Host-Based Anomaly Detection with Extended BPF

COMP4906 Honours Thesis

William Findlay

March 30, 2020

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 *Extended Berkeley Packet Filter* (*eBPF*) into the Linux Kernel [54, 55] along with the recent development of more usable interfaces such as the *BPF Compiler Collection* (*bcc*) [20] has resulted in a highly compelling, performant, and (perhaps most importantly) safe subsystem for both kernel and userland instrumentation.

The scope, safety, and performance of eBPF system introspection has potentially powerful applications in the domain of computer security. In order to demonstrate this, I present *ebpH*, an eBPF implementation of Somayaji's [49] *Process Homeostasis* (*pH*). ebpH is an intrusion detection system (IDS) that uses eBPF programs to instrument system calls and establish normal behavior for processes, building a profile for each executable on the system; subsequently, ebpH can warn the user when it detects process behavior that violates the established profiles. Experimental results show that ebpH can detect anomalies in process behavior with negligible overhead. Furthermore, ebpH's anomaly detection comes with zero risk to the system thanks to the safety guarantees of eBPF, rendering it an ideal solution for monitoring production systems.

This thesis will discuss the design and implementation of ebpH along with the technical challenges which occurred along the way. It will then present experimental data and performance benchmarks that demonstrate ebpH's ability to monitor process behavior with minimal overhead. Finally, it will conclude with a discussion on the merits of eBPF IDS implementations and potential avenues for future work therein.

ebpH is licensed under GPLv2 and full source code is available at https://github.com/willfindlay/ebph.

Acknowledgments

First and foremost, I would like to thank my advisor, Anil Somayaji, for his tireless efforts to ensure the success of this project, as well as for providing the original design for pH along with invaluable advice and ideas. Implementing ebpH and writing this thesis has been a long process and not without its challenges. Dr. Somayaji's support and guidance have been quintessential to the success of this undertaking.

I would also like to thank the members and contributors of the *Iovisor Project*, especially Yonghong Song[1] and Teng Qin[2] for their guidance and willingness to respond to issues and questions related to the *bcc* project. Brendan Gregg's[3] writings and talks have been a great source of inspiration, especially with respect to background research. Sasha Goldstein's[4] *syscount.py* was an invaluable basis for my earliest proof-of-concept experimentation, although none of that original code has made it into this iteration of ebpH.

For their love and tremendous support of my education, I would like to thank my parents, Mark and Terri-Lyn. Without them, I am certain that none of this would have been possible. I would additionally like to thank my mother for suffering through the first draft of this thesis, and finding the many errors that come with writing a paper this large in Vim with no grammar checker.

Finally, I want to thank my dear friend, Amanda, for all the support she has provided me throughout my university career. I couldn't have made it this far without you.

[1] https://github.com/yonghong-song

[2] https://github.com/palmtenor

[3] https://github.com/brendangregg and http://www.brendangregg.com

[4] https://github.com/goldshtn

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 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 [54, 55], 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 nearly 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* (a portmanteau of eBPF and pH), 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 the *BPF Compiler Collection* (*bcc*) [20], 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 Homeostasis* [49], 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.

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 an invidivual userspace process	[58, 59]
`ltrace`	Uses the `ptrace` system call to trace library calls in an individual userland process	[6, 44]
`SystemTap`	Dynamically generates loadable kernel modules for instrumentation; newer versions can optionally use eBPF as a backend instead	[34, 42]
`ftrace`	Sysfs pseudo filesystem for tracepoint instrumentation located at `/sys/kernel/debug/tracing`	[43]
`perf_events`	Linux subsystem that collects performance events and returns them to userspace	[65]
`LTTng`	Loadable kernel modules, userland libraries	[29]
`dtrace4linux`	A Linux port of DTrace via a loadable kernel module	[12]
`sysdig`	Loadable kernel modules for system monitoring; native support for containers	[60]
`eBPF`	In-kernel execution of pre-verified, JIT-compiled bytecode	[13, 20, 54, 55]

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.

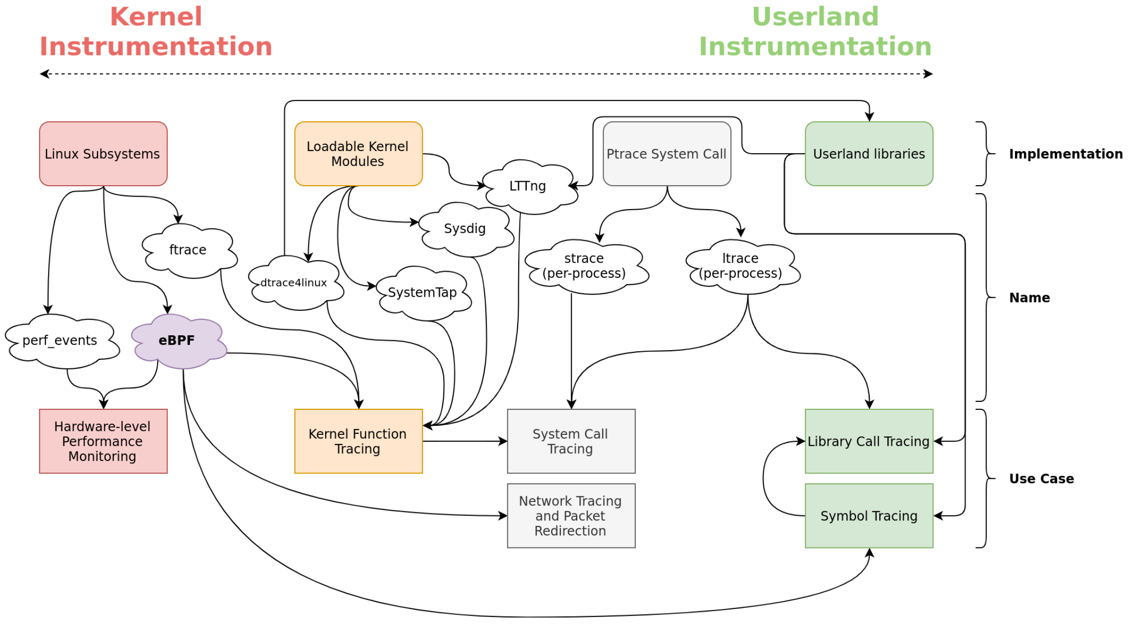

Figure 2.1: A high level overview of the broad categories of Linux instrumentation. This does not represent a complete picture of all available tools and interfaces, but instead presents many of the most popular ones. Note how eBPF covers every presented use case.

Applications such as strace [58, 59] 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 [41]. The scope of ptrace-based solutions is also limited by ptrace's lack of scalability; ptrace's API is conducive to tracing single 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 [25].

Although library call instrumentation through software such as ltrace [6, 44] 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 [60] and SystemTap [42] 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. Additionally, such kernel-module-based solutions are highly reliant on specific versions of the Linux kernel; changes to Linux's API may cause them to break, which in turn requires updates; these updates then increase the risk of introducing bugs into the codebase, which may in turn lead to the aforementioned consequences of code failure in kernelspace.

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 [54, 55] itself constitutes one such solution. However, for the time being, we will focus on a few others, namely ftrace [43] and perf_events [65] (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. The original Dtrace [5] was designed in 2004 for Solaris and lives on to this day in several operating systems, including Solaris, FreeBSD, MacOS X [17], and Linux [12] (we will examine the dtrace4linux implementation with more scrutiny later in this section).

In general, the original Dtrace and the current version of eBPF share much of the same family and cover similar use cases [5, 54, 55]. This includes perhaps most notably dynamic instrumentation in both userspace and kernelspace, arbitrary context instrumentation (i.e. the ability to instrument essentially any aspect of the system), and guarantees of safety and data integrity. The difference between the two systems generally boils down to the following points [16, 17]:

1) eBPF supports a superset of Dtrace's functionality;
2) Dtrace provides only a high level interface, while eBPF provides both low level and high level interfaces (see Figure 2.2);
3) Dtrace is useful for writing one-liner scripts, but not for writing more complex programs;
4) eBPF is natively supported in Linux, while Dtrace ports are purely LKM-based.

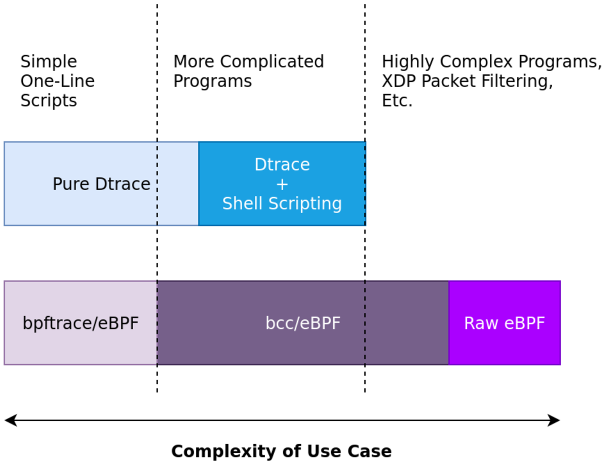

Figure 2.2: Comparing Dtrace and eBPF functionality with respect to API design (adapted from [15]). Note that eBPF covers more complex use cases and supports both low level and high level APIs. Dtrace needs to be used in tandem with shell scripting to support more complex use cases.

dtrace4linux [12] is a free and open source port of Sun's Dtrace [5] 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 [15] 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 [20], 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 [15, 21] 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 [15], 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 [64] 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 higher level advantages of Dtrace with the existing eBPF virtual machine.

2.2 Classic BPF

In 1992, McCanne and Jacobson [30] introduced the original BPF[5] or *Berkeley Packet Filter* as a mechanism for capturing, monitoring, and filtering network traffic in the BSD kernel. Classic BPF's primary insights were two-fold:

1) network traffic events are *frequent* and *fast*, and therefore an efficient filtering mechanism was needed;
2) a limited, register-based bytecode being run in an in-kernel virtual machine provides precisely the mechanism described in point (1).

The virtual machine described above was used to implement the *filter* component of BPF, while in-kernel network function tracepoints implemented the *tap* component. The tap forwarded packet events to the filter, which then decided what to do with the packets according to a user-defined BPF program. McCanne and Jacobson showed that their approach was much faster than other contemporary packet filtering techniques, namely NIT [38] and CSPF [35].

While Classic BPF is certainly a powerful technique for filtering packets, Starovoitov [54, 55] realized that its tap and filter mechanism represented a desirable approach for general system introspection. Therefore, in 2013, he proposed *Extended BPF* (eBPF), a superset of Classic BPF, which vastly increased the capabilities of the original BPF virtual machine.

Since its original introduction, eBPF has offered a consistent, powerful, and production-safe mechanism for general Linux introspection and continues to improve rapidly over time. We

[5] Hereafter, we will refer to the original BPF as *Classic BPF* to avoid confusion with eBPF and the BPF programming paradigm.

discuss eBPF in more detail in the following section.

2.3 eBPF: Linux Tracing Superpowers

In 2016, eBPF was described by Brendan Gregg [14] 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* [18], we can inspect, modify, redirect, and even drop packets entirely before they even reach the main kernel network stack. Figure 2.3 provides a high level overview of these use cases and the corresponding eBPF instrumentation required.

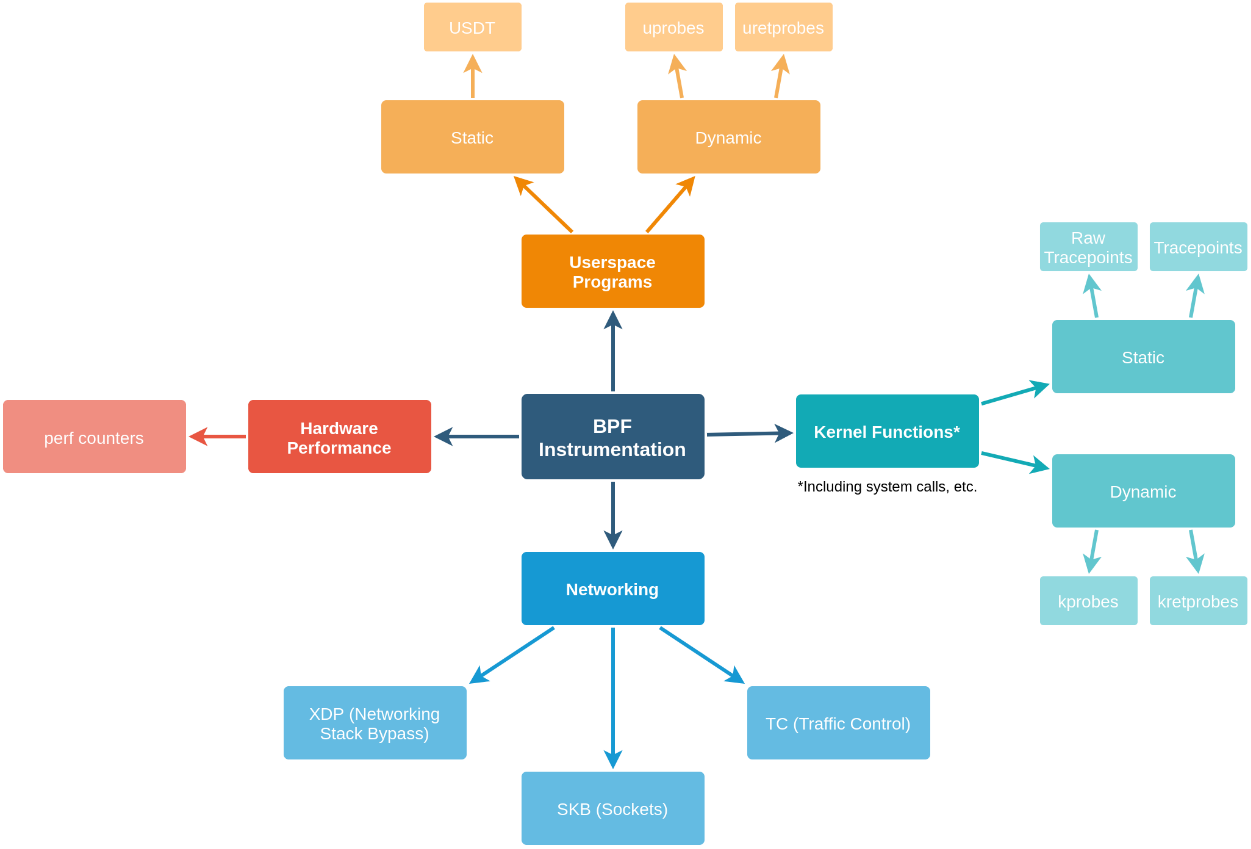

Figure 2.3: A high level overview of various eBPF use cases. Note the high level of flexibility that eBPF provides with respect to system tracing.

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 [57] 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 Section 2.3.2.

eBPF's superior performance can be attributed to several factors. On supported architectures,[6] 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 direct 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.

2.3.1 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. IOVisor's bcc [20] offers bindings for several languages including Python, Go, and 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 2.4 presents an overview of eBPF's architecture and dataflow, including the interaction between userspace programs, eBPF programs in kernelspace, and the rest of the kernel. This interaction occurs via the `bpf(2)` system call [4] which is used to load and verify BPF programs, issue commands to BPF programs, and interact with eBPF maps. These maps are the mechanism for sending data between BPF programs and other BPF programs or BPF programs and userspace.

There are several map types available in eBPF which cover a wide variety of use cases. These map types along with a brief description are provided in Table 2.2. Thanks to this wide arsenal of maps, eBPF developers have a powerful set of both general-purpose and specialized data structures at their disposal; as we will see in coming sections, many of these maps are quite versatile and have use cases beyond what might initially seem pertinent. For example, the `ARRAY` map type may be used to initialize large data structures to be copied into

[6] x86-64, SPARC, PowerPC, ARM, arm64, MIPS, and s390 [10]

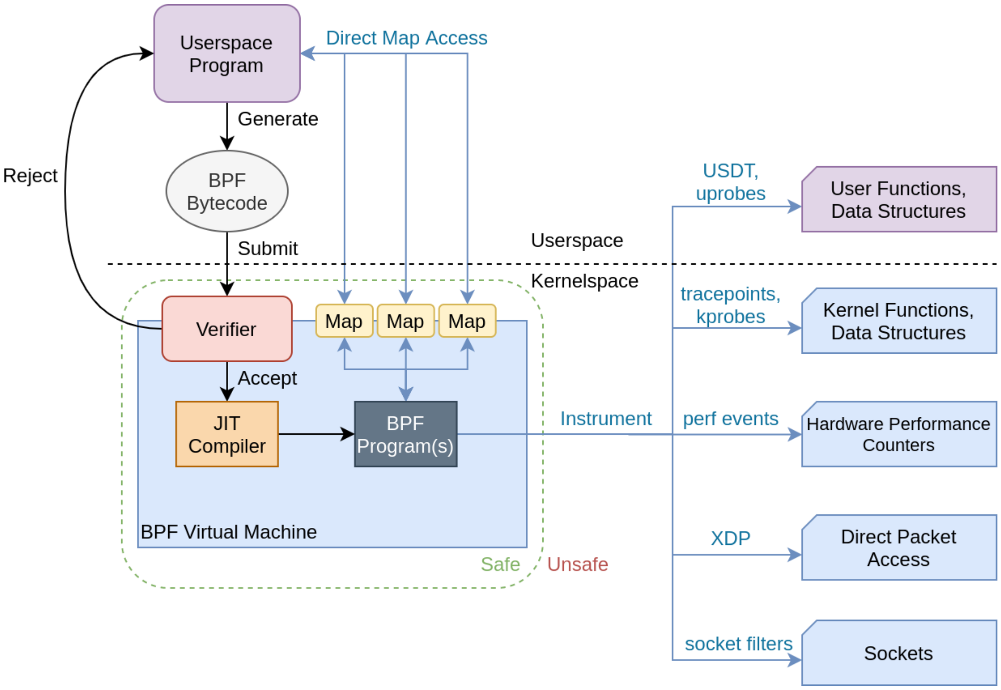

Figure 2.4: Basic topology of eBPF with respect to userland and the kernel. Note the bidirectional nature of dataflow between userspace and kernelspace using maps.

Table 2.2: Various map types [16, 20] available in eBPF programs, as of Linux 5.3.

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

a general purpose HASH map (refer to Listing A.1 in Appendix A). This can be effectively used to bypass the verifier's stack space limitations, which are discussed in detail in Section 2.3.2.

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

The verifier is responsible for eBPF's unprecedented safety, one of its most attractive qualities with respect to system tracing. While this verifier is quintessential to the safety of eBPF given its impressive scope and power, it is not without its drawbacks. In this section, I describe how the verifier works, its nuances and drawbacks, and recent work that has been done to improve the verifier's support for increasingly complex eBPF programs.

