

Demonstration of the usage of eBPF by a Keylogger

J Component Project

Aarush Bhat 19BCE0564
Jyotir Aditya 19BCE0846
Slot: F1
Submitted To: Prof. Aju D
B.Tech. Computer Science and Engineering



School of Computer Science and Engineering
Vellore Institute of Technology
Vellore

29 November 2021

Abstract

Motivation:

eBPF is a revolutionary technology in the Linux kernel that can run sandboxed programs in an operating system kernel. Making a non-intrusive monitoring tool like a KeyLogger seems like a must-have thing in the security space.

When you google keylogger, the first thing that comes up is an article titled, “Keyloggers: How they work and how to detect them”, building a non-intrusive, kernel-level keylogger using The Power of Programmability from eBPF motivates us to do the project.

Quoting from ebpf.io: “Historically, the operating system has always been an ideal place to implement observability, security, and networking functionality due to the kernel’s privileged ability to oversee and control the entire system. At the same time, an operating system kernel is hard to evolve due to its central role and high requirement towards stability and security.”

Aim:

Explore the possibilities of eBPF technology and its power of programmability and create a non-intrusive KeyLogger to prevent any malicious activity on an OS independent system.

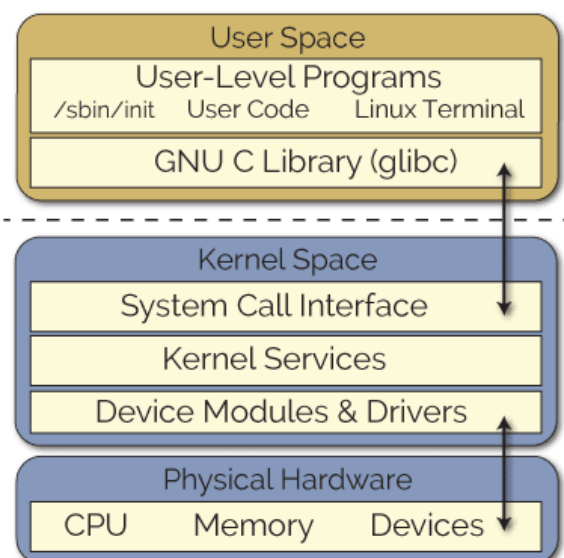
Objective:

- To explore the possibilities of eBPF.
- To learn more about a way to make the kernel programmable.
- To explore a new realm of Security, Observability & Monitoring
- To build a non-intrusive monitoring Keylogger sandboxed in OS kernel.

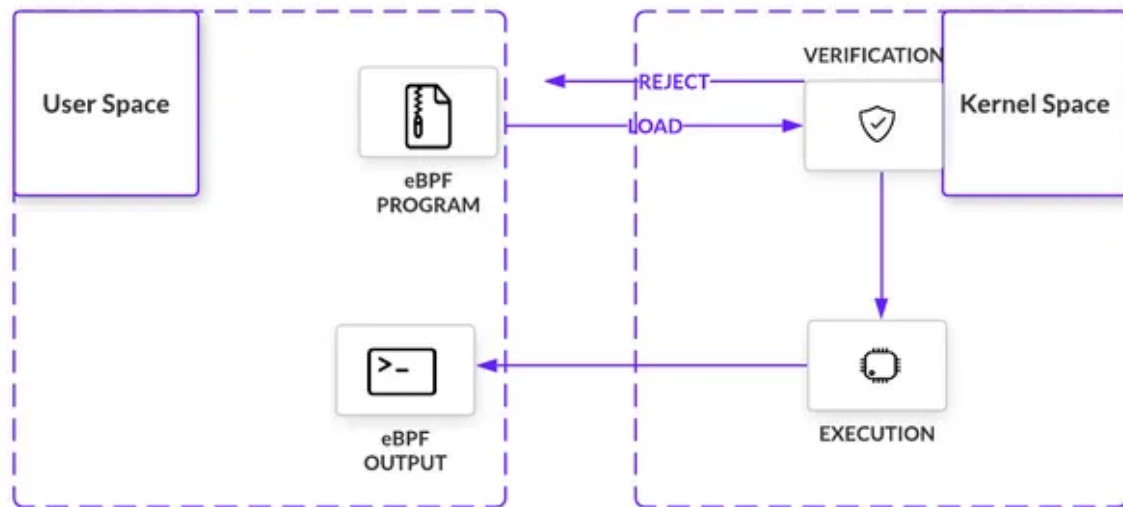
Methodology:

Non-intrusive measurement refers to devices or measurement procedures that induce minimal impact on the person involved.

