System Introspection for Improving Performance, Reliability, and Security in Distributed Systems

COMP5102 Final Project

by

William Findlay

December 22, 2019

Abstract

Performance, reliability, and security are three oft-cited metrics for gauging the success of a given system. This sentiment is particularly poignant with respect to *distributed systems*, especially as our distributed systems increase in complexity and scale over time. Increasingly the problems poised by distributed computation become particularly nuanced with scale: the probability of a given node or link being in failure increases dramatically as the number of nodes and links in a network increase; unforeseen circumstances cause systems to fail; slow links and nodes introduce performance bottlenecks; and errant hardware and software can cause potentially catastrophic damage.

Understanding the root causes of performance, reliability, and security issues can greatly benefit the development of solutions to overcome them – or indeed comprise said solutions entirely. Recently, system introspection and observability have come to play a significant role in the identification of and response to performance, reliability, and security pain-points (distributed and otherwise). Furthermore, nascent technologies like *eBPF* (Extended Berkeley Packet Filter) and *XDP* (Express Data Path) in the Linux Kernel are making system introspection a particularly attractive option, due to low overhead, guaranteed production safety, application transparency, and flexibility. This literature review will present several applications for system introspection and observability in distributed systems through the examination of several case studies related to these techniques. While a particular emphasis will be placed on eBPF-based solutions under the Linux Kernel, alternative solutions will also be examined and compared in order to present a more comprehensive overview of the surrounding research.

Contents

List of Figures

List of Tables

1 Introduction

As modern systems grow more complex, so too does the task of identifying and understanding system state at a given point in time. *System introspection* refers to the concept of observing, processing, and analyzing system state in a way that is either directly useful to humans, or is at least useful to other applications in a pipeline. Potential applications for system introspection include use cases related to system performance optimization, reliability, security, and presenting system state in a way that is easy to understand.

In this literature review, we will focus on the applications of system introspection to the performance, reliability, and security of *distributed* systems, placing a particular emphasis on newer technologies like eBPF and XDP that comport with much lower overhead, higher scalability, and more breadth than previously possible. We will compare these newer technologies with other approaches such as log-based analysis and offer justification as to why eBPF proposes a more robust solution overall.

1.1 Why Trace Our Distributed Systems?

Before examining distributed systems introspection in detail, it is worth asking ourselves why we care about tracing distributed systems in the first place. In particular, what benefits can we hope to gain from tracing distributed systems and how are these benefits unique to system introspection?

As we scale our distributed systems, performance, reliability, and security become increasingly difficult to ensure at a level consistent with user expectations. Unforeseen circumstances like node failure, high latency connections, network partitioning, software bugs, security vulnerabilities, and many, many more become the *norm* rather than the *exception*. System introspection can help us to get a better of idea of the root cause of these issues and provide a means of either fixing them entirely or at least getting tighter bounds on performance and reliability discrepancies so that they become more manageable. This can either be in the form of manual intervention by developers or system administrators, or part of an automated solution for optimizing performance or mitigating problems.

In order to achieve this, we need tracing technologies that describe system state in a way that is conducive to identifying and potentially adapting to causes of performance, reliability, and security issues. Furthermore, these technologies need be efficient, and maintain their properties at scale, particularly with respect to the distributed nature of said scale.

In this literature review, we will focus on the applications of system tracing technologies

to the the following areas of distributed system optimization and analysis:

(1) *performance optimization*;
(2) *reliability improvements*; and
(3) *security benefits*.

While we will describe many different techniques for distributed systems introspection with respect to these broader themes, we will place a particular emphasis on *eBPF* (Extended Berkeley Packet Filter) [11], [19], [33], [35] and *XDP* (Express Data Path) [18], [19] under the Linux Kernel, as these tracing technologies show particular promise with respect to their efficiency, scope, power, and safety. We will then make a broader argument about the importance of eBPF and XDP to distributed systems introspection moving forward.

1.2 Structure of the Literature Review

The rest of this literature review will be structured as follows:

(1) we present an overview of system introspection and tracing technologies as they are applicable to distributed systems;
(2) we present eBPF/XDP-based solutions for improving performance, reliability, and security in distributed systems,
(3) we compare the eBPF/XDP-based solutions presented with alternatives;
(4) we discuss `Falcon` [25], [26], a general-purpose optimization framework for distributed systems that combines eBPF with other technologies in the form of a pipeline;
(5) we conclude with a discussion of the contributions of eBPF and XDP to the distributed systems landscape, as well as potential future work in this area.

2 An Overview of System Introspection

There are *many* technologies available for system introspection; each tool comes with its respective advantages and disadvantages, and some are far more suitable for *distributed systems introspection* than others. This section will discuss Classic BPF, extended BPF, and its newer cousin XDP, as well as several other system introspection options, contemporary and otherwise. By providing a general overview of the tracing landscape, we will be able to develop a clearer picture of what makes eBPF and XDP particularly attractive options with respect to distributed systems introspection.

2.1 eBPF

Extended Berkeley Packet Filter, or *eBPF*, is a relatively new system introspection technology in the Linux Kernel, dating back to a 2013 RFC [35] which was subsequently mainlined in 2014; however, the original *BSD Packet Filter*, hereafter referred to as *Classic BPF*, is actually much older, dating back to 1992 [21]. This section will provide a brief overview of Classic BPF, eBPF, and the even newer XDP [18] packet processing framework in eBPF.

2.1.1 Classic Berkeley Packet Filter (BPF)

The original incarnation of BPF was introduced to the world in a 1992 paper by McCanne and Jacobson [21] as a new system for capturing and filtering packets at the operating system level. In particular, Classic BPF consists of a virtual machine based entirely on registers for filtering packets as well as a tap mechanism to hook into the networking stack. The *tap* copies network packets and delivers them to listening filter applications, while the *filter* determines whether a packet is accepted or not.