Proving the safety of arbitrary code is by definition a difficult problem. This is thanks in part to theoretical limitations on what we can actually prove; a famous example is the halting problem described by Turing circa 1937 [63]. This difficulty is further compounded by stricter requirements for safety in the context of an eBPF program; in particular, we don't want BPF programs to crash or otherwise damage the kernel.

To illustrate the importance of this problem of safety with respect to eBPF, let us consider a simple example. We will again consider the halting problem described above. Suppose we have two eBPF programs, program A and program B, that each hook onto a mission-critical kernel function (`schedule()`, for example). The only difference between these two programs is that program A always terminates, while program B runs forever without stopping. What this means in practice is that the call to `schedule()` will never succeed, and program B effectively constitutes a denial of service attack [19] on our system, intentional or otherwise; allowing untrusted users to load this code into our kernel spells immediate disaster for our system.

By the same token, unbounded memory access attempts within a BPF program may permit buffer overflows, which may in turn be manipulated to gain arbitrary code execution in kernelspace [7] (the kind that actually *can* damage the system). In order to aid static analysis of memory access, the verifier prohibits memory access using registers with unbounded values. For example, accessing an array with induction variable i in a `for` loop would be prohibited unless it could be shown that this variable's set of possible values exists within a memory-safe range.

While we have established that verifying the safety of eBPF programs is an important problem to solve, the question remains as to whether it is *possible* to solve. For the reasons outlined above, this problem should intuitively seem impossible, or at least far more difficult than should be feasible. So, what can we do? The answer is to *change the rules* to make it easier. In particular, while it is difficult to prove that the set of all possible eBPF programs are safe, it is much easier[7] to prove this property for a subset of all eBPF programs. Figure 2.5 depicts the relationship between potentially valid eBPF code and verifiably valid eBPF code.

The immediate exclusion of eBPF programs meeting certain criteria is the crux of eBPF's safety guarantees. Unfortunately, it also rather intuitively limits what we are actually able to do with eBPF programs. In particular, eBPF is not a Turing complete language; it prohibits arbitrary jump instructions, cycles in execution graphs, and unverified memory access. Further, we limit stack allocations to only 512 bytes – far too small for many practical use cases. From a security perspective, these limitations are a *good thing*, because they allow us to immediately exclude eBPF programs with unverifiable safety; but from a usability standpoint, particularly that of a new eBPF developer, the trade-off is not without its drawbacks.

Fortunately, the eBPF verifier is getting better over time (Figure 2.6). When we say *better*, what we mean is that it is able to prove the safety of increasingly complex programs. Perhaps the best example of this steady improvement is a recent kernel patch [57] that added support for bounded loops in eBPF programs. With this patch, the set of viable eBPF programs was *greatly* increased; in fact, ebpH in its current incarnation relies heavily on bounded loop support. Prior to bounded loops, eBPF programs relied on *unrolling* loops at compile time, a technique that was both slow and highly limited. This is just one example of the critical work that is being done to improve the verifier and thus improve eBPF as a whole.

2.4 System Calls

ebpH (and the original pH system upon which it is based) works by instrumenting *system calls* in order to establish behavioral patterns for all binaries running on the system. Understanding pH and ebpH requires a reliable mental model of what a system call is and how programs use them to communicate with the kernel.

At the time of writing this thesis, the Linux Kernel [62] supports an impressive 439 distinct system calls, and this number generally grows with subsequent releases. In general, userspace libraries such as the C standard library implement a subset of these system calls, with the exact specifications varying depending on architecture. These system calls are used to request services from the operating system kernel; for example, a program that needs to write to a file would make an `open` call to receive a file descriptor into that file, followed by one or more `write` calls to write the necessary data, and finally a `close` call to clean up the file

[7] *Easier* here means *computationally easier*, certainly not trivial.

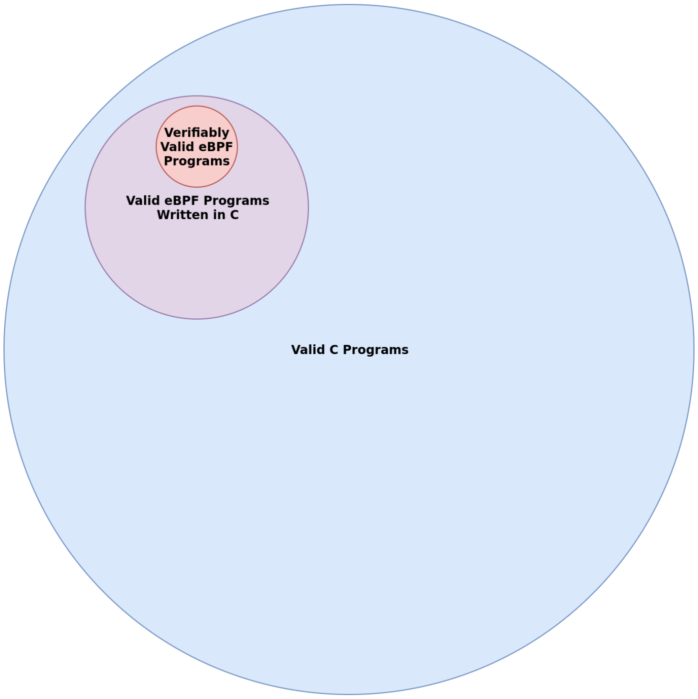
Verifiably Valid eBPF Programs
Valid eBPF Programs Written in C
Valid C Programs

Figure 2.5: The set participation of valid C and eBPF programs. Valid eBPF programs written in C constitute a small subset of all valid C programs. Verifiably valid eBPF programs constitute an even smaller subset therein.

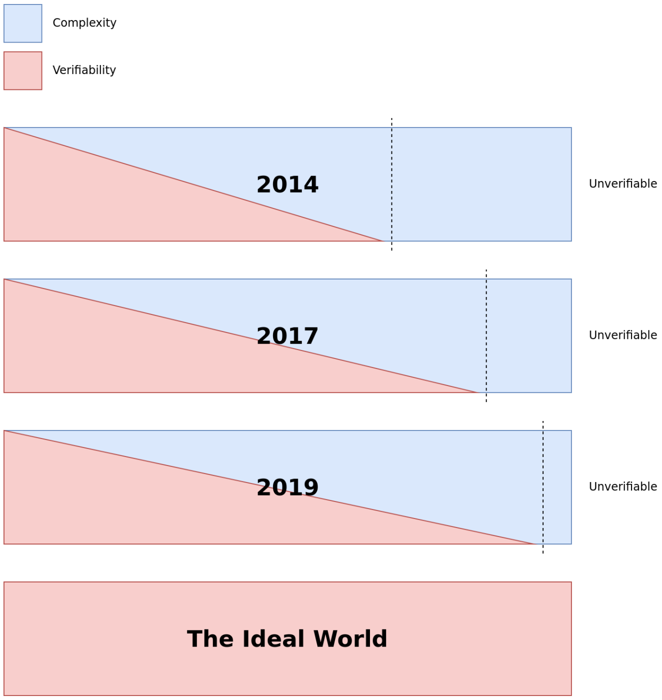

Figure 2.6: Complexity and verifiability of eBPF programs. Safety guarantees for eBPF programs rely on certain compromises. Ideally we would like to have a relationship as shown on the bottom; in practice, we have something that is getting closer over time, but is still far from the ideal.

descriptor. These system calls form the basis for much of our process behavior, from I/O as seen above, to process management, memory management, and even the execution of binaries themselves.

Critically, from a security perspective, system calls provide the interface to the kernel's *reference monitor* [2, 23, 39], an abstraction that refers to the kernel's facilities for mediating access by subjects (i.e. users and their processes) onto system objects (i.e. security-sensitive resources). This means that system calls provide a highly representative picture of a given process' attempts to access resources that we care about – whether this access is valid or otherwise.

Through the instrumentation of system calls, we can establish a clear outline of exactly how a process is behaving, the critical operations it needs to perform, and how these operations interact with one another. In fact, system call-based instrumentation forms a primary use case for several of the tracing technologies previously discussed in Section 2.1, perhaps most notably strace. We will discuss the behavioral implications of system calls further in Section 2.6.1.

2.5 Intrusion Detection

At a high level, intrusion detection systems (IDS) strive to monitor systems at a particular level and use observed data to make decisions about the legitimacy of these observations [24]. Intrusion detection systems can be broadly classified into several categories based on data collection, detection technique(s), and response. Figure 2.7 presents a broad and incomplete overview of these categories.

In general, intrusion detection systems can either attempt to detect anomalies (i.e. mismatches in behavior when compared to normally observed patterns) or misuse-based, which generally refers to matching known attack patterns to observed data [24, 39]. In general, anomaly-based approaches cover a wider variety of attacks while misuse-based approaches tend to yield fewer false positives. Misuse-based approaches can also be further broken down into specification-based and signature-based, which deal in behavioral blacklists and whitelists respectively. A hybrid approach between any of these techniques is also possible.

Data collection is generally either host-based or network based. Network-based IDSes examine network traffic and analyze it to detect attacks or anomalies. In contrast, host-based IDSes analyze the state of the local system [24, 49].

Responses can vary significantly based on the system, but can be classified into two main

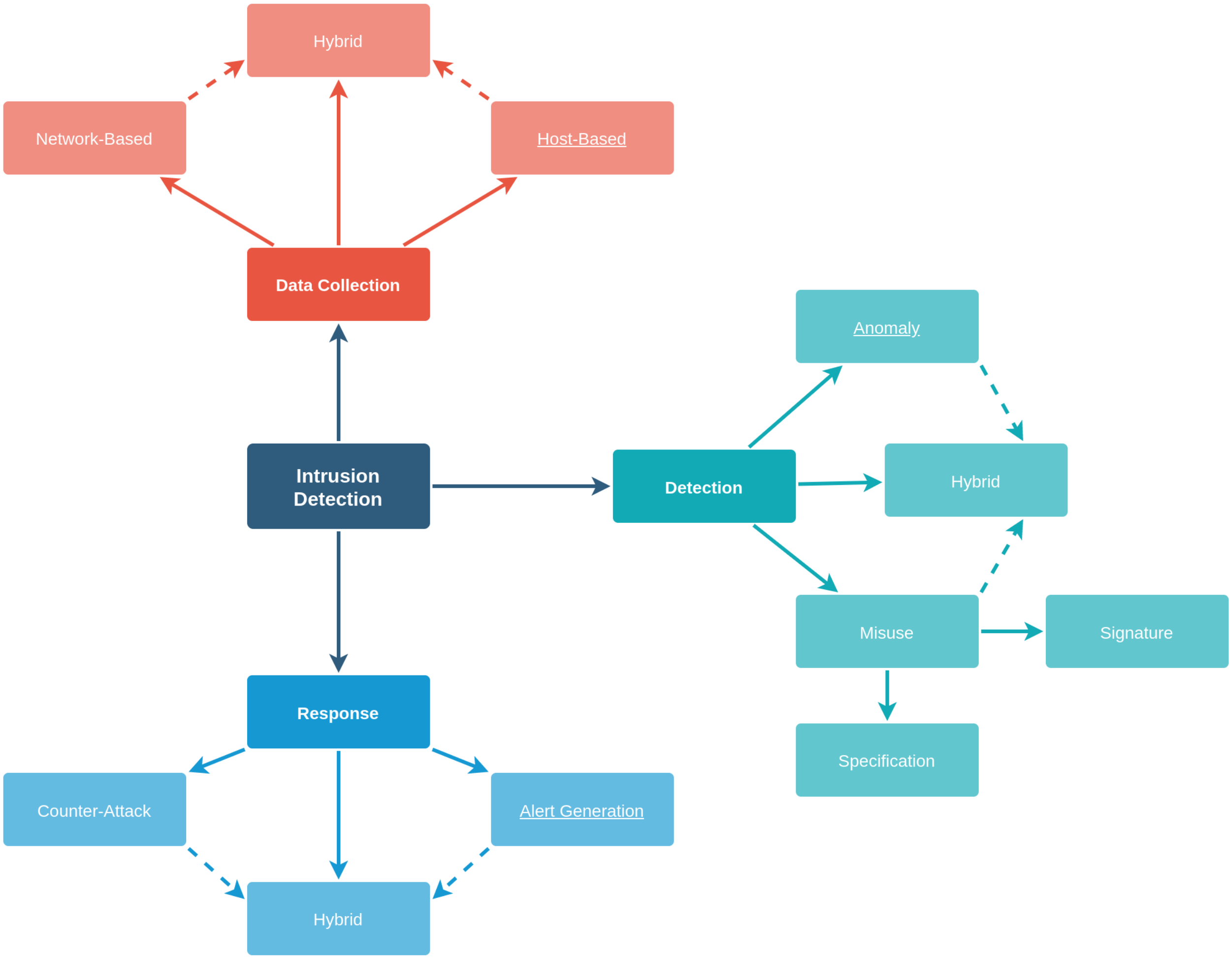

Figure 2.7: A broad overview of the basic categories of IDS. The current version of ebpH can be classified according to the categories that have been underlined. Note that intrusion detection system classification can often be more nuanced than the basic overview presented here. However, this should present a good enough overview to understand IDSes in the context of ebpH.

categories: alerts and counter-attacks. Systems can either choose to alert an administrator about the potential issue, or choose to mount counter-measures to defeat or mitigate the perceived attack [24]. Naturally, systems also have the option to take a hybrid approach here.

Using the above metrics, ebpH can be broadly classified as a host-based anomaly detection system that responds to anomalies by issuing alerts. This is generally quite similar to the original pH (Section 2.6) with one major exception: As we will see, the original pH also responds to anomalies by delaying system calls outright and preventing anomalous `execve(2)` calls [49]. Implementing this functionality in ebpH is a topic for future work (c.f. Section 5.2.2).

2.5.1 A Survey of Data Collection in Intrusion Detection Systems

We have presented the general classification of intrusion detection systems through the establishment of three core elements of an IDS and several categories therein. As it relates to eBPF, the *data collection* component of intrusion detection systems is of particular interest; what is especially exciting about eBPF is its impressive scope, safety, and performance with respect to general system introspection; this presents a perfect trifecta of traits for collecting system data. As such, it is worth examining data collection techniques from various intrusion detection systems in more detail.

We have established that data collection in intrusion detection systems can primarily be separated into two relatively broad categories:

1) host-based data collection which involves collecting data about the use and behavior of a local machine;
2) network-based data collection which involves monitoring network traffic and looking for established patterns.

While the above two categories are generally sufficient for understanding intrusion detection at a high level, there are in fact several distinct subcategories therein. Figure 2.8 presents an overview of the most common data collection subcategories in IDSes.

Internal and External Sensors. Kerschbaum et al. [26] introduce the concept of *internal* sensors for intrusion detection and contrast them with the far more popular *external* sensors. An internal sensor by definition is included in the source code of a monitored component, while an external sensor is implemented outside of the monitored component. These two categories of sensors each present unique advantages and disadvantages [26]. In

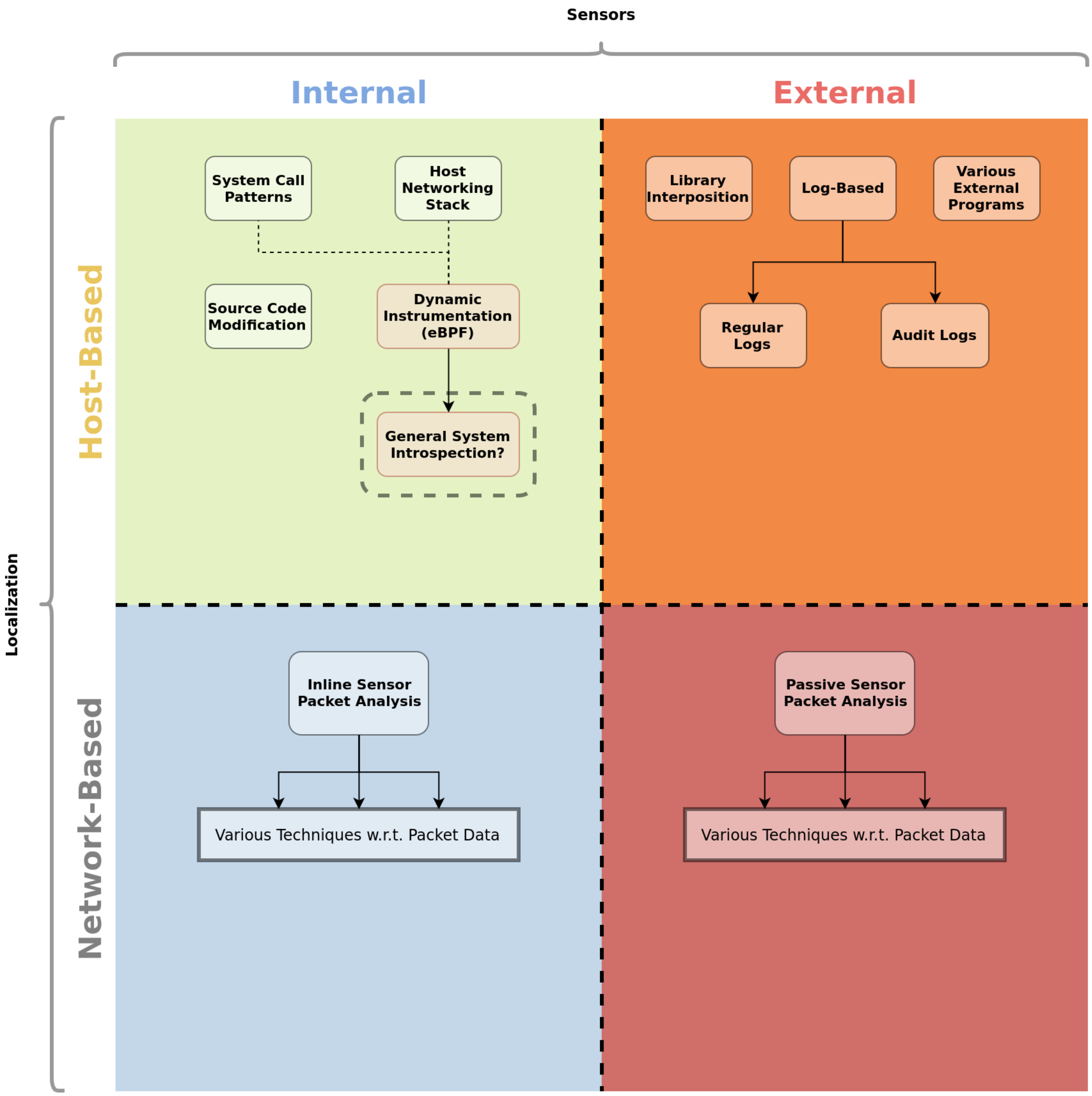
Sensors
Internal
External
Localization
Host-Based
Network-Based
System Call Patterns
Host Networking Stack
Source Code Modification
Dynamic Instrumentation (eBPF)
General System Introspection?
Library Interposition
Log-Based
Various External Programs
Regular Logs
Audit Logs
Inline Sensor Packet Analysis
Various Techniques w.r.t. Packet Data
Passive Sensor Packet Analysis
Various Techniques w.r.t. Packet Data

Figure 2.8: An overview of the most common data collection categories and subcategories in IDS, as well as a potentially new and promising category, *general system introspection*, thanks to eBPF. This figure primarily synthesizes the technologies presented in [26, 53].

particular, external sensors are easily modifiable and extensible, although they introduce potential delays, and are generally weaker to tampering by intruders; internal sensors minimize overhead (assuming correct implementation) and are much more resistant to intruder tampering, but suffer from reduced portability, difficulty in implementation, and may incur severe performance penalties if implemented incorrectly.

eBPF would fall under the internal sensor classification [26] due to its implementation within the Linux Kernel; however, eBPF presents a rather unique case, as it overcomes many of the disadvantages proposed by Kerschbaum et al. while maintaining the advantages. Specifically, eBPF is completely application transparent, portable to any modern version of Linux[8], easy to update and modify, and has guaranteed performance and safety due to the verifier.

