitoring solutions might often detect a faulty behavior observing a smaller set of events. For instance, a failure may be discovered observing the lack or the unexpected presence of specific processes, and relationships among them. This moves the problem from monitoring all processes and relationships, to identifying which ones characterize a normal behavior. Thus, given a target application, we need to address two problems: (i) identification of the indicators of normal behavior; (ii) definition of an effective system to perform online monitoring of such indicators.

2.1 Identifying indicators

Identifying in a large set of processes and relationships which ones are relevant to describe the application's normal behavior is a challenging task. However, we can address it offline, which gives us the opportunity to employ more sophisticated analysis techniques. That is, we can observe *normal* system

[1] We call *application* a software system that typically runs in User space, on a single or multiple nodes, to differentiate it from OS-level software.

operations for a variable amount of time, and until we collect relevant information for the analysis. For instance, this can be done temporarily instrumenting the target operational system, or by running the application in a controlled environment. Then, the information collected in this way can be analyzed offline to extract the relevant indicators, during a *training phase* (see Figure 1).

In particular, to represent the application state we resort to a graph structure, which is well-suited to capture both information about system's processes and their relationships. This finally enables the application of state-of-the-art graph analysis methods to identify common features that may serve as indicators of normal behavior. Here, the choice can be made among several methods to transform a provided graph into a *vector* of features, which captures several properties of the input graph. The extracted features are then the basis on which we apply unsupervised machine learning techniques to finally derive models of the normal system behavior. In fact, we expect significant statistical features to emerge from the collected data, since the data we monitor is mostly related to the application control flow, and less dependent on the workload dynamics.

In summary, during the training phase `syslrn`: (i) generates a graph representation of the system state; (ii) selects and applies a technique to represent the graph in a vector of features; (iii) selects and applies an unsupervised machine learning method to model the normal application behavior. The selection of the graph vector representation influences the unsupervised learning method that can be applied later, and both have direct influence on the type of monitoring performed during the online phase. In this paper we discuss a single method, but in future we envision `syslrn` to include multiple methods that can be then selected depending on the target performance goals and application properties.

2.2 Online monitoring

Once the training phase provides information about what to monitor, the online monitoring phase can start. In this phase, `syslrn` is deployed alongside the monitored system, and it is thus important to minimize its processing overhead.

The overhead depends on: (i) how monitoring data is collected; (ii) the type of *inference* performed on the collected data. In both cases, the outcome of the training phase directly influences the operations during the online phase.

To flexibly support different monitoring requirements, `syslrn` uses eBPF. An eBPF program can be dynamically attached to different in-Kernel hooks, to intercept several system events and perform simple processing on them. This allows us to tailor the monitoring to extract only the features identified during training. Furthermore, `syslrn` takes advantage of applications' external interfaces to drive the monitoring process. That is, applications usually expose external interfaces that trigger the execution of specific tasks, and the provisioning of the intended services. These interfaces, e.g., a web interface, can be easily monitored even with minimal knowledge about the application itself, e.g., at the OS network socket level. Thus, the overall set of processes and relationships that require monitoring can be further reduced when relating the monitoring to the reception of specific events, such as a new network connection.

In summary, to perform online monitoring, `syslrn`: (i) instruments data collection using eBPF programs tailored to the target feature extraction; and (ii) defines triggers to focus monitoring only on the relevant events of interest.

2.3 Case Study: OpenStack

OpenStack is a distributed cloud infrastructure orchestrator. It comprises several modules in charge of different orchestration tasks (e.g., compute, networking, storage, identity, etc), which interact among them and with third-party software to provide their services. In this paper we focus on an OpenStack deployment that includes three main modules. First, the compute module *Nova*, which is in charge of the Virtual Machine (VM) management and interacts directly with the hypervisor, to control VMs life-cycle, and with the other OpenStack components. Second, the networking module *Neutron*, which is in charge of the network provisioning, and interacts with Nova and with several network functions such as virtual switches and firewalls (e.g., provided by the OS kernel). Third, the storage module *Cinder*, which is in charge of the virtual disk management. It interacts with Nova and external filesystem management services. Overall, OpenStack is a complex distributed system with many components and inter-dependencies among them and on third-party systems. Given its central role in many cloud and telecom operator infrastructures, monitoring OpenStack and ensuring its correct operations is a task that attracted relevant research work, and which is usually used as a benchmark [2, 10, 22].

