An In-depth Analysis of Performance Metrics and Comparative Assessment of Network Telemetry Tools in Linux Environments: Insights into Efficiency, Scalability, and Real-time Monitoring Capabilities

***Abstract*— In light of the escalating cyber threats targeting public cloud infrastructure, ensuring robust network security measures across Linux machines has become imperative. Recent statistics reveal a stark reality; a staggering 39% of businesses encountered security breaches within their cloud environments in 2022, marking a substantial 35% surge from the preceding year. These breaches impacted approximately 400 million individuals, underscoring the gravity of the situation. As organizations increasingly transition their operations to the cloud, they must diligently address the accompanying security risks. This paradigm shift necessitates a comprehensive approach to cloud security, with particular emphasis on monitoring and surveillance of cloud infrastructure usage by customers. Effective security observability mandates the deployment of monitoring and alerting systems capable of promptly detecting and mitigating potential threats in real-time.**

The Linux community has emerged as a vanguard in this endeavor, widely embracing Berkeley Packet Filter (BPF) technology. Leveraging the flexibility and extensibility of BPF, a myriad of sophisticated tools have been developed, offering unparalleled capabilities in enhancing security observability and response mechanisms. To delve deeper into the realm of network monitoring tools, this study commences by scrutinizing legacy solutions such as auditd, which inherently facilitate auditing of all facets of Linux machines. Furthermore, the origins and evolution of BPF within the Linux ecosystem are explored, shedding light on its transformative impact.

Subsequently, an array of BPF-based monitoring tools tailored for scrutinizing Linux system processes are examined. In addition to elucidating their functionalities, the performance of select tools and technologies is meticulously assessed. To quantify CPU consumption accurately, a rigorous experimental methodology is employed. Virtual machines with identical specifications are subjected to three cycles of network load simulation, ensuring reliable and unbiased performance evaluations. Through rigorous experimentation, invaluable insights into resource consumption patterns for each tool are gleaned, facilitating informed decision-making in tool selection and deployment strategies.

***Keywords* — Auditd, bpf, bpftrace, ebpf, linux network monitoring, network monitoring, portability in linux distributions, performance comparison of network tools, security in public cloud, tcpdump.**

I. INTRODUCTION

The paradigm shift towards cloud computing continues to accelerate, heralding a new era of technological innovation and enterprise transformation. As organizations increasingly harness the power of cloud technologies to drive agility, scalability, and operational efficiency, the digital landscape becomes both a playground for innovation and a battleground for cyber warfare. The proliferation of cloud-based infrastructures has unleashed a torrent of opportunities, yet it has also exposed a vulnerable underbelly susceptible to malicious exploits and cyber-attacks.

In cyber warfare annals, the year 2022 stands as a stark reminder of the perils that lurk in the digital abyss. Network intrusion, the specter haunting enterprises and organizations across the United States has emerged as a preeminent threat vector, accounting for 45% of all security incidents. Against this backdrop of escalating cyber threats, fortifying the bastions of network security within cloud infrastructures is of paramount importance.

The imperatives of network security resound with heightened urgency, echoing the corridors in cyberspace. As adversaries deploy increasingly sophisticated attack vectors to breach the sanctity of digital fortresses, it has become imperative to

erect formidable defenses. Safeguarding the integrity and confidentiality of data traversing cloud networks requires a multifaceted approach that, integrating cutting-edge technologies, robust policies, and vigilant monitoring mechanisms.

The tripartite framework of security orchestration, encompassing the realms of collection, detection, and response, emerges as lodestar - guiding enterprises through the labyrinthine corridors of cybersecurity. In the crucible of digital warfare, data emerge as lifeblood coursing through the veins of network defenses, providing raw material for threat intelligence and situational awareness.

The pantheon of network security monitoring, a bastion of resilience amidst the tempest of cyber threats, stands as the vanguard of organizational defenses. Powered by a panoply of open-source and commercial tools, the armory of network defenders bristles with the arsenal of threat detection, analysis, and mitigation capabilities. Yet, amidst the cacophony of tools and techniques, discerning the proverbial wheat from the chaff emerges as an arduous endeavor, demanding judicious selection and meticulous integration.

