Trace-based Behaviour Analysis of Network Servers

Nik Sultana*, Achala Rao*, Zihao Jin†, Pardis Pashakhanloo*, Henry Zhu*, Vinod Yegneswaran‡, Boon Thau Loo*
*University of Pennsylvania, †Tsinghua University, ‡SRI

***Abstract*—Analysing software and networks can be done using established tools, such as debuggers and packet analysers, but using established tools to analyse *network software* is difficult and impractical because of the sheer detail the tools present and the performance overheads they typically impose. This makes it difficult to precisely diagnose performance anomalies in network software to identify their causes (is it a DoS attack or a bug?) and determine what needs to be fixed.**

We present Flowdar: a practical tool for analysing software traces to produce intuitive summaries of network software behaviour by abstracting unimportant details and demultiplexing traces into different sessions' subtraces. Flowdar can use existing state-of-the-art tracing tools for lower overhead during trace gathering for offline analysis. Using Flowdar we can drill down when diagnosing performance anomalies without getting overwhelmed in detail or burdening the system being observed.

We show that Flowdar can be applied to existing real-world software and can digest complex behaviour into an intuitive visualisation.

***Index Terms*—trace analysis, network servers, denial-of-service**

I. Introduction

Network software, such as servers and protocol implementations, provide the bedrock on which widely-used services are built. Such software typically needs to securely manage resources to provide a rich array of features to large numbers of users and is architected to make efficient use of system resources. This tends to make network software complex – for example Apache and Nginx, the two most widely-used web servers, are both in excess of 100KLOC.

Because of the importance of network software, it is important to precisely diagnose performance anomalies in network software, identify their causes and fix them. To begin with, is the network to blame or is it the software? And is the performance anomaly caused by a malicious attack, an accidental misconfiguration, a bug in the software or its dependencies, or a combination of causes? Reproducing the conditions of an observed performance anomaly can be very challenging, and usually relies heavily on human ingenuity [7].

Despite its importance, network software remains challenging to analyse for its performance and behaviour. Established techniques for analysing software, such as attaching debuggers or running profilers, are not feasible in production deployments. Application performance management (APM, see Table I) is feasible in production but has restricted language, platform or visibility support – for instance it typically excludes external libraries from its monitoring.

Another technique, *software tracing* [9], [20], [26], [28] produces information that can be analysed online or offline. This technique is not only feasible in production but can also provide a detailed understanding of deployment workloads. Traces can contain a configurable amount of data, such as function parameters, and explain the software's execution.

Unfortunately, there is little support available to process traces, and nothing that specifically assists with network software. This is because systematising application-trace analysis is challenging: **(a)** It is hard to make a general facility for a class of software and for a wide range of details that could be included in traces. For example, Apache and Nginx are both web servers but their internals are completely different and their code-bases are disjoint. Which function calls can be elided in the trace, and how to distinguish different clients' sessions, differs for each implementation – possibly even across different versions of the same software. **(b)** This is an interdisciplinary problem involving the application domain, data management and understanding low-level internals; **(c)** Traces might require significant pre-processing before analysis. For instance, if an application is event-based then its trace is highly dependent on the sequence of data it is operating on.

To address this problem we present Flowdar: a practical tool for analysing software traces to produce intuitive summaries of network software behaviour by abstracting unimportant details and demultiplexing traces into different sessions' subtraces.

To reduce the problem's complexity we formulate analysis primitives to systematise trace analysis. We make Flowdar freely available as open source.[1] Flowdar can use existing state-of-the-art tracing tools for lower overhead during trace gathering for offline analysis. Using Flowdar, we can drill down when diagnosing performance anomalies without getting overwhelmed in detail or burdening the system being observed.

Flowdar can work with traces from any kind of software, but one of the benefits it brings to network software is the ability to *compare* traces gathered from different connections, sessions, or workload types. We used this feature to analyse the dynamics of application-level denial-of-service (DoS), which is very difficult to analyse using other tools.