3 `syslrn`

We now introduce our preliminary `syslrn` design, with the subset of currently implemented components. We split the presentation addressing the offline training phase and online monitoring phase separately.

3.1 Training Phase

System graph. `syslrn` builds a graph of the system state extracting information from the OSes running the monitored application. Such graph contains all processes (identified by their PIDs) and any interactions among them. Interactions are of three types: process creation, inter-process and network communication. Communication interactions indicate that at least one message was exchanged between two processes residing in the same, or different, hosts. That is, the graph can equally capture interactions within a single node, or across the multiple nodes the applications might run on.

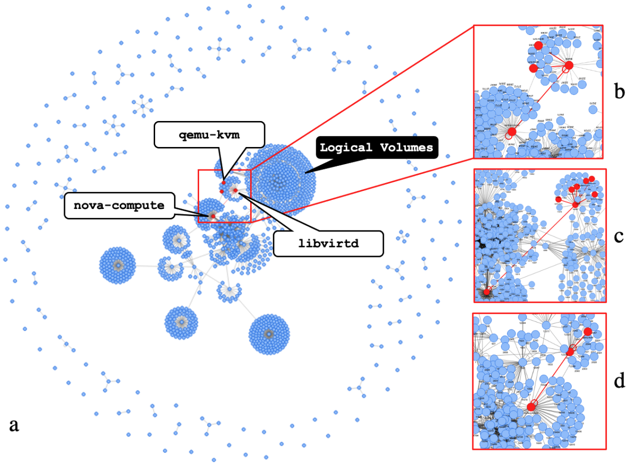

Figure 2. The system graph built by syslrn for OpenStack.

For example, Figure 2a shows the system graph for an OpenStack deployment, when handling the provisioning of a single virtual machine (VM). Peripheral nodes/components are system and background processes, while the largest connected components contain the processes related to OpenStack. Among these we can distinguish between OpenStack components (processes belonging to Nova, Neutron, Cinder, etc) and external services, i.e., processes not part of the OpenStack code base, but used to perform essential operations. External processes can provide several functionalities such as: API access (e.g., httpd), storage (e.g., mysqld), and interfaces to the VM hypervisor (e.g., libvirtd).

The system graph changes depending on the observation time and the workload served by the application. For instance, periodic Logical Volumes checks performed by Cinder are visible as one of the big clusters in Figure 2. The highlighted boxes show instead examples of dynamic interactions between internal and external processes in response to a service request. That is, when a user requests the instantiation of VMs, the request is handled by nova-compute, which interacts with libvirtd that finally communicates with the qemu-kvm process to create the VM. syslrn discovers and learns about the application behavior observing such features and their evolution. For example, when the workload is composed by a creation request for a single VM (Figure 2b), libvirtd creates two qemu-kvm instances while for three VMs (Figure 2c), libvirtd creates six qemu-kvm instances. Interestingly, if the VM creation fails (Figure 2d), only one instance of qemu-kvm is present.

Feature extraction During training, syslrn monitors the application in different states - i.e., before the startup, when in idle, while serving one or more workloads. In fact, by observing the graph changes depending on the state of the application it is possible to discover: i) the system background processes; ii) the application background/maintenance processes; iii) and the processes related to workload handling, i.e., those that answer service requests. For this last class of processes it is often important to understand their behavior in relation to the received service requests. Therefore, syslrn monitors application's service interfaces, e.g., a socket in listening mode. This enable syslrn to e.g., relate the amount of requests received with the changes in substructures (features) of the graph.[2]