Enter the epoch of the Berkeley Packet Filter (BPF), an epochal paradigm shift heralding a renaissance in network monitoring and analysis. BPF, the veritable cornerstone of Linux evolution, empowers developers with unprecedented freedom to craft bespoke network - monitoring solutions within the sanctum of a sandbox-like environment. The ascendance of BPF-based tools, epitomized by stalwarts such as tcpdump, bcc, and bpftrace, augurs a new dawn in network telemetry, replete with insights and capabilities hitherto unimaginable.

In the crucible of technological innovation, this paper endeavors to unravel the labyrinthine corridors of network security monitoring, traversing the trodden paths of legacy solutions while charting the uncharted territories of BPF-based innovations. Through a prism of inquiry and introspection, we shall dissect the nuances of network monitoring tools, navigating the turbulent seas of implementation challenges and performance bottlenecks.

In conclusion, as the maelstrom of digital transformation continues to sweep across the global landscape, the clarion call for network security vigilance reverberates with undiminished fervor. By embracing the ethos of continuous innovation and adaptive resilience, organizations can fortify their digital citadels against the ravages of cyber threats, ensuring the sanctity of their networks and the integrity of their data.

II. Network monitoring with tcpdump

Reference [5] introduces the tcpdump utility as an influential and adaptable tool intrinsic to the Linux command line environment. Its primary function revolves around the capture and analysis of bidirectional network traffic, thereby furnishing users with indispensable insights into network performance and security paradigms. A salient advantage of tcpdump is its pervasive integration within major Linux distributions ex gratia, expediting deployment and obviating the need for manual installation procedures. This facilitates rapid initiation of network analysis endeavors by users with requisite privileges.

Tcpdump affords users granular control over packet capture operations through the specification of network interfaces and packet count limitations, as exemplified by the "interface" argument and packet count flag "-c 5." The synergy of these parameters enables tailored capture configurations conducive to targeted analysis objectives. Moreover, tcpdump proffers an array of filtering mechanisms encompassing hostname, port number, and protocol type, augmenting the precision and relevance of captured data. The "x" flag facilitates packet content inspection in ASCII and Hexadecimal formats, enhancing data interpretability and facilitating nuanced analysis.

A scrutiny of tcpdump's implementation at scale reveals its facile integration across diverse computing environments facilitated by its inclusion within standard Linux distributions. This multifaceted utility simplifies the capture and storage of network traffic data for subsequent analysis through the generation of .pcap files. These files serve as repositories of captured data amenable to analysis by tcpdump itself or compatible tools such as Wireshark. The latter, distinguished by its intuitive user interface and robust filtering capabilities, emerges as an indispensable adjunct for network administrators and analysts in dissecting the intricacies of network traffic.

Pros:

1. Pre-installed Ubiquity**:** One of tcpdump's most lauded attributes is its ubiquity, as it comes pre-installed with most major Linux distributions. This ensures seamless integration into operational workflows, obviating the need for arduous installation rituals and facilitating

rapid deployment across diverse network environments.

2. Granular Interface Enumeration: Tcpdump empowers users with granular control over network interfaces, facilitating precise packet capture from designated sources. Through the judicious use of interface arguments, users can tailor their analysis to specific network segments, thereby enhancing the fidelity and relevance of captured data.
3. Sophisticated Filtering Capabilities: Tcpdump offers a pantheon of filtering mechanisms, encompassing hostname, port number, and protocol type, among others. This affords users the flexibility to sculpt their analysis according to bespoke criteria, facilitating targeted investigation and elucidation of pertinent network anomalies.
4. Flexible Output Options: Tcpdump facilitates the seamless exportation of captured network traffic to .pcap files, ensuring the preservation and portability of captured data for subsequent analysis. This versatile output format is compatible with a plethora of analysis tools, including the widely acclaimed Wireshark, thereby enhancing interoperability and facilitating collaborative analysis endeavors.

Cons:

1. Absence of HTTP Session Displays: A notable limitation of tcpdump is the absence of dedicated displays for HTTP sessions. Consequently, users are compelled to manually sift through voluminous packets to discern individual sessions, a laborious and time-intensive endeavor that can impede workflow efficiency and detract from the overall user experience.
2. Processing Overhead: Tcpdump's efficacy in capturing network traffic at scale may be marred by processing overhead, particularly in scenarios characterized by high throughput and data volume. This can manifest as latency in packet capture and analysis, potentially impeding real-time responsiveness and compromising the timeliness of threat detection and mitigation efforts.

