Efficient Audit Data Collection for Linux

A Thesis presented

by

Rohit Aich

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

August 2021

Stony Brook University

The Graduate School

Rohit Aich

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis

R. Sekar
Professor, Computer Science

Scott Stoller
Professor, Computer Science

Amir Rahmati
Assistant Professor, Computer Science

This thesis is accepted by the Graduate School

Eric Wertheimer

Dean of the Graduate School

Abstract of the Thesis

Efficient Audit Data Collection for Linux

by

Rohit Aich

Master of Science

in

Computer Science

Stony Brook University

2021

Forensic analysis is a well-established practice for detecting long-term and stealthy attacks such as APTs. System audit logs provide crucial information required for such detailed analyses. Contemporary audit logging techniques available for Linux-based systems require kernel modifications, risky and impractical for the real world. We present an audit logging system based on eBPF, a framework that can run sandboxed programs safely inside the kernel without modifying the kernel source code. Our system can record audit data at the granularity of system calls and is safe to use. Experiments show that the system adds a run-time overhead of just 1%.

Dedication

I DEDICATE THIS THESIS TO MY FATHER, SIDDHARTHA AICH. HE TAUGHT ME TO BE A GOOD HUMAN BEING AND HELPED ME IN ALL THINGS GREAT AND SMALL. THANK YOU "*baba,*" I LOVE YOU VERY MUCH.

I DEDICATE THIS THESIS TO ANINDITA, MY BELOVED WIFE. SHE HAS BEEN A CONSTANT SOURCE OF SUPPORT AND ENCOURAGEMENT DURING THE CHALLENGES OF GRADUATE SCHOOL AND LIFE.

MY GRANDFATHER, AJIT KUMAR AICH, HAD TO QUIT COLLEGE AND WORK THREE JOBS TO SUPPORT OUR FAMILY. HE TAUGHT ME THE VALUE OF HARD WORK. HE WOULD HAVE BEEN VERY HAPPY TODAY. THIS THESIS IS DEDICATED TO HIS FOND MEMORIES.

Contents

Acknowledgements

I am extremely grateful and indebted to Professor R. Sekar for his invaluable advice, patience, and continuous support during my study. His immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life. I would also like to thank my professors in the Computer Science Department and my colleagues in the Security Research Group. It is their kind help and support that have made my study and life at Stony Brook University a wonderful time. Finally, I gratefully acknowledge the financial support from our sponsors NSF (through the grant CNS-1918667) and ONR (through the grant N00014-17-1-2891). This work would not have been possible without their support.

1 Introduction

Despite the use of defense mechanisms such as firewalls, encryption and anti-malware systems, large enterprises continue to experience stealthy and long-term cyber-attacks, commonly known as Advanced Persistent Threats (APTs). Recent APTs such as "Crouching Yeti" [4], "Darkhotel" [10], "LightSpy" [31] and the "SUNBURST" [9] are rapidly infecting thousands of systems worldwide. "Crouching Yeti" and "Darkhotel" used three methods to infect the victims: Spearphishing using PDF documents embedded with flash exploits, trojans inside legitimate software installers, and waterhole attacks using a variety of exploits. The iOS-based "LightSpy" attack is a watering hole attack discovered in January 2020. It lures victims from a news forum to another, disguised as a legitimate one. The attackers then inject an iframe that loads the exploit. LightSpy attack takes control of the machine and exfiltrates crucial and sensitive data, such as hardware information, contacts, browser history, etc. Another very recent supply-chain attack was recently discovered by FireEye that trojanizes software updates from SolarWinds Orion business software to distribute a malware called "SUNBURST".

The attack techniques described above are stealthy and long-term in nature. In many cases, the malware introduced would remain dormant for a long time before starting operation. By employing advanced social engineering and exploit techniques, these APTs can circumvent deployed security mechanisms such as firewalls, antivirus software, and so on. Often, the APTs stay hidden and operate for months or years. Ultimately, when security personnel detect suspicious activity, they need to determine the root cause, so that they can eliminate all infected files and malware from the system. Forensic analysis is the method used to identify the root cause as well as the after-effects of such malware. For a successful forensic analysis, we need a detailed record of system activity, typically obtained from various system and application logs.

Many types of audit logs are available. We can log activities of web servers, specific software applications, etc. These logs could be useful for the specific purposes they're originally designed for. For example, logs for a banking website will show details of successful and failed transactions, etc. These are high-level logs and are easy to read and understand from the application's perspective. However, when we are dealing with malicious software, their operations on the system may go unrecorded by these application-specific logs. Let us consider a scenario where a malware gains access to a system via a compromised web server. The program remained dormant for several months and then started corrupting applications inside the victim machine. Now, if we just maintain an audit log for the webserver, the corrupted files or applications would not be reflected in the logs.

To detect such attacks, and to find the malware's footprint across a system, we need to capture system-wide activities. We need complete mediation, i.e., track the system activities at such a granular level that every possible action of malicious software could be captured. A major share of the servers and systems of commercial enterprises run Linux or UNIX like operating systems [30, 34]. In this thesis, we'll focus on Linux-based systems, and talk about audit logging techniques available for Linux.

There's an existing audit logging framework present for Linux systems, known as the Linux Audit Daemon. However, the Linux Audit Daemon suffers from many known problems, such as run-time performance and space overhead. That is why many current pieces of research have focused on developing alternative mechanisms that try to provide full system audit data with improved performance. Different audit loggers have different goals. There are audit logging systems that collect audit logs to detect attacks and find malicious processes running in the background, while other types of systems are interested to find the complete ownership history of kernel objects. Either way, it is imperative to achieve complete mediation, i.e., no system activities should escape the logger.

The main issues with available audit logging systems are *deployability* and *performance*.

To our knowledge, all of the available techniques to log audit data require loading one or more kernel modules in the system. In large organizations comprising of hundreds of systems, it is not possible to rebuild kernels with new modules. Such changes are not only infeasible but also are very difficult to maintain (for instance, a scheduled upgrade in the Linux kernel may require changes to the audit logger). A few solutions require users to use a modified kernel [29, 27, 24], and another few require the application developers to use certain logger libraries while coding [1], significantly increasing the user efforts. These solutions, hence, are impractical and not feasible to implement in large enterprises.

On the other hand, widespread auditing systems such as Linux Audit Daemon are often unnecessarily verbose, and it has a high run-time overhead. Our experiments show that with Linux Audit Daemon, the run-time of resource-heavy operations can increase five-fold. Moreover, most existing auditing systems generate a lot of data, usually in the order of several gigabytes per day. Storing and maintaining this much data is always a challenge, and doing successful forensic analyses through a large data to uncover malicious activity is time consuming. With the increasing number of large organizations falling victim to APTs worldwide, the need for a smart auditing technique offering better performance (both run-time and storage-wise) is also increasing.

Our goal is to build an audit data collection system that is suitable to deploy in real systems and offers better performance. Towards that, we present a lightweight audit logging system that leverages a new performance and security monitoring framework for Linux called eBPF. Our system does not introduce any new module to the kernel, and hence, does not require a kernel build or any type of kernel modifications. The system hooks into system calls at predefined tracepoints and traces the arguments and return values of system calls. We discuss the eBPF framework and our system implementation in subsequent sections.

Our experiments show that our eBPF-based system has a minimal run-time overhead: about 1%–4% and the output log generated by our audit log reducer is much smaller in size compared to Linux Auditd logs. On top of this, this eBPF-based system is thread-safe, and suited for multiprocessing environments.

The rest of the report is structured as follows: Section 2 describes the eBPF framework. Section 3 then presents our solution and Section 4 presents a performance analysis. Section 5 discusses related works in the area. Section **??** concludes the report.

