

# HOST-BASED ANOMALY DETECTION WITH EXTENDED BPF

COMP4906 HONOURS THESIS

**William Findlay**

April 4, 2020



**Carleton**  
**UNIVERSITY**

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 [59, 60] along with the recent development of more usable interfaces such as the *BPF Compiler Collection* (*bcc*) [25] 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 [54] *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<sup>1</sup> and Teng Qin<sup>2</sup> for their guidance and willingness to respond to issues and questions related to the *bcc* project. Brendan Gregg's<sup>3</sup> writings and talks have been a great source of inspiration, especially with respect to background research. Sasha Goldshtein's<sup>4</sup> *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.

---

<sup>1</sup><https://github.com/yonghong-song>

<sup>2</sup><https://github.com/palmtenor>

<sup>3</sup><https://github.com/brendangregg> and <http://www.brendangregg.com>

<sup>4</sup><https://github.com/goldshtn>

# Contents

<b>1</b>	<b>Introduction and Motivation</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>2</b>
2.1	An Overview of the Linux Tracing Landscape . . . . .	2
2.1.1	Dtrace . . . . .	5
2.2	Classic BPF . . . . .	7
2.3	eBPF: Linux Tracing Superpowers . . . . .	8
2.3.1	How eBPF Works at a High Level . . . . .	9
2.3.2	Tracepoints, Kprobes, and Up probes . . . . .	11
2.3.3	The Verifier: The Good, the Bad, and the Ugly . . . . .	12
2.4	System Calls . . . . .	13
2.5	Intrusion Detection . . . . .	15
2.5.1	A Survey of Data Collection in Intrusion Detection Systems . . . . .	17
2.6	Process Homeostasis . . . . .	21
2.6.1	Anomaly Detection Through Lookahead Pairs . . . . .	21
2.6.2	Homeostasis Through System Call Delays . . . . .	22
2.7	Other Related Work . . . . .	22
<b>3</b>	<b>Implementing ebpH</b>	<b>22</b>
3.1	Why an eBPF Implementation? . . . . .	23
3.2	Architectural Overview . . . . .	24
3.3	Userspace Components . . . . .	25
3.3.1	The ebpH Daemon . . . . .	26
3.3.2	ebph-ps . . . . .	27
3.3.3	ebph-admin . . . . .	28
3.3.4	ebpH Logs . . . . .	29
3.4	ebpH Profiles . . . . .	30
3.4.1	Writing Profiles to Disk and Reading Profiles from Disk . . . . .	32
3.5	Tracing Processes . . . . .	32
3.5.1	Profile Creation and Association . . . . .	34
3.5.2	Profile Association and Sequence Duplication . . . . .	34
3.5.3	Dealing with Signal Handlers and Non-Determinism . . . . .	34
3.5.4	Reaping Processes . . . . .	35
3.6	Training, Testing, and Anomaly Detection . . . . .	36
3.6.1	A Simple Example of ebpH Anomaly Detection . . . . .	36
3.7	Soothing the Verifier . . . . .	38
3.8	Dealing a Lack of Concurrency Control Mechanisms . . . . .	40
<b>4</b>	<b>Measuring ebpH's Overhead</b>	<b>41</b>
4.1	Methodology . . . . .	41
4.1.1	lmbench Micro-Benchmark . . . . .	42
4.1.2	bpfbench Macro-Benchmarks . . . . .	43
4.2	Results . . . . .	44

4.2.1	bronte-lmbench System Latency Micro-Benchmark . . . . .	45
4.2.2	bronte-kernel Kernel Compilation Micro-Benchmark . . . . .	51
4.2.3	bronte-7day . . . . .	52
4.2.4	homeostasis-3day . . . . .	55
4.2.5	arch-3day . . . . .	56
4.2.6	Summary . . . . .	57
4.3	Comparing Results with the Original pH . . . . .	59
<b>5</b>	<b>Discussion</b>	<b>60</b>
5.1	Shortcomings of eBPF . . . . .	60
5.1.1	Lack of Concurrency Control Mechanisms in Tracing Programs . . . .	61
5.1.2	Limited Support for Necessary Kernel Helpers . . . . .	61
5.1.3	Verifier Bugs . . . . .	62
5.1.4	Dropped Perf Buffer Submissions . . . . .	63
5.2	Future Work . . . . .	64
5.2.1	Controlling for Further Sources of Non-Deterministic Behavior . . . .	64
5.2.2	Automating ebpH Response . . . . .	64
5.2.3	Security Analysis . . . . .	65
5.2.4	Refactoring Profile and Process Hashmaps to Other Map Types . . . .	66
5.2.5	Reintroducing the ebpH GUI and Conducting a Usability Study . . . .	67
5.2.6	General System Introspection and the Future of ebpH . . . . .	67
<b>6</b>	<b>Conclusion</b>	<b>67</b>
	<b>References</b>	<b>68</b>
	<b>Appendices</b>	<b>74</b>
<b>A</b>	<b>eBPF Design Patterns</b>	<b>74</b>
<b>B</b>	<b>bpfbench Source Code</b>	<b>75</b>
<b>C</b>	<b>Full Macro-Benchmarking Datasets</b>	<b>78</b>

## List of Figures

2.1	A high level overview of the broad categories of Linux instrumentation . . .	4
2.2	Comparing Dtrace and eBPF functionality with respect to API design . . . .	6
2.3	A high level overview of various eBPF use cases . . . . .	8
2.4	Basic topology of eBPF with respect to userland and the kernel . . . . .	10
2.5	The set participation of valid C and eBPF programs. . . . .	14
2.6	Complexity and verifiability of eBPF programs. . . . .	14
2.7	An overview of the basic categories of IDS . . . . .	16
2.8	An overview of the most common data collection categories and subcategories in IDS . . . . .	19
3.1	The architecture of ebpfH . . . . .	25
3.2	Dataflow of a request from ebpfH-admin . . . . .	29
3.3	A sample (read, close) lookahead pair in the ebpfH profile for ls. . . . .	31
3.4	Two sample (write, write) lookahead pairs in the ebpfH profile for anomaly.c	38
4.1	Mean system call times from the bronte-lmbench dataset . . . . .	46
4.2	Mean select(2) times from the bronte-lmbench dataset . . . . .	47
4.3	Mean signal handler times from the bronte-lmbench dataset . . . . .	48
4.4	Mean process creation times from the bronte-lmbench dataset . . . . .	50
4.5	Mean IPC times from the bronte-lmbench dataset . . . . .	51
4.6	Top 20 system call overheads by count in the bronte-7day dataset . . . . .	54
4.7	Top 20 system call overheads by count in the homeostasis-3day dataset . . .	56
4.8	Top 20 system call overheads by count in the arch-3day dataset . . . . .	58
5.1	The instruction flow of ebpfH's sys_exit tracepoint program . . . . .	62

## List of Tables

2.1	A summary of various system introspection technologies available for GNU/Linux systems. . . . .	3
2.2	Various map types available in eBPF programs, as of Linux 5.5 . . . . .	10
3.1	Comparing the current prototype of ebpH with the original pH system . . .	23
3.2	Main perf event categories in ebpH. . . . .	26
3.3	eBPF tracepoints and kprobes used in ebpH. . . . .	33
4.1	Systems used for the collection of ebpH benchmarking data . . . . .	42
4.2	ebpH macro-benchmarking datasets . . . . .	44
4.3	Results of the system call benchmarks from the bronte-1mbench dataset . . .	45
4.4	Results of the <code>select(2)</code> benchmarks from the bronte-1mbench dataset . . .	46
4.5	Results of the signal handler benchmarks from the bronte-1mbench dataset .	48
4.6	Results of the process creation benchmarks from the bronte-1mbench dataset	49
4.7	Results of the IPC benchmarks from the bronte-1mbench dataset . . . . .	50
4.8	Kernel compilation times from the bronte-kernel dataset . . . . .	52
4.9	Top 20 system call overheads by count in the bronte-7day dataset . . . . .	53
4.10	Top 20 system call overheads by count in the homeostasis-3day dataset . . .	55
4.11	Top 20 system call overheads by count in the arch-3day dataset . . . . .	57
C.1	All system call overhead data from the homeostasis-3day dataset. . . . .	78
C.2	All system call overhead data from the arch-3day dataset. . . . .	82

## List of Listings

3.1	Sample output from <code>ebph-ps</code> . . . . .	27
3.2	Sample output from <code>ebph-ps -p</code> . . . . .	28
3.3	A simplified definition of the <code>ebpH</code> profile struct. . . . .	30
3.4	A simplified definition of the <code>ebpH</code> process struct. . . . .	32
3.5	<code>anomaly.c</code> , a simple program to demonstrate anomaly detection in <code>ebpH</code> . . .	36
3.6	The flagged anomaly in the <code>anomaly</code> binary as shown in the <code>ebpH</code> logs . . .	37
A.1	Handling large datatypes in eBPF programs. . . . .	74
B.1	The eBPF component of <code>bpfbench</code> . . . . .	75



# 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 [59, 60], 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 eBPF effectively provides unlimited, highly performant, production-safe system introspection capabilities that can be used to monitor as much or as little system state as desired.

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, powerful applications can be constructed 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 anomaly 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*) [25], 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* [54], 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

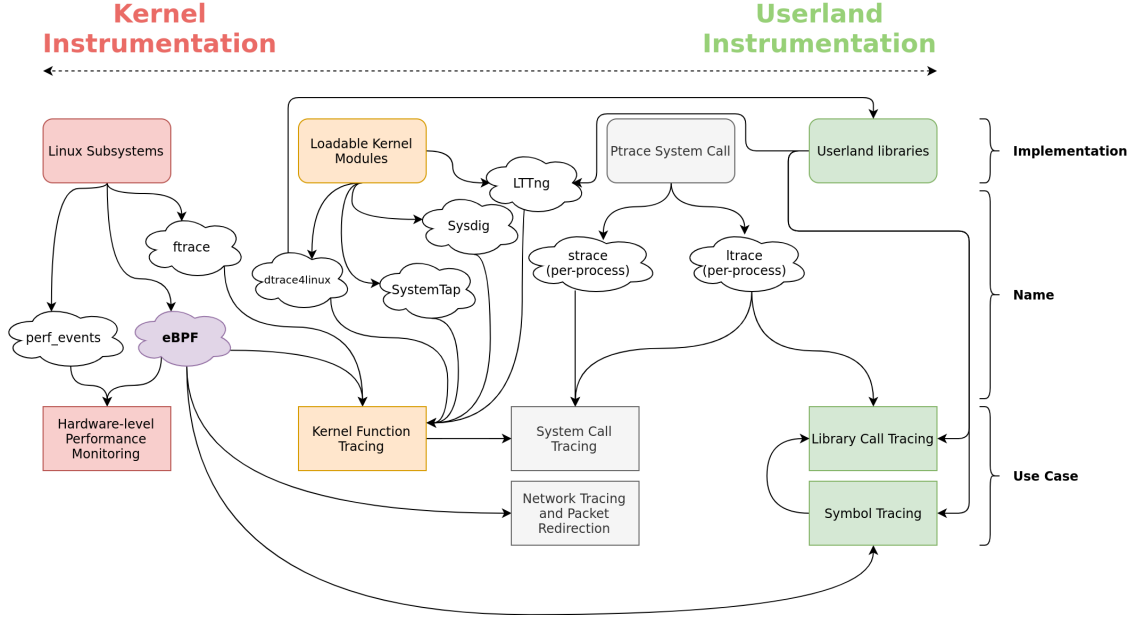
System introspection is hardly a novel concept; for years, developers have been thinking about the best way to solve this problem and have come up with several unique solutions, each with a variety of benefits and drawbacks. [Table 2.1](#) presents an overview of some prominent examples relevant to GNU/Linux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

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

name	Interface and Implementation	Citations
strace	Uses the <code>ptrace</code> system call to trace an individual userspace process	[63, 64]
ltrace	Uses the <code>ptrace</code> system call to trace library calls in an individual userland process	[8, 49]
SystemTap	Dynamically generates loadable kernel modules for instrumentation; newer versions can optionally use eBPF as a backend instead	[40, 47]
ftrace	Sysfs pseudo filesystem for tracepoint instrumentation located at <code>/sys/kernel/debug/tracing</code>	[48]
perf_events	Linux subsystem that collects performance events and returns them to userspace	[71]
LTTng	Loadable kernel modules, userland libraries	[36]
dtrace4linux	A Linux port of DTrace via a loadable kernel module	[16]
sysdig	Loadable kernel modules for system monitoring; native support for containers	[65]
eBPF	In-kernel execution of pre-verified, JIT-compiled bytecode	[18, 25, 59, 60]

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` [63, 64] 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 the user is 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 [46]. 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 [30].

Although library call instrumentation through software such as `ltrace` [8, 49] 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 [65] and SystemTap [47] 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 [59, 60] itself constitutes one such solution. However, for the time being, I will focus on a few others, namely ftrace [48] and perf\_events [71] (eBPF programs actually *can* and *do* use both of these interfaces anyway). While both of these solutions are safe to use (assuming the system can trust the root 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.

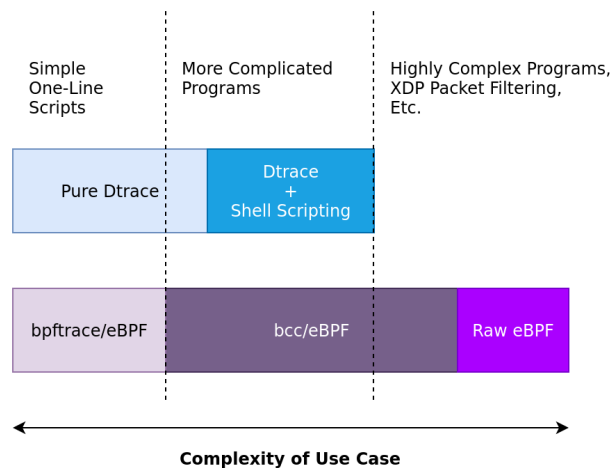
### 2.1.1 Dtrace

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 [7] was

designed in 2004 for Solaris and lives on to this day in several operating systems, including Solaris, FreeBSD, MacOS X [22], and Linux [16] (the `dtrace4linux` implementation will be examined 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 [7, 59, 60]. 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 [21, 22]:

- 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 [20]). 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` [16] is a free and open source port of Sun’s Dtrace [7] 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 [20] 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` [25], 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 `bpftool` [20, 26] which has been designed to provide a high-level Dtrace-like tracing language for Linux using eBPF as a backend. Although `bpftool` only provides a subset of Dtrace’s functionality [20], 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 [69] 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 [37] introduced the original BPF<sup>5</sup> 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 [43] and CSPF [41].

While Classic BPF is certainly a powerful technique for filtering packets, Starovoitov [59, 60] 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.

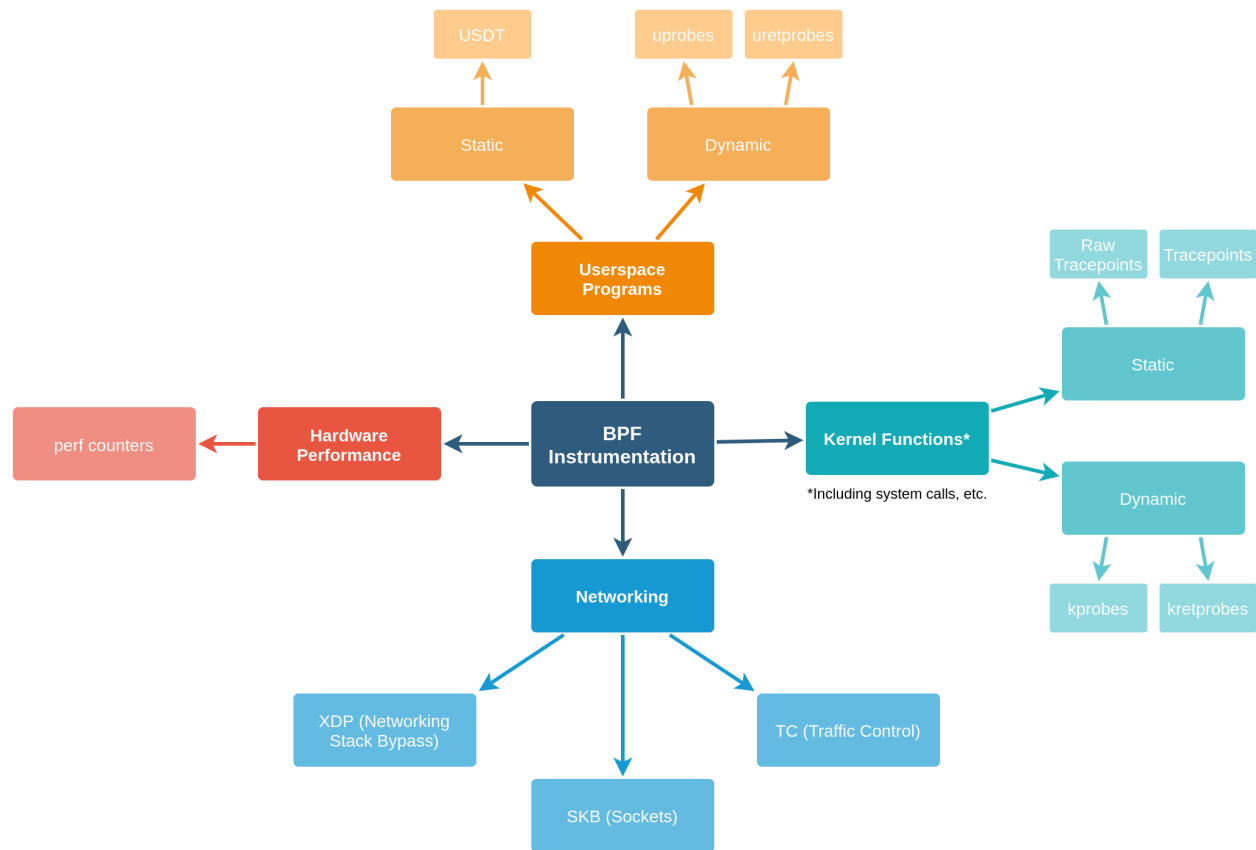
---

<sup>5</sup>Hereafter, I will refer to the original BPF as *Classic BPF* to avoid confusion with eBPF and the BPF programming paradigm.

eBPF is discussed in more detail in the following section.