III. Network monitoring with auditd

The Linux Audit system provides a robust framework for logging events within Linux environments. At its core, auditd facilitates comprehensive auditing capabilities, including monitoring filesystems, processes, and network activities. Notably, auditd comes pre-installed with most major Linux distributions, simplifying deployment and eliminating the need for manual setup.

The audit system comprises two key components: Audit.rules and Audit.conf, located within the /etc/audit/ directory. Audit.rules allows for the configuration of auditing rules, while Audit.conf defines essential parameters such as log file location and buffer size.

Initiating auditd is straightforward using the auditctl command. However, it's important to note that initial network tracing may yield extraneous socket-related calls. To focus on IPv4 and IPv6 network connections, specific audit rules are configured accordingly.

In terms of scalability, installing auditd across various systems is standardized, with subsequent configuration managed through auditctl. This ease of deployment and management is a significant advantage, particularly for organizations with diverse computing environments.

Augmenting auditd are utilities like ausearch and augenrules, which assist in querying audit logs and enforcing persistent audit rules, respectively. Additionally, tools like aulast and aulastlog provide insights into user login activities.

Despite its utility, auditd is not without scalability challenges. While installation and configuration are standardized, managing audit rules across a large number of systems can become cumbersome. Furthermore, performance overhead, particularly during context switching and information transfer between kernel and user space, can be significant, impacting scalability in high-volume environments.

In conclusion, while auditd is a valuable component of network monitoring, organizations should consider its scalability limitations and complement it with appropriate tools to effectively navigate modern security landscapes.

The burgeoning challenge confronting practitioners revolves around the efficient management of the prodigious data deluge engendered by auditd. While ausearch serves as a viable recourse for event scrutiny, its efficacy diminishes when confronted with the Herculean task of traversing data sets at scale, particularly within the milieu of sprawling infrastructures comprising millions of interconnected machines. In response to this exigency, a slew of sophisticated tools, exemplified by AuditBeat, has emerged within the market landscape. These solutions specialize in the aggregation of logs emanating from disparate

nodes scattered across the organizational infrastructure. Leveraging the capabilities conferred by such tools, administrators can craft intuitive dashboards, orchestrate alerting mechanisms, and devise discerning detection algorithms, thereby fortifying their vigilance vis-vis security threats and compliance imperatives.

Notwithstanding the commendable strides made by the audit framework in elucidating the labyrinthine pathways traversed by network packets, its utility is tempered by inherent drawbacks. Foremost among these is the pronounced sluggishness exhibited by auditd in the realm of data acquisition. This lethargy is attributed to the arduous process of context switching and data transfer from kernel space to user space, exacting a significant toll on auditd's performance profile. In contrast, Berkeley Packet Filter (BPF) usurps a vanguard position by predominantly executing computational tasks within the confines of kernel space, endowing it with a swifter operational cadence under certain operational contexts.

Another salient deficiency intrinsic to auditd pertains to its restricted concurrency model, wherein only a single program can be accommodated at any given instance. Consequently, once event records are ingested, they are irrevocably consigned to oblivion, precluding the maintenance of a comprehensive audit trail delineating system events. In navigating these impediments, prudent consideration must be accorded to these limitations whilst delineating the contours of the security monitoring strategy and espousing judicious tool selection endeavors tailored to the idiosyncrasies of the organizational infrastructure.

IV. Network monitoring with BCC

Reference [2]: The Berkeley Packet Filter Compiler (BCC) represents a potent toolkit engineered to facilitate the development of efficient kernel tracing and manipulation programs, harnessing the capabilities conferred by the extended Berkeley Packet Filter (eBPF) paradigm. Noteworthy for its versatility, BCC streamlines the process of crafting BPF programs by affording developers the flexibility to express kernel instructions in C and construct a user-friendly front-end utilizing Python scripting. This discourse will concentrate on the utilization of BCC for network tracing endeavors.

A BCC program, scripted in Python, is affixed to kernel entry or exit points, endowing it with the capability to retrieve socket information and unveil intricate details pertaining to network events. For comprehensive insights into authoring BCC programs or perusal of sample implementations, refer to the iovisor/bcc repository available at: BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more (github.com).