To reason about graph features and thereby classify software behaviors, syslrn builds graph representations in the form of numerical vectors, called graph *embeddings* [1]. Effective ways to build graph embeddings is an area of active research, therefore syslrn can implement several approaches to address the issue, from *bag-of-components* [11] to more advanced graph representation learning techniques based on Graph Neural Networks (GNN) [17]. Focusing on the case of OpenStack, syslrn implements a graph embedding based on *bag-of-nodes*. The bag-of-nodes builds representations defining a vector with 2 dimensions for each node type (i.e., process executable names in our case): one to count the number of such nodes, and the other with the total count of their corresponding degrees (both indegree and outdegree).

Normal behavior model The embeddings are the starting point syslrn uses to learn the *normal* software behavior. Like in the previous case, several methods can be used here, e.g., hand-tuned heuristics, clustering methods, etc. In this study, we implemented a simple heuristic that looks at the relationship between the obtained graph embeddings and the number of received service requests. Intuitively, this heuristic captures cases such as the one of the qemu-kvm process described earlier: both its instances counter and its degrees counter grow linearly with the number of VM requests (Fig 2). In particular, syslrn fits a Linear Regression (LR) model for each feature, i.e., each embedding vector's dimension, and selects among them the ones for which the LR fits well the relation with the number of processed workloads. Here, syslrn uses the *Coefficient of Determination* R^2 as measure of goodness-of-fit. That is, syslrn discovers this way which features have a linear relationship with the number of received service requests. In the case of OpenStack, only 26 features out of 152 were selected from the embeddings vector. For example, among the selected features, some processes have an instance counter which is linear with the number of requests (e.g. nova, lvcreate), others have instead a degree counter which is linear (e.g. ovsdb-server) and for others both the types of counter have a linear behaviour (e.g. brctl, qemu-kvm, iscsiadm). It is worth noticing that the set of selected features includes both processes associated to the three main OpenStack components as well as generic OS processes required for the VMs operations.

[2] During training, when using synthetic workloads, this information is readily available as part of the test description.

3.2 Monitoring Phase

Features monitoring syslrn performs online monitoring using a set of small eBPF programs attached to kernel probes (kprobe). These programs can be changed at runtime, effectively leading to different monitoring instrumentation configurations. To define these configurations, syslrn backtracks the features selected during the training phase, mapping them to the OS primitive used to monitor them in first place. As mentioned earlier, this step is directly dependent on the graph embedding adopted during the training phase. For instance, to collect the 26 features required for the OpenStack case, syslrn monitors only OS' blocking stream sockets, and some process creation primitives. This boils down to the use of only 8 kprobe and corresponding eBPF programs. Furthermore, the eBPF programs record only the information needed to build the features: process name; parent PID and PID for process spawn; PIDs and network endpoints in case of communication primitives. This reduces to the minimum the overhead of executing the program each time the kprobe is invoked (i.e., the average size of the syslrn's eBPF program is 60 instructions). Most of the monitored functions are executed only at the process creation or when the communication channel is established, further reducing the monitoring overhead.

To perform the backtracking of the features, currently, the developer of a syslrn training pipeline has to explicitly define the backtracking rules. We leave the automation of this step to future work.

Anomaly detection The anomaly detection module is periodically triggered to process the monitored features to check if the collected values fit the normal behavior model learned during the training phase. For our OpenStack case study, this corresponds to checking a simple ensemble of linear regression models, which verify that each feature is linearly evolving according to the number of service requests processed by the system.[3]

4 Evaluation

We evaluate here syslrn failure detection capabilities, comparing it with DeepLog [6] a state-of-the-art log-based failure detection system. Then we perform a microbenchmark to estimate the introduced runtime overhead.