## 2.3 eBPF: Linux Tracing Superpowers

In 2016, eBPF was described by Brendan Gregg [19] as nothing short of *Linux tracing superpowers*. I echo that sentiment here, as it summarizes eBPF’s capabilities perfectly. Through eBPF programs, one 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* [23], one 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 [62] 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 2.3.3](#).

eBPF’s superior performance can be attributed to several factors. On supported architectures,<sup>6</sup> 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

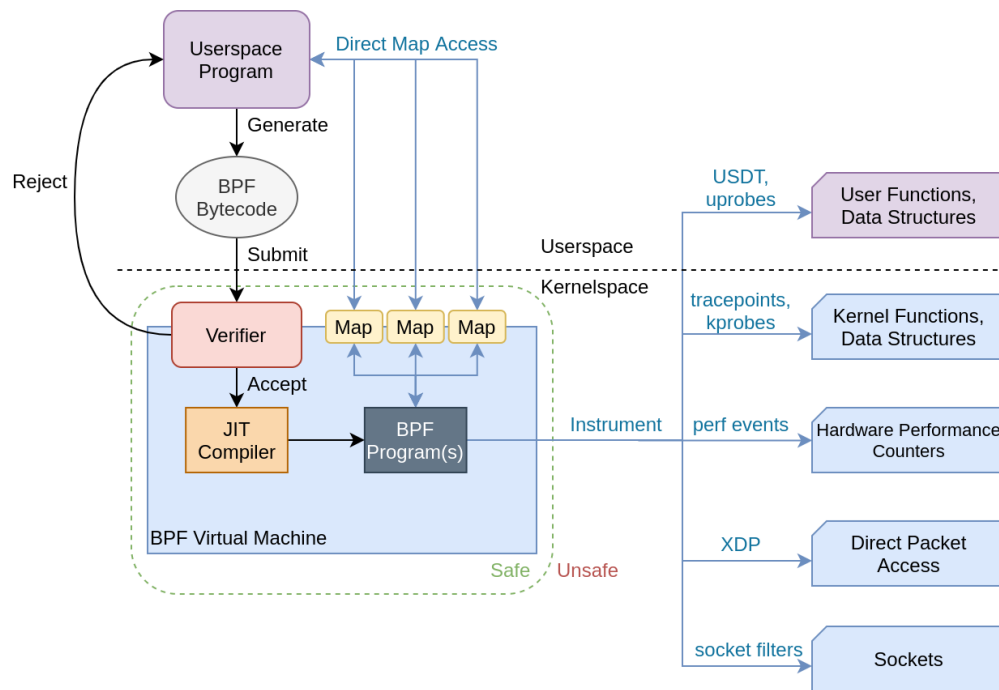
From the perspective of a user, the eBPF workflow is surprisingly simple. Users can elect to write eBPF bytecode directly (not recommended) or use one of many front ends to write in higher level languages that are then used to generate the respective bytecode. IOVisor’s bcc [25] offers bindings for several 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 [6] 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 shown 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

---

<sup>6</sup>x86-64, SPARC, PowerPC, ARM, arm64, MIPS, and s390 [14]



**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 [21, 25] available in eBPF programs, as of Linux 5.5.

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 kernelspace 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 <code>ifindex</code> of various network devices; used in XDP programs
SOCKMAP	Holds references to <code>sock</code> structs; used for socket redirection
CPUMAP	Allows for redirection of packets to remote CPUs; used in XDP programs

a general purpose HASH (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 2.3.3](#).

### 2.3.2 Tracepoints, Kprobes, and Upbes

As shown previously in [Figure 2.3](#) on page 8, eBPF supports a number of distinct program types which may be used to instrument and interact with various aspects of system functionality. ebpfH’s functionality mainly relies on three specific program types: the *tracepoint*, the *kprobe*, and the *uprobe* [21, 25]. Here, I will describe what each program type does and how they work at a high level. These concepts will be revisited frequently when discussing ebpfH’s implementation in [Section 3](#).

**Tracepoints.** Tracepoints [21] define the stable kernel tracing API of eBPF; at a high level, they are predefined sections in kernel code that trap to eBPF handlers when these handlers are defined. Tracepoints are stable in the sense that the information exposed to eBPF by a tracepoint will not be likely to change between kernel versions, which means that they are ideal for use in production where forward compatibility with new versions of Linux is a desirable property. Although using tracepoints is ideal when possible, they have a few caveats; in particular, a limited number of tracepoints are defined by the kernel, and they do not cover an exhaustive list of kernel functionality. Linux 5.5 defines 1,872 tracepoints in total.

ebpfH uses tracepoints to implement the bulk of its kernelspace functionality (c.f. [Section 3.4](#) and [Section 3.5](#)). System call tracepoints are used to track system call sequences for its anomaly detection functionality; scheduler tracepoints are used to maintain the set of traced processes and to associate these processes with the correct profiles.

**Kprobes.** Whereas tracepoints define a stable kernel tracing API, kprobes [21] can be thought of as their dynamic counterparts. Although kprobes may be used to trace any exported kernel function (that is, any kernel function that is not in-lined by the compiler), they are not considered stable as the API of a kprobe changes whenever the corresponding kernel function changes. Thus, tracepoints are preferred when possible and kprobes are generally used as a last resort. Kprobes work by inserting a breakpoint at a specific address in kernel memory; when this breakpoint is hit, the kernel traps to the associated BPF handler. Kprobes can also be used to instrument function returns, in which case they are known as kretprobes.

ebpfH uses only one kretprobe, and does so to keep track of when processes receive a signal

that would trap to a signal handler (c.f. [Section 3.5.3](#)).

**Uprobes.** Uprobes and uretprobes work in a similar fashion to kprobes and kretprobes; the primary difference here is that we are now instrumenting *userspace* rather than *kernelspace*. A breakpoint is inserted at the target userspace address and this breakpoint traps to the appropriate BPF handler.

ebpH uses uprobes to implement sending complex commands to the BPF program from a userspace library (c.f. [Section 3.3.3](#)).

### 2.3.3 The Verifier: The Good, the Bad, and the Ugly

The verifier is responsible for eBPF’s unprecedented safety given its scope, one of its most attractive qualities with respect to system tracing. While this verifier is quintessential to the safety of eBPF, 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 is actually provable; a famous example is the halting problem described by Turing circa 1937 [68]. This difficulty is further compounded by stricter requirements for safety in the context of an eBPF program; in particular, developers don’t want BPF programs to crash or otherwise damage the kernel [14].

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 [24] 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* [9] (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 I 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 the verifier 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<sup>7</sup> 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 developers 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, eBPF limits stack allocations to only 512 bytes [21] – 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.

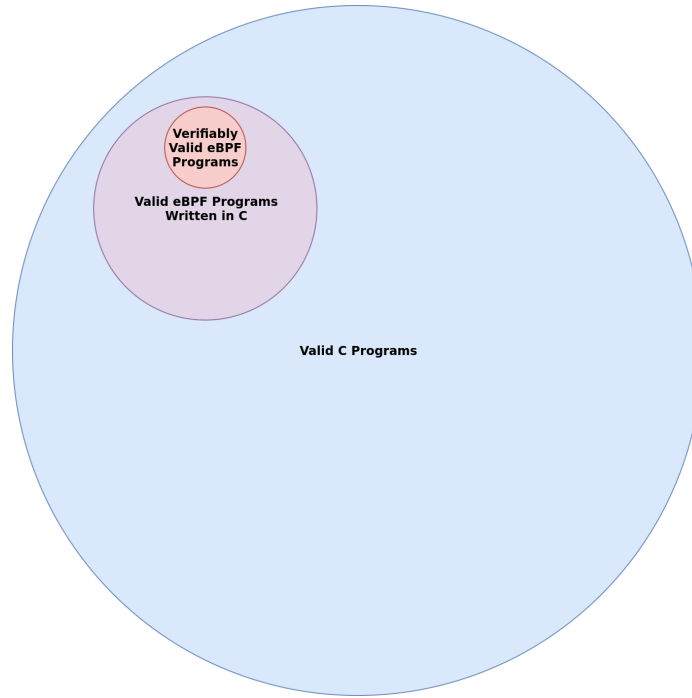
Fortunately, the eBPF verifier is getting better over time (Figure 2.6). When I say *better*, what I mean is that it is able to prove the safety of increasingly complex programs. Perhaps the best example of this steady improvement is a recent kernel patch [62] that added support for bounded loops in eBPF programs. With this patch, the set of viable eBPF programs was *greatly* increased; in fact, ebpfH 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

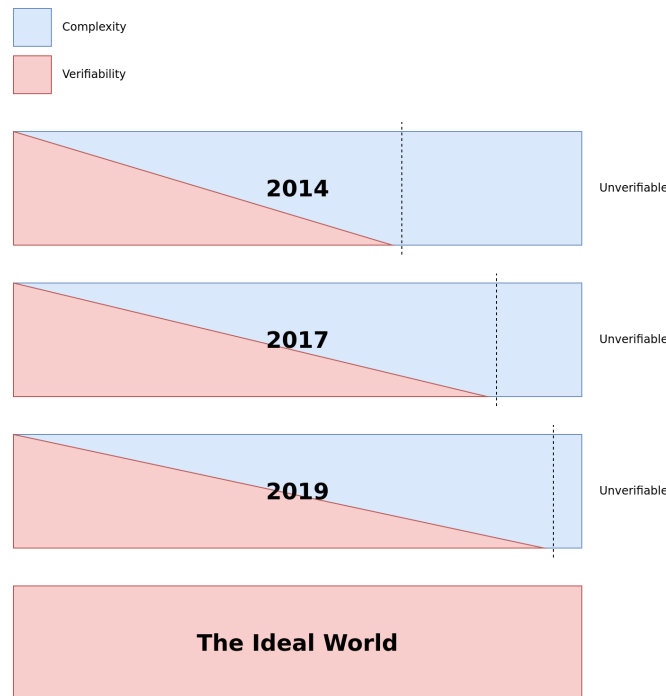
ebpfH (and the original pH system upon which it is based) works by instrumenting *system calls* in order to establish behavioral patterns for all binaries running on the system. Understanding pH and ebpfH requires a reliable mental model of what a system call is and how programs use them to communicate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

---

<sup>7</sup>Easier here means *computationally easier*, certainly not trivial.



**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 the relationship would be as shown on the bottom; in practice, the relationship is getting closer over time, but is still far from the ideal.

At the time of writing this thesis, the Linux Kernel [67] 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 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.

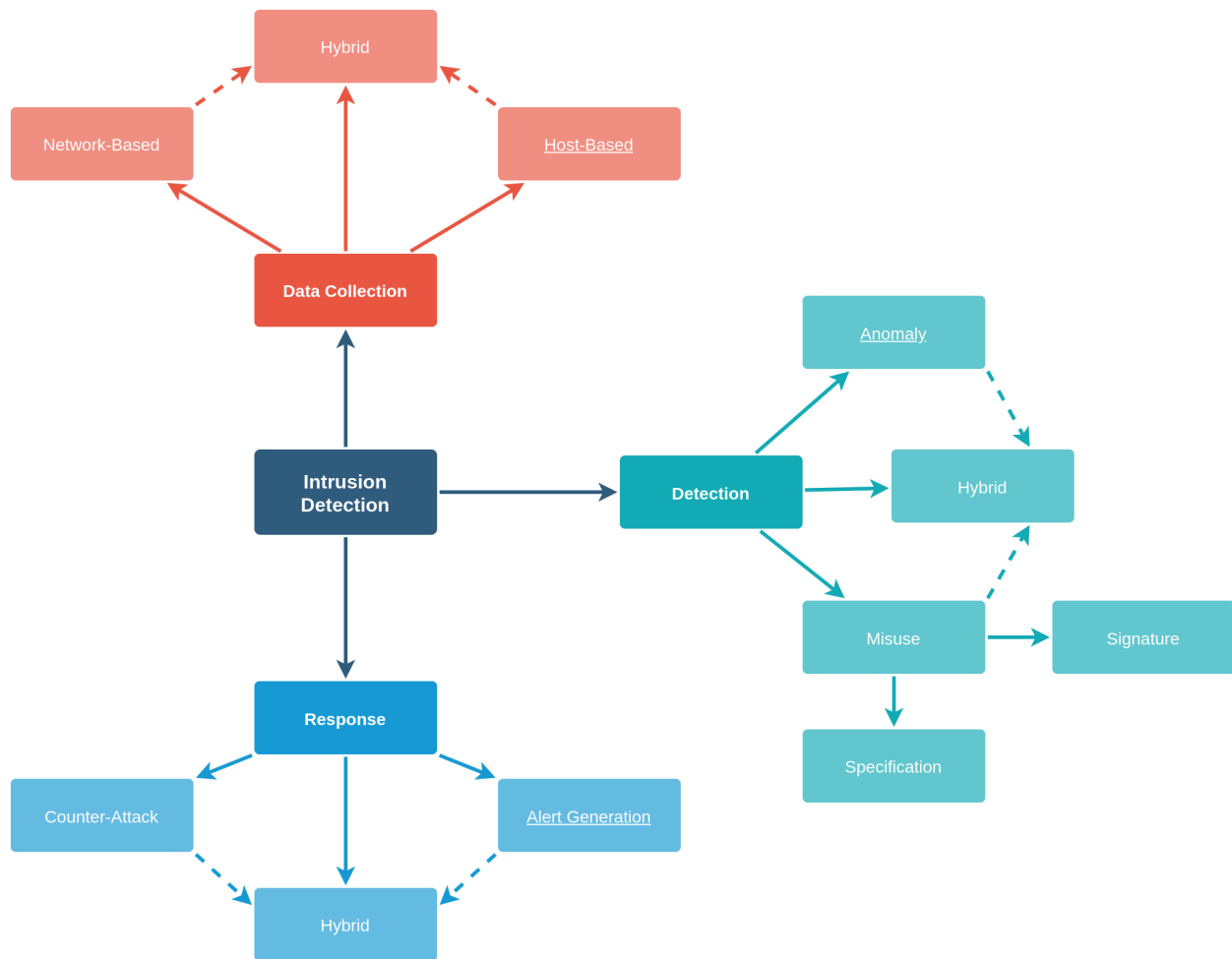
Critically, from a security perspective, system calls provide the interface to the kernel’s *reference monitor* [2, 28, 44], an abstraction that refers to the kernel’s facilities for mediating access by subjects (i.e. users and their processes) onto system objects (i.e. security-sensitive resources). 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

The concept of intrusion detection has seen prevalent attention in academic work since the early 1980’s [3, 12, 13]. 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 [29]. 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 [29, 44]. In general, anomaly-based approaches cover a wider variety of attacks while misuse-based approaches



**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.



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 [29, 54].

Responses can vary significantly based on the system, but can be classified into two main 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 [29]. 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 [54]. Implementing this functionality in ebpH is a topic for future work (c.f. Section 5.2.2), and currently ebpH only supports the anomaly detection aspect of its predecessor.

### 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. [31] 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 [31]. In 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 [31] 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<sup>8</sup>, easy to update and modify, and has guaranteed performance and safety due to the verifier.

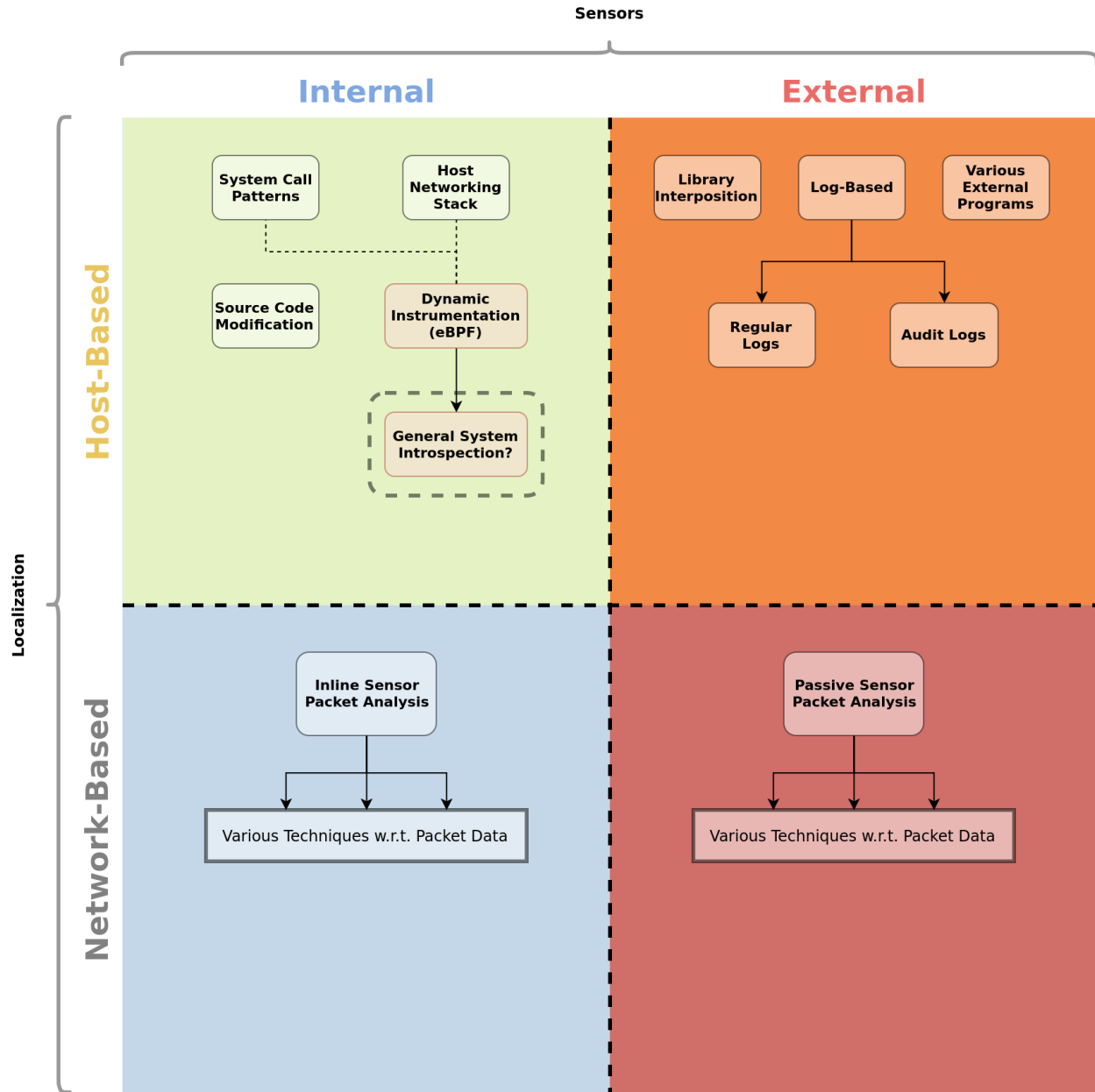
**Internal Host-Based Approaches.** System call pattern analysis was examined in detail by Forrest et al. [15] and culminated in the development of the original pH system [54] on which ebpH is based. Somayaji and Inoue [55] compared two methods of organizing system call data for intrusion detection (full 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 [31]. However, the primary caveat here is that such checks need to be integrated into a specific application early enough such that refactoring is minimized [31, 45]. 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 [37] and eBPF/XDP [23, 25, 59, 60] are quite excellent at this. Host-based network introspection allows the analysis of network traffic at various points in the kernel’s

---

<sup>8</sup>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 2.3.3](#) for more details.



**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 [31, 58].

networking stack, and XDP packet redirection [23] 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.

**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. [31], particularly with respect to ease of implementation and portability.

AAFID [57] 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 [31].

In 1999, Kuperman and Spafford [32] 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 [58] 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 [58]. 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 [58]. This result is consistent with Kerschbaum et al.’s observation that external sensor approaches tend to be favored over their internal counterparts [31].

## 2.6 Process Homeostasis

Anil Somayaji’s *Process Homeostasis* [54], 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 [54]. 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* [54, 55] 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 [55] 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 [55].

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 [54]. 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 [54].

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 [54] 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).

## 2.7 Other Related Work

## 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 reimplement of the original pH system described by Somayaji [54]. 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 [\[54\]](#) 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.

**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 <a href="#">[54]</a>	Kernel Patch	✗	✗	✓	✓	✓	✓
ebpH	eBPF + Userspace Daemon	✓	✓	✗	✓	✓	✗

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, 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  $\geq 5.3$  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](#).

