Extended Berkeley Packet Filter for Intrusion Detection Implementations

COMP4906 Honours Thesis Proposal

by

William Findlay

November 15, 2019

Under the supervision of Dr. Anil Somayaji
Carleton University

Abstract

System introspection is becoming an increasingly attractive option for maintaining operating system stability and security. This is primarily due to the many recent advances in system introspection technology; in particular, the 2013 introduction of *eBPF* (*Extended Berkeley Packet Filter*) into the Linux Kernel [1] along with the recent development highly usable interfaces such as *bcc* (*BPF Compiler Collection*) [2] has resulted in highly compelling, performant, and (perhaps most importantly) safe subsystem for both kernel and userland instrumentation.

The proposed thesis seeks to test the limits of what eBPF programs are capable of with respect to the domain of computer security; specifically, I present *ebpH* (*Extended Berkeley Process Homeostasis*), an eBPF-based intrusion detection system based on Anil Somayaji's [3] *pH* (*Process Homeostasis*). Preliminary testing has shown that ebpH is able to detect anomalies in process behavior by instrumenting system call tracepoints with negligible overhead. Future work will involve testing and iterating on the ebpH prototype, as well as the implementation of several kernel patches to further extend its functionality.

Keywords: eBPF, intrusion detection, system calls, Linux Kernel introspection

Acknowledgments

Contents

List of Figures

List of Tables

List of Listings

1 Introduction and Motivation

As our computer systems grow increasingly complex, so too does it become more and more difficult to gauge precisely what they are doing at any given moment. Modern computers are often running hundreds, if not thousands of processes at any given time, the vast majority of which are running silently in the background. As a result, users often have a very limited notion of what exactly is happening on their systems, especially beyond that which they can actually see on their screens. An unfortunate corollary to this observation is that users *also* have no way of knowing whether their system may be *misbehaving* at a given moment, whether due to a malicious actor, buggy software, or simply some unfortunate combination of circumstances.

Recently, a lot of work has been done to help bridge this gap between system state and visibility, particularly through the introduction of powerful new tools such as *Extended Berkeley Packet Filter* (eBPF). Introduced to the Linux Kernel in a 2013 RFC and subsequent kernel patch [1], eBPF offers a promising interface for kernel introspection, particularly given its scope and unprecedented level of safety therein; although eBPF can examine any data structure or function in the kernel through the instrumentation of tracepoints, its safety is guaranteed via a bytecode verifier. What this means in practice is that we effectively have unlimited, highly performant, production-safe system introspection capabilities that can be used to monitor as much or as little system state as we desire.

Certainly, eBPF offers unprecedented system state visibility, but this is only scratching the surface of what this technology is capable of. With limitless tracing capabilities, we can construct powerful applications to enhance system security, stability, and performance. In theory, these applications can perform much of their work autonomously in the background, but are equally capable of functioning in a more interactive role, keeping the end user informed about changes in system state, particularly if these changes in state are undesired. To that end, I propose *ebpH* (*Extended Berkeley Process Homeostasis*), an intrusion detection system based entirely on eBPF that monitors process state in the form of system call sequences. By building and maintaining per-executable behavior profiles, ebpH can dynamically detect when processes are behaving outside of the status quo, and notify the user so that they can understand exactly what is going on.

A prototype of ebpH has been written using the Python interface provided by *bcc* (*BPF Compiler Collection*) [2], and preliminary tests show that it is capable of monitoring system state under moderate to heavy workloads with negligible overhead. What's more, zero kernel panics occurred during ebpH's development and early testing, which simply would not have

been possible without the safety guarantees that eBPF provides. The rest of this proposal will cover the necessary background material required to understand ebpH, describe several aspects of its implementation, including the many findings and pitfalls encountered along the way, and discuss the planned methodology for testing and iterating on this prototype going forward.

2 Background

In the following sections, I will provide the necessary background information needed to understand ebpH; this includes an overview of system introspection and tracing techniques on Linux including eBPF itself, and some background on system calls and intrusion detection.

While my work is primarily focused on the use of eBPF for maintaining system security and stability, the working prototype for ebpH borrows heavily from Anil Somayaji's *pH* or *Process Homeostais* [3], an anomaly-based intrusion detection and response system written as a patch for Linux Kernel 2.2. As such, I will also provide some background on the original pH system and many of the design choices therein.