Contributions. **(i)** We formulate five general *primitives* for analysing the traces of software, catering to the needs of network software. **(ii)** We develop an intuitive visualisation for traces based on UML sequence diagrams [18]. **(iii)** We develop Flowdar, an open source tool that implements the primitives and visualisation described in this paper, and show how Flowdar can be applied to existing real-world software to digest complex behaviour into an intuitive visualisation.

[1] https://gitlab.com/DeDos/flowdar

978-3-903176-24-9

Trace	Describes control/data-flow through software (and possibly its dependencies). This usually consists of a stream of updates about the execution's progress.
Metrics	Tuples of values made available by the OS to describe resource usage (e.g., RAM, CPU load, storage, etc), and by applications in an ad hoc way. Metrics are usually captured in regular snapshots.
Logs	Consist of time-stamped reports of system activity, and are usually streamed to an analyser or storage.
Alerts	Special messages from a system, indicating exceptional circumstances – e.g., running out of resources. Alerts are usually sent to a listener for handling.
APM	*Application Performance Management* is a class of services offered by cloud and third-party providers to monitor software running in a cloud. Cloud-provided offerings tend to be specific to a provider, and risk lock-in. Third-party offerings tend to be language-specific, use a vendor-specific API, and risk a different sort of lock-in.

TABLE I
SOURCES FOR INFORMATION ON SOFTWARE'S RUNTIME BEHAVIOUR.

```
> main:3e ngx_strerror_init:0
< main:3e ngx_strerror_init:0
> main:64 ngx_get_options:0
< main:64 ngx_get_options:0
```

Fig. 1. This example shows a *function call trace* from when Nginx starts up, and shows the memory offsets within `main` where it calls `ngx_strerror_init`other functions which then return control.

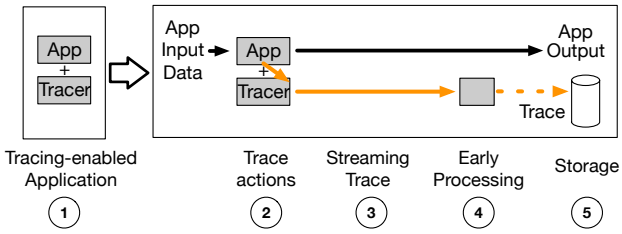

Fig. 2. Trace information flows through these stages: ① an application is enabled for tracing (e.g., through recompilation or through OS services), ② a trigger is stimulated by an input to take a tracing action, ③ trace information is emitted, then ④ this information might be lightly processed before it is stored ⑤. We reserve as much processing as possible to run offline.

II. Background and Related Work

Traces help developers to relate the program's behaviour back to the source code, and help focus debugging and optimisation efforts. They are used for *performance analysis* [11], [29], *automated testing* [12], and *system comprehension* [15]. Traces can be configured to trade overhead for detail – for example by including function parameters, return values, or part of the call stack. Fig. 1 shows an example raw trace from our experiments.

Support for tracing exists in three forms. The first is *compiler-supported* tracing, making use of special hooks provided by compiler frameworks like GCC and Clang, or implementing transformations at the source, intermediate representation, or at link-time. This is the approach used by Google's XRay system, for example [10]. The second form is *system-supported*, involving tools such as DTrace [11], eBPF [3], LTTng [4], and SystemTap [6], to insert and activate probes. The third is based on *dynamic instrumentation* and involves dynamic binary patching using systems such as Pin [24] or DynamoRIO as in instant profiling [25] and DEP [33]. Compiler-supported tracing usually does not have visibility into the kernel, while system-supported tracing does. Dynamic instrumentation could better eliminate the overhead of sometimes-enabled probles. System-supported methods try to provide safe writing of actions, and separate between providers and users of probes, while compiler-supported tracing is more ad hoc.

Tracing is organised into the stages shown in Fig. 2, after which traces are analysed. Trace analysis has been researched for trace summarisation [19], [21], visualisation [14], [26], comparison [8], factoring [16] and compression [17]. In comparison, Flowdar is designed around trace primitives derived from the pattern of needs we found in network software.

III. Generalising Trace Analysis

The limit of current practice is reached at the point where trace information is produced: then one must either use front-end tools that are limited to producing specific reports [2], or craft analysis scripts to consume the traces.