The primary motivating factor behind Classic BPF was the desire to establish an efficient technique for capturing and filtering packets. McCanne and Jacobson showed that their approach was significantly more efficient than other contemporary packet filtering mechanisms, i.e. NIT [27] and CSPF [24]; unsurprisingly, BPF is the only one of these three systems to have stood the test of time.

Classic BPF is still used to this day in a variety of network diagnostic services, perhaps most notably *tcpdump* [41] and *libcap*. It is also worth noting that eBPF (introduced in Subsection 2.1.2) implements a superset of Classic BPF; in other words, Classic BPF forms the basis for many modern eBPF programs in Linux.

2.1.2 Extended BPF (eBPF)

Starovoitov created eBPF [35] in 2013 in order to fill what he perceived to be a gap in Linux tracing functionality. In particular, his insight was that the original BPF virtual machine offered a powerful mechanism to interface efficiently with the kernel for network stack tracing, and could be extended to interact with other kernel subsystems in a similar manner. With eBPF, we now have a powerful, safe, and performant method for tracing not only any kernel function or data structure, but any userspace function as well. Figure 2.1 summarizes eBPF's capabilities as of 2019.

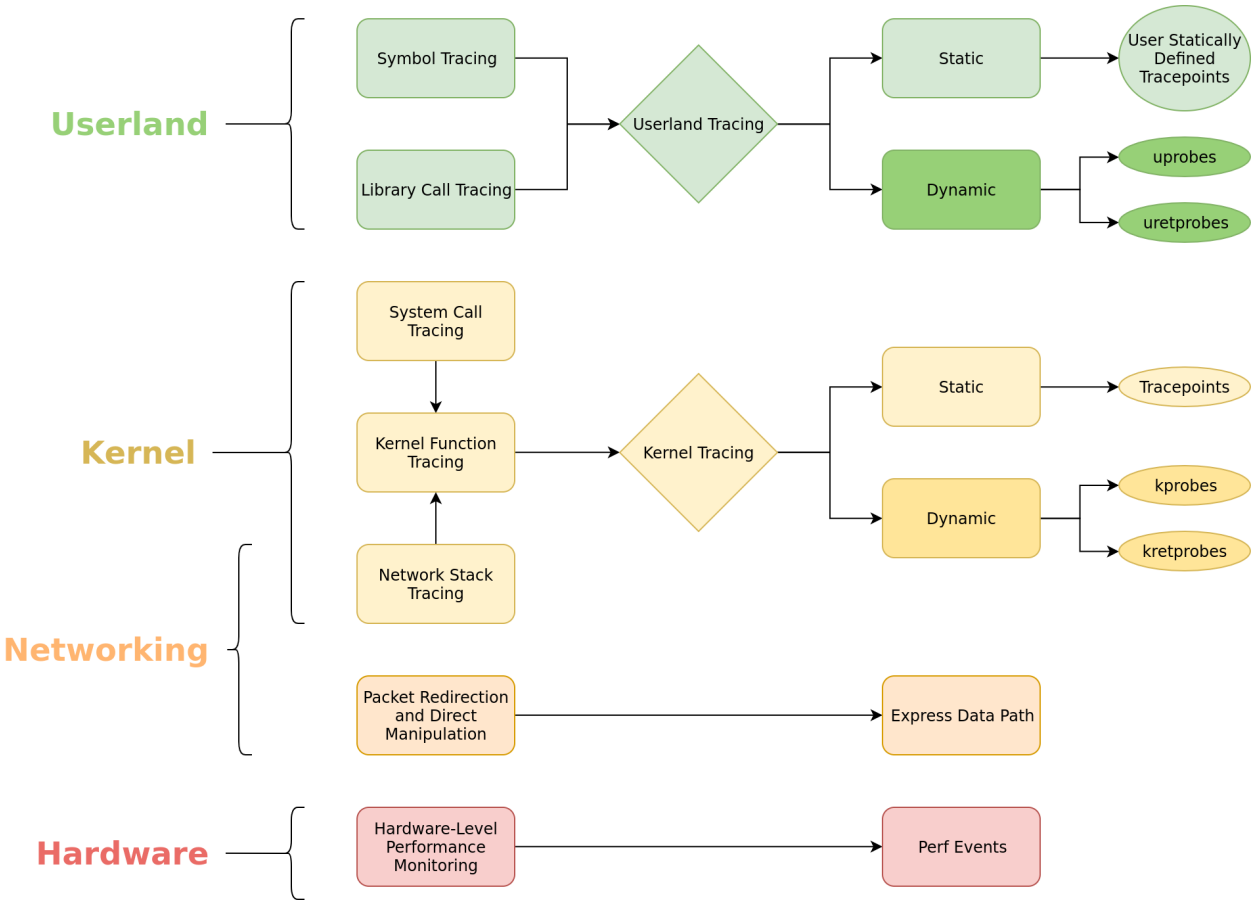

Figure 2.1: The various capabilities of eBPF. Note that we can trace any aspect of a system, from userland all the way down to the physical hardware itself through performance counters. Taken from [10] with permission.

eBPF Bytecode. Like Classic BPF [21], eBPF implements its own bytecode which is run in an in-kernel virtual machine; this bytecode, however, is greatly expanded from the original in order to support more use cases beyond simple packet introspection and filtering [35]. Unlike Classic BPF programs, eBPF programs are typically not written in raw bytecode, and are instead compiled down to bytecode from higher level languages like C. This is typically done using the LLVM compiler through higher level user APIs like `bcc` [19].

