Extended Berkeley Packet Filter for Intrusion Detection Implementations

COMP4906 Honours Thesis Proposal

by

William Findlay

December 9, 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 of more usable interfaces such as *bcc* (*BPF Compiler Collection*) [2] has resulted in a 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, in order to extend its functionality beyond that of the current prototype system.

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 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 (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 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 proposal, 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.

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 Homeostasis* [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]

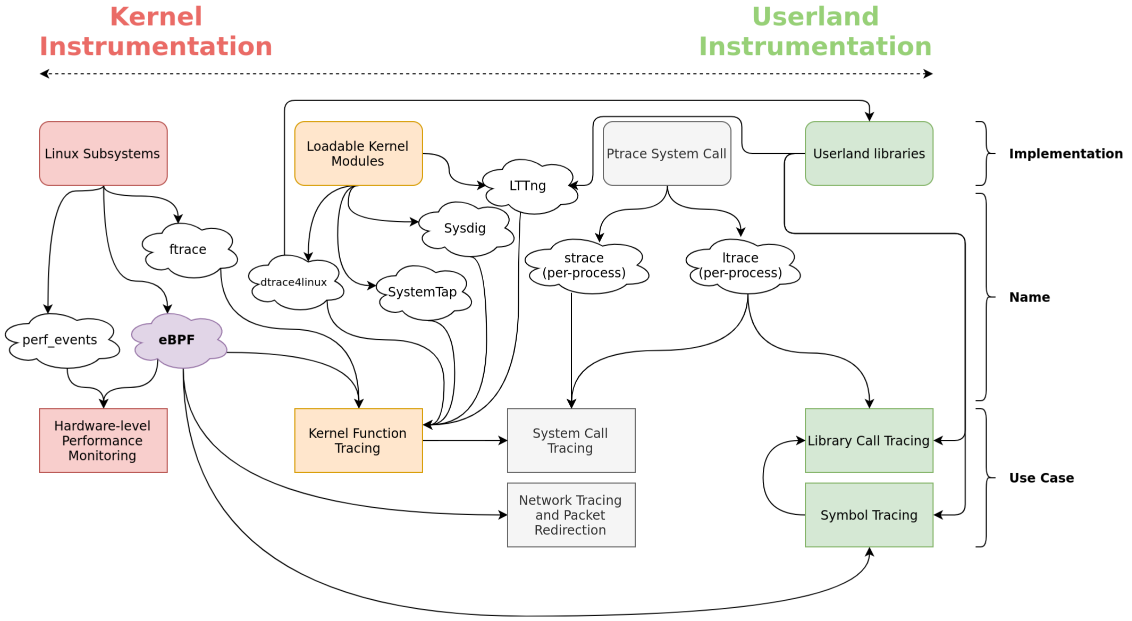


Figure 2.1: A high level overview of the broad categories of Linux instrumentation. This does not represent a complete picture of all available tools and interfaces, but instead presents many of the most popular ones. Note how eBPF covers every presented use case.

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. The original Dtrace [18] was designed in 2004 for Solaris and lives on to this day in several operating systems, including Solaris, FreeBSD, MacOS X [19], and Linux [13] (we will examine the dtrace4linux implementation with more scrutiny later in this section).

In general, the original Dtrace and the current version of eBPF share much of the same family and cover similar use cases [1], [18]. 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 three points:

(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); and
(3) eBPF is natively supported in Linux, while Dtrace ports are purely LKM-based.

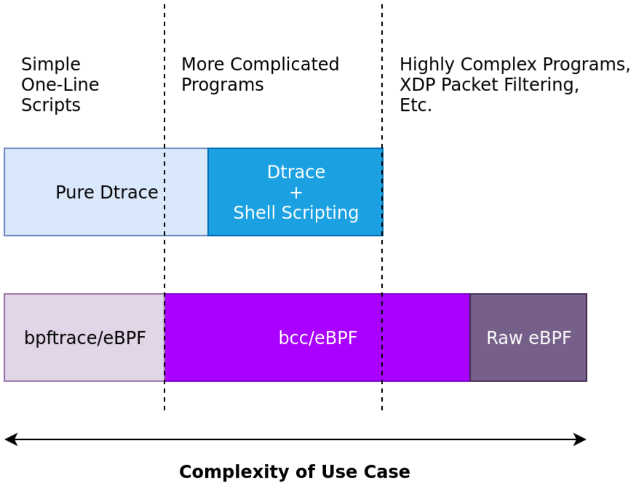


Figure 2.2: Comparing Dtrace and eBPF functionality with respect to API design. Note that eBPF covers more complex use cases and supports both low level and high level APIs. Dtrace needs to be used in tandem with shell scripting to support more complex use cases.

dtrace4linux [13] is a free and open source port of Sun's Dtrace [18] 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 [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 [20], [21] which has been designed to provide a high-level Dtrace-like tracing language for Linux using eBPF as a backend. Although bpftrace only provides a subset of Dtrace's functionality [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 [22] 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

The original incarnation of BPF was introduced to the world in a 1992 paper by McCanne and Jacobson [23] as a new system for capturing and filtering packets at the operating system level. In particular, Classic BPF implemented a filter virtual machine based entirely on registers as well as a tap mechanism to hook into packets. The *tap* copies network packets and delivers them to listening filter applications, while the *filter* determines whether a packet is accepted or not.

The primary motivating factors behind Classic BPF were the desire to establish an efficient technique for capturing and filtering packets. McCanne and Jacobson showed that their approach was significantly more efficient than other contemporary packet filtering mechanisms, i.e. NIT [24] and CSPF [25]; unsurprisingly, BPF is the only one of these three systems to have stood the test of time.

Classic BPF is still used to this day in a variety of network diagnostic services, perhaps most notably *tcpdump* [26] and *libcap*. It is also worth noting that eBPF implements a superset of Classic BPF; in other words, Classic BPF forms the basis for many modern eBPF programs in Linux.

2.3 eBPF: Linux Tracing Superpowers

In 2016, eBPF was described by Brendan Gregg [27] 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* [28], we can inspect, modify, redirect, and even drop packets entirely before they even reach the main kernel network stack. Figure 2.3 provides a high level overview of these use cases and the corresponding eBPF instrumentation required.