Internal Host-Based Approaches. System call pattern analysis was examined in detail by Forrest et al. [11] and culminated in the development of the original pH system [49] on which ebpH is based. Somayaji and Inoue [50] compared two methods of organizing system call data for intrusion detection (full sequences and lookahead pairs), which we will discuss further in Section 2.6.1.

Kerschbaum et al. also describe a generic method of application-specific internal sensors through the addition of audit checks in program source code [26]. However, the primary caveat here is that such checks need to be integrated into a specific application early enough such that refactoring is minimized [26, 40]. This approach is also far less generic than other internal sensor approaches described here.

Another potential internal source for data is through host-based network stack introspection. Classic BPF [30] and eBPF/XDP [18, 20, 54, 55] are quite excellent at this. Host-based network introspection allows the analysis of network traffic at various points in the kernel's networking stack, and XDP packet redirection [18] allows fast detection and response before a packet even reaches the main networking stack.

ebpH itself constitutes an internal host-based approach; that is, it uses eBPF for in-kernel instrumentation of system calls (internal) on a given host (host-based). As we discuss in Section 5.2.6, a potential avenue for future research in ebpH is moving beyond system call monitoring to *general system introspection* (c.f. Figure 2.8). This is specifically a possibility due to eBPF's unique classification as an internal sensor capable of monitoring the entire system dynamically, safely, and with minimal overhead.

[8] Although eBPF is available on all modern kernels, some of its features are specific to the very newest versions. In particular, recent verifier updates which allow for increased complexity have only been available since version 5.3. See Section 2.3 and Section 2.3.2 for more details.

External Host-Based Approaches. External host-based data collection is very popular in intrusion detection. This can be primarily attributed due to the advantages described by Kerschbaum et al. [26], particularly with respect to ease of implementation and portability.

AAFID [52] uses a *combined* internal/external approach based on separate autonomous agents running continuously on multiple hosts. These agents make use of various data sources, such as external programs (i.e. `ps`, `netstat`, and `df`), file system state, and network interface packet capture (i.e. hooking into the host's networking stack). Agents supplement collected data by analyzing audit logs generated by the system [26].

In 1999, Kuperman and Spafford [27] proposed the use of library interpolation for intrusion detection in dynamically linked binaries. Library interpolation is a method of interposing a custom library implementation between a dynamically linked executable and its shared objects. This effectively allows the generation of custom audit data on each library call that a process makes.

Internal and External Network-Based Approaches. Network-based approaches [53] to intrusion detection involve the inspection of network traffic en route to its destination. This typically comes in the form of inspecting packets headers, payloads, and frequency to establish patterns for analysis. Generally, network-based approaches have a choice between using either inline (internal) sensors, or passive (external) sensors for data collection [53]. An inline sensor either hooks into a network device, or is built into specialized hardware; traffic passes directly through the sensor and is analyzed directly.

In contrast, passive sensors create copies of network traffic for analysis. This approach is typically favored since it does not introduce delays into the traffic itself, instead sacrificing the ability to respond to threats before they reach their destination [53]. This result is consistent with Kerschbaum et al.'s observation that external sensor approaches tend to be favored over their internal counterparts [26].

2.6 Process Homeostasis

Anil Somayaji's *Process Homeostasis* [49], styled as *pH*, forms the basis for ebpH's core design; as such, it is worth exploring the original implementation, design choices, and rationale therein. Using the same IDS categories from the previous section, we can classify pH as a host-based anomaly detection system that responds by both issuing alerts *and* mounting countermeasures to reduce the impact of anomalies; in particular pH responds to anomalies by injecting delays into a process' system calls proportionally to the number of recent anomalies that have been observed [49]. It is in this way that pH lives up to its name: these delays make process behavior *homeostatic*.

2.6.1 Anomaly Detection Through Lookahead Pairs

pH uses a technique known as *lookahead pairs* [49, 50] for detecting anomalies in system call data. This is in stark contrast to other anomaly detection systems at the time that primarily relied on *full sequence analysis*. Here we describe lookahead pairs, their use for anomaly detection, and offer a comparison with the more widely-known full sequence analysis.

In order to identify normal process behavior, profiles are constructed for each executable on the system. On calls to `execve`, pH associates the correct profile with a process and begins monitoring its system calls, modifying the lookahead pairs associated with the testing data of a profile. Once enough normal samples have been gathered and the profile has reached a specified maturity date, the process is then placed into training mode wherein sequences of system calls are compared with the existing lookahead pairs for a given profile.

Somayaji and Inoue [50] contrasted full sequence analysis with lookahead pairs and found that lookahead pairs produce fewer false positives than full sequences and maintain this property even with very long window lengths. This comes at the expense of potentially reduced sensitivity to some attacks as well as more vulnerable to mimicry attacks. However, as part of their work, Somayaji and Inoue showed that longer sequences can help mitigate these shortcomings in practice [50].

Both pH and ebpH use lookahead pair window sizes of 9, which has been shown to be effective at both minimizing false positive rates and mitigating potential mimicry attacks [49]. This window size also carries the advantage that lookahead pairs can be expressed with exactly 8 bits of information (one bit for every previous position $i \in \{1..9\}$).

2.6.2 Homeostasis Through System Call Delays

Perhaps the most unique aspect of pH's approach is the means by which it achieves the eponymous concept of *process homeostasis*: system call delays. Inspired by the biological process of the same name, pH attempts to maintain homeostatic process behavior by injecting delays into system calls that are detected as being anomalous [49].

By scaling this response in proportion to the number of recent anomalies detected in a profile, pH is able to effectively mitigate attacks while minimizing the impact of occasional false positives. For example, a process that triggers several dozen anomalies will be slowed down to the point of effectively stopping, while a process that triggers only one or two might only be delayed by a few seconds. Admittedly, this relies upon the assumption of low burstiness for false positives. While this assumption generally holds, Somayaji acknowledges in his dissertation [49] that the possibility of attackers purposely provoking pH into causing denial of service attacks is a potential problem. Additionally, users may become frustrated with pH's refusal to allow otherwise legitimate behavior simply due to the fact that it has not yet been observed.

In its current incarnation, ebpH does not yet delay system calls like its predecessor. The primary reason for this gap in functionality is that a solution still needs to be developed that works well with the eBPF paradigm; in particular, injecting delays via eBPF tracepoints or probes seems untenable due to the verifier's refusal to accommodate the code required for such an implementation. The addition of system call delays into ebpH is currently a topic for future work (c.f. Section 5.2.2).

3 Implementing ebpH

At a high level, ebpH is an anomaly detection system that profiles executable behavior by sequencing the system calls that processes make to the kernel; this essentially serves as a reimplementation of the original pH system described by Somayaji [49]. What makes ebpH unique is its use of BPF programs for system call instrumentation and profiling (in contrast to the original pH which was implemented as a Linux 2.2 kernel patch).

3.1 Why an eBPF Implementation?

In light of the various approaches presented in Section 2, it is worth comparing the approach taken by the original pH [49] system with the new ebpH prototype. In doing so, I will attempt to justify why an eBPF implementation of a system like pH makes sense, and why such an implementation carries key advantages that would not otherwise be tenable through traditional kernel-based implementations. To begin with, let us compare the rough features of pH with ebpH; Table 3.1 provides a rough framework for doing so.

As discussed in previous sections (see Section 2.3 and Section 2.5.1), eBPF offers several unique advantages over traditional solutions, particularly with respect to intrusion detection data collection. eBPF can match the scope of kernel-based implementations while providing safety guarantees that previously would not have been possible. This essentially caused an implicit trade-off between production safety and scope in IDS implementations. Furthermore, eBPF's forward-compatibility ensures that new versions of the kernel will not break old code,

Table 3.1: Comparing the current prototype of ebpH with the original pH system.

System	Implementation	Portable	Production Safe	Low Mem. Overhead	Low Perf. Overhead	Detection	Response
`pH` [49]	Kernel Patch	✗	✗	✓	✓	✓	✓
`ebpH`	eBPF + Userspace Daemon	✓	✓	✗	✓	✓	✗

and that in general it is not necessary to upgrade to a new kernel version once one has access to the minimum set of features required to compile and run a given BPF program. For instance, since ebpH depends on Linux 5.3, all Linux kernel versions ≥ 5.3 will be able to support ebpH's current set of features. This ensures perfect forward compatibility with production systems and minimizes the impact of integrating ebpH into a production security stack.

The primary disadvantage of using eBPF is that BPF programs are necessarily more limited in scope than kernel modules. That is not to say that BPF programs cannot be complex, but rather that constructs that work well in kernel implementations often need to be reworked for use in eBPF. A good example of this is the inability for ebpH to issue system call delays in the same manner as the original pH. This is something that remains a topic for future work, but there are alternative ways that it can be done, for example the method using `bpf_signal` described in Section 5.2.2.

Since eBPF disallows global variables in the traditional sense, data storage and communication between BPF programs needs to occur through the variety of maps. Further, limitations on memory allocation and access restrict the dynamic allocation of data. To cope with these restrictions, the current version of ebpH takes a less memory-efficient approach than its predecessor; in particular, the sparse array of lookahead pairs is not dynamically allocated – instead, profiles themselves are dynamically allocated at runtime via a special hashmap. There are plans to rework the way ebpH stores profile data to move it into separate `map-in-map` structures that should allow a more memory-efficient approach to lookahead pair storage. This is outlined in more detail in Section 5.2.4.

In summary, despite a few shortcomings of an eBPF implementation compared to a kernel implementation, the benefits of portability and production safety in general outweigh these detractors. Additionally, the problems with the current ebpH implementation *are* solvable in eBPF, and future versions of the system should be significantly more memory efficient and offer the capability to respond to attacks in real time, just as in the original pH [49].

3.2 Architectural Overview

ebpH can be thought of as a combination of several distinct components, functioning in both userspace and kernelspace. In particular it includes a daemon and several CLI programs (described in Section 3.3) in userspace as well as several BPF programs in kernelspace (described in Section 3.4 and onwards). The userspace components interact with each other through sending requests and replies over a UNIX domain socket and the daemon interacts with the BPF program via direct access to hashmaps and polling perf event buffers. The architecture of ebpH is depicted in Figure 3.1. This section will present the design and implementation of ebpH, with a particular emphasis on both the similarities and differences between ebpH and the original pH [49].

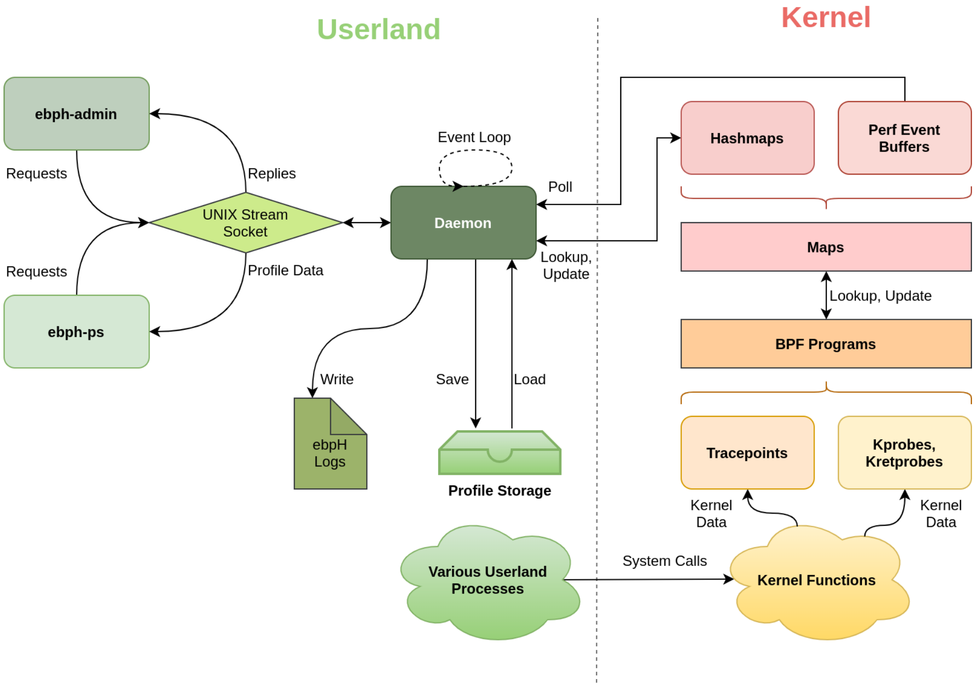

Figure 3.1: The architecture of ebpH. Note how the interaction between userspace programs and BPF programs is centered around the ebpH daemon. `ebph-admin` and `ebph-ps` are used to issue commands to and query info from the daemon, which interacts with the BPF programs on their behalf. The BPF programs instrument various kernel functions which are triggered by system calls from userspace.

3.3 Userspace Components

The userspace components of ebpH are comprised of several distinct and related programs. In particular, these programs can be divided into two sets: the ebpH daemon (`ebphd`) and several CLI (command line interface) programs used to interact with it. The daemon is responsible for submitting BPF programs to the kernel, managing their state, and providing an API to other userspace programs. The CLI programs used to interact with the daemon include `ebph-ps`, used to list actively traced processes, threads, and profiles, providing information about each, and `ebph-admin`, used to issue commands to the daemon and to check the status of the BPF program. In order to issue more complex commands to the BPF program, `ebphd` leverages a userspace shared library, `libebph.so` which provides functions that can be connected to arbitrary BPF programs via `uprobes`. Earlier versions of ebpH also included a GUI, however the GUI needs to be refactored in order to work with ebpH's new architecture and this is currently a topic for future work.

3.3.1 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 [20] 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 3.2 presents an overview of the most important events that ebpH cares 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 in this way significantly reduces the number of required context switches between kernelspace and userspace.

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 also uses direct map access to save and load profiles to and from the disk. Profiles are saved automatically at regular intervals, configurable by the user, as well as any time ebpH stops monitoring the system. These profiles are automatically loaded every time ebpH starts.

Table 3.2: Main perf event categories in ebpH.

Event	Description	Memory Overhead[9]
`ebpH_on_executable_processed`	Reports when a profile has been created	2^8 pages
`ebpH_on_anomaly`	Reports anomalies in specific processes and which profile they were associated with	2^8 pages
`ebpH_on_anomaly_limit`	Reports when a profile hits its anomaly limit	2^8 pages
`ebpH_on_tolerize_limit`	Reports when a process hits its tolerize limit	2^8 pages
`ebpH_on_start_normal`	Reports when a profile starts normal monitoring	2^8 pages
`ebpH_on_new_sequence`	Reports new sequences for logging (when enabled)	2^8 pages
`ebpH_warning`	Reports generic warnings	2^2 pages
`ebpH_error`	Reports generic errors	2^2 pages

In order to facilitate communication with the daemon, `ebphd` exposes a UNIX domain stream socket at `/var/run/ebph.sock`. This socket is owned by the superuser, `root`, and has permissions `600` in order to prevent unauthorized processes from attempting to issue commands to the daemon. The CLI applications, `ebph-ps` (c.f. Section 3.3.2) and `ebph-admin` (c.f. Section 3.3.3), use this socket to send commands to and receive replies from the daemon.

3.3.2 `ebph-ps`

`ebph-ps` is the most common tool that a system administrator will use to get a quick overview of process state on their system with respect to ebpH profiles. When run with its default settings, `ebph-ps` lists all currently monitored processes on the system with their PID, comm, current status (e.g. training, frozen, or normal), total system call count, system calls since last modification, anomaly count, and normal time. When the user invokes `ebph-ps`, it sends a JSON-encoded request to the daemon via the `ebphd`'s UNIX domain stream socket. The daemon replies on that same socket with a JSON-encoded list of processes or profiles. Users who are acquainted with the popular `ps` command line utility will find `ebph-ps`'s interface quite familiar. Listing 3.1 shows sample output from `ebph-ps` running on a system.

Listing 3.1: Sample output from `ebph-ps`.

```
PID  COMM             STATUS   COUNT     LAST_MOD ANOMALIES NORMAL TIME
727  lightdm          Training 58177     1543     0         2020-02-23 16:13:54
739  Xorg             Training 80663410  1115643  0         2020-02-23 16:13:53
742  accounts-daemon  Training 172644    22043    0         2020-02-26 20:18:40
747  polkitd          Training 461714    4454     0         2020-03-03 15:00:18
799  lightdm          Training 58177     1543     0         2020-02-23 16:13:54
817  systemd          Training 93688     8133     0         2020-03-03 15:00:19
824  i3               Training 5043705   59416    0         2020-03-03 15:21:55
831  dbus-daemon      Training 61979     3620     0         2020-02-23 16:13:54
835  redshift         Training 366557    136577   0         2020-02-26 20:18:57
864  polybar          Training 16085886  14860    0         2020-02-26 20:48:56
866  xbindkeys        Training 12890     1227     0         2020-02-23 16:13:59
868  volnoti          Training 21305     3329     0         2020-02-23 16:13:59
872  pulseaudio       Training 16869390  1292     0         2020-02-23 16:13:59
873  rtkit-daemon     Training 104315    23351    0         2020-02-26 20:18:58
878  gsettings-hel... Training 9924      124      0         2020-02-23 16:14:00
885  alacritty        Training 47085184  10249    0         2020-03-03 15:21:55
908  zsh              Training 21847582  194246   0         2020-03-03 15:21:56
917  python3.8        Training 28569801  4254     0         2020-02-21 15:01:42
1071 sudo             Training 3640      2407     0         2020-04-01 10:09:15
1072 python3.8        Training 28569801  4254     0         2020-02-21 15:01:42
```

In addition to listing information per-process, `ebph-ps` can also show information per-thread using an optional `-t` flag. This can be used to get an idea of the number of tasks that ebpH is *actually* monitoring (since ebpH's view of a "process" is actually an individual thread rather than the entire thread group). Further, the `-p` flag can be specified to list all profiles on the system instead of processes. This can be used to find duplicate profiles for pruning, find the key associated with a given profile, or get an idea of the overall behavior of all binaries on the system. Listing 3.2 shows a truncated example of listing all profiles on a system using the `-p` flag.