- eBPF is a mechanism for Linux applications to execute code in Linux kernel space.
- eBPF has already been used to create networking, debugging, tracing, firewalls, and more programs.
- eBPF can run sandboxed programs in the Linux kernel without changing kernel source code or loading kernel modules.



- Several complex components are involved in the functioning of eBPF programs and their execution.



Expected Outcome:

- To explore the possibilities of eBPF.
- To learn more about a way to make the kernel programmable.
- To explore a new realm of Security, Observability & Monitoring
- To build a non-intrusive monitoring Keylogger sandboxed in the OS kernel.

Keywords

Linux kernel, eBPF, bcc, Observability & Monitoring, Security, Keylogger

Literature Survey / Related Works

Name	Citation	Summary
The rise of eBPF for non-intrusive performance monitoring	C. Cassagnes, L. Trestioreanu, C. Joly and R. State, "The rise of eBPF for non-intrusive performance monitoring," NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1-7, DOI: 10.1109/NOMS47738.2020.9110434.	This paper explains that container engines are strengthening their isolation mechanisms and how non-intrusive monitoring becomes a must-have for the performance analysis of containerised user-space applications in production environments. Profiling and tracing of several Interledger connectors is carried out using two full-fledged implementations of the Interledger protocol specifications
eBPF-based Content and Computation-aware Communication for Real-time Edge Computing	Sabur Baidya ¹ , Yan Chen ² and Marco Levorato ¹ ¹ Donald Bren School of Information and Computer Science, UC Irvine e-mail: {sbaidya, levorato}@uci.edu ² America Software Laboratory, Huawei, e-mail: y.chen@huawei.com	This paper proposed framework is instantiated to address a case-study scenario where video streams from multiple cameras are transmitted to the edge processor for real-time analysis. Numerical results demonstrate the advantage of the proposed framework in terms of programmability, network bandwidth and system resource savings.
eBPF - From a Programmer's Perspective	Niclas Hedam IT University of Copenhagen nhed@itu.dk	This paper shows how to write, compile and load eBPF programs. Furthermore, the different types of eBPF programs are presented. Differences between user-space C program and eBPF C program is included. Use cases of eBPF, including a DDoS firewall and file system extensions, are also discussed.

Survey of keylogger Technologies	Yahye Abukar Ahmed, Mohd Aizaini Maarof, Fuad Mire Hassan and Mohamed Muse Abshir Emails: yahy@simad.edu.so, aizaini@utm.my, fuaadmire@gmail.com and inaboqormuse@gmail.com Department of Computer Science, Faculty of Computer Science & Information Systems Universiti Teknologi Malaysia	This paper presents an overview of keylogger programs, types, characteristics of keyloggers, and their methodology. Finally, it analyses the current detection techniques and explores several proactive methods. A case study on Blackberry is used as a real-time example in this paper.
Investigating the keylogging threat in android — A user perspective	F. Mohsen, E. Bello-Ogunu and M. Shehab, "Investigating the keylogging threat in android — User perspective (Regular research paper)," 2016 Second International Conference on Mobile and Secure Services (MobiSecServ), 2016, pp. 1-5, DOI: 10.1109/MOBISECSERV.2016.7440223.	This paper discusses users' and keyboard developers' roles in increasing/decreasing the chance of successful keylogger attacks. An Android app was developed, KBsChecker, and asked participants to install it on their devices. The app collects data from participants' devices and prompts them to complete a survey
Keyloggers: silent cyber security weapons	Dr Akashdeep Bhardwaj, School of Computer Science, University of Petroleum & Energy Studies, Dehradun, India and Dr Sam Goundar, University of South Pacific, Suva, Fiji	In this research, the authors demonstrated a successful keylogger technique, gathering keystrokes and screenshots and online transactions without any scanner detecting the activities. Using random spacing, the proposed layout for virtual keyboards involves randomly exchanging vertically adjacent keys from the existing QWERTY layout.
Combining System	Luca Deri ^{1,2} , Samuele	This paper describes the

