Extracting Reliable Data for Short-lived Processes using eBPF for Linux Security Threat Analysis

Avtansh Gupta
MDE Linux
Microsoft Corporation
Noida, India
avtanshgupta@microsoft.com

Ayush Garg
MDE Linux
Microsoft Corporation
Hyderabad, India
ayugarg@microsoft.com

Meghna Vasudeva
MDE Linux
Microsoft Corporation
Hyderabad, India
meghna.vasudeva@microsoft.com

Lakshmy A V
MDE Linux
Microsoft Corporation
Hyderabad, India
lakshmyav@microsoft.com

Ankit Garg
MDE Linux
Microsoft Corporation
Noida, India
gargank@microsoft.com

Kranthi Mullaguru
MDE Linux
Microsoft Corporation
Hyderabad, India
kmullaguru@microsoft.com

Abstract— **Endpoint Detection and Response (EDR) solutions continuously monitor the events occurring on an endpoint, create detailed process timelines, and analyze these data, to reveal suspicious patterns that could indicate threats such as ransomware. To create these detailed timelines, EDR solutions collect a variety of information about each process running on the endpoint, such as timestamp, PID, process name, path, command line, etc. On Linux systems, this is often done using the proc filesystem provided by the operating system, which provides rich information for each process currently running on the system. However, if a process is short-lived and exits quickly, the proc filesystem entries for it get cleared out before the EDR solution can read them, leading to incomplete timelines. To take a simple example, suppose a malicious actor runs a script that downloads a binary from the network and then executes it. This downloaded binary quickly spawns a bunch of long-running malicious processes and exits itself. If EDR solution is unable to extract the complete process information about the execution of the downloaded binary from proc filesystem (being a short-lived process), it'll miss details about the creator of the malicious process in the system. Hence, EDR solution will have visibility gaps about the downloaded binary.**

We propose a solution to address the gaps by attaching extended Berkeley Packet Filter (eBPF) programs dynamically to the Linux kernel system calls (such as fork, exec and exit) and extracting all the required information directly from the kernel hooks using BPF Type Format (BTF). Our proof of concept (PoC) shows that eBPF-based process data extraction provides process timelines with upto 100% reliability, compared to proc-filesystem-based approaches which have a reliability of only around 83%.

Keywords—eBPF, BTF, EDR

I. INTRODUCTION

Endpoint Detection and Response (EDR) solutions are essential tools in the cybersecurity landscape, designed to continuously monitor end-user devices to detect and respond to cyber threats targeting endpoints. These solutions log behaviours on endpoints around the clock, analysing this data to reveal suspicious activity that could indicate threats such as ransomware. The continuous monitoring and logging are crucial for creating process timelines that help in understanding the full history and scope of a security breach.

Process timelines are created by collecting a variety of information about the processes on the endpoint, such as process id (PID), process creation time, process name, process path, command line arguments, process termination time, and more. On Linux systems, the most prevalent approach adopted by EDR solutions to obtain this information is using the in-memory /proc[2] file system (*procfs*)[3], which is a virtual file system provided by the Linux kernel that provides an interface to kernel data structures for each process currently running on the system. For long-lived processes, this provides reliable information about processes on the system. However, there are some processes which have a very short life span. These short-lived processes may get terminated before any information can be extracted from *procfs*. Hence, this results in unreliable or missing information on the timeline. EDR detections are dependent on accurate sequence of events happening on the endpoint, any missing information can hamper such detections.

This paper aims to solve this problem by leveraging extended Berkeley Packet Filter (eBPF). We present a proof-of-concept for extracting all the required information about the processes within an eBPF program which runs in kernel space and is attached to system calls like fork, exec, exit and extracts all information within the kernel hook by utilizing BTF (BPF Type Format). BTF provides detailed type information about kernel data structures, such as the task structure, which further provides all information about the process. The information is then passed on to the user space for the creation of the process timelines. This prevents reliance on *procfs* by collecting the required information within the kernel itself which would be more reliable and complete. This method is particularly effective for short-lived processes where the process exits quickly. The *procfs*-based approach fails primarily for short-lived processes as data in *procfs* is erased as soon as the corresponding process exits, making it unreliable for capturing information about short-lived processes.

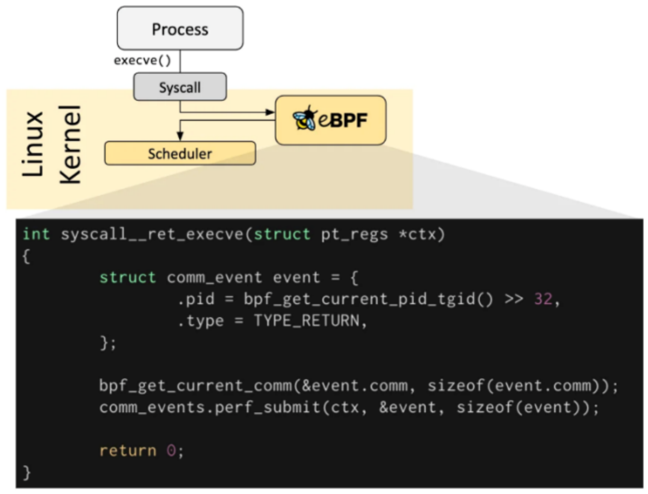
The proof-of-concept implementation using eBPF and BTF shows that reliability of data increased to 100% for short lived process as compared to method based on *procfs* which was previously 83%. This enhanced visibility is crucial for effective threat detection and response, ultimately improving the overall security posture of the organization

This paper is organized as follows. In Section II, we discuss the background on eBPF and BTF. The related work on process data extraction is explored in Section III. Section IV presents the architecture, design and implementation of our PoC. In Section V, we discuss the evaluation and results of our PoC compared to *procfs*-based solutions. The limitations of our PoC are documented in section VI.