The myriad of tools encompassed within the BCC framework can be perused within the bin directory. To commence tracing TCP connections, execute the following command:

A. Examining the Challenges and Trade-offs with BCC: Evaluating Advantages and Drawbacks

The utilization of BCC hinges on a unique approach, wherein the compilation of Berkeley Packet Filter (BPF) code transpires dynamically. However, this methodology engenders a spectrum of considerations warranting thorough scrutiny before embarking on its adoption:

1. Scalability Conundrums: While BCC stands as an invaluable asset for crafting intricate applications, its scalability is contingent upon the installation of kernel header packages on host machines. This requisite presents formidable hurdles when endeavoring to deploy BCC across expansive networks comprising millions of interconnected machines. Additionally, the inherent intricacies of building a robust network tracing tool at scale pose formidable challenges, necessitating meticulous planning and resource allocation.
2. Impediments with Clang Front-End Integration: BCC programs interface with the Clang front-end to effectuate modifications to user-crafted BPF programs. This coupling introduces a layer of complexity, rendering the identification of underlying issues arduous. Consequently, troubleshooting endeavors are plagued by protracted timelines and heightened resource expenditure, detracting from the developer experience and impeding operational agility.
3. Performance Quandaries: Notwithstanding its utility, the performance of BCC programs lags behind alternative BPF-based technologies, precipitating suboptimal outcomes, particularly in projects where performance optimization assumes paramount significance. This performance delta necessitates judicious consideration when appraising the suitability of BCC for specific operational contexts, lest operational efficacy be compromised.
4. Compromised Compilation Workflow: The compilation process inherent to BCC necessitates redundancy when deployed across a fleet of machines, engendering superfluous

resource consumption and escalating operational costs. This redundancy underscores the imperative of streamlining the compilation workflow to mitigate inefficiencies and enhance deployment agility in large-scale operational environments.

Considering the multifaceted challenges enumerated above, the efficacy of BCC as a panacea for large-scale and scalable BPF-based programs is cast into doubt. Prudent decision-making mandates a meticulous weighing of the pros and cons, ensuring alignment with the exigencies of specific operational paradigms and strategic imperatives.

V. Network monitoring with bpftrace: A comprehensive guide to effective network tracing.

Reference [1]: Bpftrace stands as a sophisticated high-level tracing language designed explicitly for eBPF (extended Berkeley Packet Filter) and boasts compatibility with most Linux distributions. Crafted as an invaluable tool for system administrators and developers alike, its robust capabilities aid in system monitoring and troubleshooting endeavors. Leveraging LLVM as its backend compiler, Bpftrace seamlessly transcribes scripts into BPF bytecode, thereby ensuring optimal efficiency and performance, particularly in the realm of network monitoring. Drawing inspiration from the likes of awk and C, and predecessors such as DTrace and SystemTap, the bpftrace language amalgamates powerful functionalities with a familiar syntax.

Accessible through Linux package managers or compilation from open-source repositories, Bpftrace exhibits a tendency towards compatibility issues with certain Linux kernel versions when obtained directly from package managers, as per personal experiences. To mitigate such concerns, it is advisable to procure the latest compiled version from the open-source repository, ensuring seamless functionality and compatibility. The repository link can be accessed here: iovisor/bpftrace: High-level tracing language for Linux eBPF (github.com).

Focusing on network tracing, we delve into select scripts within bpftrace tailored for TCP passive and active connections. These scripts furnish invaluable insights into network behavior, facilitating the identification of bottlenecks, diagnosis of issues, and optimization of performance. Links to pertinent scripts can be found below:

- Script for TCP passive connections: bpftrace/tools/tcpaccept.bt at master · iovisor/bpftrace (github.com)
- Script for TCP active connections: bpftrace/tools/tcpconnect.bt at master · iovisor/bpftrace (github.com)

To initiate the scripts, the binary must be downloaded from the following link: Release v0.19.1 · iovisor/bpftrace (github.com).

This command initiates monitoring of active TCP connections and outputs pertinent details including process ID, time, command executing the TCP connection, source and destination IP addresses, and ports. Preliminary tests indicate minimal CPU usage and memory consumption, even on medium-sized virtual machines. Detailed results will be expounded later in this discourse.

Bpftrace garners distinction through its diminutive installation footprint, boasting a mere 1 KB for the program itself and a lightweight binary. With an extensive repository of pre-made tracing tools, facile one-liner scripting capabilities, and broad support across various distributions, it emerges as a stalwart tool for scaling network observability. However, it's noteworthy that bpftrace remains in the BETA release phase for ARM 64 processors, signifying an area for potential improvement. Detailed implementation insights will be elucidated subsequently, furnishing a comprehensive guide on maximizing the efficacy of bpftrace for network monitoring endeavors.