2.1 An Overview of the Linux Tracing Landscape

System introspection is hardly a novel concept; for years, developers have been thinking about the best way to solve this problem and have come up with several unique solutions, each with a variety of benefits and drawbacks. Table 2.1 presents an overview of some prominent examples relevant to GNU/Linux systems.

These technologies can, in general, be classified into a few broad categories (Figure 2.1), albeit with potential overlap depending on the tool:

(1) Userland libraries.
(2) Ptrace-based instrumentation.
(3) Loadable kernel modules.
(4) Kernel subsystems.

Applications such as strace [4], [5] which make use of the ptrace system call are certainly a viable option for limited system introspection with respect to specific processes. However, this does not represent a complete solution, as we are limited to monitoring the system calls made by a process to communicate with the kernel, its memory, and the state of its registers, rather than the underlying kernel functions themselves [16]. The scope of ptrace-based solutions is also limited by ptrace's lack of scalability; ptrace's API is conducive to tracing single

Table 2.1: A summary of various system introspection technologies available for GNU/Linux systems.

Name	Interface and Implementation	Citations
strace	Uses the ptrace system call to trace userland processes	[4], [5]
ltrace	Uses the ptrace system call to trace library calls in userland processes	[6], [7]
SystemTap	Dynamically generates loadable kernel modules for instrumentation; newer versions can optionally use eBPF as a backend instead	[8], [9]
ftrace	Sysfs pseudo filesystem for tracepoint instrumentation located at `/sys/kernel/debug/tracing`	[10]
perf_events	Linux subsystem that collects performance events and returns them to userspace	[11]
LTTng	Loadable kernel modules, userland libraries	[12]
dtrace4linux	A Linux port of DTrace via a loadable kernel module	[13]
sysdig	Loadable kernel modules for system monitoring; native support for containers	[14]
eBPF	In-kernel virtual machine for running pre-verified bytecode	[1], [2], [15]

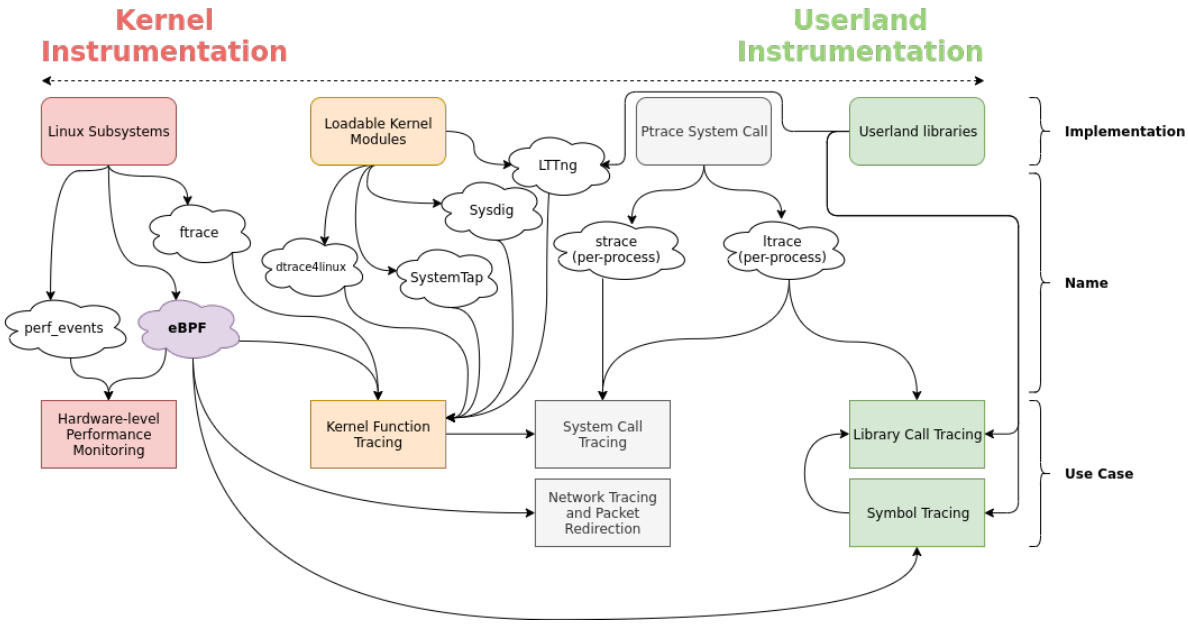

Figure 2.1: A high level overview of the broad categories of Linux instrumentation. This does not represent a complete picture of all available tools and interfaces, but instead presents many of the most popular ones. Note how eBPF covers every presented use case.

processes at a time rather than tracing processes system wide. Its limited scale becomes even more obvious when considering the high amount of context-switching between kernel space and user space required when tracing multiple processes or threads, especially when these processes and threads make many hundreds of system calls per second [17].

