Extended Berkeley Packet Filter for Intrusion Detection Implementations

COMP4906 Honours Thesis Proposal

by

William Findlay

November 29, 2019

Under the supervision of Dr. Anil Somayaji
Carleton University

Abstract

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

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

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

Acknowledgments

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 proposal has been a long process and not without its challenges. Dr. Somayaji's support and guidance has 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 (https://github.com/yonghong-song) and Teng Qin (https://github.com/palmtenor) for their guidance and willingness to respond to issues and questions related to the *bcc* project. Sasha Goldstein's (https://github.com/goldshtn) *syscount.py* was also 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. 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.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction and Motivation

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

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

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

A prototype of ebpH has been written using the Python interface provided by *bcc* (*BPF Compiler Collection*) [2], and preliminary tests show that it is capable of monitoring system state under moderate to heavy workloads with negligible overhead. What's more, zero kernel panics occurred during ebpH's development and early testing, which simply would not have been possible without the safety guarantees that eBPF provides. The rest of this proposal will cover the necessary background material required to understand ebpH, describe several aspects of its implementation, including the many findings and pitfalls encountered along the way, and discuss the planned methodology for testing and iterating on this prototype going forward.

2 Background

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

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

2.1 An Overview of the Linux Tracing Landscape

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

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

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

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

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

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

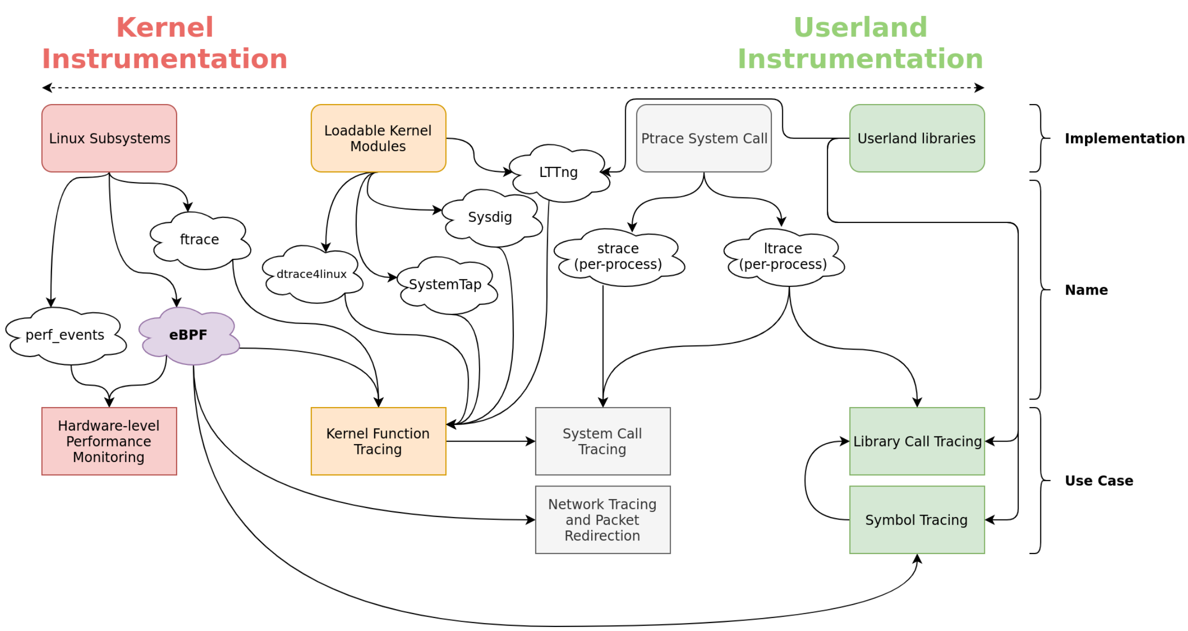

Figure 2.1: A high level overview of the broad categories of Linux instrumentation. This does not represent a complete picture of all available tools and interfaces, but instead presents many of the most popular ones. Note how eBPF covers every presented use case.

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

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

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

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

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

2.1.1 Dtrace

It is worth spending a bit more time comparing eBPF with Dtrace, as both APIs are quite full-featured and designed with similar functionality in mind. dtrace4linux [13] is a free and open source port of Sun's Dtrace for the Linux Kernel, implemented as a loadable kernel module (LKM). While Dtrace offers a powerful API for full-system tracing, its usefulness is, in general, eclipsed by that of eBPF [18] and requires extensive shell scripting for use cases beyond one-line tracing scripts. In contrast, with the help of powerful and easy to use front ends like bcc [2], developing complex eBPF programs for a wide variety of use cases is becoming an increasingly painless process.

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

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

2.2 eBPF: Linux Tracing Superpowers

In 2016, eBPF was described by Brendan Gregg [21] as nothing short of *Linux tracing superpowers*. I echo that sentiment here, as it summarizes eBPF's capabilities perfectly. Through eBPF programs, we can simultaneously trace userland symbols and library calls, kernel functions and data structures, and hardware performance. What's more, through an even newer subset of eBPF, known as *XDP* or *Express Data Path* [22], we can inspect, modify, redirect, and even drop packets entirely before they even reach the main kernel network stack. Figure 2.2 provides a high level overview of these use cases and the corresponding eBPF instrumentation required.