2 Overview of eBPF Framework

The Linux kernel is the best place to implement tracing, networking, and security functionalities. However, installing new instrumentation code inside the Linux kernel always comes with a substantial risk. Kernel coding is difficult and involves much more safety checking than user-space coding. Also, the kernel space is an unforgiving environment, and loading a poorly designed module in the kernel could have disastrous results.

eBPF and related technologies have helped in developing profiling, tracing, and performance monitoring tools that can give us information at run-time with low performance impact. In Linux, eBPF (Extended Berkeley Packet Filters) [6] is a virtual machine-based framework that can run sandboxed programs inside the kernel *without requiring kernel source-code modifications or loading new kernel modules.*

eBPF traces its origin back to the Berkeley Packet Filter [22] aka BPF, which became a part of the Linux kernel in 1993. The Berkeley Packet Filter was originally designed for profiling and filtering network packets before they were processed by the kernel or copied into userspace. This filtering functionality was implemented as functions in a register-based virtual machine inside the kernel, with two 32 bit registers and a small RISC-based ISA.

While the idea of executing user-supplied programs safely inside the kernel proved to be very useful, the capacities of the original BPF were only limited to network packet filtering and profiling. It was not capable of doing other things, like tracing kernel functions and performance monitoring. This prompted the creation of eBPF, which significantly extended the functionalities of BPF. Moreover, eBPF uses just-in-time (JIT) compiled, which ensured faster execution of eBPF functions inside the kernel.

The eBPF framework and the projects built around it are still in an early stage. It is rapidly evolving, and newer functionalities and support get added to the framework regularly. Some of these features are available only for the latest Linux kernel versions (5.8 and above) and require more documentation. The tracing abilities of the framework that we have used in our projects are available for kernel version(s) above 5.4 [11, 6, 5, 14, 23, 3].

2.1 eBPF Architecture

Figure 2.1 presents a diagram to show the eBPF architecture. Below, we discuss the salient features of eBPF programs, and how they work.

- *Hooks*: eBPF programs are event-driven. When the kernel or an application passes a certain execution point (known as a hook), the eBPF program will be triggered. There are pre-defined hooks already defined in the kernel, for system calls, function entry/exit, kernel tracepoints, network events, and several other points. In case we want to trace a kernel/user application function that has got no pre-defined hooks, we can use a 'kprobe' or 'uprobe' features of eBPF to monitor them.
- *eBPF Programs*: Linux expects eBPF programs to be in byte-code. While it is possible to write the byte-code directly, programs are typically written indirectly, via projects

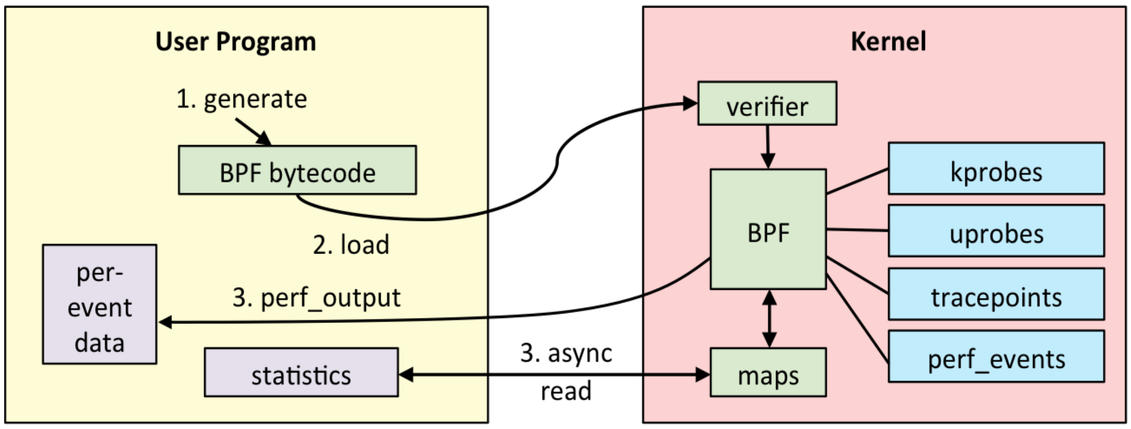

Figure 2.1: eBPF Architecture and Workflow (taken from [11])

such as BCC(explained in further detail in the later sub-section) or bpftrace. Here, programs written in BPF-C (a restricted version of C) are compiled with LLVM to generate eBPF byte-code.

- *eBPF Verifier & JIT Compiler*: Once the hooking event has been identified, the eBPF program can be loaded in the kernel using the *bpf(2)* system call. After this, there remain two more steps for the program to run. Since the programs are allowed to directly run inside the kernel, it is of extreme importance that the code does not corrupt memory or crash the kernel. The eBPF verifier takes care of this. It validates that the eBPF program meets several safety conditions (no loops or unauthorized memory accesses etc.). We have discussed the safety features of eBPF in detail in the next subsection.

  Once the eBPF program has been verified, the eBPF byte-code is translated to a machine-specific instruction set. This makes the eBPF programs run as efficiently as code loaded as a kernel module or natively compiled programs.

- *eBPF Maps & Helper functions*: One interesting feature of eBPF is the ability to share data between different eBPF programs, as well as with user-space applications with a system call. The programs achieve these by using a set of data structures, collectively known as “eBPF maps” [6, 3]. There can be various kinds of maps depending on their usages, such as hashmaps, arrays, stack traces, ring-buffers, etc.

  Also, eBPF programs cannot call arbitrary kernel utility functions. This ensures the safety and also addresses compatibility issues (for instance, if eBPF programs were allowed to directly call into kernel functions, they would be bound with specific kernel versions, leading to compatibility issues). Instead, eBPF programs have a set of well-documented helper functions. For example, helper functions can generate random numbers, provide accurate timestamps for events, provide pids (process ids) and other details of traced processes, etc.

2.2 Safety Features in eBPF

To ensure that the kernel remains safe, the eBPF framework verifies programs against certain safety conditions before they are executed in the eBPF virtual machine. We have described these below:

- Only privileged processes are allowed to run eBPF programs. So, users who have no root access will not be able to start an eBPF program.
- The verifier checks that the program does not cause any unauthorized memory access so that it does not harm or crash the kernel at any point.
- The eBPF bytecode does not allow any loops. This ensures that no programs could run indefinitely inside the kernel.
- The eBPF verifier also ensures that all branches of the code are essentially complete and reach the end of the program.

2.3 Programming in eBPF

As explained in the earlier section, it is not very convenient to write eBPF byte-code directly. A better and more practiced way is to use toolkits such as "*bpftrace*" or "*BPF Compiler Collection*". These toolkits provide developer-friendly environments and ways for writing efficient eBPF programs. In our experiments, we have used both of these toolkits and evaluated their performances.

eBPF sends the tracing outputs to user-space in three main ways. For sending small one-liner performance profiling outputs, eBPF has a feature that prints into the common trace_pipe of the system (/sys/kernel/debug/tracing/trace_pipe). For more structured and complex outputs, eBPF uses a data structure called PERF_OUTPUT. This is a high-speed ring-buffer shared between the user and kernel spaces. The PERF_OUTPUT takes the data and the size as an input and sends it from kernel to userspace. There is another way, where we can asynchronously read data from shared eBPF maps. These maps can have many utilities, such as displaying frequency counts, performance histograms, summary statistics, etc. In our experiments, we have used the first two techniques, i.e., the trace_pipe-based simple printing technique, and the ring buffer based data sharing technique.

2.3.1 bpftrace

bpftrace is a small toolkit developed over the eBPF framework. bpftrace uses a simple script-based coding language, much like shell scripts. These scripts are typically of one or two lines and can trace the arguments and return values of kernel programs. bpftrace is typically used for very specific program monitoring or profiling purposes. For example, if we want to list the number of successful read(2) system calls over some time, we can write a one-liner bpftrace code for that.

bprtrace uses a built-in printf() function that prints to the system's common trace_pipe.

Hence, we would be able to directly see the output data on the console.

2.3.2 BPF Compiler Collection

The BPF Compiler Collection, more commonly known as BCC, is a toolkit for creating efficient kernel profiling and manipulation programs. BCC makes eBPF programs easier to write. BCC provides kernel instrumentation in C (and includes a C wrapper around LLVM), and front-end programs in Python or Lua.

A BCC program comprises a Python (or Lua) front-end, which contains two parts: A string that comprises the entire kernel-space program written in BPF C, and a method that polls from the PERF_OUTPUT buffer. BCC provides seamless integration between these two parts. As a result, the Python front-end gets the accurate data-type information of the traced function arguments and return values. This makes it further easy for the Python front end to encapsulate the incoming data into the correct data type for further processing.

BCC is suited for many tasks, including performance analysis and network traffic filtering. BCC provides a basic troubleshooting mechanism as well. For example, if there's an error while loading the eBPF program (for example, the eBPF verifier throws an error), BCC provides messages aka *'hints'* to tell the programmer why the loading could have failed.

BCC also provides few workarounds for some of the limitations of the eBPF bytecode. In the next section, we have discussed these in detail.

2.4 Limitations

Despite being a very useful technology, eBPF has its own set of limitations. A few of the major limitations are as follows:

- No loops are allowed in eBPF bytecode. In case we are using BCC, there is a workaround. In BCC We can only use bounded loops in the BPF-C side, with the pragma unroll directive of LLVM. This directive helps to unroll the C-code and generates an eBPF bytecode that does not have any backward branches.
- The total eBPF program stack size has to be under 512 bytes.
- An eBPF program can contain no more than 4096 instructions. However, eBPF provides a functionality known as *tail calls*. These tail calls can be made from one eBPF program to execute another eBPF program and replace the execution context.
- The current toolkits (such as BCC or bpftrace) built over the eBPF framework does not allow the usage of global variables yet [5]. This is not a limitation of the eBPF virtual machine [14], as the virtual machine does not prohibit the usage of static memory. As a workaround, the BCC toolkit provides a specific eBPF map called BPF_PER_-CPU_ARRAY, which can be used to achieve the same purpose.
- While sending the traced data to user-space, BCC uses a high-speed ring-buffer called 'BPF_PERF_OUTPUT'. This buffer is not lossless, and if the user-space application is

not consuming the data fast enough, there could be a loss of data. In case of such a data loss, BCC provides functionality to show exactly how many bytes were lost.

3 Our Solution

As explained in Section 1, performance and deployability are the two most desirable qualities of any audit logging system, and our goal is to build a system that displays both of these qualities. From an implementation perspective though, there are two parts of a good audit logging system: A) *Recording the system activities* and B) *Presenting the data concisely.* To ensure that a system can be deployed without any kernel modifications, and also offers superior run-time performance, we need to choose a suitable technique to record system data. Also, for an audit log to be useful for forensic analyses, we need to arrange the audit records concisely, so that the most relevant and important information could be identified easily.

Since our target is to uncover APTs, we need to track all kernel activities at the granularity of system calls. To do so, we need to instrument system calls to record arguments and return values as necessary. There are two ways to achieve instrumentation, *static instrumentation* and *dynamic instrumentation.*

- *Static Instrumentation* means that the instrumentation code for tracing information is already present in the kernel. The toolkits built on the eBPF framework, such as "*bpftrace*" or "*BCC*" provide a macro called TRACEPOINT_PROBE, that is built around kernel tracepoints. Tracepoints are predefined kernel hooks, available in many kernel functions including system call implementations. Tracepoints are defined in the path "/sys/kernel/debug/tracing/events". The TRACEPOINT_PROBE creates a wrapper around kernel tracepoints and provides a built-in structure that holds the arguments and returns values of the system calls. We can just hook our eBPF code into these static points and read the values of the arguments as we want.
- *Dynamic Instrumentation* techniques enable programmers to insert instrumentation code into functions at run-time, and record the arguments and return values. Again, dynamic instrumentation techniques such as *kprobe*, *kretprobe*, *kfunc* etc. available in eBPF toolkits such as "*bpftrace*" or "*BCC*" enables us to place custom hooks directly into any kernel functions, at any points that we choose.

Naturally, static instrumentation is simpler and faster. So in our experiments, we have hooked our eBPF programs to static tracepoints.

We have developed two systems in total. After each experiment, we have evaluated the system performance and analyzed the flaws of the system. In the next iteration of experiments, we have tried to build a better system and again repeat the performance analyses. The first version of our system is written using "bpftrace," a script-based project built around the eBPF framework. The next version of the system is written using three languages. The kernel tracing code uses a restricted version of C, known as BPF C. This tracing code reads system call arguments and return values at specific tracepoints, and puts the

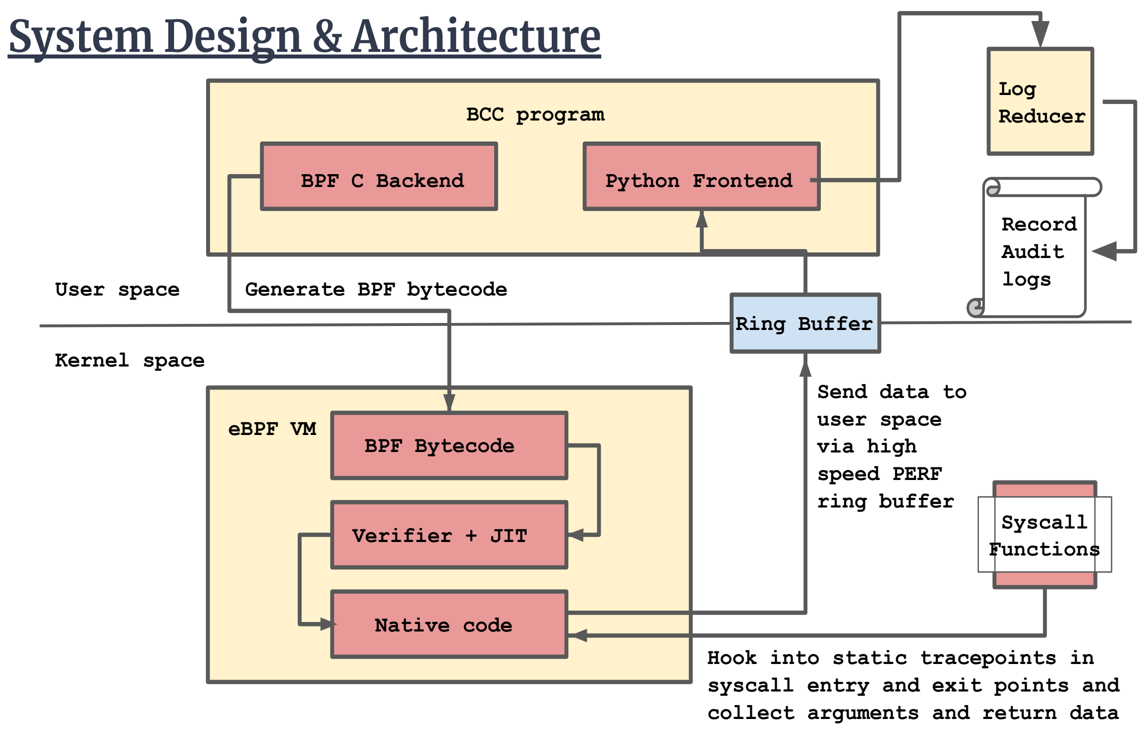
System Design & Architecture
BCC program
BPF C Backend
Python Frontend
Log Reducer
Record Audit logs
User space
Generate BPF bytecode
Ring Buffer
Kernel space
eBPF VM
BPF Bytecode
Verifier + JIT
Native code
Send data to user space via high speed PERF ring buffer
Syscall Functions
Hook into static tracepoints in syscall entry and exit points and collect arguments and return data

Figure 3.1: System Architecture (inspired from [6, 11])

data inside a high-speed shared ring-buffer. The user-space application has two components: a python engine to read from the ring-buffer, and a c++ audit log reducer which generates a compact and concise audit log, ideal for forensic analyses. The basic architecture of our system is given in the figure 3.1.

In the coming sections, we have detailed the experiments one by one. But before that, we need to describe our experimental setup.

3.1 Experimental Setup

There are many system calls in Linux [21]. However, when it comes to audit logging, not all of these are equally important. For example, system calls such as execve(2) play a crucial role in forensic analysis, as do system calls which operate on processes or files are important as well. System calls related to network communication are also very important. Naturally, the most important syscalls needed to be monitored more closely. During our experiments, we need to understand which should be the most important system calls to include in our system. Also, in most cases, the arguments are the most important artifacts to trace and record. However, in syscalls such as fstat(2) or lstat(2), the pointers in the arguments are updated when the call completes successfully. So, in those cases, we needed to trace the system calls during exit, and trace the updated values of the pointer-valued arguments. To understand which are the most frequent system calls, we have relied on a system developed in one of our lab projects, called the "*Linux Audit Consumer*".

3.1.1 Linux Audit Consumer

The Linux Audit Consumer project was created to parse the logs generated by the Linux Audit Daemon. The consumer can parse these logs and generate an output log that is compact and is optimized for forensic analyses. Naturally, for forensic analyses, the most important artifacts are file descriptors and network endpoint details, user and process ids, frequency of system calls, crucial arguments such as programs and paths passed into execve(2), etc. Also, memory access requests such as mmap(2) hold important clues to find malicious software. The consumer can identify all of these artifacts and concisely present them. We refer to the output of Audit Consumer as "*record files*".

To understand the most frequent system calls, we ran Linux Audit Daemon during normal system activities (e.g.: during system startup, kernel build), and ran the log through the Audit Consumer to generate such a summary. From the summary, we found that the top 20 frequent system calls form more than 95% of the total system calls made by the system. These top 20 frequent syscalls are *open, openat, close, read, write, mmap, mprot, munmap, execve, wait, fcntl, fork, dup, exit, connect, setuid, setgid, accept, recvmsg and sendto.* Our initial goal was to cover these most frequent syscalls first, so that our audit logging system can cover most of the system activity. Our system currently supports most of these frequent system calls, and also a few others which are not in this list, but is crucial for detecting malicious activity in the system. The complete list is given in the next subsection.

3.1.2 Syscalls supported by our system

The system calls which operate on processes and files are most crucial, as they help us to detect possible threats in the system. System calls like execve(2) are also very important for forensic analyses. For our purpose, we have picked the system calls that perform file, process, and network-related operations. Some of these system calls are the most frequent ones as well. So far, our system can support the following system calls. It can trace all arguments and return values as necessary, with crucial data such as process ids, user ids, group ids, timestamps, etc.

- *File operation related system calls*: `open`, `openat`, `close`, `read`, `write`, `stat`, `fcntl`, `dup`.
- *Network operation related system calls*: `accept`, `getpeername`, `connect`, `socket`.
- *Process and user related system calls*: `waitid`, `fork`, `exit`, `setgid`, `setuid`.
- *Memory mapping related system calls*: `mmap`, `mprotect`, `munmap`.
- *System calls for executing programs*: `execve`.

Few other system calls such as *remove* and *rename* are also important to trace from a security perspective. However, due to time constraints, we have not added the support for those system calls in our audit logger yet.

3.1.3 Evaluation Criteria

For each system that we have built in the course of this project, we have defined three benchmarks to evaluate their performance. These benchmarks were designed or chosen because they would put heavy workloads on the system. We wanted to see that how an eBPF-based audit logging framework would perform when heavy resource-consuming processes are already running in the system.

We used the bash commands "`tar`" and "`find`" to design these benchmarks. Specifically, we used the commands: "`tar cf - -one-file-system / | wc -l`" and "`find / -type f -exec stat {}  -print | wc -l`".

Apart from this, we have used two macro-benchmarks. First, we have used Linux kernel build and as a benchmark. Linux kernel build is resource-heavy and utilizes different components of a system. Second, we ran the PostMark [25] benchmark with Phoronix Test Suite [28], simulating the operation of an email server. Both these macro-benchmarks have previously been used for other audit logging systems [29, 27].

After each experiment, we have recorded and compared the run-time with a normal run (run-time of the benchmarks without our audit-logging system running in the background) and presented the data. We have used the bash command "`time`" to measure the run-times of these benchmarks in all our experiments. We have compared and reported the "*elapsed time*" for all experiments. Also, for each experiment, we ran our benchmarks thrice and took the average of the three-time records.

We have also compared the run-time of our with the Linux Auditd logs. For this comparison, we have configured the Linux Audit Daemon to record exactly similar system calls as our system. The performance tables 4.1, 4.2, and 4.3 discussed in Section 4 present these experimental results in detail. We have also compared our logged data with the Linux Auditd logs to check correctness. To do this, we have run the two systems on the same benchmark, and compared the names of opened files, or the counts of reading and write operations. We found that on all occasions, our eBPF-based audit logging system has captured the details correctly.

3.1.4 System Specifications

We have run our experiments in a virtual machine. It simulates a Intel Xeon (Skylake, IBRS) processor with 64 bit CPU with 4 cores. The clock speed of the processor is 2.2 GHz, and it has 16 GB RAM.

4 Experiments & Performance

This section presents the implementation steps in detail and the experimental results. The goal of any implementation should be to write code that is maintainable and reusable. Our final system is well structured and designed, and support for a new system call tracing can be added by adding only about 5-10 lines of code.

4.1 Experiment with bpftrace

We planned to start with bpftrace because it was the most basic wrapper project around eBPF. It is relatively small, but still supports the main eBPF features such as the ability to hook into static/ dynamic instrumentation points and send data to user-space quickly. Our system consists of 40 one-liner scripts, one each for entry and exit tracepoints of the 20 most frequent system calls described above. Once it was run, bpftrace uses a simple printf() method to directly print the values of arguments and return types to the Linux Terminal. We wrote a few lines of additional shell scripts to redirect the output to a log file instead.

Though the system call data was captured by bpftrace, there were few problems with this bpftrace based tracing program. Though it provided a better performance result as compared to the Linux Audit Daemon, it was not fast enough. The main reasons for the poor performance were:

- *Output handling*: Since it was originally intended for short one-liner performance monitoring scripts, bpftrace did not provide tailor-made data structures to send output to the end-user. It had an in-built print function, which wrote directly into the terminal. Of course, we did not want our audit log to be lost, so we redirected this output to a file. This entire process comprises a lot of system I/Os; the system writing on the terminal, and the same thing is written into a file, etc. This resulted in more time consumption.
- *Loss of events*: Since our system mediated a lot of system calls, it generated a lot of printed outputs as well. Unfortunately, the Linux terminal could not process this many print statements at a time, as it has a print buffer of a limited small size. As a result, many audit events would be captured but would be lost as the Linux terminal is a very slow consumer.
- *Tracing arguments in exit tracepoints*: bpftrace does not provide us with any shared data structures such as arrays or maps. Also, there is no way to trace arguments during exit tracepoints. In bpftrace, exit instrumentation points are only for tracing return values, and entry instrumentation points are only capable of tracing the arguments. Now, there are system calls with pointer-valued arguments, and these arguments are updated with valid data when the system call returns successfully. In case we want to trace these arguments, bpftrace does not give us a chance to do that (neither does it allow such instrumentation, nor does it have any shared data structures to hold the addresses of the pointer values arguments, so that we can get the updated values later). This lack of functionality seriously impedes the usefulness of a bpftrace based tracing system.

So how good is this bpftrace based implementation, in terms of recording audit data and presenting it? As explained above, obviously the tracing capacities are limited. Also, bpftrace uses a simple printf() method to print audit data out. Using only printf(), the only way to represent that log legibly and concisely is to add additional comments into the printed strings, so that the individual arguments and return values and process ids be understood from seeing the log. Other than this, the capacities of bpftrace are quite limited in this regard.

The next section presents the performance comparison of the bpftrace system with the Linux Audit Daemon. Though technically, the bpftrace based system does not offer a full-fledged audit data collection mechanism, it offers a basic idea of how an eBPF based logging system would work. Also, if logging in on a low scale (e.g., logging the arguments only for one system call in a standalone system with timestamp), bpftrace can still be a good choice.

4.1.1 Performance Analyses

Table 4.1 compares the run-time and log-sizes of an audit logger built using the bpftrace framework with the Linux Audit Daemon. The four evaluation benchmarks correspond to the experiments described in Section 3.1.3. The second column shows the base run-time of these benchmarks; i.e., time taken for these operations to complete without any audit logging operations running in the background. The columns for "Linux audit daemon" and "bpftrace system" display the percentage increases in run-time. There are a couple of interesting observations that can be made from this table.

- Firstly, the run-time overhead of the bpftrace based system is about 50%, when we log the audit records.
- To find out why the audit logging system is taking this much time, we designed an experiment where we just traced the system call argument data but did not log it in any file. This system, with no file writing operations, shows only about 5% run-time overhead. This tells us that the main issue with a bpftrace based audit logging system is the framework's bad handling of output.
- This system outperforms the Linux Audit Daemon both in run time and space consumption. Nevertheless, due to significant run-time overhead, this system does not serve the purpose of a full-fledged audit logger.

4.2 Experiments with BCC

The above problems with bpftrace pushed us to use BCC, a more robust and complete project built on the eBPF framework. As compared to bpftrace, there are a few features of BCC that offers much more flexibility to programmers.

- *Ability to use eBPF data-structures*: BCC offers an easy-to-use interface to use and operate on the eBPF Maps and other data structures. These eBPF maps are thread-safe, and they are protected by an RCU locking mechanism. As a result, systems written in BCC are robust enough to work on multiprocessing environments.

  One point to note here is, "eBPF Maps" does not necessarily mean Map-type data structures only. EBPF offers a set of various types of data structures, but all of them

Evaluation Benchmarks	Base run-time (minutes)	bpftrace system		Linux Auditd
		Overhead without logging	Overhead with logging	Overhead with logging
tar	2:23.50	4%	48%	327%
find	16:34.45	3%	39%	195%
kernel	10:45.31	5%	43%	203%
postmark	12:47.23	5%	47%	369%[a]

Table 4.1: Performance of the `bpftrace` based system

[a]The run with Linux auditd took too long to complete, so we terminated it after one hour. We used one hour as the runtime for overhead calculations.

are collectively referred to as “maps”. These maps add a lot of other benefits from the implementation perspective. In our case, we have leveraged these data structures in four salient ways:

– *Handle entry and exit points of syscalls*: Typically, the syscall arguments hold the most crucial information, and tracing these values is more important. So, it is useful to trace most syscalls at their entry points. However, in certain cases (such as fstat(2)), there are pointer values arguments of the syscall which only gets populated with the information once the system call has returned. In these cases, we need to track the updated values of the arguments at the time of return.

  The BPF-C back-end of BCC provides us a way to achieve this, with eBPF maps. We store the pointer-valued arguments in system call specific eBPF hashmaps, with the process id as their key. While tracing the exit point of the syscall, we check the process id and get the corresponding pointer-valued argument from the map, which, by then, has been updated with return data. In this way, we can trace the updated values of arguments as well.

– *create thread-safe global variables*: eBPF bytecodes do not have any access to the .data section, and cannot have static memory such as global variables. However, BCC allows us to use eBPF maps, which can be used much like static memory. A particular map called BPF_PER_-CPU_ARRAY can be re-purposed to create a static shared memory, which could be used as a global variable if needed.

– *large memory to store filenames etc.*: eBPF tracing functions have a maximum stack size limit of 512 bytes. In rare cases, the size of a full pathname passed as an argument to a syscall can be of more than 512 bytes. Again, by declaring eBPF maps outside the tracing methods, we can create a space to hold bigger filenames.

– *Ring buffer to send output*: BCC uses a high-performance ring buffer called

BPF_PERF_OUTPUT to send data to the userspace. The userspace code is written in Python, which can interpret the datatype of the variables sent, and process them accordingly. This buffer is thread-safe and can work in a multi-processing environment.

- *Better performance*: Since BCC avoids the pitfalls of the script-based bpftrace toolkit, the run-time performance is superior. Also, the structured output buffer is a far better option than printing directly to the trace_pipe. The buffer ensures that we do not lose data unnecessarily. Furthermore, we have full control of the audit data and can print only pertinent information in the log, resulting in more crisp and concise audit log representations.

As we can see, BCC is a more versatile toolkit as compared to bpftrace, and using it, we can build full-fledged audit systems which are lightweight and yet robust. Not only it can record all kinds of arguments and return values in non-trivial scenarios, but also it helps the developers to process and present audit data more smartly.

4.2.1 Experiment with a Structured Buffer

The BPF Compiler Collection provides us with a macro called TRACEPOINT_PROBE. This macro provides a structure called "args," which holds the arguments traced in that tracepoint. In our first BCC-based system, we have created similar structures. Whenever a syscall is traced, we would read all pertinent information (arguments, process ids, etc.) from the args structure to the structure we defined. Along with this, we would invoke eBPF helper methods to fill in important information such as user and group ids, or timestamps.

Once the structures are filled with the tracing information, we can send it to the python front-end with the help of the ring buffer. Once we have got the information in the python end, we can choose to print the information as we like. For example, we can print out all the information in the logs in detail, or we can just print a few necessary information in a tabulated form.

This sounds like a good audit logging system, but there are still potential problems with this implementation. For a start, the structure has to be initialized inside the individual tracing functions for each syscall. This will mean that the structures will be initialized in the stack, and can be of a maximum of 512 bytes. So, we will not be able to send information larger than that size.

There were other cases as well, since we were using a generic structure definition for all system calls, there were plenty of fields inside the structure which would often go unused. For example, if we have defined a field called "pathname," that might be useful while tracing openat(2) or fstat(2), but will not be useful while tracing fork(2). However, for each syscall, we have to send this huge amount of unnecessary data to the userspace. One solution would be to define individual structures for individual syscalls. But that would make the addition of new syscalls into the system quite difficult. We'll also have to keep in mind that eBPF programs can be of a maximum size of 4096 bytes, so size is a concern as well.

Table 4.2 shows the overall performance of this system. The column "BCC system with

Evaluation Benchmarks	Base run-time (minutes)	BCC system with structure buffer			Linux Auditd
		Overhead of syscall tracing	Overhead of syscall processing	Overhead of syscall logging	Overhead of audit logging
tar	2:23.50	0.1%	0.45%	0.89%	327%
find	16:34.45	0.14%	0.48%	1%	195%
kernel	10:45.31	0.15%	0.51%	0.92%	203%
postmark	12:47.23	0.15%	0.51%	0.96%	369%[a]

Table 4.2: Performance of the "BCC" based system with a structured buffer

[a]The run with Linux auditd took too long to complete, so we terminated it after one hour. We used one hour as the runtime for overhead calculations.

structured buffer" displays the run-time overheads for each set of experiments. There are three sub-columns under this, as we conducted three experiments in total. Apart from recording the run-time overhead of the BCC based audit logging system, we wanted to know the run-time overhead if a) we just intercepted the system calls and traced the data (we call this interception overhead), and b) we intercepted the data and process it (fill our predefined structures with the arguments and other process-related tracing data), but did not send it to the user-space. This table shows the results from those experiments. We see that there is a negligible overhead just to trace the system calls. This low overhead is primarily due to the usage of static tracepoints to instrument our system calls.

The most interesting thing to be noticed is the decrease in the run-time overhead of the BCC system with logging enabled. As explained before, the "BCC" framework provides a shared ring buffer to send data to user-space. The user-space comes with a python front-end, which has built-in methods to poll data from the buffer and process them. In our case, we have just inserted the data into a log file.

The run-time performance of this system looked good. Presentation-wise, we still had scopes of improving the log, and present it in a more readable and concise way. These bottlenecks pushed us to develop a system more generic in nature. Specifically, we were trying to address two problems:

- A generic and versatile way to send the traced data to user-space, that uses and transmits only the required data. This would ensure that syscalls such as fork(2), where there's a little tracing data to be sent, and syscalls such as fstat(2), where there are quite a lot of bytes to be sent, can be handled by one single buffer.
- A more structured and concise way to represent the audit log. In this regard, we thought of integrating the logs generated by the system with the Linux Audit Consumer project, described in Section 3.1.1. The Audit Consumer already processed logs from the Linux Audit Daemon, so it was perfectly able to process and concisely present the logs from our BCC-based system as well.

- Adding new code to support tracing and logging of newer system calls should be easy and hassle-free.

4.2.2 Experiment with a Generic Buffer

For our new buffer, we just created an array of unsigned 8-bit integer types. If we would get character-type arguments or 8-bit integers, we would directly copy them to the 8-bit buffer elements. For larger types, we would copy them serially into continuous elements of the buffer. We added a header section to the buffer so that at the recipient end, the Audit Consumer would know the count, size, and type of each type of element in the buffer.

With this setup, we repeated the previous experiment. Once the data reached the Python end, we sent it to the Audit Consumer, written in C++. For a seamless communication between the Python and the erstwhile C++ system, we have used a common Python utility known as "ctypes".

One important aspect of this third experiment is that the Audit Consumer processes the audit records generated by the BCC-based system. Then, it produces a compressed and concise log with the system call arguments and a summary, as opposed to the logs generated by the previous eBPF-based systems where we were just printing out the audit data to a file. As a result, this version of the auditing system runs for a longer time. In the end, the audit consumer produces a record file, which consumes much lesser space than the Linux Auditd logs.

4.2.3 Performance Analyses

The final performance data of this generic buffer-based system is given in table 4.3. In this table, we compare the run-time overheads of our system with Linux Auditd. This table also compares the sizes of the output log files. Since the Audit Consumer generates a compressed record file as output, we have compared the size of the file with compressed Linux Auditd logs.

- We see that the run-time overhead has increased from the earlier structured buffer-based system. This additional run-time overhead is added due to the post-processing and reduction of the audit data done by the Audit Consumer. The Audit Consumer also compresses the output record file, which is time-consuming. In the end, the entire pipeline adds 15% run-time overhead. However, the resultant record file is ready for accurate forensic analyses.
- The table also compares the size of compressed log files, generated by the Audit Consumer and Linux Auditd. The sizes of the record files produced by Audit Consumer are much smaller than the sizes of corresponding Linux Auditd logs. So, not only do we get concise audit records, but we also consume less storage space.
- If we modify the Audit Consumer to produce non-compressed record files, then the run-time overhead decreases significantly. The third column of table 4.3 shows the

Evaluation Benchmarks	Base run-time (minutes)	BCC System with generic buffer			Linux Auditd	
		Overhead of tracing & producing record files	Overhead of tracing & producing compressed record files	Size of the compressed record files (MB)	Overhead of audit logging	Size of compressed audit log (MB)
tar	2:23.50	3%	14%	12	327%	165
find	16:34.45	4%	15%	14	195%	220
kernel	10:45.31	4%	15%	15	203%	229
postmark	12:47.23	3%	14%	14	369% [a]	453

Table 4.3: Performance of the "BCC" based system with generic buffer & Audit Consumer

[a]The run with Linux auditd took too long to complete, so we terminated it after one hour. We used one hour as the runtime for overhead calculations.

run-times. This proves that most of the run-time overhead comes from compressing the record files.

5 Related Work

Audit logs can be used for different purposes. Many auditing systems try to meticulously capture the full movement of data through every layer of the system. On the other hand, there are auditing systems that try to identify threats in the system, and generates logs more suited for forensic analyses. Depending on their particular purpose, auditing systems employ different techniques (such as building around Linux Security Modules, mediating kernel objects, tracing system calls, etc.). There have been researches in the past years to find better audit logging solutions as an alternate for the Linux audit daemon.

The property that defines completeness of audit logs, i.e., logs should capture all system events faithfully, is called *fidelity*. The best way to achieve fidelity is by tracing kernel objects, as in that way, all system-level activities could be traced. The auditing systems whose goal is to achieve high fidelity try to capture the movement of data between different layers of a system. Whenever kernel objects (like an inode, a process, etc.) are invoked/ accessed, hooks placed inside the kernel are used to mediate those accesses and collect audit data.

Mediating kernel objects require researchers to put instrumentation code inside various pre-defined kernel hooks. For example, Hi-Fi [29] has built their auditing system on the existing LSM and Netfilter frameworks. While LSM was created for mediating access to kernel objects, Netfilter provides hooks to mediate and process Network packets. Another auditing system, CamFlow [27], uses the same LSM and Netfilter hooks. Camflow also creates graphs that capture the state changes of Kernel Objects. While these techniques require new modules and code to be loaded into the Kernel, auditing systems such as PASS [24] modifies the Kernel with a new storage system, which can track the complete audit information and lineage of data.

However, for detecting and preventing APTs, we need to track down malicious processes, for which we need to do comprehensive forensic analyses. The best and optimum way to do this is to track processes in the granularity of system calls, as the system calls would give us the most crucial information for a successful forensic analysis. To achieve this, we need to trace the arguments and return values of system calls.

ProTracer [19] uses a set of pre-defined static hooks in the Linux Kernel called tracepoints to hook into Kernel Functions, and record the parameters. We have already discussed tracepoints in Section 3. ProTracer collects logs of almost all system calls. There could be custom-defined tracepoints as well, such as the Linux Trace Tools New Generation, more commonly known as LTTng [8]. LTTng is a tracer core, offering an OS instrumentation. This allows us to insert tracepoints inside kernel code, and record system activity. Kohyarnejadfard et al. has proposed a system performance anomaly detection system that has been built on top of LTTng [16]. This work uses the LTTng core to trace system calls and collect audit logs for distributed systems. Another work, LPROV [35] offers a kernel-level system call tracer that combines the audit data for dynamic libraries as well. Typically just syscall tracing would not include audit details of libraries that are linked dynamically and executed in run-time. Upon a system call, LPROV finds out the library calls along with the stack that induces it. In this way, this work tries to detect and protect systems against malicious library attacks.

As explained above, other than introducing a huge run-time overhead, Linux audit daemon also brings with it a very large storage space requirement, generating log data in the order of several gigabytes per day. To reduce this storage overhead, yet preserve the dependencies of events in the log (for forensic analyses), many research works focus on log-reduction techniques. There can be two ways for log reduction. Either we can have the audit logging systems generating fewer logs, or use specific dependency preserving log-reduction algorithms. When it comes to developing audit systems that generate fewer logs themselves, different works employ different techniques to generate reduced logs. ProTracer, described above, uses 'logging and tainting' to store only the writes and create taint-sets for the read operations. BEEP [17], on the other hand, divides program binaries into logical units based on event-handling loops. Then, it proceeds to selectively log the operation of those units and their dependencies. Another work also follows a similar style of adaptive auditing technique; the researchers study precursory events to identify possible system or network activity that may follow and logs the important events adaptively. Another system, KCAL [18] is an in-kernel cache-based log-reduction system. KCAL uses multilayered caching schemes, distributed in many kernel data structures. to find and discard redundant audit events. The scheme indexes largely scattered syscall events and suppresses the events if already recorded.

Other techniques to reduce unnecessary logs from storage is to include kernel modules and libraries, that need to be installed into the kernel and can be used by user-space applications to generate customized audit logs. CPL[20] or Core Provenance Library presents a work, which can be used as an API while writing user-space programs (written in C, Perl, or Java) and applications. With nominal extra coding efforts, this library can generate application-specific audit logs.

All of the above ways of log reduction involve kernel modifications in some way or the other. Besides, often the above techniques result in loss of dependencies among events, and as a result, forensic analyses using such logs are difficult. In certain cases, fine-grained instrumentation is required. Techniques such as using a separate library require human intervention and are impractical for large organizations to implement.

Other log-reduction techniques take the raw audit logs as inputs and develop parsing mechanisms to analyze them to reduce size and verbosity, without losing information crucial for forensic analysis. Xu et al. [36]proposed a similar work, where the researchers have proposed aggregation algorithms to preserver the dependence of events during the logging of audit data, and later using a reduction algorithm that preserves the dependence. Hossain et al. [12] proposes efficient log-reduction techniques while preserving crucial dependence information. These reduced logs can be used for crucial forensic analyses like backtracking or impact analysis.

While the above works talk about developing audit loggers with better run-time and storage performances than Linux Audit Daemon, it's equally important to safeguard the logger systems and data. If and when a system is compromised, the attackers will naturally try to hide all the traces and would try to erase log data and corrupt the logging systems. Protecting log data is trivial. We can simply redirect the logs to a remote secure server, ensuring the logs stay unscathed even if the system is compromised. However, securing the logger application is not trivial. Linux Provenance Modules [1] provides a trusted auditing environment. This paper uses custom-made hooks rather than LSM or Netfilter hooks used in previous works. This system can be extended to implement Hi-Fi and CamFlow systems discussed above, making those more trustworthy and tamper-proof. Further, works like SGX-LOG [13], CUSTOS [26] etc. uses Intel SGX, a hardware extension, so that the logging system can execute in a separate secure container. SGX-LOG also uses *syslog* server to generate audit data and store it in persistent storage.

eBPF is yet another framework, which uses a virtual machine inside the kernel to trace system calls (and all kernel activities) by hooking into specified Kernel Functions. eBPF code is JIT compiled and needs no new modules to be installed in the Kernel. It is a very new framework, and we believe eBPF's full potential is yet to be explored. So far, eBPF has been used to intercept external communications (network or peripheral based communications), kernel function tracing and profiling, and securing the system against threats. Researchers have tried to use eBPF in different kinds of tracing and security policing scenarios, and develop complex network functions to intercept network communications using the tracing capacities of eBPF [23]. Bertrone et al. [2]uses eBPF hooks to intercept and filter network traffic based on source and destination IP addresses. They try to develop a prototype system of an eBPF-based iptable.

Linux (e)BPF Modules [33], an eBPF based framework, intercepts peripheral communication from devices connected via Bluetooth, USB/ NFC, etc. The design incorporates eBPF hooks in these drivers to intercept the communication traffic, and discard the packets with suspicious contents. Another recent work(ITASEC) uses eBPF hooks in specific tracepoints of network functions to develop a system-level security policer. This system can create fine-grained security policies for different users, processes, or containers.

Sysdig [15] is yet another eBPF-based an open-source, cross-platform, powerful, and flexible system monitoring and troubleshooting tool for Linux. Sysdig also works on Windows and Mac OSX (but with limited functionality) and can be used for system analysis, inspection, and debugging. There are other works that are built upon Sysdig.

Deri et al. [7] use Sysdig for collecting streams of system calls produced by all or selected processes on the hosts and sending them over the network to a monitoring server. Thereafter, machine learning algorithms are used to identify changes in process behavior due to malicious activity, hardware failures, or software errors. Falco [32] is yet another eBPF based framework that provides cloud-based run-time security.

6 Conclusion

The eBPF framework provides a safe and convenient way to write sandboxed code inside the Linux kernel, without the requirement of loading new modules in the kernel. In this project, we have used *bpftrace* and *BPF Compiler Collection*, two toolkits built over the eBPF framework to collect system audit data. To trace system call arguments, we hook our eBPF tracing code into predefined kernel tracepoints.

Our audit logging system offers low run-time overhead. So far, our system supports 20 system calls. However, this can support all the system calls in Linux Kernel. It has the potential to be a full-fledged system for collecting audit data, and thus help in detecting threats such as malware and APTs.

7 Future Work

The next goal is to extend the functionalities of this system so that it can support all other system calls. It takes only a few lines of code to add support for a new system call in our audit logger.

As explained in Section 2, the eBPF bytecode can only handle programs that have a maximum of 4096 instructions. Our system has almost reached that limit. As a result, the addition of a lot of newer system calls will not be possible if we continue to use the current system. To solve this, the best way is to use different BCC programs for different groups of syscalls, and use them separately or collectively as needed.

Other future work involves developing a selective logging mechanism in the system so that the system could be configured to log only specific arguments of a system call. It is possible to develop an audit-reduction mechanism as well. For instance, the system would be able to detect and drop redundant audit records. The system could also be configured to generate aggregated output records (e.g.: show how many times a certain syscall was invoked, rather than the individual arguments of each of those calls).

8 Appendix

This section lists the full set of system calls supported by the Audit Consumer, but not supported by our audit logging system yet. There are multiple reasons for our system not supporting a certain syscall. In most cases, we did not manage enough time to add support for all of them. The space constraint mentioned in Section 7 has also been a bottleneck.

In many cases where we haven't added support for a certain syscall, we have added it for at least one syscall with almost similar functionality. For example, support for preadv(2), pread64(2) is not there, but read(2) is supported, and it will require a trivial effort to add support for preadv and pread64.

Some network-related system calls have been left out in this version. Adding support for these syscalls is non-trivial, because the address types (ipv6, ipv4), address length, etc. depend on the socket families and other syscall-specific arguments. Our system supports syscalls like connect(2) and accept(2), but the functionality for choosing the correct address type and length is yet to be added. In few cases, such as stat(2), our system has logging support, though the Audit Consumer does not support them yet.

Syscall Number	**Syscall Name**	**Traced?**	**Remarks**	**Trivial/ Non-trivial**
0	read	yes		Trivial
1	write	yes		Trivial
2	open	yes		Trivial
3	close	yes		Trivial
9	mmap	yes		Trivial
10	mprotect	yes		Trivial
11	munmap	yes		Trivial
17	pread64	no	We have already added support for read. We could not add support for it due to time-constraint.	Trivial
18	pwrt64	no	We have already added support for write. We could not add support for it due to time-constraint.	Trivial
19	readv	no	We have already added support for read. We could not add support for it due to time-constraint.	Trivial
20	writev	no	We have already added support for write. We could not add support for it due to time-constraint.	Trivial
22	pipe	no	This is an important syscall. eBPF doesn't provide access to the flag argument passed in this syscall. Also, the file descriptors are passed on as a pointer array which needs to be dereferenced.	Non-trivial
32	dup	yes		Trivial
33	dup2	no	We have already added support for dup. We could not add support for it due to time-constraint.	Trivial
41	socket	yes		Trivial
42	connect	yes	Our system has support for this syscall. But the address format needs to be handled.	Non-Trivial
43	accept	yes	Our system has support for this syscall. But the address format needs to be handled.	Non-Trivial

Syscall Number	Syscall Name	Traced?	Remarks	Trivial/ Non-trivial
44	sendto	no	The dest_addr field needs to be traced based on the socket type.	Non-Trivial
45	recvfrm	no	The address length needs to be figured out depending on the source address and socket type.	Non-Trivial
46	sendmsg	no	The address length, type and field needs to be decided based on the msg_header.	Non-Trivial
47	recvmsg	no	The address length needs to be figured out depending on the source address and socket type.	Non-Trivial
49	bind	no	The length and type of the address depends on the sockaddr structure. This has to be traced first and then the address can be recorded.	Non-Trivial
50	listen	no	This is trivial. We could not add a support for this due to time-constraint.	Trivial
51	getsknm	no	The address and length of the socket has to be recorded from the pointer -valued sockaddr buffer.	Non-Trivial
52	getpeer	yes	Similar to accept(2), our system has support for this syscall. But the address format needs to be handled.	Non-Trivial
53	sckpair	no	Similar to pipe(2), the pointer valued fd array needs to be dereferenced.	Non-Trivial
56	clone	yes		Trivial
57	fork	yes		Trivial
58	vfork	no	We have already added support for fork. We could not add support for it due to time-constraint.	Trivial
59	execve	yes		Trivial
60	exit	yes		Trivial

