Deepstitch: Deep Learning for Cross-Layer Stitching in Microservices

Richard Li*
Facebook

Min Du†
Palo Alto Networks

Hyunseok Chang
Nokia Bell Labs

Sarit Mukherjee
Nokia Bell Labs

Eric Eide
University of Utah

ABSTRACT

While distributed application-layer tracing is widely used for performance diagnosis in microservices, its coarse granularity at the service level limits its applicability towards detecting more fine-grained system level issues. To address this problem, cross-layer stitching of tracing information has been proposed. However, all existing cross-layer stitching approaches either require modification of the kernel or need updates in the application-layer tracing library to propagate stitching information, both of which add further complex modifications to existing tracing tools. This paper introduces Deepstitch, a deep learning based approach to stitch cross-layer tracing information without requiring any changes to existing application layer tracing tools. Deepstitch leverages a global view of a distributed application composed of multiple services and learns the global system call sequences across all services involved. This knowledge is then used to stitch system call sequences with service-level traces obtained from a deployed application. Our proof of concept experiments show that the proposed approach successfully maps application-level interaction into the system call sequences and can identify thread-level interactions.

1 INTRODUCTION

As the software architecture of cloud applications increasingly migrates from monolithic designs to the container-based microservices architecture, new challenges are arising from their deployments. Among the most prominent challenges is performance diagnosis. Real-world microservice deployments often include more than hundreds of microservice instances with highly complex and dynamic interaction patterns. To better understand and diagnose such large-scale and complex microservices-based applications, most existing microservice development frameworks are equipped with distributed application-layer tracing capability [13, 19].

While distributed application-level tracing is useful to identify service-level dependencies and diagnose application performance at service granularity, the lack of kernel-level information hinders more fine-grained performance diagnosis within each service. For example, a particular application transaction may slow down due to lock contention, inefficiency in network communication, or I/O bottlenecks. These types of anomalous behaviors at lower levels are not visible to application-layer tracing.

There are three reasons why kernel-level information is missing in existing distributed application-level tracing. First, the majority of modern tracing tools follow Dapper's approach [19] that only logs high-level events, e.g., RPCs, which is sufficient to diagnose most but not all service-level behaviors. Second, instrumenting trace context that resides in userspace to collect kernel-level information would introduce much complexity to the kernel. Third, for a production system, it is impractical to propagate operational contexts across all the layers of the software stack down to the kernel, as it would introduce too much performance overhead.

On the other hand, without application-level contexts, kernel tracing alone cannot deduce useful application-specific performance insight. For example, a pure system-call-based approach (e.g., Khadke et al. [15]) can only derive whether a particular server process is under a heavy disk/network workload, with no way of attributing the workload to particular application-level behaviors.

This motivates people to enrich distributed application-level tracing with kernel-level information. For example, Ardelean et al. [1] propose propagating RPC-level context information to a kernel trace by injecting artificial system calls from applications. Sheth et al. [18] incorporate a system-call tracing layer within application-level tracing by leveraging a custom kernel module. These approaches all require

*Work performed while at the University of Utah and Nokia Bell Labs.
†Work performed while at UC Berkeley.

WOC'20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-8209-0/20/12…$15.00
https://doi.org/10.1145/3429885.3429965

modifying the tracing library to some extent, which can be burdensome especially because the tracing library is developed in multiple programming languages. In addition, they are purpose-designed to stitch only system-call traces with application-layer traces, making it infeasible to generalize for other kernel-layer information (e.g., I/O operations, stack traces, and CPU scheduling events). This weakness impedes them from embracing richer kernel-layer traces enabled by dynamic tracing technologies such as eBPF [4] to construct a more holistic view across kernel and application layers.

However, cross-layer stitching can be very challenging for several reasons. Take system call tracing as an example. First, typical microservices are multi-threaded, while application-level tracing only provides process-level information for individual application transactions. Within a process, multiple threads may perform different transactions concurrently. Second, the volume of system call traces grows very quickly, and it is difficult to find patterns to match the upper-layer events efficiently. A particular system call sequence pattern may be matched with multiple different application transactions. Third, time granularity may be different across tracing layers, complicating accurate stitching. These challenges make cross-layer trace stitching very difficult even for a single process [1].

In this paper, we tackle the cross-layer stitching problem with a deep learning based approach called *Deepstitch*. In a controlled environment, Deepstitch constructs a global view of a distributed application composed of multiple services. It learns the system call sequence patterns across all services involved using Long Short-Term Memory (LSTM) networks [12]. This knowledge is used to stitch system call sequences with service-level traces obtained from a deployed application. We demonstrate that with a global view of system call traces, the application-layer's operational context can be naturally embedded into the system call traces, thereby enriching tracing knowledge for further learning and reasoning. A feasibility study shows that this new perspective, together with deep learning, successfully stitches application and system call layer traces without modifying the tracing library or customizing the kernel.

We make the following contributions. (1) We articulate the trade-off between different kinds of cross-layer stitching techniques and highlight the practical need in a non-intrusive alternative approach. (2) We build Deepstitch, the first system to perform cross-layer stitching without patching any components in deployed microservices, tracing utilities and the kernel. (3) We evaluate Deepstitch with an e-commerce microservices-based application running on a real-world cloud deployment and report its stitching accuracy.

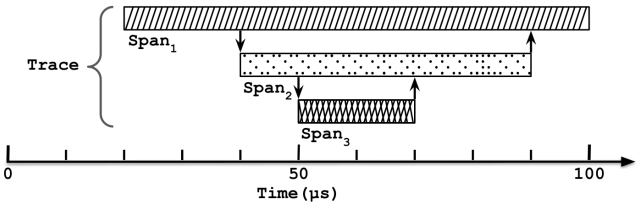

Figure 1: Trace and Spans in distributed application-level tracing.

2 BACKGROUND

Distributed application-layer tracing. Distributed tracing at the application layer allows IT and DevOps to monitor application transactions that traverse multiple microservices. It works by collecting application events in the form of *Traces* and *Spans*. As defined by OpenTracing [17], a Trace represents a particular application-level transaction triggered by an external event (e.g., a user's access request). A Span represents a particular action (e.g., an RPC call or a function call) that is executed as part of a Trace. A single Trace typically involves multiple Spans that are executed by different microservices. Conceptually, a Trace is represented by a directed acyclic graph (DAG) of Spans. Each Span records an application-level context including API name, start/end timestamps, and its relationship with other Spans. A directed edge between a Span and its child Span in the DAG indicates a caller-callee relationship. For example, Figure 1 shows a Trace composed of a nested call chain of $Span_1 \rightarrow Span_2 \rightarrow Span_3$.

Kernel-layer tracing. Tracing at the kernel layer [8] has been instrumental in many use cases, including performance troubleshooting, integrity testing, anomaly root cause analysis, security assessment, etc. The emergence of non-intrusive, flexible, and efficient tracing technology such as eBPF has made kernel-layer tracing readily deployable in production systems [9]. The available kernel-layer tracing techniques, however, have not been fully exploited in the distributed and dynamic microservice contexts. Common usage of the kernel-layer trace information is limited to a single-host analysis with raw data [10] or aggregated statistical analysis when collected for distributed systems [2, 16]. It is inherently difficult to stitch kernel-layer traces with userspace operational context in a general and unobtrusive way.

3 SYSTEM CALL SEQUENCE ACROSS SERVICES

System call tracing has been used for microservices in several different contexts. For example, the system-call-level behavior of each microservice can be profiled to detect and block any abnormal system calls possibly generated by malicious behaviors [21]. Tahir et al. [20] applied machine learning to system call traces to enable effective security monitoring.

Traditional malware analysis approaches [7] use system call traces to understand how malware infects a service instance.

What these existing works have in common is that system calls are examined from a single service's perspective, *not* as part of a pattern that extends beyond the boundary of a single service or a host. We refer to this single-service view of system call sequence as a *local view*. However, we find that there is a value in considering system call sequences *across* different services that communicate with each other. We refer to the combined set of system call traces generated across different services for application transactions as a *global view*. From the test runs of an actual microservices-based application, we find that the local view is hardly useful to characterize the system call behavior of individual services and their Spans. For example, the "GET /user" and "GET /login" operations handled by a particular web service all receive an HTTP request from another service, perform service-specific computation in userspace, and then send back a response. At the system call level, these two operations are not very differentiable.

We use an example to demonstrate why the global view of system call traces can help with pattern identification. Figure 2 shows a hypothetical application with three services, A, B, and C. Each service has three active threads represented as vertical lines, which serve requests. Time advances from top to bottom. The horizontal dashed lines represent the start/end times of three Spans belonging to a sample Trace $\mathbb{T}$. Black blocks indicate system calls triggered by the Trace, while gray blocks indicate system calls of other irrelevant Traces. We use `W` to represent *write*(), `R` for *read*(), and `X` for other system calls. The number after `R/W/X` indicates the time of the system call. The goal in this example is to figure out *which thread in each service is contributing to the Trace* $\mathbb{T}$, i.e., find the threads with black blocks.

To better characterize threads that belong to a Span based on their system calls, a natural approach would be to collect additional metadata (e.g., system call arguments) to enrich the Traces, so that any distinct pattern in individual local views becomes more pronounced. However, this approach would add more overhead to system call collection, and cannot be generalized for other types of kernel-level traces. An alternative approach would be to use machine learning to *learn* the threads that could be triggered by a Trace. However, if we train a model merely by reading the system call sequences along the vertical lines of each service, the learned patterns would be either {R-W} or {W-R}, which is hardly sufficient to differentiate the threads with black blocks from other threads. For example, both the first and second threads in service C have {R-W} within the Span, and thus both can possibly be considered the thread of interest.

Instead, our solution is to make use of the global information provided by application-layer tracing, and learn the

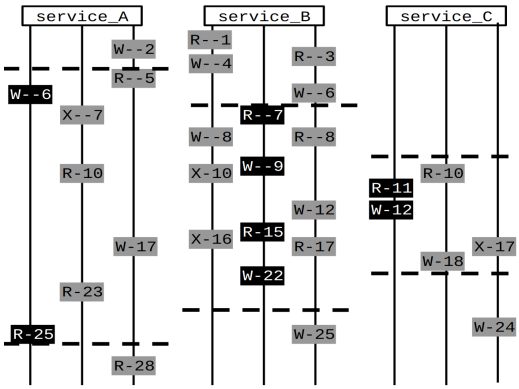

Figure 2: Hypothetical system call sequences of a trace in multi-threaded services A, B, and C.

Table 1: Enumerate possible sequences for each combination.

Combination	Global system call sequence
A1-B1-C1	Aw-Bw-Bx-Cr-Cw-Bx-Ar
A1-B1-C2	Aw-Bw-Bx-Cr-Bx-Cw-Ar
A1-B1-C3	Aw-Bw-Bx-Bx-Cx-Ar
A1-B2-C1	**Aw-Br-Bw-Cr-Cw-Br-Bw-Ar**
A1-B2-C2	Aw-Br-Bw-Cr-Br-Cw-Bw-Ar
...	...
A3-B3-C3	Ar-Br-Bw-Aw-Br-Cx

global system call sequence $\mathbb{S}$ triggered by Trace $\mathbb{T}$. Let's assume that we have already learned in advance $\mathbb{S}$ ={Aw-Br-Bw-Cr-Cw-Br-Bw-Ar}. Also assume that we know this Trace consists of three Spans, each of which is associated with a thread in a service. The learning phase is one of our major contributions, which we will elaborate in Section 4.

According to the start/end timestamps of the Spans provided by application-layer tracing, we slice the system call sequences of threads belonging to each service. We then take one thread from each service to construct a potential global system-call sequence. By enumerating all possible combinations of thread selection for different services, we can build Table 1. In Table 1, we read system call sequences according to timestamps of system calls across multiple services: on the left column, "A1" represents the first thread in service A, considered as the thread serving the request; on the right, "Aw" indicates a *write*() system call issued by the service A. The global system call sequence is read according to timestamps associated with the system calls. From this table, we can see that, when sorted by time, the system-call sequence of the combination of A1, B2, and C1 matches the expected global system call sequence $\mathbb{S}$ the best, so we can infer that A1, B2, and C1 are most likely the threads contributing to the Trace $\mathbb{T}$. In reality, any single Span may have a much larger set of system calls and there may be deeper DAGs of many Spans. This observation still holds in those environments.

4 FEASIBILITY STUDY

In this section, we describe how Deepstitch uses deep learning to learn global system-call sequence patterns for different types of Traces, which in turn are used to perform cross-layer stitching for a live application. In a nutshell, Deepstitch works as follows: (i) We first train an *idle model* for each service, which characterizes its system call behavior when the service is idle. (ii) In an isolated environment, we then systematically load the application (e.g., with a workload generator) to collect global system-call sequences for different types of Traces, and perform sanitization on the collected sequences. (iii) For each type of Trace, we train a separate model, called a *trace model*, to characterize the sanitized version of its global system-call sequences. (iv) Finally, in a live application deployment environment, we use the idle models and trace models together to perform thread prediction and cross-layer stitching.

As a proof-of-concept model construction and validation, we use an e-commerce application called Sock Shop [22], which is composed of 14 microservices developed in multiple languages (Java, Python, Go, and Node.js). We trace Sock Shop at the application layer as well as at the system call layer by using Jaeger [13] and `vltrace` [3], respectively. We generate workload for Sock Shop using `Locust` [11], and use `Kubernetes` for container orchestration.

The Sock Shop services are deployed across three Dell PowerEdge R430 servers, each with two Intel Xeon E5-2630v3 8-core CPUs, provisioned within Emulab [6]. The deep learning training and prediction by Keras [14] are performed on a Cisco UCS C240 M5 Rack Server with two Intel Xeon Silver 4114 10-core CPUs and one NVIDIA 12 GB PCI P100 GPU, within CloudLab [5]. All machines are running Ubuntu 16.04.

4.1 Learn Idle Sequences

As a baseline, we need to learn the system call behavior of an application in idle status (i.e., while not handling any workload). For example, the Sock Shop application generates 1.2 million system calls during 5 minutes of idle time. These so-called *idle sequences* are attributed to application-specific routines, e.g., periodic health check, port polling, thread yielding, garbage collection, TCP keepalive, etc. These idle sequences have their own patterns when viewed on a thread granularity. One thread in the `user` service in Sock Shop, for example, periodically generates a few *read*() and *write*() calls with *sched_yield*() in between to generate TCP keepalive for the `frontend` service.

So as the first step, we collect idle sequences of each service and train a corresponding LSTM neural network model to learn all possible patterns of its idle sequences. The system-call sequence in each thread typically follows specific patterns that can be learned by the LSTM model. As input to the LSTM model, we use n-grams that are extracted from the original idle sequences with a sliding window of length n, and label the n-grams with the (n+1)-th system call. In case there are multiple idle threads running within a service, we obtain n-grams from the idle sequence of each thread, and shuffle them up. To train a robust LSTM network, we keep the services under idle load for 48 hours to collect enough training data (more than half a billion system calls in total).

These idle models are later used by Deepstitch to eliminate idle sequences from collected system call traces. This is essential to detect whether a thread is idle or actively serving a request during particular Span periods.

4.2 Profile Trace

The next step is to profile the representative behavior of Traces at the thread level (i.e., which threads are involved for a particular Trace). To this end, we deploy Sock Shop in an isolated environment, inject different types of requests (e.g., "GET /login", "GET /orders"), one at a time, and collect Trace/Span information as well as system call traces for each request. Given the collected system call traces, we first eliminate non-Span-related idle sequences as follows. For each Span, we obtain the system call sequences of each thread in processes associated with a service. We then feed those sequences to the corresponding idle model of the service. The Keras prediction API produces a probability array that predicts the next system call after a given sequence. We use the probability of the system call that actually comes as a *prediction score*. If there is any system call within the sequence that gets a prediction score lower than a threshold, we consider the corresponding thread to be active. We call those system calls whose prediction scores are lower than the threshold (and thus can help identify active threads) *signal system calls*. We filter out all the system calls whose prediction scores are above the threshold as background activities.

Even after filtering out idle sequences, profiling a Trace is still non-trivial. The main challenge comes from the fact that there can be multiple active threads in the period of a Span. This is because multiple Spans of a Trace can be handled by different threads in parallel. So when we get active threads for a Span, some of them may actually be serving other Spans, which we call *false active threads*. To prevent the false association of threads and discount irrelevant threads from the global system-call sequences, we exploit the timestamps of individual signal system calls. More specifically, we refine the active time range of a thread by using system call timestamps. Imagine that Span_A and Span_B both detect thread_1 to be active. Suppose Span_A happens during [20μs, 30μs], while Span_B occurs during [25μs, 35μs]. Suppose that thread_1 generates the first signal system call at 22μs+100ns,

and the last call at $29\mu s$+$70ns$. In this scenario, Deepstitch considers $thread_1$ to be associated with $Span_A$ only, because its active range is not within $Span_B$. Thus this thread is considered a false active thread for $Span_B$ and excluded. With nanosecond timestamp precision, the thread active range effectively excludes most of the false active threads from Spans.

There are also cases where multiple Spans are processed within the same thread. For example, within a service, the call chain of $func_A \rightarrow func_B \rightarrow func_C$ can be represented by three Spans in the application-layer trace. In this case, there is actually only one thread active in the profiling stage, shared by all the three Spans. Since in terms of system call sequence, the active thread covers the duration from the beginning of $func_A$ to the return of $func_C$, we merge these three Spans into one and associate it with the thread. In this way, we also get a precise profile of Traces specifying how many Spans are associated with each thread in a service.

To ensure the representativeness of the profiling result, we generate, for each type of Trace, one thousand Trace instances and get a statistically stable profile.

4.3 Train Global System Call Sequences

Once we correctly identify threads that are associated with the individual Spans of a Trace, the next step is to build a deep learning model for each type of Trace. As input to the model, we create the global system-call sequences of each Trace by combining and sorting (based on timestamp) the system-call sequences from different services and threads involved. Here system calls are encoded with their service names prepended. For example, if the `orders` service generates *sendto*(), we encode it as *orders_sendto*() and map it into a numeric space. That way, we incorporate service-level interaction in the system-call sequences. We generate training data with this global system-call sequence in the same way we do with idle sequences (i.e., n-grams labeled with (n+1)-th system call). For each type of Trace (e.g., generated by "GET /user", "GET /catalogue/id", etc.), we train a corresponding LSTM model based on encoded global system-call sequences.

Clock drift does not appear to be a problem in this approach, mainly because in typical microservices, network latency is a dominant factor in application delay, and the default time synchronization protocol (e.g., NTP) can handle it well at that granularity. This observation is in line with that of previous work [1].

4.4 Identify Target Threads

The idle models and trace models presented so far are trained in an isolated environment. The final step is to utilize these models to identify threads and perform cross-layer stitching in an actual deployment environment, where multiple requests are concurrently served in Sock Shop. In this realistic environment, we can no longer assume that only one Trace is generated at a time, so we need to slice and detect all the active threads according to timestamp data provided by application-layer Spans. Since there can be multiple active Traces, the number of active threads available can be more than the count of threads of a specific Span in its profile. Among those active threads, we construct all possible combinations of them according to the Trace profile, making a table similar to Table 1. Among the listed candidates, we pick the most likely combination by using a likelihood metric, which is defined as the average prediction scores of all system calls within a sequence. Since the threads that do not belong to this specific Trace could have many system calls that get much lower prediction scores than others, they will get much lower average scores than the correct one.

Table 2: Prediction accuracy for different traces.

Trace	# Services Involved	Accuracy
GET /orders	7	91.38%
GET /catalogue/{id}	3	89.47%
GET /customers	3	89.31%
GET /login	3	84.54%
GET /health	1	74.25%

4.5 Verify Results

To verify the correctness of the thread prediction, we need labeled ground-truth data about which threads are associated with individual Spans. However, no existing application-layer tracing tools are available for this purpose, supporting multiple languages (Go, Java, Python, Node.js).

As a workaround, we use the PID information of Spans (which is already available in application-layer tracing) as label data instead. If a service is realized with multiple container replicas to serve requests, Jaeger can report which PID instance is serving which request. As verification, we ignore this ground-truth PID information provided by Jaeger, and check which process Deepstitch considers to be involved in each request. In the experiment, we spawn two instances for each service and let the `Locust` load generator send requests with a concurrency level of two to ensure that two instances of each service actively handle incoming requests concurrently. For each type of Trace, we run 100 experiments and get average prediction accuracy. The result is summarized in Table 2. It shows that Deepstitch can achieve reasonably accurate predictions. It also shows that the more services are involved in a Trace, the more accurate the prediction is. Prediction for "GET /health" is relatively less accurate because it is a periodic health-check routine that involves only a single service, and the global system call view is not applicable in this case. Essentially, Deepstitch works better when there is richer inter-service communication.

5 USE CASES

Section 4 shows that a global view of service-level interaction can be helpful for learning the patterns within the system call sequences of a Trace. This learning can in turn be used for other tasks such as cross-layer trace stitching, but without extending existing tracing tools.

Microburst troubleshooting. Performance anomalies in production systems are often caused by *microbursts* in resource usage. Troubleshooting microbursts is challenging because a microburst typically lasts for a very short duration (a few milliseconds), and yet it can cause non-negligible performance degradation. Cross-layer stitching is one viable approach to detect and explain microbursts. For example, Ardelean et al. [1] stitched system calls and application layer traces to identify what events led to a microburst in CPU utilization, which in turn introduced abnormal latency to end-to-end requests in a large production system. To do that, they had to patch both the applications and tracing libraries. With Deepstitch, an analyst can easily get the system-call sequences associated with the Trace/Span of interest, and then further understand the behavior of this particular performance anomaly at the system-call level. It is worth noting that to achieve the same goal, Deepstitch requires no modification in the kernel or tracing utility software. Deepstitch can also easily be extended to support pushing the extracted system call sequences as the value of a tag in an application-layer Span, which can be shown in the WebUI of Jaeger in a user-friendly way for analysis purposes.

Performance tuning. The system call sequences sanitized by service-specific idle models can be used as a magnifier to help application developers better understand the kernel-level interactions for individual Spans and find opportunities to optimize an existing application. For example, if one sees within a Span multiple *sendto*() and *recvfrom*() pairs between two services, one might consider batch operations or pipeline optimization to avoid unnecessary network round trips. This kind of optimization can be critical because many microservices-based applications are refactored from a monolithic version, in which circumstance this type of suboptimal interaction is not uncommon.

6 CONCLUSION

Cross-layer stitching is an emerging technology in the microservices world, where observability is one of the pillars to support highly reliable and efficient distributed systems. We argue that evolvability should be taken into consideration in existing tracing solutions to enable kernel-layer and application-layer stitching, and to embrace emerging dynamic kernel tracing. By using deep learning and a new perspective—a global view of kernel tracing aided by application-layer tracing—Deepstitch makes a first step towards a general, evolvable, data-oriented, cross-layer stitching solution.

ACKNOWLEDGMENTS

We thank Jingwen Peng at QingCloud Inc. for providing help on Kubernetes deployment. This material is based upon work supported in part by the National Science Foundation under Grant Numbers 1642158 and 1743363.

REFERENCES

[1] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *Proc. USENIX NSDI*.
[2] Cloudflare, Inc. 2020. Prometheus exporter for custom eBPF metrics. https://github.com/cloudflare/ebpf_exporter.
[3] Lukasz Dorau. 2018. vltrace: Syscall Tracer using eBPF. https://github.com/pmem/vltrace.
[4] Matt Fleming. 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/.
[5] The Flux Research Group. 2020. CloudLab. https://cloudlab.us/.
[6] The Flux Research Group. 2020. Emulab. https://emulab.net/.
[7] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. 2008. The Evolution of System-call Monitoring. In *Proc. ACSAC*.
[8] Mohamad Gebai and Michel R. Dagenais. 2018. Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead. *Comput. Surveys* 51, 2 (2018).
[9] Brendan Gregg. 2016. Linux 4.x Tracing Tools: Using BPF Superpowers. In *Proc. USENIX LISA*.
[10] Brendan Gregg. 2018. Linux Extended BPF (eBPF) Tracing Tools. http://www.brendangregg.com/ebpf.html.
[11] Jonatan Heyman, Carl Byström, Joakim Hamrén, and Hugo Heyman. 2018. Locust – A Modern Load Testing Framework. https://locust.io/.
[12] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural computation* 9, 8 (1997), 1735–1780.
[13] The Jaeger Authors. 2020. Jaeger. https://jaegertracing.io.
[14] Keras Team. 2019. Keras. https://keras.io/.
[15] Nikhil Khadke, Michael P. Kasick, Soila P. Kavulya, Jiaqi Tan, and Priya Narasimhan. 2012. Transparent System Call Based Performance Debugging for Cloud Computing. In *Proc. USENIX MAD*.
[16] Alex Maestretti and Brendan Gregg. 2017. Security Monitoring with eBPF. Proc. BSidesSF.
[17] OpenTracing Specification Council. 2017. The OpenTracing Semantic Specification. https://opentracing.io/specification/.
[18] Harshal Sheth and Andrew Sun. 2018. Skua: Extending Distributed Tracing Vertically into the Linux Kernel. In *Proc. DevConf*.
[19] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Google Technical Report.
[20] Rashid Tahir, Matthew Caesar, Ali Raza, Mazhar Naqvi, and Fareed Zaffar. 2017. An Anomaly Detection Fabric for Clouds Based on Collaborative VM Communities. In *Proc. CCGrid*.
[21] Chenxi Wang. 2016. Protect Containerized Applications with System Call Profiling. AppSec USA.
[22] Weaveworks, Inc. 2018. Sock Shop. https://microservices-demo.github.io.