Extended Berkeley Packet Filter for Intrusion Detection Implementations

COMP4906 Honours Thesis Proposal

by

William Findlay

November 25, 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 byte-code	[1], [2], [15]

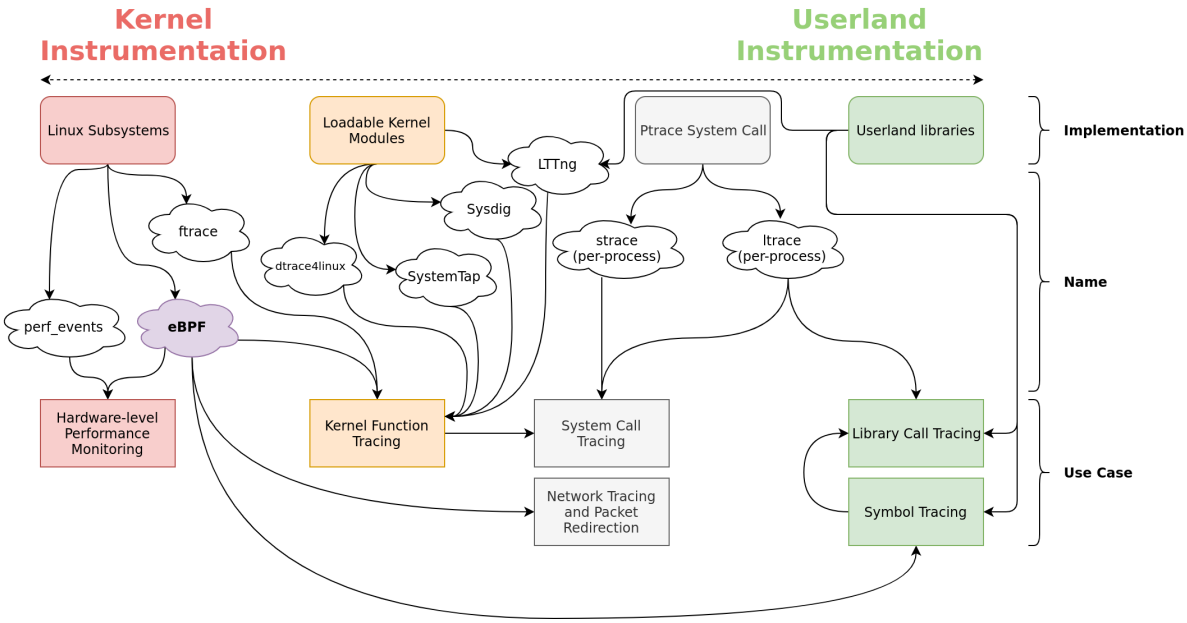

Figure 2.1: A high level overview of the broad categories of Linux instrumentation. This does not represent a complete picture of all available tools and interfaces, but instead presents many of the most popular ones. Note how eBPF covers every presented use case.

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). While Dtrace offers a powerful API for full-system tracing, its usefulness is, in general, eclipsed by that of eBPF [18] and requires extensive shell scripting for use cases beyond one-line tracing scripts. In contrast, with the help of powerful and easy to use front ends like bcc [2], developing complex eBPF programs for a wide variety of use cases is becoming an increasingly painless process.

Not only does eBPF cover more complex use cases than Dtrace, but it also provides support for simple one-line programs through tools like bpftrace [18], [19] which has been designed to provide a high-level Dtrace-like tracing language for Linux using eBPF as a backend. Although bpftrace only provides a subset of Dtrace's functionality [18], its feature set has been carefully curated in order to cater to the most common use cases and more functionality is being added on an as-needed basis.

Additional work is being done to fully reimplement Dtrace as a new BPF program type [20] which will further augment eBPF's breadth and provide full backward compatibility for existing Dtrace scripts to work with eBPF. This seems to be by far the most promising avenue for Linux Dtrace support thus far, as it seeks to combine the advantages of Dtrace with the speed, power, and safety of eBPF.