Listing 3.2: Sample output from `ebph-ps -p`. Note how the `PID` column has been replaced with the profile `KEY` and `ebph-ps` now lists each profile exactly once, regardless of whether the profile is currently running.

```
KEY     COMM             STATUS   TRAIN_COUNT LAST_MOD ANOMALIES NORMAL TIME
5259631 systemd-sleep    Normal   19039       0        0         2020-03-03 17:50:58
5259625 systemd-sleep    Training 1274        422      32        2020-03-08 17:51:52
5259628 systemd-sysctl   Training 554         74       0         2020-02-25 18:41:25
5259377 systemd-tmpfi... Training 29199       5014     0         2020-02-26 20:34:00
5259378 systemd-tty-a... Normal   17348       17348    0         2020-03-03 15:44:19
5259635 systemd-user-... Training 5786        1529     0         2020-02-23 16:13:54
5259642 systemd-user-... Normal   1452        1452     0         2020-03-15 09:29:04
5259636 systemd-user-... Training 1630        405      0         2020-02-26 20:18:40
5259643 systemd-user-... Normal   409         409      0         2020-03-15 09:29:39
5255864 tail             Training 137946      1326     0         2020-03-03 19:14:51
5264292 tbl              Training 3200        2081     0         2020-02-21 15:50:44
5255865 tee              Training 1616        323      0         2020-02-28 22:16:31
```

```
      5255866 test             Normal   75          75       0         2020-03-09 17:25:21
      5380461 thunderbird      Training 1906232     1        0         2020-02-26 21:07:36
      5275208 thunderbird      Training 427         237      0         2020-02-26 21:07:35
      5275250 tmux             Training 1334866     36988    0         2020-02-23 16:02:25
      5255868 touch            Training 1749        837      0         2020-03-03 16:54:23
      5247138 tput             Training 300534      147106   32        2020-02-21 15:03:37
```

3.3.3 ebph-admin

ebph-admin is responsible for issuing more complex commands to ebpH, as well as making generic queries about ebpH's status. Status queries include information about whether ebpH is currently monitoring, how many system calls it has observed, how many process and threads are currently being monitored, and how many profiles are loaded in memory.

Complex commands are issued via libebph.so, a dynamic library written in C whose job it is to expose functions that are then attached to the BPF program via uprobes. These uprobes are restricted to only trace calls that originate from the daemon's own PID, which prevents another binary from simply loading that library code and issuing unauthorized commands to the BPF program. Figure 3.2 depicts the process of using ebph-admin to make a request to the daemon. Section 3.7 discusses the various underlying mechanisms used to support commands in further detail.

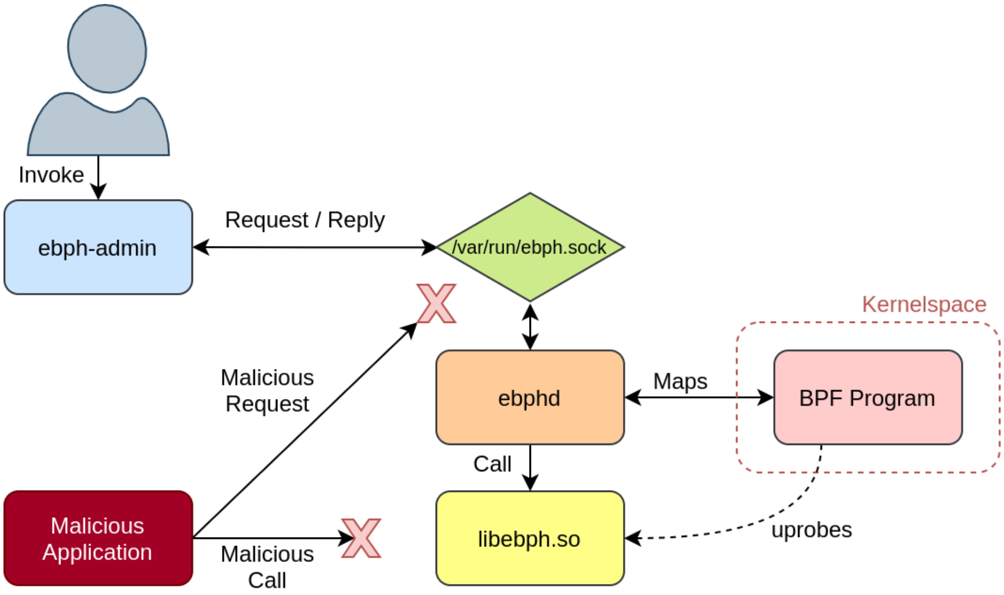

Figure 3.2: Dataflow of a request from ebph-admin. The program issues requests to the daemon, which then either directly accesses a map or triggers the execution of a uprobe BPF program with libebph.so, depending on the complexity of the request. Note that malicious applications cannot abuse libebph.so to issue their own commands – only the daemon can do this.

3.3.4 ebpH Logs

The current iteration of ebpH uses log data to communicate events to the user. The daemon logs all important events to log files located at `/var/log/ebpH` by default, including detected anomalies with corresponding sequences, profile creation, and process normalization. Event logging categories roughly correspond to the perf event buffers depicted in Table 3.2 on page 25, since the daemon writes a log message whenever one of these events is observed.

Since the current version of ebpH does not include a GUI (although there are plans to reintroduce a GUI in the future), logs must be frequently checked to keep track of system behavior and any detected anomalies. As a temporary stopgap, scripts can watch the logfile on behalf of the user and send more conspicuous notifications to the user. A few such scripts are included with ebpH; for instance, `watch-anom.sh` watches for anomaly events in the logfile with `tail` and sends a push notification to the user via the `notify-send` command.

Using a combination of scripts and manual log analysis, the user can gain a clear picture of how their system is behaving and whether any anomalies have occurred in said behavior. When the ebpH GUI is reintroduced in future iterations, it will be much easier to observe system behavior in detail. Refer to Section 5.2.5 for a description of future work with respect to ebpH's GUI.

3.4 ebpH Profiles

In order to monitor process behavior, ebpH keeps track of a unique profile (Listing 3.3) 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, ebpH takes 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.

Listing 3.3: A simplified definition of the ebpH profile struct.

```
struct ebpH_profile_data
{
    u8 flags[SYSCALLS][SYSCALLS]; /* System call lookahead pairs */
    u64 last_mod_count; /* Syscalls since profile was last modified */
    u64 train_count;    /* Syscalls seen during training */
};

```

```
struct ebpH_profile
{
    struct ebpH_profile_data train; /* Training data */
    struct ebpH_profile_data test;  /* Testing data */
    u8 frozen;          /* Is the profile frozen? */
    u8 normal;          /* Is the profile normal? */
    u64 normal_time;    /* Minimum system time required for normalcy */
    u64 anomalies;      /* Number of anomalies in the profile */
    char comm[128];     /* Name of the executable file */
};
```

The profile itself is a C data structure that keeps track of information about the executable, as well as a sparse two-dimensional array of lookahead pairs [50] to keep track of system call patterns. Each entry in this array consists of an 8-bit integer, with the i^{th} bit corresponding to a previously observed distance i between the two calls. When ebpH observes this distance, it sets the corresponding bit to `1`. Otherwise, it remains `0`. Each profile maintains lookahead pairs for each possible pair of system calls. Figure 3.3 presents a sample (`read`, `close`) lookahead pair for the `ls` binary.

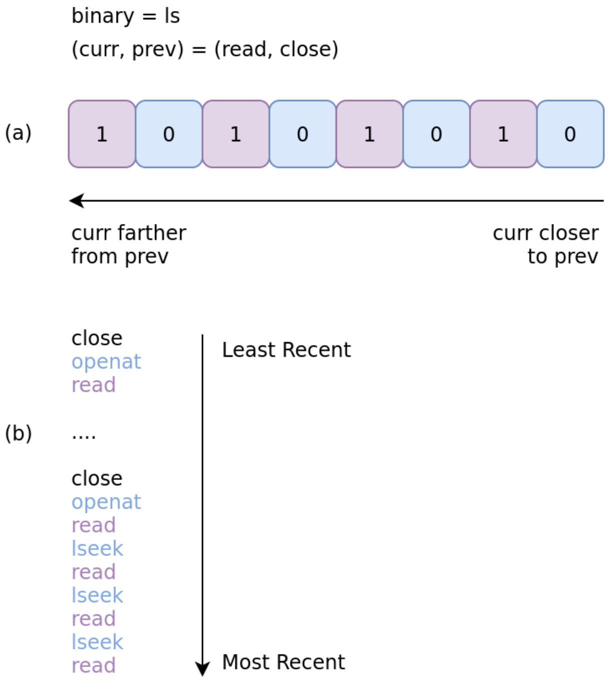

Figure 3.3: A sample (`read`, `close`) lookahead pair in the ebpH profile for `ls`. (a) shows the lookahead pair and (b) shows two relevant system call sequences, separated by several omitted calls. Note that the first three system calls in both the first and second sequence are responsible for the two least significant bits of the lookahead pair.

Each process (c.f. Section 3.5) is associated with exactly one profile at a time. Profile association is updated whenever ebpH observes 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. This forms the crux of how ebpH is able to monitor process behavior.

Just like in the original pH [49], profile state is tracked using the `frozen` and `normal` fields. When a profile's behavior has stabilized, it is marked frozen. If a profile has been frozen for one week (i.e. system time has reached `normal_time`), the profile is then marked normal. Profiles are unfrozen when new behavior is observed and anomalies are only flagged in normal profiles.

3.4.1 Writing Profiles to Disk and Reading Profiles from Disk

In order to allow profile data to persist across machine reboots, ebpH periodically writes profile data to disk, at an interval configurable the user, as well as when the BPF program is unloaded by the user. Profiles are read from disk when ebpH first loads.

In the original pH, profile data was saved and loaded in kernelspace [49] which meant that it required kernelspace file I/O, which is often regarded as an unsafe practice. ebpH solves this problem by moving all file I/O operations into userspace. This is made possible due to the bidirectional nature of dataflow with respect to eBPF maps. Specifically, when writing to disk, the daemon queries profile data from each entry in the profile map and writes that data to a file (`/var/lib/ebpH/<profile_key>`). When reading from disk, the daemon reads profile data from the appropriate files (`/var/lib/ebpH/<profile_key>`) and associates that data with keys in the newly created profile map.

3.5 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 3.4 for a simplified definition of the ebpH process struct.

Listing 3.4: A simplified definition of the ebpH process struct.

```
struct ebpH_sequence
{
    long seq[9];        /* Remember 9 most recent system calls in order */
    u8 count;           /* How many system calls are in our sequence? */
};

struct ebpH_sequence_stack
{
    ebpH_sequence[3]; /* Keep track of up to 3 sequences at a time */
    int top;            /* Top of the sequence stack, values from 0-2 */
    int should_pop;     /* Pop from the stack on next system call */
};
```

```

struct ebpH_process
{
    struct ebpH_sequence_stack;
    u32 pid;          /* Kernel tgid */
    u32 tid;          /* Kernel pid */
    u64 profile_key;  /* Associated profile key */
    u8 in_execve;     /* Are we in the middle of an execve? */
};
```

ebpH monitors process behavior by instrumenting tracepoints all system calls. On every system call return, ebpH adds the corresponding system call number to the process' current sequence (ebpH actually maintains a *stack* of sequences in order to handle non-deterministic behavior; this will be covered in more detail shortly). This sequence is subsequently used to index into the corresponding profile's lookahead pairs and flip the correct bits. If the process' profile is normal, new sequences will trigger ebpH's anomaly detection mechanism and a warning will be sent to userspace.

In addition to the system call tracepoints described above, ebpH also keeps track of a few other tracepoints to keep track of profile creation, process creation and deletion, and profile association on `exec`-family system calls. The `sched` family exposes the necessary tracepoints in order to do this. Additionally, ebpH defines one kprobe in order to detect when a process invokes its signal handler. Table 3.3 summarizes the tracepoints and kprobes used by ebpH along with their side effects on ebpH's state.

Table 3.3: eBPF tracepoints and kprobes used in ebpH.

Tracepoint/Kprobe	Description	ebpH Side Effect
`sys_enter`	Tracepoint invoked just after system call entry	Check for return from a signal handler and pop from sequence stack if necessary
`sys_exit`	Tracepoint invoked just before system call return	Operate on per-process system call sequences and per-profile lookahead pairs
`sched_process_fork`	Tracepoint invoked just after a call to `fork(2)`, `vfork(2)`, or `clone(2)`	Create a new process struct and add it to the hashmap
`sched_process_exec`	Tracepoint invoked just after an `exec`-family system call	Create a profile if necessary, adding it to the hashmap, and associate it with a process
`sched_process_exit`	Tracepoint invoked just after a thread exits	Delete a process struct from the hashmap
`get_signal`	Kretprobe invoked when a process' signal handler is about to be called	Push a new frame onto the process' sequence stack

[9] The majority of these values are subject to significant optimization in future iterations of ebpH. The

3.5.1 Profile Creation and Association

There are several important considerations here. First, ebpH needs a way to assign profiles to processes, which is done by instrumenting the result of an `execve(2)` system call using the `sched_process_exec` tracepoint. This tracepoint allows us to access information provide by the `linux_bin_prm` struct, which is used to store information about the executable or interpreted script that `execve(2)` has loaded. In particular, the executables inode and filesystem device number are used in combination to compute a key that uniquely maps to an individual executable on disk. Without this, ebpH would be unable to differentiate between two paths that resolve to a binary with the same name, for example `/usr/bin/ls` and `./ls`; this is due to an unfortunate nuance in `execve(2)`'s treatment of pathnames (i.e. it only considers relative paths when provided in order to save on memory).

In addition to associating a process with the correct profile, ebpH also wipes the process' current sequence of system calls, to ensure that there is no carryover between two unrelated profiles when constructing their lookahead pairs. This is important in order to prevent `execve (2)` calls from being used to construct artificially good sequences in a profile which may be later used to mask malicious behavior [49].

3.5.2 Profile Association and Sequence Duplication

Another special consideration is with respect to `fork(2)` and `clone(2)` family system calls. A forked process should begin with the same state as its parent and should (at least initially) be associated with the same profile as its parent. A subsequent `execve(2)` (i.e. the `fork-execve` pattern) would then overwrite this association. In order to accomplish this, ebpH instruments tracepoints for the `fork(2)`, `vfork(2)`, and `clone(2)` system calls, ensuring that it associates the child process with the parent's profile, if such a profile exists. If ebpH detects an `execve(2)` as outlined above, it will simply overwrite the initial profile association provided by the fork. The parent's current system call sequence is also copied to the child to prevent forks from being used to break sequences.

3.5.3 Dealing with Signal Handlers and Non-Determinism

As an anomaly-based intrusion detection system, it is critical that ebpH be able to establish normal profiles of program behavior in a timely manner. As presented in previous sections, establishing the normalcy of a profile requires that the it has been active for at least a week and that the ratio of total system calls seen during training to system calls the last time

2^8 value is a sensible default chosen by bcc. In practice, many of these events are infrequent enough that smaller buffer sizes would be sufficient.

the profile was modified be sufficiently large. As a corollary, every time a process makes a system call that results in a previously unobserved sequence, this ratio becomes increasingly difficult to achieve. In practice, this means that it is much harder to normalize profiles that exhibit less deterministic behavior. As a practical example, consider the time required to stabilize a relatively simple binary, such as `ls` versus a complex web browser like `firefox`; not only does `firefox` make significantly more system calls during an average run, but it is also far more likely to produce a previously unseen sequence at any given time.

This problem of normalizing profiles is compounded by the non-deterministic behavior introduced by signals and signal handlers. This phenomenon was first noted by Amai et al. [1] in a 2005 technical paper on the original pH system. In particular, they noted that signal handlers were a significant source of non-deterministic behavior in processes that ultimately led to significantly longer wait times until profile normalcy. This effect is not difficult to see in practice, especially in the context of complex programs that run for extended periods of time, such as the above `firefox` example. Suppose that we have some sequence of system calls (A, B, C, D, E) and a signal handler that invokes system calls (F, G, H); depending on when this signal is caught during the initial sequence, the resulting sequence can vary significantly. For example, we might see (A, F, G, H, B, C, D, E) in one instance and (A, B, C, D, F, G, H, E) in another. This results in a significant deterioration in profile stability, and subsequently in profile normalcy times.

ebpH deals with the problem of signal handlers in the same manner proposed by Amai et al. [1]. Specifically, it maintains a stack of system call sequences in each process struct; each time a traced process receives a signal, ebpH pushes a frame onto this stack, and when the process exits from its signal handler, ebpH pops the frame. This has the effect of temporarily wiping ebpH's memory of a process' current system call sequence whenever it enters a signal handler, allowing subsequent lookahead pairs to be unaffected by the execution context prior to the handler's invocation and vice versa. In order to decide when to push, ebpH instruments a kprobe on the kernel's `get_signal` implementation; this allows it to detect when a process receives a signal that will be handled. Subsequently, ebpH detects a return from a signal handler by checking for the `rt_sigreturn` system call; when ebpH detects such a return, it pops the top frame from the sequence stack.

3.5.4 Reaping Processes

ebpH reaps tasks from its process map whenever detects that they have exited. By reaping process structs from our map as ebpH is finished with them, it is able to ensure that the map neither fills up, nor does it consume more memory than necessary. In order to detect when a task exits, ebpH instruments the `sched_process_exit` tracepoint provided by the kernel's trace API. This tracepoint is triggered whenever the scheduler handles the termination of a task. Within the BPF program associated with the tracepoint, ebpH simply determines the task's PID and deletes that key from the process map.