4.1 Dataset Generation

While there are public available datasets for log-based failure detection, to the best of our knowledge none of them records both OS-level events and logs. Thus, we generated a new dataset[4] that records both application logs and OS-level events during OpenStack (*Pike* version) normal and failure runs. To record failures, we extended the failure injection framework presented in [3], e.g., to support the execution of multiple concurrent requests. With this setup, we performed 935 experiments injecting a single failure point in one OpenStack component (Nova, Neutron or Cinder), while running one or more homogeneous workloads composed by all the operations needed to create, start, stop and delete a VM. The final dataset contains 190 failure-free experiments and 745 experiments where a failure was injected, with each experiment lasting at most 30 minutes.

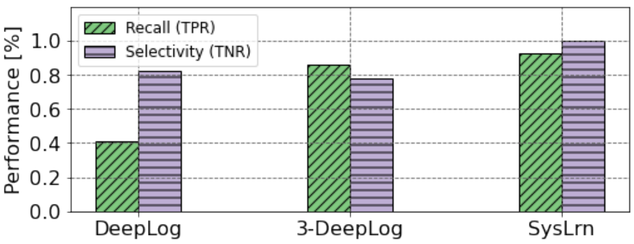

Figure 3. Failure detection performance for: DeepLog in its original configuration; 3 combined DeepLog instances, each monitoring a different OpenStack component; and syslrn

We used the generated dataset to train both syslrn and DeepLog. It should be noted that the former only uses OS-level events, while the latter only uses application logs. In our dataset we record both of them in each experiment to enable a fair comparison of the performance of the two approaches. Both systems use unsupervised mechanisms, thus we train only on failure-free data. [5] Detection performance is instead evaluated using both failure-free and fail-run experiments. We perform 10-fold cross validation to split the failure-free experiments between train/test sets. The test set includes always all the fail-runs (which are not used for training, due to the unsupervised training strategy).

4.2 Failure detection

Baselines DeepLog [6] is a log-based anomaly detection system which is based on a Long Short-Term Memory (LSTM) model trained on sequences of log messages. We compare syslrn against two different baselines. In the first one, a single DeepLog model is used to evaluate sequences of logs (identified by the instance-id) coming from a single OpenStack component (Nova). This is the default configuration of DeepLog when monitoring OpenStack [6], and we verified that on the original OpenStack dataset presented in [5] we could get similar detection performance. However, failures experienced by other components (Neutron and Cinder) are not necessarily reflected into Nova logs. Indeed, this is a typical limitation of log-based systems, which need to perform monitoring of logs belonging to different components in custom ways. We therefore designed a second baseline to

[3] As mentioned earlier, monitoring the service interface is a way to estimate the number of requests at runtime.

[4] syslrn dataset is available at [18].

[5] We share the same hypothesis of DeepLog: labeled anomalous data are hard to obtain and anomalies not in the training data might be missed by supervised methods.

overcome this limitation, using a dedicated DeepLog model per each OpenStack component, and combining their results. Here, it should be noted that it is not possible to combine all components' logs into a single logs stream, since anyway DeepLog requires a way to relate logs belonging to a common execution flow together. To do so, DeepLog uses rules based on identifiers mined within the logs. However, Nova, Neutron and Cinder have no obvious identifier among their logs that could be used to link logs from one component with those of the others.

Metrics We measure the True Positive Rate (Recall) and the True Negative Rate (Selectivity). The former tells how well the system identifies the failures, whereas the latter tells how well the system identifies non-failures. We do not report other commonly-used metrics, such as Precision, since they are misleading when considering highly imbalanced data, like in our case [19]. Figure 3 shows average results across the 10 cross-validation splits: `syslrn` outperforms both baselines for both metrics. The following subsections provide additional details.

4.2.1 Single DeepLog.

When using log sequences extracted from Nova, a single DeepLog instance cannot detect 59% of the failures, and wrongly classifies 17% of the non failure cases (False Positives). This poor performance is caused by the inability to detect Neutron and Cinder failures from Nova logs, and it is a general problem for any log-based anomaly detection system. In fact, DeepLog can detect failures when critical errors appear in the logs (e.g., *[instance <inst_id>] Instance failed to spawn*) or when logs contain incorrect event sequences (e.g., *[instance <inst_id>] Instance destroyed successfully*, followed by *[instance <inst_id>] VM Resumed*). However, (i) not all error conditions are logged with IDs required by DeepLog to build execution flow, and (ii) not all errors cause an incorrect sequence of events. Moreover, relevant events happening in the other components are not considered for the detection. For instance, Nova logs when a volume is attached to a VM (*[instance <inst_id>] Attaching volume <vol_id> to dev...*), but does not log any information on the correct functioning of the volume itself, which is instead contained inside Cinder logs. That is, some failures cannot be detected monitoring only Nova logs.