The Verifier. eBPF's safety can be primarily attributed to its use of a *verifier* [11] to ensure code safety. This verifier checks eBPF bytecode before it is submitted to the kernel in order to ensure that it will not potentially violate safety requirements and thus compromise the kernel. In order to be able to do its job, the verifier makes certain basic assumptions about the program that may not be violated. Specifically, eBPF programs are *not* Turing complete, although recent progress has been made to push the theoretical complexity limits of verifiable eBPF programs. In particular, a recent patch [6], [32], [37] introduced bounded loop support in eBPF, which marks an astonishing increase in program complexity.

eBPF Maps. In order to facilitate communication with userspace, the storage of various abstract datatypes, and a variety of miscellaneous functionality, eBPF makes liberal use of a variety of map data structures. These maps range from standard key-value hash maps to special purpose maps for packet redirection [19]. Since maps are the primary method of moving data to and from userspace, eBPF is able to mitigate the number of expensive userspace/kernelspace context switches required for communication. Table 2.1 presents an overview of the different map types available in the latest version of eBPF.

`bcc` and `bpftrace`. The recent advent of front ends for eBPF programs has rendered BPF programming easier than ever before. These front ends provide methods for the generation and compilation of eBPF programs from higher level languages, and expose userspace APIs for interacting with eBPF programs running in the kernel. `bcc` or *BPF Compiler Collection* [19] is a set of tools that expose Python3, Golang, and C++ APIs for interacting with eBPF programs written in C; it provides a reliable method for writing flexible, complex, and deeply functional eBPF programs. For simpler system introspection requirements, `bpftrace` [20] provides a higher level language for the implementation of simple eBPF programs. While it lacks the depth provided by `bcc`, it compensates by offering an API that is more conducive to simple one-liner tracing scripts.

Express Data Path (XDP). *Express Data Path* (*XDP*) [18] is a relatively new eBPF program type, initially released in 2016. It is designed for filtering packets *before* they reach

Table 2.1: Various map types available in eBPF programs. Taken from [10] with permission.

Map Type	Description
`HASH`	A hashtable of key-value pairs
`ARRAY`	An array indexed by integers; members are zero-initialized
`PROG_ARRAY`	A specialized array to hold file descriptors to other BPF programs; used for tail calls
`PERF_EVENT_ARRAY`	Holds perf event counters for hardware monitoring
`PERCPU_HASH`	Like `HASH` but stores a different copy for each CPU context
`PERCPU_ARRAY`	Like `ARRAY` but stores a different copy for each CPU context
`STACK_TRACE`	Stores stack traces for userspace or kernelspace functions
`CGROUP_ARRAY`	Stores pointers to cgroups
`LRU_HASH`	Like a `HASH` except least recently used values are removed to make space
`LRU_PERCPU_HASH`	Like `LRU_HASH` but stores a different copy for each CPU context
`LPM_TRIE`	A "Longest Prefix Matching" trie optimized for efficient traversal
`ARRAY_OF_MAPS`	An `ARRAY` of file descriptors into other maps
`HASH_OF_MAPS`	A `HASH` of file descriptors into other maps
`DEVMAP`	Maps the `ifindex` of various network devices; used in XDP programs
`SOCKMAP`	Holds references to `sock` structs; used for socket redirection
`CPUMAP`	Allows for redirection of packets to remote CPUs; used in XDP programs

the Kernel's primary networking stack, and offers a number of performance improvements over classic socket-based BPF approaches. In particular, XDP uses direct memory access to parse packet headers, which results in significant speed improvements over traditional methods; a 2018 paper by Høiland-Jørgensen et al [18] cites metrics as high as 24 million packets per second. With the addition of new map types and helpers [7], [18], [19], XDP programs can also be used to *redirect* packets to other network devices and even remote CPUs. This has clear implications for distributed systems performance optimization, and we will discuss several such systems that use XDP in coming sections.

BPF as a Programming Paradigm. BPF as we know it today has completely transcended any notions of Berkeley, packets, or filtering [15]. Instead, it has grown to encompass a wide variety of use cases ranging from firewall implementations [22] to block I/O latency monitoring [14]. While it is still capable of the original packet tracing and filtering functionality of Classic BPF [21], [35], it has clearly moved far beyond that with respect to its system introspection and performance analysis capabilities. Starovoitov envisions BPF as a powerful new programming paradigm, capable of running safe, event-driven, user-defined programs in the kernel, without compromising the integrity of the system [15].

eBPF in Distributed Systems. Increasingly, we see Linux as the growing trend for distributed computing; it powers the massive server clusters at corporations like Facebook, Google, and Netflix, and forms the basis for many modern cloud, cluster computing, and grid computing architectures; simply put, the modern Internet runs on Linux. As a core feature of the Linux Kernel, eBPF is a natural choice for monitoring our distributed systems; compound this with the fact that eBPF is completely production-safe, performant, and encompasses a rich variety of features. In Subsection 3.1, we will examine in detail the various use cases for eBPF in distributed systems.

2.2 Other System Introspection Techniques[1]

While we have established that eBPF is certainly a viable option for tracing and observing distributed systems, it remains to be seen whether it is the *best* option; eBPF is far from the only system introspection technology, and many others merit at least some discussion therein. In this section, we will highlight some techniques that are comparable with eBPF and discuss what makes eBPF a more viable choice overall.

Perf, Ftrace, Ltrace, and Ptrace. Before eBPF, classical Linux tracing was relegated to several distinct components[2], most notably perf, ftrace, ltrace, and the ptrace system call, which each present their own distinct APIs, nuances, and use cases. Perf [28] is used for monitoring hardware performance events through the incrementation of counters; its functionality is included in eBPF via direct access to hardware performance counters. Ftrace [30] presents a virtual filesystem interface for instrumenting kernel functions; eBPF provides equivalent functionality through kernel tracepoints and probes. Ltrace [31] provides library call instrumentation in userland; eBPF provides the same feature using *user statically defined tracepoints* (USDT) [12]. Finally, ptrace [29] is a system call used for instrumentation of userland processes, particularly their interaction with the kernel through system calls. The strace [38] program is a famous example of this. eBPF provides the same functionality as ptrace through system call and kernel function tracepoints; what's more, eBPF does this in a far more performant manner, since it does not need to make expensive context switches between userspace and kernelspace on each event.