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

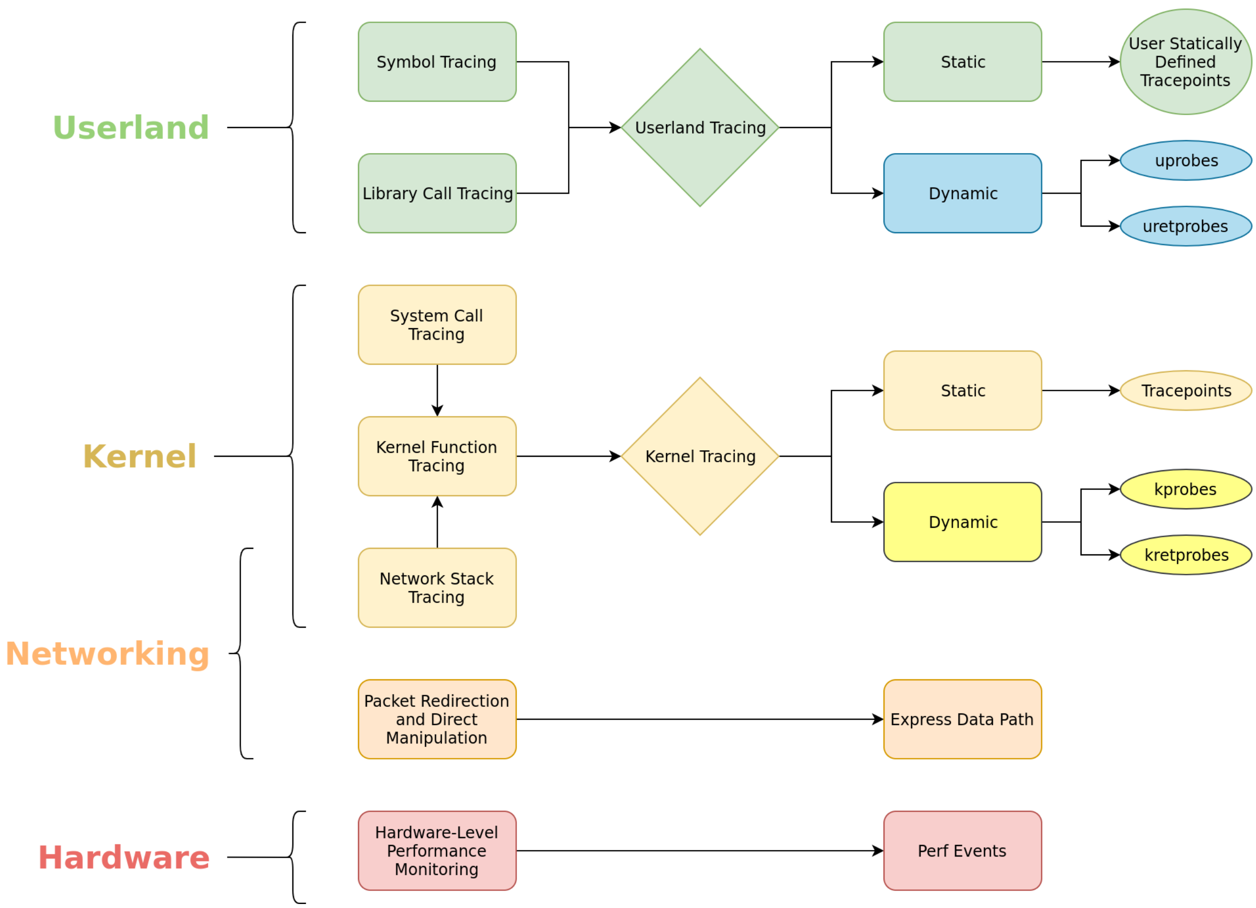

Figure 2.2: A high level overview of various eBPF use cases. Note the high level of flexibility that eBPF provides with respect to system tracing.

Safety is guaranteed with the help of an in-kernel verifier that checks all submitted bytecode before its insertion into the BPF virtual machine. While the verifier does limit what is possible (eBPF in its current state is **not** Turing complete), it is constantly being improved; for example, a recent patch [23] that was mainlined in the Linux 5.3 kernel added support for verified bounded loops, which greatly increases the computational possibilities of eBPF. The verifier will be discussed in further detail in Subection 2.2.2.

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

2.2.1 How eBPF Works at a High Level

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

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

The main data structure used in eBPF is the map; these maps are used to store data as well as for communication between userspace and the eBPF program. There are several map types available in eBPF programs which cover a wide variety of use cases. These map types along with a brief description are provided in Table 2.2 [2], [24], [25]. Thanks to this wide arsenal of maps, eBPF developers have a powerful set of both general-purpose and specialized

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

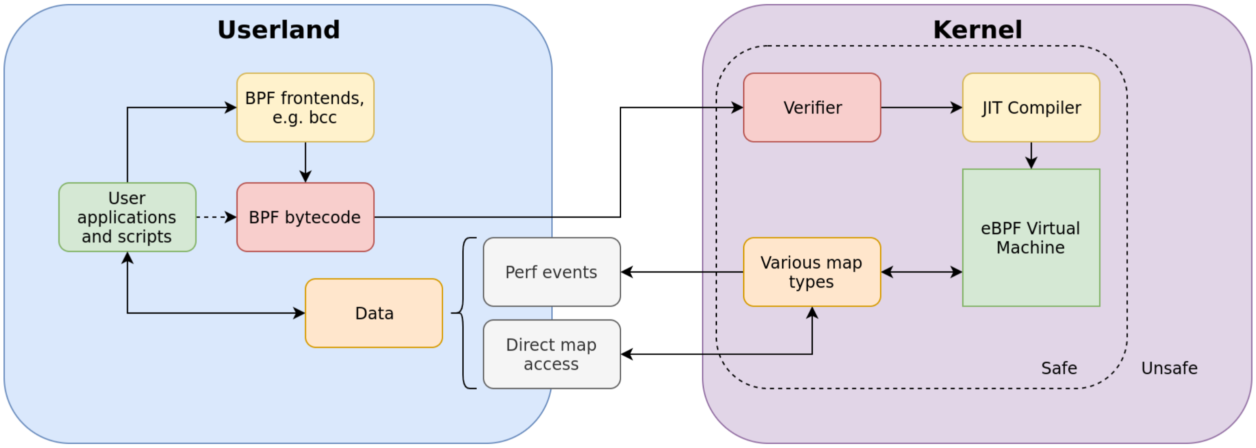

Figure 2.3: Basic topology of eBPF with respect to userland and the kernel. Note the bidirectional nature of dataflow between userspace and kernelspace using maps.

data structures at their disposal; as we will see in coming sections, many of these maps are quite versatile and have use cases beyond what might initially seem pertinent. For example, the `ARRAY` map type may be used to initialize large data structures to be copied into a general purpose `HASH` map (refer to Listing A.1 in Appendix A). This can be effectively used to bypass the verifier's stack space limitations, which are discussed in detail in Subection 2.2.2.

Table 2.2: Various map types available in eBPF programs.

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

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

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

Proving the safety of arbitrary code is by definition a difficult problem. This is thanks in part to theoretical limitations on what we can actually prove; a famous example is the halting problem described by Turing circa 1937 [26]. This difficulty is further compounded by stricter requirements for safety in the context of eBPF. In fact, the problem that we are effectively trying to solve is one of *untrusted* code running in the kernel, an implicitly trusted environment.