2.2 eBPF: Linux Tracing Superpowers

In 2016, eBPF was described by Brendan Gregg [21] 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* [22], 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.

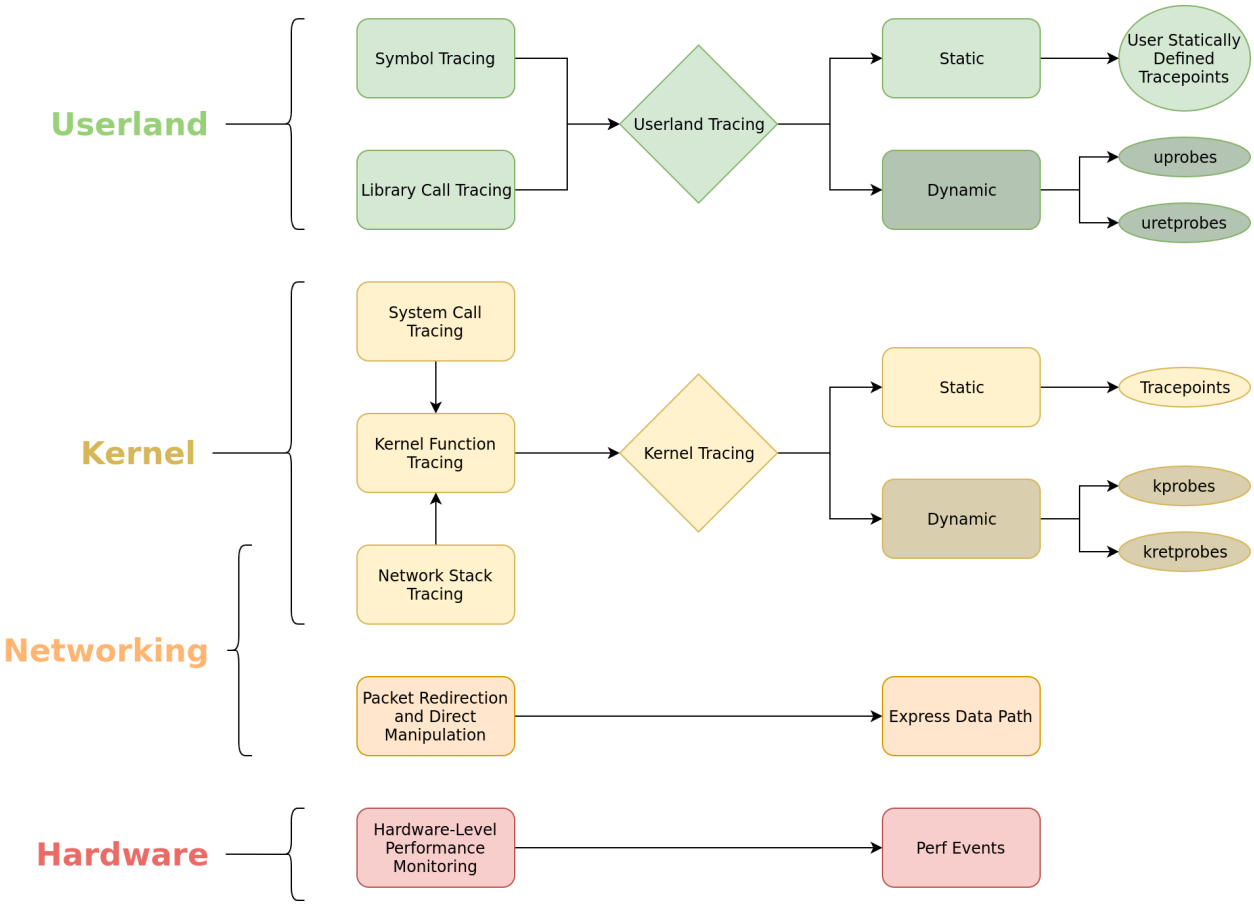

Figure 2.2: A high level overview of various eBPF use cases. Note the high level of flexibility that eBPF provides with respect to system tracing.

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 [23] 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, especially compared to approaches such as the ptrace system call.