Dtrace. Dtrace [4] can in many ways be thought of as the original eBPF[3]. Originally introduced in 2004 by Cantrill et al., Dtrace offered dynamic instrumentation of both

[1] This does not present a complete overview of system introspection techniques. Rather, it presents a selected sample of techniques either similar to eBPF in functionality or widely used in existing distributed systems tracing.

[2] These components still see use today, although their functionality is largely eclipsed by eBPF.

userspace and kernelspace for production Solaris systems and exposed a high level language API for doing so. So what makes eBPF different than Dtrace? At a high level, answering this question boils down to three major differences:

(1) eBPF implements a superset of Dtrace's functionality (e.g., packet filtering, XDP, perf events);
(2) Dtrace provides only a higher level interface that struggles with the implementation of more complex programs while eBPF provides both a high level interface through `bpftrace` [20] and a lower level interface through APIs like `bcc` [19];
(3) eBPF is natively supported in Linux (a huge benefit to distributed systems tracing), while Dtrace support in Linux is only available via kernel modification.

There have been some efforts to bring Dtrace to Linux, most notably `dtrace4linux` [13], which adds Dtrace functionality via a loadable kernel module that exposes an API to userspace. Full integration with eBPF has also been proposed, via the edition of a new Dtrace BPF program type [42] which seeks to add full backwards compatibility with Dtrace scripts to eBPF.

Log-Based Analysis. At a high level, log-based analysis is the concept of using system logs, logs generated by various applications, or some combination of the two to make inferences about system state. Depending on implementation, log-based approaches generally require a trade-off between application transparency and accuracy of predictions, since heterogeneous systems produce heterogeneous log data that does not lend itself nicely to uniform analysis. Many solutions also introduce non-negligible system overhead and sampling rates are often downscaled to account for this, resulting in a less complete picture of system state. We will examine many log-based solutions in great detail in Subsection 3.2, so we will not cover them extensively here.

3 Comparing Distributed Systems Introspection Approaches for Performance, Reliability, and Security

Performance monitoring and optimization are perhaps the most obvious use cases for distributed systems introspection. By monitoring system state and developing a comprehensive model, we can establish accurate records of precisely which operations are causing bottlenecks, how much time a given message m takes to arrive over link ℓ_i, the percentage of queries

[3] Although eBPF technically descends from Classic BPF, its current incarnation arguably shares more with Dtrace in terms of functionality and design goals.

delegated to a given node, and many other facets of performance. Once we have this data, the question remains as to what should be done with it.

In general, we can broadly classify performance and reliability optimization through distributed systems introspection according to three heuristics, albeit with moderate overlap in some cases. These heuristics are presented in Table 3.1, while Figure 3.1 provides an overview of the classification for techniques discussed in this section.

Table 3.1: Three heuristics for classifying distributed systems introspection with respect to performance and reliability optimization.

Heuristic	**Values**
Type of Optimization	(a) Observability focused approaches that require manual intervention (b) Automatic approaches
Level of Abstraction	(a) Host-level (b) Network-level
Implementation	Several possibilities, but we focus on distinguishing between eBPF and alternatives

In this section, we will examine many examples of system introspection for improving performance, reliability, and security in distributed systems; in Subsection 3.1, we will examine various eBPF- and XDP-based approaches and in Subsection 3.2, we will discuss alternate approaches.

3.1 eBPF Approaches

Due to its impressive scope, high degree of production safety, strong performance, and full integration into the Linux Kernel, eBPF is a natural choice for monitoring the performance of distributed systems. This becomes particularly evident when we consider eBPF from the perspective of its strong background in network tracing [21], [23], [35], but its usefulness also encompasses a variety of other performance metrics. Here, we discuss examples of eBPF's use in distributed systems performance monitoring from both an academic and a production perspective.

eBPF has seen use in production-grade distributed computing at a variety of modern technology companies. In fact, at the time of writing this paper, Facebook has approximately 40 BPF programs running on each of its servers at any given time [36] and Netflix has about 14 [15]. At Facebook, these programs in general fit into the automated performance optimization category; daemons typically listen for specific changes in system state as detected by the eBPF program and act accordingly [36]. Brendan Gregg, senior performance architect at Netflix, takes the opposite approach of running eBPF programs designed for maximal system