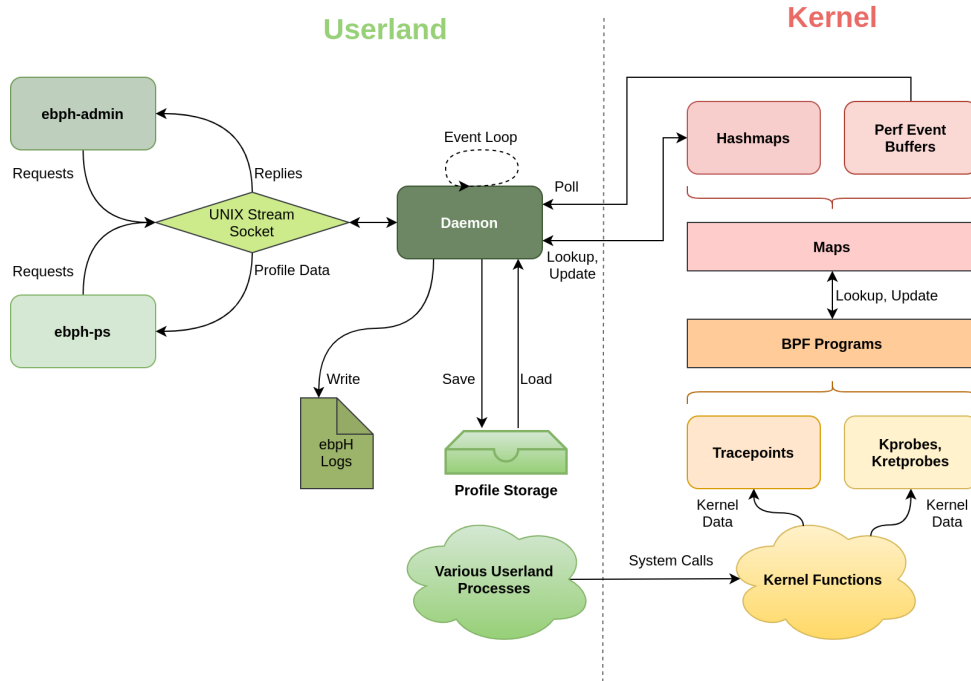
Since eBPF disallows global variables in the traditional sense, data storage and communication between BPF programs needs to occur through the variety of maps. Further, limitations on memory allocation and access restrict the dynamic allocation of data. To cope with these restrictions, the current version 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 [54].

## 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 [54].





**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 [25] 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 <sup>9</sup>
ebpH_on_executable_processed	Reports when a profile has been created	2 <sup>8</sup> pages
ebpH_on_anomaly	Reports anomalies in specific processes and which profile they were associated with	2 <sup>8</sup> pages
ebpH_on_anomaly_limit	Reports when a profile hits its anomaly limit	2 <sup>8</sup> pages

<code>ebpH_on_tolerize_limit</code>	Reports when a process hits its tolerize limit	2 <sup>8</sup> pages
<code>ebpH_on_start_normal</code>	Reports when a profile starts normal monitoring	2 <sup>8</sup> pages
<code>ebpH_on_new_sequence</code>	Reports new sequences for logging (when enabled)	2 <sup>8</sup> pages
<code>ebpH_warning</code>	Reports generic warnings	2 <sup>2</sup> pages
<code>ebpH_error</code>	Reports generic errors	2 <sup>2</sup> 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	alacrity	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.

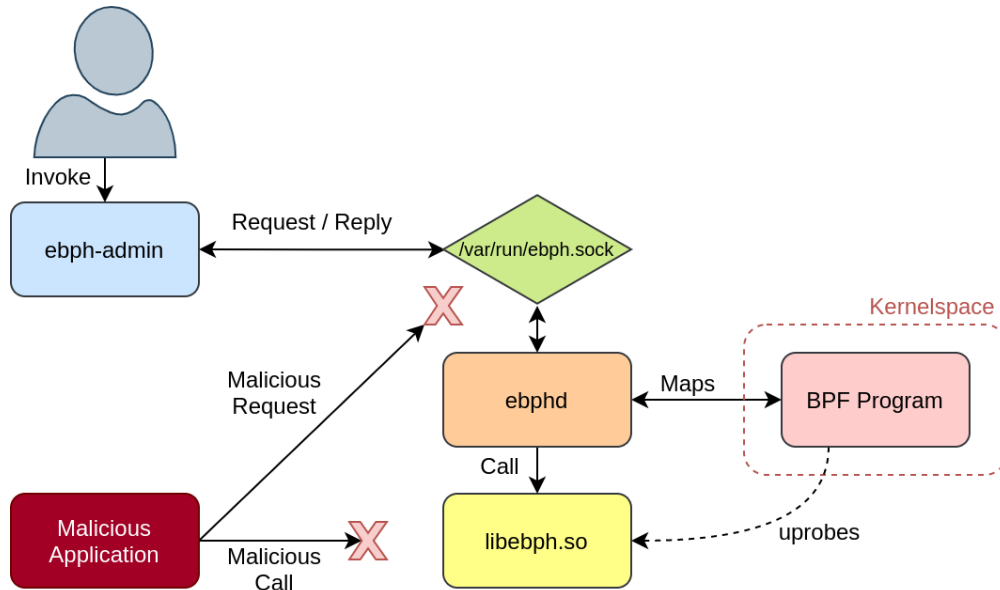
KEY	COMM	STATUS	TRAIN_COUNT	LAST_MOD	ANOMALIES	NORMAL	TIME
5259631	systemd-sleep	Normal	19039	0	0	2020-03-03	17:50:58
5259625	systemd-sleep	Training	1274	422	32	2020-03-08	17:51:52
5259628	systemd-sysctl	Training	554	74	0	2020-02-25	18:41:25
5259377	systemd-tmpfi...	Training	29199	5014	0	2020-02-26	20:34:00
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.



**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 26, 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 ebp<sub>H</sub>; 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 ebp<sub>H</sub> 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 ebp<sub>H</sub>'s GUI.

### 3.4 ebp<sub>H</sub> Profiles

In order to monitor process behavior, ebp<sub>H</sub> 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:

$$\text{key} = (\text{device number} \ll 32) + \text{inode number}$$

where  $\ll$  is the left bitshift operation. In other words, ebp<sub>H</sub> 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 ebp<sub>H</sub> profile struct.

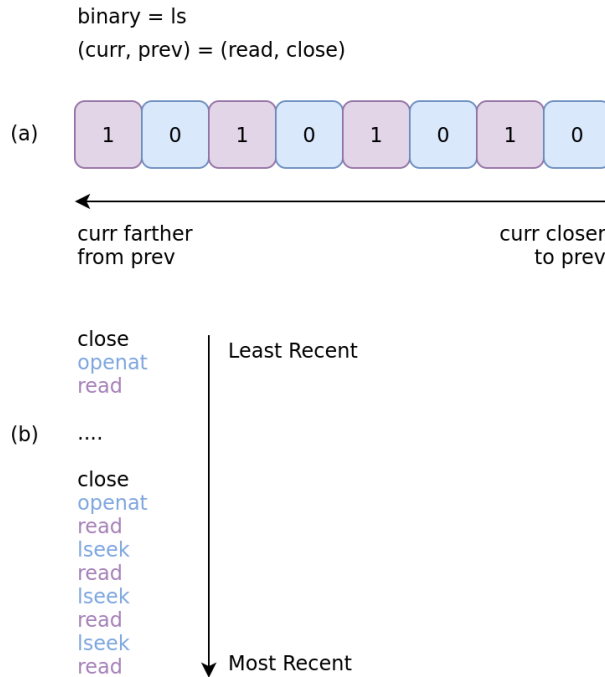
```

1  struct ebpH_profile_data
2  {
3      u8 flags[SYSCALLS][SYSCALLS]; /* System call lookahead pairs */
4      u64 last_mod_count; /* Syscalls since profile was last modified */
5      u64 train_count; /* Syscalls seen during training */
6  };
7
8  struct ebpH_profile
9  {
10     struct ebpH_profile_data train; /* Training data */
11     struct ebpH_profile_data test; /* Testing data */
12     u8 frozen; /* Is the profile frozen? */
13     u8 normal; /* Is the profile normal? */
14     u64 normal_time; /* Minimum system time required for normalcy */
15     u64 anomalies; /* Number of anomalies in the profile */
16     char comm[128]; /* Name of the executable file */
17 };

```

The profile itself is a C data structure that keeps track of information about the executable

as well as two copies of profile data, one for training, and one for testing. This profile data consists of a sparse two dimensional array of lookahead pairs [54, 55] used to keep track of observed system call patterns. Each entry in this array consists of an 8-bit integer, with the  $i^{\text{th}}$  bit corresponding to a previously observed distance  $i$  between the two calls. When ebp<sub>H</sub> observes this distance, it sets the corresponding bit to 1 using a bitwise OR operation. Otherwise, it remains 0. Each profile maintains lookahead pairs for each possible pair of system calls, and these lookahead pairs are checked against new sequences when a profile becomes normal. 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 ebp<sub>H</sub> 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 ebp<sub>H</sub> observes a process making a call to `execve`. Whenever a process makes a system call, ebp<sub>H</sub> 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 ebp<sub>H</sub> is able to monitor process behavior.

Just like in the original pH [54], 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, ebp<sub>H</sub> 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 ebp<sub>H</sub> first loads.

In the original pH, profile data was saved and loaded in kernelspace [54] which meant that it required kernelspace file I/O, which is often regarded as an unsafe practice. ebp<sub>H</sub> 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 ebp<sub>H</sub> process struct.

**Listing 3.4:** A simplified definition of the ebp<sub>H</sub> process struct.

```

1  struct ebpH_sequence
2  {
3      long seq[9];          /* Remember 9 most recent system calls in order */
4      u8 count;             /* How many system calls are in our sequence? */
5  };
6
7  struct ebpH_sequence_stack
8  {
9      ebpH_sequence[3]; /* Keep track of up to 3 sequences at a time */
10     int top;           /* Top of the sequence stack, values from 0-2 */
11     int should_pop;    /* Pop from the stack on next system call */
12 };
13
14 struct ebpH_process
15 {
16     struct ebpH_sequence_stack;
17     u32 pid;             /* Kernel tgid */

```



```

18     u32 tid;           /* Kernel pid */
19     u64 profile_key;   /* Associated profile key */
20     u8 in_execve;      /* Are we in the middle of an execve? */
21 };

```

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 instruments a few other tracepoints to keep track of profile creation, process creation and deletion, and profile association on exec-family system calls. The sched class of tracepoints exposes hooks on the necessary system functionality 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

<sup>9</sup>The majority of these values are subject to significant optimization in future iterations of ebpH. The 2<sup>8</sup> value is a sensible default chosen by bcc. In practice, many of these events are infrequent enough that smaller buffer sizes would be sufficient.

### 3.5.1 Profile Creation and Association

There are several important considerations here. First, ebp<sub>H</sub> 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 provided by the `linux_binprm` struct, which is used to store information about the executable or interpreted script that `execve(2)` has loaded. In particular, the executable's inode and filesystem device number are used in combination to compute a key that uniquely maps to an individual executable on disk. Without this, ebp<sub>H</sub> 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, ebp<sub>H</sub> 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 [54].

### 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, ebp<sub>H</sub> 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 ebp<sub>H</sub> 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 ebp<sub>H</sub> 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 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 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.

ebp<sub>H</sub> 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, ebp<sub>H</sub> pushes a frame onto this stack, and when the process exits from its signal handler, ebp<sub>H</sub> pops the frame. This has the effect of temporarily wiping ebp<sub>H</sub>'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, ebp<sub>H</sub> 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, ebp<sub>H</sub> detects a return from a signal handler by checking for the `rt_sigreturn` system call; when ebp<sub>H</sub> detects such a return, it pops the top frame from the sequence stack.

### 3.5.4 Reaping Processes

ebp<sub>H</sub> reaps tasks from its process map whenever detects that they have exited. By reaping process structs from our map as ebp<sub>H</sub> 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, ebp<sub>H</sub> 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, ebp<sub>H</sub> simply determines the task's PID and deletes that key from the process map.

## 3.6 Training, Testing, and Anomaly Detection

ebp<sub>H</sub> 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 ebp<sub>H</sub> to remove the profile's normal flag and return it to training mode.

### 3.6.1 A Simple Example of ebp<sub>H</sub> 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 ebp<sub>H</sub> if the profile has been previously marked normal.

**Listing 3.5:** `anomaly.c`, a simple program to demonstrate anomaly detection in ebp<sub>H</sub>.

```
1  /* anomaly.c */
2
3  #include <stdio.h>
4  #include <unistd.h>
5
6  int main(int argc, char **argv)
7  {
8      /* Execute this fake anomaly
9       * when we provide an argument */
10     if (argc > 1)
11         printf("Oops!\n");
12     /* Say hello */
13     printf("Hello world!\n");
14
15     return 0;
16 }
```

In order to test this, I artificially lower ebp<sub>H</sub>'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

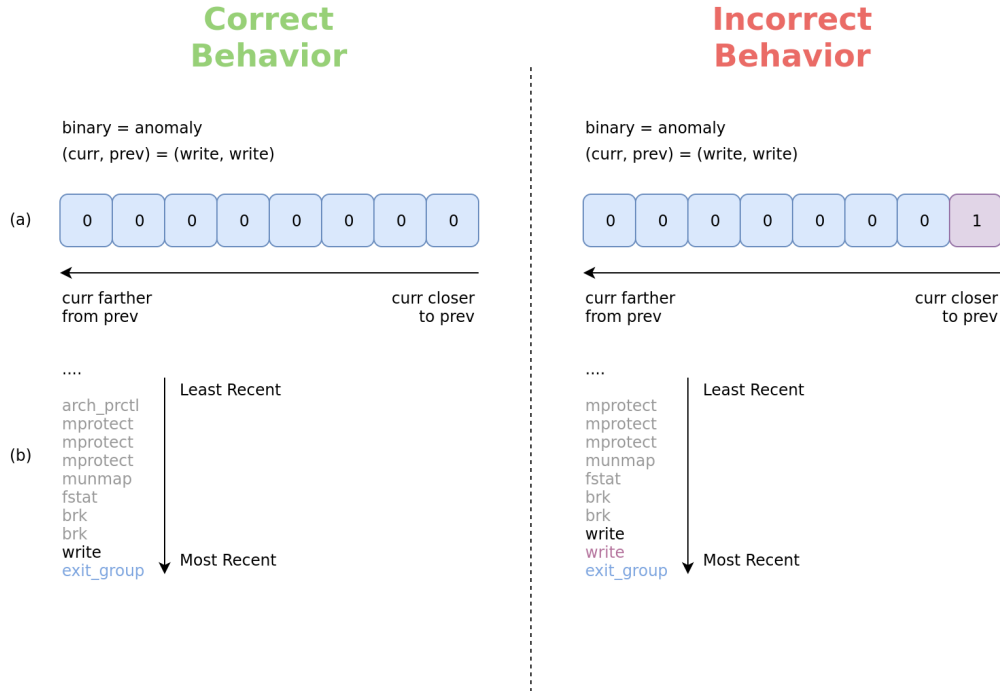
with an argument to produce the anomaly. ebp<sub>H</sub> immediately detects the anomalous system calls and flags them. These anomalies are then reported to userspace via a perf buffer as shown in [Listing 3.6](#).

**Listing 3.6:** The flagged anomaly in the `anomaly` binary as shown in the ebp<sub>H</sub> logs. Note that ebp<sub>H</sub> 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 ebp<sub>H</sub>'s anomaly detection, process behavior in practice is often more nuanced. ebp<sub>H</sub> 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 ebp<sub>H</sub> 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 ebp<sub>H</sub> 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 Soothing the Verifier

The development of ebp<sub>H</sub> elicited many challenges with respect to the eBPF verifier. As seen in [Section 2.3.3](#), 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 [54] 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 [72], 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*<sup>10</sup> 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 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 [61] 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

<sup>10</sup>This is not the same thing as a *pathname*.

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 [10]. 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 [10]. 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 2.3.3](#), 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.

### 3.8 Dealing a Lack of Concurrency Control Mechanisms

Due to a lack of sufficient preemption checks in the verifier [34, 35], tracing programs are currently forbidden from using the `bpf_spin_lock` primitive included in kernel 5.1. This means that ebpH has no means of managing concurrency within its data structures in the traditional sense, and there no immediately obvious way of guaranteeing that modifications to profiles are consistent. Notably, the map used to keep track of processes does not suffer



from this problem, as each `ebpH_process` data structure keeps track of its own thread.

Since profiles may be modified by multiple processes at once, it is important that these modifications be kept relatively synchronized to avoid mismatches. One useful aspect of `ebpH`’s design is that entries within lookahead pairs are tracked with bits, which means that each entry is either 1 or 0 at a given time, and that entries are set using a bitwise OR operation. Since  $a \times |1$  is always equal to 1, the operation of setting a lookahead pair is actually immune to concurrency-related issues. Similarly, lookahead pairs are only checked for anomalies once a profile has been frozen, and this check is done on a separate copy of the training data from the one being modified. This means that the operation of checking a lookahead pair is also safe.

On the other hand, `ebpH` profiles have several flags and counters that need to be kept synchronized in order for them to work properly. Although locking in the traditional sense is impossible, `eBPF` *does* have atomic add and subtract instructions [34] that combine the operation of checking a value with the operation of incrementing or decrementing it. `ebpH` uses these to keep its flags and counters at least semi-consistent with what is expected. Although this is not a perfect solution, it is sufficient as a temporary stopgap until a better alternative is made available. Section 5.1.1 further discusses the need for concurrency control mechanisms in `eBPF`.

## 4 Measuring `ebpH`’s Overhead

One of the primary advantages of `eBPF` is its relatively low overhead [21, 59, 60] 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

The experimental methodology used to determine `ebpH`’s performance overhead includes both macro and micro-benchmarks, to establish both real-world behavior and highly controlled experimental results respectively. Benchmarks were primarily concerned with `ebpH`’s

overhead on system calls, although other factors were considered in the micro-benchmark tests, such as signal handler overhead, IPC (interprocess-communication), and process creation latency. Macro-benchmarking data was collected on various systems under various workloads, including: a server used in production; a personal computer; and a CCSL (Carleton Computer Security Lab) workstation. Micro-benchmarking data was collected on the CCSL workstation only under an otherwise idle workload, in order to prevent corruption of results by outside factors. Table 4.1 summarizes each of the systems used for the collection of benchmarking data, including relevant hardware specifications.

**Table 4.1:** Systems used for the collection of ebp<sub>H</sub> 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

#### 4.1.1 lmbench Micro-Benchmark

McVoy's lmbench [38, 39] is a Linux micro-benchmarking suite that has seen prominent use in academia [4, 11, 42, 53] for establishing various performance metrics of UNIX-like systems. The *OS-category* benchmarks in lmbench are most relevant to ebp<sub>H</sub>'s overhead. This category provides performance metrics such as:

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

- IPC (inter-process communication) latency for pipes and UNIX stream sockets (c.f. [Table 4.7](#) and [Figure 4.5](#)).

Simple system call and `select(2)` latency will give 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.

#### 4.1.2 bpfbench Macro-Benchmarks

Since ebpH's kernelspace functionality resides in system call hooks, its imposed overhead on the system can be established by running macro-benchmarks on the time required to make system calls. Initially, I planned to use `syscount` [27] 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, an ad-hoc benchmarking tool, `bpfbench`<sup>11</sup>, 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 `PERCPU_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.

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 4.2](#) provides a description of each dataset, including the system and the workload tested.

---

<sup>11</sup>Full source code available at <https://github.com/willfindlay/bpfbench>.

**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**.

Many system calls in Linux are designed to wait and return when some property becomes true on a system resource; such calls are referred to as *blocking system calls*. Since many blocking system calls introduce a high degree of variance in results, they have been pruned from the results presented here. In particular, any system call with a standard deviation in runtime higher than 10 microseconds has been removed from the presented results. This is an effective heuristic for weeding out blocking system calls that could impact the integrity of the dataset while preserving those that have acceptable impact on results. Full, unadulterated results are provided in [Appendix C](#).

## 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. Macro-benchmark results have been trimmed for brevity, and pruned for outliers and results with unacceptably high variance. As mentioned, full datasets

are available in [Appendix C](#).