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. Program B effectively constitutes a denial of service attack [27] on our system, intentional or otherwise; allowing untrusted users to load this code into our kernel spells immediate disaster for our system.

While we have established that verifying the safety of eBPF programs is an important problem to solve, the question remains as to whether it is *possible* to solve. For the reasons outlined above, this problem should intuitively seem impossible, or at least far more difficult than should be feasible. So, what can we do? The answer is to *change the rules* to make it easier. In particular, while it is difficult to prove that the set of all possible eBPF programs are safe, it is much easier[2] to prove this property for a subset of all eBPF programs. Figure 2.4 depicts the relationship between potentially valid eBPF code and verifiably valid eBPF code.

The immediate exclusion of eBPF programs meeting certain criteria is the crux of eBPF's safety guarantees. Unfortunately, it also rather intuitively limits what we are actually able to do with eBPF programs. In particular, eBPF is not a Turing complete language; it prohibits jump instructions, cycles in execution graphs, and unverified memory access. Further, we limit stack allocations to only 512 bytes – far too small for many practical use cases. From a security perspective, these limitations are a *good thing*, because they allow us to immediately

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

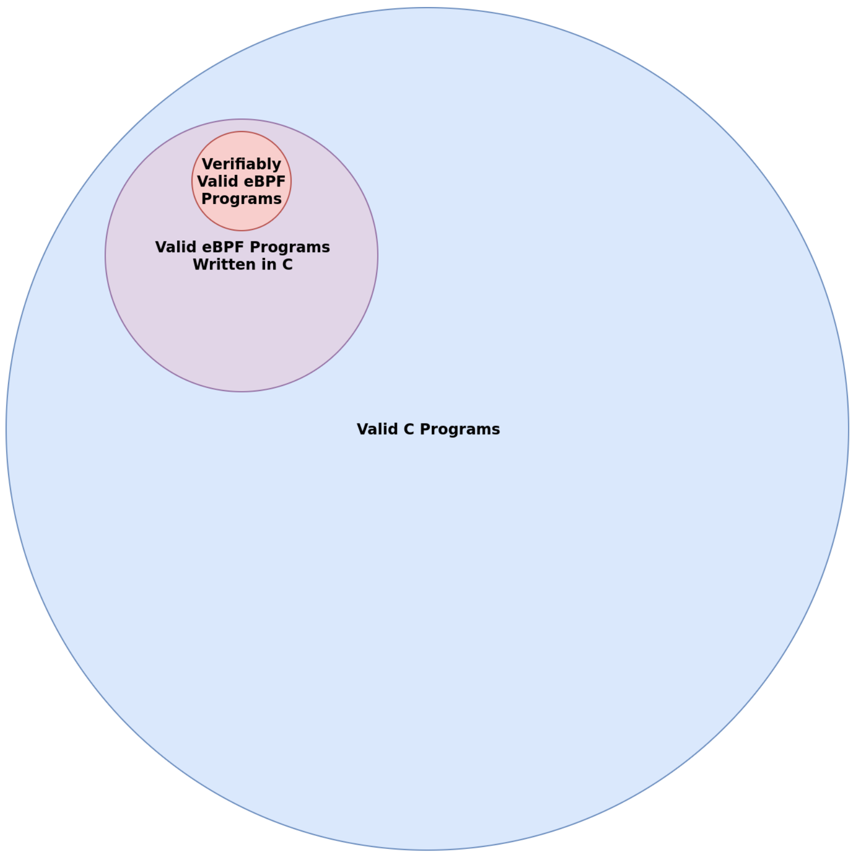

Figure 2.4: The set participation of valid C and eBPF programs. Valid eBPF programs written in C constitute a small subset of all valid C programs. Verifiably valid eBPF programs constitute an even smaller subset therein.