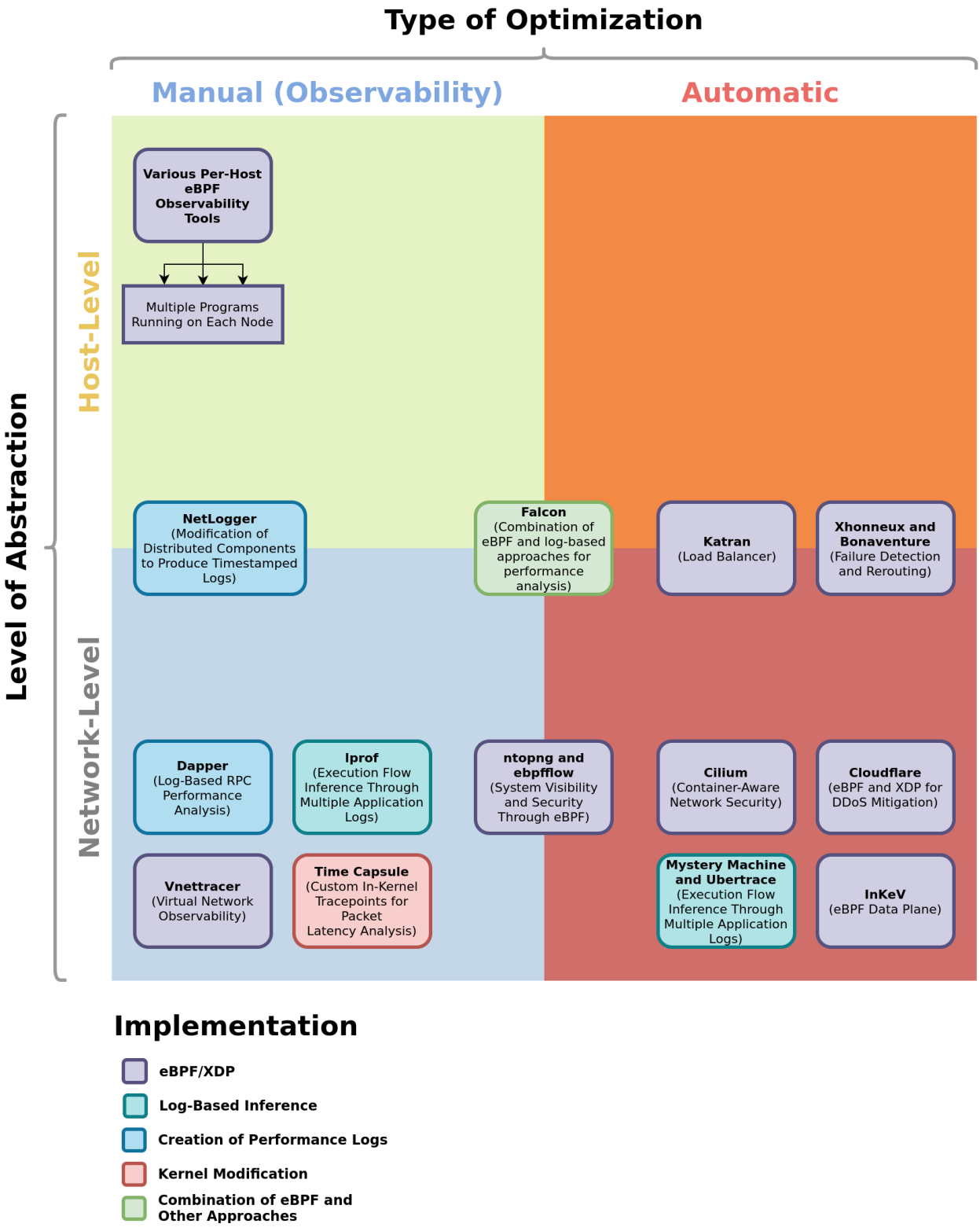

Figure 3.1: A broad overview of distributed systems introspection for performance and reliability. Techniques are classified according to the heuristics defined in Table 3.1 (note that there is overlap between categories). The reader is encouraged to return here as new systems are presented. Best viewed in color.

observability; these solutions fit into the manual response category – system administrators run the eBPF programs and act accordingly based on observations [15].

`Katran` [8] is a scalable, XDP-based load balancer used in production at Facebook to dynamically re-balance traffic between various nodes. By using an XDP program running co-located with each backend server, Facebook was able to greatly increase capacity compared to their original load balancing solution based on the IPVS kernel module [8]. The end result was a scalable load balancer capable of forwarding packets from a virtual IP to physical nodes without sacrificing performance.

Academia has also produced several examples of eBPF programs which offer promising performance and reliability advantages for distributed computing. One such example, `Vnettracer` [40], leverages eBPF tracepoints to monitor network performance, identify performance bottlenecks, and improve reliability by identifying software bugs in distributed applications. It takes advantage of several features of eBPF to do this, primarily its extremely low overhead and high degree of flexibility. Since eBPF tracepoints can be dynamically loaded and unloaded at runtime and are completely application-independent, they offer a powerful method for establishing custom performance metrics for a variety of distributed systems.

Many modern distributed applications require multi-tenant solutions capable of isolating parties from one another. Virtual networks are an important component of such multi-tenant solutions, partitioning physical networks into virtual components, isolated from one another. Implementing such virtual networks in-kernel is often costly, due to expensive context switching and packet copying from kernelspace to userspace; custom in-kernel solutions also present the risk of compromising system state due to bugs and security flaws. `InKeV` [1] takes a new approach to solving the problem of virtualized networks, using eBPF-based data plane functions. Thanks to eBPF's low overhead and high safety, `InKeV` presents a fully-programmable, performant, and production-safe data plane solution for network virtualization in distributed applications. Perhaps the most impressive contribution here is that `InKeV`'s eBPF-based implementation retains the performance benefits of an in-kernel data plane implementation without sacrificing the programmability available in userspace implementations [1].

eBPF-based data plane functions also have use cases in the realm of node failure detection and fast re-routing response. Xhonneux and Bonaventure [43] described a system for failure detection and fast re-routing in the SRv6 data plane. Consistent with the other systems we have described here, the primary advantages touted for eBPF in this context were high performance (consistent with previous kernel-based implementations) and robustness,

particularly when compared with traditional userspace implementations. Since node failure is a frequent issue in distributed systems (particularly as these systems scale), the benefits of fast and robust failure detection and re-routing in distributed computer networks cannot be understated.

Miano et al. [23] discuss the development of several complex eBPF network services like the ones we have described above, and attempt to offer insight into both the positive aspects of eBPF that contributed to their success as well as any pitfalls related to eBPF implementations (particularly with respect to limitations introduced by the verifier). In particular, they cite eBPF's inclusion in vanilla Linux[4], dynamic loading and compiling of BPF programs, support for arbitrary service chains, and the efficiency and power of XDP. What is particularly noteworthy however, is the *disadvantages* they cite – namely that many of these disadvantages have been addressed since the time the paper was originally written. Table 3.2 presents an overview of the disadvantages presented by Miano et al. and describes how each one has been refuted in the latest versions of eBPF. This highlights an important trend with respect to eBPF's trade-offs between safety and complexity; namely, these trade-offs are improving *significantly* and *rapidly* over time.

Table 3.2: Some of the disadvantages cited by Miano et al. in their paper along with how these disadvantages have been refuted in subsequent versions of eBPF.