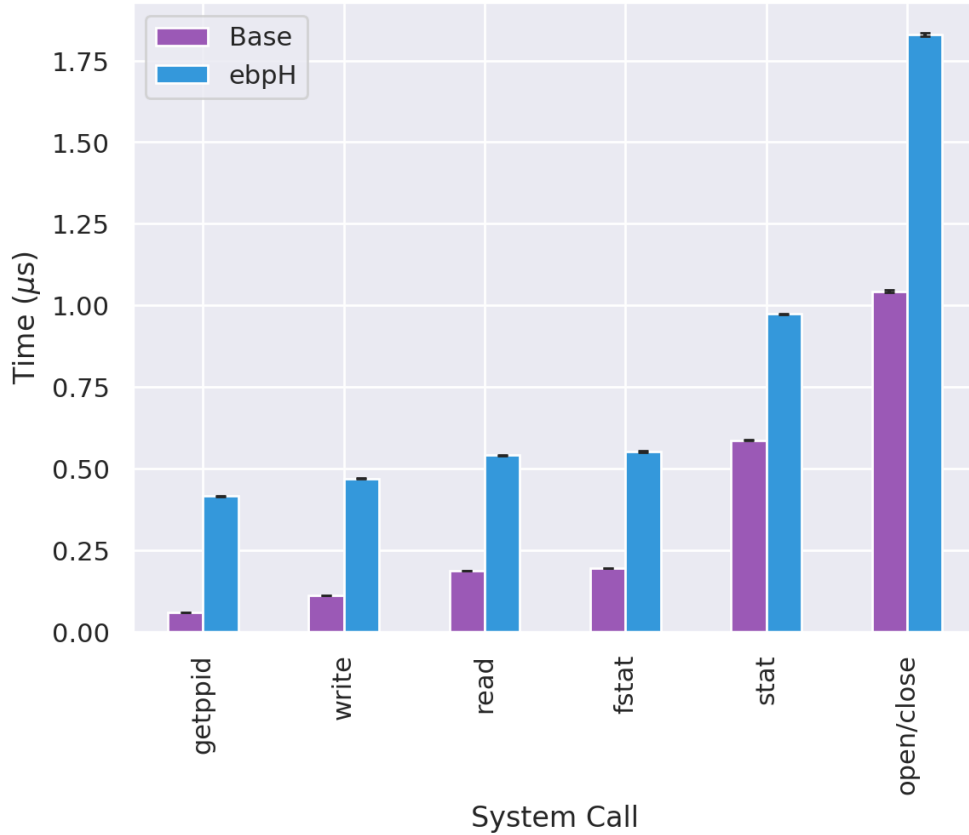
#### 4.2.1 bronte-lmbench System Latency Micro-Benchmark

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 4.3:** 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	$T_{\text{base}} (\mu\text{s})$	$T_{\text{ebpH}} (\mu\text{s})$	Diff. ( $\mu\text{s}$ )	% Overhead
<code>getppid</code>	0.058 (0.0023)	0.416 (0.0157)	0.357811	614.784969
<code>write</code>	0.111 (0.0039)	0.469 (0.0168)	0.357955	321.179901
<code>read</code>	0.187 (0.0064)	0.540 (0.0185)	0.353581	189.189001
<code>fstat</code>	0.194 (0.0062)	0.552 (0.0171)	0.357821	184.176095
<code>stat</code>	0.587 (0.0146)	0.973 (0.0250)	0.386082	65.765787
<code>open/close</code>	1.043 (0.0348)	1.830 (0.0567)	0.787454	75.509370

As shown in [Table 4.3](#), 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.

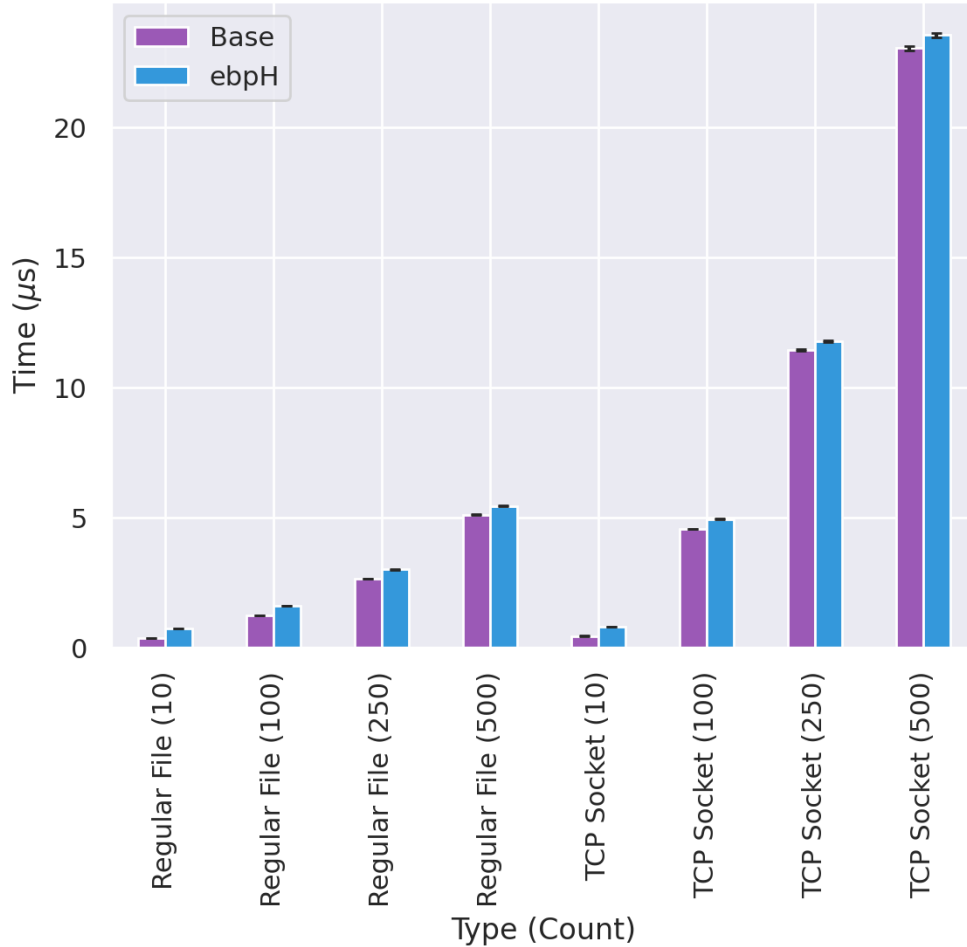


**Figure 4.1:** Mean system call times from the `bronte-lmbench` dataset. Standard error is given as error bars. Smaller difference in times is better.

**Table 4.4:** Results of the `select(2)` benchmarks from the `bronte-lmbench` dataset. Standard deviations are given in parentheses and smaller overhead is better.

Type	Count	$T_{\text{base}}$ ( $\mu\text{s}$ )	$T_{\text{ebpH}}$ ( $\mu\text{s}$ )	Diff. ( $\mu\text{s}$ )	% 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 provide an idea of the overhead imposed on a blocking system call in a controlled environment; the more time the kernel spends blocking, the smaller effect `ebpH`'s overhead has on system call runtime. `select(2)` [50] is used to wait until one



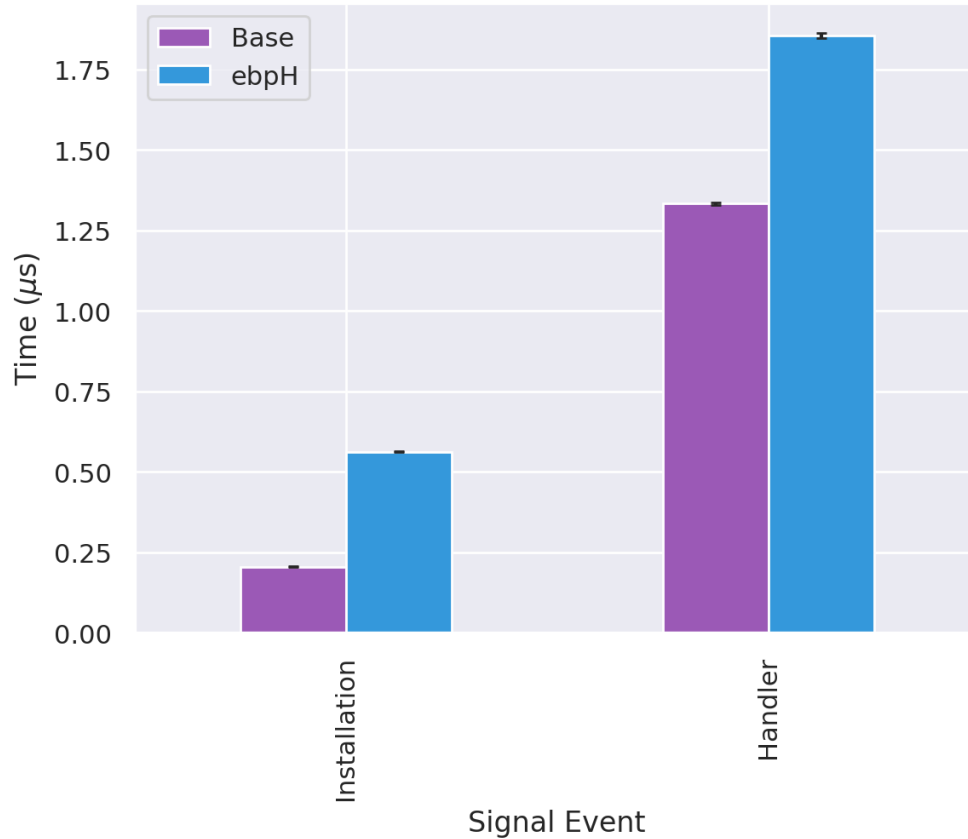
**Figure 4.2:** Mean `select(2)` times from the `bronte-lmbench` dataset. Standard error is given as error bars. Smaller difference in times is better.

or more file descriptors become available for a given operation; the `select(2)` benchmarks from `lmbench` invoke this system call on predefined sets of file descriptors, shown in [Table 4.4](#). The results here demonstrate that the overhead imposed by `ebpH` rapidly diminishes as the duration spent blocking increases, 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 4.5](#) 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 4.3](#)) imposed on `rt_sigreturn(2)` [52], which is invoked upon return from a signal handler.

**Table 4.5:** 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	$T_{\text{base}}$ ( $\mu\text{s}$ )	$T_{\text{ebpH}}$ ( $\mu\text{s}$ )	Diff. ( $\mu\text{s}$ )	% 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 ebp<sub>H</sub>, 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 ebp<sub>H</sub>'s practical overhead. Table 4.6 presents the overhead of running three distinct process creation C programs as follows:



- `fork+exit` forks<sup>12</sup> 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)` [66] 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. Starting with `fork+exit`, Table 4.6 shows that `ebpH` imposes very little overhead on basic process creation, on the order of 5 microseconds, or about 2.7%.

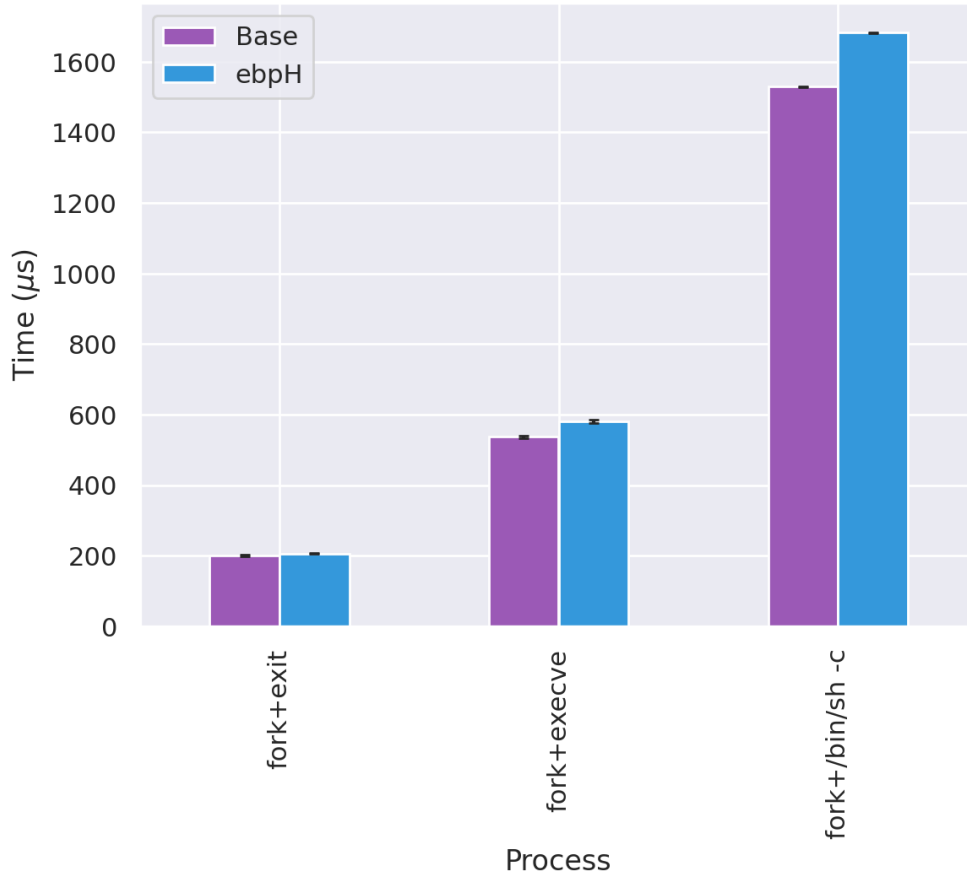
The `fork+execve` case introduces more overhead, due to the special operations that `ebpH` must perform when a process first executes, such as looking up a binary’s inode information and associating it with a profile (creating this profile if it does not yet exist). While this operation is not free, 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 4.6:** Results of the process creation benchmarks from the `bronte-lmbench` dataset. Standard deviations are given in parentheses and smaller overhead is better.

Process	$T_{\text{base}}$ ( $\mu\text{s}$ )	$T_{\text{ebpH}}$ ( $\mu\text{s}$ )	Diff. ( $\mu\text{s}$ )	% Overhead
<code>fork+exit</code>	200.503 (17.3410)	205.998 (11.2935)	5.494621	2.740415
<code>fork+execve</code>	536.914 (30.5695)	580.532 (47.9242)	43.617913	8.123821
<code>fork+/bin/sh -c</code>	1529.053 (20.5609)	1682.445 (13.9791)	153.392500	10.031866

<sup>12</sup>All forks produce the `clone(2)` system call.

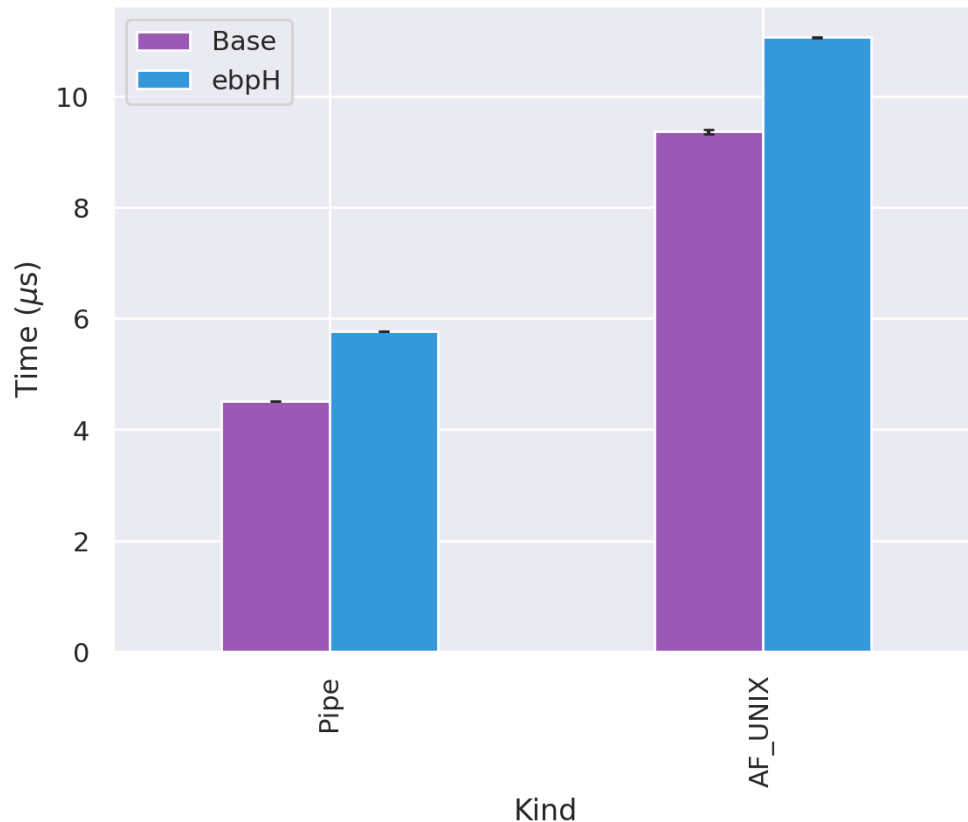


**Figure 4.4:** Mean process creation times from the `bronte-lmbench` dataset. Standard error is given as error bars. Smaller difference in times is better.

Table 4.7 shows the overhead caused by ebp<sub>H</sub> on two methods of IPC, pipes and Unix domain stream sockets. UNIX stream socket IPC, ebp<sub>H</sub> imposes an overhead of 1.7 microseconds, or about 18%. For pipes, it imposes an overhead of 1.25 microseconds, or about 28%. While these results are significant, they shouldn't pose much of a problem for modern applications.

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

Type	$T_{\text{base}}$ ( $\mu\text{s}$ )	$T_{\text{ebpH}}$ ( $\mu\text{s}$ )	Diff. ( $\mu\text{s}$ )	% 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-1mbench` dataset. Standard error is given as error bars. Smaller difference in times is better.

#### 4.2.2 `bronte-kernel` Kernel Compilation Micro-Benchmark

While `1mbench` provides a good representation of the overhead associated with system simple system calls and various simple operations, it is not necessarily indicative of performance impact as a whole. In order to ascertain how resource-intensive operations are affected by `ebpH`, I ran a benchmark of Linux 5.6 kernel compilation times. Five trials were run without `ebpH` running and five more trials were run with `ebpH` running. [Table 4.8](#) shows the results of the benchmark.

According to [Table 4.8](#), `ebpH` has relatively small impact on the overhead of kernelspace operations during compilation, with a `System` overhead of only 10.6%. This makes sense, as most of the system calls being made during kernel compilation are relatively long to begin with, such as `execve(2)`. Longer system calls will have a higher base time and thus `ebpH`'s sub-microsecond runtime has limited impact on total overhead. As expected, `ebpH` has negligible impact on the `User` time, well within the margin of error. The total impact that `ebpH` had on compilation times is reflected by the `Elapsed` time, which shows that `ebpH` only

**Table 4.8:** Kernel compilation times from the `bronte-kernel` dataset. `System` represents CPU time spent in kernelspace, `User` represents CPU time spent in userspace, and `Elapsed` represents real time elapsed. Note that the test was run using all 16 of `bronte`'s logical cores, therefore true elapsed time is significantly shorter than system and user CPU times. Standard deviations are given in parentheses and smaller overhead is better.

Category	$T_{\text{base}}$ (s)	$T_{\text{ebpH}}$ (s)	Diff. (s)	% Overhead
System	1525.412 (1.7603)	1687.833 (8.0621)	162.421667	10.647727
User	12333.737 (27.8529)	12370.957 (4.1244)	37.220000	0.301774
Elapsed	915.173 (3.9876)	924.032 (1.1194)	8.858333	0.967940

had a 1% performance impact overall.

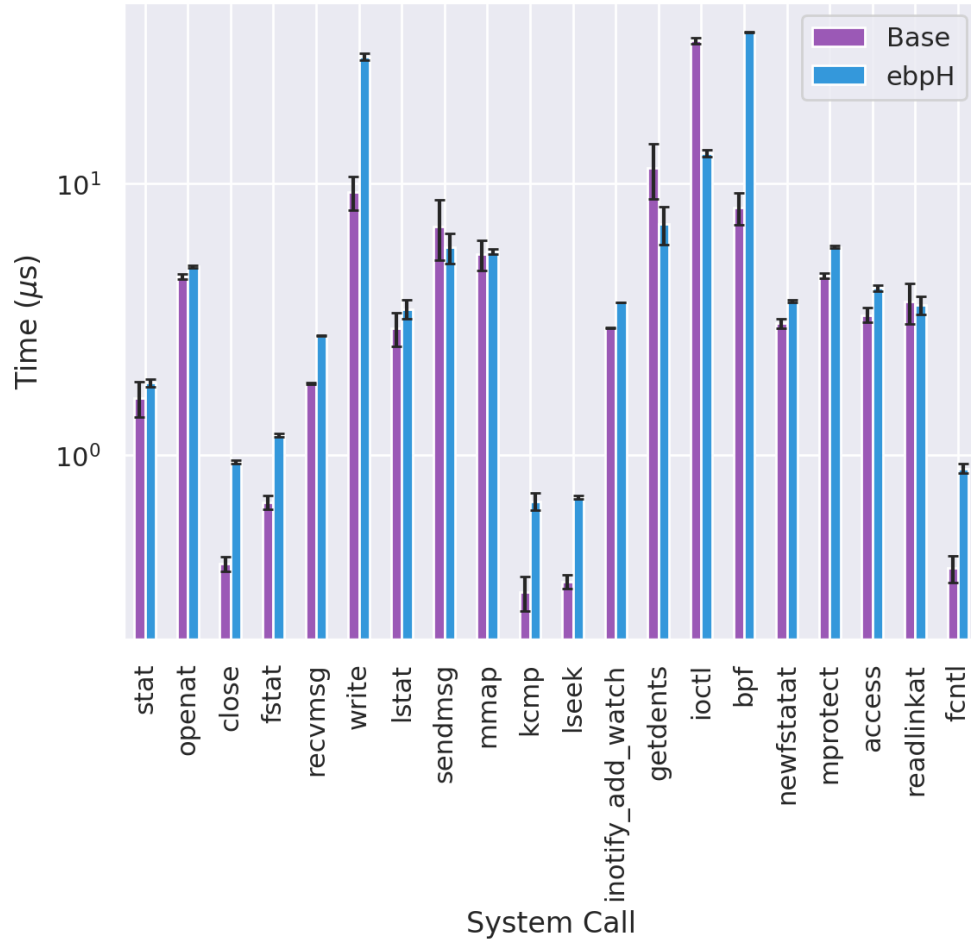
### 4.2.3 `bronte-7day`

The `bronte-7day` macro-benchmark was collected using `bpfbench` over a period of 14 days in total: seven days with `ebpH` and seven without. `bronte` is a workstation in the CCSL lab at Carleton University. Tests were run under an idle workload. [Table 4.9](#) and [Figure 4.6](#) show the top 20 system calls by count (after removing outliers and high variance blocking system calls) over the 14 day period along with associated overheads for the base and `ebpH` tests.

The data in [Table 4.9](#) show that `ebpH` imposes anywhere from relatively minor to severe overhead on the most frequency executed system calls in `bronte-7day`. A few results show slight performance improvements under `ebpH`, but these are anomalous. Such anomalous results are likely due to ambient system factors such as caching, availability of resources, or changes in behavior based on flags, such as in the case of `send_msg(2)` which blocks on busy sockets by default but can be made to immediately return instead. Notably, the `bpf(2)` system call appears to add significant overhead (over 28 microseconds or about 344%), but this can be explained by the fact that it is being made by `ebpH` itself to lookup and update maps. These map operations require significantly more runtime, and so this is reflected in the results. Besides the aforementioned anomalies, these results are mostly indicative of the overhead that `ebpH` imposes on frequent system calls; however, the next two sections will present the same benchmark run under production and ordinary use workloads, which will be more representative of `ebpH`'s overhead in practice.

**Table 4.9:** Top 20 system call overheads by count, with standard deviations of less than 10 microseconds, in the `bronte-7day` dataset. Standard deviations are given in parentheses.

System Call	Count	$T_{\text{base}}$ ( $\mu\text{s}$ )	$T_{\text{ebpH}}$ ( $\mu\text{s}$ )	Diff. ( $\mu\text{s}$ )	% Overhead
<code>stat</code>	7460932	1.620 (0.7368)	1.843 (0.3344)	0.222444	13.727372
<code>openat</code>	6941855	4.532 (0.3065)	4.945 (0.3266)	0.412563	9.102896
<code>close</code>	6307379	0.396 (0.0722)	0.943 (0.0663)	0.546845	137.976090
<code>fstat</code>	5145258	0.669 (0.1174)	1.185 (0.0953)	0.515897	77.127432
<code>recvmsg</code>	4053832	1.832 (0.0330)	2.746 (0.0361)	0.913306	49.840832
<code>write</code>	2417067	9.273 (3.9555)	29.324 (4.8695)	20.050857	216.228374
<code>lstat</code>	1192839	2.917 (1.2434)	3.446 (1.4707)	0.529575	18.155487
<code>sendmsg</code>	978352	6.956 (5.1987)	5.801 (3.9481)	-1.155147	-16.606216
<code>mmap</code>	973233	5.472 (2.0974)	5.606 (0.6333)	0.134444	2.457002
<code>kcmp</code>	953636	0.312 (0.1350)	0.675 (0.2596)	0.363155	116.520244
<code>lseek</code>	919076	0.341 (0.0574)	0.697 (0.0548)	0.355956	104.318044
<code>inotify_add_watch</code>	841595	2.942 (0.0421)	3.652 (0.0321)	0.710123	24.141071
<code>getdents</code>	816107	11.368 (7.8602)	7.070 (5.9365)	-4.297345	-37.803231
<code>ioctl</code>	762010	33.661 (2.5606)	12.949 (1.9564)	-20.712873	-61.532930
<code>bpf</code>	760143	8.140 (3.3270)	36.176 (0.7250)	28.036798	344.446508
<code>newfstatat</code>	721477	3.054 (0.3767)	3.687 (0.2255)	0.632849	20.720470
<code>mprotect</code>	626527	4.573 (0.3004)	5.836 (0.3272)	1.263115	27.622490
<code>access</code>	572543	3.278 (0.6211)	4.113 (0.5105)	0.834988	25.469896
<code>readlinkat</code>	527330	3.663 (1.8975)	3.565 (1.4156)	-0.097857	-2.671503
<code>fcntl</code>	512561	0.383 (0.1288)	0.892 (0.1908)	0.509302	133.131405



**Figure 4.6:** Top 20 system call overheads by count, with standard deviations of less than 10 microseconds, in the *bronte-7day* dataset. Time scale is logarithmic. Standard error is given as error bars.

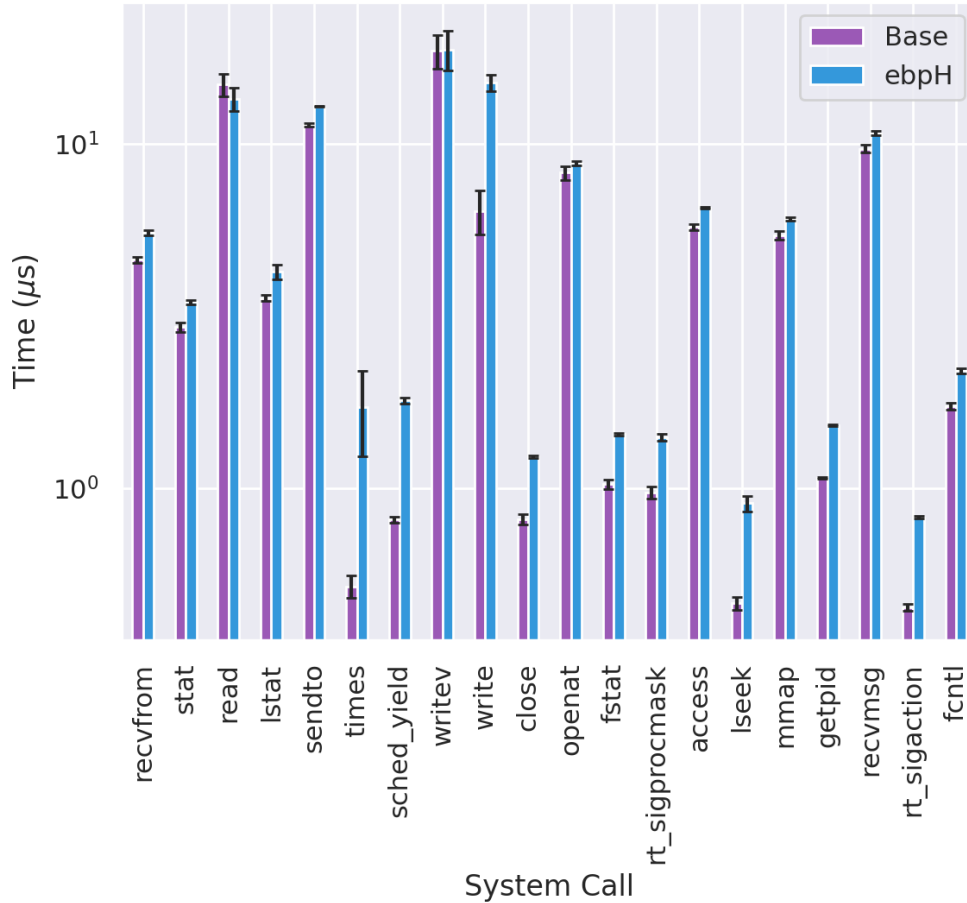
#### 4.2.4 homeostasis-3day

The `homeostasis-3day` macro-benchmark was collected using `bpfbench` over a period of six days in total: three days with `ebpH` and three without. `homeostasis` is a Mediawiki server used to host the COMP3000 course wiki at Carleton University. Tests were run under the normal workload associated with running the webserver and SQL database. [Table 4.10](#) and [Figure 4.7](#) show the top 20 system calls by count (after removing outliers and high variance blocking system calls) over the six day period along with associated overheads for the base and `ebpH` tests.

**Table 4.10:** Top 20 system call overheads by count, with standard deviations of less than 10 microseconds, in the `homeostasis-3day` dataset. Standard deviations are given in parentheses.

System Call	Count	$T_{\text{base}} (\mu\text{s})$	$T_{\text{ebpH}} (\mu\text{s})$	Diff. ( $\mu\text{s}$ )	% Overhead
<code>recvfrom</code>	22311033	4.599 (0.3375)	5.505 (0.2877)	0.906167	19.705344
<code>stat</code>	16578427	2.929 (0.3409)	3.467 (0.1842)	0.537833	18.361831
<code>read</code>	12023727	14.824 (3.8911)	13.446 (3.5872)	-1.378083	-9.296142
<code>lstat</code>	11953074	3.560 (0.2693)	4.238 (0.6952)	0.678280	19.052372
<code>sendto</code>	11070639	11.330 (0.4644)	12.800 (0.2013)	1.469417	12.968969
<code>times</code>	9085541	0.518 (0.1353)	1.712 (1.6402)	1.194250	230.587289
<code>sched_yield</code>	7951003	0.810 (0.0547)	1.795 (0.1257)	0.985750	121.760165
<code>writew</code>	6195312	18.546 (7.1331)	18.717 (8.5272)	0.171500	0.924753
<code>write</code>	6102652	6.362 (3.2141)	15.007 (2.7556)	8.645000	135.890282
<code>close</code>	5701429	0.811 (0.0969)	1.236 (0.0387)	0.424833	52.405428
<code>openat</code>	4985018	8.221 (1.3344)	8.762 (0.3944)	0.540833	6.578347
<code>fstat</code>	4196505	1.025 (0.1166)	1.434 (0.0439)	0.408750	39.878049
<code>rt_sigprocmask</code>	3746321	0.969 (0.1343)	1.405 (0.1113)	0.436083	44.987964
<code>access</code>	3302798	5.721 (0.3865)	6.499 (0.1562)	0.778000	13.599219
<code>lseek</code>	3010731	0.463 (0.0679)	0.903 (0.1624)	0.440083	95.067507
<code>mmap</code>	2438472	5.424 (0.5294)	6.022 (0.2403)	0.597917	11.024045
<code>getpid</code>	2063206	1.068 (0.0177)	1.520 (0.0247)	0.452333	42.350004
<code>recvmsg</code>	1745288	9.652 (0.8664)	10.700 (0.5092)	1.047917	10.857084
<code>rt_sigaction</code>	1308454	0.450 (0.0363)	0.822 (0.0272)	0.371833	82.690882
<code>fcntl</code>	828473	1.727 (0.1388)	2.189 (0.1229)	0.461917	26.749349

As with the previous macro-benchmark, [Table 4.10](#) shows that `ebpH` has minor to significant impact on the runtime overhead of the most frequently executed system calls. Of the five most frequent system calls, all present with an overhead of less than 20%, and `read(2)` in particular shows a slight performance improvement under `ebpH`. As in the previous section,



**Figure 4.7:** Top 20 system call overheads by count, with standard deviations of less than 10 microseconds, in the `homeostasis-3day` dataset. Time scale is logarithmic. Standard error is given as error bars.

this result is clearly pathological and is likely a result of ambient factors such as caching and availability of resources. The overheads of ordinary, non-blocking system calls such as `getpid(2)` and `lseek(2)` are consistent with previously observed results. In general, the overheads presented here are unlikely to have significant impact on performance of modern applications.

#### 4.2.5 arch-3day

Similar to the `homeostasis` tests, the `arch-3day` macro-benchmark was collected over a period of six days on `arch`, my personal desktop computer; the idea was to see what sort of overhead `ebpH` caused during the everyday use of a personal workstation. While these results certainly have a higher variance than previous results due to inconsistent usage and workload, it is important to see how `ebpH` behaves on a variety of systems under a variety of use cases.



Table 4.11 and Figure 4.8 show the top 20 system calls by count (after removing outliers and high variance blocking system calls) over the six day period along with associated overheads for the base and ebp<sub>H</sub> tests.

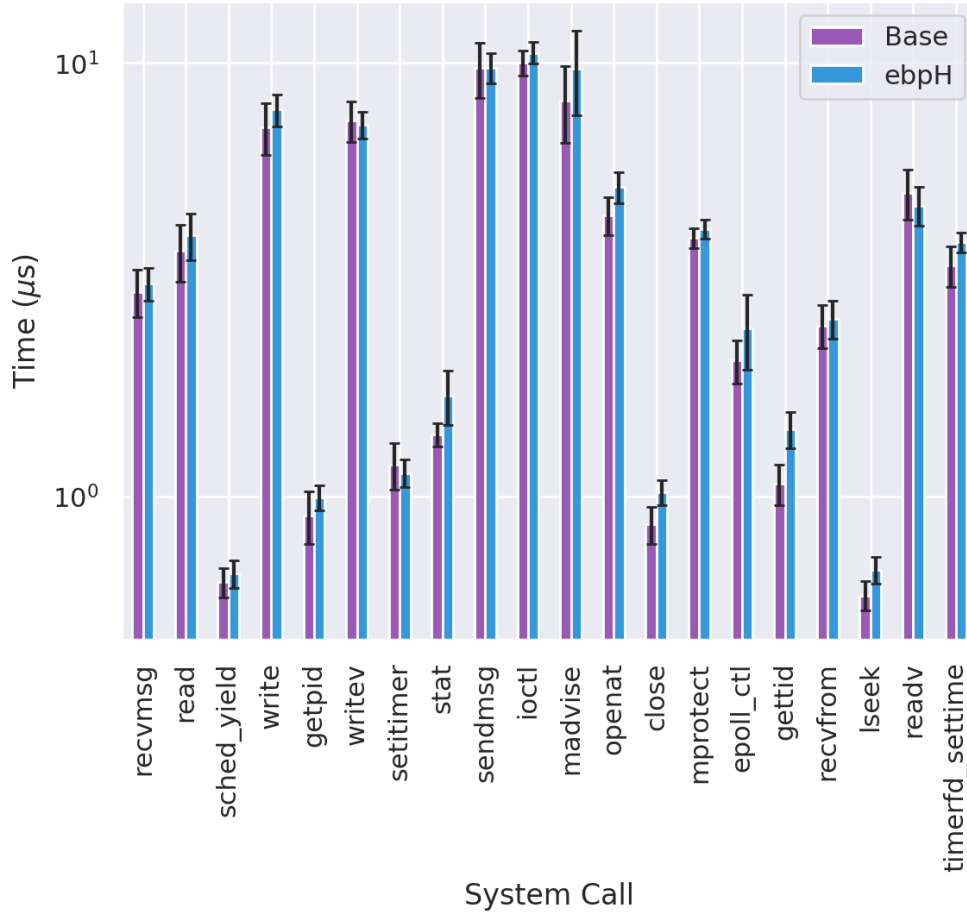
**Table 4.11:** Top 20 system call overheads by count, with standard deviations of less than 10 microseconds, in the arch-3day dataset. Standard deviations are given in parentheses.

System Call	Count	$T_{\text{base}}$ ( $\mu\text{s}$ )	$T_{\text{ebpH}}$ ( $\mu\text{s}$ )	Diff. ( $\mu\text{s}$ )	% 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
writew	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	101386112	10.028 (2.0969)	10.558 (1.9098)	0.530000	5.285043
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

The results shown in Table 4.11, are roughly consistent with previous macro-benchmarks. In this dataset, the five most frequent calls present with an overhead of less than approximately 10%, an even better even better than previous trials. A few system calls show moderate performance improvement, but as before, this is likely explained by ambient factors in the system.

## 4.2.6 Summary

Section 4.2.1 shows that ebp<sub>H</sub> has significant impact on the overheads of short system calls and that this impact diminishes as kernel runtime increases. These findings extend to other aspects of system performance, such as process creation, interprocess communication, and



**Figure 4.8:** Top 20 system call overheads by count, with standard deviations of less than 10 microseconds, in the arch-3day dataset. Time scale is logarithmic. Standard error is given as error bars.

signal handling. [Section 4.2.2](#) demonstrates that ebp<sub>H</sub> imposes negligible overhead on kernel compilation, a task that is highly intensive in CPU usage and involves the creation of many processes. This further reinforces the notion that ebp<sub>H</sub>'s overhead is acceptable in practice, its relatively high impact on short system call runtimes. The results from the `bpfbench` macro-benchmarks ([Section 4.2.3](#), [Section 4.2.4](#), [Section 4.2.5](#)) show that while ebp<sub>H</sub> has significant impact on the overhead of short system calls, this impact diminishes significantly as base system call runtime increases. Further, the majority of frequent system calls on a variety of different work loads tend to have a high enough base runtime that ebp<sub>H</sub> has minimal impact in practice.

### 4.3 Comparing Results with the Original pH

In Somayaji's dissertation [54], he provides performance metrics on selected system calls as well as kernel compilation benchmarks and X11 performance statistics. Some of the methodology I have used for measuring ebpH's performance directly mirrors this approach in order to facilitate easy comparison between the two systems. In particular, the `lmbench` micro-benchmarks and the `kernel-build` micro-benchmark will be informative in this regard.

The `bronte-lmbench` system call results in Table 4.3 on page 45 show that ebpH consistently adds just over a third of a microsecond of runtime to system calls on `bronte`. Depending on the call in question, this can result in minor to significant overhead – different system calls require different amounts of processing in kernelspace, depending on their design and implementation. In the pH dissertation [54], Somayaji presents a small variety of system call overheads with varying base times and shows that pH adds approximately 1.9 microseconds of runtime. Although ebpH adds only about one sixth of this overhead, this result is misleading due to the difference in hardware specifications between `lydia`, the system which pH was tested on in 2002, and `bronte`, the system that ebpH was tested on; in particular, `bronte` is a *significantly* faster and more powerful machine, which means that base runtime will not be directly comparable between the two systems. The percent overhead statistic may be slightly more informative here. For null system calls (that is, system calls which require next to no thinking on the part of the kernel), such as `getpid` or `getppid`, ebpH adds 614% overhead, which may seem quite significant. In contrast, pH adds only about 165%. However, if pH were tested on `bronte` today, this overhead would likely be much larger, as the base execution time for system calls would be significantly smaller. As the complexity of calls increases, the percent overheads expressed by pH and ebpH approach each other. For instance, `write(2)` has an overhead of approximately 133% in pH and about 321% in ebpH. `sigaction(2)` can also be compared with the signal handler install results from Table 4.5 on page 48 (since that is in essence just a call to `rt_sigaction(2)`); pH achieves an overhead of about 75% while ebpH adds about 175%.

The dynamic process creation latency results from `bronte-lmbench` will also be quite informative for establishing a comparison between pH and ebpH. In the pH dissertation [54], Somayaji presents the overheads of three distinct process creation benchmarks, exactly the same ones that I have used here to test ebpH. For the `fork+exit` test, pH achieves 3.3% overhead, while ebpH achieves 2.7%; in this case, ebpH actually begins to outperform the original pH. These results are also reflected in the next two tests, `fork+execve` and `fork+/bin/sh -c`. For `fork+execve`, ebpH performs astonishingly well compared to its predecessor, with an overhead of 8.1% compared to pH's 273.6%. This result, however, is slightly misleading

as pH loads profiles from disk into kernel memory on every `execve(2)` call, whereas ebpH maintains them in a map. Thus, ebpH’s overhead does not include the overhead required to load a profile into memory. Similarly, ebpH’s results in the `fork+/bin/sh -c` test show an overhead of about 10%, while pH’s overhead is closer to 29%. The impact of the differences in handling of profiles is more diminished here, although it is still a factor. Regardless, these results show that ebpH is consistently able to either outperform or keep up with pH in real applications.

Finally, the kernel compilation benchmarks presented in [Section 4.2.2](#) show improvement over the original pH results [54]. In particular, ebpH only adds about 10% overhead to `System` time, compared to pH’s 38%; however, this large improvement is most likely due to ebpH’s reduced overhead on `execve(2)` calls, which make up a large portion of kernelspace overhead for compilation tasks. Even so, the end result is a 1% total performance overhead for ebpH, compared to 3% for the original pH, which shows the ebpH can keep up with pH in practice.

## 5 Discussion

Previous sections have presented the design, implementation, and testing of ebpH, and offered a comparison between ebpH and its predecessor, pH, in light of design and implementation differences between the two. Past sections have shown that ebpH supports many of the same features as the original pH while offering significantly higher portability and adaptability. Experimental results presented in the previous section have shown that its performance overhead can compete with the original version. This section will discuss further the viability of eBPF-based anomaly detection in light of the results, and present topics for future work to improve and extend future versions of ebpH. [Section 5.1](#) discusses the shortcomings of eBPF that I believe need to be resolved in order to permit more complex intrusion detection software within this paradigm, while [Section 5.2](#) presents topics for future work in development, testing, and design of future iterations of ebpH.

### 5.1 Shortcomings of eBPF

In previous sections, I have highlighted the important factors that make the eBPF paradigm an excellent choice for the development and deployment of host-base intrusion detection systems. While the experimental results in [Section 4](#) have shown that eBPF can be as efficient as kernel-based implementations and [Section 3](#) has described how eBPF can be used to implement many of the same features as kernel-based implementations, I have not yet

touched on many of the shortcomings of the technology. This section will attempt to rectify this gap in light of empirical observations from the development of ebpH.

### 5.1.1 Lack of Concurrency Control Mechanisms in Tracing Programs

As discussed in previous sections (Section 3.8), the lack of concurrency control mechanisms in eBPF tracing programs [34, 35] is detrimental to the use of eBPF for the creation of complex, security-sensitive applications. While the risks associated with non-deterministic data are fine for simple tracing programs designed for use cases such as performance analysis, this assumption quickly breaks down for more complex applications that rely on accurate results. The current version of ebpH mostly gets away with this due to the way it handles lookahead pairs combined with the use of atomic add and subtract operations for profile flags. However, future iterations of ebpH may depend on more complex behavioral tracking and analysis which is currently not possible in eBPF to an acceptable degree of certainty.

eBPF *does* have restricted concurrency primitives, such as `bpf_spin_lock` [34], but these are limited to non-tracing (and non-socket) programs due to insufficient checks by the verifier. This is to prevent buggy BPF programs from causing kernel functions to timeout, which could potentially crash the system. Resolving this problem would require updates to the verifier to ensure that it can properly check preemptions related to spin locks in tracing programs. While this functionality is currently not available the BPF maintainers have indicated that they are planning to support locking in tracing programs in the future [34].

### 5.1.2 Limited Support for Necessary Kernel Helpers

One of the major improvements of extended BPF over classic BPF is the introduction of the `bpf_call` instruction to its bytecode [59, 60]. In particular, eBPF programs can invoke a predetermined set of helper functions provided by the kernel [5, 21]. Due to verifiability requirements on BPF programs, the set of kernel functions that can be invoked is *highly* limited in scope. As of Linux 5.5, eBPF supports 117 distinct helper functions [34]. However, many of these helpers relate specifically to operations on eBPF maps and lookups on architecture-specific kernel data structures and more still are limited to specific niche program types, such as XDP, socket filter, or traffic classifier programs.

One particular pain-point that I encountered during the development of ebpH is the lack of a reliable means of constructing pathnames in eBPF. The kernel provides helpers for doing so, but these are not available for BPF programs. This means that ebpH is unable to support hashing profiles by pathname, and instead must rely on the computation of a unique key from filesystem metadata. While this solution is pragmatically the same, ebpH's usability

suffers as a result. In the original pH, profiles were stored on disk in subdirectories that mirrored their pathname in the original filesystem; in ebpH however, they are simply stored with the same filename as the corresponding profile key. This makes it difficult for users to interact with ebpH without using `ebph-ps` to figure out the key of the profile they want first. While there are potential workarounds for this, including the determination of pathnames in userspace, these are not ideal in terms of performance or reliability. It is worth noting, however, that a patch is currently under review to remedy this gap in eBPF’s functionality [72].

### 5.1.3 Verifier Bugs

Although the verifier provides critical safety guarantees to eBPF programs, it suffers from a few bugs that, in the best case, make it difficult to work with. In particular, the verifier can be inconsistent when performing static analysis on large and complex programs, such as the BPF programs employed by ebpH. To illustrate this complexity empirically, consider Figure 5.1 which depicts the instruction flow of ebpH’s `sys_exit` tracepoint program.



**Figure 5.1:** The instruction flow of ebpH’s `sys_exit` tracepoint program. Note the complexity of the BPF program. This figure was generated using `bpftool` and `graphviz`’s `osage` tool.

The verifier itself is a rather complex program; as of Linux 5.5, it consists of over 10,000 lines of C code [35]. As a consequence of this complexity, the probability of bugs increases significantly. If the verifier fails at any stage in the verification process, it errs on the side of caution and rejects the program. Unfortunately, this behavior does impact ebpfH to an extent. Due to a presently unknown bug in the verifier, it occasionally rejects the `sys_exit` program depicted in Figure 5.1; this issue can be resolved by restarting the system. While this behavior is certainly annoying, it is an acceptable trade-off for the safety guarantees that the verifier provides when it is working properly.

One argument that may arise from this notion of inconsistent verifier behavior is whether it truly protects the system at all from buggy BPF programs. After all, one of the primary advantages cited for eBPF programs over kernel-based implementations is the ability to guarantee production safety despite BPF code running in ring 0 with full access to the kernel. A counter-point to this argument is that a 99% probability of guaranteeing safety is better than a 0% probability – that is to say, having a verifier that works almost all of the time is better than not having one at all.

#### 5.1.4 Dropped Perf Buffer Submissions

Another primary advantage for using eBPF over traditional kernel-based implementations is the ability to easily buffer communication with userspace. Context switches between userspace and kernelspace are expensive [17]; eBPF largely mitigates this by allowing userspace programs to buffer map access, which in turn allows for variable granularity in the amount of context switches required per event. For instance, ebpfH reads events from its BPF programs using the perf event buffer interface provided by eBPF; to do this, it simply polls each map every second via an event loop.

While perf buffers do reduce program overhead in practice, they have caveats of their own that need to be addressed. For instance, BPF programs may outright refuse to submit some perf events (this behavior was encountered during the development of ebpfH) and events that occur too frequently may fill the buffer completely, which in turn causes events to be dropped. Although these caveats do pose significant challenges to the development of reliable BPF programs, it is possible to circumvent them with careful design choices. For instance, submission failure can be checked within the BPF program and backup mechanisms can then be employed to be sure that the data makes it to userspace; dropped submissions that occur due to high frequency events may be solved by tuning the size of the buffer or adjusting the frequency at which the BPF program polls it.

## 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 [54] 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 [54], 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 [21]. In particular, Linux 5.3 introduced two critical helpers [25] 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 de-



livered instantly, without the usual delays associated with sending signals from userspace. By sending a process the signal `SIGSTOP` [51], it will be possible to stop its execution in *real time*, during the offending system call. Subsequently, a `SIGCONT` [51] 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 [54]: `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 of the system. In anomaly detection, there are a few important heuristics to consider when determining the efficacy of a system: false positive rate (FPR), false negative rate (FNR), true positive rate (TPR), true negative rate (TNR), and alarm precision (AP) [44]. When combined, these heuristics provide an accurate representation of:

- How often the system flags legitimate behavior as anomalous (FPR);
- How often the system misses anomalous behavior (FNR);
- How often the system detects anomalous behavior (TPR);
- How often the system allows legitimate behavior (TNR);
- The ratio of true positives to total positives (AP).

According to the above definitions, FPR and FNR provide an indication of the *error rate* of an anomaly detection system, while TPR and TNR provide an indication of the *correctness rate*. Finally, AP provides an indication of what percentage of all flagged anomalies are

correct. Determining these five heuristics for ebpH will require carefully planned testing strategies comprised of building known-good profiles, mounting various known attacks, and measuring rates of flagged events against predetermined values. Additionally, the same known-good profiles should be tested for extended periods of time under normal system behavior to ensure that the rate of false positives is acceptable.

As a general-purpose anomaly-based IDS, it is important to show that ebpH is capable of detecting a wide variety of attacks. The mimicry attacks described in Wagner and Soto’s paper [70] are particularly interesting, as they were directly designed to defeat the original pH system (albeit an earlier version with much shorter lookahead pair window length) by constructing attack patterns that generate false negatives. The results depicted in their paper should be compared against the results from testing and used to inform later changes to ebpH.

#### 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 [61] 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) Refactoring all hashmaps into the LRU\_HASH type;
- 2) Refactoring profile data storage to use a HASH\_OF\_MAPS.

**Refactoring Profiles and Processes to Use LRU\_HASH.** The LRU\_HASH [21, 25] 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 Profile Data to Use HASH\_OF\_MAPS.** HASH\_OF\_MAPS [21, 25] 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 [33], HASH\_OF\_MAPS and ARRAY\_OF\_MAPS have only been supported in bcc [25] since a November 2019 patch [56]. 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 allocate 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] J. P. Anderson, “Computer Security Threat Monitoring and Surveillance,” James P. Anderson Co., Fort Washington, PA, Tech. Rep., 1980.
- [4] 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](https://doi-org.proxy.library.carleton.ca/10.1145/1165389.945462). [Online]. Available: <https://doi-org.proxy.library.carleton.ca/10.1145/1165389.945462>.
- [5] *bpf-helpers(7) Linux Programmer’s Manual*, Linux, Nov. 2019.
- [6] *bpf(2) Linux Programmer’s Manual*, Linux, Aug. 2019.
- [7] 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](https://www.usenix.org/legacy/publications/library/proceedings/usenix04/tech/general/full_papers/cantrill/cantrill.pdf).
- [8] J. Cespedes and P. Machata, *Ltrace(1) linux user’s manual*, Ltrace project, Jan. 2013.
- [9] 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](https://doi-org.proxy.library.carleton.ca/10.1145/2103799.2103805). [Online]. Available: <https://doi-org.proxy.library.carleton.ca/10.1145/2103799.2103805>.
- [10] J. Corbet, *Bounded loops in BPF programs*, Dec. 2018. [Online]. Available: <https://lwn.net/Articles/773605/>.
- [11] 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](https://doi-org.proxy.library.carleton.ca/10.1145/263109.263133). [Online]. Available: <https://doi-org.proxy.library.carleton.ca/10.1145/263109.263133>.

- [12] D. E. Denning, “An Intrusion-Detection Model,” *IEEE Transactions on Software Engineering*, vol. SE-13, no. 2, pp. 222–232, 1987.
- [13] D. Denning and P. G. Neumann, *Requirements and Model for IDES - A Real-Time Intrusion-Detection Expert System*. SRI International, 1985, vol. 8.
- [14] M. Fleming, *A thorough introduction to eBPF*, Dec. 2017. [Online]. Available: <https://lwn.net/Articles/740157/>.
- [15] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, “A sense of self for unix processes,” in *Proceedings 1996 IEEE Symposium on Security and Privacy*, May 1996, pp. 120–128. DOI: [10.1109/SECPRI.1996.502675](https://doi.org/10.1109/SECPRI.1996.502675).
- [16] P. D. Fox, *Dtrace4linux/linux*, Sep. 2019. [Online]. Available: <https://github.com/dtrace4linux/linux>.
- [17] M. Gebai and M. R. Dagenais, “Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead,” *ACM Comput. Surv.*, vol. 51, no. 2, Mar. 2018, ISSN: 0360-0300. DOI: [10.1145/3158644](https://doi.org/10.1145/3158644). [Online]. Available: <https://doi.org/10.1145/3158644>.
- [18] S. Goldshtein, “The Next Linux Superpower: eBPF Primer,” USENIX SRECon16 Europe, Jul. 2016. [Online]. Available: <https://www.usenix.org/conference/srecon16europe/program/presentation/goldshtein-ebpf-primer>.
- [19] B. Gregg, *Linux BPF Superpowers*, Mar. 2016. [Online]. Available: <http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html>.
- [20] B. Gregg, *bpfftrace (DTrace 2.0) for Linux 2018*, Oct. 2018. [Online]. Available: <http://www.brendangregg.com/blog/2018-10-08/dtrace-for-linux-2018.html>.
- [21] B. Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019, ISBN: 0-13-655482-2.
- [22] B. Gregg, J. Mauro, and B. M. Cantrill, *DTrace: dynamic tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall, 2014.
- [23] 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](https://doi.org/10.1145/3281411.3281443). [Online]. Available: <http://doi.acm.org/10.1145/3281411.3281443>.

- [24] 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](https://doi.org/10.1145/863955.863968). [Online]. Available: <http://doi.acm.org.proxy.library.carleton.ca/10.1145/863955.863968>.
- [25] IOVisor, *Iovisor/bcc*, Oct. 2019. [Online]. Available: <https://github.com/iovisor/bcc>.
- [26] IOVisor, *Iovisor/bpfttrace*, Nov. 2019. [Online]. Available: <https://github.com/iovisor/bpfttrace>.
- [27] IOVisor, *Syscount.py*, Oct. 2019. [Online]. Available: <https://github.com/iovisor/bcc/blob/master/tools/syscount.py>.
- [28] T. Jaeger, *Operating System Security*. Morgan & Claypool Publishers, 2008.
- [29] 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](https://doi.org/10.1109/MC.2002.1012428).
- [30] 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>.
- [31] 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](https://doi.org/10.3233/JCS-2002-101-203).
- [32] B. Kuperman and E. Spafford, “Generation of Application Level Audit Data via Library Interposition,” Sep. 1999.
- [33] 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>.
- [34] Linux Kernel, *include/uapi/linux/bpf.h*, Mar. 2020. [Online]. Available: <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/uapi/linux/bpf.h?h=v5.5.13>.
- [35] Linux Kernel, *kernel/bpf/verifier.c*, Mar. 2020. [Online]. Available: <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/bpf/verifier.c?h=v5.5.13>.

- [36] *LTTng v2.11 - LTTng Documentation*, Oct. 2019. [Online]. Available: <https://ltnng.org/docs/v2.11/>.
- [37] 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>.
- [38] L. W. McVoy, *lmbench*, Intel, Dec. 2019. [Online]. Available: <https://github.com/intel/lmbench>.
- [39] 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.
- [40] 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/>.
- [41] 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](https://doi.org/10.1145/41457.37505). [Online]. Available: <http://doi.acm.org/10.1145/41457.37505>.
- [42] 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.
- [43] *NIT(4p) SunOS 4.1.1 Reference Manual*, Sun Microsystems Inc., Sep. 1990.
- [44] 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).
- [45] W. W. Peng and D. R. Wallace, *Software Error Analysis*. Silicon Press, 1995.
- [46] *ptrace(2) Linux User’s Manual*, Oct. 2019.
- [47] 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](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/systemtap_beginners_guide/understanding-how-systemtap-works).
- [48] S. Rostedt, *Documentation/ftrace.txt*, 2008. [Online]. Available: <https://lwn.net/Articles/290277/>.



- [49] 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>.
- [50] *select(2) Linux Programmer's Manual*, Linux, Nov. 2019.
- [51] *signal(7) Linux Programmer's Manual*, Linux, Aug. 2019.
- [52] *sigreturn(2) Linux Programmer's Manual*, Linux, Sep. 2017.
- [53] 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](https://doi-org.proxy.library.carleton.ca/10.1145/1272996.1273025). [Online]. Available: <https://doi-org.proxy.library.carleton.ca/10.1145/1272996.1273025>.
- [54] 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>.
- [55] 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>.
- [56] Y. Song, *bpf: Add hash of maps support*, Nov. 2019. [Online]. Available: <https://github.com/iovisor/bcc/commit/149c1c8857652997622fc2a30747a60e0c9c17dc>.
- [57] 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](https://doi.org/10.1016/S1389-1286(00)00136-5). [Online]. Available: [http://dx.doi.org/10.1016/S1389-1286\(00\)00136-5](http://dx.doi.org/10.1016/S1389-1286(00)00136-5).
- [58] W. Stallings and L. Brown, *Computer security: principles and practice*. [Online]. Available: <http://www.informit.com/articles/article.aspx?p=782118>.
- [59] 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>.
- [60] 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%206d0fd55dff551b8>.



- [61] A. Starovoitov, *bpf: hash map pre-alloc*, Mar. 2016. [Online]. Available: <https://lwn.net/Articles/679074/>.
- [62] 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>.
- [63] Strace Project, *Strace*. [Online]. Available: <https://strace.io/>.
- [64] *strace(1) Linux User's Manual*, 5.3, Strace Project, Sep. 2019.
- [65] Sysdig Inc., *Draios/sysdig*, Nov. 2019. [Online]. Available: <https://github.com/draios/sysdig>.
- [66] *system(3) Linux Programmer's Manual*, Linux, Mar. 2019.
- [67] L. Torvalds, *Linux/uapi/asm-generic/unistd.h*. [Online]. Available: <https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/unistd.h>.
- [68] 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](https://doi.org/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>.
- [69] 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/>.
- [70] D. Wagner and P. Soto, "Mimicry Attacks on Host-based Intrusion Detection Systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02, Washington, DC, USA: ACM, 2002, pp. 255–264, ISBN: 1-58113-612-9. DOI: [10.1145/586110.586145](https://doi.org/10.1145/586110.586145). [Online]. Available: <http://doi.acm.org/10.1145/586110.586145>.
- [71] V. Weaver, *Perf\_event\_open(2) linux user's manual*, Oct. 2019.
- [72] W. Zhang, *[bpf-next] bpf: add new helper fd2path for mapping a file descriptor to a path-name*, Oct. 2019. [Online]. Available: <https://patchwork.ozlabs.org/patch/1179287/>.

# Appendices

## A eBPF Design Patterns

**Listing A.1:** Handling large datatypes in eBPF programs.

```
1  /* This is way too large to fit within
2   * the eBPF stack limit of 512 bytes */
3  struct bigdata_t
4  {
5      char foo[4096];
6  };
7
8  /* We read from this array every time we want to
9   * initialize a new struct bigdata_t */
10 BPF_ARRAY(__bigdata_t_init, struct bigdata_t, 1);
11
12 /* The main hashmap used to store our data */
13 BPF_HASH(bigdata_hash, u64, struct bigdata_t);
14
15 /* Suppose this is a function where we need to use our
16 * bigdata_t struct */
17 int some_bpf_function(void)
18 {
19     /* We use this to look up from our
20      * __bigdata_t_init array */
21     int zero = 0;
22     /* A pointer to a bigdata_t */
23     struct bigdata_t *bigdata;
24     /* The key into our main hashmap
25      * Its value not important for this example */
26     u64 key = SOME_VALUE;
27
28     /* Read the zeroed struct from our array */
29     bigdata = __bigdata_t_init.lookup(&zero);
30     /* Make sure that bigdata is not NULL */
31     if (!bigdata)
32         return 0;
33     /* Copy bigdata to another map */
34     bigdata = bigdata_hash.lookup_or_try_init(&key, bigdata);
35
36     /* Perform whatever operations we want on bigdata... */
37
38     return 0;
39 }
```

## B bpfbench Source Code

**Listing B.1:** The eBPF component of bpfbench.

```
1  /* bpfbench A better benchmarking tool written in eBPF.
2  * Copyright (C) 2020 William Findlay
3  *
4  * Heavily inspired by syscount from bcc-tools:
5  * https://github.com/iovisor/bcc/blob/master/tools/syscount.py
6  *
7  * This program is free software: you can redistribute it and/or modify
8  * it under the terms of the GNU General Public License as published by
9  * the Free Software Foundation, either version 3 of the License, or
10 * (at your option) any later version.
11 *
12 * This program is distributed in the hope that it will be useful,
13 * but WITHOUT ANY WARRANTY; without even the implied warranty of
14 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
15 * GNU General Public License for more details.
16 *
17 * You should have received a copy of the GNU General Public License
18 * along with this program. If not, see <https://www.gnu.org/licenses
19   />. */
20 #include <uapi/asm/unistd_64.h>
21 #include <linux/sched.h>
22 #include <linux/signal.h>
23
24 struct intermediate_t
25 {
26     u64 pid_tgid;
27     u64 start_time;
28 };
29
30 struct data_t
31 {
32     u64 count;
33     u64 overhead;
34 };
35
36 BPF_PERCPU_ARRAY(intermediate, struct intermediate_t, 1);
37 BPF_PERCPU_ARRAY(syscalls, struct data_t, NUM_SYSCALLS);
38
39 #ifdef FOLLOW
40 BPF_HASH(children, u32, u8);
41
42 RAW_TRACEPOINT_PROBE(sched_process_fork)
43 {
44     struct task_struct *p = (struct task_struct *)ctx->args[0];
45     struct task_struct *c = (struct task_struct *)ctx->args[1];
46
47     u32 ppid = p->tgid;
```

```

48
49     /* Filter ppid */
50     if (ppid != TRACE_PID && !children.lookup(&ppid))
51     {
52         return 0;
53     }
54
55     u32 cpid = c->tgid;
56
57     u8 zero = 0;
58
59     children.update(&cpid, &zero);
60
61     return 0;
62 }
63
64 RAW_TRACEPOINT_PROBE(sched_process_exit)
65 {
66     u32 pid = (bpf_get_current_pid_tgid() >> 32);
67
68     /* Filter ppid */
69     if (pid != TRACE_PID && !children.lookup(&pid))
70     {
71         return 0;
72     }
73
74     children.delete(&pid);
75
76     return 0;
77 }
78 #endif
79
80 TRACEPOINT_PROBE(raw_syscalls, sys_enter)
81 {
82     u64 pid_tgid = bpf_get_current_pid_tgid();
83
84     /* Maybe filter by PID */
85     #if defined(TRACE_PID) && defined(FOLLOW)
86     u32 pid = (pid_tgid >> 32);
87     if (pid != TRACE_PID && !children.lookup(&pid))
88     {
89         return 0;
90     }
91     #elif defined(TRACE_PID)
92     if (pid_tgid >> 32 != TRACE_PID)
93     {
94         return 0;
95     }
96     #endif
97
98     /* Don't trace self */
99     if (pid_tgid >> 32 == BPFBENCH_PID)
100     {
101         return 0;

```

```

102     }
103
104     int zero = 0;
105     struct intermediate_t *start = intermediate.lookup(&zero);
106     if (!start)
107     {
108         return 0;
109     }
110
111     /* Record pit_tgid of initiating process,
112      * we use this for error checking later */
113     start->pid_tgid = pid_tgid;
114     /* Record start time */
115     start->start_time = bpf_ktime_get_ns();
116
117     return 0;
118 }
119
120 TRACEPOINT_PROBE(raw_syscalls, sys_exit)
121 {
122     u64 pid_tgid = bpf_get_current_pid_tgid();
123
124     /* Maybe filter by PID */
125     #if defined	TRACE_PID) && defined(FOLLOW)
126     u32 pid = (pid_tgid >> 32);
127     if (pid != TRACE_PID && !children.lookup(&pid))
128     {
129         return 0;
130     }
131     #elif defined	TRACE_PID)
132     if (pid_tgid >> 32 != TRACE_PID)
133     {
134         return 0;
135     }
136     #endif
137
138     /* Don't trace self */
139     if (pid_tgid >> 32 == BPFBENCH_PID)
140     {
141         return 0;
142     }
143
144     int zero = 0;
145     int syscall = args->id;
146
147     /* Discard restarted syscalls due to system suspend */
148     if (args->id == __NR_restart_syscall)
149     {
150         return 0;
151     }
152
153     struct data_t *data = syscalls.lookup(&syscall);
154     struct intermediate_t *start = intermediate.lookup(&zero);
155     if (start && data)

```

```

156     {
157         /* We don't want to count twice for calls that return in two
           places */
158         if (pid_tgid != start->pid_tgid)
159         {
160             return 0;
161         }
162         data->count++;
163         data->overhead += bpf_ktime_get_ns() - start->start_time;
164     }
165     if (start)
166     {
167         start->pid_tgid = 0;
168         start->start_time = 0;
169     }
170
171     return 0;
172 }

```

## C Full Macro-Benchmarking Datasets

**Table C.1:** All system call overhead data from the homeostasis-3day dataset.

System Call	Count	$T_{\text{base}}$ ( $\mu\text{s}$ )	$T_{\text{ebpH}}$ ( $\mu\text{s}$ )	Diff. ( $\mu\text{s}$ )	% Overhead
futex	165360413	9564.857 (569.3384)	8989.095 (161.3186)	-575.762250	-6.019559
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
read	12023727	14.824 (3.8911)	13.446 (3.5872)	-1.378083	-9.296142
lstat	11953074	3.560 (0.2693)	4.238 (0.6952)	0.678280	19.052372
poll	11243870	2769.346 (1104.6709)	10626.125 (1978.6730)	7856.778750	283.705162
sendto	11070639	11.330 (0.4644)	12.800 (0.2013)	1.469417	12.968969
times	9085541	0.518 (0.1353)	1.712 (1.6402)	1.194250	230.587289
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
select	2695331	94941.201 (81713.2046)	95547.131 (28433.0566)	605.930000	0.638216
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

Continued on next page

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
pwrite64	1211214	33.987 (13.1969)	32.209 (15.3464)	-1.777917	-5.231141
nanosleep	831765	4337.248 (2760.6910)	6387.868 (3558.7967)	2050.620583	47.279305
fcntl	828473	1.727 (0.1388)	2.189 (0.1229)	0.461917	26.749349
munmap	799214	24.262 (9.0015)	23.642 (6.1774)	-0.620750	-2.558475
mprotect	763916	3.719 (0.3316)	4.859 (0.3913)	1.139500	30.638584
inotify_add_watch	731642	9.684 (0.3076)	10.397 (0.0379)	0.712667	7.359218
sendmsg	715096	28.613 (2.9826)	31.410 (1.7861)	2.796667	9.774140
io_submit	653975	24.521 (0.6282)	24.721 (0.5233)	0.200417	0.817338
getrusage	630232	18.067 (0.8315)	18.832 (0.4311)	0.765833	4.238949
getcwd	623946	2.435 (0.0655)	2.986 (0.0561)	0.550750	22.614200
wait4	588534	139.245 (41.9280)	128.121 (33.9076)	-11.123083	-7.988167
bpf	568667	35.044 (nan)	42.441 (0.2640)	7.396917	21.107512
pread64	529807	33.586 (13.5777)	62.706 (31.8538)	29.119167	86.699021
sched_getaffinity	519130	6.886 (0.0408)	6.540 (0.0695)	-0.345500	-5.017670
epoll_wait	511372	32147.893 (6916.3569)	27957.134 (5687.0478)	-4190.759083	-13.035875
setsockopt	499293	3.288 (0.0582)	4.058 (0.0681)	0.769667	23.407760
chdir	486415	4.782 (2.3057)	5.519 (1.9869)	0.737083	15.412898
fsync	474263	2252.371 (64.4770)	2201.437 (48.4990)	-50.933833	-2.261343
newfstatat	465724	2.720 (0.0633)	3.152 (0.1279)	0.432439	15.899481
socket	447104	22.022 (0.9650)	22.674 (0.3447)	0.651333	2.957604
readlinkat	424003	5.472 (0.2309)	6.018 (0.2487)	0.546583	9.989187
fchdir	387764	0.538 (0.1867)	1.022 (0.2632)	0.483833	89.945778
getuid	345883	1.064 (0.0683)	1.549 (0.0496)	0.484583	45.536413
geteuid	342207	0.628 (0.0181)	1.070 (0.0070)	0.441500	70.283895
getsockopt	337970	4.683 (0.3171)	4.522 (0.1451)	-0.161167	-3.441281
timerfd_settime	303645	5.899 (0.1007)	6.848 (0.0742)	0.948750	16.083689
getgid	295956	0.550 (0.0498)	0.919 (0.0065)	0.369083	67.116230
getegid	294647	0.412 (0.0547)	0.951 (0.0422)	0.538333	130.531420
chmod	285318	10.123 (3.5617)	4.148 (0.1914)	-5.974417	-59.021157
pselect6	279870	251.165 (188.7605)	155.688 (12.5032)	-95.476667	-38.013574
getdents64	254935	23.730 (6.6393)	32.190 (17.7984)	8.459667	35.649420
semop	232070	8.505 (0.7238)	10.919 (6.2202)	2.414250	28.387078
setitimer	188116	1.719 (0.0174)	1.944 (0.0270)	0.224667	13.065814
brk	187968	2.518 (0.4246)	3.461 (0.9743)	0.942833	37.440021
shutdown	146915	9.517 (0.8884)	9.753 (0.2704)	0.235583	2.475373
connect	146375	24.046 (1.6188)	22.594 (0.9292)	-1.452000	-6.038364
readlink	144049	5.951 (0.4600)	6.675 (0.2125)	0.724083	12.167423
kill	142861	2.877 (0.1130)	3.384 (0.1554)	0.507500	17.640414
epoll_ctl	139687	2.537 (0.2490)	3.178 (0.2130)	0.640167	25.229072
prlimit64	118213	0.763 (0.0538)	1.239 (0.0358)	0.475167	62.248908
getsockname	111310	2.303 (0.0919)	2.732 (0.0686)	0.428667	18.612729
fdatasync	109234	2068.600 (31.8273)	2078.008 (61.8955)	9.407500	0.454776

Continued on next page

gettid	98157	1.415 (0.0394)	2.328 (0.0727)	0.912500	64.483835
accept	92248	20.061 (0.3255)	20.050 (0.3696)	-0.010833	-0.054003
alarm	91390	2.602 (0.9071)	3.456 (0.9397)	0.853667	32.805995
unlink	71782	15.968 (1.9847)	15.066 (1.7178)	-0.901750	-5.647202
dup2	71312	0.938 (0.0508)	1.370 (0.0407)	0.431250	45.955066
accept4	70134	23.477 (0.8786)	23.794 (0.5145)	0.317167	1.350968
clone	61044	233.648 (49.8875)	192.581 (13.1702)	-41.066833	-17.576368
arch_prctl	60885	0.637 (0.0344)	1.281 (0.0412)	0.644083	101.178165
madvise	59557	8.337 (0.4086)	9.459 (0.5577)	1.121917	13.456675
getrandom	51802	4.266 (0.2140)	4.309 (0.0402)	0.042894	1.005420
uname	51600	2.191 (0.0981)	2.835 (0.0857)	0.643250	29.355391
set_robust_list	48352	0.828 (0.0442)	1.469 (0.0742)	0.641000	77.384306
timerfd_create	43215	9.049 (0.1569)	8.912 (0.0856)	-0.137333	-1.517593
kcmp	41534	0.685 (0.1117)	1.093 (0.1234)	0.407917	59.528153
setgroups	31508	4.485 (0.2355)	4.598 (0.2037)	0.112000	2.496935
execve	30754	294.823 (46.3304)	276.889 (26.6911)	-17.933750	-6.082884
pipe	30128	9.576 (0.7654)	10.268 (0.5391)	0.691500	7.220864
statfs	28140	6.026 (0.4818)	6.442 (0.3628)	0.416500	6.912194
setresuid	26758	2.876 (0.0801)	3.573 (0.0975)	0.696750	24.222847
utimensat	23176	16.632 (7.8245)	15.674 (8.0638)	-0.958417	-5.762370
prctl	23112	32.299 (12.5897)	37.082 (8.4547)	4.783417	14.809836
fchmod	21370	3.707 (1.3392)	4.759 (1.8607)	1.051417	28.362369
fchown	20985	8.342 (4.5080)	9.314 (4.9012)	0.971500	11.645423
getpeername	18616	1.345 (0.0737)	1.765 (0.1063)	0.420083	31.238768
set_tid_address	18288	0.712 (0.0497)	1.296 (0.0568)	0.584083	82.034176
getppid	17790	0.771 (0.0370)	1.250 (0.0372)	0.479667	62.240484
syslog	17000	3934.925 (105.4348)	1857.811 (56.5502)	-2077.113917	-52.786622
setresgid	16354	8.792 (1.0384)	9.924 (0.8317)	1.131750	12.871888
setgid	16336	2.603 (0.4763)	2.800 (0.1279)	0.197750	7.597976
setuid	15675	2.124 (0.2415)	2.450 (0.1094)	0.325667	15.332706
epoll_pwait	15275	783.504 (513.8938)	701.483 (443.8357)	-82.020500	-10.468422
rmdir	14971	16.843 (5.5855)	20.313 (2.0889)	3.470167	20.603122
umask	14759	0.500 (0.0279)	0.915 (0.0414)	0.415083	83.085905
dup	13776	0.892 (0.1010)	1.565 (0.0473)	0.672667	75.425154
fstatfs	13545	2.515 (0.6362)	3.084 (0.4063)	0.568667	22.611001
bind	13399	6.968 (0.3196)	7.555 (0.3560)	0.586167	8.411662
epoll_create	13348	5.717 (0.1087)	5.974 (0.2773)	0.257000	4.495103
name_to_handle_at	11975	1.939 (0.1219)	2.018 (0.0692)	0.078667	4.056551
waitid	8969	24.830 (9.1743)	21.439 (4.4987)	-3.391606	-13.659158
ftruncate	8484	47.795 (5.2608)	41.835 (2.2947)	-5.960000	-12.469837
epoll_create1	8411	4.990 (0.3246)	4.914 (0.3390)	-0.076167	-1.526437
mkdir	8075	37.098 (10.5016)	39.364 (3.1570)	2.265167	6.105845
mount	7959	13.323 (1.2319)	10.789 (0.8719)	-2.534333	-19.022242
rename	7481	77.937 (31.2395)	358.368 (463.0603)	280.430841	359.817761
sysinfo	7195	3.617 (0.1721)	3.911 (0.1018)	0.294167	8.133828

Continued on next page



signalfd4	7161	3.055 (0.0743)	3.295 (0.2081)	0.240417	7.869827
faccessat	5508	6.266 (1.7744)	8.141 (2.1988)	1.875583	29.933502
fadvise64	5449	2.892 (0.6150)	2.754 (0.8081)	-0.137500	-4.755043
capget	5438	1.160 (0.0362)	1.504 (0.0355)	0.343682	29.632388
sigaltstack	5072	0.901 (0.0337)	1.631 (0.0462)	0.730250	81.048835
socketpair	5005	16.537 (1.4244)	15.864 (1.0003)	-0.672917	-4.069281
seccomp	5002	85.576 (36.6381)	46.715 (15.2980)	-38.860833	-45.410990
setsid	4253	32.749 (0.3734)	31.422 (0.6935)	-1.327189	-4.052577
flock	4008	2.199 (0.3793)	2.700 (0.6815)	0.501250	22.797908
fallocate	3020	60.552 (22.9733)	58.087 (18.2098)	-2.465333	-4.071421
getpgrp	2888	0.687 (0.0571)	1.447 (0.0826)	0.759667	110.604222
chroot	2822	10.291 (0.5346)	10.825 (0.5676)	0.533667	5.185845
utimes	2439	6.554 (1.8607)	11.930 (4.4321)	5.375800	82.020689
eventfd2	2332	4.390 (0.4100)	4.578 (0.6410)	0.188373	4.291048
unlinkat	2311	42.960 (29.0804)	31.892 (8.8560)	-11.068250	-25.763833
getpriority	1961	0.964 (0.1168)	1.464 (0.0902)	0.499250	51.767044
pipe2	1824	9.789 (1.6143)	9.424 (1.2376)	-0.365167	-3.730378
setpriority	1740	1.624 (0.2098)	2.106 (0.2062)	0.482333	29.706426
dup3	1486	0.994 (0.1734)	1.439 (0.1485)	0.445000	44.749853
capset	1475	5.283 (0.1149)	5.943 (0.1023)	0.659667	12.486592
setpgid	1456	2.158 (0.1516)	2.784 (0.1221)	0.625833	28.998378
timer_create	1364	3.798 (0.3037)	4.647 (0.3755)	0.848833	22.351444
timer_settime	1364	3.635 (0.3187)	4.396 (0.3167)	0.761250	20.945109
keyctl	1330	9.872 (0.6101)	10.434 (1.2237)	0.562083	5.693761
msync	1000	412.459 (451.8068)	229.793 (459.2825)	-182.666400	-44.287166
inotify_rm_watch	958	12.272 (2.6979)	13.329 (0.8896)	1.057083	8.613899
setxattr	917	13.895 (3.2458)	14.535 (0.8582)	0.640083	4.606573
symlink	901	16.511 (4.6990)	16.361 (3.2118)	-0.150500	-0.911491
io_setup	768	10.353 (1.1765)	10.887 (1.6963)	0.534283	5.160704
mremap	716	4.343 (3.8611)	7.367 (3.4055)	3.024929	69.657543
removexattr	656	10.892 (3.8090)	12.710 (3.6043)	1.818417	16.695614
add_key	647	15.060 (0.8460)	14.815 (0.8831)	-0.244833	-1.625746
sched_getscheduler	596	0.725 (0.2024)	0.972 (0.3740)	0.246300	33.963045
setregid	346	2.240 (0.3999)	2.610 (0.6880)	0.370235	16.531416
setreuid	333	2.398 (0.2789)	2.799 (0.5427)	0.401144	16.731178
listen	331	2.380 (0.6436)	2.967 (0.6739)	0.586803	24.657316
mbind	312	8.921 (0.8724)	9.573 (1.4589)	0.651826	7.306576
unshare	312	176.909 (10.3202)	150.648 (16.1198)	-26.260917	-14.844322
tgkill	292	8.369 (2.6964)	8.501 (2.4907)	0.132017	1.577433
sendmmsg	285	69.693 (5.0860)	70.061 (5.4376)	0.368083	0.528152
lchown	255	12.362 (5.4578)	21.226 (4.6904)	8.863667	71.700911
getgroups	230	0.627 (0.0841)	1.107 (0.0611)	0.479911	76.516440
renameat	192	111.234 (53.9123)	119.582 (53.9572)	8.348000	7.504900
chown	186	8.103 (2.2176)	14.442 (19.3852)	6.339117	78.235748
umount2	184	522.444 (248.8937)	652.738 (220.6495)	130.293583	24.939240

Continued on next page

getpgid	166	1.691 (0.5238)	3.606 (1.0481)	1.914750	113.237396
setfsuid	135	1.141 (0.4644)	1.252 (0.3599)	0.110167	9.652453
setfsgid	132	0.940 (0.3062)	1.157 (0.0863)	0.216600	23.032752
get_mempolicy	128	1.369 (0.1373)	2.022 (0.3528)	0.653233	47.727716
renameat2	126	13.231 (9.2806)	12.183 (10.6867)	-1.048679	-7.925664
fsetxattr	114	8.822 (2.0807)	9.897 (2.1883)	1.074333	12.177198
sched_setscheduler	105	1.841 (0.4756)	2.999 (nan)	1.158333	62.930098
fgetxattr	95	4.308 (1.8595)	4.750 (1.8358)	0.442000	10.260775
inotify_init1	82	3.812 (1.2335)	6.808 (3.6418)	2.995548	78.576157
utime	72	8.615 (2.8295)	13.382 (1.9445)	4.767000	55.331151
getxattr	63	2.986 (1.3993)	4.087 (0.6636)	1.100500	36.849155
ioprio_get	62	0.655 (0.2609)	0.796 (0.1028)	0.141083	21.531222
mknod	62	9.282 (3.5964)	8.215 (5.0489)	-1.067400	-11.499677
fchmodat	48	12.872 (4.0285)	12.813 (2.0723)	-0.059500	-0.462226
getresuid	47	636.000 (1681.0002)	1.162 (0.1115)	-634.838024	-99.817322
getresgid	45	0.448 (0.0485)	0.919 (0.0686)	0.471167	105.171131
ioprio_set	43	2.584 (0.6984)	3.105 (0.8668)	0.520778	20.153072
clock_getres	37	1.082 (0.1411)	2.524 (2.1706)	1.442250	133.294824
semctl	36	3.270 (0.4456)	3.403 (0.0491)	0.132667	4.056671
flistxattr	33	2.926 (3.7714)	1.301 (0.0653)	-1.625333	-55.547961
sched_getparam	31	1.026 (0.8521)	0.657 (0.0854)	-0.369000	-35.964912
memfd_create	31	8.433 (0.9810)	7.860 (nan)	-0.573250	-6.797498
timer_delete	18	2.322 (0.2158)	2.873 (0.1288)	0.551214	23.735851
semget	12	3.805 (0.6143)	3.935 (0.1430)	0.129333	3.398739
personality	10	0.511 (0.1032)	0.858 (nan)	0.347333	68.015666
sched_setaffinity	7	12.673 (5.4024)	14.526 (5.2623)	1.853250	14.623898
mlock	5	22.155 (2.0455)	16.791 (nan)	-5.364000	-24.211239
creat	2	53.996 (nan)	45.028 (nan)	-8.968000	-16.608638

Table C.2: All system call overhead data from the arch-3day dataset.

System Call	Count	$T_{\text{base}}$ ( $\mu\text{s}$ )	$T_{\text{ebpH}}$ ( $\mu\text{s}$ )	Diff. ( $\mu\text{s}$ )	% Overhead
recvmsg	1581788435	2.964 (1.2181)	3.101 (0.8429)	0.137255	4.631430
futex	1153914634	426.257 (433.9952)	254.204 (202.6858)	-172.053455	-40.363741
poll	637450332	450.252 (426.8380)	313.592 (222.8349)	-136.659736	-30.351858
epoll_wait	572388943	747.918 (734.5878)	554.937 (412.2118)	-192.980727	-25.802400
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
writetv	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

Continued on next page

ioctl	101386112	10.028 (2.0969)	10.558 (1.9098)	0.530000	5.285043
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
ppoll	29211927	994.708 (728.0881)	956.219 (428.0754)	-38.489118	-3.869389
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
mmap	19616351	7.108 (2.2771)	7.155 (1.2866)	0.046700	0.657006
rt_sigprocmask	19322596	0.904 (0.3942)	1.170 (0.2496)	0.266082	29.439751
sendto	18508712	16.004 (4.3331)	15.475 (2.6258)	-0.529264	-3.306996
clock_nanosleep	18388107	2118.791 (2285.1533)	1313.352 (1314.0287)	-805.438900	-38.014078
epoll_pwait	15455024	110.613 (142.5396)	133.009 (229.8098)	22.395818	20.246970
fstat	14829552	0.770 (0.2474)	1.002 (0.2450)	0.231045	29.988201
rt_sigaction	10427406	0.769 (0.3289)	0.945 (0.2235)	0.176200	22.912874
pread	10228161	0.975 (0.3771)	1.687 (0.5250)	0.712100	73.050882
nanosleep	7551879	79.607 (4.3678)	81.194 (2.1049)	1.587355	1.994000
munmap	7541621	32.795 (13.7197)	35.188 (11.3567)	2.393000	7.296844
fcntl	7019123	1.025 (0.4219)	1.137 (0.2211)	0.112491	10.975696
lstat	6498123	1.863 (0.9083)	1.681 (0.3226)	-0.182100	-9.775082
getdents64	6356074	14.202 (13.4776)	40.874 (51.4953)	26.672500	187.812023
readlink	6032931	4.919 (1.3350)	5.082 (1.1287)	0.163400	3.321813
access	5917735	2.353 (0.7906)	3.216 (1.2654)	0.862609	36.658554
socketpair	3883000	6.927 (2.3782)	7.165 (2.0906)	0.238000	3.435930
statx	3457706	4.155 (1.4674)	4.215 (1.7684)	0.059600	1.434347
select	3209767	4084.786 (165.7681)	2063.271 (371.1844)	-2021.514667	-49.488880
ftruncate	3038511	4.402 (1.5136)	4.370 (0.7055)	-0.032400	-0.736012
unlink	3001449	10.086 (3.6849)	11.317 (4.9733)	1.231200	12.207383
dup	2903144	1.150 (0.3380)	1.306 (0.2482)	0.155800	13.550183
chdir	2602453	2.595 (0.7500)	3.053 (0.5966)	0.457000	17.607397
getcwd	2592760	2.261 (0.8058)	2.593 (0.4898)	0.331500	14.659709
fstatfs	2476834	0.565 (0.0595)	0.614 (0.0307)	0.049200	8.707965
getuid	2257867	0.999 (0.3976)	1.137 (0.3254)	0.137627	13.772744
mincore	1751318	24.995 (8.4480)	22.203 (10.1664)	-2.791600	-11.168634
brk	1647181	2.171 (0.6745)	3.046 (0.8761)	0.875764	40.345940
sched_getaffinity	1456976	1.545 (0.3119)	1.661 (0.2691)	0.115200	7.454381
clock_gettime	938122	2.535 (1.2211)	2.917 (1.0884)	0.381700	15.055418
uname	888133	1.546 (0.5469)	1.873 (0.3111)	0.326136	21.090535
statfs	805016	4.882 (1.5734)	4.595 (0.8522)	-0.286400	-5.866929
quotactl	774996	1.971 (0.4720)	1.977 (0.4507)	0.006300	0.319651
geteuid	766345	0.694 (0.2369)	0.919 (0.2211)	0.224900	32.411010

Continued on next page

fadvice64	721235	0.394 (0.0725)	0.453 (0.0691)	0.058300	14.789447
bpf	698910	28.445 (2.3965)	32.355 (4.9609)	3.910222	13.746714
fallocate	697544	79.579 (13.6969)	96.844 (21.6456)	17.265400	21.695979
newfstatat	653553	2.166 (0.5787)	3.002 (1.3165)	0.836400	38.614958
getgid	629415	0.533 (0.2002)	0.677 (0.1275)	0.143900	26.977878
getegid	615838	0.522 (0.1944)	0.675 (0.1306)	0.153600	29.430925
getsockname	516755	1.653 (0.6037)	1.807 (0.4669)	0.153045	9.256103
socket	495486	9.919 (2.9520)	10.885 (2.5149)	0.965782	9.736864
tgkill	471981	6.439 (1.9738)	7.313 (0.7155)	0.873636	13.567123
prctl	463015	1.637 (0.4987)	2.112 (0.3647)	0.475455	29.052328
dup2	385341	1.498 (0.4686)	1.685 (0.3447)	0.186800	12.470792
arch_prctl	381049	0.959 (0.3157)	1.133 (0.2038)	0.174100	18.154327
prlimit64	356127	1.114 (0.2980)	1.473 (0.3399)	0.358800	32.214042
getrusage	352911	10.543 (1.3806)	10.455 (2.2438)	-0.087800	-0.832820
wait4	352070	189.392 (101.9830)	150.898 (80.6353)	-38.494045	-20.325113
setsockopt	309329	1.799 (0.3855)	1.915 (0.3828)	0.116300	6.465421
bind	288387	4.466 (1.5361)	4.967 (0.8361)	0.500809	11.213589
clone	279682	182.837 (50.9671)	169.594 (46.2882)	-13.242727	-7.242925
getsockopt	269406	1.346 (0.2717)	1.539 (0.2333)	0.193000	14.340912
getpeername	266794	1.014 (0.4244)	1.020 (0.1482)	0.006300	0.621363
epoll_create1	262265	5.366 (1.7271)	5.801 (0.9044)	0.435355	8.113892
set_robust_list	257765	0.940 (0.2987)	1.213 (0.2947)	0.272600	29.000000
chmod	254583	2.636 (0.8072)	2.611 (0.4255)	-0.025200	-0.955958
connect	236663	9.504 (2.2393)	10.303 (1.6276)	0.798300	8.399356
execve	195762	284.836 (72.0236)	292.361 (62.8953)	7.524900	2.641835
getrandom	181029	2.697 (2.0750)	2.268 (0.9067)	-0.428645	-15.896096
rename	145515	78.842 (25.2564)	101.074 (32.9050)	22.232555	28.199034
pwrite	143187	19.871 (6.5413)	22.931 (12.5533)	3.060200	15.400177
mkdir	110833	7.485 (2.1245)	5.352 (2.5529)	-2.133700	-28.504823
pipe2	106580	7.725 (1.7620)	8.167 (1.9245)	0.442400	5.726861
set_tid_address	96643	0.931 (0.2872)	1.136 (0.2117)	0.204900	21.996779
kill	94056	6.944 (2.2153)	7.549 (2.7507)	0.604545	8.705442
mremap	92326	1.307 (0.8703)	1.111 (0.4992)	-0.196667	-15.044624
pipe	91285	6.426 (1.9846)	7.346 (1.5886)	0.919645	14.310308
inotify_add_watch	88453	7.338 (1.5806)	7.515 (1.0037)	0.176700	2.407849
sysinfo	77370	2.240 (0.4693)	2.600 (0.2817)	0.359900	16.064813
setpriority	62110	1.841 (0.5345)	2.077 (0.3978)	0.237000	12.876936
getppid	59182	1.188 (0.3525)	1.523 (0.2649)	0.335200	28.215488
getpgrp	50166	1.088 (0.3430)	1.329 (0.2089)	0.241800	22.232438
shutdown	48317	2.239 (0.6166)	2.478 (0.4161)	0.238800	10.666428
setsid	47385	19.503 (6.3805)	23.045 (4.5943)	3.542855	18.166114
accept	42669	6.974 (1.4040)	7.212 (1.0157)	0.238000	3.412480
mlock	40158	1.584 (nan)	11.873 (12.2713)	10.288667	649.537037
munlock	40136	1.183 (nan)	4.303 (4.3049)	3.120000	263.736264
getpriority	38439	1.037 (0.2788)	1.201 (0.2328)	0.164100	15.830600

Continued on next page

alarm	32861	1.114 (0.7017)	1.124 (0.2229)	0.010289	0.923930
rmdir	32321	4.789 (1.8910)	6.170 (3.3836)	1.380400	28.823186
accept4	31871	8.109 (2.0107)	8.310 (1.1277)	0.201000	2.478758
umask	26058	0.500 (0.1909)	0.548 (0.1504)	0.047900	9.578084
sigaltstack	22842	1.181 (0.4533)	1.228 (0.3013)	0.047000	3.977994
unlinkat	21483	4.268 (1.8606)	8.449 (5.2968)	4.180300	97.933700
eventfd2	21004	2.344 (0.5672)	2.593 (0.5763)	0.248600	10.605350
readlinkat	20383	3.852 (1.2550)	4.772 (1.6175)	0.920400	23.895322
shmctl	17817	1.114 (0.3635)	1.077 (0.2955)	-0.036900	-3.313875
shmat	17811	8.006 (2.6191)	7.958 (2.5415)	-0.047300	-0.590844
shmdt	17517	116.159 (90.8843)	181.750 (173.6531)	65.590900	56.466287
waitid	16406	16.620 (5.2803)	19.179 (8.5403)	2.559200	15.398315
capget	12660	1.070 (0.5001)	1.222 (0.3610)	0.152200	14.224299
kcmp	12435	0.782 (0.2690)	0.919 (0.2704)	0.137800	17.632758
memfd_create	11916	3.845 (0.8666)	4.199 (0.8787)	0.353100	9.182161
name_to_handle_at	10587	0.976 (0.6553)	0.809 (0.0823)	-0.166900	-17.102162
sendmmsg	10343	36.938 (13.5596)	36.736 (5.0793)	-0.202100	-0.547133
fsync	10298	4990.939 (1090.4855)	4503.783 (945.2196)	-487.156375	-9.760816
shmget	8900	8.393 (2.7209)	8.694 (2.9172)	0.300500	3.580151
getxattr	8385	3.475 (2.0258)	3.771 (1.4989)	0.296700	8.538621
setpgid	6325	2.378 (0.7939)	2.777 (0.6730)	0.399744	16.812232
mount	5446	3.413 (0.3336)	3.470 (0.6521)	0.056667	1.660156
fdatasync	5148	2214.513 (406.2752)	1957.492 (182.1718)	-257.021200	-11.606216
seccomp	4618	38.161 (17.8770)	48.395 (18.0017)	10.234600	26.819598
times	3704	10.924 (2.3170)	5.509 (4.3076)	-5.415179	-49.572524
renameat2	3631	4.360 (2.6561)	8.606 (9.1713)	4.245333	97.365068
getgroups	3512	0.311 (0.0703)	0.470 (0.1792)	0.159000	51.125402
utimensat	3385	8.766 (5.4400)	8.127 (5.6680)	-0.639500	-7.294815
link	3196	5.357 (0.6636)	6.804 (3.6961)	1.447264	27.018204
splice	2451	27.920 (26.0124)	17.747 (9.6127)	-10.172267	-36.434142
flock	2398	1.182 (0.4555)	1.243 (1.1375)	0.060946	5.157849
sched_getscheduler	2261	0.514 (0.1910)	0.655 (0.2537)	0.141000	27.426571
getresgid	2200	0.522 (0.1408)	0.639 (0.1931)	0.117133	22.453674
getresuid	2199	0.823 (0.1626)	0.926 (0.1610)	0.103211	12.542533
inotify_rm_watch	1834	13.258 (4.6938)	13.187 (3.5123)	-0.070700	-0.533275
symlink	1828	10.715 (5.1349)	12.937 (7.3054)	2.222500	20.742338
faccessat	1786	3.413 (0.9939)	3.619 (1.4270)	0.206200	6.042314
sched_setattr	1465	3.444 (1.1068)	3.543 (0.9176)	0.099500	2.889502
fchmod	1426	3.247 (3.0129)	2.335 (1.7384)	-0.911861	-28.082227
dup3	1296	1.727 (0.6069)	2.029 (0.5720)	0.302800	17.538372
inotify_init	1208	3.263 (1.2634)	3.211 (0.6462)	-0.051800	-1.587496
getpgid	1175	0.633 (0.2693)	1.097 (0.4017)	0.463200	73.129144
epoll_create	1102	2.209 (0.6462)	2.382 (0.4900)	0.173000	7.831953
utimes	1008	1.691 (0.0720)	3.455 (4.7928)	1.763306	104.260491
timerfd	935	2.152 (0.6294)	2.196 (0.6670)	0.043900	2.039678

Continued on next page

fchown	906	8.562 (6.3805)	3.895 (2.5272)	-4.666571	-54.504196
msync	820	560.886 (106.0623)	604.759 (75.3636)	43.872833	7.822052
capset	769	1.974 (0.6319)	2.260 (0.8072)	0.285900	14.486218
sched_setscheduler	734	2.020 (0.7744)	2.774 (1.2989)	0.754200	37.340331
setresuid	594	1.077 (0.6333)	1.862 (1.0768)	0.785143	72.920260
utime	565	4.578 (1.1228)	5.938 (1.9210)	1.360133	29.712372
listen	563	1.117 (0.4182)	1.252 (0.2490)	0.135667	12.148045
sched_getattr	561	1.764 (0.7531)	1.906 (0.5220)	0.141700	8.031514
setresgid	560	0.913 (0.3810)	1.529 (0.5204)	0.616429	67.548528
mknod	518	2.152 (0.5041)	2.181 (0.5006)	0.028625	1.330158
umount2	511	99.740 (80.9811)	123.978 (83.9955)	24.237306	24.300379
fchdir	486	0.697 (0.1698)	1.322 (0.6761)	0.624792	89.597275
setxattr	467	4.740 (3.5943)	4.230 (1.7825)	-0.510575	-10.771170
sched_get_priority_min	464	0.523 (0.3039)	0.647 (0.2750)	0.123667	23.645634
sched_get_priority_max	464	0.392 (0.2729)	0.525 (0.2460)	0.132278	33.701345
signalfd4	452	1.706 (0.5179)	1.945 (0.4618)	0.238900	14.006801
inotify_init1	313	3.180 (0.4790)	3.309 (0.4687)	0.129000	4.057029
keyctl	237	4.616 (2.6928)	3.685 (1.5061)	-0.930875	-20.166269
chown	227	10.721 (nan)	10.145 (9.9877)	-0.575750	-5.370301
removexattr	212	3.555 (2.5681)	2.107 (1.1652)	-1.447825	-40.729879
setgroups	173	2.361 (1.4779)	2.542 (1.4836)	0.181333	7.681446
unshare	166	110.662 (48.7060)	98.943 (48.5908)	-11.718556	-10.589503
creat	162	12.145 (6.5308)	3.939 (0.6329)	-8.205500	-67.562783
timer_delete	153	4.474 (3.6236)	2.562 (1.3384)	-1.911444	-42.727634
timer_create	153	1.551 (0.1406)	1.562 (0.0717)	0.011111	0.716384
chroot	148	5.674 (2.5170)	6.894 (7.9650)	1.219944	21.502291
readahead	134	7.488 (9.5739)	20.021 (32.7488)	12.532875	167.361621
add_key	117	6.293 (2.9372)	5.218 (1.5291)	-1.074375	-17.073897
personality	70	0.291 (0.0599)	0.384 (0.0530)	0.092889	31.908397
sched_getparam	55	0.737 (0.1475)	1.005 (0.1522)	0.268086	36.389374
fchownat	53	5.012 (nan)	5.849 (3.4310)	0.837333	16.706571
pselect6	50	4.612 (nan)	3.816 (nan)	-0.796000	-17.259324
setuid	46	1.287 (0.2118)	1.684 (1.0236)	0.396667	30.813050
setgid	42	1.687 (0.9743)	1.126 (0.6118)	-0.561000	-33.254298
ioprio_set	39	1.614 (1.1916)	1.663 (0.7974)	0.049900	3.092464
lchown	37	2.304 (nan)	2.536 (0.6222)	0.232400	10.086806
fsetxattr	33	2.908 (0.6858)	3.566 (1.9079)	0.658450	22.644267
getsid	32	0.586 (0.1004)	0.848 (0.5375)	0.262250	44.752560
ioperm	28	1.386 (0.0419)	1.240 (0.1513)	-0.146250	-10.550045
fgetxattr	15	2.635 (0.4325)	2.727 (0.4321)	0.091750	3.482304
fchmodat	14	16.657 (nan)	6.085 (1.0062)	-10.572500	-63.471814
iopl	14	0.355 (0.0294)	0.353 (0.0368)	-0.002000	-0.563380
setfsuid	5	0.471 (nan)	1.037 (nan)	0.566000	120.169851
clock_getres	3	1.017 (nan)	1.629 (nan)	0.612000	60.176991