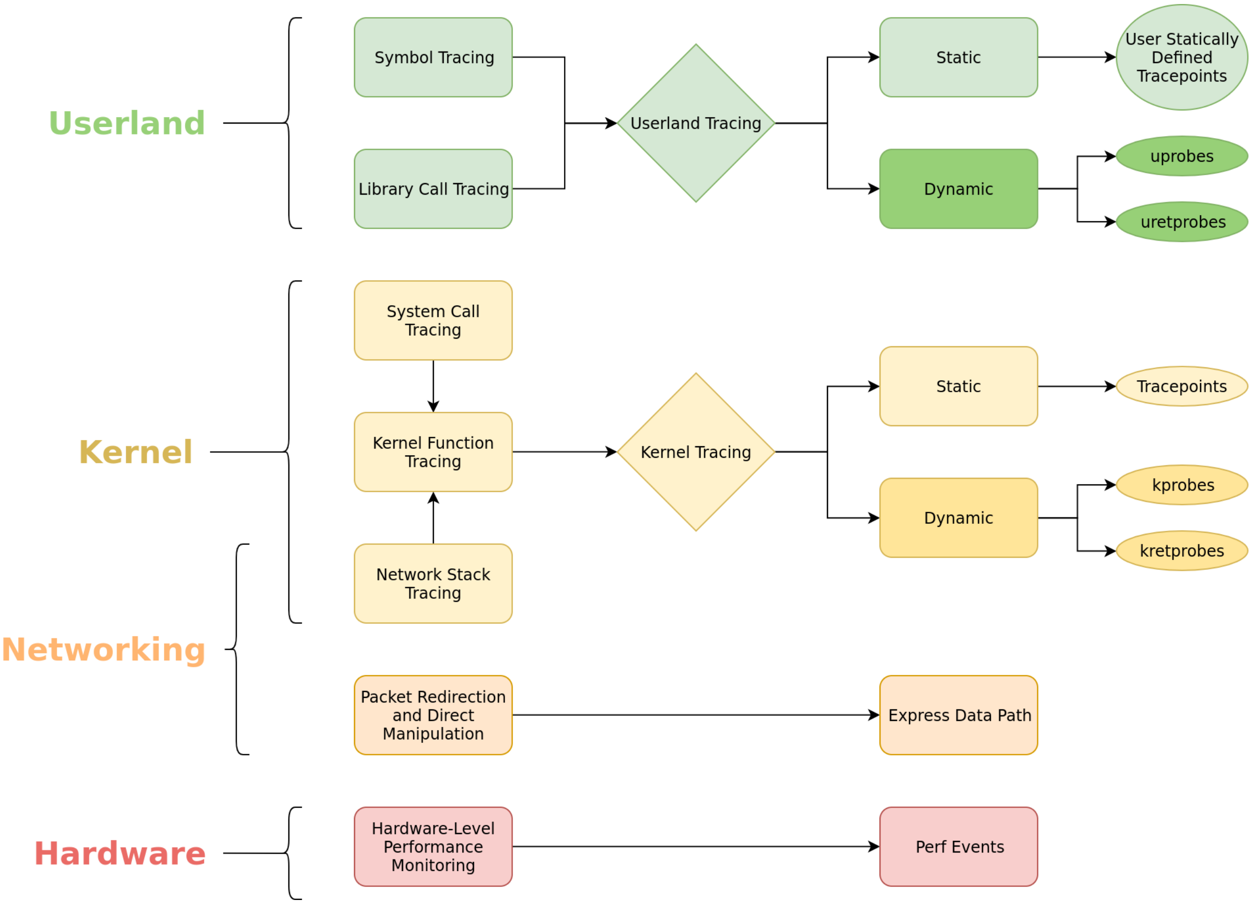


Figure 2.3: A high level overview of various eBPF use cases. Note the high level of flexibility that eBPF provides with respect to system tracing.

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

Safety is guaranteed with the help of an in-kernel verifier that checks all submitted bytecode before its insertion into the BPF virtual machine. While the verifier does limit what is possible (eBPF in its current state is *not* Turing complete), it is constantly being improved; for example, a recent patch [29] 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 Subsection 2.3.2.

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.

2.3.1 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. 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 2.4 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], [30], [31]. Thanks to this wide arsenal of maps, eBPF developers have a powerful set of both general-purpose and specialized data structures at their disposal; as we will see in coming sections, many of these maps are

[1] x86-64, SPARC, PowerPC, ARM, arm64, MIPS, and s390 [30]

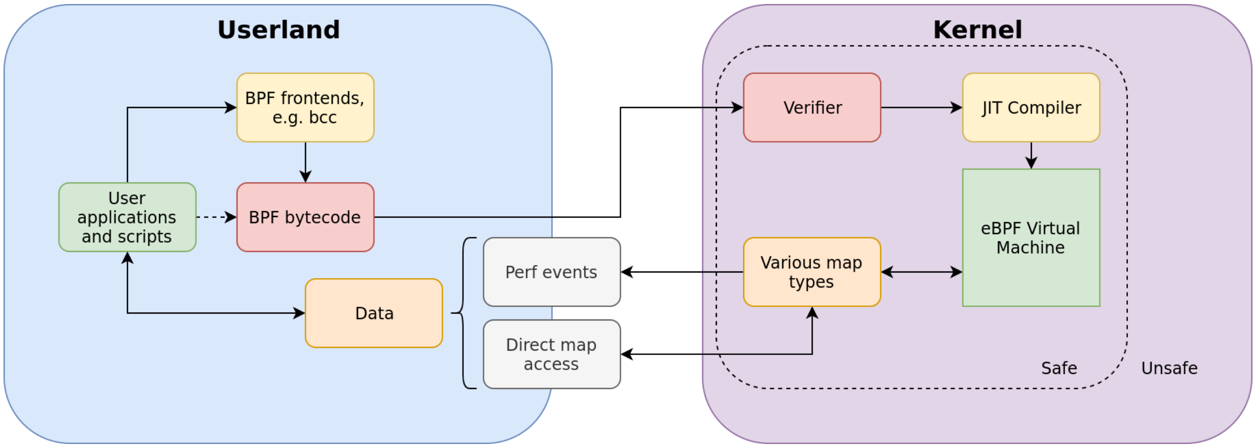


Figure 2.4: Basic topology of eBPF with respect to userland and the kernel. Note the bidirectional nature of dataflow between userspace and kernelspace using maps.

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 Subsection 2.3.2.

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

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

The verifier is responsible for eBPF's unprecedented safety, one of its most attractive qualities with respect to system tracing. While this verifier is quintessential to the safety of eBPF given its impressive scope and power, it is not without its drawbacks. In this section, 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 [32]. 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 [33] 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 2.5 depicts the relationship between potentially valid eBPF code and verifiably valid eBPF code.