A. BPFTrace Implementation at Scale: Overcoming Challenges for Linux Distributions:

Implementing bpftrace based logging was very simple since it supports writing one liner programs. This simplicity greatly benefits developers when creating and maintaining their code.

There are a couple of things to note when starting to build scalable solutions for Linux machines. Since Linux supports various distributions and versions, finding the binary that’s statically built becomes more important than ever. We will investigate the concepts of statically linked binaries shortly.

Until now, we have talked about the official binary release on bpftrace repository artifact release page. Here is the link: Releases · iovisor/bpftrace (github.com)

As of this writing, the binary provided on the above page is a dynamically linked binary, which means the dependencies will be pulled at runtime on the respective distributions. This becomes a challenge when building a widespread solution for various distributions. To support various distributions with this solution, we must build this binary using each distro and use that executable with the respective distros and kernel versions. Obviously, this solution is not going to scale with new Linux kernel versions and distributions growing rapidly over time. Hence, the need for a solution to create a statically linked binary became apparent. This solution was designed separately to solve the portability issues with Linux.

Bpftrace also supports this solution and is explained in more detail over here : bpftrace/INSTALL.md at master · iovisor/bpftrace (github.com)

However, I would like to delve deeper into the Appimage tooling, as it is open source and is required for implementing bpftrace on a larger scale. The Appimage team provides various tools and utilities to simplify the shipping and packaging process. One of the most used tools in Appimage is linuxdeploy. Linuxdeploy is an AppDir maintenance tool.

In the case of any modern build system such as CMake, you can use the regular make install commands to create a directory-like structure which will then be used by linuxdeploy for packaging. CMake also comes with an inbuilt parameter to specify where the files should be installed instead of the root directory, called DESTDIR. This feature allows developers to have more control over the installation process and maintain a cleaner system structure.

For our use case, we logged tcp connect events and it required us to attach our bpftrace script to tcpconnect kernel probe. It also required us to listen to the socket object to fetch the connection details. Tcpconnect script is available in the bpftrace open-source project and most of it can be used as-is. Once the script is ready, we can place it in some folder in our VM.

This will start recording all the events and printing them on Linux terminal. As I already mentioned the downside of using Linux terminal as consumer, we can also write these events into a file, but that will increase cpu consumption because of I/O operations.

To trace the UDP connections, there is no example script available in bpftrace. But we can write our own script and listen to the *udp_sendmsg* and *udp_recvmsg* probe to trace the outgoing and incoming UDP connections respectively.

Deploying bpftrace-based logging solutions presents a straightforward process, owing to its support for writing concise one-liner programs. This inherent simplicity significantly streamlines code creation and maintenance efforts, enhancing developer productivity and codebase manageability.

However, as we embark on building scalable solutions for Linux environments, several considerations come into play. Given the diverse landscape of Linux distributions and versions, the importance of statically linked binaries cannot be overstated. The quest for statically built binaries, which encapsulate all dependencies within the executable itself, becomes imperative to ensure seamless deployment across heterogeneous environments.

Presently, the official binary release available on the bpftrace repository artifact release page is dynamically linked, necessitating runtime dependency resolution on respective distributions. This dynamic linkage poses a formidable challenge when orchestrating widespread deployments across diverse Linux ecosystems. To address this challenge effectively, a solution must be devised to create statically linked binaries, thereby obviating the need for dependency resolution at runtime.

Bpftrace, acknowledging the significance of this challenge, extends support for creating statically linked binaries, as elaborated in detail in the documentation available at: bpftrace/INSTALL.md. However, I advocate for a deeper exploration into the utilization of the Appimage tooling, an open-source initiative crucial for implementing bpftrace at scale. The Appimage ecosystem furnishes an array of tools and utilities geared towards simplifying the packaging and shipping processes. Notably, linuxdeploy emerges as a pivotal tool within the Appimage toolkit, facilitating the maintenance of AppDir structures.

Incorporating modern build systems such as CMake enables developers to leverage conventional `install` commands to generate directory-like structures, which are subsequently

utilized by linuxdeploy for packaging. Furthermore, CMake offers the flexibility of specifying the installation destination via the *DESTDIR* parameter, empowering developers to exert greater control over the installation process and maintain an organized system structure.

In our specific use case, wherein we logged TCP connect events, integration required attaching our bpftrace script to the *tcpconnect* kernel probe and monitoring socket objects to fetch connection details. Leveraging the readily available *tcpconnect* script from the bpftrace open-source project expedited this process. Upon script preparation, deployment entailed placing it in a designated folder within the virtual machine.