Although library call instrumentation through software such as ltrace [6], [7] does not necessarily suffer from the same performance issues as described above, it still constitutes a suboptimal solution for many use cases due to its limited scope. In order to be effective and provide a complete picture of what exactly is going on during a given process' execution, library tracing needs to be combined with other solutions. In fact, ltrace does exactly this; when the user specifies the `-S` flag, ltrace uses the ptrace system call to provide strace-like system call tracing functionality.

LKM-based implementations such as sysdig [14] and SystemTap [8] offer an extremely deep and powerful tracing solution given their ability to instrument the entire system, including the kernel itself. Their primary detriment is a lack of safety guarantees with respect to the modules themselves. No matter how vetted or credible a piece of software might be, running it natively in the kernel always comports with an inherent level of risk; buggy code might cause system failure, loss of data, or other unintended and potentially catastrophic consequences.

Custom tracing solutions through kernel modules carry essentially the same risks. No sane manager would consent to running untrusted, unvetted code natively in the kernel of a production system; the risks are simply too great and far outweigh the benefits. Instead, such code must be carefully examined, reviewed, and tested, a process which can potentially take months. What's more, even allowing for a careful testing and vetting process, there is always some probability that a bug can slip through the cracks, resulting in the catastrophic consequences outlined above.

Built-in kernel subsystems for instrumentation seem to be the most desirable choice of any of the presented solutions. In fact, eBPF [1] itself constitutes one such solution. However, for the time being, we will focus on a few others, namely ftrace [10] and perf_events [11] (eBPF programs actually *can* and *do* use both of these interfaces anyway). While both of these solutions are safe to use (assuming we trust the user), they suffer from limited documentation and relatively poor user interfaces. These factors in tandem mean that ftrace and perf_events, while quite useful for a variety of system introspection needs, are less extensible than other approaches.

2.1.1 Dtrace

It is worth spending a bit more time comparing eBPF with Dtrace, as both APIs are quite full-featured and designed with similar functionality in mind. dtrace4linux [13] is a free and open source port of Sun's Dtrace for the Linux Kernel, implemented as a loadable kernel module (LKM).

TODO: cite [18] here to talk about differences.

2.2 eBPF: Linux Tracing Superpowers

In 2016, eBPF was described by Brendan Gregg [19], a prominent member of the Iovisor project, as nothing short of *Linux tracing superpowers*. I echo that sentiment here, as it summarizes eBPF's capabilities perfectly. Through eBPF programs, we can simultaneously trace userland symbols and library calls, kernel functions and data structures, and hardware performance. What's more, through an even newer subset of eBPF, known as *XDP* or *Express Data Path* [20], we can inspect, modify, redirect, and even drop packets entirely before they even reach the main kernel network stack. Figure 2.2 provides a high level overview of these use cases and the corresponding eBPF instrumentation required.

The advantages of eBPF extend far beyond scope of traceability; eBPF is also extremely performant, and runs with guaranteed safety. In practice, this means that eBPF is an ideal tool for use in production environments, and at scale.

Safety is guaranteed with the help of an in-kernel verifier that checks all submitted bytecode before its insertion into the BPF virtual machine. While the verifier does limit what is possible (eBPF in its current state is **not** Turing complete), it is constantly being improved; for example, a recent patch [21] that was mainlined in the Linux 5.3 kernel added support for verified bounded loops, which greatly increases the computational possibilities of eBPF. The verifier will be discussed in further detail in Subection 2.2.2.

eBPF's superior performance can be attributed to several factors. On supported architectures,[1] eBPF bytecode is compiled into machine code using a *just-in-time* (*JIT*) compiler; this both saves memory and reduces the amount of time it takes to insert an eBPF program into the kernel. Additionally, since eBPF runs in-kernel and communicates with userland via map access and perf events, the number of context switches required between userland and the kernel is greatly diminished – we can simply access the necessary data as needed.