Disadvantage Cited by Miano et al. [23]	**Refutation**	**Linux Version**
Limited to 4,096 instructions per BPF program	Limit has been raised to 1,000,000 instructions for privileged users	5.2
Limited support for loops	Full bounded loop support (without compile-time unrolling) has been introduced	5.3
No support for locks for concurrency control	eBPF now supports spinlocks in a limited subset of program types	5.1

eBPF also has promising implications with respect to distributed systems security, particularly with a focus on network security from a host-based perspective. Cloudflare [3] is now using eBPF and XDP as a major part of their DDoS mitigation pipeline, replacing their previous methodology of combining Iptables with Classic BPF. This is largely thanks to XDP's ability to efficiently implement complex packet filtering functionality with minimal overhead. Deri et al. [9] discuss the implementation of `ebpfflow` that uses eBPF for traffic event generation and `ntopng`, a tool that leverages the former to provide increased network visibility and analysis capabilities to end users, such as system administrators. Cilium [2]

[4]Thanks to Linux's prevalence with respect to distributed system implementations, this is a quintessential boon.

uses eBPF to provide container-aware networking security for production software running in multi-tenant container-based systems. This is particularly relevant in today's climate where distributed systems are tending towards container-based approaches for multi-tenant isolation.

3.2 Alternate Approaches

The eBPF approaches described in the previous section were noteworthy primarily for three reasons when compared with other solutions:

(1) low overhead;
(2) production-safety and robustness; and
(3) flexibility, scope, and programmability.

In this section, we will examine some other approaches to system introspection in distributed systems with respect to performance optimization and reliability. We will attempt to contrast these with the eBPF and XDP solutions described in the previous section and show why eBPF offers a more attractive approach overall.

A large portion of the literature related to distributed systems performance and reliability analysis comes in the form of log-based introspection. Notably, much of this literature comes from *before* (or around the same time as) the advent of eBPF in 2013, although some of it is more recent than that. In general, log-based analysis can be effective, but struggles in competing with other solutions in terms of both efficiency and depth; that is, we can *only get so much* information from parsing system logs (even custom ones) and we can *only do it so quickly*.

Distributed log-based approaches to performance and reliability generally follow one of two patterns:

(1) they monitor events[5] and generate uniformly formatted log data for analysis; or
(2) they parse existing logs generated by distributed applications and attempt to reconstruct flow of events.

In the first category, we have systems like `NetLogger` [16], [17] and `Dapper` [34]. The key observation behind these systems is simple enough: distributed systems are complex, and generally require the integration of many software and hardware components, which can often interact in unpredictable ways and don't necessarily present the same level of detailed

[5]Generally either by modifying distributed application code or event generation through a per-host userspace daemon.

debugging information to the user; as such, an approach that can unify components and consolidate performance data into system logs should help to solve this problem. Unfortunately, this solution is not without its drawbacks.

First, we consider application transparency consequences – ideally, we want our tracing systems to be fully transparent; that is, from the application developer's perspective, the interface into a system which is being traced should be identical to that of a system which is not being traced. In `NetLogger` [16], [17], this is simply not the case. In order to instrument an application in `NetLogger`, developers needs to make calls to the `NetLogger` API at strategic points in their distributed applications; the refactoring overhead[6] related to such a task is certainly non-trivial, and the instrumentation of applications for which one does not possess the source code (with the exception of instrumenting shared libraries) becomes impossible. On the other hand, eBPF tracing is *entirely* application transparent, and it remains so even as our instrumentation needs cross the boundary between kernelspace and userspace thanks to *user statically defined tracepoints* (USDT) [12].

While `Dapper` [34] takes a similar approach of providing an application-level API for instrumentation, they combine this with a per-host daemon capable of generating generic log data for RPC invocations on each host. This approach adds a base level of application transparency to `Dapper` deployments; every system is traced to a certain degree, and these traces are augmented with per-application solutions in the form of optional *RPC annotations* as necessary. However, this leads into a second important consideration with respect to log-based analysis: it is *slow*. In fact, to maintain acceptable overhead, Google purposely limits the rate at which `Dapper` collects system data by tracing system events with configurable random probability; this is a hugely suboptimal approach for the obvious reason of simply missing out on potentially valuable data. As we have seen, eBPF's competitive advantage here is that it is capable of system wide introspection while introducing minimal overhead; this makes it a perfect substitute for purely userspace solutions like `Dapper`'s daemon.

Thus far, we have seen distributed tracing systems that generate events in userspace to produce custom logs. Now, we will examine the alternative approach of parsing existing log data from distributed applications in order to reconstruct events retroactively; `lprof` [44] and `Mystery Machine` [5] are two such systems. While this approach clearly does not suffer from the need to intrusively modify applications as we have seen in prior log-based solutions, it remains to be seen whether inference of control flow from various system logs presents a complete solution. For instance, inferential techniques may work well for a certain subset

[6]Here, we refer to overhead in the sense of spending developer time and money, rather than application overhead.

of applications, but even one or two applications that are resistant to static analysis can potentially cause serious performance bottlenecks to go completely undetected.

`lprof` [44] achieves its event extraction through static analysis of binaries in order to determine how to parse individual log files. It boasts a precision rate of 88% and a performance anomaly diagnosis rate of 65% in empirical studies conducted by the authors. This is certainly an impressive feat given that the entirety of this analysis is done using inferred data from heterogeneous log data. However, as we discussed above, this solution is not strictly extensible to an arbitrary number of distributed applications; it is enough to have one performance bottleneck that is immune to `lprof`'s static analysis for the entire solution to fall apart.

`Mystery Machine` [5] attempts to address the concern of scalability under heterogeneous applications by eliminating the static analysis component present in `lprof` entirely. In order to do this it leverages existing logging infrastructure at Facebook known as `Ubertrace`. These `Ubertrace` log messages have a predictable structure and can be used to construct causal relationships between distributed events in response to Facebook requests. However, what we again find is that this method does not offer a *complete* reconstruction of system state because a certain proportion of operations go unlogged by `Ubertrace`.