II. Background

A. *Extended Berkeley Packet Filter (eBPF)*

eBPF[9] is a technology for Linux that allows dynamically loading custom programs into the kernel and running these programs within a sandboxed environment in the kernel space. eBPF programs can be attached to various pre-defined hooks in the Linux kernel, such as system calls, function entry/exit, kernel trace points, sockets, etc., and perform some action whenever an application passes these hooks. eBPF programs are written in a safe, limited instruction set, and the eBPF Just-In-Time (JIT) compiler and verification engine ensure that an eBPF program is loaded into the kernel only if it is safe to run, thereby guaranteeing that any eBPF program we load into the kernel will not lead to undesirable side effects such as kernel crashes/hangs. Thus, eBPF programs allow us to gain complete visibility into events happening on the system by modifying the behavior of the kernel on-the-fly, without requiring direct changes to the kernel source code (which could take years) or loading kernel modules (which may not guarantee runtime safety). eBPF programs can be used for a variety of tasks, such as packet filtering, network performance monitoring, and event tracing for endpoint security solutions.



```
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

Figure (i): eBPF overview [1]

B. *BPF Type Format (BTF)*

Most eBPF programs use hooks that retrieve information from the system call or trace point arguments, but some hooks involve extracting additional information from kernel data structures. To do that, the eBPF program would have to know the internal details of these data structures, such as the exact location of specific data fields in memory at run-time. Linux kernel data structures can vary from one Linux distribution to another, or from one kernel version to another, and can even change based on the compilation flags used for building a specific kernel. How, then, can we write an eBPF program that can be portable across kernel versions? This problem is solved using BPF Type Format, or BTF, which describes the internal layout of kernel data types and data structures across different kernel versions and distributions. The abstraction provided by BTF allows programmers to write eBPF programs that can be compiled once but run everywhere (the CO-RE philosophy). The eBPF verifier uses this *type* information to ensure type-correctness and ensure runtime safety of eBPF programs across kernel versions and platforms. [7]

C. *Short-lived process*

A short-lived process can be defined as a process which has a very short lifespan. The lifespan of these processes is so short that they are not visible on most of the interval-based sampling tools like top. For example, ls, cat, chmod.

Adding details on existing ways to extract process data and their shortcomings:

D. *The procfs*

On Linux systems, enriching process event details to gather comprehensive information about the processes running on the system is typically done using the /proc filesystem, also known as *procfs*. The *procfs* is a virtual file system that provides an interface to kernel data structures, offering detailed information about each process, such as its PID, command line arguments, and current working directory. This information is crucial for monitoring, reporting, and security tools.
However, there are certain pitfalls in relying on *procfs*, as data is erased as soon as the corresponding process exits. For example, if a process *execs* soon after it is created, the original process's information may be lost before it can be extracted from *procfs*. While *procfs* remains a valuable resource for enriching process event details for long-lived processes, it is unreliable for short-lived processes due to the rapid erasure of data.

E. *Netlink process connector*

The Netlink process connector is a powerful mechanism in the Linux kernel that facilitates communication between the kernel and user-space processes. It is particularly useful for monitoring and managing process events such as creation, termination and other state changes. The Netlink process connector provides a standardized interface for sending and receiving messages related to process events, allowing user-space applications to efficiently track and respond to these events.
However, there are some shortcomings when using the Netlink process connector, the predominant one being that it is limited to providing only numerical data regarding the process events, such as the process id, but does not more details such as the corresponding executable path or current working directory. This would mean that any consumer of this information would have to additionally rely on the *procfs* for the other details, which could lead to race conditions especially in the case of short-lived processes.
[6]

F. *Auditd*

Extracting process information using auditd[4] involves utilizing the Linux Auditing System's user-space component to monitor and record system activities based on rules defined by the system administrator. Auditd captures detailed

information about security-relevant events, logging them for later analysis. This can range from system startup and shutdown to file access, network events, and even security breach attempts. Auditd is instrumental in security monitoring and compliance, helping track any changes or attempts to change sensitive parts of the system, ensuring that any unauthorized access attempts are logged and can be investigated.
However, there are some disadvantages to using auditd. One major issue is the potential performance impact on the system, as auditd can generate a large volume of logs, especially in environments with high activity levels. Additionally, configuring and managing auditd rules can be complex as these are system wide and can result in conflicts with rules added by other applications on the system.

Using eBPF we're able to overcome these issues of reliability, performance and completeness of data for short-lived processes.