4.2.2 Combined DeepLog.

When using 3 DeepLog instances, for each component we use a different identifier to correlate logs in a flow: instance-id, network-id and volume-id, for Nova, Neutron and Cinder, respectively. The system declares an experiment as anomalous whenever at least one of the models detects an anomaly. In this case Recall jumps up to 86%, but we also observe a small reduction in Selectivity (78%), since false positives from each model are combined. Despite the significant effort we invested in trying to improve DeepLog performance in this case, we could not identify a better strategy to detect failure cases using multiple components' logs. Indeed, some error-related events may appear in the logs across components, but this information is hard to extract since such logs report different sequence ids (e.g., instance-id in Nova, and network-id in Neutron logs). Relating these logs would require extensive use of expert knowledge, to change DeepLog's log template definitions or to instrument directly OpenStack to carry the identifier across components.

Operation	Baseline (no mon)	Log-based monitoring	eBPF program (w/ user code)
SET	48.8k	17.2k (-64.73%)	47.4k (-2.78%)
GET	48.3k	17.8k (-63.43%)	47.1k (-2.61%)
LPUSH	48.6k	17.2k (-64.51%)	47.4k (-2.36%)
LPOP	49.7k	17.1k (-65.63%)	48.6k (-2.19%)

Table 1. redis-server throughput in req/s: logging vs eBPF

4.2.3 `syslrn`.

`syslrn` achieves 98% Recall, meaning that only 2% of fail runs are undetected (false negatives), and 99% Selectivity, i.e., 1% of failure-free experiments are misclassified as failures (false positives). `syslrn` has significantly better detection performance since: (i) it naturally relates failures happening to different components, thanks to the graph structure used to learn the application behavior; (ii) it detects failures that do not have any effect on logs but that do affect other processes in the system. For instance in some failures of the VM disk creation routines, logs do not show any error, yet `syslrn` detects an unexpectedly high number of `lvcreate` process instances. Likewise, some failures in the creation of Neutron's virtual router interfaces are not reflected in the logs, but they affect the number of `ovs-vsctl` commands issued in the system.

4.3 Monitoring Overhead

To investigate the overhead of running OS-level feature extraction with eBPF, we run a microbenchmark using a different application as monitoring target. In fact, the OpenStack VM generation workload has a relatively low number of service requests over time, and it is unsuitable to perform a stress test with a larger number of requests per second (it would imply the creation of many VMs) [12, 20]. Instead, we monitor Redis, a high performance key-value store that heavily relies on communication to perform operations like get, set and push and pop [15]. Redis allows us to generate stress load to show more clearly the eBPF monitor overhead.

We configured the redis-server to receive requests over a unix socket, and we used the redis-benchmark tool with 50 concurrent clients. Connection keep alive is disabled to ensure that connection operations happen on each request. The eBPF monitoring function is invoked once per each client request. We also enable/disable logging on the server side to measure the overhead of logs collection related to the requests. Table 1 shows the results. Activating eBPF monitoring introduces a 3% drop in the requests per second handled by the system. Enabling system logging reduces performance by up to 65%, an over 10x higher overhead. In most cases logs are required, however, for some performance critical deployments syslrn may provide a more efficient monitoring alternative.

5 Discussion

syslrn uses a graph representation of the OS-level events to learn the normal software behavior during an offline training phase, and then uses this knowledge to tailor the monitoring instrumentation and reduce the online monitoring overhead. In a case study focused on failure detection for OpenStack, syslrn outperforms state-of-the-art log-based anomaly detection systems, both in Recall and Selectivity. Here a key advantage is the syslrn ability to relate failures happening across different software components, and the ability to reconstruct system state beyond what is reported in the logs.