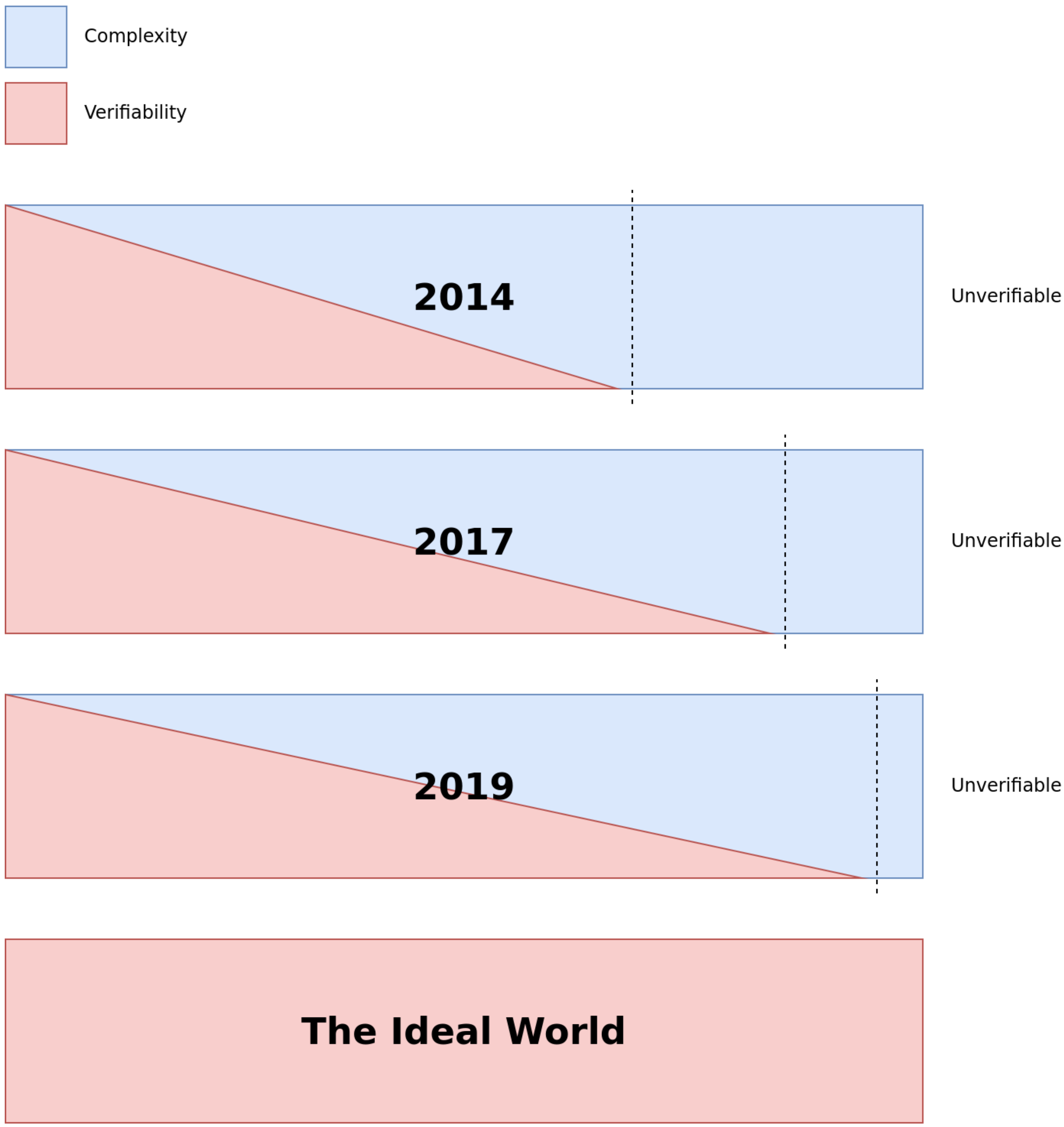

Figure 2.5: Complexity and verifiability of eBPF programs. Safety guarantees for eBPF programs rely on certain compromises. Ideally we would like to have a relationship as shown on the bottom; in practice, we have something that is getting closer over time, but is still far from the ideal.

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 2.5). When we say *better*, what we mean is that it is able to prove the safety of increasingly complex programs. Perhaps the best example of this steady improvement is a recent kernel patch [23] that added support for bounded loops in eBPF programs. With this patch, the set of viable eBPF programs was *greatly* increased; in fact, ebpH in its current incarnation relies heavily on bounded loop support. Prior to bounded loops, eBPF programs relied on *unrolling* loops at compile time, a technique that was both slow and highly limited. This is just one example of the critical work that is being done to improve the verifier and thus improve eBPF as a whole.

2.3 System Calls

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

At the time of writing this paper, the Linux kernel [28] supports an impressive 436 distinct system calls, and this number generally grows with subsequent releases. In general, userspace implements 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.

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 Subection 2.1, perhaps most notably strace. We will discuss the behavioral implications of system calls further in Subection 2.5.2.

2.4 Intrusion Detection

At a high level, intrusion detection systems (IDS) strive to monitor systems at a particular level and use observed data to make decisions about the legitimacy of these observations [29]. IDS can be broadly classified into several categories based on data collection, detection technique(s), and response. Figure 2.6 presents a broad and incomplete overview of these categories.

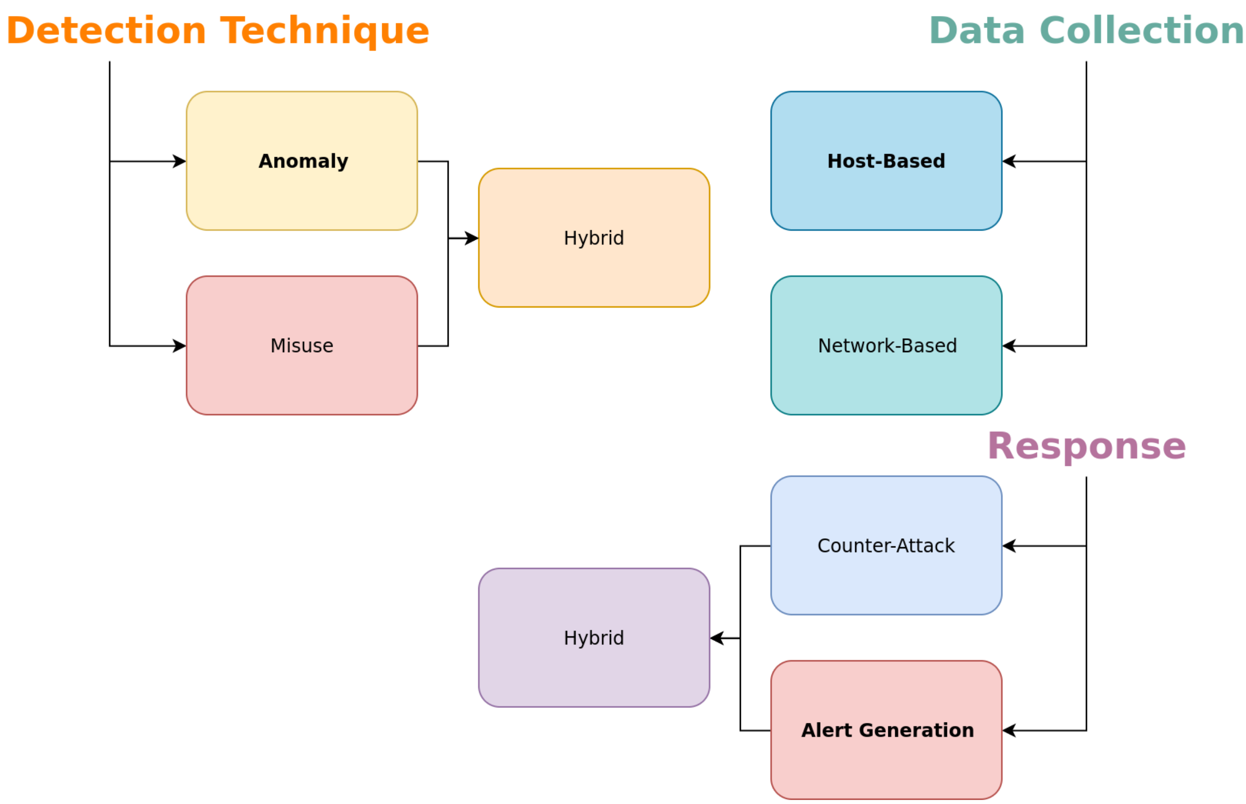

Figure 2.6: A broad overview of the basic categories of IDS. The current version of ebpH can be classified according to the categories labeled in **boldface**. Note that intrusion detection system classification can often be more nuanced than the basic overview presented here. However, this should present a good enough overview to understand IDS in the context of ebpH.