The immediate exclusion of eBPF programs meeting certain criteria is the crux of eBPF's safety guarantees. Unfortunately, it also rather intuitively limits what we are actually able to do with eBPF programs. In particular, eBPF is not a Turing complete language; it prohibits 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.

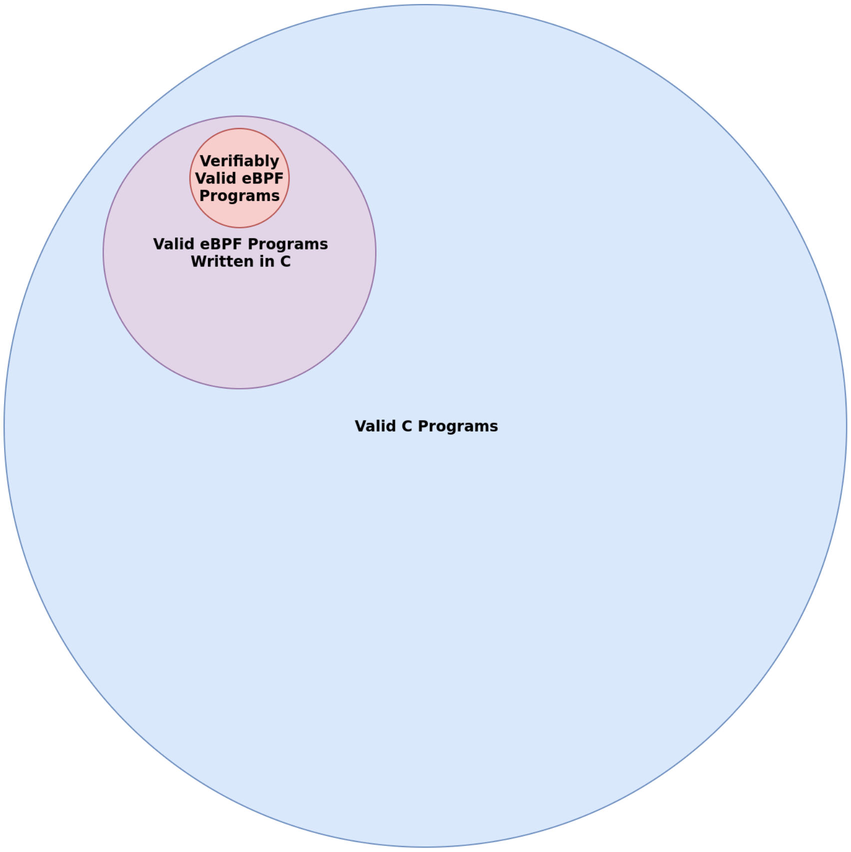


Figure 2.5: The set participation of valid C and eBPF programs. Valid eBPF programs written in C constitute a small subset of all valid C programs. Verifiably valid eBPF programs constitute an even smaller subset therein.

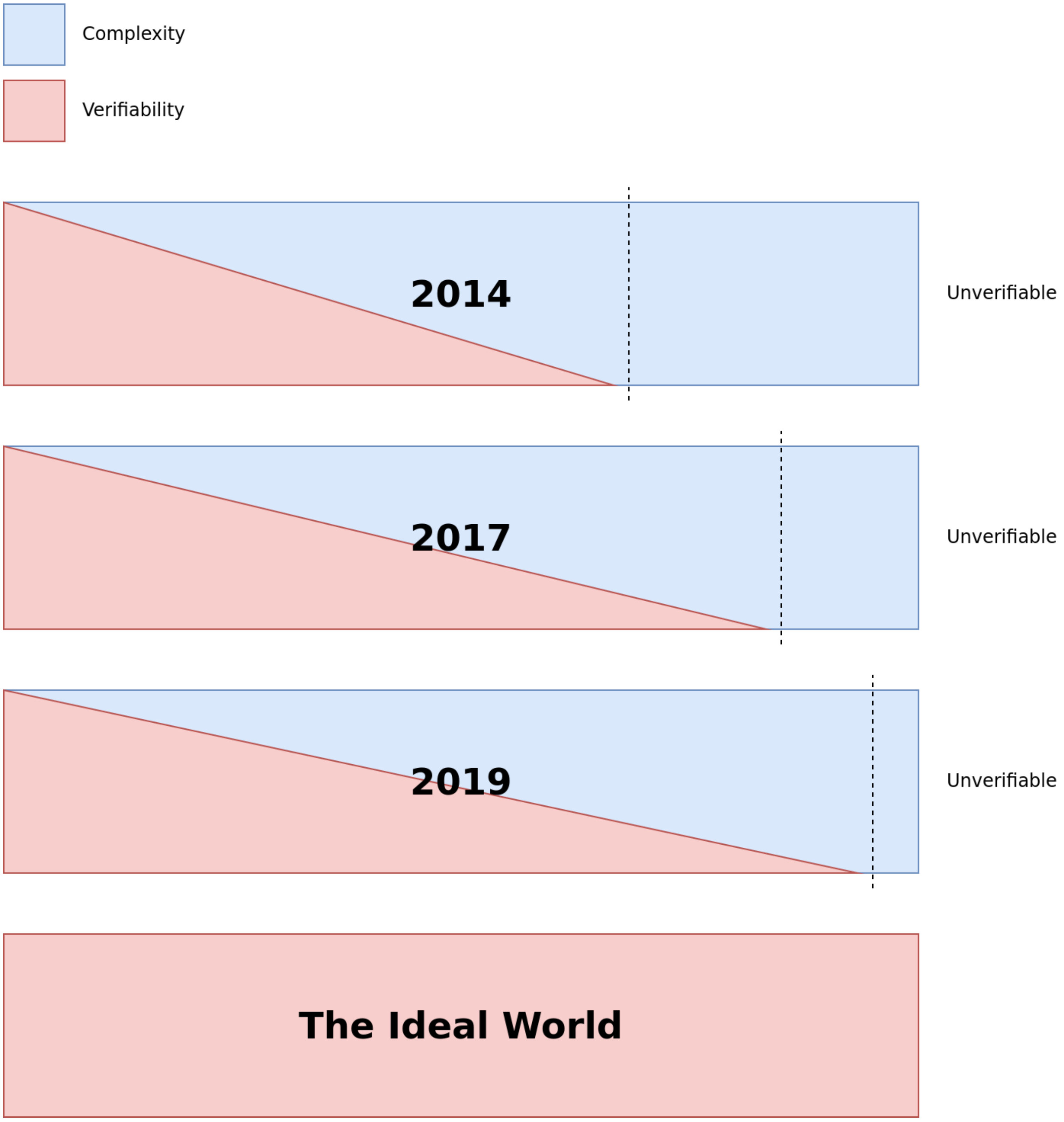


Figure 2.6: Complexity and verifiability of eBPF programs. Safety guarantees for eBPF programs rely on certain compromises. Ideally we would like to have a relationship as shown on the bottom; in practice, we have something that is getting closer over time, but is still far from the ideal.

exclude eBPF programs with unverifiable safety; but from a usability standpoint, particularly that of a new eBPF developer, the trade-off is not without its drawbacks.

Fortunately, the eBPF verifier is getting better over time (Figure 2.6). When we say *better*, what we mean is that it is able to prove the safety of increasingly complex programs. Perhaps the best example of this steady improvement is a recent kernel patch [29] that added support for bounded loops in eBPF programs. With this patch, the set of viable eBPF programs was *greatly* increased; in fact, ebpH in its current incarnation relies heavily on bounded loop support. Prior to bounded loops, eBPF programs relied on *unrolling* loops at compile time, a technique that was both slow and highly limited. This is just one example of the critical work that is being done to improve the verifier and thus improve eBPF as a whole.

2.4 System Calls

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

At the time of writing this paper, the Linux Kernel [34] 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 Subsection 2.1, perhaps most notably strace. We will discuss the behavioral implications of system calls further in Subsection 2.6.1.

2.5 Intrusion Detection

At a high level, intrusion detection systems (IDS) strive to monitor systems at a particular level and use observed data to make decisions about the legitimacy of these observations [35]. 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.

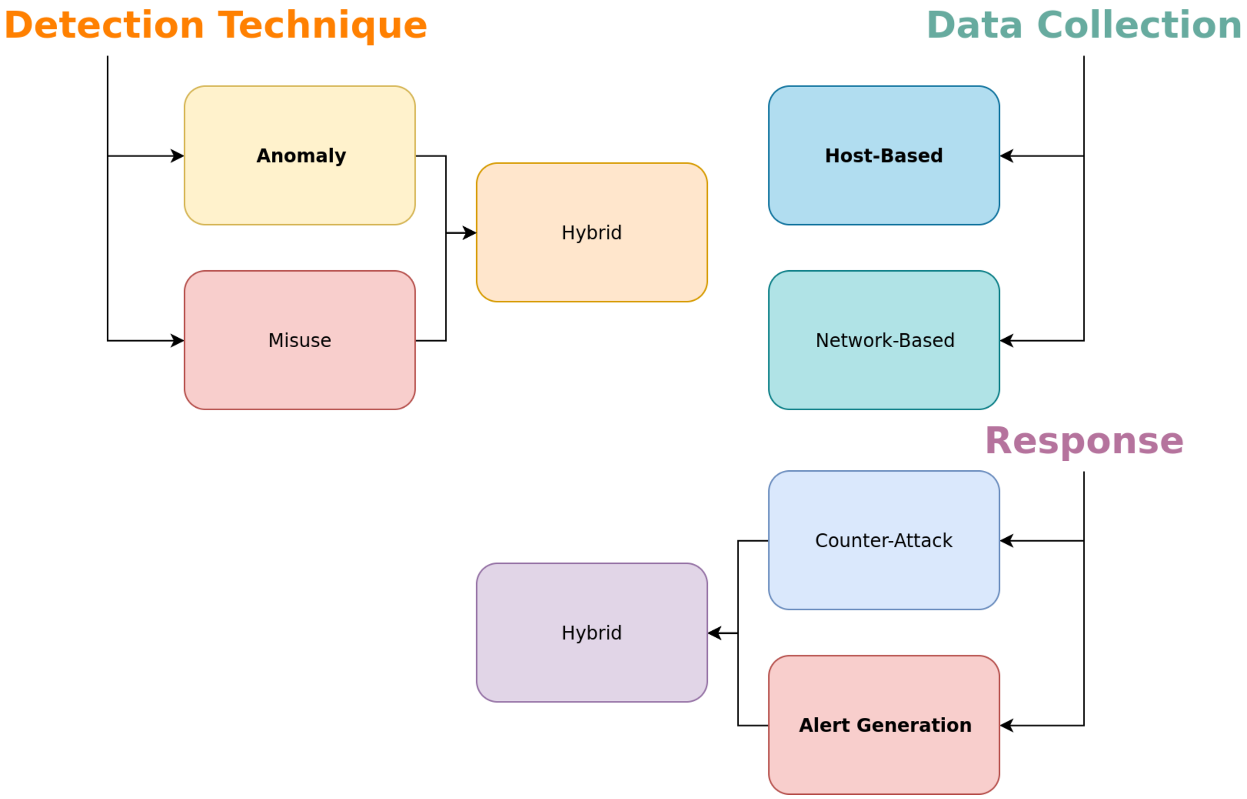


Figure 2.7: A broad overview of the basic categories of IDS. The current version of ebpH can be classified according to the categories labeled in **boldface**. Note that intrusion detection system classification can often be more nuanced than the basic overview presented here. However, this should present a good enough overview to understand IDSes 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 [35]. 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 IDSes examine network traffic and analyze it to detect attacks or anomalies. In contrast, host-based

IDSes analyze the state of the local system [3], [35].

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 [35]. 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 (Subsection 2.6) 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 Subsection 4.2).

2.5.1 Gathering Data for Intrusion Detection

TODO: come back here

2.6 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, 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 [3]. 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* [3], [36] 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 [36] 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 [36].

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 [3]. 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 [3].

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 [3] 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 (see Subsection 4.2).

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 Subsection 3.1) as well as an eBPF program (described in Subsection 3.2 and onwards). The dataflow between these components is depicted in Figure 3.1.

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

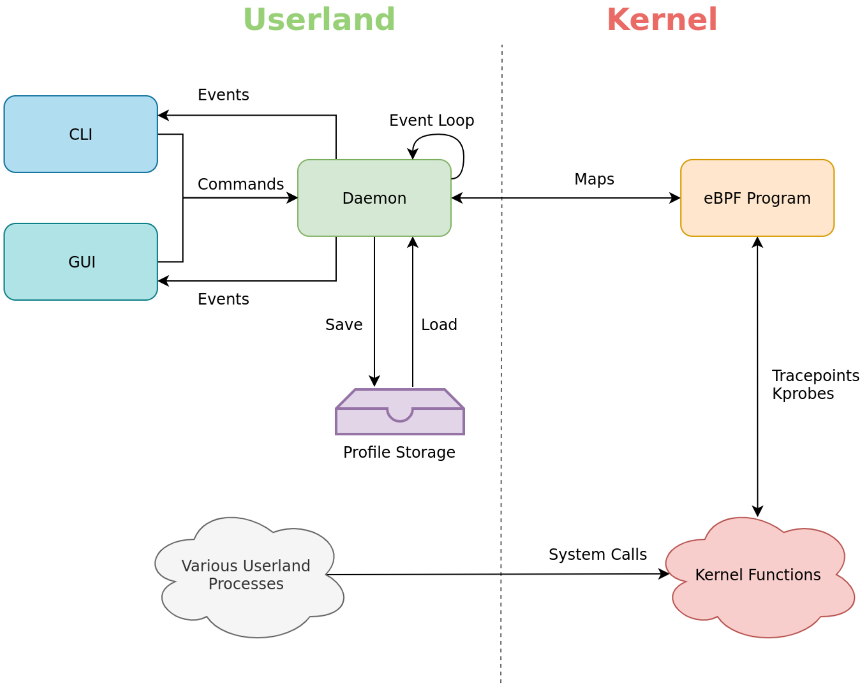


Figure 3.1: The dataflow between various components of ebpH.

3.1 Userspace Components

The userspace components of ebpH are comprised of two distinct programs. The **ebpH Daemon** (*ebpHD*) is responsible for initially compiling and submitting the eBPF program, as well as communication between userspace and the in-kernel eBPF program. As part of this communication, it loads existing profiles from the disk and saves new and modified profiles to the disk at regular intervals. Users can interact with the daemon either directly on the command line or by 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.

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.

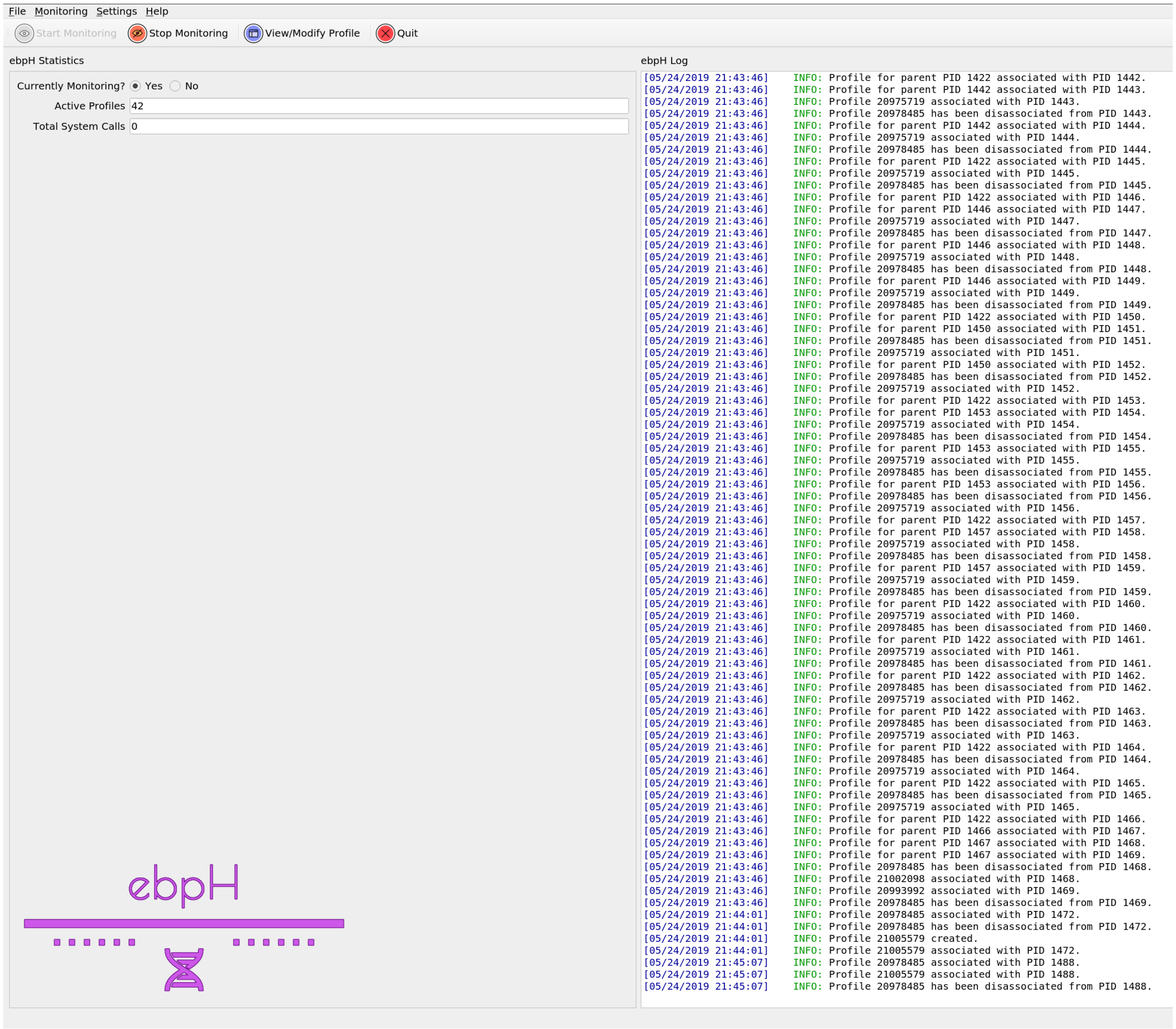
Table 3.1: Main event categories in ebpH.

Event	Description	Memory Overhead[3]
`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

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 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 processes and profiles. Figure 3.2 depicts an early version of the GUI.

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



Currently Monitoring? ● Yes ○ No

Active Profiles 42

Total System Calls 0

```
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1442.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1442 associated with PID 1443.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1443.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1443.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1442 associated with PID 1444.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1444.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1444.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1445.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1445.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1445.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1446.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1446 associated with PID 1447.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1447.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1447.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1446 associated with PID 1448.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1448.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1448.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1446 associated with PID 1449.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1449.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1449.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1450.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1450 associated with PID 1451.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1451.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1451.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1450 associated with PID 1452.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1452.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1452.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1453.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1453 associated with PID 1454.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1454.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1454.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1453 associated with PID 1455.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1455.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1455.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1453 associated with PID 1456.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1456.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1456.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1457.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1457 associated with PID 1458.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1458.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1458.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1457 associated with PID 1459.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1459.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1459.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1460.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1460.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1460.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1461.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1461.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1461.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1462.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1462.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1462.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1463.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1463.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1463.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1464.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1464.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1464.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1465.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1465.
[05/24/2019 21:43:46]    INFO: Profile 20975719 associated with PID 1465.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1422 associated with PID 1466.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1466 associated with PID 1467.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1467 associated with PID 1468.
[05/24/2019 21:43:46]    INFO: Profile for parent PID 1467 associated with PID 1469.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1468.
[05/24/2019 21:43:46]    INFO: Profile 21002098 associated with PID 1468.
[05/24/2019 21:43:46]    INFO: Profile 20993992 associated with PID 1469.
[05/24/2019 21:43:46]    INFO: Profile 20978485 has been disassociated from PID 1469.
[05/24/2019 21:44:01]    INFO: Profile 20978485 associated with PID 1472.
[05/24/2019 21:44:01]    INFO: Profile 20978485 has been disassociated from PID 1472.
[05/24/2019 21:44:01]    INFO: Profile 21005579 created.
[05/24/2019 21:44:01]    INFO: Profile 21005579 associated with PID 1472.
[05/24/2019 21:45:07]    INFO: Profile 20978485 associated with PID 1488.
[05/24/2019 21:45:07]    INFO: Profile 21005579 associated with PID 1488.
[05/24/2019 21:45:07]    INFO: Profile 20978485 has been disassociated from PID 1488.
```

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.

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 */
};
```

The profile itself is a C data structure that keeps track of information about the executable, as well as a sparse two-dimensional array of lookahead pairs [36] to keep track of system call patterns. Each entry in this array consists of an 8-bit integer, with the i^{th} bit corresponding to a previously observed distance i between the two calls. When 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 (Subsection 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.

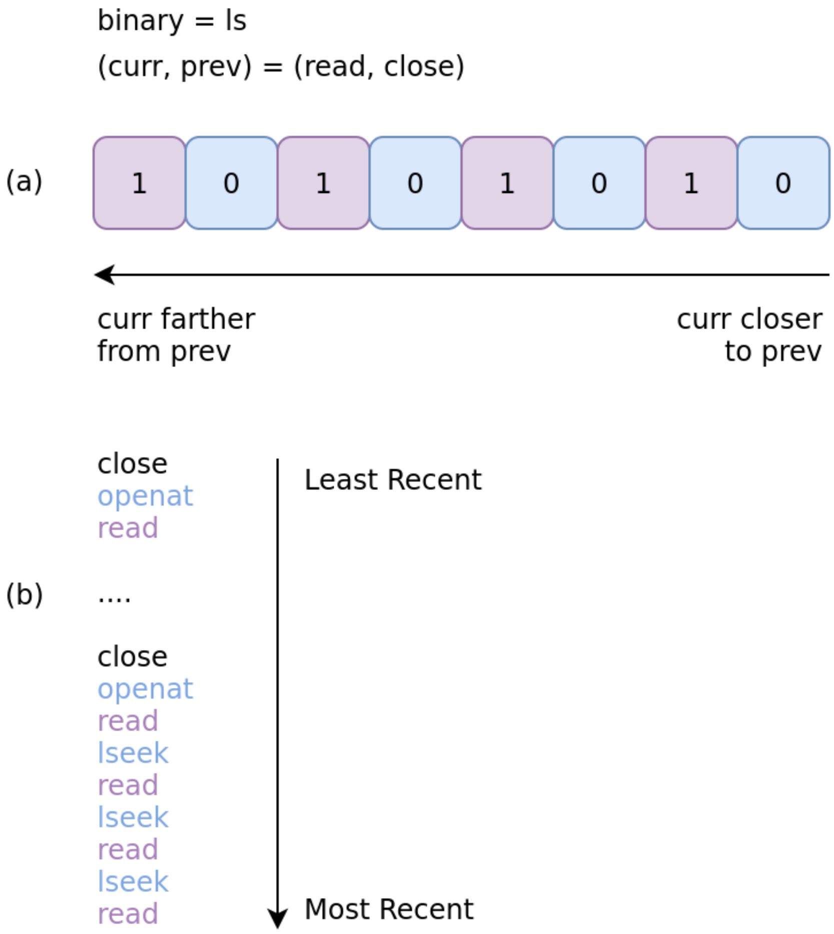


Figure 3.3: A sample (`read`, `close`) lookahead pair in the ebpH profile for `ls`. (a) shows the lookahead pair and (b) shows two relevant system call sequences, separated by several omitted calls. Note that the first three system calls in both the first and second sequence are responsible for the two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

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.

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

```c
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? */
};
```

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

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

Profile Creation and Association with `execve` 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 between two paths that resolve to a binary with the same name, for example `/usr/bin/ls` and `./ls`.

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; wipe the process' current sequence of system calls
`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	

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.

In addition to associating a process with the correct profile, we also wipe the process' current sequence of system calls, to ensure that there is no carryover between two unrelated profiles when constructing their lookahead pairs.

Profile Association and Sequence Duplication with `fork`, `vfork`, and `clone`. Another special consideration is with respect to `fork` and `clone` 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. 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. The parent's current system call sequence is also copied to the child to prevent forks from being used to break sequences.

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 Subsection 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.

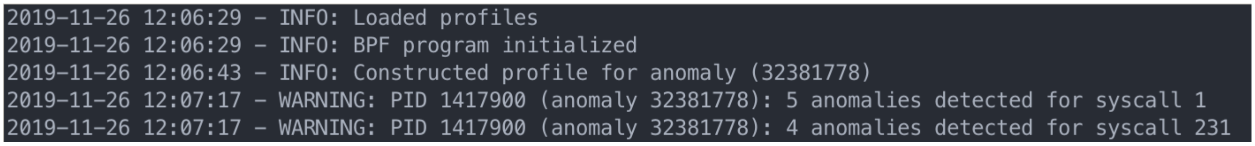
```c
/* anomaly.c */

#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 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 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.

3.5 Soothing the Verifier

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

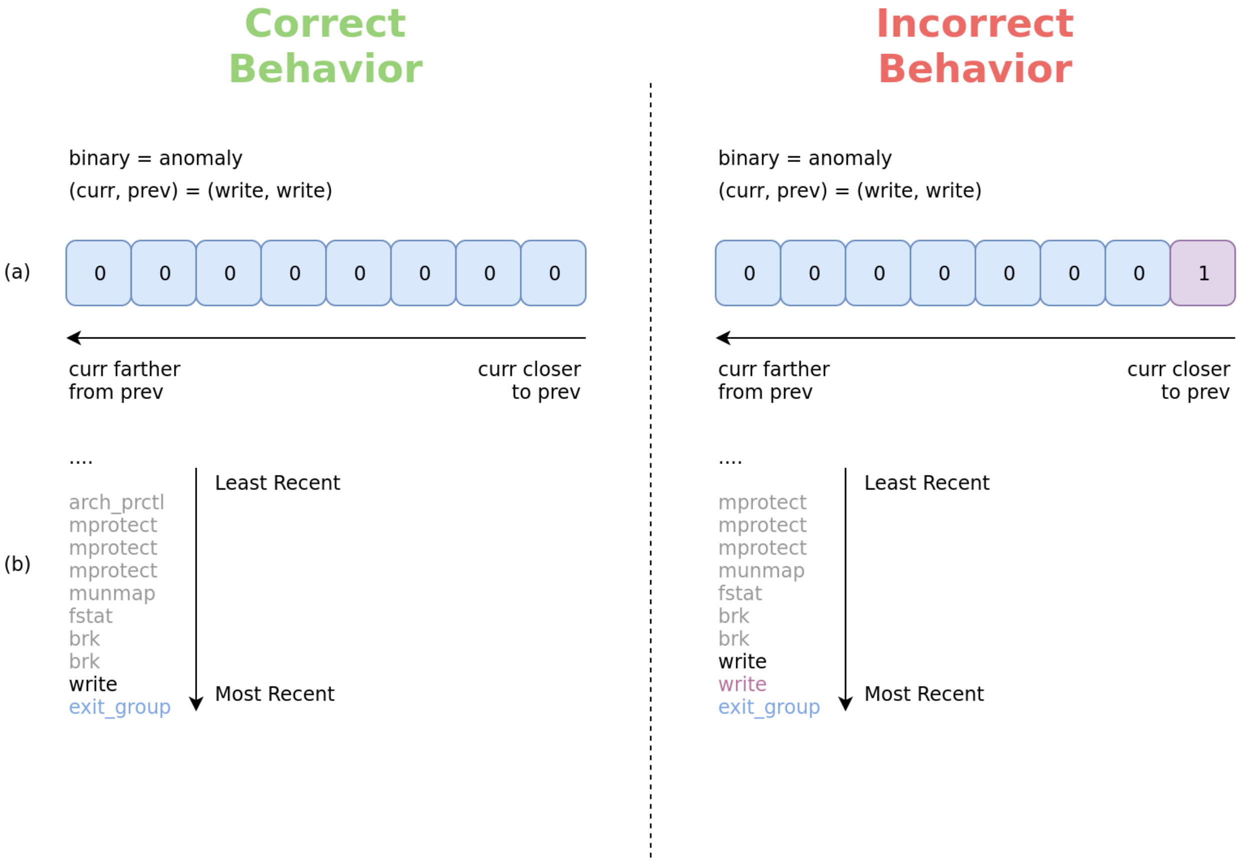


Figure 3.5: Two sample (`write`, `write`) lookahead pairs in the ebpH profile for `anomaly.c`. (a) shows the lookahead pair and (b) shows two relevant system call sequences. The left hand side depicts normal program behavior, while the right hand side depicts our artificially generated anomaly. There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

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 non-negligible 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 Subsection 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 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 [37]. 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 [37]. 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 for each possible state of the loop. For these reasons, development of eBPF programs that require bounded loops is still far from perfect, but we now at least have the tools with which to implement complex eBPF programs like ebpH.

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 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. Fortunately, well-constructed eBPF programs do not often suffer from false verifier 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. Table 4.1 summarizes the currently planned systems and any relevant details therein. Additional testing will be done on virtual machines to simulate multiple systems under artificially constructed workloads and attacks.

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

System	Description
arch	My personal desktop computer, which has been running the current ebpH prototype for one month
archlaptop	My personal laptop computer, which has been running the current ebpH prototype for one month
homeostasis	Dr. Somayaji's computer at Carleton University, which also serves the Wiki for his courses
CCSL Servers	The servers at the Carleton Computer Security Lab
Assorted Virtual Machines	Several virtual machines running a variety of test workloads

4.1.1 Gathering and Analyzing Profile Data

ebpH will be retrofitted with the ability to generate CSV datasets and plots that will summarize profile data. Through the examination of profile data (specifically lookahead pair patterns), we can get a clear picture of ebpH's understanding of system behavior. Results will be gathered for a wide variety of systems as depicted above in Table 4.1; this will yield the opportunity to test ebpH on production systems of various scale (i.e. *homeostasis* and the *CCSL Servers*) as well as personal computers for everyday use (i.e. *arch* and *archlaptop*).

Furthermore, the addition of several virtual machines for testing will provide the means to conduct reproducible experiments across various conditions, including measuring ebpH's response to a variety of simulated attacks. Snapshots will ensure controlled and consistent system state between runs, and will be particularly useful in controlling for initial ebpH profile state during each round of testing.

During normal testing, we are particularly interested in the rate of false positives and false negatives observed by ebpH. A false positive will be defined as any anomaly detected by ebpH that corresponds to ordinary system behavior, while a false negative will be defined as a failure to detect the presence of an attack. In order to test for both false positive and false negative rates, we will observe ebpH on ordinary systems as well as systems under attack and compare results.

As a general-purpose anomaly-based intrusion detection system, 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 [38] 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.

4.1.2 Gathering and Analyzing Performance Data

One of the primary advantages cited for using eBPF to build intrusion detection systems is lower overhead. In order to test the validity of this claim, we need reliable metrics to measure ebpH's memory and CPU overhead under a variety of workloads and systems. A recent patch to the Linux Kernel has added the ability to measure individual eBPF program performance [39]. Additionally, we can combine this approach through hardware performance measurement with eBPF perf events. This approach should provide the combined advantage of measuring the specific overhead associated with ebpH along with its impact on the overall memory and CPU usage of its environment.

Another important consideration with respect to overhead is ebpH's direct impact on the user; in particular, we want to avoid annoying the user by introducing noticeable delays into their workflow. Therefore, in addition to rigorous quantitative testing, ebpH's overhead will also be qualitatively tested for noticeable impact on system performance during everyday use.

4.2 Potential Improvements to ebpH

The system described in this proposal is a prototype, designed to implement the basic functionality of the pH intrusion detection system in eBPF, in order to ascertain whether such an implementation would be viable. While I believe I have achieved that goal, there is still plenty of room for future work on ebpH. Topics for future work include adding a mechanism for delaying system calls, using ebpH to increase overall system visibility, and the potential introduction of alternative behavioral metrics to provide a more comprehensive picture of system state and make better predictions about its validity. Furthermore, I plan to make extensive improvements to the ebpH GUI to complement the aforementioned augmentations to ebpH daemon functionality.