What we have seen in the above two examples is a clear indication that log-based performance analysis is incapable of inferring a complete and accurate model of system state. In contrast, eBPF is capable of monitoring as much or as little of system state as desired. A tracepoint in eBPF is *guaranteed* to capture 100% of the events that it instruments, and we can define such tracepoints for arbitrary functions in both userspace *and* kernelspace. This allows for both coarse and fine-grained analysis of production systems without missing any important details.

We now turn our attention to production tracing systems that rely on kernel modification instead of log-based analysis. Such systems would alter the source code of whatever kernel they are residing on in order to instrument carefully chosen tracepoints (e.g. at key parts in the networking stack). `Time Capsule` [39] takes such an approach for measuring packet latency in distributed applications; it does this by instrumenting custom tracepoints and key locations in the network, timestamping packets as they are received by embedding the information into packet payloads. The authors cite many of the problems with log-based approaches that we discussed above as motivation for their approach. However, `Time Capsule`'s method is not without its own limitations; namely, Dom0 and DomU kernels need to be recompiled in order to instrument new manually-defined tracepoints. The authors do mention the possibility of migrating such a solution to use eBPF for dynamic instrumentation, but cite its inability to

modify packet payloads as the primary deterrent for such an implementation; as we have seen, eBPF is now *fully capable* of modifying packet payloads thanks to XDP, so revisiting the potential for such an approach may be a topic for future work.

3.3 `Falcon`: Combining eBPF with Other Approaches

We have examined a variety of systems that employ a diverse range of techniques for performance and reliability analysis and optimization. In particular, we have emphasized eBPF- and XDP-based solutions and contrasted them with the highly popular technique of log-based analysis; what we have not yet considered, is a *unified approach* between eBPF and event logging that offers the best of both solutions. `Falcon` [25], [26] presents one such implementation in the form of a pipeline that joins common system tracing techniques, including eBPF, and unifies them in a standardized event log for analysis.

Earlier in this literature review, we presented a variety of drawbacks to common log-based techniques for performance analysis; this included difficulty analyzing heterogeneous log data, suboptimal performance, limited application transparency, and gaps in event logging data. `Falcon` *addresses each of these concerns* through clever use of a tracing pipeline that incorporates technologies like eBPF in order to generate a seamless, system-wide event stream in a performant, production-safe manner. It then uses this event stream to generate detailed system logs for further analysis.

Approaches like `Falcon`'s may well represent the future of distributed systems introspection. In fact, introducing eBPF into the methodology of many of the alternative approaches we have looked at offers a great deal of promise with respect to ameliorating their respective shortcomings. Moving forward, we are likely to see eBPF incorporated into more and more distributed systems instrumentation solutions, particularly as the technology continues to improve at a rapid rate.

4 Conclusion and Future Work

We have discussed several applications of system introspection to improving the performance, reliability, and security of distributed systems. Conventionally, this space has been dominated by log-based approaches that present largely suboptimal solutions, particularly with respect to system heterogeneity, performance degradation at scale, and low application transparency. As we have shown, eBPF solutions offer a promising method to trace distributed system performance in a production-safe, scalable, and dynamic manner that covers as much or as

little behavior as desired. The integration of eBPF solutions with traditional approaches as done in `Falcon` [25], [26] presents a promising topic for future work in this field. Many existing distributed systems introspection technologies can benefit from eBPF's unique set of characteristics, and these potential benefits will only improve as eBPF gains more functionality and complexity.

References

[1] Z. Ahmed, M. H. Alizai, and A. A. Syed, "Inkev: In-kernel distributed network virtualization for dcn," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 3, pp. 1–6, 2018. DOI: 10.1145/3243157.3243161. [Online]. Available: http://web.lums.edu.pk/~alizai/pubs/2016-zaafar-ccr.pdf.

[2] *Api-aware networking and security powered by bpf*. [Online]. Available: https://cilium.io/.

[3] G. Bertin, "Xdp in practice: Integrating xdp into our ddos mitigation pipeline," in *Netdev 2.1, The Technical Conference on Linux Networking*, Netconf, 2017. [Online]. Available: https://netdevconf.info/2.1/papers/Gilberto_Bertin_XDP_in_practice.pdf.

[4] 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*, ser. ATEC '04, Boston, MA: USENIX Association, 2004, pp. 2–2. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix04/tech/general/full_papers/cantrill/cantrill.pdf.

[5] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 217–231, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow.

[6] J. Corbet, *Bounded loops in bpf programs*, Dec. 2018. [Online]. Available: https://lwn.net/Articles/773605/.

[7] J. Dangaard Brouer, *Xdp - express data path xdp now with redirect*, May 2018. [Online]. Available: http://people.netfilter.org/hawk/presentations/LLC2018/XDP_LLC2018_redirect.pdf.

[8] N. S. Dasineni, N. Shirokov, and R. Dasineni, *Open-sourcing katran, a scalable network load balancer*, Nov. 2018. [Online]. Available: https://engineering.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[9] L. Deri, S. Sabella, and S. Mainardi, "Combining system visibility and security using ebpf," *CEUR-WS*, [Online]. Available: http://ceur-ws.org/Vol-2315/paper05.pdf.

[10] W. Findlay, "Extended berkeley packet filter for intrusion detection implementations," Honours Thesis Proposal, Carleton University, Dec. 2019.

[11] M. Fleming, *A thorough introduction to ebpf*, Dec. 2017. [Online]. Available: https://lwn.net/Articles/740157/.

[12] M. Fleming, *Using user-space tracepoints with bpf*, May 2018. [Online]. Available: https://lwn.net/Articles/753601/.

[13] P. D. Fox, *Dtrace4linux/linux*, Sep. 2019. [Online]. Available: https://github.com/dtrace4linux/linux.