2.2.1 How eBPF Works at a High Level

From the perspective of a user, the eBPF workflow is surprisingly simple. Users can elect to write eBPF bytecode directly (not recommended) or use one of many front ends to write in higher level languages that are then used to generate the respective bytecode. bcc [2] offers front ends for several languages including Python, Go, C/C++; users write eBPF programs in C and interact with bcc's API in order to generate eBPF bytecode and submit it to the kernel.

Figure 2.3 presents an overview of the eBPF workflow with respect to the interaction between userland applications and eBPF programs. Considering bcc's Python front end as an example: The user writes their BPF program in C and a user interface in Python. Using a provided BPF class, the C code is used to generate bytecode which is then submitted to the verifier to be checked for safety. Assuming the BPF program passes all required checks, it is then loaded into an in-kernel virtual machine. From there, we are able to attach onto various probes and tracepoints in the kernel.

Communication between userland and the kernel is done through various maps.

TODO: finish this

[1] x86-64, SPARC, PowerPC, ARM, arm64, MIPS, and s390 [24]

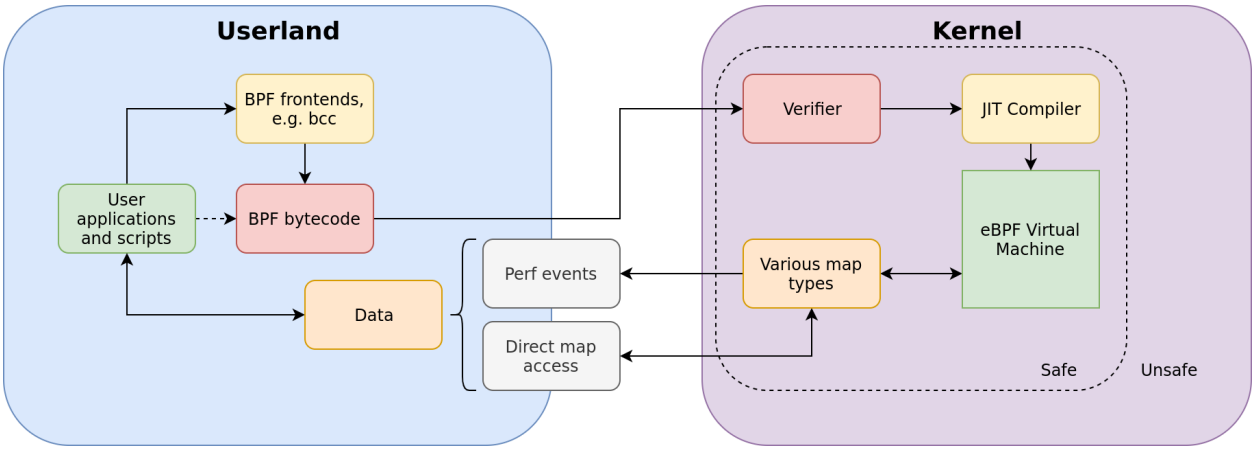

Figure 2.3: Basic topology of eBPF with respect to userland and the kernel. Note the bidirectional nature of dataflow between userspace and kernelspace using maps.

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

2.3 System Calls

2.4 Intrusion Detection

2.5 Process Homeostasis

2.5.1 Homeostasis

2.5.2 System Call Sequencing Through Lookahead Pairs

2.6 Other Related Work

3 Implementing ebpH

At a high level, ebpH is an intrusion detection system that profiles executable behavior by sequencing the system calls that processes make to the kernel. This functionality serves as an implementation of the original pH system described by Somayaji [3]. What makes ebpH unique is its use of an eBPF program for system call instrumentation and profiling (in contrast to the original pH which was implemented as a Linux 2.2 kernel patch).

ebpH can be thought of as a combination of several distinct components, functioning in two different parts of the system: userspace, and kernelspace (specifically within the eBPF virtual machine). In particular it includes a daemon, a CLI, and a GUI (described in Subection 3.1) as well as a BPF program (described in Subection 3.2 and onwards). The dataflow between these components is depicted in Figure 3.1.