In general, intrusion detection systems can either attempt to detect anomalies (i.e. mismatches in behavior when compared to normally observed patterns) or misuse, which generally refers to matching known attack patterns to observed data [29]. In general, anomaly-based approaches cover a wider variety of attacks while misuse-based approaches tend to yield fewer false positives. A hybrid approach between these two techniques is also possible.

Data collection is generally either host-based or network based. Network-based IDS examine network traffic and analyze it to detect attacks or anomalies. In contrast, host-based attacks analyze the state or the local system [3], [29].

Responses can vary quite a bit based on the system, but can be classified into two main categories: alerts and counter-attacks. Systems can either choose to alert and 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 (Subection 2.5) with one major exception: As we will see, the original pH also responds to anomalies by delaying system calls [3] and preventing anomalous `execve`s. Implementing this functionality in ebpH is a topic for future work (refer to Subection 4.2).

2.5 Process Homeostasis

Anil Somayaji's *Process Homeostasis* [3], styled as *pH*, forms the basis for ebpH's core design; as such, it is worth exploring the original implementation, design choices, 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 [3]. It is in this way that pH lives up to its name: These delays make process behavior *homeostatic*.

2.5.1 Homeostasis

The concept of homeostasis in pH is inspired by the biological process of the same name [3].

TODO: come back here when more inspired

2.5.2 Anomaly Detection Through Lookahead Pairs

pH uses a technique known as *lookahead pairs* [3], [30] 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 [30] 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. These longer window lengths also significantly reduce the potential for mimicry attacks masking malicious system call sequences. This comes at the expense of potentially missing out on

2.5.3 System Call Delays

2.6 Other Related Work

3 Implementing ebpH

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

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

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

3.1 Userspace Components

The userspace components of ebpH are comprised of two distinct programs. The **ebpH Daemon** (*ebpHD*) is responsible for initially compiling and submitting the eBPF program,

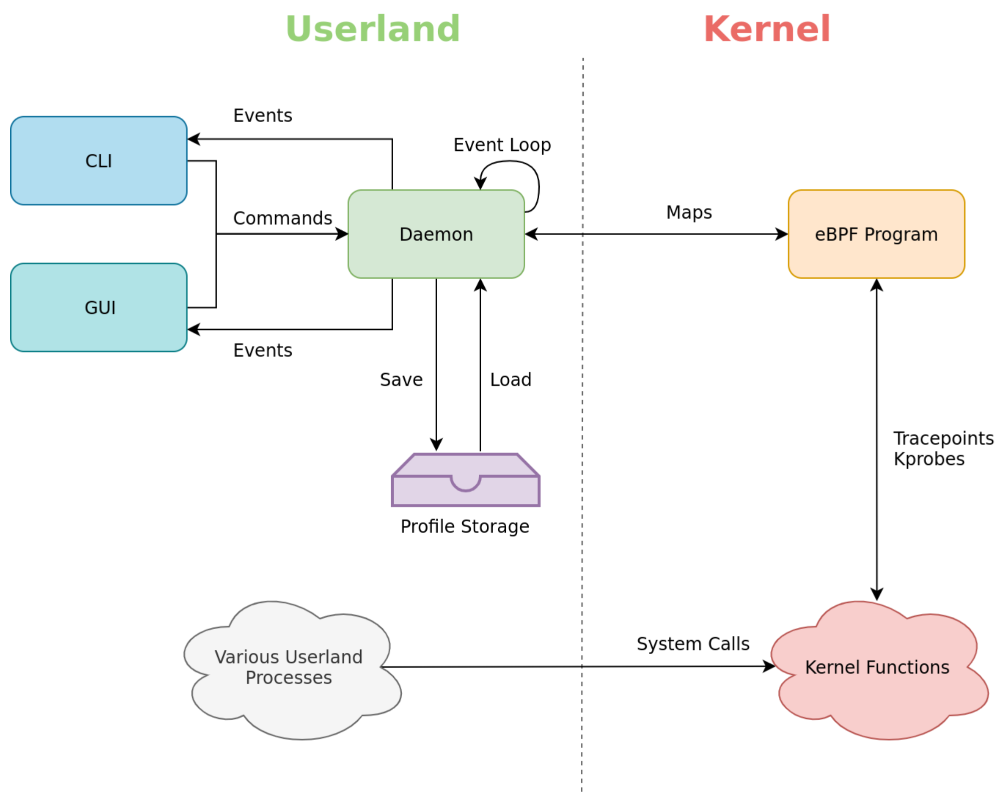

Figure 3.1: The dataflow between various components of ebpH.

as well as communication between userspace and the in-kernel eBPF program. As part of this communication, it loads existing profiles from the disk and saves new and modified profiles to disk at regular intervals. Users can interact with the daemon either directly on the command line or using the **ebpH GUI**. The GUI performs the same basic functions as the command line interface, except it presents information and commands in the form of a graphical user interface.

3.1.1 The ebpH Daemon

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

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

Table 3.1: Main event categories in ebpH.

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

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

The daemon 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.

3.1.2 The ebpH GUI

The ebpH GUI (hereafter referred to as the GUI) provides a graphical user interface for interacting with the daemon. This GUI is still a work in progress and will be improved considerably during the testing and iteration phase (see Section 4). In its current incarnation, the GUI can be used to inspect process profiles, examine the overall state of ebpH, and check the ebpH logs. It can also be used to issue rudimentary commands such as profile deletion. Future versions of the GUI will include more commands for controlling the state of ebpH, as well as increased system visibility and more information about process profiles. Figure 3.2 depicts an early version of the GUI.