The lack of a general trace analysis framework is surprising. In comparison, log analysis has received much interest over the years, for anomaly detection for example [31]. Trace analysis bears a resemblence to log analysis [23], [27] both conceptually and also practically – for example systems like Pensieve [32] rely on logs to reproduce software faults.

As outlined in Table I, traces offer richer information about an application, and give a clearer picture of control and data flow. Logs usually contain summary information, whereas traces can contain a causal account of why the log entry was emitted.

Various trace analyses are described in the literature [8], [14]–[17], [19], [21], [26]. Our goal is to design more generic facilities for consuming tracing information, specialised to the needs of network software.

IV. Trace Analysis Primitives

While iterating through the development of Flowdar– described in the next section – we extracted general operations on traces. Due to their generality and simplicity we call these operations *primitives*.

We list 5 primitives in this section, some of which take others parameters in addition to traces. For example, the comparator primitive accepts a function parameter that implements the comparison.

a) Demultiplexing: Traces might consist of composites of several traces. This primitive splits a composite trace out into its separate constituents. The splitting is done based on a function parameter that inspects individual entries or sequences of trace entries. For example, in our experiments with Apache, we used a thread ID to split the trace emitted by an Apache process into a trace for each thread. But for Nginx, which is event-based, we looked for short signatures of trace sequences that indicate that control is being passed to a different event handler, and found unique session identifiers to demultiplex the original trace.

b) Summarising: Traces often contain information that can easily be summarised by factoring repeated behaviour. This behaviour can consist of repeated patterns of function calls. We noticed this most often for recursive functions. If a function recurses N times, then we could shorten the trace by $N - 1$ steps by indicating this recursion and omitting the information about each call. We also summarise non-recursive function call sequences, and developed two algorithms to do this: the first simply slides a window along the trace and looks for patterns, while the second tries harder by iteratively varying the size of the window.

c) Coarsening: As with summarising traces, coarsening traces loses information in order to simplify downstream analysis. This primitive is intended to shed redundant information. When tracing function calls, coarsening consists of ignoring calls beyond a given depth. As with phase-ordering in compilers [13], these primitives can affect each other according to the order in which they are applied. By losing information, it makes traces similar to each other, so we apply it carefully.

d) Comparison: Network software is perhaps the richest source of traces for comparative analysis, since the software's behaviour induced by each user, session, and connection could be compared against each other. The comparison primitive accepts two similar traces – they might have been made similar by being coarsened or summarised, using the primitives described above. These traces are then compared using a custom comparator. In our experiments we used this primitive to detect abnormal behaviour through traces. This abnormal behaviour consisted of misbehaving clients who were stalling our web server. This primitive can also accepts other parameters, such as a threshold value, to avoid making the comparison too sensitive.

e) Alignment: The comparison primitive, described above, compares traces emitted by the same code, more or less. The alignment primitive is intended to compare traces that are emitted by *different* code. Specifically, we found the need for this primitive when we traced different software that was working together: specifically an SSL proxy and an HTTP server. The SSL proxy terminated SSL connections and decrypted the incoming bytestream into cleartext queries, which were then forwarded to the HTTP server. Responses followed the opposite direction.

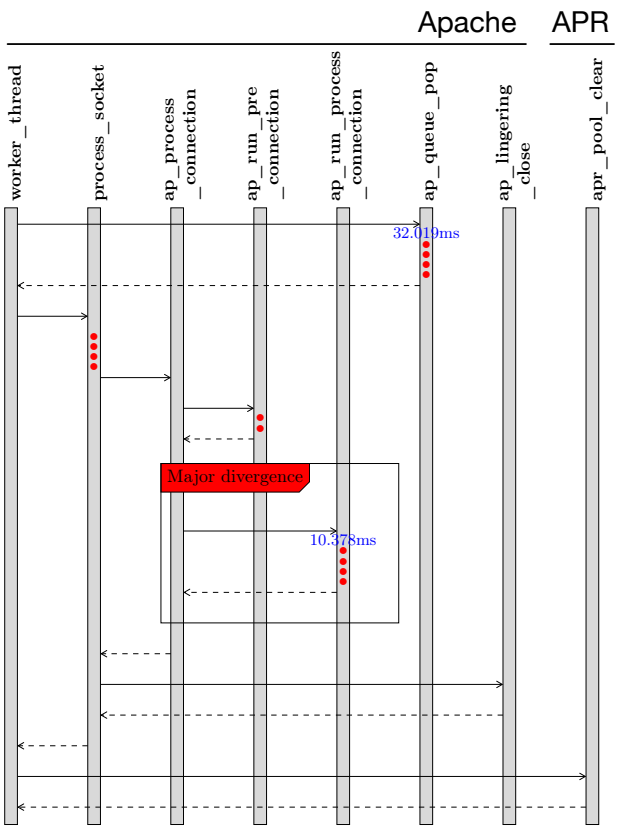

Fig. 3. Sequence diagram generated by Flowdar showing function calls and their durations from traces of Apache 2.4.26. A worker thread pops an item from the work queue, after which a chain of functions are called beginning with `process_socket`. The "Major divergence" box is a manual mark-up to indicate the point at which behaviour diverges between a normal and a misbehaving client, where we observed this function's duration to increase over $\times 400$ on average. This point was located automatically through a *horizontal analysis* (§V). Here APR refers to a support library Apache uses for portability, which we included in the tracing.

V. Flowdar

Flowdar is a trace analyser we built from the primitives described in the previous section. It consists of around 2500 lines of Python, and 1500 lines of C and C++. The latter provide tracing-support tools, such as for in-memory buffering and offline validation of traces, to check that a trace is well-formed.

In Flowdar we implemented all the stages shown in Fig. 2, as well as a conversion of trace format used by the Google XRay tracing system. We use compiler-supported tracing (§II) for stage ①, using GCC's function-call hooks as actions in stage ②. We implemented a simple and lightweight binary format that is streamed to another process for storage in ③, and can optionally filter it eagerly in ④. In stage ⑤ we store the trace in binary format. We later expand traces into an ASCII format offline, for ease of inspection and debugging.

Our analysis code is all implemented in Python, and we arranged the analyses into two categories: *vertical* analyses are applied to single traces, and *horizontal* analyses are applied to sets of traces. Our vertical analyses exploit opportunities

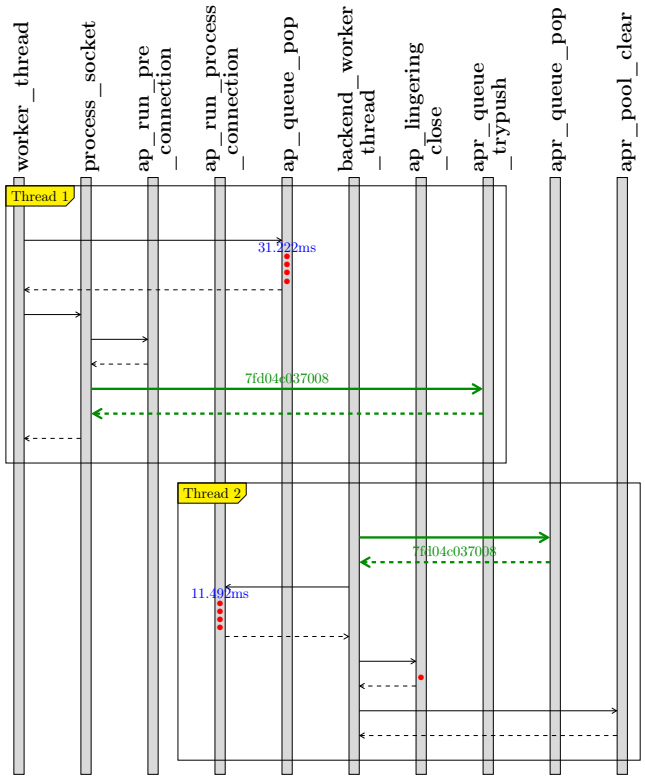

Fig. 4. We modified Apache to split worker threads into a front-half and back-half, with both halves communicating over a queue. This visualisation shows this new architecture. The value in green is the memory reference that is pushed onto the queue by `worker_thread` and then popped by `backend_worker_thread`. This value is obtained from the traces, which had included extra information for simple memory tracing in this case.

to shorten traces, and rely on the Coarsening and Summary primitives from the previous section. These analyses usually contribute towards preprocessing, summarisation, and visualisation. Our horizontal analysis partitions traces according to their similarity, or if pre-partitioned – by separating traces that were generated from known "good" and "bad" workloads – seeks to find what distinguishes the classes.