4.2.1 Delaying System Calls

The most obvious improvement to ebpH is the introduction of system call delays in a future iteration. This feature comprises a large part of the original pH's response strategy and would be a vital part of a full reimplementation. As previously discussed, this is not necessarily an easy thing to accomplish due to the eBPF verifier's restrictions on program safety. From the perspective of what eBPF is trying to accomplish, this makes sense. In Somayaji's dissertation [3], he discussed the potential for pH itself to cause denial of service on a system, due to intentional provocation from an attacker or simply an edge case in program behavior. eBPF is designed with safety in mind; allowing eBPF programs to cause denial of service in this way would be the antithesis of what eBPF is trying to accomplish. Therefore, another solution is needed.

A kernel-based implementation for process delays would certainly work, but would be far from ideal – this sacrifices a lot of the advantages that come with an eBPF implementation of pH in the first place, namely easy portability between Linux systems and guaranteed safety. Such an implementation would either be in the form of a kernel patch or a loadable kernel module; both of these solutions suffer from safety issues as we have discussed at length in Subsection 2.1. Additionally, a kernel patch in particular limits the portability of ebpH, which currently runs on any vanilla Linux Kernel above version 5.3.

We can also consider the possibility of busy waiting within the eBPF tracepoints themselves, although this also carries a few obvious drawbacks. Firstly, busy waiting means that the process that is being slowed down will continue to occupy CPU time instead of yielding it to another process by ceding time to the scheduler. Another obvious drawback is with respect to the verifier itself; verifier support for the bounded loops required for busy waiting is conditional on several factors. This may result in the rejection of busy waiting due to perceived safety violations.

A third possibility is issuing delays from userspace via the `SIGSTOP` and `SIGCONT` signals; the daemon would simply issue these signals to offending processes and space them proportionally with respect to recent anomalies produced. While this solution *could* work, it suffers from a few obvious flaws. Firstly, there is no guarantee that we can prevent another signal from waking up the process early; in fact, an attacker with the ability to send arbitrary signals has already completely circumvented this type of response. Additionally, there is no guarantee that a process will receive this signal in time to stop the offending system call(s). By the time the process receives the signal, it may already be too late to stop the attack.

None of these solutions seem ideal; it is likely that a presently unknown fourth alternative will present the best approach. Perhaps there may be an entirely eBPF-based solution on the horizon pending updates to the verifier – time will tell. For now, it may be worthwhile exploring what options are available at the present to determine if any are suitable for use in practice.

4.2.2 Measuring Other System Behavior

In his dissertation [3], Somayaji discusses the potential of having multiple homeostatic systems at work on a given machine. This approach would more closely resemble the concept of homeostasis we know in biology, wherein multiple subsystems work together to add stability to overall system state. pH was a great starting point, and pending the introduction of system call delays as discussed in the above section, ebpH will follow in its footsteps in that regard. However, much of the true power of eBPF comes from its ability to monitor *so much* system state at once; there's no reason ebpH has to stop at system call tracing.

By using eBPF to monitor multiple facets of system state, we can get a clearer picture of normal process behavior, which could in turn yield more accurate anomaly detection results. Perhaps these metrics could include memory allocations, number of incoming network connections, socket I/O, file I/O, CPU time per process, or any number of such metrics. eBPF can measure all of that and more; and it can do so reliably, efficiently, and with guaranteed safety.

4.2.3 Overall System Visibility

As an intrusion detection system, ebpH's role is well-defined: monitor the system, detect misbehaving processes, and report them to the user. However, there is one glaring problem with this approach, particularly as we venture into the territory of automated responses via system call delays: users do not necessarily *want* a system that chooses not to perform a requested action; they also do not necessarily *want* a system that harasses them with warnings about program behavior that they either don't care about or don't necessarily understand.

One potential solution to this problem is providing other benefits to the user through ebpH *in addition to* intrusion detection and response. For example, future versions of ebpH could include a performance analysis component, a debugger component, or any number of other metrics for increased system visibility. After all, one of the primary use cases for system introspection is precisely that: allowing a user to observe their system. By adding this extra functionality, we can provide complimentary benefits to the user that may incentivize them to run ebpH in the first place.

It should also not be overlooked that, in many cases, increased system state visibility can provide implicit security benefits to the experienced user. For example, an experienced system administrator could use a future version of ebpH to find vulnerabilities in their system before an attack even occurs.

4.3 Improvements to the ebpH GUI

The GUI has potentially the most room for improvement of the entire ebpH system. Many of these improvements will be in tandem to those discussed in previous sections; in particular, system visibility enhancements to ebpH will almost certainly manifest themselves through the GUI wherein we will be able to visually present information to the user as they request it.

Future versions of the GUI will also have more options for inspecting and making changes to system profiles. Somayaji describes three important operations in his dissertation [**soma2**], *tolerize*, *sensitize*, and *normalize*, which have not yet been implemented in ebpH. These options will be added to the GUI when they are implemented to allow users to have fine grained control over how ebpH treats the profiles on their system. Additionally, users should be able to inspect individual profiles and look at their data in a format that is easy to understand; potentially this could be achieved via plotting profile data or perhaps presenting the top n system call lookahead pairs by process.

Ultimately, the GUI is perhaps among the most important components of ebpH, particularly given the usability requirements we have discussed in previous sections. As such, it is one of the most important factors controlling the potential future adoption of ebpH, and is therefore important to get exactly right.

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. 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.

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

[20] 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.

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

[22] 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/.

[23] 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.

[24] *Nit(4p) sunos 4.1.1 reference manual*, Sun Microsystems Inc., Sep. 1990.

[25] J. Mogul, R. Rashid, and M. Accetta, “The packer filter: An efficient mechanism for user-level network code,” in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, ser. SOSP ’87, Austin, Texas, USA: ACM, 1987, pp. 39–51, isbn: 0-89791-242-X. doi: 10.1145/41457.37505. [Online]. Available: http://doi.acm.org/10.1145/41457.37505.

[26] *Tcpdump/libpcap public repository*, Sep. 2010. [Online]. Available: https://www.tcpdump.org/.

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

[28] 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.

[29] 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.

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

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

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

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

[38] 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. [Online]. Available: http://doi.acm.org/10.1145/586110.586145.

[39] A. Starovoitov, *[v2,bpf-next,1/4] bpf: Enable program stats 1047415 diff mbox series*, Feb. 2019. [Online]. Available: https://patchwork.ozlabs.org/patch/1047415/.

Appendix A eBPF Design Patterns

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

```c
/* 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;
}
```