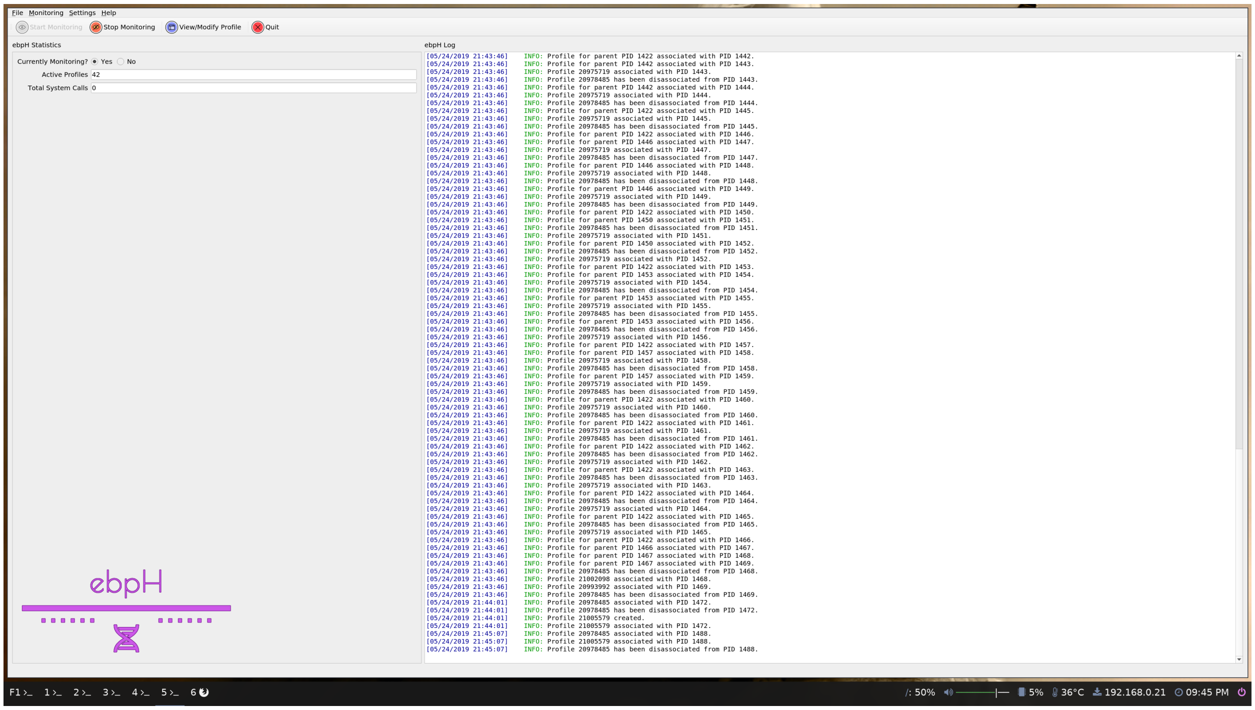

Figure 3.2: A screenshot of an early version of the ebpH GUI.

3.2 ebpH Profiles

In order to monitor process behavior, ebpH keeps track of a unique profile (Listing 3.1) for each executable on the system. It does this by maintaining a hashmap of profiles, hashed by a unique per-executable ID; this ID is a 64-bit unsigned integer which is calculated as a unique combination of filesystem device number and inode number:

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

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

The profile itself is a C data structure that keeps track of information about the executable, as well as a sparse two-dimensional away of lookahead pairs [30] to keep track of system call patterns. Each entry in this array consists of an 8-bit integer, with the i^{th} bit corresponding to

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

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

a previously observed distance i between the two calls. When we observe this distance, we set the corresponding bit to `1`. Otherwise, it remains `0`. Each profile maintains lookahead pairs for each possible pair of system calls. Figure 3.3 presents a sample (`read`, `close`) lookahead pair for the `ls` binary.

Each process (Subection 3.3) is associated with exactly one profile at a time. Profile association is updated whenever we observe a process making a call to `execve`. Whenever a process makes a system call, ebpH looks up its associated profile, and sets the appropriate lookahead pairs according to the process' most recent system calls. This forms the crux of how ebpH is able to monitor process behavior.

Just like in the original pH [3], 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.

3.3 Tracing Processes

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

ebpH monitors process behavior by instrumenting tracepoints for both system call entry and return. The nine most recent system calls made by each process are stored in its respective

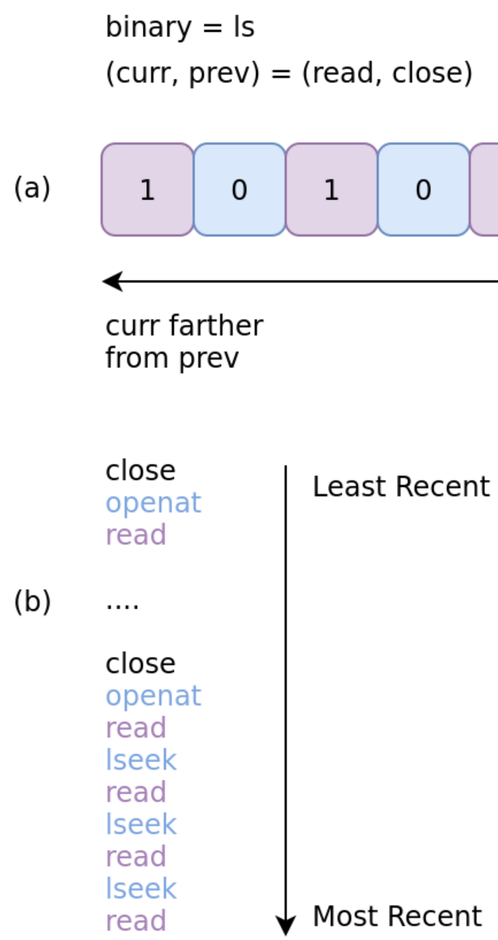

Figure 3.3: A sample (`read`, `close`) lookahead pair in the ebpH profile for `ls`. (a) shows the lookahead pair and (b) shows two relevant system call sequences, separated by several omitted calls. Note that the first three system calls in both the first and second sequence are responsible for the two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

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

```
struct ebpH_process
{
    long seq[9];   /* Remember 9 most recent system calls in order */
    u8 count;      /* How many system calls are in our sequence? */
    u32 pid;       /* What is our PID? */
    u64 profile_key; /* Associated profile key */
    u8 in_execve; /* Are we in the middle of an execve? */
};
```

process struct, and are used to set the correct lookahead pairs in the associated profile struct.

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

Table 3.2: Important system calls in ebpH.