III. Architecture / Implementation

The new architecture was designed keeping in mind the problems faced by the previously available solutions. As discussed in the problem statement the current solutions are failing as there is a delay in reading the event data causing a loss of data which in turn points to data invalidity. So, the proposed solution here is to start obtaining the data at the source. The eBPF event which is received at the hook can be leveraged to gather the data. This data will be close to the *syscall* event generation itself.

Below we can find the details on the hooks used and the algorithm followed for the capturing of the data for these events.

A. Architecture Design and Implementation

In the architecture we will first discuss the syscall hooks and trace points that we are using to capture the data, and this will be followed by the exact data capturing flow and information about which hook will be giving us which part of the data.

- Exec Hooks:
  - **tracepoint/syscalls/sys_enter_execve and tracepoint/syscalls/sys_enter_execveat:**
    sys_enter_execve is a tracepoint that is used in Linux systems to monitor the entry of the execve system call, which is responsible for executing a program. This tracepoint provides valuable insight into what programs are being executed on a system. This tracepoint helps us capture the pid, operation time and the command line for the process. Note that this is one of the tracepoint hooks that provides us with the command line details, which are not provided by other hooks for exec system call such as sched_process_exec. The details on how each event is captured is mentioned below in the implementation section.
  - **tracepoint/syscalls/sys_exit_execve and tracepoint/syscalls/execveat:**
    sys_exit_execve is a tracepoint in the Linux kernel that is triggered when the execve system call exits. This tracepoint provides information about the outcome of the execve system call. This tracepoint helps us capture some critical data related to the process. The data captured from this tracepoint are parent process id, process start time, process path and current working directory of the process. The parent pid, process path and current working directory are captured using the task structure of the process. The details of the same are mentioned below.
- Fork Hooks:
  - **tracepoint/task/task_newtask:**
    task_newtask is a tracepoint in the Linux kernel that is triggered whenever a new task (process or thread) is created. When a new task is created (through fork, clone, or a similar system call), the task_newtask tracepoint is triggered. Here, we have chosen this particular hook over other hooks such as sys_enter_fork, because we are interested in extracting information from the task data structure, which is guaranteed to be populated at the time when task_newtask is called, but may not have been populated at the time of sys_enter_fork, etc. This tracepoint provides us with details about the process id, process thread id, process start time, process path and process current working directory. The process and current working directory for this tracepoint are also captured using the task structure for the event. Since the fork/clone process is a copy of the parent process the task structure for this also gives us the information here for the parent process. The details on how this information is captured using the task structure are mentioned below.

- For all the hooks used in our implementation, we check for the return code of the corresponding system call in the kernel space, and if it is negative, indicating a failure in the syscall invocation, we do not forward this further to the user space.

- **Dentry Structure**: All the paths mentioned above like the process path, mount point paths and the current working directory are captured using the dentry structure which is used to store the paths in the kernel.
  The dentry (directory entry) structure in the Linux kernel is a fundamental part of the Virtual Filesystem (VFS) layer, which provides a uniform interface for different filesystems. The dentry structure represents a single component of a pathname.
  The exact algorithm used to calculate the paths from the dentry structure is mentioned below in the implementation.

- **Task Structure**: As mentioned above we have used task_struct for enriching various information from the various hooks.
  The task_struct is a fundamental structure in the Linux kernel that represents a process or thread. It

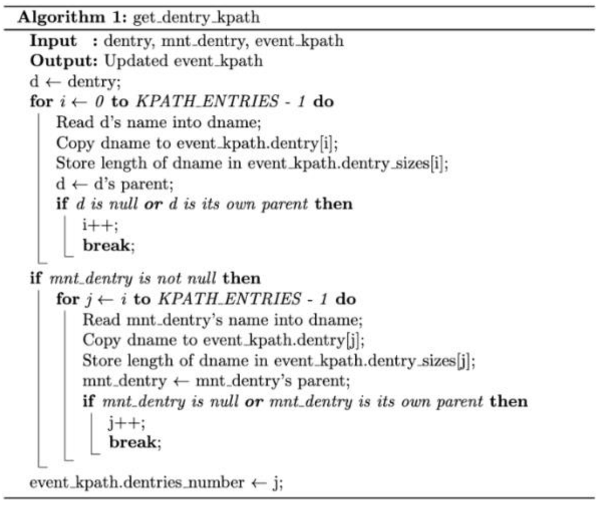
contains all the information the kernel needs to manage and schedule processes, including information about process state, scheduling, memory management, file descriptors, and more. The task_struct in our case is used to extract parent pid, process paths dentry, current working directory dentry and mount point paths dentry.

- **BPF_CORE_READ_INTO**: It is an eBPF helper function that safely reads kernel memory into a user-provided buffer, handling potential structure changes across different kernel versions and ensuring compatibility and security in eBPF programs. [8]

B. Implementation

This section will capture the various implementations that were used to capture the different data points.

- **Extracting current working directory and path:** Two key functions are used to retrieve file paths in the Linux kernel:
  - **get_dentry_kpath**: This function constructs a file path by traversing the directory structure from a given dentry upwards. It iterates through the dentry chain, reading each directory name and storing it in the event_kpath structure. If a mount point dentry is provided, it continues the traversal from this point, allowing the function to cross filesystem mount boundaries. The process stops when it reaches the root directory or when it has filled the maximum number of entries (KPATH_ENTRIES). This bottom-up approach builds the full path.



```
Algorithm 1: get_dentry_kpath
  Input  : dentry, mnt_dentry, event_kpath
  Output: Updated event_kpath
  d ← dentry;
  for i ← 0 to KPATH_ENTRIES - 1 do
      Read d's name into dname;
      Copy dname to event_kpath.dentry[i];
      Store length of dname in event_kpath.dentry_sizes[i];
      d ← d's parent;
      if d is null or d is its own parent then
          i++;
          break;
  if mnt_dentry is not null then
      for j ← i to KPATH_ENTRIES - 1 do
          Read mnt_dentry's name into dname;
          Copy dname to event_kpath.dentry[j];
          Store length of dname in event_kpath.dentry_sizes[j];
          mnt_dentry ← mnt_dentry's parent;
          if mnt_dentry is null or mnt_dentry is its own parent then
              j++;
              break;
  event_kpath.dentries_number ← j;
```

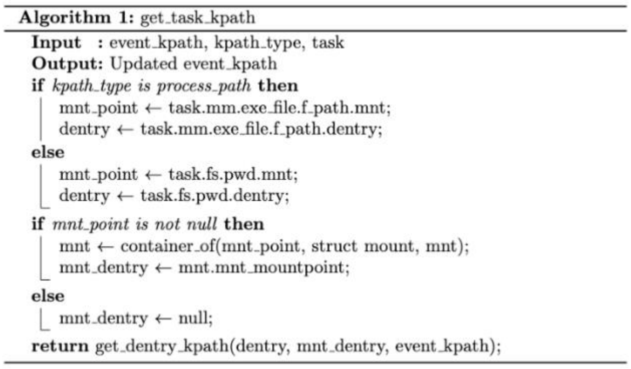
Algorithm 1: get_dentry_kpath

Input: *dentry, mnt_dentry, event_kpath*
Output: *Updated event_kpath*

```
d = dentry;
for i : 0 to KPATH_ENTRIES – 1; do
 dname = d's name;
 event_kpath.dentry[i] = dname;
 event_kpath.dentry_sizes[i] = len(dname);
 d = d's parent;
 if d is null or d is its own parent:
   i++;
   break;

if mnt_dentry is not null:
 for j : i to KPATH_ENTRIES – 1; do
   dname = mnt_dentry's name;
   event_kpath.dentry[j] = dname;
   event_kpath.dentry_sizes[j] = len(dname);
   mnt_dentry = mnt_dentry's parent
   if mnt_dentry is null or mnt_dentry is its own parent:
    j++;
    break;
```

  - **get_task_kpath**: This function retrieves file paths within the Linux kernel to populate an event structure with enriched information. It focuses on two key fields: kpath_binary (the executable file path) and kpath_cwd (the current working directory path) of a given process (task). The function determines which path to retrieve based on input parameters. For kpath_binary, it accesses the task's memory management structure (mm) and its exe_file field. For kpath_cwd, it uses the task's filesystem information structure (fs) and its pwd field. Both structures contain a dentry (directory entry) and a vfsmount (mount point information), which the function extracts. If a mount point is found, it uses the container_of macro to get the full mount structure for handling files on different mount points. Finally, it calls get_dentry_kpath to construct the full path by traversing the directory structure upwards. This approach allows get_task_kpath to handle the complexities of the Linux filesystem, including crossing mount boundaries, and provides crucial context about the process's binary and working directory for event analysis.



```
Algorithm 1: get_task_kpath
  Input  : event_kpath, kpath_type, task
  Output: Updated event_kpath
  if kpath_type is process_path then
      mnt_point ← task.mm.exe_file.f_path.mnt;
      dentry ← task.mm.exe_file.f_path.dentry;
  else
      mnt_point ← task.fs.pwd.mnt;
      dentry ← task.fs.pwd.dentry;
  if mnt_point is not null then
      mnt ← container_of(mnt_point, struct mount, mnt);
      mnt_dentry ← mnt.mnt_mountpoint;
  else
      mnt_dentry ← null;
  return get_dentry_kpath(dentry, mnt_dentry, event_kpath);
```

Algorithm 2: get_task_kpath

Input: *event_kpath, kpath_type, task*
Output: *Updated event_kpath*

```
if kpath_type == process_path:
  mnt_point = task.mm.exe_file.f_path.mnt;
  dentry = task.mm.exe_file.f_path.dentry;
else:
  mnt_point = task.fs.pwd.mnt;
  dentry = task.fs.pwd.dentry;