Flowdar consists of a collection of tools, several of which implement analysis primitives, and other tools that use primitives to build larger analyses. It also relies on heuristics that are specific to an application's traces. These heuristics receive application-specific parameters via the command line. We run analysis offline since it can be performance-intensive.

Horizontal analysis uses most of the analyses we implemented, and is therefore the most complex overall. It is structured into two complementary phases: first the *structural analyser* seeks to minimise unimportant differences between traces from two sets in order to magnify important ones. This minimisation is done by iterative coarsening and summarisation, either fully automatically or with user involvement. Control is then passed to the *temporal analyser* which is based on the Comparison primitive from §IV. It checks if the two traces exhibit very different time behaviour beyond a threshold parameter.

VI. Evaluation

We evaluated Flowdar on two examples of real-world open-source software: Apache and Nginx. We generated visualisations to fulfill two objectives: (i) understanding the dynamics of a DoS attack; (ii) checking the functioning of a non-trivial modification of Apache.

Apache and Nginx use different *execution models*: we used Apache in thread-based mode, and Nginx is event-based. Supporting an application with a different execution model requires a one-time manual inspection of trace samples from each application to extract markers for use by the demultiplexor primitive (§IV).

a) Understanding a DoS attack: We compared normal requests to Apache against requests from the Slowloris HTTP DoS attack, which attempts to stall connections to prevent the HTTP server from serving other clients.

Fig. 3 shows the results of summarisation, visualisation and also anomaly detection. Our visualisations also show the duration of functions: we can see that this thread spends 32ms waiting for work to become available, and later spends 10ms servicing the request. Each red dot indicates the passage of 500μs, and after 4 dots the elapsed time is shown in blue. The timings are inflated by overhead introduced by tracing but nonetheless allow us to compare a normal workload against an abnormal one.

b) Observing behaviour of modified Apache: In earlier work [30] we modified Apache to experiment with having different clients sent to different worker thread pools, to measure whether the effects of a DoS attack can be contained in a single pool. We used Flowdar to observe the hand-over between threads in successive pools along this pipeline, and the result is shown in Fig. 4.

VII. Conclusions and Future Work

A general trace analysis facility can serve our persistent need to better understand complex system behaviour. This is especially true for network software, which involves complex processing over resources shared by different sessions or users. We design a general trace analysis facility by extracting trace-analysis primitives, and show that it can digest the behaviour of real-world software into intuitive visualisations.

Future work can extend these primitives and investigate a notion of *completeness* for such primitives, based on which extensions to Flowdar can be built and evaluated. Second, our work can be extended to work with more kinds of tracing systems from §II and other formats such as CTF [1], FTA [22] or OTF [5]. And finally, Flowdar could be used or adapted to analyse traces from other domains. This would involve reusing our analysis primitives and possibly integrate them with new domain-specific features.

Acknowledgment

We thank Bob DiMaiolo and John Frommeyer for prototyping and systems help. This work is supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056 and HR0011-17-C-0047.

References

[1] Common Trace Format. http://diamon.org/ctf/. Accessed: 2018-05-12.

[2] DTrace for Linux 2016. http://www.brendangregg.com/blog/2016-10-27/dtrace-for-linux-2016.html. Accessed: 2018-05-12.

[3] Linux Enhanced BPF (eBPF) Tracing Tools. http://www.brendangregg.com/ebpf.html. Accessed: 2018-05-12.

[4] LTTng. https://lttng.org/docs/v2.10/#doc-nuts-and-bolts. Accessed: 2018-05-12.

[5] Open Trace Format. http://www.paratools.com/otf/. Accessed: 2018-05-12.

[6] SystemTap. https://sourceware.org/systemtap/. Accessed: 2018-05-12.