3.6 Training, Testing, and Anomaly Detection

ebpH profiles are tracked in two phases, *training mode* and *testing mode*. Profile data is considered training data until the profile becomes normal (as described in Section 3.4). Once a profile is in testing mode, the lookahead pairs generated by its associated processes are compared with existing data. When mismatches occur, they are flagged as anomalies which are reported to userspace via a perf event buffer. The detection of an anomaly also prompts ebpH to remove the profile's normal flag and return it to training mode.

3.6.1 A Simple Example of ebpH Anomaly Detection

As an example, consider the simple program shown in Listing 3.5. This program's normal behavior is to simply print a message to the terminal. However, when issued an extra argument (in practice, this could be a secret keyword for activating a backdoor), it prints one extra message. This will cause a noticeable change in the lookahead pairs associated with the program's profile, and this will be flagged by ebpH if the profile has been previously marked normal.

Listing 3.5: `anomaly.c`, a simple program to demonstrate anomaly detection in ebpH.

```c
/* anomaly.c */

#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    /* Execute this fake anomaly
     * when we provide an argument */
    if (argc > 1)
        printf("Oops!\n");
    /* Say hello */
    printf("Hello world!\n");

    return 0;
}
```

In order to test this, I artificially lower ebpH's normal time to three seconds instead of one week. Then, I run the above test program several times with no arguments to establish normal behavior. Once the profile has been marked as normal, I then run the same test program with an argument to produce the anomaly. ebpH immediately detects the anomalous system calls and flags them. These anomalies are then reported to userspace via a perf buffer as shown in Listing 3.6.

Listing 3.6: The flagged anomaly in the `anomaly` binary as shown in the ebpH logs. Note that ebpH also logs the offending sequence, reordering it so that most recent system calls appear on the right.

```
    WARNING: Anomalies in PID 11162 (anomaly 38803844):
        MPROTECT, MPROTECT, MPROTECT, MUNMAP, FSTAT, BRK, BRK, WRITE, WRITE
```

From here, one can figure out exactly what went wrong by inspecting the system call sequences produced by the `anomaly` program, in both cases and comparing them with their respective lookahead pair patterns. Figure 3.4 provides an example of this comparison.

While this contrived example is useful for demonstrating ebpH's anomaly detection, process behavior in practice is often more nuanced. ebpH collects at least a week's worth of data about a process' system calls before marking it normal, which often corresponds with several branches of execution. In a real example, the multiple consecutive write calls might be a perfectly normal execution path for this process; by ensuring that ebpH takes its time before deciding whether a process' profile has reached acceptable maturity for testing, it decreases the probability of any false positives.

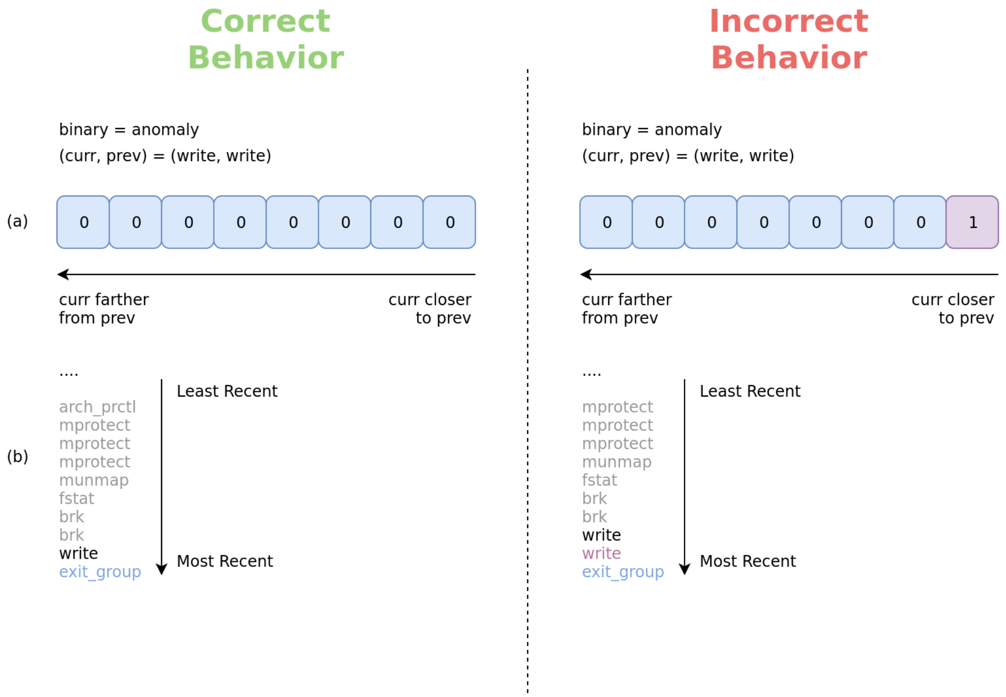

Figure 3.4: Two sample (`write`, `write`) lookahead pairs in the ebpH profile for `anomaly.c`. (a) shows the lookahead pair and (b) shows two relevant system call sequences. The left hand side depicts normal program behavior, while the right hand side depicts our artificially generated anomaly. There are several other anomalous lookahead pairs which result from this extra write call, but we focus on (`write`, `write`) for simplicity.

3.7 The `ebphd` Commands API

3.7.1 Setting Runtime Parameters

3.7.2 Examining Profiles and Processes

3.7.3 Issuing More Complex Commands

3.8 Soothing the Verifier: Technical Challenges of ebpH's Implementation

The development of ebpH elicited many challenges with respect to the eBPF verifier. As seen in Section 2.3.2, eBPF programs become more difficult to verify as they increase in complexity; as a corollary, when developing large and complex eBPF programs, a great deal of care and attention must be paid to ensure that the verifier will not reject the code.

The problem of dealing with the eBPF verifier can be expressed in the form of several subproblems as follows:

1) Many kernel functions and programming constructs are prohibited in eBPF;
2) eBPF programs come with a hard stack space limit of 512 bytes;
3) Traditional C-style dynamic memory allocation is prohibited;
4) Support for bounded loops is in its infancy and such loops must be carefully constructed to avoid verifier issues;
5) The verifier tends to err on the side of caution and will produce false positives with non-negligible frequency.

Subproblem (1) poses a particular challenge for a few aspects of ebpH's design: namely, profile keying and storage, `execve(2)` abortion, and issuing system call delays. This is due to the fact that eBPF programs do not have access to many of the helper functions available for traditional kernel development. As such, profile keying and storage have been fundamentally changed in ebpH compared to its predecessor, and `execve(2)` abortion and system call delays have been left as topics for future work (see Section 5.2.2). The original pH [49] stored profiles as a linked list and indexed them using executable pathnames. Unfortunately, the kernel helper required to build pathnames from a `dentry` is not available in eBPF and, while a partial solution has been submitted for review in the kernel upstream [66], this will likely not be merged into the mainline until a much later version of Linux. As befits the BPF paradigm, ebpH stores its profiles in a global hashmap instead of a linked list and indexes this hashmap by a uniquely computed integer based on executable metadata, namely its inode and filesystem device number. Profiles are then augmented with the executables *filename*[10] for usability purposes.

From subproblems (2) and (3), one immediate issue arises: with no means of explicit dynamic memory allocation and a stack space limit of 512 bytes, ebpH needs an alternative method of instantiating the relatively large data structures described in Section 3.4 and Section 3.5, as both the `ebpH_profile` and `ebpH_process` structs are larger than would be allowed in the eBPF stack. Fortunately, a creative solution exists this problem which leverages the `BPF_ARRAY`'s unique property of zero-initializing elements on creation. What this means is that ebpH can maintain a size `1` array for each data structure it requires; when it needs to instantiate a struct, all it needs to do is look up this value from the array, and copy it into the corresponding global hashmap. Fortunately, we can creatively solve this problem by using a `BPF_ARRAY` for initialization. This technique constitutes the design pattern outlined in Listing A.1 of Appendix A.

In order to prevent ebpH's maps from consuming all of the system's memory, they are flagged with `BPF_F_NO_PREALLOC` which notifies the kernel that these maps should be dynamically

[10] This is not the same thing as a *pathname*.

allocated at runtime as opposed to statically allocated at load time. While this lessens the burden on the system, it is not an ideal solution. There are known issues with dynamically allocated maps [56] which may cause deadlocks when used in certain high-volume tracing events such as kernel spinlock counters. For ebpH's prototype, this trade-off in reliability is acceptable, but future versions will be refactored to make use of a combination of `LRU_HASH` for low memory overhead and `HASH_OF_MAPS` for lookahead pair storage. This will provide a more reliable and more memory efficient approach than the current dynamic hashmap allocation. Section 5.2.4 discusses this future refactor in more detail.

From subproblem (4), the obvious issue arises that loops need to be "simple" enough for the eBPF verifier to reason about them. For example, loops that have entrypoints in the middle of iteration will potentially be flagged if the verifier is unable to correctly identify the loop structure [8]. Since the verifier relies on pattern matching in order to identify induction variables, LLVM optimizations to eBPF bytecode introduce an element of fragility to loop verification [8]. Bounded loops that perform memory access using the induction variable are also quite finicky at best; the verifier must be able to show that memory access is safe in all possible states – this precludes induction variables from having an unsafe lower *or* upper bound when they are used to index into a buffer. These limitations affect ebpH and its design in non-trivial ways; for example, ebpH requires specially crafted helper functions to perform simple operations such as indexing into the array of lookahead pairs or per-process system call sequences. These helpers perform extra checks on the bounds of the variable being used to index into the array and are designed to handle failure gracefully. Additionally, special compiler macros are employed to ensure that LLVM optimizations do not affect their integrity.

Subproblem (5) is perhaps the most difficult to reckon with, but is quite understandable when considering the gravity and difficulty of the problem that the verifier is trying to solve. As shown in Section 2.3.2, guaranteeing the safety of arbitrary untrusted programs is a difficult problem and concessions need to be made in order for such guarantees to be tenable. False positives are unfortunately one of those concessions. When the verifier rejects code due to a false positive, there is simply no better solution than to try a different approach. ebpH triggered many false positives during its development which required significant refactoring of otherwise reasonable code. While these verifier false positives are unfortunate, they are a far cry from the vexing kernel panics, data corruptions, and other crashes that so often occur during ordinary kernel development – ebpH crashed the system precisely *zero* times during testing *and* development. This extraordinary feat is made possible by eBPF's safety guarantees.

4 Measuring ebpH's Overhead

One of the primary advantages of eBPF is its relatively low overhead [16, 54, 55] compared to many other system introspection solutions (c.f. Section 2.1 and Section 2.3). In order to justify this claim in the context of an eBPF intrusion detection system, it is necessary to ascertain the overhead associated with running ebpH on a variety of systems under a variety of workloads (artificial and otherwise). Here I describe the tests that were conducted in order to determine this overhead. Section 4.1 outlines the systems and tools used for testing and provides an overview of the collected datasets. The specifics of each benchmarking test along with the results are provided in Section 4.2.

4.1 Methodology

Since ebpH's kernelspace functionality resides in system call hooks, we can get an idea of what overhead it imposes on the system by running macro-benchmarks on the time required to make system calls. Initially, I planned to use `syscount` [22] from `bcc-tools` for this purpose, however this tool currently has a race condition that may affect results due to its use of `BPF_HASH` rather than `BPF_PERCPU_ARRAY` for data storage (c.f. Table 2.2 on page 10). Instead, a custom benchmarking tool, `bpfbench`[11], was written in eBPF for this purpose. Like `syscount`, `bpfbench` measures system call overhead by taking the difference of `ktime` (in nanoseconds) between system call entry and return; this difference along with the number of calls observed is stored in an eBPF map for later analysis. Unlike `syscount`, `bpfbench` stores this data in a per-cpu array, aggregating data at the end when necessary; this means that neither the system call count nor the system call overhead is subject to race conditions like its predecessor. See Appendix B for the BPF portion of `bpfbench`'s source code.

Macro-benchmarking data was collected on various systems, including a server used in production, a personal computer, and a CCSL (Carleton Computer Security Lab) workstation. Tests were run under a variety of workloads and benchmarking data was collected using `bpfbench`. For each dataset, the same test was conducted on the system twice: once with ebpH running, and once without. All ebpH data was collected while ebpH was monitoring the entire system (i.e. started immediately on boot via a `systemd` unit) and running with normal parameters and logging settings. Table 4.1 summarizes each of the systems used for the collection of benchmarking data, including relevant hardware specifications, and Table 4.2 provides a description of each dataset, including the system and the workload tested.

[11] Full source code available at https://github.com/willfindlay/bpfbench.

Table 4.1: Systems used for the collection of ebpH benchmarking data.

System	Description	Specifications	
`arch`	Personal workstation	Kernel	5.5.10-arch1-1
		CPU	Intel i7-7700K (8) @ 4.500GHz
		GPU	NVIDIA GeForce GTX 1070
		RAM	16GB DDR4 3000MT/s
		Disk	1TB Samsung NVMe M.2 SSD
`bronte`	CCSL workstation	Kernel	5.3.0-42-generic
		CPU	AMD Ryzen 7 1700 (16) @ 3.000GHz
		GPU	AMD Radeon RX
		RAM	32GB DDR4 1200MT/s
		Disk	250GB Samsung SATA SSD 850
`homeostasis`	Mediawiki server	Kernel	5.3.0-42-generic
		CPU	Intel i7-3615QM (8) @ 2.300GHz
		GPU	Integrated
		RAM	16GB DDR3 1600MT/s
		Disk	500GB Crucial CT525MX3

Table 4.2: ebpH macro-benchmarking datasets.

Dataset	System	Workload	Description
`bronte-7day`	`bronte`	Idle	`bpfbench`, 7 days with ebpH and 7 days without
`homeostasis-3day`	`homeostasis`	Production	`bpfbench`, 3 days with ebpH and 3 days without
`arch-3day`	`arch`	Normal use	`bpfbench`, 3 days with ebpH and 3 days without

After benchmarking data was collected, overhead was calculated according to the following equation:

$$\text{Overhead}_{\text{syscall}} = \frac{T_{\text{ebph}_{\text{syscall}}} - T_{\text{base}_{\text{syscall}}}}{T_{\text{base}_{\text{syscall}}}}$$

where,

$$T_{\text{syscall}} = \frac{\text{Total time}}{\text{Number of occurrences}}$$

as measured by bpfbench.

In addition to system call overhead, we are also interested in how ebpH affects overall system performance. In particular, ebpH should have negligible impact on normal system use. In order to ascertain this, micro-benchmarking data was collected for a variety of test cases. Table 4.3 provides a description of each micro-benchmark dataset, including the system and the workload tested. Additional details of each micro-benchmark test are provided in their respective results sections.

Table 4.3: ebpH micro-benchmarking datasets.

Dataset	System	Workload	Description
bronte-lmbench	bronte	Artificial	OS micro-benchmarks from the lmbench [32, 33] suite, with and without ebpH
arch-x11perf	arch	Artificial	x11perf [31, 36] full benchmarking suite, with and without ebpH

4.2 Results

This section presents the results of all benchmarks. Micro-benchmarking results will be presented first in order to provide a more statistically significant depiction of ebpH's overhead, followed by macro-benchmarking data collected with bpfbench to cover ebpH's behavior in production environments. Many of the macro-benchmarking datasets have been trimmed for brevity and the exclusion of outlier datapoints; these datasets will be provided in full in Appendix C.

4.2.1 bronte-lmbench: Measuring System Latency with lmbench Micro-Benchmark

McVoy's lmbench [32, 33] is a Linux micro-benchmarking suite that has seen prominent use in academia [3, 9, 37, 48] for establishing various performance metrics of UNIX-like systems. In particular, we are interested in the *OS-category* benchmarks provided by lmbench. This category provides performance metrics such as:

- Simple system call latency (c.f. Table 4.4 and Figure 4.1);
- select(2) latency on various file types (c.f. Table 4.5 and Figure 4.2);
- Signal handler latency (c.f. Table 4.6 and Figure 4.3);
- Dynamic process creation latency (c.f. Table 4.7 and Figure 4.4);

- IPC (inter-process communication) latency for pipes and UNIX stream sockets (c.f. Table 4.8 and Figure 4.5).

Simple system call and `select(2)` latency will give us an idea of how ebpH affects system call overhead directly, while signal handler latency will show the overhead caused by both ebpH's treatment of the underlying system calls as well as the signal-aware stack discussed in Section 3.5.3. Finally, the process creation and IPC latency metrics will provide a better picture of ebpH's overhead in a more practical context. 1000 ebpH and 1000 non-ebpH OS-category trials were run on `bronte`, a workstation in the CCSL (Carleton Computer Security Lab) at Carleton University. The results were then averaged and compared to determine overhead.

Table 4.4: Results of the system call benchmarks from the `bronte-lmbench` dataset. Standard deviations are given in parentheses and smaller overhead is better. Note that the `open/close` benchmark shows the times of *both* system calls taken together, which explains why the difference between base and ebpH times is doubled. This was an unfortunate design choice by the developers of `lmbench`.

System Call	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`getppid`	0.058 (0.0023)	0.416 (0.0157)	0.357811	614.784969
`write`	0.111 (0.0039)	0.469 (0.0168)	0.357955	321.179901
`read`	0.187 (0.0064)	0.540 (0.0185)	0.353581	189.189001
`fstat`	0.194 (0.0062)	0.552 (0.0171)	0.357821	184.176095
`stat`	0.587 (0.0146)	0.973 (0.0250)	0.386082	65.765787
`open/close`	1.043 (0.0348)	1.830 (0.0567)	0.787454	75.509370

As shown in Table 4.4, ebpH adds non-negligible overhead to simple system calls. However, this result is misleading, as the actual difference between base and ebpH times is less than a microsecond (about one third of a microsecond to be more precise). As soon as base times for system calls approach one microsecond, (e.g. in the case of `stat(2)`), overhead drops significantly. For extremely short calls like `getppid(2)`, the overhead is just over 614%, which is representative of the worst case, but longer system calls like `stat(2)`, overhead drops to about 66%. This overhead is more representative of the general case.