if mnt_point is not null:
  mnt = container_of(mnt_point, struct_mount, mnt);
  mnt_dentry = mnt.mnt_mountpoint;
else:
  mnt_dentry = null;
```

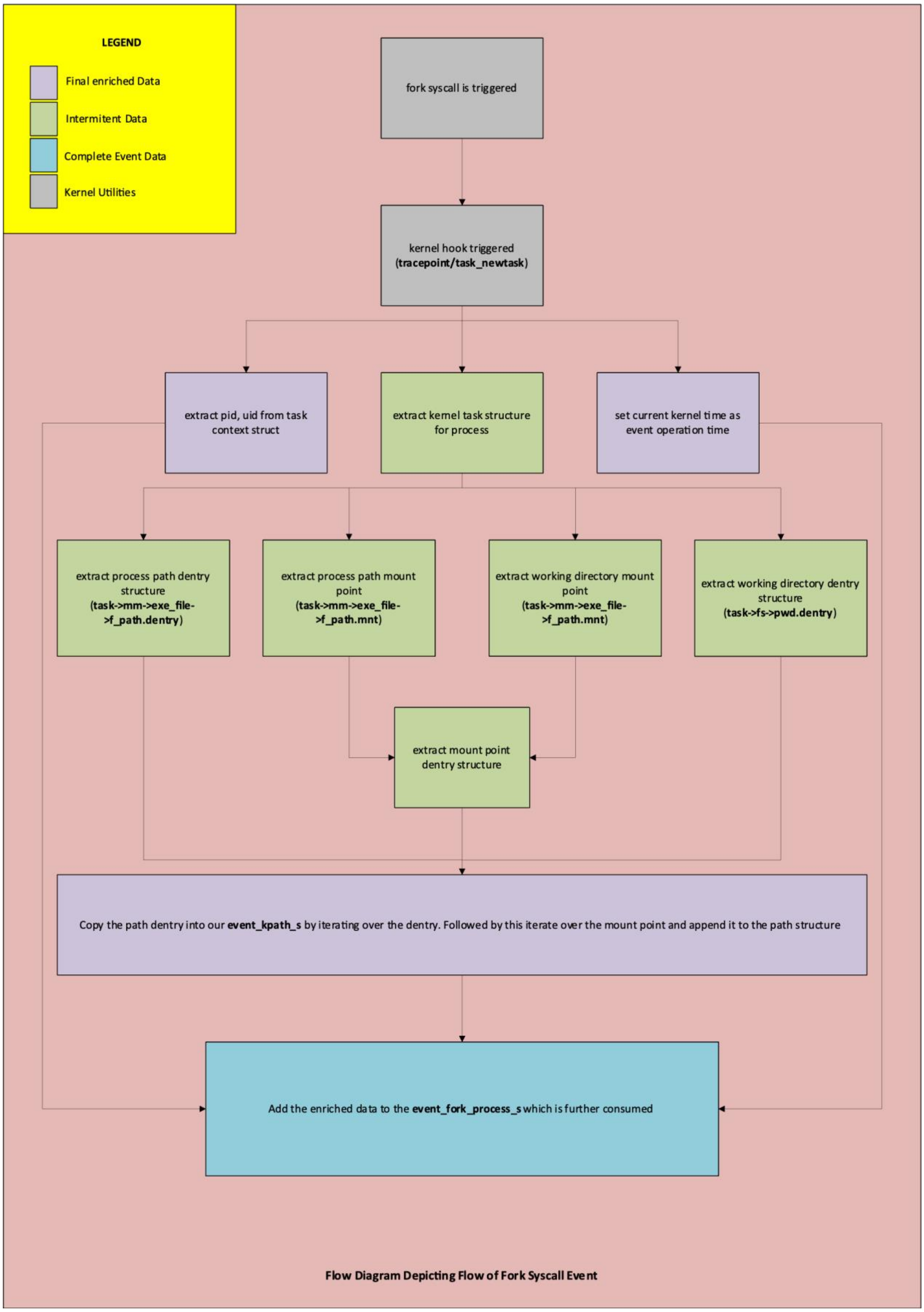
- **Extracting command line argument**: We use BPF_CORE_READ_INTO to safely access the command line from the syscall context (ctx). This API allows eBPF programs to read kernel memory safely. The command line provides the actual command used to start the process, giving insight into how the process was initiated.
- **Extracting start time**: Again, using BPF_CORE_READ_INTO, the function reads the start_boottime field from the task structure. This structure is the kernel's representation of a process. The start time is relative to system boot, offering a precise timestamp of when the process began executing.
- **Extracting parent PID**: This involves a two-step process:
  - The function first uses BPF_CORE_READ_INTO to access the real_parent field of the current task structure. This field points to the parent process's task structure.
  - Then, it extracts the parent's PID from this structure.
- **Extracting pid**: The PID is directly accessed from the syscall context (ctx) structure, providing the unique identifier of the current process executing the system call.

  We have captured the hooks and structure used to extract the corresponding parameters:
  Exec event:

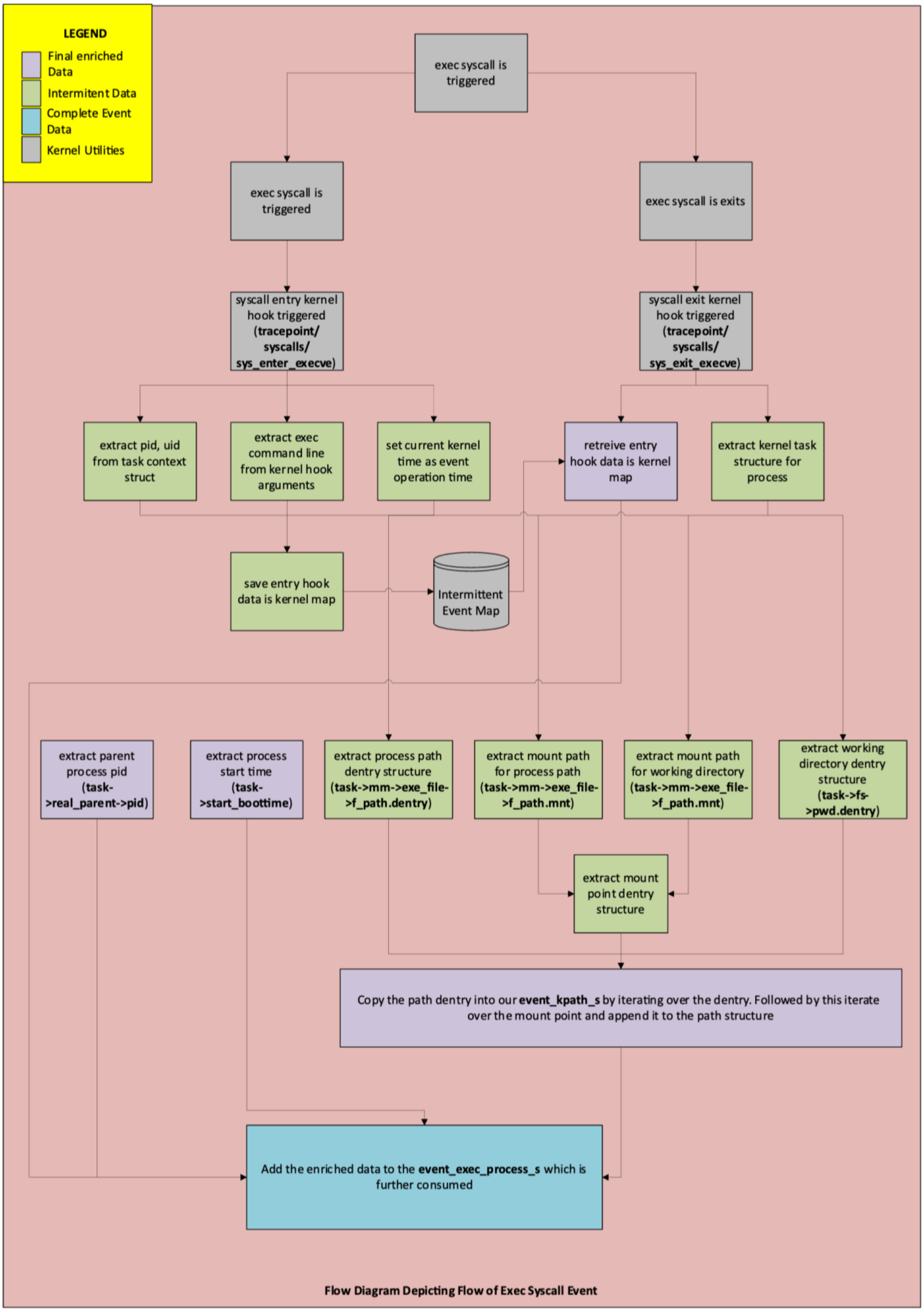
Parameter	Read from which hook (entry/ exit)	Read from which structure
pid	sys_enter_execve	bpf_get_current _pid
ppid	sys_exit_execve	Task structure
Start time	sys_exit_execve	Task structure
cwd	sys_exit_execve	Dentry structure
path	sys_exit_execve	Dentry structure
Cmdline args	sys_enter_execve	syscall arguments

Fork event:

Parameter	Read from which hook (entry/ exit)	Read from which structure
pid	tracepoint/task_newtask	bpf_get_current_pid _tgid
ppid	tracepoint/task_newtask	Task structure
Start time	tracepoint/task_newtask	Task structure
cwd	tracepoint/task_newtask	Dentry structure
path	tracepoint/task_newtask	Dentry structure


LEGEND
Final enriched Data
Intermitent Data
Complete Event Data
Kernel Utilities
fork syscall is triggered
kernel hook triggered (tracepoint/task_newtask)
extract pid, uid from task context struct
extract kernel task structure for process
set current kernel time as event operation time
extract process path dentry structure (task->mm->exe_file->f_path.dentry)
extract process path mount point (task->mm->exe_file->f_path.mnt)
extract working directory mount point (task->mm->exe_file->f_path.mnt)
extract working directory dentry structure (task->fs->pwd.dentry)
extract mount point dentry structure
Copy the path dentry into our event_kpath_s by iterating over the dentry. Followed by this iterate over the mount point and append it to the path structure
Add the enriched data to the event_fork_process_s which is further consumed

Flow Diagram Depicting Flow of Fork Syscall Event


LEGEND
Final enriched Data
Intermitent Data
Complete Event Data
Kernel Utilities
exec syscall is triggered
exec syscall is triggered
exec syscall is exits
syscall entry kernel hook triggered (tracepoint/ syscalls/ sys_enter_execve)
syscall exit kernel hook triggered (tracepoint/ syscalls/ sys_exit_execve)
extract pid, uid from task context struct
extract exec command line from kernel hook arguments
set current kernel time as event operation time
retreive entry hook data is kernel map
extract kernel task structure for process
save entry hook data is kernel map
Intermittent Event Map
extract parent process pid (task->real_parent->pid)
extract process start time (task->start_boottime)
extract process path dentry structure (task->mm->exe_file->f_path.dentry)
extract mount path for process path (task->mm->exe_file->f_path.mnt)
extract mount path for working directory (task->mm->exe_file->f_path.mnt)
extract working directory dentry structure (task->fs->pwd.dentry)
extract mount point dentry structure
Copy the path dentry into our event_kpath_s by iterating over the dentry. Followed by this iterate over the mount point and append it to the path structure
Add the enriched data to the event_exec_process_s which is further consumed

Flow Diagram Depicting Flow of Exec Syscall Event

IV. EVALUATION / RESULTS

A. Benchmarking

We ran experiments that generated a lot of short-lived processes in the system and our aim was to quantify and compare the completeness and data correctness of the per-process information. The following commands were run in random order using a shell script with randomized time constraints. Each command was run 10000 times.[5]

1) *ls -lrth*
2) *uname -ar*
3) *cat /etc/hostname*
4) *chmod +x test.sh*
5) *./test.sh*

test.sh is a script present in the file system and it aims to print date and locate some secret file named "secret" in the system. The script as follows:

#! /bin/bash
date
locate secret

The output and accuracy of the experiments was determined by looking at the data received from the program. For our experiments and validation of data, we looked at the process path and command line args (args/ cmd args) of the events received.

As per the commands run the expected data is as follows:

command	**Process path**	**Cmd args**
ls -lrth	/usr/bin/ls	-lrth
Unam -ar	/usr/bin/uname	-ar
Cat /etc/hostname	/usr/bin/cat	/etc/hostname
Chmod +x test.sh	/usr/bin/chmod	+x test.sh
./test.sh	/usr/bin/bash	./test.sh

We ran these experiments in the presence of two ebpf programs in the system under various scenarios. The programs were as follows:

- The program1 is our proof-of-concept program where we enrich our data from the source itself. In this, hooks are attached to tracepoint/task/task_newtask, tracepoint/syscalls/sys_enter_execve and tracepoint/syscalls/sys_exit_execve. The events are enriched in the kernel space itself as per our approach.
- The program2 is the traditional approach where we attach hooks in kernel to extract only pids of the process. In this to extract the fork events in are attached to tracepoint/task/task_newtask. Ffor the exec events the hooks are attached to tracepoint tracepoint/syscalls/sys_enter_execve and tracepoint/syscalls/sys_exit_execve . The hooks are responsible for collecting the pids of fork and execve events and then sending these to user space. Once the event has reached the user space, we try to enrich our data by reading the process path and command line arguments for each event received. The data is enriched by reading the proc file system.

All the experiments were done on an Ubuntu 22 machine with 32GB RAM and 8 cores. Experiments were conducted under different scenarios as below

For each scenario, for any program we received can categorize the output in three categories, a) where complete data was received, b) where only process path was received, c) no data was received except pid.

Scenario1
On idle machine, each of commands were run 10000 times in sequential manner. The machine was in an idle state with no extra workload running. The generation of events were triggered from a shell script within random time intervals.
The results were as follows

- For program1, we received all the events completely enriched thus attaining 100% accuracy.
- For program2 the data is as follows

Command	Complete data	Only path	No data
Ls -lrth	9930	24	46
Uname -r	9763	7	230
Cat /etc/hostname	9291	23	686
Chmod +x test.sh	8761	22	1217
./test.sh	9730	0	270
Percentage	95.0%	0.2%	4.9%

Scenario2
On idle machine as similar in scenario 1, each of commands were run 10000 times parallel. The machine was in an idle state with no extra workload running. The generation of events were triggered from a shell script within random time intervals.

The results were as follows

- For program1, we received all the events completely enriched thus attaining 100% accuracy.
- For program2 the data is as follows

Command	Complete data	Only path	No data
Ls -lrth	8466	2	1532
Uname -r	7468	16	2516
Cat /etc/hostname	7671	20	2309
Chmod +x test.sh	6179	25	3796
./test.sh	8857	5	1138
Percentage	77.3%	0.1%	22.6%

Scenario3
A postgresql workload is running on the server. The workload mimics an on-field linux postgresql sever profile with high frequency of database operations. Each of commands were

run 10000 times parallel. The generation of events were triggered from a shell script within random time intervals.
The results were as follows

- For program1, we received all the events completely enriched thus attaining 100% accuracy.
- For program2 the data is as follows

Command	Complete data	Only path	No data
Ls -lrth	7970	20	2010
Uname -r	6650	36	3314
Cat /etc/hostname	5769	45	4186
Chmod +x test.sh	5960	39	4001
./test.sh	9397	117	486
Percentage	71.5%	0.5%	28.0%

Scenario4
A openssl compilation workload is running on the server. The workload continuously compiles openssl on multiple threads. Each of commands were run 10000 times parallelly. The generation of events was triggered from a shell script within random time intervals.
The results were as follows

- For program1, we received all the events completely enriched thus attaining 100% accuracy.
- For program2 the data is as follows

Command	Complete data	Only path	No data
Ls -lrth	8894	20	1086
Uname -r	8006	45	1949
Cat /etc/hostname	7733	36	2231
Chmod +x test.sh	6747	49	3204
./test.sh	9688	140	172
Percentage	82.1%	0.6%	17.3%

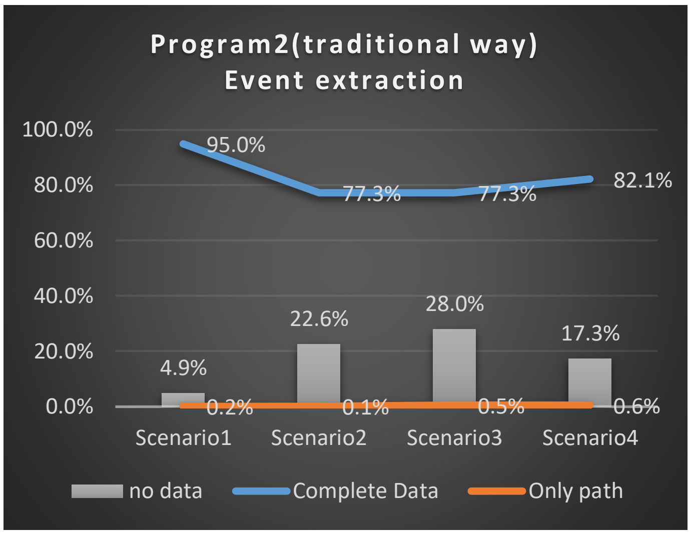
Combined results for all scenarios

Program1: our proof of concept (source enriched ebpf events)

	Complete Data	Only path	no data
Scenario1	100.0%	0%	0%
Scenario2	100.0%	0%	0%
Scenario3	100.0%	0%	0%
Scenario4	100.0%	0%	0%
Average	100.0%	0%	0%

Program2 (Traditional method) results for event extraction

	Complete Data	Only path	no data
Scenario1	95.0%	0.2%	4.9%
Scenario2	77.3%	0.1%	22.6%
Scenario3	77.3%	0.5%	28.0%
Scenario4	82.1%	0.6%	17.3%
Average	82.9%	0.3%	18.2%



V. Limitations