However, we presented only a preliminary prototype of syslrn, which includes a minimal subset of functionality, and our evaluation and deployment models are still preliminary. In first place, syslrn was demonstrated in a single application case study, and using a simplified subset of potential workloads. While this was enough to show the advantages of the approach when compared to other state-of-the-art solutions, a more thorough evaluation is required to properly understand syslrn benefits and limitations. We plan to test our approach on a larger set of applications, focusing on the ones based on microservices. Furthermore, in this paper we did not address issues such as the timing of the features collection and anomaly detection. In our current tests we make simplifying assumptions, performing anomaly detection leveraging information about the expected completion time of a service request. While we believe these issues can be all addressed, we have not explored them in depth. Finally, in future we plan to extend syslrn with multiple graph representation and normal behavior modeling methods, and to automate the selection among those based on the measured performance. We release the datasets used to build the results in this paper to enable the community to add and evaluate additional anomaly detection methods.

Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101017171 ("MARSAL") and No. 883335 ("PALANTIR"). This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. 2018. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering* 30, 9 (2018), 1616–1637.

[2] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, and Roberto Natella. 2020. Fault injection analytics: A novel approach to discover failure modes in cloud-computing systems. *IEEE Transactions on Dependable and Secure Computing* (2020).

[3] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. 2019. How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 200–211.

[4] Datadog. 2022. https://www.datadoghq.com/product/log-management/. Online; accessed 16-February-2022.

[5] OpenStack dataset. 2022. https://github.com/logpai/loghub/tree/master/OpenStack/. Online; accessed 16-February-2022.

[6] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.

[7] Dynatrace. 2022. https://www.dynatrace.com/platform/observability/. Online; accessed 16-February-2022.

[8] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM symposium on cloud computing*. 1–14.

[9] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. 2016. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 1–16.

[10] Xiaoen Ju, Livio Soares, Kang G Shin, Kyung Dong Ryu, and Dilma Da Silva. 2013. On fault resilience of OpenStack. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–16.

[11] Nils M Kriege, Fredrik D Johansson, and Christopher Morris. 2020. A survey on graph kernels. *Applied Network Science* 5, 1 (2020), 1–42.

[12] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 218–233.

[13] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. 2016. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 215–224.

[14] OpenStack. 2022. https://www.openstack.org/. Online; accessed 16-February-2022.

[15] Redis. 2022. https://redis.io/. Online; accessed 16-February-2022.

[16] RHEL Audit System Reference. 2019. https://access.redhat.com/articles/4409591#audit-record-types-2. Online; accessed 16-February-2022.

[17] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[18] syslrn dataset. 2022. https://github.com/nec-research/syslrn-EuroMLSys22. Online.

[19] Hoang Van Le and Hongyu Zhang. 2022. Log-based Anomaly Detection with Deep Learning: How Far Are We?. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE. to appear.

[20] Pier Luigi Ventre, Claudio Pisa, Stefano Salsano, Giuseppe Siracusano, Florian Schmidt, Paolo Lungaroni, and Nicola Blefari-Melazzi. 2016. Performance evaluation and tuning of virtual infrastructure managers for (micro) virtual network functions. In *2016 IEEE Conference on*

Network Function Virtualization and Software Defined Networks (NFV-SDN). IEEE, 141–147.

[21] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A Gunter, et al. 2020. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis.. In *NDSS*.

[22] Yong Yang, Yifan Wu, Karthik Pattabiraman, Long Wang, and Ying Li. 2020. How far have we come in detecting anomalies in distributed systems? an empirical study with a statement-level fault injection method. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 59–69.

[23] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. 2016. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 489–502.

[24] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: Enhancing failure diagnosis with proactive logging. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 293–306.

[25] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 102–112.

[26] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 807–817.