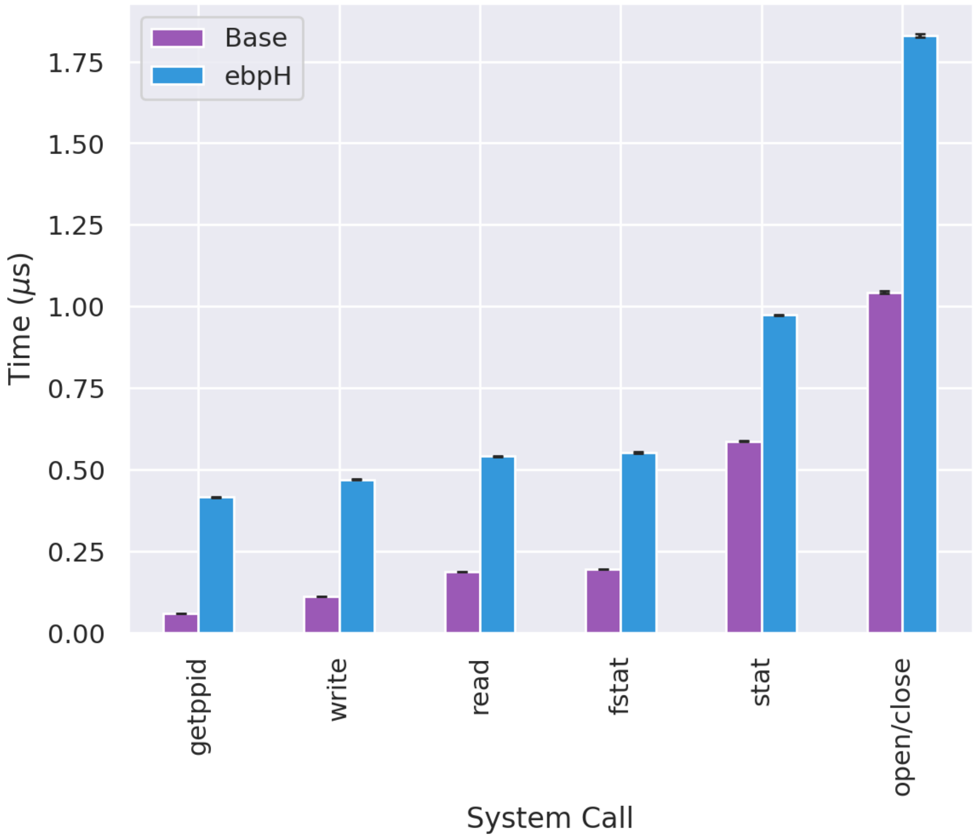

Figure 4.1: Mean system call times from the `bronte-lmbench` dataset. Standard error is given as error bars. Smaller difference in times is better.

Table 4.5: Results of the `select(2)` benchmarks from the `bronte-lmbench` dataset. Standard deviations are given in parentheses and smaller overhead is better.

Type	Count	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`Regular File`	10	0.362 (0.0128)	0.723 (0.0282)	0.360632	99.565990
`Regular File`	100	1.231 (0.0372)	1.596 (0.0443)	0.365494	29.699868
`Regular File`	250	2.639 (0.0799)	2.996 (0.0956)	0.356587	13.510287
`Regular File`	500	5.091 (0.1183)	5.426 (0.1490)	0.335187	6.584345
`TCP Socket`	10	0.436 (0.0144)	0.796 (0.0267)	0.360081	82.674990
`TCP Socket`	100	4.547 (0.1258)	4.928 (0.1792)	0.380938	8.378431
`TCP Socket`	250	11.433 (0.3849)	11.766 (0.3369)	0.332886	2.911606
`TCP Socket`	500	23.028 (0.8414)	23.530 (0.9567)	0.501917	2.179609

The `select(2)` system call benchmarks allow us to get an idea of the overhead imposed on a blocking system call in a controlled environment. `select(2)` [45] is used to wait until one or more file descriptors become available for a given operation; the `select(2)` benchmarks from

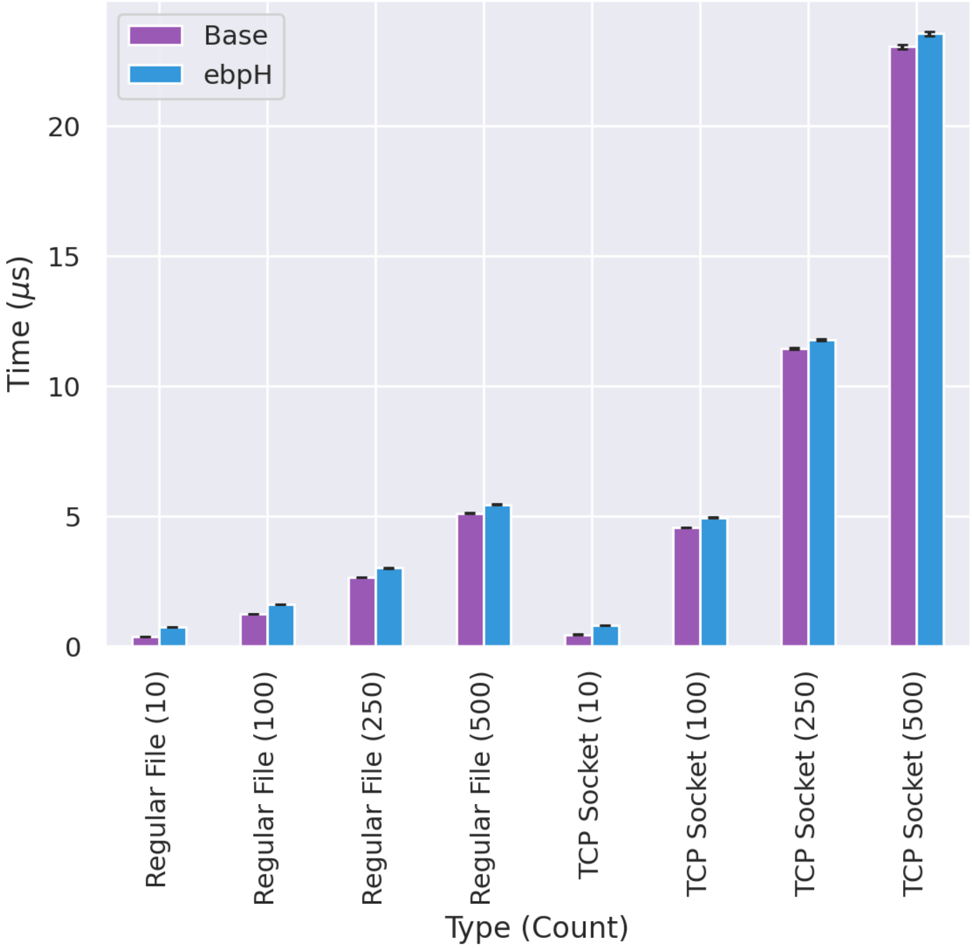

Figure 4.2: Mean `select(2)` times from the `bronte-lmbench` dataset. Standard error is given as error bars. Smaller difference in times is better.

`lmbench` invoke this system call on predefined sets of file descriptors, shown in Table 4.5. The results here demonstrate that the overhead imposed by ebpH rapidly diminishes for blocking system calls, and in some cases drops below the standard third of a microsecond that was observed previously; the likely explanation here is that the overhead incurred by ebpH is occurring during time that would otherwise be spent blocking.

As discussed in Section 3.5.3, ebpH makes use of special logic to separate the non-deterministic behavior caused by signal handlers from other observed process behavior. Table 4.6 shows that the overhead imposed on the execution of simple signal handlers is relatively low, around 39%. This result is especially impressive considering that it includes the standard per-system-call overhead (c.f. Table 4.4) imposed on `rt_sigreturn(2)` [47], which is invoked upon return from a signal handler.

Table 4.6: Results of the signal handler benchmarks from the `bronte-lmbench` dataset. "Installation" represents the registration of a signal handler with `rt_sigaction(2)` and "Handler" represents the time taken to complete a simple signal handler. Standard deviations are given in parentheses and smaller overhead is better.

Type	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`Installation`	0.205 (0.0061)	0.562 (0.0177)	0.357275	174.179379
`Handler`	1.333 (0.0420)	1.855 (0.0750)	0.522106	39.179999

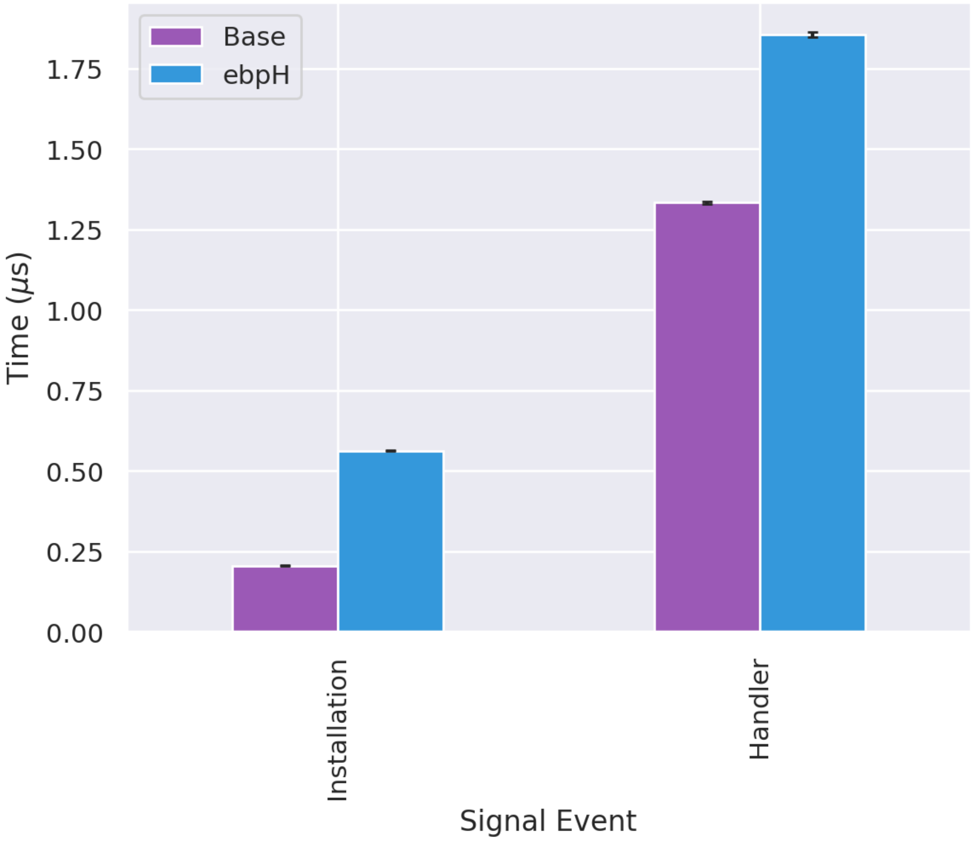

Figure 4.3: Mean signal handler times from the `bronte-lmbench` dataset. "Installation" represents the registration of a signal handler with `rt_sigaction(2)` and "Handler" represents the time taken to complete a simple signal handler. Standard error is given as error bars. Smaller difference in times is better.

While the previous benchmarking results have been informative with respect to the per-system-call and per-signal overhead of ebpH, they neglect to provide an accurate depiction of what this overhead might look like in practice. To that end, the dynamic process creation and IPC benchmarks offered by `lmbench` present a much clearer picture of ebpH's practical overhead. Table 4.7 presents the overhead of running three distinct process creation C programs as follows:

- `fork+exit` forks[12] itself and the child immediately exits;
- `fork+execve` forks itself and immediately executes a simple "hello world" program in the child;
- `fork+/bin/sh -c` forks itself and spawns a shell which then invokes the same "hello world" program described above. This roughly corresponds to the implementation of the C standard library's `system(3)` [61] interface.

The above three methods of process creation each involve increasing degrees of complexity with respect to their system calls and, as a corollary, the overhead caused by ebpH increases for each one. Stating with `fork+exit`, Table 4.7 shows that ebpH imposes very little overhead on basic process creation, on the order of 5 microseconds, or about 2.7%.

The `fork+execve` case introduces more overhead, due to the special operations that ebpH must perform when a process first executes, such as looking up a binary's inode information and associating it with a profile (creating this profile if it does not yet exist). While this operation is not free, it is inexpensive relative to the existing overhead of an `execve(2)` system call and imposes a total performance overhead of just 8%.

Finally, `fork+/bin/sh -c` imposes the most overhead of all three methods; this makes sense as it involves *two* `execve(2)` calls, one for `/bin/sh` and one for the "hello world" program, as well as the additional per-system-call overhead from `/bin/sh` itself. Still, the overhead for this method is only about 10%, which is acceptable in practice.

Table 4.7: Results of the process creation benchmarks from the `bronte-lmbench` dataset. Standard deviations are given in parentheses and smaller overhead is better.

Process	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`fork+exit`	200.503 (17.3410)	205.998 (11.2935)	5.494621	2.740415
`fork+execve`	536.914 (30.5695)	580.532 (47.9242)	43.617913	8.123821
`fork+/bin/sh -c`	1529.053 (20.5609)	1682.445 (13.9791)	153.392500	10.031866

[12] All forks produce the `clone(2)` system call.

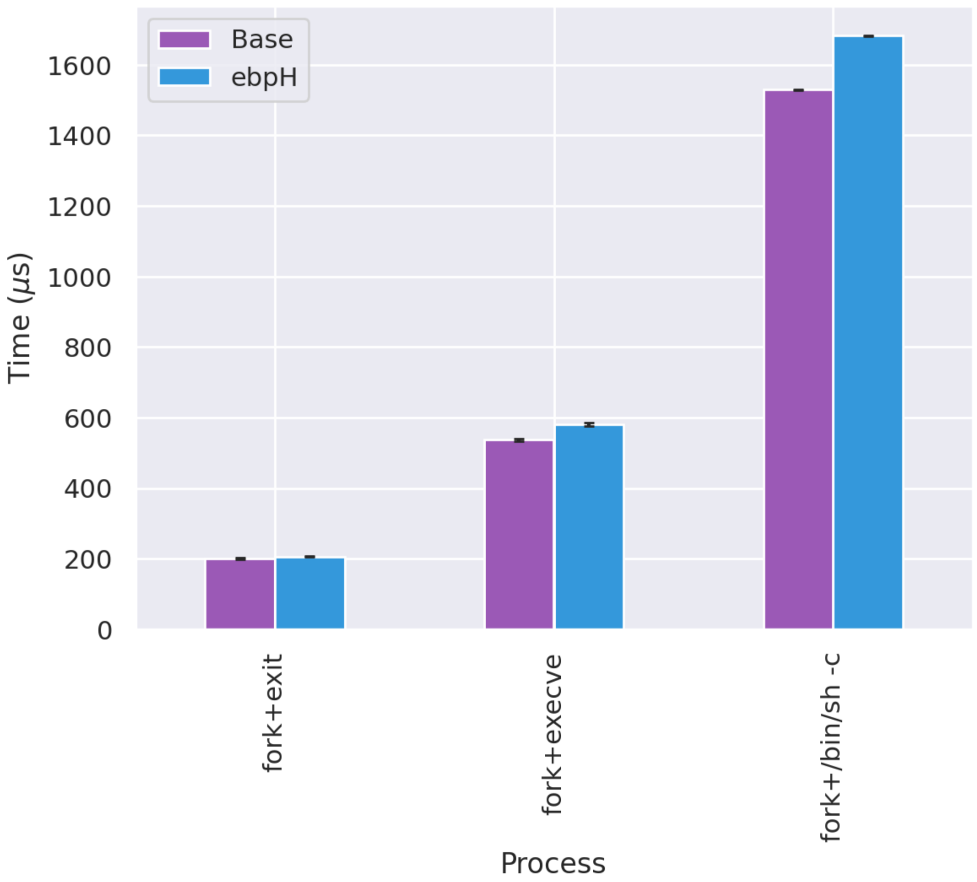

Figure 4.4: Mean process creation times from the `bronte-lmbench` dataset. Standard error is given as error bars. Smaller difference in times is better.

Table 4.8 shows the overhead caused by ebpH on two methods of IPC, pipes and Unix domain stream sockets. UNIX stream socket IPC, ebpH imposes an overhead of 1.7 microseconds, or about 18%. For pipes, it imposes an overhead of 1.25 microseconds, or about 28%.

Table 4.8: Results of the IPC benchmarks from the `bronte-lmbench` dataset. Standard deviations are given in parentheses and smaller overhead is better.

Type	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`Pipe`	4.510 (0.0236)	5.768 (0.0394)	1.257634	27.886271
`AF_UNIX`	9.367 (0.3300)	11.067 (0.1340)	1.699890	18.148105

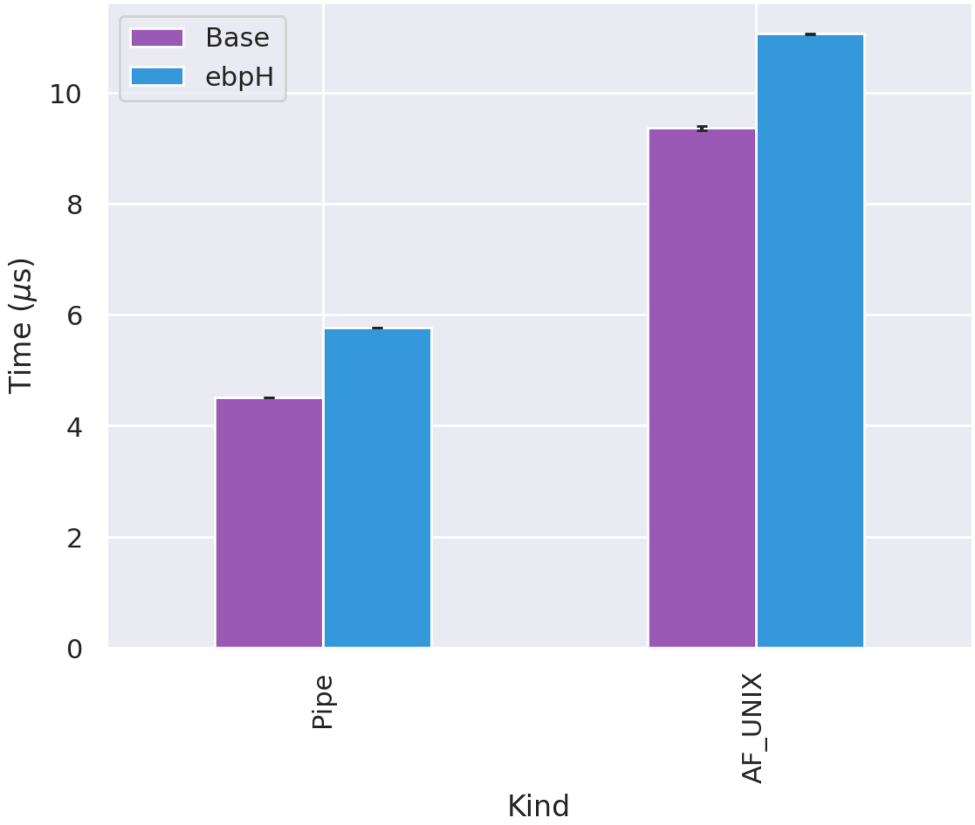

Figure 4.5: Mean IPC times from the `bronte-lmbench` dataset. Standard error is given as error bars. Smaller difference in times is better.