This command initiates event recording and displays outputs on the Linux terminal. Recognizing the limitations of terminal-based consumption, an alternative approach involves directing event outputs to a file, albeit at the expense of increased CPU consumption due to heightened I/O operations.

While bpftrace offers extensive capabilities for tracing TCP connections, a similar out-of-the-box solution for UDP connections is absent. However, developers can craft custom scripts leveraging **udp_sendmsg** and **udp_recvmsg** probes to monitor outbound and inbound UDP connections, respectively.

VI. Performance comparison between audit and bpf based programs.

In our relentless pursuit of network data insights, we embark on a meticulous examination of performance between two stalwart tools: BPFTrace and Auditd. Our endeavor revolves around network-based logging, with a keen focus on scrutinizing CPU and memory consumption to unravel the depths of efficacy and efficiency.

Leveraging standard TCP kernel probes as our benchmark, we embark on a journey to record events, delving into the comparative performance of these tools. Harnessing the intrinsic file output capabilities of both BPFTrace and Auditd, our analysis maintains a laser-like focus, ensuring a discerning evaluation void of extraneous factors.

For our experimentation, we deploy a virtual machine endowed with robust specifications—an Intel Xeon processor boasting 4 cores and 16 GB of RAM. This judicious allocation of resources ensures an environment ripe for exhaustive performance analysis. Noteworthy is the adherence to standard operating system configurations, enhancing the universality of our findings.

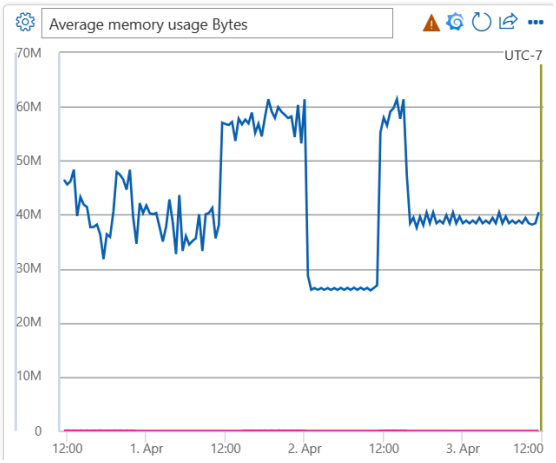

Fig 1. Average memory usage with bpftrace

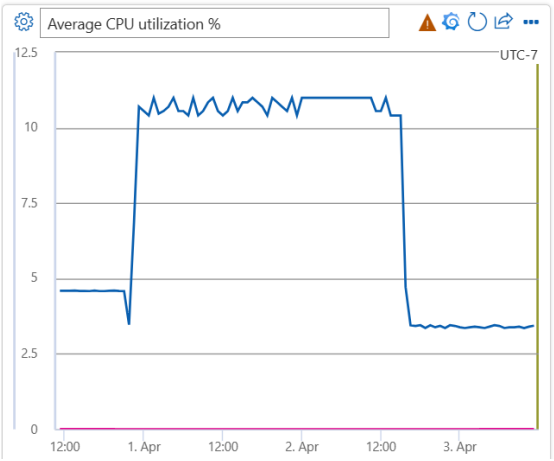

Fig 2. Average CPU utilization with bpftrace

Fig 1 and Fig 2 shows the graphs showing the average cpu and memory utilization when running bpftrace tracing 1000 connections/sec, which is huge for Linux machines. With peak load, it is seen that the CPU utilization went upto 11%, which is expected. Average memory usage was 60M, but this is a runtime memory usage. This doesn't include the memory used by bpftrace process to perform dynamic compilation using LLVM, which is approx. 150M. This makes bpftrace difficult to use for very small workloads. However, as part of last year's conference, it looks like the bpftrace is coming up with options to mitigate the dynamic compilation issue.

On the other hand, with the same load auditd was not able to perform such. Average CPU utilization for auditd with 1000 connections/sec was above

24%. The memory utilization was variable ranging from 150 to 300 MB.

Our investigation unfolds against a backdrop of concrete metrics:

- CPU and Memory Utilization with BPFTrace: Graphical representations unveil the intricate interplay of CPU and memory during BPFTrace's execution. With an imposing load of 1000 connections/sec, the CPU's utilization peaks at a modest 11%, a testament to its efficiency. However, the snapshot of memory usage—averaging 60M—barely scratches the surface, as it excludes the overhead incurred by dynamic compilation processes, pegged at approximately 150M. This nuance underscores the necessity of nuanced optimization efforts, especially in light of the tool's unsuitability for minute workloads. Notably, sample sizes of 1000 connections/sec were rigorously selected to mirror real-world scenarios.
- Comparison with Auditd: In stark contrast, Auditd grapples with performance under identical loads. Bearing witness to CPU utilization exceeding 24% with 1000 connections/sec, coupled with a volatile memory footprint ranging from 150 to 300 MB, Auditd struggles to attain resource efficiency.

Amidst BPFTrace's commendable CPU performance, a labyrinth of limitations beckons:

- Script Complexity and Tooling Support: While BPFTrace excels in one-liner scripts, the specter of complexity looms large over intricate tasks. The absence of robust tooling support presents a bottleneck, hindering seamless integration with high-performance systems. Although file-based output is a viable recourse, the specter of heightened CPU consumption due to amplified I/O operations demands immediate attention.
- Terminal Processing Limitations: The confines of the Linux terminal's print buffer impede concurrent print statement processing, fostering an environment ripe for data loss amidst surging event volumes.
- Memory Usage Challenges: In the crucible of production environments, runtime memory usage alongside LLVM compilation casts a long shadow over workloads with meager RAM (< 2 GB), warranting optimization endeavors of the highest order.
- Executable Portability: BPFTrace's dalliance with portability across Linux distributions poses an insidious threat to compatibility. Though recent strides towards Appimage adoption offer a glimmer of hope, formidable challenges persist.

Despite Auditd's valiant efforts, particularly in terms of elevated CPU consumption, BPFTrace emerges as the paragon of choice for constructing high-performance observability tools. However, a nuanced understanding of BPFTrace's limitations serves as the lodestar, beckoning concerted optimization efforts to unleash its full potential in crafting efficient and dependable observability solutions.

Conclusion

Following an exhaustive exploration of various tools, bpftrace emerges as the unequivocal champion in terms of performance—a pivotal consideration in tool evaluation for network observability, particularly in the realm of the latest Linux kernel versions.

To establish an observability framework at scale, both bpftrace and auditd necessitate additional functionalities to be developed around them, optimizing their performance and ensuring event preservation without loss. Auditd, as a conventional logging system, lacks kernel namespacing, resulting in a deficiency of crucial details regarding the container that triggered network or system calls, rendering it less suitable for containerized environments. Furthermore, contemporary applications often encrypt network traffic, rendering packet capture prohibitively expensive and diminishing the efficacy of auditd in cloud-native settings.

While bpftrace and auditd were thoroughly evaluated, numerous other tools remain unexplored within this article's purview. When selecting the appropriate tool for a specific use case, performance remains paramount. Additionally, the investigation primarily focused on monitoring virtual machines, while commercial products cater to containers and Kubernetes workload monitoring—a topic left unaddressed.

Integration ease with existing infrastructure, the learning curve associated with tool adoption, security, scalability, and customizability are crucial factors in tool selection for network observability. As the landscape of network inspection tools continues to evolve, staying abreast of the latest developments, updates, and best practices ensures informed decision-making tailored to an organization's unique requirements and infrastructure.

References

[1] iovisor, "bpftrace," GitHub. [Online]. Available: https://github.com/iovisor/bpftrace. [Accessed: 01-May-2024].

[2] iovisor, "bcc," GitHub. [Online]. Available: https://github.com/iovisor/bcc. [Accessed: 01-May-2024].

[3] D. Calavera and L. Fontana, "Linux Observability with BPF," 2020.

[4] C. Humber, "eBPF — a new Swiss army knife in the system," Medium. [Online]. Available: https://medium.com/@chivierhumber/ebpf-a-new-swiss-army-knife-in-the-system-8964ad280eab. [Accessed: 01-May-2024].

[5] "How to Capture Network Traffic in Linux with tcpdump," MakeUseOf. [Online]. Available: https://www.makeuseof.com/tag/capture-network-traffic-linux-tcpdump/. [Accessed: 01-May-2024].

[6] "39% of businesses faced a cloud environment data breach last year," Security Magazine. [Online]. Available: https://www.securitymagazine.com/articles/95044-of-businesses-faced-a-cloud-environment-data-breach-last-year. [Accessed: 01-May-2024].