[1] x86-64, SPARC, PowerPC, ARM, arm64, MIPS, and s390 [22]

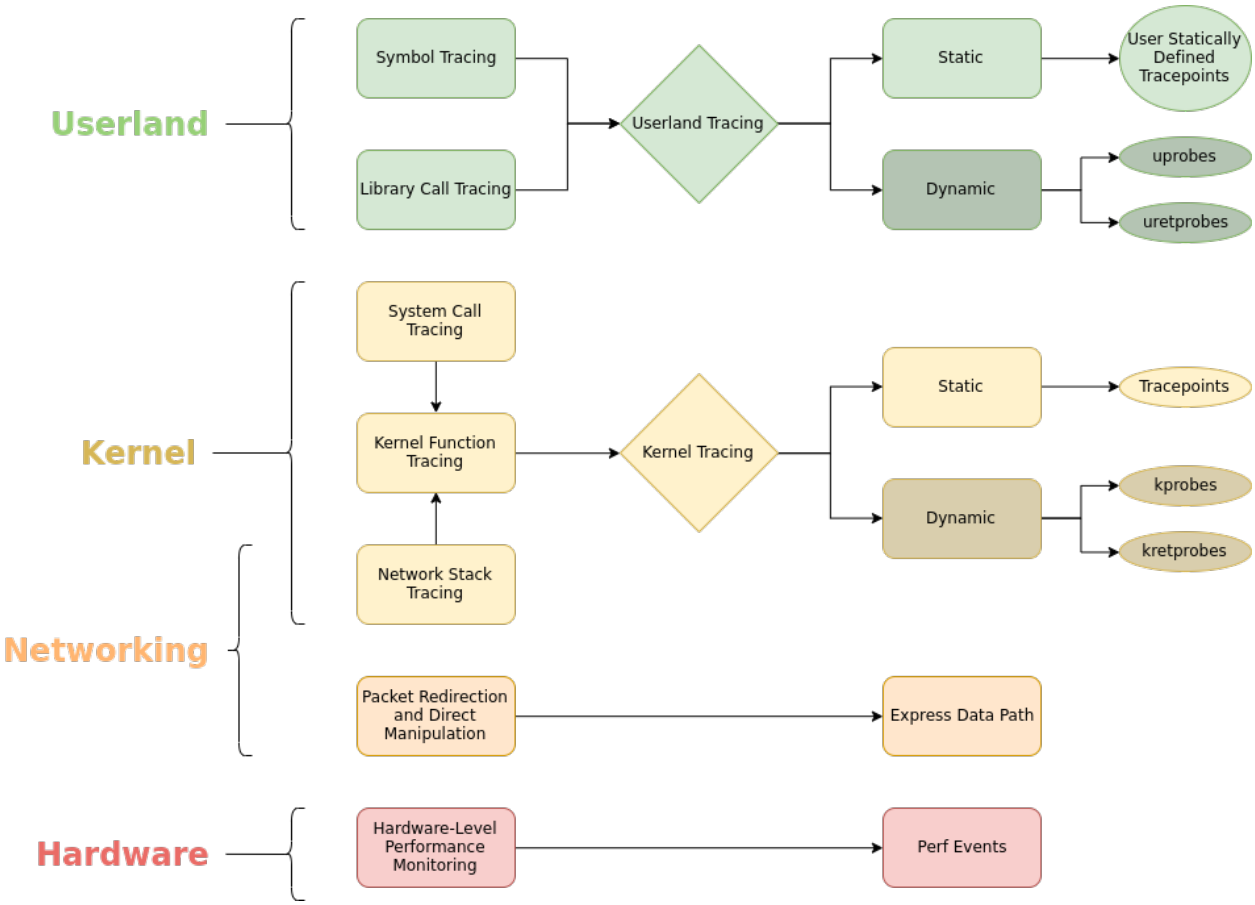

Figure 2.2: A high level overview of various eBPF use cases. Note the high level of flexibility that eBPF provides with respect to system tracing.