Syscall Number	**Syscall Name**	**Traced?**	**Remarks**	**Trivial/ Non-trivial**
61	wait4	yes		Trivial
62	kill	no	We could not add a support for this syscall due to time constraint.	Trivial
72	fcntl	yes		Trivial
76	trunc	no	We could not add a support for this syscall due to time constraint.	Trivial
77	ftrunc	no	We could not add a support for this syscall due to time constraint.	Trivial
82	rename	yes		Trivial
83	mkdir	no	We could not add a support for this syscall due to time constraint.	Trivial
84	rmdir	no	We could not add a support for this syscall due to time constraint.	Trivial
85	create	no	We could not add a support for this syscall due to time constraint.	Trivial
86	link	no	We could not add a support for this syscall due to time constraint.	Trivial
87	unlink	no	We could not add a support for this syscall due to time constraint.	Trivial
90	chmod	no	We could not add a support for this syscall due to time constraint.	Trivial
91	fchmod	no	We could not add a support for this syscall due to time constraint.	Trivial
92	chown	no	We could not add a support for this syscall due to time constraint.	Trivial
93	fchown	no	We could not add a support for this syscall due to time constraint.	Trivial

Syscall Number	**Syscall Name**	**Traced?**	**Remarks**	**Trivial/ Non-trivial**
94	lchmod	no	We could not add a support for this syscall due to time constraint.	Trivial
101	ptrace	no	We need to trace the pointer valued structures addr and data.	Non-trivial
105	setuid	yes		Trivial
106	setgid	yes		Trivial
113	setreu	no	We have already added support for setuid. We could not add support for it due to time-constraint.	Trivial
114	setreg	no	We have already added support for setgid. We could not add support for it due to time-constraint.	Trivial
117	setresu	no	We have already added support for setuid. We could not add support for it due to time-constraint.	Trivial
119	setresg	no	We have already added support for setgid. We could not add support for it due to time-constraint.	Trivial
231	exitgrp	no	We have already added support for exit. We could not add support for it due to time-constraint.	Trivial
257	openat	yes		Trivial
258	mkdirat	no	We could not add a support for this syscall due to time constraint.	Trivial
260	fchwnat	no	We could not add a support for this syscall due to time constraint.	Trivial
263	unlnkat	no	We could not add a support for this syscall due to time constraint.	Trivial
264	renmat	no	We have already added support for rename. We could not add support for it due to time-constraint.	Trivial
265	linkat	no	We could not add a support for this syscall due to time constraint.	Trivial

Syscall Number	**Syscall Name**	**Traced?**	**Remarks**	**Trivial/ Non-trivial**
268	fchmdat	no	We could not add a support for this syscall due to time constraint.	Trivial
288	accept4	no	We have already added support for accept. We could not add support for it due to time-constraint.	Non-trivial
292	dup3	no	We have already added support for dup. We could not add support for it due to time-constraint.	Trivial
293	pipe2	no	This is an important syscall. The file descriptors are passed on as a pointer array which needs to be dereferenced.	Non-trivial
295	preadv	no	We have already added support for read. We could not add support for it due to time-constraint.	Trivial
296	pwritev	no	We have already added support for write. We could not add support for it due to time-constraint.	Trivial
299	rcvmmsg	no	The address length needs to be figured out depending on the source address and socket type.	Non-Trivial
307	sndmmsg	no	The address length, type and field needs to be decided based on the msg_header.	Non-Trivial
316	renmat2	no	We have already added support for rename. We could not add support for it due to time-constraint.	Trivial
322	execat	no	We already have support for execve(2). We could not add support for this due to time-constraint.	Trivial
327	preadv2	no	We have already added support for read. We could not add support for it due to time-constraint.	Trivial
328	pwrtv2	no	We have already added support for write. We could not add support for it due to time-constraint.	Trivial

References

[1] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security*, 2015.

[2] M. Bertrone, S. Miano, F. Risso, and M. Tumolo. Accelerating linux security with ebpf iptables. In *ACM SIGCOMM Posters and Demos*, 2018.

[3] L. K. bpf helpers Manpage. bpf-helpers(7) — linux manual page. `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`.

[4] E. Cao, J. C. Chen, W. G. Sanchez, L. Wu, and E. Xu. Energetic bear: more like a crouching yeti. https://documents.trendmicro.com/assets/Tech-Brief-Operation-Poisoned-News-Hong-Kong-Users-Targeted-with-Mobile-Malware-via-Local-News-Links.pdf.

[5] cilium. Bpf and xdp reference guide. `https://docs.cilium.io/en/latest/bpf/`.

[6] cilium. ebpf official documentation. `https://ebpf.io/what-is-ebpf/`.

[7] L. Deri, S. Sabella, and S. Mainardi. Combining system visibility and security using ebpf. In *ITASEC*, 2019.

[8] M. Desnoyers and M. R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. Citeseer, 2006.

[9] Fireeye. Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with sunburst backdoor. https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html.

[10] K. L. Global Research & Analysis Team. The darkhotel apt. `https://securelist.com/the-darkhotel-apt/66779/`.

[11] B. Gregg. ebpf architecture. `https://www.brendangregg.com/eBPF/linux_ebpf_internals.png`.

[12] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller. Dependence preserving data compaction for scalable forensic analysis. In *USENIX Security*, 2018.

[13] V. Karande, E. Bauman, Z. Lin, and L. Khan. Sgx-log: Securing system logs with sgx. In *ASIA CCS*, 2017.

[14] kernel.org. Linux socket filtering aka berkeley packet filter (bpf). `https://www.kernel.org/doc/Documentation/networking/filter.txt`.

[15] A. Kili. Sysdig – a powerful system monitoring and troubleshooting tool for linux. `https://www.tecmint.com/sysdig-system-monitoring-and-troubleshooting-tool-for-linux/`.

[16] I. Kohyarnejadfard, M. Shakeri, and D. Aloise. System performance anomaly detection using tracing data analysis. In *ICCTA*, 2019.

[17] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[18] S. Ma, J. Zhai, Y. Kown, K. Hyung Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran,

D. Xu, and S. Jha. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX ATC*, 2018.

[19] S. Ma, X. Zhang, and D. Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[20] P. Macko and M. Seltzer. A general-purpose provenance library. In *USENIX TaPP*, 2012.

[21] L. K. S. C. Manpage. syscalls(2) — linux manual page. `https://man7.org/linux/man-pages/man2/syscalls.2.html`.

[22] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Winter USENIX Conference*, 1993.

[23] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *HPSR*, 2018.

[24] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *USENIX ATC*, 2006.

[25] OpenBenchMarking.org. Postmark. `https://openbenchmarking.org/test/pts/postmark`, 2020.

[26] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *NDSS*, 2020.

[27] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon. Practical whole-system provenance capture. In *SoCC*, 2017.

[28] phoronix. Phoronix test suite. `https://github.com/phoronix-test-suite/phoronix-test-suite/`, 2021.

[29] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. Hi-Fi: Collecting high-fidelity whole-system provenance. In *ACSAC*, 2012.

[30] A. Prakash. Linux runs on all of the top 500 supercomputers, again! `https://itsfoss.com/linux-runs-top-supercomputers/`, 2014.

[31] P. Shoshin. Lightspy spyware infects ios. https://usa.kaspersky.com/blog/lightspy-watering-hole-attack/21301/.

[32] Sysdig. The falco project. `https://falco.org/`.

[33] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. Butler. Lbm: a security framework for peripherals within the linux kernel. In *IEEE (SP)*, 2019.

[34] S. J. Vaughan-Nichols. Can the internet exist without linux? `https://www.zdnet.com/article/can-the-internet-exist-without-linux/`, 2014.

[35] F. Wang, Y. Kwon, S. Ma, X. Zhang, and D. Xu. Lprov: Practical library-aware provenance tracing. In *ACSAC*, 2018.

[36] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM CCS*, 2016.