In order to implement the ebpH prototype described here, it was necessary to circumvent several challenges associated with the eBPF verifier and make several critical design choices with respect to dataflow between userspace and the eBPF virtual machine running in the kernel. This section attempts to explain these design choices, describe any specific challenges faced, and justify why eBPF was ultimately well-suited to an implementation of this nature.

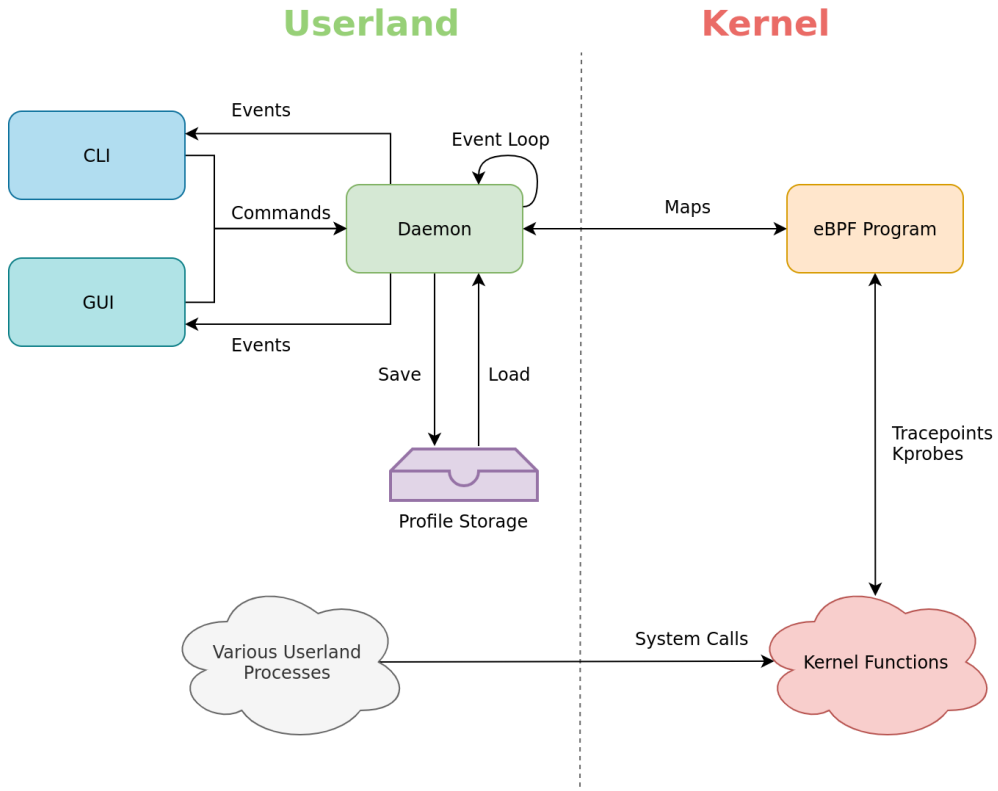

Figure 3.1: The dataflow between various components of ebpH.

3.1 Userspace Components

The userspace components of ebpH are comprised of three distinct programs. The **ebpH Daemon** (*ebpHD*) is responsible for initially compiling and submitting the eBPF program, as well as communication between userspace and the in-kernel eBPF program. As part of this communication, it loads existing profiles from the disk and saves new and modified profiles to disk at regular intervals. The **ebpH CLI** is a command line interface with which the user can send commands to the ebpH Daemon and inspect an overview of their system including all existing profiles and traced processes. The **ebpH GUI** performs the same functions as the ebpH CLI, except it presents information and commands in the form of a graphical user interface.

3.1.1 The ebpH Daemon

The ebpH Daemon is implemented as a Python3 script that runs as a daemonized background process. When started, the daemon uses bcc's Python front end [2] to generate the BPF bytecode responsible for tracing system calls, building profiles, and detecting anomalous behavior. It then submits this bytecode to the verifier and JIT compiler for insertion into the eBPF virtual machine.