4.2.2 `bronte-7day` Macro-Benchmark

In order to get an idea of how ebpH interacts with the system as a whole, I ran a macro-benchmark of all system calls using `bpfbench` for a period of 14 days on `bronte`, the same CCSL workstation tested in Section 4.2.1. For the first seven days of testing, the system had been running ebpH since boot, and so all processes were being monitored. ebpH was run with normal parameters, profile saving, loading, and logging enabled. For the final seven days, the benchmark was run once again, this time without ebpH. All benchmarks for this dataset were run under an idle workload. After concluding the benchmark tests, overhead was compared between ebpH and non-ebpH data in order to ascertain what effect ebpH had on the system.

In order to eliminate outliers, we exclude any results more than 3 standard deviations from the mean. Outliers may be caused by a number of factors (since the measurements are for the entire system over an arbitrary period of time), although one common explanation would be blocking system calls taking an inordinate amount of time to finish, resulting in highly irregular execution times. Simple system calls (e.g. system calls with an execution time of less than about 3μs) offer the best measure of overhead caused by ebpH, and so will be the focus here. Table 4.9 and Figure 4.6 present the top 20 most frequent system calls with execution time of less than 3μs from the `bronte-7day` dataset.

Table 4.9: Top 20 most frequent system calls with base times of under 3μs from the `bronte-7day` dataset (after discarding outliers). Smaller overhead is better.

System Call	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`ioperm`	1.423	1.433	0.010	0.70
`personality`	0.801	1.059	0.258	32.21
`sched_get_priority_max`	0.291	1.149	0.858	294.85
`sched_get_priority_min`	0.446	1.327	0.881	197.53
`clock_gettime`	1.316	1.896	0.580	44.07
`timer_delete`	1.477	2.213	0.736	49.83
`timer_settime`	2.706	3.527	0.821	30.34
`fchownat`	2.905	4.138	1.233	42.44
`ioprio_set`	2.189	3.148	0.959	43.81
`listen`	1.082	1.821	0.739	68.30
`shmctl`	1.148	2.054	0.906	78.92
`clock_getres`	1.343	2.254	0.911	67.83
`setregid`	1.620	2.728	1.108	68.40
`getpgrp`	0.622	1.595	0.973	156.43
`setreuid`	1.807	2.787	0.980	54.23
`llistxattr`	2.170	3.134	0.964	44.42
`fgetxattr`	2.619	3.513	0.894	34.14
`linkat`	2.334	2.522	0.188	8.05
`sched_getparam`	0.602	1.333	0.731	121.43
`flistxattr`	0.794	1.721	0.927	116.75

System calls in the dataset shown in Table 4.9 do not present with overhead higher than about 294%. While 294% may seem high, it is important to remember that these overheads are on system calls that have an execution time of a few microseconds or less. In practice, ebpH is only adding a few microseconds of additional execution to these system calls, a slowdown that will likely go unnoticed by users.

In order to provide a better comparison with the experimental results from the `lmbench` system calls micro-benchmark, Table 4.10 and Figure 4.9 show the same system calls from the `bronte-7day` dataset. Note that the system-wide overhead is generally either the same or better than the `lmbench` results and `getppid(2)` in particular is seems to be significantly improved.

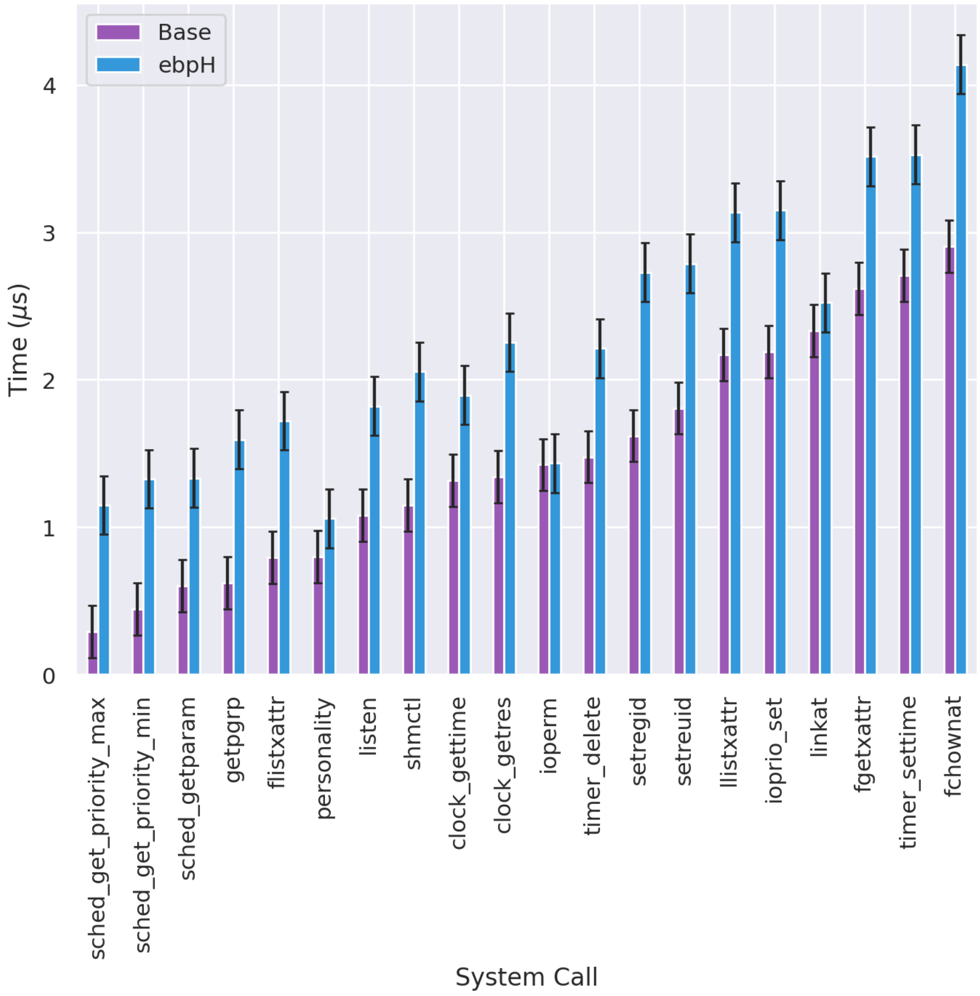

Figure 4.6: Top 20 most frequent system calls with base times of under 3μs from the `bronte-7day` dataset (after discarding outliers). Smaller difference in times is better. Standard error is given as error bars.

These results may be slightly misleading, however, since we are examining overhead across *the entire system* over and extended period of time, and also need to account for the overhead of `bpfbench` itself on base times. Regardless, these results may be slightly more indicative of the impact of ebpH on a real system, since they capture the behavior of the system in its entirety. We will have an opportunity to compare these results with a system under load in subsequent sections.

Table 4.10: Selected system calls from the `bronte-7day` dataset, for easy comparison with the `lmbench` data from Section 4.2.1.

System Call	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`close`	0.483	1.037	0.554	114.70
`fstat`	0.659	1.210	0.551	83.61
`getppid`	0.756	1.585	0.829	109.66
`stat`	1.042	2.112	1.070	102.69
`open`	1.782	2.284	0.502	28.17
`write`	13.576	19.980	6.404	47.17
`read`	45.982	60.200	14.218	30.92

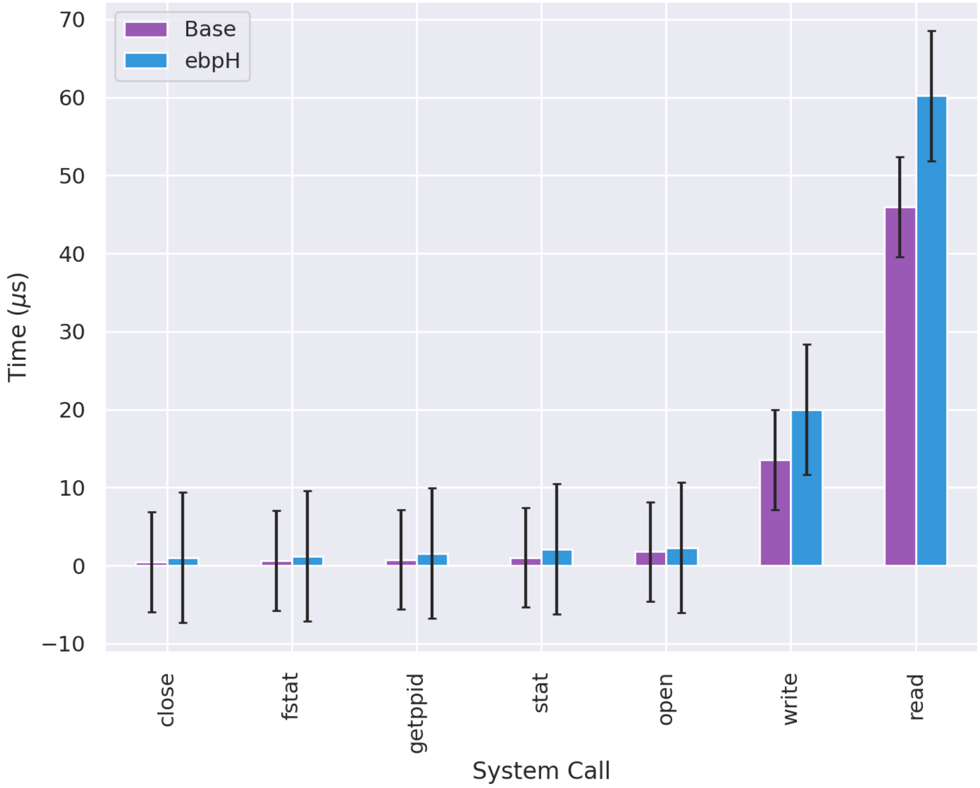

Figure 4.7: Selected system calls from the `bronte-7day` dataset, for easy comparison with the `lmbench` data from Section 4.2.1. Smaller difference in times is better. Standard error is given as error bars.

Table 4.11: Top 20 system call overheads by count in the `homeostasis-3day` dataset. Standard deviations are given in parentheses.

System Call	Count	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`recvfrom`	22311033	4.599 (0.3375)	5.505 (0.2877)	0.906167	19.705344
`stat`	16578427	2.929 (0.3409)	3.467 (0.1842)	0.537833	18.361831
`lstat`	12620479	3.560 (0.2693)	5.840 (5.5880)	2.280000	64.043445
`read`	12023727	14.824 (3.8911)	13.446 (3.5872)	-1.378083	-9.296142
`sendto`	11070639	11.330 (0.4644)	12.800 (0.2013)	1.469417	12.968969
`sched_yield`	7951003	0.810 (0.0547)	1.795 (0.1257)	0.985750	121.760165
`writev`	6195312	18.546 (7.1331)	18.717 (8.5272)	0.171500	0.924753
`write`	6102652	6.362 (3.2141)	15.007 (2.7556)	8.645000	135.890282
`close`	5701429	0.811 (0.0969)	1.236 (0.0387)	0.424833	52.405428
`ppoll`	5166406	650.215 (71.1027)	672.276 (29.9606)	22.060750	3.392837
`openat`	4985018	8.221 (1.3344)	8.762 (0.3944)	0.540833	6.578347
`fstat`	4196505	1.025 (0.1166)	1.434 (0.0439)	0.408750	39.878049
`rt_sigprocmask`	3746321	0.969 (0.1343)	1.405 (0.1113)	0.436083	44.987964
`access`	3302798	5.721 (0.3865)	6.499 (0.1562)	0.778000	13.599219
`lseek`	3010731	0.463 (0.0679)	0.903 (0.1624)	0.440083	95.067507
`mmap`	2438472	5.424 (0.5294)	6.022 (0.2403)	0.597917	11.024045
`getpid`	2063206	1.068 (0.0177)	1.520 (0.0247)	0.452333	42.350004
`recvmsg`	1745288	9.652 (0.8664)	10.700 (0.5092)	1.047917	10.857084
`ioctl`	1600634	75.770 (21.1585)	39.999 (5.9090)	-35.771583	-47.210644
`rt_sigaction`	1308454	0.450 (0.0363)	0.822 (0.0272)	0.371833	82.690882

4.2.3 `homeostasis-3day` Macro-Benchmark: ebpH in Production

4.2.4 `arch-3day` Macro-Benchmark: ebpH on a Personal Computer

4.3 Comparing Results with the Original pH

In order to facilitate easy comparison with the original pH system, much of the experimental methodology surrounding the testing of ebpH was designed to closely mimic that of Somayaji's original dissertation [49]. In particular, the `lmbench` and `x11perf` micro-benchmarks will be especially informative in this regard.

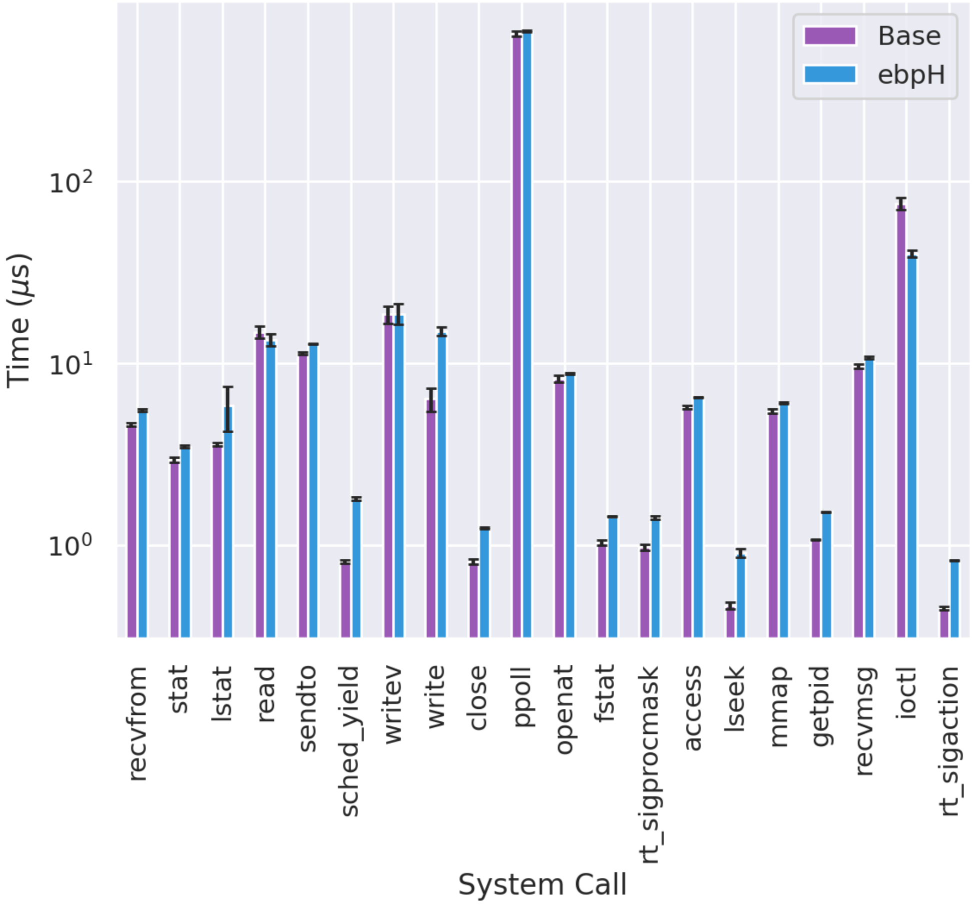

Figure 4.8: Top 20 system call overheads by count in the `homeostasis-3day` dataset. Time scale is logarithmic. Standard error is given as error bars.

Table 4.12: Top 20 system call overheads by count in the `arch-3day` dataset. Standard deviations are given in parentheses.

System Call	Count	T_{base} (μs)	T_{ebpH} (μs)	Diff. (μs)	% Overhead
`recvmsg`	1581788435	2.964 (1.2181)	3.101 (0.8429)	0.137255	4.631430
`read`	559839058	3.684 (1.8254)	4.005 (1.5622)	0.321018	8.713419
`sched_yield`	557122573	0.634 (0.1591)	0.663 (0.1554)	0.029755	4.696513
`write`	422940971	7.108 (3.2055)	7.808 (2.0907)	0.700155	9.850863
`getpid`	357026536	0.903 (0.4142)	0.994 (0.2102)	0.091282	10.110764
`writev`	297206760	7.358 (2.5764)	7.193 (1.6120)	-0.164718	-2.238682
`setitimer`	246047072	1.181 (0.4767)	1.133 (0.2606)	-0.048364	-4.093882
`stat`	233556922	1.389 (0.2925)	1.706 (0.7753)	0.316873	22.817491
`sendmsg`	176379278	9.729 (4.7259)	9.744 (2.4594)	0.014409	0.148103
`ioctl`	101399625	16.049 (20.0673)	10.558 (1.9098)	-5.490700	-34.212100
`madvise`	58208675	8.194 (5.4377)	9.717 (6.8087)	1.523955	18.599452
`openat`	47987948	4.453 (1.4879)	5.174 (1.3422)	0.721355	16.200947
`close`	43600631	0.862 (0.2795)	1.022 (0.2182)	0.159327	18.477596
`mprotect`	39922557	3.950 (0.6694)	4.142 (0.6807)	0.191336	4.843513
`epoll_ctl`	39571935	2.058 (0.7853)	2.441 (1.5081)	0.382518	18.585247
`gettid`	39531180	1.068 (0.3854)	1.426 (0.4344)	0.357536	33.465793
`recvfrom`	39304167	2.478 (0.9323)	2.567 (0.8106)	0.089282	3.603243
`lseek`	25079617	0.591 (0.1506)	0.677 (0.1540)	0.085891	14.535385
`readv`	22072501	5.009 (2.1978)	4.692 (1.5339)	-0.317555	-6.339104
`timerfd_settime`	21359440	3.406 (1.2268)	3.866 (0.6438)	0.459327	13.484747

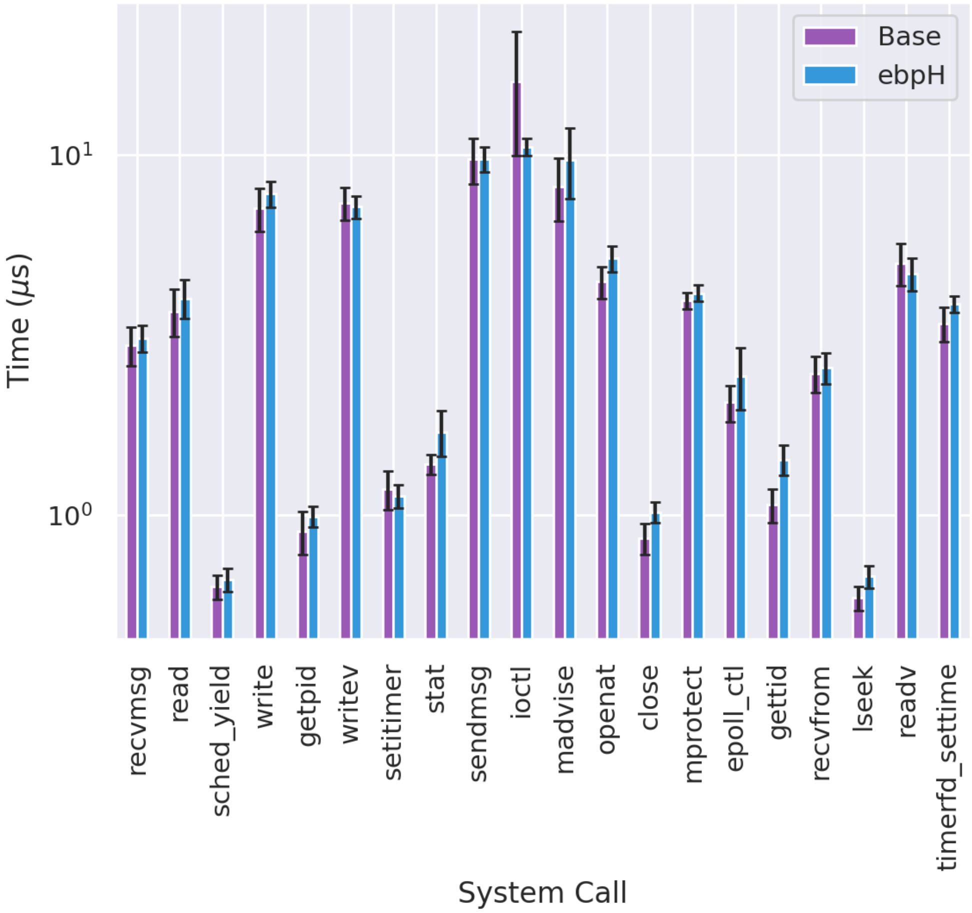

Figure 4.9: Top 20 system call overheads by count in the `arch-3day` dataset. Time scale is logarithmic. Standard error is given as error bars.