[7] P. Alvaro and S. Tymon. Abstracting the Geniuses Away from Failure Testing. *Commun. ACM*, 61(1):54–61, Dec. 2017.

[8] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007.

[9] T. Arts and L.-A. Fredlund. Trace analysis of erlang programs. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Erlang*, ERLANG '02, pages 16–23, New York, NY, USA, 2002. ACM.

[10] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, and N. Wang. XRay: A Function Call Tracing System. Technical report, 2016. A white paper on XRay, a function call tracing system developed at Google.

[11] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[12] F. Chang and J. Ren. Validating System Properties Exhibited in Execution Traces. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 517–520, New York, NY, USA, 2007. ACM.

[13] C. Click and K. D. Cooper. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, Mar. 1995.

[14] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252 – 2268, 2008. Best papers from the 2007 Australian Software Engineering Conference (ASWEC 2007), Melbourne, Australia, April 10-13, 2007.

[15] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, Sept 2009.

[16] M. Diep, S. Elbaum, and M. Dwyer. Reducing Irrelevant Trace Variations. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 477–480, New York, NY, USA, 2007. ACM.

[17] E. N. Elnozahy. Address Trace Compression Through Loop Detection and Reduction. *SIGMETRICS Perform. Eval. Rev.*, 27(1):214–215, May 1999.

[18] M. Fowler and C. Kobryn. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2004.

[19] A. Hamou-Lhadj and T. Lethbridge. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 181–190, 2006.

[20] D. Joumblatt, R. Teixeira, J. Chandrashekar, and N. Taft. Perspectives on Tracing End-hosts: A Survey Summary. *SIGCOMM Comput. Commun. Rev.*, 40(2):51–55, Apr. 2010.

[21] J. Kiernan and E. Terzi. Constructing Comprehensive Summaries of Large Event Sequences. *ACM Trans. Knowl. Discov. Data*, 3(4):21:1–21:31, Dec. 2009.

[22] D. Kondo, B. Javadi, A. Iosup, and D. Epema. The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 398–407, May 2010.

[23] T. Li, Y. Jiang, C. Zeng, B. Xia, Z. Liu, W. Zhou, X. Zhu, W. Wang, L. Zhang, J. Wu, L. Xue, and D. Bao. FLAP: An End-to-End Event Log Analysis Platform for System Management. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1547–1556, New York, NY, USA, 2017. ACM.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[25] S. Mahlke, T. Moseley, R. Hank, D. Bruening, and H. K. Cho. Instant Profiling: Instrumentation Sampling for Profiling Datacenter Applications. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–10, Washington, DC, USA, 2013. IEEE Computer Society.

[26] A. D. Malony, D. H. Hammerslag, and D. J. Jablonowski. Traceview: a trace visualization tool. *IEEE Software*, 8(5):19–28, Sept 1991.

[27] A. Oliner, A. Ganapathi, and W. Xu. Advances and Challenges in Log Analysis. *Queue*, 9(12):30:30–30:40, Dec. 2011.

[28] V. Paxson. Automated Packet Trace Analysis of TCP Implementations. *SIGCOMM Comput. Commun. Rev.*, 27(4):167–179, Oct. 1997.

[29] L. A. D. Rose and D. A. Reed. SvPablo: A multi-language architecture-independent performance analysis system. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 311–318, 1999.

[30] N. Sultana, A. Rao, Z. Jin, P. Pashakhanloo, H. Zhu, K. Zhong, and B. T. Loo. Making Break-ups Less Painful: Source-level Support for Transforming Legacy Software into a Network of Tasks. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, FEAST '18, pages 14–19, New York, NY, USA, 2018. ACM.

[31] H. S. Vaccaro and G. E. Liepins. Detection of anomalous computer session activity. In *Proceedings. 1989 IEEE Symposium on Security and Privacy*, pages 280–289, May 1989.

[32] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 19–33, New York, NY, USA, 2017. ACM.

[33] Q. Zhao, J. E. Sim, W.-F. Wong, and L. Rudolph. DEP: Detailed Execution Profile. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 154–163, New York, NY, USA, 2006. ACM.