Once the eBPF program is running in the kernel, the daemon continuously polls a set of specialized BPF maps called perf buffers which are updated on the occurrence of specific events. Table 3.1 presents an overview of the most important events we care about. As events are consumed, they are handled by the daemon and removed from the buffer to make room for new events. These buffers offer a lightweight and efficient method to transfer data from the eBPF program to userspace, particularly since buffering data significantly reduces the number of required context switches.

Table 3.1: Main event categories in ebpH.

Event	Description	Memory Overhead
`ebpH_on_anomaly`	Reports anomalies in specific processes and which profile they were associated with	2^8 pages
`ebpH_on_create_profile`	Reports when new profiles are created	2^8 pages
`ebpH_on_pid_assoc`	Reports new associations between PIDs and profiles	2^8 pages
`ebpH_error`	A generic event for reporting errors to userspace	2^2 pages
`ebpH_warning`	A generic event for reporting warnings to userspace	2^2 pages
`ebpH_debug`	A generic event for reporting debug information to userspace	2^2 pages
`ebpH_info`	A generic event for reporting general information to userspace	2^2 pages

In addition to perf buffers, the daemon is also able to communicate with the eBPF program through direct access to its maps. We use this direct access to issue commands to the eBPF program, check program state, and gather several statistics, such as profile count, anomaly count, and system call count. At the core of ebpH's design philosophy is the combination of system visibility and security, and so providing as much information as possible about system state is of paramount importance.

The daemon's final task is to automatically save profiles to disk at a specified interval

configurable by the user. It loads these profiles every time the eBPF program is reloaded so that profile data remains consistent between runs.

3.1.2 The ebpH CLI

3.1.3 The ebpH GUI

3.2 ebpH Profiles

In order to monitor process behavior, ebpH keeps track of a unique profile for each executable on the system. It does this by maintaining a hashmap of profiles, hashed by a unique per-executable ID; this ID is a 64-bit unsigned integer which is calculated as a unique combination of filesystem device number and inode number:

$$\texttt{key} = (\texttt{device number} << 32) + \texttt{inode number}$$

where $<<$ is the left bitshift operation. In other words, we take the filesystem's device ID in the upper 32 bits of our key, and the inode number in the lower 32 bits. This method provides a simple and efficient way to uniquely map keys to profiles.

The profile itself is a C data structure that keeps track of information about the executable, as well as a sparse two-dimensional away of lookahead pairs to keep track of system call sequences. Each entry in this array consists of an 8-bit integer, with the i^{th} bit corresponding to a previously observed distance of i between the two calls. Listing 3.1 presents a simplified definition for the ebpH profile struct.

Listing 3.1: A simplified definition of the ebpH profile struct.

```
1  struct ebpH_profile
2  {
3      u8 frozen;          /* Is the profile frozen? */
4      u8 normal;          /* Is the profile normal? */
5      u64 normal_time;    /* Minimum system time required for normalcy */
6      u64 normal_count;   /* Normal syscall count */
7      u64 last_mod_count; /* Syscalls since profile was last modified */
8      u64 train_count;    /* Syscalls seen during training */
9      u64 anomalies;      /* Number of anomalies in the profile */
10     u8 flags[SYSCALLS][SYSCALLS]; /* System call lookahead pairs */
11     u64 key;            /* Uniquely computed executable key */
12     char comm[16];      /* Name of the executable file */
13 };
```

Each process (Subection 3.3) is associated with exactly one profile at a time. Profile

association is updated whenever we observe a process making a call to `execve`. Whenever a process makes a system call, ebpH looks up its associated profile, and sets the appropriate lookahead pairs according to the process' most recent system calls.

3.3 Tracing Processes

Like profiles, process information is also tracked through a global hashmap of process structs. The process struct's primary purpose is to maintain the association between a process and its profile, maintain a sequence of system calls, and keep track of various metadata. See Listing 3.2 for a simplified definition of the ebpH process struct.