5 Discussion

5.1 Shortcomings of eBPF

5.2 Future Work

This thesis was primarily focused on three important points:

1) Establishing the viability of eBPF as a method for host-based anomaly detection;
2) Showcasing and describing ebpH, a partial reimplementation of Somayaji's pH [49] in eBPF;
3) Determining the experimental and practical overhead of ebpH on system performance.

Although these points are enough to define a significant contribution in the context of an undergraduate thesis, there remains several aspects of the project that can be improved upon or more thoroughly analyzed, and used for determining the direction of future iterations or other related research endeavors. To that end, I propose several topics for future work on ebpH and related projects in this section. Many of these will be explored in depth as part of my work for my upcoming Master of Computer Science thesis. In this section, I will be covering the following points:

1) The need to control for further sources of non-determinism (c.f. Section 5.2.1);
2) Potential avenues for adding automated response to ebpH (Section 5.2.2);
3) A security analysis of ebpH (c.f. Section 5.2.3);
4) Refactoring ebpH to use new hashmap types to reduce memory overhead and squash bugs (c.f. Section 5.2.4);
5) The need for a graphical user interface and subsequent usability study (c.f. Section 5.2.5);
6) Retrofitting ebpH to make use of other sources of system data, beyond system calls (c.f. Section 5.2.6).

5.2.1 Controlling for Further Sources of Non-Deterministic Behavior

5.2.2 Automating ebpH Response

ebpH's predecessor, pH [49], was capable of responding to attacks by issuing delays to system calls proportionally to recent anomalous behavior. The current version of ebpH lacks this functionality due to implementation constraints imposed by eBPF. However, recent additions to eBPF have made it more conducive to automated response [16]. In particular, Linux

5.3 introduced two critical helpers [20] for policy enforcement from BPF: `bpf_signal` and `bpf_override_return`.

`bpf_signal` provides the ability for BPF programs to send arbitrary signals to the current task directly from kernelspace. Since the signal is coming from the kernel, it will be delivered instantly, without the usual delays associated with sending signals from userspace. By sending a process the signal `SIGSTOP` [46], it will be possible to stop its execution in *real time*, during the offending system call. Subsequently, a `SIGCONT` [46] can be issued to wake the process once its delay has been observed. This second signal could either be sent from userspace (since we no longer have the same sense of urgency associated with the initial response) or issued from some frequently invoked BPF tracepoint, for example `sched_switch`, after a predetermined amount of time has passed.

`bpf_override_return` could be used to implement the second response category employed in pH [49]: `execve(2)` abortion, cited by Somayaji's dissertation as being necessary to defeat certain classes of attacks (e.g. buffer overflows for shell code execution). With `bpf_override_return`, ebpH can issue targeted error injections one of the helper functions used by `execve(2)`-family calls to load binaries.

By combining the above two techniques, it will be possible to convert ebpH into a fully functional intrusion prevention system, like its predecessor. Signals can be used to implement process delays and targeted error injections can be used to implement `execve(2)` abortion. With these two changes, ebpH's functionality will become a superset of the original pH's, which will facilitate direct comparison between the two systems when conducting a security analysis (c.f. Section 5.2.3).

5.2.3 Security Analysis

In order to measure ebpH's effectiveness at detecting and (in future versions) mitigating attacks, it is necessary to conduct a thorough security analysis.

5.2.4 Refactoring Profile and Process Hashmaps to Other Map Types

As discussed in previous sections, ebpH is not as memory-efficient as its predecessor due to implementation details with how it stores profile and process data. Currently, ebpH uses two ordinary hashmaps to store profiles and process information, which are created with a special flag that signals the kernel to dynamically allocate them rather than preallocate. While this saves on the overhead of allocating all profiles and processes beforehand (which would be prohibitively expensive), there are several problems with this design choice. In particular, known issues with dynamic map allocation may cause deadlocks under conditions with high event frequency [56] and the granularity of allocation is too large to efficiently store the large sparse data structures required by ebpH to manage lookahead pairs in profile data.

To that end, I plan to make the following changes in a future iteration of ebpH:

1) Refactor all hashmaps into the `LRU_HASH` type;
2) Refactor profile data storage to use a `HASH_OF_MAPS`.

Refactoring to Use `LRU_HASH`. The `LRU_HASH` [16, 20] is an eBPF map type of size n that keeps n entries preallocated at all times. When a BPF program attempts to add an entry to a full `LRU_HASH`, it discards the least recently used data from the map to make room for the new entry. While this behavior may not seem ideal, it serves as a reasonable compromise between the current dynamic map allocation approach that may cause deadlocks and a preallocation approach that would be infeasible due to memory restrictions. With an `LRU_HASH`, the size of the preallocated map can be a fraction of the size of the current maps that ebpH uses. For instance, ebpH's process map has 4,194,304 entries by default, one for each possible thread ID on the system, to ensure that new processes will always have space in the map. With an `LRU_HASH`, this would no longer be needed, as adding a new process to the map would simply cause ebpH to forget about the least recently used process. Even using the generous default map size of 10,240 entries would represent a 99.7% reduction in map size, which would in turn reduce the overhead of preallocating the entire map significantly.

Refactoring to Use `HASH_OF_MAPS`. `HASH_OF_MAPS` [16, 20] is a type of map-in-map data structure that allows BPF programs to define and store maps inside of other maps. While support for this was added in 2017 to Linux 4.12 [28], `HASH_OF_MAPS` and `ARRAY_OF_MAPS` have only been supported in bcc [20] since a November 2019 patch [51]. With map-in-map support, ebpH can redefine the way it stores profile data, significantly reducing the granularity of profile data allocation. In particular, lookahead pair data can be stored in a per-profile two-layer hashmap, indexed by two keys: current and previous system call. This means that ebpH would only need to allocated the lookahead pairs that are currently in use by a given profile, which would save significantly on memory overhead compared to the current design.

5.2.5 Reintroducing the ebpH GUI and Conducting a Usability Study

5.2.6 General System Introspection and the Future of ebpH

6 Conclusion

References

[1] W. A. Amai, E. A. Walther, and A. B. Somayaji, "An Immunological Basis for High-Reliability Systems Control," Sandia National Laboratories, Tech. Rep., Mar. 2005.

[2] J. P. Anderson, "Computer Security Technology Planning Study," US Air Force, Tech. Rep., Oct. 1972. [Online]. Available: http://seclab.cs.ucdavis.edu/projects/history/CD/ande72a.pdf.

[3] P. Barham, B. Dragovic, K. Fraser, *et al.*, "Xen and the Art of Virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003, ISSN: 0163-5980. DOI: 10.1145/1165389.945462. [Online]. Available: https://doi-org.proxy.library.carleton.ca/10.1145/1165389.945462.

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

[5] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04, Boston, MA: USENIX Association, 2004, pp. 2–2. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix04/tech/general/full_papers/cantrill/cantrill.pdf.

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

[7] H. Chen, Y. Mao, X. Wang, *et al.*, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, ser. APSys '11, Shanghai, China: Association for Computing Machinery, 2011, ISBN: 9781450311793. DOI: 10.1145/2103799.2103805. [Online]. Available: https://doi-org.proxy.library.carleton.ca/10.1145/2103799.2103805.

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

[9] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 3–14, Oct. 1997, ISSN: 0146-4833. DOI: 10.1145/263109.263133. [Online]. Available: https://doi-org.proxy.library.carleton.ca/10.1145/263109.263133.

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

[11] 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.

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

[13] 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.

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

[15] 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.

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

[17] B. Gregg, J. Mauro, and B. M. Cantrill, *DTrace: dynamic tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall, 2014.

[18] 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*, ser. CoNEXT '18, Heraklion, Greece: ACM, 2018, pp. 54–66, ISBN: 978-1-4503-6080-7. DOI: 10.1145/3281411.3281443. [Online]. Available: http://doi.acm.org/10.1145/3281411.3281443.

[19] A. Hussain, J. Heidemann, J. Heidemann, and C. Papadopoulos, "A Framework for Classifying Denial of Service Attacks," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '03, Karlsruhe, Germany: ACM, 2003, pp. 99–110, ISBN: 1-58113-735-4. DOI: 10.1145/863955.863968. [Online]. Available: http://doi.acm.org.proxy.library.carleton.ca/10.1145/863955.863968.

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

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

[22] IOVisor, *Syscount.py*, Oct. 2019. [Online]. Available: https://github.com/iovisor/bcc/blob/master/tools/syscount.py.

[23] T. Jaeger, *Operating System Security*. Morgan & Claypool Publishers, 2008.

[24] R. A. Kemmerer and G. Vigna, "Intrusion Detection: A Brief History and Overview," *IEEE Security and Privacy 2002*, vol. 35, no. 4, pp. 27–30, Apr. 2002, ISSN: 1558-0814. DOI: 10.1109/MC.2002.1012428.

[25] 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.

[26] F. Kerschbaum, E. Spafford, and D. Zamboni, "Using Internal Sensors and Embedded Detectors for Intrusion Detection," *Journal of Computer Security*, vol. 10, pp. 23–70, Jan. 2002. DOI: 10.3233/JCS-2002-101-203.

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

[28] M. K. Lau, *bpf: Add hash of maps support*, Mar. 2017. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bcc6b1b7ebf857a9fe56202e2be3361131588c15.

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

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

[31] J. McCormack, P. Karlton, S. Angebranndt, *et al.*, *x11perf(1)*, Xorg. [Online]. Available: https://www.x.org/releases/X11R7.7/doc/man/man1/x11perf.1.xhtml.

[32] L. W. McVoy, *lmbench*, Intel, Dec. 2019. [Online]. Available: https://github.com/intel/lmbench.

[33] L. W. McVoy, C. Staelin, *et al.*, "lmbench: Portable tools for performance analysis," in *USENIX annual technical conference*, USENIX, San Diego, CA, USA, 1996, pp. 279–294.

[34] 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/.

[35] J. Mogul, R. Rashid, and M. Accetta, "The Packer Filter: An Efficient Mechanism for User-level Network Code," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, ser. SOSP '87, Austin, Texas, USA: ACM, 1987, pp. 39–51, ISBN: 0-89791-242-X. DOI: 10.1145/41457.37505. [Online]. Available: http://doi.acm.org/10.1145/41457.37505.

[36] M. Moraes and J. McCormack, *x11perfcomp(1)*, Xorg. [Online]. Available: https://www.x.org/releases/X11R7.7/doc/man/man1/x11perfcomp.1.xhtml.

[37] J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *USENIX Security Symposium*, ACM Berkeley, CA, 2002, pp. 17–31.

[38] *NIT(4p) SunOS 4.1.1 Reference Manual*, Sun Microsystems Inc., Sep. 1990.

[39] P. C. van Oorschot, *Computer Security and the Internet: Tools and Jewels*, Sep. 2019. [Online]. Available: https://people.scs.carleton.ca/~paulv/toolsjewels.html (visited on 01/12/2020).

[40] W. W. Peng and D. R. Wallace, *Software Error Analysis*. Silicon Press, 1995.

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

[42] 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.

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

[44] 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.

[45] *select(2) Linux Programmer's Manual*, Linux, Nov. 2019.

[46] *signal(7) Linux Programmer's Manual*, Linux, Aug. 2019.

[47] *sigreturn(2) Linux Programmer's Manual*, Linux, Sep. 2017.

[48] S. Soltesz, H. Pötzl, M. E. Fiuczynski, *et al.*, "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, Lisbon, Portugal: Association for Computing Machinery, 2007, pp. 275–287, ISBN: 9781595936363. DOI: 10.1145/1272996.1273025. [Online]. Available: https://doi-org.proxy.library.carleton.ca/10.1145/1272996.1273025.

[49] 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.

[50] A. B. Somayaji and H. Inoue, "Lookahead Pairs and Full Sequences: A Tale of Two Anomaly Detection Methods," in *Proceedings of the 2nd Annual Symposium on Information Assurance Academic track of the 10th Annual 2007 NYS Cyber Security Conference*. NYS Cyber Security Conference, 2007, pp. 9–19. [Online]. Available: http://people.scs.carleton.ca/~soma/pubs/inoue-albany2007.pdf.

[51] Y. Song, *bpf: Add hash of maps support*, Nov. 2019. [Online]. Available: https://github.com/iovisor/bcc/commit/149c1c8857652997622fc2a30747a60e0c9c17dc.

[52] E. H. Spafford and D. Zamboni, "Intrusion Detection Using Autonomous Agents," *Comput. Netw.*, vol. 34, no. 4, pp. 547–570, Oct. 2000, ISSN: 1389-1286. DOI: 10.1016/S1389-1286(00)00136-5. [Online]. Available: http://dx.doi.org/10.1016/S1389-1286(00)00136-5.

[53] W. Stallings and L. Brown, *Computer security: principles and practice*. [Online]. Available: http://www.informit.com/articles/article.aspx?p=782118.

[54] 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.

[55] 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.

[56] A. Starovoitov, *bpf: hash map pre-alloc*, Mar. 2016. [Online]. Available: https://lwn.net/Articles/679074/.

[57] 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.

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

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

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

[61] *system(3) Linux Programmer's Manual*, Linux, Mar. 2019.

[62] L. Torvalds, *Linux/uapi/asm-generic/unistd.h*. [Online]. Available: https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/unistd.h.

[63] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, Jan. 1937, ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230. eprint: http://oup.prod.sis.lan/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf. [Online]. Available: https://doi.org/10.1112/plms/s2-42.1.230.

[64] 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/.

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

[66] W. Zhang, *[bpf-next] bpf: add new helper fd2path for mapping a file descriptor to a pathname*, Oct. 2019. [Online]. Available: https://patchwork.ozlabs.org/patch/1179287/.

Appendices

A eBPF Design Patterns

Listing A.1: Handling large datatypes 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;
}
```

B bpfbench Source Code

Listing B.1: The eBPF component of bpfbench.

```c
/* bpfbench  A better benchmarking tool written in eBPF.
 * Copyright (C) 2020  William Findlay
 *
 * Heavily inspired by syscount from bcc-tools:
 * https://github.com/iovisor/bcc/blob/master/tools/syscount.py
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <https://www.gnu.org/licenses/>. */

#include <uapi/asm/unistd_64.h>

struct intermediate_t
{
    u64 pid_tgid;
    u64 start_time;
};

struct data_t
{
    u64 count;
    u64 overhead;
};

BPF_PERCPU_ARRAY(intermediate, struct intermediate_t, 1);
BPF_PERCPU_ARRAY(syscalls, struct data_t, NUM_SYSCALLS);

TRACEPOINT_PROBE(raw_syscalls, sys_enter)
{
    u64 pid_tgid = bpf_get_current_pid_tgid();

    /* Maybe filter by PID */
    #ifdef TRACE_PID
    if (pid_tgid >> 32 != TRACE_PID)
        return 0;
    #endif

    /* Don’t trace self */
    if (pid_tgid >> 32 == BPFBENCH_PID)
        return 0;

    int zero = 0;
    struct intermediate_t start = {};

    /* Record pit_tgid of initiating process */
    start.pid_tgid = pid_tgid;
    /* Record start time */
    start.start_time = bpf_ktime_get_ns();

    /* Pass data off to sys_exit */
    intermediate.update(&zero, &start);

    return 0;
}

TRACEPOINT_PROBE(raw_syscalls, sys_exit)
{
    u64 pid_tgid = bpf_get_current_pid_tgid();

    /* Maybe filter by PID */
    #ifdef TRACE_PID
    if (pid_tgid >> 32 != TRACE_PID)
        return 0;
    #endif

    /* Don't trace self */
    if (pid_tgid >> 32 == BPFBENCH_PID)
        return 0;

    int zero = 0;
    int syscall = args->id;

    /* Discard restarted syscalls due to system suspend */
    if (args->id == __NR_restart_syscall)
    {
        return 0;
    }

    struct data_t *data = syscalls.lookup(&syscall);
    struct intermediate_t *start = intermediate.lookup(&zero);
    if (start && data)
    {
        /* We don't want to count twice for calls that return in two places */
        if (pid_tgid != start->pid_tgid)
        {
            return 0;
        }
        data->count++;
        data->overhead += bpf_ktime_get_ns() - start->start_time;
    }

    return 0;
}
```

C Full Macro-Benchmarking Datasets