The eBPF and BTF approach, while highly effective for extracting process information, does have some limitations. One major limitation is the complexity involved in writing and maintaining eBPF programs. These programs require a deep understanding of kernel internals and the eBPF instruction set, which can be a steep learning curve for developers. Additionally, eBPF programs must be thoroughly vetted by the eBPF verifier to ensure they do not introduce security vulnerabilities or cause kernel crashes. This vetting process can be time-consuming and may result in the rejection of valid programs if they are not carefully crafted.. Also, older kernels < 4.18, BTF is not supported, hence such approach cannot work for lower kernel versions. eBPF programs may also result in performance overhead involving large data transfers. Transferring large amounts of data from eBPF to user space can lead to increased CPU utilization and memory pressure. As the volume of data transferred increases, it can create a bottleneck in the kernel-to-user space communication channel.

VI. Conclusion

In this paper, we have addressed the challenge of extracting comprehensive process information on Linux systems, particularly for short-lived processes, to aid in the proper functioning of Endpoint Detection and Response (EDR) solutions. Traditional methods relying on the /proc filesystem have proven unreliable for short-lived processes, as the data is erased as soon as the corresponding process exits, leading to incomplete process timelines and potential visibility gaps. To overcome this limitation, we proposed a solution leveraging extended Berkeley Packet Filter (eBPF) and BPF Type Format (BTF). By dynamically attaching eBPF programs to system calls such as fork, exec, and exit, we can extract all the required process information directly from kernel hooks. This approach ensures that the information is

captured within the kernel itself, providing a more reliable and complete dataset for EDR solutions.
Our proof-of-concept implementation demonstrated that eBPF-based process data extraction significantly improves the reliability of process timelines, achieving up to 100% reliability for short-lived processes compared to the 83% reliability of procfs-based methods. This enhanced visibility is crucial for effective threat detection and response, ultimately improving the overall security posture of the organization.
In conclusion, the eBPF and BTF approach offers a robust and efficient solution for extracting comprehensive process information on Linux systems, addressing the shortcomings of traditional methods and enhancing the capabilities of EDR solutions.

Acknowledgments

We would like to express our sincere gratitude to our advisor Srinivas Koripella for his continuous support and guidance. His insightful feedback and encouragement have been invaluable.

References

[1] LLC, M.: eBPF Documentation. https://ebpf.io/what-is-ebpf/ Accessed 2023-06- 19

[2] Kernel Documentation for procfs. https://www.kernel.org/doc/html/latest/filesystems/proc.html

[3] Prevalent EDR solutions use /proc fs : https://redcanary.com/blog/threat-detection/process-stream

[4] Linux Auditd. Configuring and auditing Linux systems with Audit daemon (linux-audit.com)

[5] The 40 most used Linux Commands: https://kinsta.com/blog/linux-commands/

[6] https://natanyellin.com/posts/tracking-running-processes-on-linux/

[7] Rice, Liz. Learning eBPF. " O'Reilly Media, Inc.", 2023. (https://cilium.isovalent.com/hubfs/Learning-eBPF%20-%20Full%20book.pdf)

[8] Andrii Nakryiko's Blog (https://nakryiko.com/posts/bpf-core-reference-guide/#bpf-core-read-into)

[9] eBPF: https://developers.redhat.com/articles/2023/10/19/ebpf-application-development-beyond-basics

[10] EDR using ML: https://www.researchgate.net/figure/Intrusion-Detection-Systems_tbl1_356742987