Listing 3.2: A simplified definition of the ebpH process struct.

```
1 struct ebpH_process
2 {
3     long seq[8];  /* Remember 8 most recent system calls in order */
4     u8 count;     /* How many system calls are in our sequence? */
5     u32 pid;      /* What is our PID? */
6     u64 profile_key; /* Associated profile key */
7     u8 in_execve; /* Are we in the middle of an execve? */
8 };
```

ebpH monitors process behavior by instrumenting tracepoints for both system call entry and return. The eight most recent system calls made by each process are stored in its respective process struct, and are used to set the correct lookahead pairs in the associated profile struct.

While we keep track of every system call made by a process, we pay special attention to a select few system calls which are directly related to profile creation, association, and disassociation. These system calls and their respective side effects are summarized in Table 3.2.

Profile Creation and Association with Execve There are several important considerations here. First, we need a way to assign profiles to processes, which is done by instrumenting the `execve` system call using a tracepoint, as well as part of its underlying implementation via a kprobe. In particular, we hook the `do_open_execat` kernel function in order to access the file's inode and filesystem information; without this, we would be unable to differentiate two paths that look like `/usr/bin/ls` and `./ls`.

The entry and exit points to the `execve` system call are used to differentiate a true `execve` call from the kernel routines responsible for loading shared libraries, which both invoke the aforementioned `do_open_execat` subroutine. When we first hit an `execve`, we set an indicator

Table 3.2: Important system calls in ebpH.

System Call	Description	ebpH Side Effect
`execve`	Execute a program	(Re)associate a process with a profile, creating the profile if necessary
`exit`	Terminate the calling process	Stop tracing a process
`exit_group`	Terminate all threads in a process	
`fork`	Create a new process by duplicating calling process	Start tracing a process and associate with parent's profile
`vfork`	Create a child process and block parent	
`clone`	Create a new process or thread	

variable in the process struct to say that we are in the middle of an `execve`. Subsequent calls to `do_open_execat` are then ignored until we hit `execve`'s return tracepoint and unset the indicator variable.

Reaping Processes with Exit, Exit_group, and Signals We use a combination of system call tracepoints and signal handler kprobes in order to determine when to stop tracing a particular PID. This is important for a few reasons, primarily due to map size considerations; by reaping process structs from our map as we are finished with them we ensure that: a) the map never fills up and; b) the map does not consume more memory than necessary.

Processes are reaped from ebpH's map whenever it detects an `exit` or `exit_group` system call. Threads are reaped whenever we observe a `SIGTERM` or `SIGKILL` signal, the latter of which forms the underlying implementation for `exit_group`.

Profile Association with Fork, Vfork, and Clone The final special consideration is with respect to `fork` and `clone` family system calls. In particular, we want to be able to track child process behavior as well as parent process behavior. In order to accomplish this, we simply instrument tracepoints for `fork`, `vfork`, and `clone` system calls, ensuring that we associate the child process with the parent's profile, if it exists. If ebpH detects an `execve` as outlined above, it will simply overwrite the profile association provided by the initial fork.

3.4 Anomaly Detection

ebpH profiles are tracked in two phases, *training mode* and *testing mode*. Once ebpH has decided that a pr

3.5 Reporting Anomalies to Userspace

4 Methodology

References

[1] 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.

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

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

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

[22] 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.

[23] 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.

[24] 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 types 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);

/* The main hashmap used to store our data */
BPF_HASH(bigdata_hash, u64, struct bigdata_t);

/* 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;
    /* A pointer to a bigdata_t */
    struct bigdata_t *bigdata;
    /* The key into our main hashmap
     * Its value not important for this example */
    u64 key = SOME_VALUE;

    /* Read the zeroed struct from our array */
    bigdata = __bigdata_t_init.lookup(&zero);
    /* Make sure that bigdata is not NULL */
    if (!bigdata)
        return 0;
    /* Copy bigdata to another map */
    bigdata = bigdata_hash.lookup_or_try_init(&key, bigdata);

    /* Perform whatever operations we want on bigdata... */

    return 0;
}
```