2.2.1 How eBPF Works at a High Level

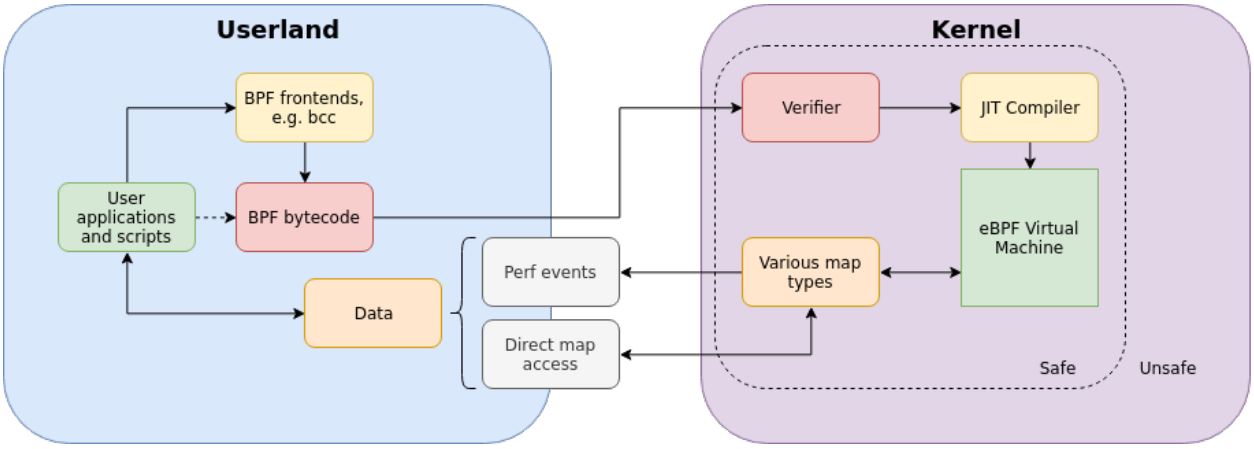

Figure 2.3: Basic topology of eBPF with respect to userland and the kernel. Note the bidirectional nature of dataflow between userspace and kernelspace using maps.

2.2.2 The Verifier: The Good, the Bad, and the Ugly

2.3 System Calls

2.4 Intrusion Detection

2.5 Process Homeostasis

2.6 Other Related Work

3 Implementing ebpH

4 Methodology

References

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

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

[3] A. B. Somayaji, "Operating system stability and security through process homeostasis," PhD thesis, Anil Somayaji, 2002. [Online]. Available: https://people.scs.carleton.ca/~soma/pubs/soma-diss.pdf.

[4] *Strace*. [Online]. Available: https://strace.io/.

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

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

[7] J. Cespedes and P. Machata, *Ltrace(1) linux user's manual*, Ltrace project, Jan. 2013.

[8] *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.

[9] A. Merey, *Introducing stapbpf - systemtap's new bpf backend*, Dec. 2017. [Online]. Available: https://developers.redhat.com/blog/2017/12/13/introducing-stapbpf-systemtaps-new-bpf-backend/.

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

[11] V. Weaver, *Perf_event_open(2) linux user's manual*, Oct. 2019.

[12] *Lttng v2.11 - lttng documentation*, Oct. 2019. [Online]. Available: https://lttng.org/docs/v2.11/.

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

[14] *Draios/sysdig*, Nov. 2019. [Online]. Available: https://github.com/draios/sysdig.

[15] S. Goldstein, "The next linux superpower: Ebpf primer," USENIX SRECon16 Europe, Jul. 2016. [Online]. Available: https://www.usenix.org/conference/srecon16europe/program/presentation/goldshtein-ebpf-primer.

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

[17] 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.linuxsecrets.com/kdocs/mirror/ols2007v1.pdf#page=41.

[18] B. Gregg, *Bpftrace (dtrace 2.0) for linux 2018*, Oct. 2018. [Online]. Available: http://www.brendangregg.com/blog/2018-10-08/dtrace-for-linux-2018.html.

[19] ——, *Linux bpf superpowers*, Mar. 2016. [Online]. Available: http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html.

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

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

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

Appendix A eBPF Design Patterns

Listing A.1: Handling large data in eBPF programs.

```
/* This is way too large to fit within the eBPF stack
 * limit of 512 bytes */
struct bigdata_t
{
    char foo[4096];
};

/* We read from this array every time we want to
 * initialize a new struct bigdata_t */
BPF_ARRAY(__bigdata_t_init, struct bigdata_t, 1);

/* Suppose this is a function where we need to use our
 * bigdata_t struct*/
int some_bpf_function(void)
{

    /* .... */

    /* We use this to look up from our __bigdata_t_init array */
    int zero = 0;
    struct bigdata_t *bigdata; /* A pointer to a bigdata_t */

    /* Read the zeroed struct from our array */
    bigdata = __bigdata_t_init.lookup(&zero);
    /* Make sure that bigdata is not NULL */
    if (!bigdata)
        return 0;
    /* Get a new address for bigdata */
    bpf_probe_read(bigdata, sizeof(struct bigdata_t), bigdata);

    /* .... */

    return 0;
}
```