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

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

The entry and exit points to the `execve` system call are used to differentiate a true `execve` call from the kernel routines responsible for loading shared libraries, which both invoke the aforementioned `do_open_execat` subroutine. When we first hit an `execve`, we set an indicator variable in the process struct to say that we are in the middle of an `execve`. Subsequent calls to `do_open_execat` are then ignored until we hit `execve`'s return tracepoint and unset the indicator variable.

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

Reaping Processes with `exit`, `exit_group`, and Signals We use a combination of system call tracepoints and signal handler kprobes in order to determine when to stop tracing a particular PID. This is important for a few reasons, primarily due to map size considerations; by reaping process structs from our map as we are finished with them we ensure that:

a) the map never fills up and;
b) the map does not consume more memory than necessary.

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

3.4 Training, Testing, and Anomaly Detection

ebpH profiles are tracked in two phases, *training mode* and *testing mode*. Profile data is considered training data until the profile becomes normal (as described in Subection 3.2). Once a profile is in testing mode, the lookahead pairs generated by its associated processes are compared with existing data. When mismatches occur, they are flagged as anomalies which are reported to userspace via a perf event buffer. The detection of an anomaly also prompts ebpH to remove the profile's normal flag and return it to training mode.

3.4.1 A Simple Example of ebpH Anomaly Detection

As an example, consider the simple program shown in Listing 3.3. This program's normal behavior is to simply print a message to the terminal. However, when issued an extra argument (in practice, this could be a secret keyword for activating a backdoor), it prints one extra message. This will cause a noticeable change in the lookahead pairs associated with the program's profile, and this will be flagged by ebpH if the profile has been previously marked normal.

Listing 3.3: A simple program to demonstrate anomaly detection in ebpH.

```c
/* anomaly.c */

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

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

    return 0;
}
```

In order to test this, we artificially lower ebpH's normal time to three seconds instead of one week. Then, we run our test program several times with no arguments to establish normal behavior. Once the profile has been marked as normal, we then run the same test program with an argument to produce the anomaly. ebpH immediately detects the anomalous system calls and flags them. These anomalies are then reported to userspace via a perf buffer as shown in Figure 3.4.

```
2019-11-26 12:06:29 - INFO: Loaded profiles
2019-11-26 12:06:29 - INFO: BPF program initialized
2019-11-26 12:06:43 - INFO: Constructed profile for anomaly (32381778)
2019-11-26 12:07:17 - WARNING: PID 1417900 (anomaly 32381778): 5 anomalies detected for syscall 1
2019-11-26 12:07:17 - WARNING: PID 1417900 (anomaly 32381778): 4 anomalies detected for syscall 231
```

Figure 3.4: The flagged anomalies in the `anomaly` binary as shown in the ebpH logs.

From here, we can figure out exactly what went wrong by inspecting the system call sequences produced by `anomaly.c` in both cases and comparing them with their respective lookahead pair patterns. Figure 3.5 provides an example of this comparison.

While this contrived example is useful for demonstrating ebpH's anomaly detection, process behavior in practice is often more nuanced. ebpH collects at least a week's worth of data about a processes 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 we take our time before deciding whether a process' profile has reached acceptable maturity for testing, we dramatically decrease the probability of any false positives.

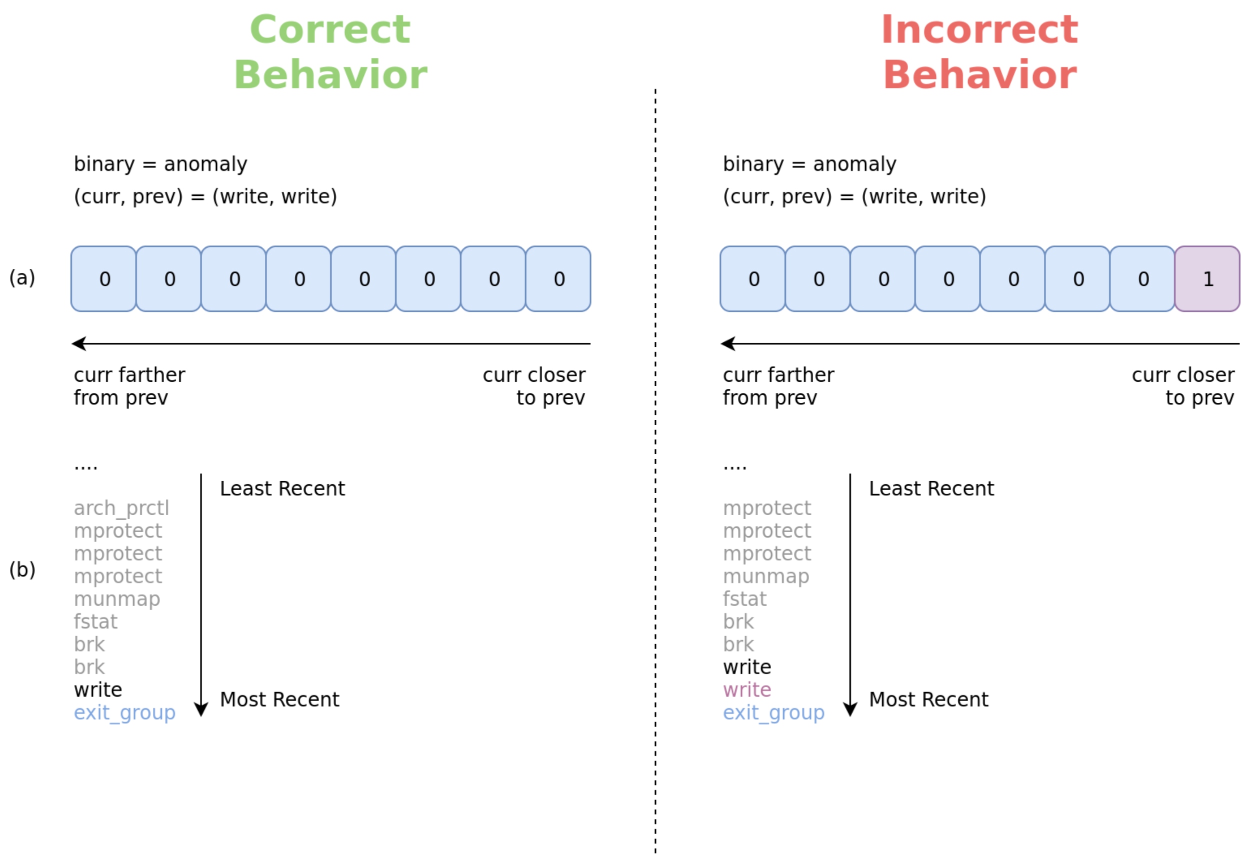

Figure 3.5: Two sample (`write`, `write`) lookahead pairs in the ebpH profile for `anomaly.c`. (a) shows the lookahead pair and (b) shows two relevant system call sequences. The left hand side depicts normal program behavior, while the right hand side depicts our artificially generated anomaly. There are several other anomalous lookahead pairs which result from this extra write call, but we focus on (`write`, `write`) for simplicity.