[14] B. Gregg, *Iovisor/bcc*, Sep. 2015. [Online]. Available: https://github.com/iovisor/bcc/blob/master/tools/biolatency.py.

[15] B. Gregg, *Ubuntumasters conference*, Oct. 2019. [Online]. Available: http://www.brendangregg.com/blog/2019-12-02/bpf-a-new-type-of-software.html.

[16] D. Gunter and B. Tierney, "Netlogger: A toolkit for distributed system performance tuning and debugging," in *IFIP/IEEE Eighth International Symposium on Integrated Network Management, 2003.*, Mar. 2003, pp. 97–100. DOI: 10.1109/INM.2003.1194164.

[17] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee, "Netlogger: A toolkit for distributed system performance analysis," in *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728)*, Aug. 2000, pp. 267–273. DOI: 10.1109/MASCOT.2000.876548.

[18] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '18, Heraklion, Greece: ACM, 2018, pp. 54–66, ISBN: 978-1-4503-6080-7. DOI: 10.1145/3281411.3281443. [Online]. Available: http://doi.acm.org/10.1145/3281411.3281443.

[19] *Iovisor/bcc*, Oct. 2019. [Online]. Available: https://github.com/iovisor/bcc.

[20] *Iovisor/bpftrace*, Nov. 2019. [Online]. Available: https://github.com/iovisor/bpftrace.

[21] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," *USENIX winter*, vol. 93, 1992. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf.

[22] S. Miano, M. Bertrone, F. Risso, M. Vasquez Bernal, Y. Lu, and J. Pi, "Securing linux with a faster and scalable iptables," *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 3, pp. 2–17, Jul. 2019. [Online]. Available: https://ccronline.sigcomm.org/wp-content/uploads/2019/07/acmdl19-304.pdf.

[23] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018. DOI: 10.1109/hpsr.2018.8850758. [Online]. Available: https://mbertrone.github.io/documents/18-eBPF-experience.pdf.

[24] J. Mogul, R. Rashid, and M. Accetta, "The packer filter: An efficient mechanism for user-level network code," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, ser. SOSP '87, Austin, Texas, USA: ACM, 1987, pp. 39–51, ISBN: 0-89791-242-X. DOI: 10.1145/41457.37505. [Online]. Available: http://doi.acm.org/10.1145/41457.37505.

[25] F. Neves, N. Machado, and J. Pereira, "Falcon: A practical log-based analysis tool for distributed systems," *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018. DOI: 10.1109/dsn.2018.00061. [Online]. Available: https://ieeexplore.ieee.org/Xplore/document/8416513.

[26] F. Neves, N. Machado, and J. Pereira, *Fntneves/falcon*, Oct. 2019. [Online]. Available: https://github.com/fntneves/falcon.

[27] *Nit(4p) sunos 4.1.1 reference manual*, Sun Microsystems Inc., Sep. 1990.

[28] *Perf(1) linux user's manual*, Nov. 2019. [Online]. Available: http://man7.org/linux/man-pages/man1/perf.1.html.

[29] *Ptrace(2) linux user's manual*, Oct. 2019.

[30] S. Rostedt, *Documentation/ftrace.txt*, 2008. [Online]. Available: https://lwn.net/Articles/290277/.

[31] R. Rubira Branco, "Ltrace internals," in *Proceedings of the Linux Symposium*, vol. 1, pp. 41–52. [Online]. Available: https://www.linuxsecrets.com/kdocs/mirror/ols2007v1.pdf#page=41.

[32] M. Rybczyńska, *Bounded loops in bpf for the 5.3 kernel*, Jul. 2019. [Online]. Available: https://lwn.net/Articles/794934/.

[33] J. Schulist, D. Borkmann, and A. Starovoitov, *Linux socket filtering aka berkeley packet filter (bpf)*, Mar. 2019. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/filter.txt.

[34] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[35] A. Starovoitov, "Tracing filters with bpf," The Linux Foundation, RFC Patch 0/5, Dec. 2013. [Online]. Available: https://lkml.org/lkml/2013/12/2/1066.

[36] A. Starovoitov, *Bpf at facebook*, Sep. 2019. [Online]. Available: https://kernel-recipes.org/en/2019/talks/bpf-at-facebook/.

[37] A. Starovoitov and D. Borkmann, *Bpf: Introduce bounded loops*, Jun. 2019. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/commit/?id=2589726d12a1b12eaaa93c7f1ea64287e383c7a5.

[38] *Strace(1) linux user's manual*, 5.3, Strace project, Sep. 2019.

[39] K. Suo, J. Rao, L. Cheng, and F. C. Lau, "Time capsule: Tracing packet latency across different layers in virtualized systems," *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems - APSys 16*, 2016. DOI: 10.1145/2967360.2967376. [Online]. Available: https://i.cs.hku.hk/~fcmlau/papers/capsule.pdf.

[40] K. Suo, Y. Zhao, W. Chen, and J. Rao, "Demo/poster abstract: Efficient and flexible packet tracing for virtualized networks using ebpf," *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2018. DOI: 10.1109/infcomw.2018.8406849. [Online]. Available: https://ieeexplore.ieee.org/document/8406849.

[41] *Tcpdump/libpcap public repository*, Sep. 2010. [Online]. Available: https://www.tcpdump.org/.

[42] K. Van Hees, "bpf, trace, dtrace: DTrace BPF program type implementation and sample use," The Linux Foundation, RFC Patch 00/11, May 2019, pp. 1–56. [Online]. Available: https://lwn.net/Articles/788995/.

[43] M. Xhonneux and O. Bonaventure, "Flexible failure detection and fast reroute using ebpf and srv6," *CoRR*, vol. abs/1810.10260, 2018. arXiv: 1810.10260. [Online]. Available: http://arxiv.org/abs/1810.10260.

[44] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "Lprof: A non-intrusive request flow profiler for distributed systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 629–644, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhao.