Security Applications of Extended BPF Under the Linux Kernel

COMP5900T OS Security Research Paper

WILLIAM FINDLAY, School of Computer Science, Carleton University

Abstract

Extended BPF (eBPF) is an emergent technology for system observability under the Linux kernel. While eBPF is commonly used for performance tracing [19], it also has tremendous potential with respect to operating system security, thanks to its ability to observe arbitrary system state under strict safety guarantees [36, 37]. This paper will discuss eBPF's role in the OS security landscape; in particular, a pattern emerges that establishes current research in this area as being predominantly focused on networking stack security and network-based data collection. To rectify this gap in host-based eBPF security research, I will present two novel applications of eBPF in OS security: `ebpH` [14], a host-based eBPF anomaly detection system, and `bpfbox`, a new sandboxing technique that leverages BPF programs to enforce seccomp rules externally and transparently to the target application.

Additional Key Words and Phrases: eBPF, Operating System Security, Linux, Sandboxing, Anomaly Detection

1 INTRODUCTION

Since its 2014 addition to the Linux kernel [37], extended BPF (eBPF) has been providing Linux users with powerful tools to observe many aspects of system behavior, both in userspace and in kernelspace. While eBPF often sees use as a means of developing performance monitoring tools, it also presents an opportunity for use within the domain of computer security. In particular, eBPF can provide a safe and efficient means of data collection for intrusion detection systems [14, 19]. New BPF program types like XDP [20] are capable of inspecting and filtering packets via direct memory access *before* they reach the kernel networking stack – a property which is ideal for responsive network intrusion detection systems [4, 11, 28] and DDoS (distributed denial of service) mitigation [2]. The enhanced observability provided by eBPF also offers system administrators the option to write tools to monitor various security-sensitive operations on the system, such as `execve(2)` calls, privilege escalation, permission errors on system calls, or even the usage of POSIX capabilities [19].

This paper will examine the role of both eBPF and classic BPF as they pertain to operating system security. In particular, I will present several eBPF solutions that focus predominantly on various layers of the kernel's networking stack, as well as classic BPF technologies like `seccomp-bpf` [1, 34] and `tcpdump` [42]. A common theme throughout eBPF-related security solutions is that they tend to focus on the networking stack specifically, despite eBPF's wide range of capabilities with respect to system introspection. To fill this perceived gap in the research, I will briefly discuss two novel approaches to OS security using eBPF. `ebpH` [14] is an anomaly detection system that I built for my thesis, which instruments system call patterns using BPF programs. `bpfbox` is a prototype sandboxing application built in eBPF that allows for `seccomp`-like policy to be enforced externally and transparently to the target application. In the context of `ebpH`, `bpfbox`, and existing work in networking stack instrumentation, I will then discuss the potential for future eBPF security solutions that take a more holistic approach to system monitoring.

Section 2 provides a high-level background on classic BPF and extended BPF, highlighting the key similarities and differences between the two, as well as presenting some security-related use cases for each. Section 3 discusses how eBPF compares to other system introspection techniques, with an emphasis on observability, scalability, performance, and production safety; this will help to establish the unique advantages that eBPF has over conventional data collection techniques. Section 4 discusses classic BPF and extended BPF approaches to securing the networking stack, highlighting the current focus of eBPF-related security research on network monitoring and policy enforcement therein. Section 5 discusses other aspects of OS security as they relate to both classic and extended BPF. In particular, it will present classic BPF approaches to sandboxing through `seccomp-bpf`, along with two new eBPF systems, `ebpH` and `bpfbox`, for host-based anomaly detection and externally enforceable sandboxing respectively. Section 6 discusses the overall research trends described in this paper and presents potential topics for future work, with emphasis on new directions for `ebpH` and `bpfbox`. Finally, the paper concludes with Section 7.

Author's address: William Findlay School of Computer Science, Carleton University, william@williamfindlay.com.

2 BACKGROUND

While extended BPF is still a relatively new technology in the Linux kernel [36, 37], its predecessor, the classic Berkeley Packet Filter (cBPF) has been around for over 20 years [27], even prior to its introduction to Linux. Here, I provide some background on classic BPF, examining its original design and early use cases. I will then highlight the various improvements and features that have been introduced in extended BPF, particularly with respect to full system observability. Finally I will briefly discuss the naming conventions that will be used throughout the rest of this paper to refer to various aspects of BPF.

2.1 Classic BPF

McCanne and Jacobson [27] first introduced the Berkeley Packet Filter in 1992 as a virtual machine for the BSD Unix kernel, capable of running custom user-specified bytecode instructions in kernelspace; this technology permitted the construction of extremely efficient *tap-and-filter* mechanisms for network traffic.

Classic BPF's primary contributions at the time were its unique register-based virtual machine language and its heavy use of buffering to reduce packet capture overhead [27]. These two key design advantages allowed it to significantly outperform its contemporary competitors. Extended BPF (c.f. Section 2.2) would later take these same design elements and extend them to provide enhanced capabilities. Refer to Figure 1 for an overview of classic BPF's architecture.

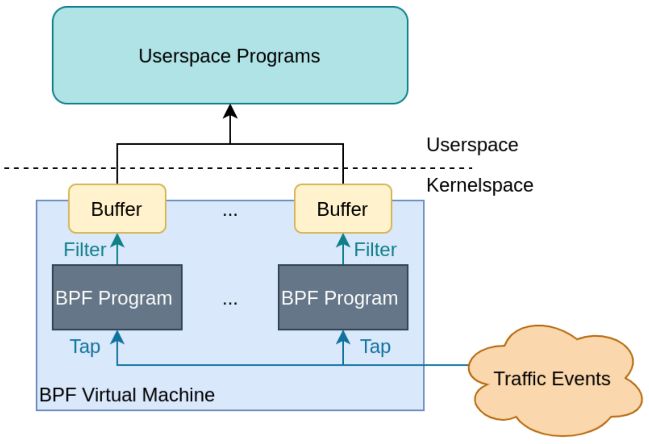

Fig. 1. An overview of classic BPF architecture. Adapted from [27].

Classic BPF was eventually merged into the Linux kernel in version 2.5 [19] and saw some further use in security applications such as `seccomp-bpf` [1, 34], a mechanism for defining rules for system call interposition in userspace. `tcpdump` [42], a popular packet tracing tool, also leverages classic BPF for its functionality. I will discuss these classic BPF tools and use cases further in Section 4.1 and Section 5.1.

2.2 Extended BPF

In 2013, Starovoitov [19, 36] had the idea to take the original BPF paradigm specified by McCanne and Jacobson [27] and *extend* it beyond its intended use case (i.e. packet filtering). His vision was to construct a general-purpose virtual machine capable of running event-based user-specified bytecode in kernelspace. In particular, eBPF introduced 10 64-bit registers[1], a 512-byte stack size, support for calling whitelisted kernel helpers with negligible overhead, various map data structures, JIT compilation to native code, and several new ways of instrumenting both kernelspace and userspace [19]. Quintessentially, eBPF also comes with guaranteed safety, due to the in-kernel verifier (c.f. Section 2.2.1) that analyzes all submitted code before it may be run in the kernel. Figure 2 depicts eBPF's architecture in detail.

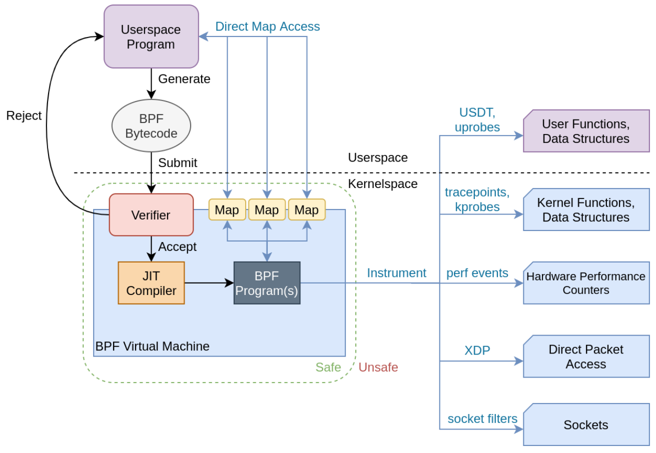

Fig. 2. An overview of extended BPF architecture. Taken from [14] with permission. Note the key differences between the architecture presented here and that of classic BPF from Figure 1. In particular, eBPF allows far more functionality than classic BPF (n.b. the eBPF program types depicted here do not comprise an exhaustive list, but are representative of the most commonly used program types).

eBPF's primary advantages over traditional system introspection techniques are three-fold:

1) Fine-grained control over context switch frequency for map access from userspace, which allows for significant overhead improvements when compared to alternatives like `ptrace(2)` [31];
2) Seamless integration between userspace and kernelspace instrumentation with bidirectional information flow between BPF programs, userspace applications, and other BPF programs;

[1]This includes an extra read-only frame pointer for a total of 11 [19].

3) Production-safe kernel introspection due to the verification step before loading BPF programs into kernelspace.

These properties position eBPF uniquely within the system introspection landscape, particularly as a tool for system performance monitoring, debugging, observability, and security, especially in the context of performance and safety-sensitive production environments.

2.2.1 The eBPF Verifier: Trusting the Untrusted. To allow untrusted[2] user code to be run safely in kernelspace, eBPF employs a verifier that checks submitted bytecode for safety before it is allowed to run. As of Linux 5.6 [25], the verifier is comprised of just over 10,000 lines of C code and is invoked on every attempt to submit a BPF program to kernelspace via the `bpf(2)` system call [7].

Ascertaining program safety involves several restrictions on both the BPF instruction set itself and on various aspects of BPF program state, including:

1) No support for unbounded loops (previously the verifier prohibited all loops, but a 5.3 kernel patch [38] added support for bounded loops on verifiable induction variables);
2) A hard limit of 512 bytes on stack space per BPF program;
3) No unbounded memory access attempts (i.e. the program is required to perform a bounds check before indexing an array);
4) Restricted access to kernel helper functions;
5) A hard limit of 1 million instructions per BPF program (previously this limit was as low as 4096, but was also raised in Linux 5.1).

While these restrictions may seem extreme, they are necessary in order to allow the verifier to accept valid programs and reject invalid programs with certainty. In practice, these restrictions are not difficult to work around and frequent patches to the verifier's functionality continue to offer improvements in this regard.

2.3 Naming Conventions in this Paper

Throughout this paper, I will be referring to the original BPF [27] as *classic* BPF (cBPF), and extended BPF as either eBPF or simply BPF. This is done for philosophical reasons and in order to avoid confusion between the two technologies, particularly as the classic BPF implementation in Linux has been completely replaced by eBPF [19], and the former is a subset of the latter.

[2]As of Linux 5.6, all BPF programs still require the `CAP_SYS_ADMIN` privilege in order to be submitted to the kernel [7], though this may change in the future. "Untrusted" in this context simply means code that comes directly from userspace via the `bpf(2)` system call.

3 COMPARING EBPF TO OTHER SYSTEM INTROSPECTION TECHNIQUES

In many ways, security is about *what we can see*; more formally, the ability to detect and respond to attacks on a system is limited by the ability to measure the state of said system. Without the ability to detect changes in system state, attacks may either go unnoticed entirely or may be detected too late to prevent any lasting or meaningful damage. System introspection provides a means of collecting data about system state which can then be used to detect violations in policy, enforce policy, and even build or generate new policy. This section will cover many of the popular system introspection techniques employed on Linux, highlighting their respective strengths and weaknesses, and ultimately showing that eBPF fills important gaps in the existing system introspection landscape that make it ideal for building robust, secure systems that can be deployed in a production environment. Table 1 provides a summary comparison of the various techniques described in this section.

The `ptrace(2)` system call [31] forms the basis for the implementation of several debugging and observability tools available on Linux, including `strace` [39, 40] and `gdb` [17]. While `ptrace(2)` provides a high degree of observability in userspace and even across the userspace-kernelspace boundary, it suffers from poor production safety due to bugs, limited scope[3], and *very* poor performance overhead, typically on the order of 10–100× [22], and up to 442× in the worst case [18]. This poor performance can be attributed to an extraordinarily high number of context switches required between userspace and kernelspace in the traced process as well as the continual pausing of process execution on both system call entry *and* return. Ultimately, these characteristics render `ptrace(2)` an unacceptable choice for use cases beyond simple debugging, and certainly untenable for the production security monitoring that we desire.

`ftrace` [33] and `perf events` [43] are Linux subsystems for instrumenting tracepoints and hardware performance counters respectively. These implementations are provided in the kernel source and expose APIs to userspace — `ftrace` in the form of a virtual filesystem under `sysfs`, and `perf_events` via the `perf_event_open(2)` [43] system call. Both of these subsystems suffer from relatively poor interfaces and are eclipsed entirely by the functionality of extended BPF (which uses both of these subsystems for its tracepoint and perf counter functionality [19]).

Library call interposition is a technique that allows tracer programs to hook into calls to shared libraries (usually by

[3]`ptrace(2)` may not be used to attach to processes outside of the tracer's hierarchy without modifying kernel security parameters. Additionally, it is not feasible to use `ptrace(2)` on every process on the system.

Table 1. Comparing the popular Linux system introspection techniques. ● represents a satisfied requirement, ◗ represents a partially satisfied requirement, and ❍ represents an unsatisfied requirement. Note that eBPF covers every requirement, including production safety.

Technique	System Wide	Production Safe	Efficient	Userspace Functions	Library Calls	Kernelspace Functions	System Calls	Sockets	Packets	Registers	Hardware Perf. Ctrs.
`ptrace(2)`	❍	❍	❍	●	◗	❍	●	❍	❍	●	❍
Library Call Interposition	●	●	●	◗	●	❍	❍	❍	❍	❍	❍
ftrace	●	●	●	❍	❍	●	●	●	❍	❍	❍
perf events	●	●	●	❍	❍	❍	❍	❍	❍	❍	●
Kernel Module	●	❍	●	◗	◗	●	●	●	●	●	●
Module + Userspace Library	●	❍	●	◗	●	●	●	●	●	●	●
Netfilter	●	●	◗	❍	❍	❍	❍	❍	●	❍	❍
Classic BPF	●	●	●	❍	❍	❍	❍	❍	●	❍	❍
eBPF	●	●	●	●	●	●	●	●	●	●	●

Scope/Safety/Efficiency **Observability**

trampolining, e.g. from a custom function into the actual implementation). Kuperman et al. [24] first proposed using this technique to generate audit data in a 1999 paper, although a more well-known use case is in debugging programs such as `ltrace` [9]. In general, library call interposition is safe to do and can be quite performant — especially when compared to solutions such as `ptrace(2)` — although it loses out on the ability to observe other aspects of system behavior, especially in kernelspace.

In principle, a solution that comprises all potential data sources in the system *must* (at least in part) be implemented in kernelspace. Traditionally, modifications to the Linux kernel must occur either in the form of loadable kernel modules or direct patches to the kernel code itself. Loadable kernel modules allow for runtime modifications to the kernel's address space, which means the ability to load arbitrary code, and look up or modify essentially any address. This naturally lends itself to system tracing, as it essentially provides the means to hook into any event on the system. Strictly speaking, it would also be possible to hook into arbitrary userspace events using a module alone, although this becomes messy and complicated quite quickly, and so the favored approach tends to be combining a kernel module with some sort of userspace library and library call interposition, as described in the previous paragraph. Many popular tracing mechanisms such as Sysdig [41], SystemTap [32], and LTTng [26] take this hybrid approach. While kernel modules represent the most powerful technique discussed thus far, they are also the most dangerous. Bugs in kernel code can be catastrophic to a system. In the best case, the result is a kernel panic, tantamount to denial of service; in the worst case, buggy kernel code can result in data loss, undefined behavior, or even the introduction of new security vulnerabilities.

Netfilter [30] is an interface in the Linux kernel that exposes various hooks within the kernel's networking stack for the instrumentation of packets. Critically, Netfilter provides the ability to track connection state in addition to the packets themselves. This interface is employed by a variety of userspace software for the instrumentation of packets, tracking of connection state, and network address translation (NAT). Perhaps the most notable example of Netfilter, particularly from a security perspective, is the `iptables` firewall, which uses Netfilter to keep track of connection state and packet metadata to enforce policy on ingress and egress. Classic BPF [27] provides similar packet filtering functionality to Netfilter, albeit with three critical differences: classic BPF is incapable of tracking connection state, but classic BPF is both more expressive and more performant than Netfilter for basic packet filtering requirements. Section 2.1 provides more details on the design and implementation of classic BPF.

Thus far, the system introspection technologies that have been presented have all suffered from at least one of three flaws: a lack of production safety, limitations in scope or observability, and performance bottlenecks. While kernel modules come close to offering all of the desired properties of system introspection, they fall short in terms of production safety, a critical property for the adoptability of security tools built using these technologies. As shown in Table 1, extended BPF offers the capabilities of kernel-based implementations with the addition of both production safety, due to its verifier, and richer userspace tracing capabilities, due to built-in mechanisms such as uprobes [19]. In summary, eBPF can trace both userspace and kernelspace safely and efficiently, which in turn can benefit security applications by providing a rich data source that can be used effectively in production.

4 SECURING THE NETWORKING STACK WITH EBPF

Since classic BPF was primarily a networking technology used for packet introspection [27, 42], it is perhaps unsurprising that its extended version has seen continued use in this regard. This section will discuss both the classic BPF implementations for traditional networking security as well as current research with respect to achieving networking stack security with eBPF. Table 2 provides a summary of the systems discussed in this section, including features and implementation.

Table 2. A summary of cBPF and eBPF/XDP solutions targeting Linux networking stack security. "Observability" refers to a system that enhances visibility of the network with respect to the users; "Detection" refers to attack detection; "Response" refers to attack response. "Automatic" refers to automatic rule generation (i.e. the user does not need to manually specify rules). ● represents a satisfied requirement and ○ represents an unsatisfied requirement.

System		Implementation	Observability	Detection	Response	Automatic
`tcpdump`	[42]	Classic BPF	●	○	○	○
`xt_bpf`	[8]	Classic BPF	○	●	●	○
`L4Drop`	[13]	eBPF/XDP	○	●	●	●
`bpf-iptables`	[3, 4]	eBPF/XDP	○	●	●	○
`bpfilter`	[5]	eBPF	○	●	●	○
`ntopng`	[10, 11]	eBPF	●	●	○	○
Nam and Kim	[29]	eBPF	●	●	○	○

4.1 Revisiting Classic BPF

Traditionally, classic BPF [27] has provided an efficient method for capturing network traffic in the kernel networking stack and passing this data back to userspace. Perhaps the most auspicious BPF packet capture tool is `tcpdump` [42], and its associated library, `libcap`. `tcpdump` offers an efficient userspace tool for TCP/IP and UDP packet capture and analysis, including both inbound and outbound traffic. To do this, it uses user-specified arguments to automatically generate appropriate classic BPF programs, and instrument the appropriate layers of the kernel networking stack. `tcpdump` is generally used as a network visibility tool, providing debugging functionality and basic traffic analysis capabilities to network administrators. `libcap` provides the same basic functionality as `tcpdump`, but instead exposes its API in the form of a C/C++ library [42].

`xt_bpf` [8] is a Linux kernel module that augments the Netfilter subsystem to include the execution of classic BPF programs. The result is the ability to use BPF programs to express rules for software that uses the Netfilter interface, such as the `iptables` firewall. By allowing users to express Netfilter rules as BPF programs, `xt_bpf` enables the creation of more expressive and highly performant packet filter rules. While this certainly does not reach the expressiveness of eBPF [3, 4], it is a step in the right direction for augmenting `iptables` with BPF functionality.

While classic BPF solutions like `tcpdump`, `libcap`, and `xt_bpf`-enabled `iptables` are fine for simple network traffic analysis, extended BPF offers significantly greater functionality. For instance, `tcpdump` is incapable of inspecting the contents of packets and is instead limited to viewing the metadata associated with packet headers [16, 42] and only at specific points within the kernel networking stack. By the time a packet has been captured and its info returned to userspace, it may be too late to perform any meaningful action. Additionally, while the rules that are expressible with classic BPF are certainly an improvement over more traditional solutions such as Netfilter, the expressiveness of classic BPF is still a far cry from eBPF, especially considering classic BPF's even more restrictive limitations on instructions, registers, and state management.

Section 4.2 will examine eBPF and XDP (*eXpress Data Path*) solutions for networking security. These solutions are often more sophisticated and more capable than the classic BPF solutions that preceded them.

4.2 Achieving Networking Stack Security with eBPF and XDP

This section will present several eBPF- and XDP-based solutions for augmenting Linux networking stack security. First, I will discuss XDP in more detail than in Section 2 and present a few systems that make use of it. Following that will be more traditional eBPF programs for enhancing both network visibility and security.

XDP [20], or *the eXpress Data Path*, is a new eBPF program type that was created by Høiland-Jørgensen et al. and merged into Linux with version 4.8. Figure 3 provides a simplified overview of XDP's architecture. XDP is unique among packet processing solutions (even other types of BPF programs) in that it is capable of processing packets *before* they reach the main kernel networking stack, via direct memory access. In practice, this allows XDP to:

1) Retain a significant performance advantage over alternative methods;
2) Make filtering decisions (e.g. `DROP`, `REDIRECT`, `ALLOW`) about packets immediately after they cross the hardware boundary into the kernel;
3) Modify packet headers, packet contents, and redirect packets to different network interfaces, CPUs, or userspace programs.

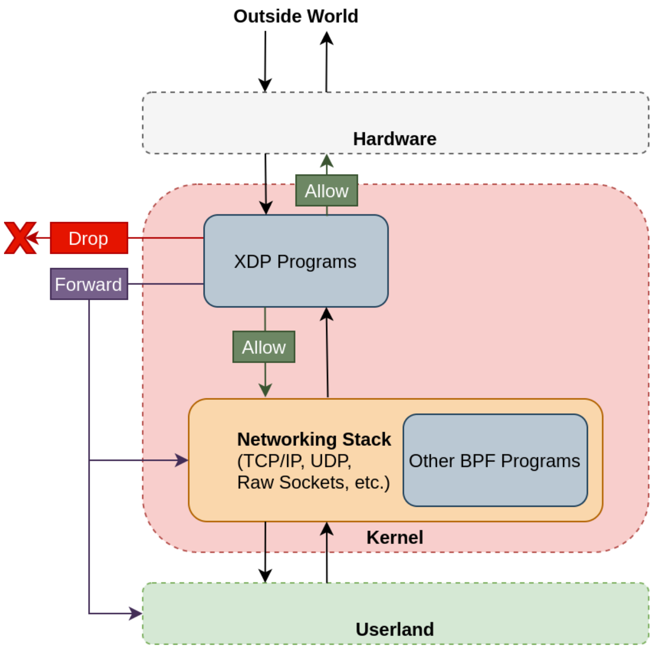

Fig. 3. A simplified overview of XDP dataflow with respect to hardware, the kernel networking stack, and userland. Note that XDP allows direct packet access, filtration, and even modification *before* reaching the networking stack and other BPF programs. Adapted from [20].

4.2.1 Cloudflare's XDP-Based DDoS Mitigation Stack. XDP's security applications are immediately apparent from the above properties. In particular, its speed coupled with its ability to preempt the kernel's main networking stack render it an ideal solution for NIDS (network intrusion detection systems) and DDoS mitigation. As of 2017, Cloudflare [2] had already begun integrating XDP into their DDoS mitigation pipeline and presented early empirical results that XDP would significantly improve performance overhead over their existing Netfilter-based approach in userspace. In 2018, Cloudflare presented `L4Drop` [13], which allows for automatically generated rules to be converted to deployable XDP programs for DDoS mitigation; `L4Drop` relies on a separate program, `Gatebot` to detect attacks; these attacks are then converted into XDP rules and deployed to prevent further packet loss. Empirical results from [13] show that `L4Drop` is capable of defeating large-scale DDoS attacks very quickly and efficiently, with an acceptable impact on performance.

4.2.2 Augmenting `iptables` with eBPF and XDP. Section 4.1 discussed `xt_bpf` [8], a kernel module which allows the expression of Netfilter rules with BPF programs. While this is an excellent starting point, it does not reach the full expressiveness of eBPF. Bertrone et al. [3, 4] proposed `bpf-iptables`, a full reimplementation of the traditional `iptables` firewall in eBPF. Their solution offers several promising characteristics:

1) Full backwards compatibility with existing `iptables` rule definitions;
2) Significant performance improvements for high numbers of rules;
3) Replacement of Netfilter connection tracking with an eBPF-based implementation.

While `bpf-iptables` is sophisticated, it suffers from a few drawbacks of eBPF at the time it was written, such as the inability to support loop constructs or limitations in the size of BPF programs [4, 28]. While these limitations are mostly circumvented using supported constructs such as eBPF tail calls, a more complete solution would be possible with the most recent updates to the eBPF verifier [25, 38]. The authors of `bpf-iptables` specifically noted that the limitations imposed by eBPF were prohibitive of attempting to implement more complex packet matching algorithms [4]; due to updates to the verifier, it may now be possible to revisit this implementation to write a more complete or more performant clone of `iptables`.

Additional work to move `iptables` functionality into eBPF has been done within the kernel community itself, in the form of `bpfilter` [5]. This work is similar to that of `bpf-iptables`, except that it functions more as a proof of concept, and does not include connection tracking logic or more complex rulesets beyond simply checking source and destination IP addresses. As such, `bpf-iptables` represents a more complete solution, especially if the authors were to revisit its implementation in light of recent additions to the verifier, as discussed previously.

4.2.3 eBPF Network Visualization for Observability and Security. One of the primary use cases for BPF programs is system-wide tracing for enhancing observability [19]. Thus far, the solutions covered in this paper have not provided enhanced network visibility – instead, they have opted to obfuscate the security from the user and focus on the generation of rules (either manually or automatically) and programmable expression of these rules through BPF. While there is nothing fundamentally wrong with this approach, it certainly begs the question as to whether or not other approaches that take advantage of eBPF's observability applications may provide additional security benefits to the user.

ntopng [10] is a network visibility and security tool designed by Deri et al. as a successor to the original ntop software. In 2019, Deri et al. [11] further extended ntopng to use eBPF for data collection; this enables ntopng to leverage the full extent of eBPF's system introspection capabilities rather than relying on the traditional packet inspection approach. In particular, ntopng makes use of tracepoints and kprobes to instrument critical functions within the kernel's networking stack and build a picture of system-level interactions with ingress and egress data. This allows for a rich visibility and security model beyond the traditional packet-based approaches. The ntopng authors obtain remarkable results both in terms of performance and security benefits [11]. The increased visibility granted by the eBPF approach enables ntopng to identify specific processes connecting to malware hosts, and determine the source and destination of network traffic in terms of processes, containers, and users. Experimental results also show that ntopng is able to outperform traditional solutions such as libcap that rely on Netfilter, while providing enhanced visibility.

Nam and Kim [29] take a different approach to network visibility and security with eBPF, electing to focus on Packet tracing but extend the traditional approach to be container aware. They do this by instrumenting a TC (traffic classifier) BPF program that targets the eth0 network interface which serves as a gateway for all inbound and outbound traffic. The eBPF program can hook into this interface to inspect various packet header data. This data is augmented by specialized tracing programs that intercept traffic on various physical and logical interfaces. The resulting data is then stored in a database for processing and visualization. The advantage of using eBPF here is that these interfaces can be traced in real time, with limited overhead, and the BPF programs can differentiate between traffic destined for specific containers and KVM virtual machines.

5 MOVING BEYOND THE NETWORKING STACK

While the systems described in the previous section each offer unique benefits in their own right, they all lack a focus on host-based observability and host-based policy, instead electing to focus on the operating system's networking stack. This approach neglects to consider the various other aspects of system behavior that eBPF can be used to observe — in other words, eBPF is not being used to its full potential in the security space.

To rectify this gap in the current research, this section will present two systems that I have built, ebpH and bpfbox, that use BPF programs to collect data about processes running on the system and enforce policy locally. Before presenting ebpH and bpfbox, however, it is worth considering some of classic BPF solutions for sandboxing that preceded them.

5.1 Revisiting Classic BPF

Although classic BPF [27] is meant to operate on packets in the link layer, Dewry [12] re-purposed its Linux implementation to allow for the definition of *system call* filtering rules which could be used to whitelist certain system calls once a process had entered secure computing (seccomp) mode [1]. This new method of defining dynamic system call filtering rules was called seccomp-bpf.

The motivation for using classic BPF to add system call filtering to seccomp came from a number of failures to integrate it with existing kernel subsystems [12], most notably the ftrace subsystem [33] which provides archaic support for tracepoint instrumentation. Instead of relying on system-wide methods for instrumenting system calls within a given process, Dewry made the observation that userland code should already have knowledge of the ABI it needs to use to communicate with kernelspace — the system calls themselves [12]. The idea was that userspace applications could use predefined BPF programs to instrument registers containing the correct system call number and arguments; these BPF programs would then be used to make filtering decisions about whether to allow or deny the system call.

seccomp-bpf is by no means a perfect solution, particularly in terms of usability [1]. Classic BPF programs must be written by hand, one instruction at a time, which can be a difficult task for programmers who are not familiar with the paradigm. Programmers must also have an intimate knowledge of the system calls that they wish to allow and deny within their application. This may not be immediately obvious, and great care must be taken to avoid certain pitfalls, such as allowing one system call that can be used to produce the same effect as another disallowed system call. These usability concerns, coupled with the fact that developers are required to modify their own applications in order to make use of seccomp-bpf ultimately diminish its adoptability and result in seccomp-bpf only being used in the most security-sensitive applications [1]. These factors motivate the creation of a fully application-transparent alternative that does not require intimate knowledge of BPF bytecode; such an alternative will be described in Section 5.2.

In addition to the usability concerns outlined above, seccomp-bpf itself only constitutes part of a complete sandboxing solution [1]. As such, it was traditionally necessary to integrate seccomp-bpf with other approaches, such as namespace isolation. Traditionally, doing this under the Linux kernel requires the CAP_SYS_ADMIN privilege, which means that processes would need to run with this privilege enabled in order to effectively sandbox themselves, a solution which is clearly far from ideal. MBOX [23] attempts to solve this problem by offering a means of sandboxing without requiring root privileges. To do so, it leverages seccomp-bpf and ptrace(2) to interpose on a process' system calls. Rather than simply restricting system calls, it *redirects* reads and writes to an isolated, checkpointed filesystem. Changes to this filesystem can then be committed to the host filesystem using version-control-like semantics. MBOX similarly handles network isolation, allowing the user to view a summary of network activity by the target application and optionally restrict network access altogether. Since MBOX relies on cooperation between seccomp-bpf and ptrace(2), it suffers from the drawbacks associated with the ptrace(2) interface — namely, relatively high overhead. seccomp-bpf allows this overhead to be reduced slightly by only interposing on relevant events, but even with this optimization, the incurred overhead is non-negligible.

5.2 bpfbox: Application-Transparent Sandboxing with eBPF

One of the primary drawbacks of seccomp-bpf's approach is its reliance on the modification of application source code in order to enforce sandboxing policy. This modification is not necessarily trivial, may be open to errors by developers, and in many cases requires significant refactoring of existing source code in order to integrate effectively [1]. Further, dynamically adding new sandboxing rules at runtime is not possible, once a process has already entered secure computing mode.

Extended BPF can solve this problem by providing a means of defining and enforcing sandboxing rules dynamically and externally to the target application. bpfbox is a prototype application that I have written in eBPF to allow users to write and enforce seccomp-bpf rules in such a manner. By moving rule definition and enforcement outside of the target application, bpfbox provides full application transparency to the target application. This means that developers no longer need to modify the source code of their applications in order for users to take full advantage of seccomp-bpf's capabilities, and these rules can be loaded and enforced dynamically, both before and during the execution of the target software.

This approach may draw immediate comparison with MBOX [23], which was presented in the previous section, but differs in a few key ways. Firstly, bpfbox relies solely on eBPF tracepoints to interpose on system calls, whereas MBOX requires the use of the ptrace(2) system call (and much of the performance overhead associated with it). Secondly, bpfbox allows for the system-wide enforcement of sandboxing policy, whereas MBOX must be used as a wrapper to execute the target application. Ultimately, bpfbox and MBOX attempt to solve fundamentally different problems, although they share a common subgoal of increased application transparency for sandboxing.

It is worth noting that although bpfbox reimplements seccomp-bpf's basic functionality, it constitutes a completely separate solution. Instead of using the seccomp(2) system call to enter a secure computing state, bpfbox uses tail-called BPF programs to define the entrypoint at which a given process should start enforcing. Once a process has started enforcing, it begins tail-calling a second set of BPF programs on every system call. These BPF programs define rules that specify which system call should be allowed; all system calls that do not match a rule are denied by default. Figure 4 provides an overview of the way bpfbox makes policy decisions.

To enforce policy, bpfbox makes use of the bpf_send_signal helper that was introduced to eBPF in Linux 5.3 [6]. This helper function allows extended BPF programs to send real-time signals to target userland processes from kernelspace. Since these signals come directly from the kernel, they are not associated with the delays that typically arise from sending signals between processes in userspace (which are instead delivered based on context switch timings). This means that bpfbox can enforce policy instantaneously with respect to the system call in question, completely externally to the target application.

While bpfbox currently attempts to preserve seccomp-bpf functionality, future work may involve modifying bpfbox to generate profiles automatically based on observed application behavior. This would likely constitute a hybrid approach, working in conjunction with user-defined rules and logging profile changes for later analysis. Naturally, user-defined rules would supersede the automatically generated ones.

5.3 ebpH: Building System Call Profiles with BPF Programs

eBPF's unique combination of broad scope, efficiency, and production safety makes it a very attractive data collection option for intrusion detection systems. However, current research tends to focus on network intrusion detection rather than host-based approaches. I wrote ebpH [14], an eBPF host-based anomaly detection system, to fill this gap in the existing research.

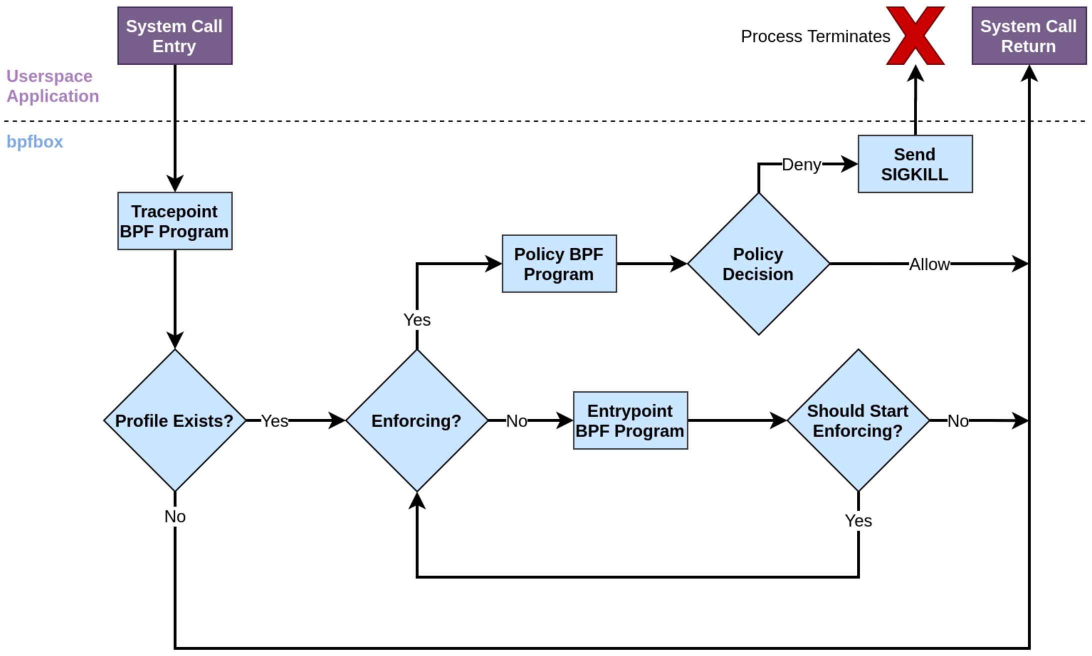

Fig. 4. An overview of the control flow of bpfbox when a traced process makes a system call.

ebpH effectively constitutes an eBPF reimplementation of pH (*Process Homeostasis*), an early anomaly detection system written by Somayaji [35]. The central idea of both systems is the same: using system call patterns to build behavioral profiles for running processes on the system, and flagging new entries to these profiles as anomalous [14, 15, 35]. The key difference between the two systems lies in implementation; whereas pH was implemented as a patch for Linux 2.2, ebpH is implemented entirely using eBPF programs and a userspace daemon [14]. In effect, this means that ebpH 's advantages over the original pH system roughly correspond to the advantages of eBPF over traditional kernel-based implementations, as presented in Section 3. In particular, the original pH system was necessarily unsafe for production deployments and lacked portability to future versions of Linux, while ebpH should be compatible with all versions of Linux higher than 5.3 and perfectly safe for use in production.

Indeed, experimental data from [14] has shown that ebpH is able to accomplish much of the same functionality as the original system (with plans for a full implementation in the future) and that in many cases it is able to either keep up with or even outperform the original system in terms of performance overhead. This is a remarkable result, and shows that eBPF can hold its own when compared with equivalent kernel-based implementations, even while instrumenting an immensely frequent, system-wide event such as system call invocations.

The results from preliminary testing of ebpH are significant beyond proving the merit of ebpH itself; rather, they are indicative that building host-based anomaly detections with eBPF is not only a feasible endeavor, but that it is worthwhile to do so in the first place. ebpH was able to implement the features of the original pH while retaining good performance, with the additional production safety guarantees associated with an eBPF implementation. This approach is extensible to other kernel-based approaches as well; in the future, it is likely that we will see many more kernel-based IDS implementations ported to eBPF, particularly as the technology continues to improve over time.

6 DISCUSSION

This paper has examined a variety of classic and extended BPF solutions for enhancing the security of Linux systems. Here, I will discuss the current research trends indicated by the results summarized in Section 4 and Section 5 and make suggestions for future work therein. Figure 5 presents an overview of the research covered in this paper and will help contextualize the rest of this section.

6.1 BPF Security Research Trends

As shown in Figure 5, with the exception of seccomp-bpf and derived solutions such as MBOX, both classic and extended BPF OS security research tends to focus on networking stack security with very little deviation. I believe that there are three factors largely responsible for this phenomenon:

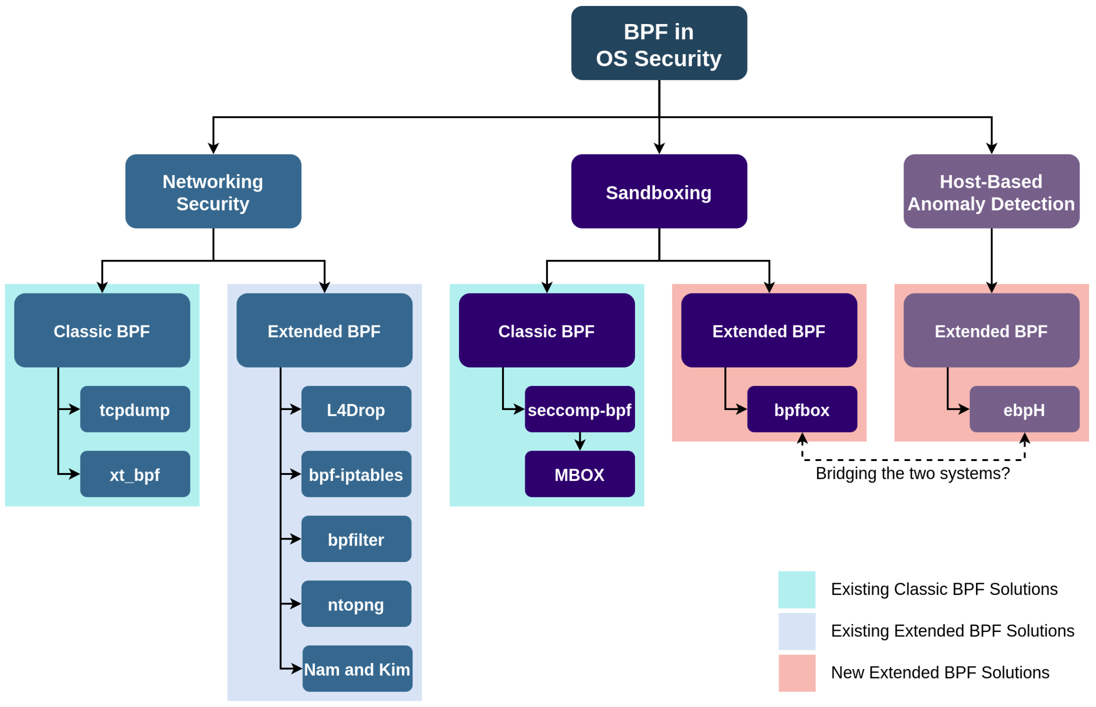

Fig. 5. An overview of the research covered in this paper. Note that existing eBPF solutions focus exclusively on networking stack security. ebpH and bpfbox are attempts at exploring other security applications of eBPF.

1) As classic BPF was largely focused on the networking stack, it is difficult to escape that same association in its extended counterpart;
2) XDP is a very powerful technology, and so presents a very compelling method to build network-focused security solutions;
3) Once a research trend has been established, it is likely that future work will continue to follow a similar path.

Existing research, while largely networking-stack-focused, may be grouped into several subcategories. For instance, some solutions, such as ntopng [11], focus on network visibility, while others focus predominantly on the definition and enforcement of policy. Both of these approaches may be effective in their own right, although there appears to be little cross-over between the two in the current literature. It may be worth examining the effectiveness of a combined approach that takes the enhanced visibility provided by a system like ntopng and combines it with the automatic rule generation provided by L4Drop [13].

Within the policy definition and enforcement subcategory, systems may be further divided into those that generate policy automatically and those that focus on user-defined rules. bpf-iptables [3, 4] and bpfilter [5] take the user-defined approach, whereas L4Drop [13] focuses on automatic policy generation. Naturally, the user-defined approach typically results in fewer false positives, but relies on experts to write policy by hand. Further, these experts must necessarily be familiar with the language used to define the rules in question. For systems that preserve widely-accepted semantics, such as bpf-iptables, this may not be such an issue, as most users will already be familiar with the syntax used to define rules, but for systems like L4Drop that rely on generated BPF programs for policy enforcement, the possibility of hand-written rules may not be a viable approach.

As eBPF is a relatively new technology in the Linux tracing landscape, it is under a constant state of change and is constantly being improved and iterated upon. Consequentially, many of the research systems discussed in this paper were limited by the idiosyncrasies of early versions of eBPF. For instance, solutions implemented before Linux 5.3 would have been constrained by a lack of bounded loop support and solutions implemented before Linux 5.1 would have been constrained by a significantly smaller upper limit on BPF instructions per program. These two changes alone invite the potential for improving many of the existing systems presented in this paper. As a simple example, L4Drop [13] relied on a separate program, Gatekeeper for attack detection, due to highly limited complexity bounds on BPF programs at the time — with modern eBPF, it would now be possible to achieve a fully eBPF/XDP-based implementation without relying on other technologies. This may have significant performance implications for `L4Drop`.

6.2 Future Work

As described in Section 3, eBPF can be used for far more than tracing network events; rather, it is a tool capable of instrumenting the entire system. Therefore, the focus of current research on the networking stack represents a small fraction of eBPF's capabilities concerning OS security. To rectify the lack of host-focused eBPF security research, I have presented two systems that use eBPF for security in novel ways, `ebpH` and `bpfbox`. Both of these systems are effective in their own right, and represent a step toward host-based policy creation and enforcement through eBPF-enabled observability. This section will present some topics for future work with respect to these two systems.

6.2.1 Extending `ebpH` *to Use Other Data Sources.* The current version of `ebpH` uses eBPF tracepoints to instrument system calls on every process on the system. These system calls are then contextualized into sequences that `ebpH` uses to inform the creation of profiles for each executable on the system [14]. While this approach mirrors that of its predecessor, `pH` [35], it is not necessarily conducive to monitoring modern applications that are often complex and non-deterministic in nature. To rectify this problem, future versions of `ebpH` might incorporate other sources of data in addition to system call sequences.

For example, `ebpH` could look at things like frequency of context switches, CPU usage, block I/O, network activity, memory allocations, or even user input. Due to eBPF's low overhead and the ability for eBPF programs to communicate with each other via direct map access, this endeavor would be significantly less complicated than previously would have been possible. By integrating several sources of data in this way, `ebpH` can get a much clearer idea of how processes behave normally, and therefore what constitutes abnormal behavior.

6.2.2 Extending `bpfbox` *to Generate Behavioral Profiles.* The results from `ebpH` [14] have shown that eBPF can be used to efficiently interpose on every system call, generating behavioral profiles for each executable on the system. It may be feasible to take a similar approach with `bpfbox`, profiling the behavior of applications under normal use and using this to generate profiles to inform future policy. If these profiles were semantically understandable by humans, they could even be modified manually to cover edge cases that did not come up in testing.

Inoue [21] has shown that this technique can be used to great effect in Java applications, although it remains to be seen whether this model can be extended to effectively capture the behavior of native applications. In order to effectively re-implement this technique in eBPF, it would be necessary to come up with a Java-like permission model for ordinary Linux applications, which would be a separate endeavor in and of itself. However, the benefits of integrating such functionality with `bpfbox` could be enormous.

7 CONCLUSION

Extended BPF represents a powerful new force in OS security, by providing an efficient, production-safe, and system-wide means of instrumenting all system behavior, from userspace functions to kernelspace functions to incoming packets, system calls, signals, and everything in between. Before the advent of eBPF, system introspection was relegated to poor alternatives that generally lacked in terms of the safety or efficiency required for reliable use in production. By solving this problem, eBPF opens new possibilities for security applications, resulting in the ability for the dynamic creation and enforcement of policy in real time, based on observed system behavior. This has implications for a wide variety of security solutions, such as intrusion detection, DDoS mitigation, and sandboxing.

Current trends in eBPF-related security research are only scratching the surface of what eBPF can do in this space. While new systems such as `ebpH` and `bpfbox` are exploring new possibilities for eBPF's security benefits, there still remains a lot of room for exploration in this field. In particular, I envision that future solutions will incorporate several aspects of system behavior in order to build a more complete model for analysis; one can already see this pattern beginning to emerge in existing solutions such as `ntopng` [11], which contextualizes network-level events based on system-level data.

Ultimately, the future of extended BPF in OS security is bright. Current solutions have shown that it can be an effective means of data collection, dynamic policy generation, and real-time policy enforcement; further, this can be achieved with the low overhead and high safety required for production use cases. As eBPF is a relatively new technology, it continues to improve rapidly over time and recent developments such as the introduction of bounded loop support and a higher upper limit on the number of BPF instructions per program have greatly increased the theoretical complexity limit of BPF programs. New helper functions like `bpf_send_signal` have provided means of enforcing policy directly from BPF programs. This trend of improvement is expected to continue for years to come, and will likely result in an even more attractive paradigm for the development of many future OS security endeavors.

REFERENCES

[1] J. Anderson, "A Comparison of Unix Sandboxing Techniques", *FreeBSD Journal*, 2017. [Online]. Available: https://www.engr.mun.ca/~anderson/publications/2017/sandbox-comparison.pdf.

[2] G. Bertin, "XDP in Practice: Integrating XDP into our DDoS Mitigation Pipeline", in *Technical Conference on Linux Networking, Netdev*, vol. 2, 2017.

[3] M. Bertrone, S. Miano, J. Pi, *et al.*, "Toward an eBPF-Based Clone of iptables", in *Netdev 18*, 2018. [Online]. Available: https://mbertrone.github.io/documents/20-eBPF-Iptables-Netdev.pdf.

[4] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating Linux Security with eBPF iptables", in *SIGCOMM '18: Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, Aug. 2018, pp. 108–110, ISBN: 978-1-4503-5915-3. DOI: 10.1145/3234200.3234228. [Online]. Available: https://doi.org/10.1145/3234200.3234228.

[5] D. Borkmann, *net: add bpfilter*, Feb. 2018. [Online]. Available: https://lwn.net/Articles/747504/.

[6] *bpf-helpers(7) Linux Programmer's Manual*, Linux, Nov. 2019.

[7] *bpf(2) Linux Programmer's Manual*, Linux, Aug. 2019.

[8] W. de Bruijn, "netfilter: x_tables: add xt_bpf match", The Linux Foundation, Kernel Patch, Jan. 2013. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e6f30c731718db45cec38096%204dfee210307cfc4a.

[9] J. Cespedes and P. Machata, *ltrace(1) Linux User's Manual*, Ltrace project, Jan. 2013.

[10] L. Deri, M. Martinelli, and A. Cardigliano, "Realtime High-Speed Network Traffic Monitoring Using ntopng", in *28th Large Installation System Administration Conference (LISA14)*, Seattle, WA: USENIX Association, Nov. 2014, pp. 78–88, ISBN: 978-1-931971-17-1. [Online]. Available: https://www.usenix.org/conference/lisa14/conference-program/presentation/deri-luca.

[11] L. Deri, S. Sabella, and S. Mainardi, "Combining System Visibility and Security Using eBPF.", in *ITASEC*, 2019. [Online]. Available: https://pdfs.semanticscholar.org/7d3f/8396a7407c62a0344ba%2034b9addc16f73bbfb.pdf.

[12] W. Dewry, "dynamic seccomp policies (using BPF filters)", The Linux Foundation, Mailing List RFC, Jan. 2012. [Online]. Available: https://lwn.net/Articles/475019/.

[13] A. Fabre, *L4Drop: XDP DDoS Mitigations*, Nov. 2018. [Online]. Available: https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/.

[14] W. Findlay, "Host-Based Anomaly Detection with Extended BPF", Honours Thesis, Carleton University, Apr. 2020. [Online]. Available: https://williamfindlay.com/written/thesis.pdf.

[15] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes", in *Proceedings 1996 IEEE Symposium on Security and Privacy*, May 1996, pp. 120–128. DOI: 10.1109/SECPRI.1996.502675.

[16] F. Fuentes and D. C. Kar, "Ethereal vs. Tcpdump: A Comparative Study on Packet Sniffing Tools for Educational Purpose", *J. Comput. Sci. Coll.*, vol. 20, no. 4, pp. 169–176, Apr. 2005, ISSN: 1937-4771.

[17] GNU Project, *GDB: The GNU Project Debugger*, 2020. [Online]. Available: https://www.gnu.org/software/gdb/.

[18] B. Gregg, *strace Wow Much Syscall*, 2014. [Online]. Available: http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html.

[19] B. Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019, ISBN: 0-13-655482-2.

[20] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, *et al.*, "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*, 2018, pp. 54–66. DOI: 10.1145/3281411.3281443. [Online]. Available: https://doi.org/10.1145/3281411.3281443.

[21] H. Inoue, "Anomaly Detection in Dynamic Execution Environments", PhD thesis, University of New Mexico, 2005. [Online]. Available: https://www.cs.unm.edu/~forrest/dissertations/inoue-dissertation.pdf.

[22] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad, "Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps", in *Proceedings of the Linux Symposium*, vol. 1, pp. 215–224. [Online]. Available: https://www.kernel.org/doc/ols/2007/ols2007v1-pages-215-224.pdf.

[23] T. Kim and N. Zeldovich, "Practical and Effective Sandboxing for Non-Root Users", in *The 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 139–144.

[24] B. Kuperman and E. Spafford, "Generation of Application Level Audit Data via Library Interposition", Sep. 1999.

[25] Linux, *kernel/bpf/verifier.c*, Mar. 2020. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/bpf/verifier.c?h=v5.6-rc5.

[26] LTTng Project, *LTTng v2.11 - LTTng Documentation*, Oct. 2019. [Online]. Available: https://lttng.org/docs/v2.11/.

[27] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", in *USENIX Winter 1993*, vol. 93, 1992. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf.

[28] S. Miano, M. Bertrone, F. Risso, *et al.*, "Creating Complex Network Services with eBPF: Experience and Lessons Learned", in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, IEEE, 2018, pp. 1–8. DOI: 10.1109/HPSR.2018.8850758. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8850758.

[29] T. Nam and J. Kim, "Open-Source IO Visor eBPF-Based Packet Tracing on Multiple Network Interfaces of Linux Boxes", in *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, Oct. 2017, pp. 324–326. DOI: 10.1109/ICTC.2017.8190996.

[30] Netfilter Project, *Netfilter*, Apr. 2020. [Online]. Available: https://www.netfilter.org/.

[31] *ptrace(2) Linux User's Manual*, Oct. 2019.

[32] Red Hat, *Understanding How SystemTap Works Red Hat Enterprise Linux 5*. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/systemtap_beginners_guide/understanding-how-systemtap-works.

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

[34] *Seccomp BPF (SECure COMPuting with Filters)*. [Online]. Available: https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.

[35] A. B. Somayaji, "Operating System Stability and Security through Process Homeostasis", PhD thesis, University of New Mexico, 2002. [Online]. Available: https://people.scs.carleton.ca/~soma/pubs/soma-diss.pdf.

[36] 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.

[37] A. Starovoitov, "net: filter: rework/optimize internal BPF interpreter's instruction set", The Linux Foundation, Kernel Patch, Mar. 2014. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e%206d0fd55dffc551b8.

[38] 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=2589726d12a1b12eaaa93c7f1ea64287e%20383c7a5.

[39] Strace Project, *strace*. [Online]. Available: https://strace.io/.

[40] *strace(1) Linux User's Manual*, 5.3, Strace Project, Sep. 2019.

[41] Sysdig Inc., *draios/sysdig*, Nov. 2019. [Online]. Available: https://github.com/draios/sysdig.

[42] *Tcpdump/Libpcap Public Repository*, Sep. 2010. [Online]. Available: https://www.tcpdump.org/.

[43] V. Weaver, *perf_event_open(2) Linux User's Manual*, Oct. 2019.