3.5 Soothing the Verifier

The development of ebpH elicited many challenges with respect to the eBPF verifier. As we have seen in Subection 2.2.2, eBPF programs become more difficult to verify as they increase in complexity; as a corollary, when developing large and complex eBPF programs, a great deal of care and attention must be paid to ensure that the verifier will not reject our 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) Dynamic memory allocation is prohibited and memory access requires strict safety checks;
4) Support for bounded loops is in its infancy and loops will not work without an easy proof that the induction variable will result in termination;
5) The verifier tends to err on the side of caution and will produce false positives with alarming frequency.

Subproblem (1) means that, at the moment, there is no simple means of injecting system call delay into system calls from within the eBPF program, an important part of the original pH's functionality [3]. Kernel scheduling and delay functions do not work in eBPF due to unsafe jump instructions, and so other means of delaying processes need to be explored. This is currently a topic for future work (see Subection 4.2).

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, how do we instantiate the large structs described in previous sections? Both the `ebpH_profile` and `ebpH_process` structs are larger than would be allowed in the eBPF stack. Fortunately, we can creatively solve this problem by using a `BPF_ARRAY` for initialization. Since a `BPF_ARRAY`'s entries are preinitialized with `0`, we can create an array of size `1` for each large datatype and copy its contents into the entries of a larger hashmap as needed. This technique constitutes the design pattern outlined in Listing A.1 of Appendix A.

On the topic of memory, another convenient feature of eBPF maps is the ability to flag them as being implicitly dynamically allocated. This means that the map will only use as much space as its current amount of entries requires. Memory management is handled automatically by the map. This combined with the aforementioned method of struct initialization gives us the means by which to safely and efficiently handle large data structures in eBPF programs.

From subproblem (4), we have the obvious issue that loops need to "simple" enough for the eBPF verifier to reason about them. Loops need to have an induction variable that can be shown to invariably lead to termination. For example, a loop like the one shown in Listing 3.4 would be prohibited.

Loops may also be rejected by the verifier for other reasons. For example, loops that have entrypoints in the middle of iteration will also be flagged due to the verifier's inability to correctly identify the loop structure [31]. Ultimately, designing verifiable loops in eBPF can be a tricky process with many pitfalls, but having support for loops at all greatly increases the potential complexity and power of eBPF programs like ebpH.

Listing 3.4: An invalid bounded loop in eBPF. Although it may be obvious to us that it will eventually terminate, the induction variable is not well-defined enough for the verifier to be confident about its termination.

```
int j = 50;
/* An invalid eBPF bounded loop */
for (int i = 0; i < j; i++)
{
    /* Increase j by 1 every 3 iterations */
    if (i % 3 == 0)
        j++;
}
```

Subproblem (5) is perhaps the most difficult to reckon with, but is quite understandable from the perspective of the verifier. As we have already seen, guaranteeing the safety of arbitrary untrusted program 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. Fortunately, well-contracted eBPF do not often suffer from false positives, particularly as one learns the nuances of how the verifier works and how to coax it into accepting programs

4 Methodology and Future Work

While the ebpH prototype is certainly capable of monitoring a system for anomalies, much testing and work remains to be done in order to completely reimplement the original pH and ascertain whether eBPF is truly the best choice for implementing such an IDS. Here, we discuss the planned strategies for testing ebpH, as well as plans for iteration on the initial prototype and future work.

4.1 Planned Testing Strategy

The ebpH prototype as well as its future iterations will be heavily tested on several machines under a variety of workloads. Additional testing will be done on virtual machines to simulate systems with lower end hardware specifications. Table 4.1 summarizes the currently planned systems as their relevant specifications.

Table 4.1: A summary of the various systems that will be used to test ebpH.

System	CPU Speed	RAM	Description
arch	test	test	test

4.1.1 Gathering and Analyzing Performance Data

4.1.2 Gathering and Analyzing Profile Data

4.2 Improvements to ebpH

4.3 Improvements to the ebpH GUI

References

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

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

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

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

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

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

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

[8] *Understanding how systemtap works red hat enterprise linux 5*. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/systemtap_beginners_guide/understanding-how-systemtap-works.

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

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

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

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

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

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

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

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

[17] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad, "Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps," in *Proceedings of the Linux Symposium*, vol. 1, pp. 215–224. [Online]. Available: https://www.linuxsecrets.com/kdocs/mirror/ols2007v1.pdf#page=41.

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

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

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

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

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

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

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

[25] *Bpf(2) linux programmer's manual*, Linux, Aug. 2019.

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

[27] A. Hussain, J. Heidemann, J. Heidemann, and C. Papadopoulos, "A framework for classifying denial of service attacks," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '03, Karlsruhe, Germany: ACM, 2003, pp. 99–110, isbn: 1-58113-735-4.

DOI: 10.1145/863955.863968. [Online]. Available: http://doi.acm.org.proxy.library.carleton.ca/10.1145/863955.863968.

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

[29] R. A. Kemmerer and G. Vigna, "Intrusion detection: A brief history and overview," *Computer*, vol. 35, no. 4, supl27–supl30, Apr. 2002, ISSN: 1558-0814. DOI: 10.1109/MC.2002.1012428.

[30] 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.

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

Appendix A eBPF Design Patterns

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

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

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

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

/* Suppose this is a function where we need to use our
 * bigdata_t struct */
int some_bpf_function(void)
{
    /* We use this to look up from our
     * __bigdata_t_init array */
    int zero = 0;
    /* A pointer to a bigdata_t */
    struct bigdata_t *bigdata;
    /* The key into our main hashmap
     * Its value not important for this example */
    u64 key = SOME_VALUE;

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

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

    return 0;
}
```