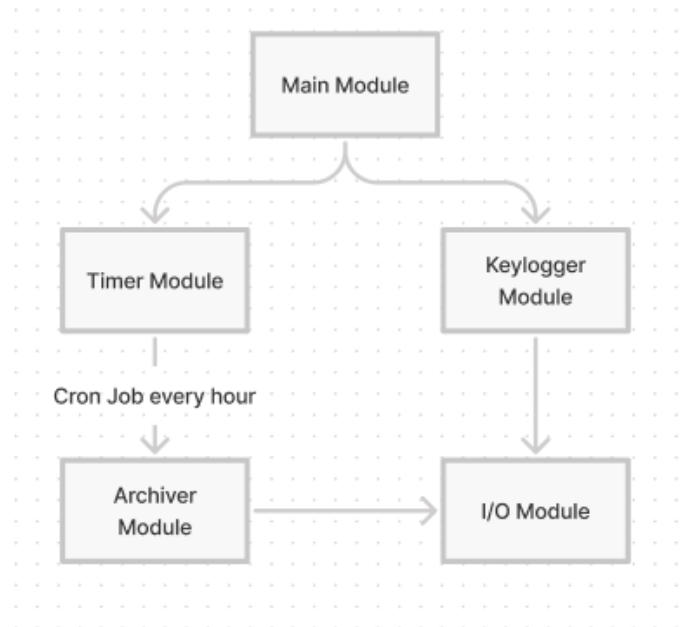
Visibility and Security Using eBPF.	Sabella ¹ and Simone Mainardi ² ¹ IIT/CNR, Via Moruzzi 1, Pisa, Italy ² ntop, Via Ponte a Piglieri 8, 56122 Pisa, Italy luca.der ¹ @iit.cnr.it, {der ¹ ,sabella ¹ ,mainardi ² }@ntop.org	design and the implementation of a novel tool that enables communications visibility at the operating-system level by monitoring network activities.
Accelerating Linux Security with eBPF iptables	Matteo Bertrone, Sebastiano Miano, Fulvio Risso, Massimo Tumolo Department of Control and Computer Engineering, Politecnico di Torino, Italy	This paper evaluated bpf-iptables by attaching it to the host interfaces' TC ingress and egress hook and compared its performance against iptables in two different cases. In the first test, an increasing number of rules to the FORWARD chain of the firewall, and we generated a unidirectional stream of 64B UDP packets, is added.
A Framework for eBPF-Based Network Functions in an Era of Microservices	S. Miano, F. Risso, M. V. Bernal, M. Bertrone and Y. Lu, "A Framework for eBPF-Based Network Functions in an Era of Microservices," in IEEE Transactions on Network and Service Management, vol. 18, no. 1, pp. 133-151, March 2021, DOI: 10.1109/TNSM.2021.3055676.	Polycube network functions, called Cubes, can be dynamically generated and injected into the kernel networking stack without requiring custom kernels or specific kernel modules, simplifying the debugging and introspection, which are two fundamental properties in recent cloud environments. This paper validates the framework by showing significant improvements over existing applications and proves the generality of the Polycube programming model through the implementation of complex use cases such as a network provider for Kubernetes.

<p>bpfbox: Simple Precise Process Confinement with eBPF</p>	<p>William Findlay Carleton University, Ottawa, ON, Canada</p> <p>Anil Buntwal Somayaji Carleton University, Ottawa, ON, Canada</p> <p>David Barrera Carleton University, Ottawa, ON, Canada</p>	<p>This Paper argues that simple, efficient, and flexible confinement can be better implemented today using eBPF, an emerging technology for safely extending the Linux kernel. It also presents a proof-of-concept confinement application, bpfbox, that uses less than 2000 lines of kernel space code and allows for confinement at the userspace function, system call, LSM hook, and kernel space function boundaries---something that no existing process confinement mechanism can do.</p>
---	--	---

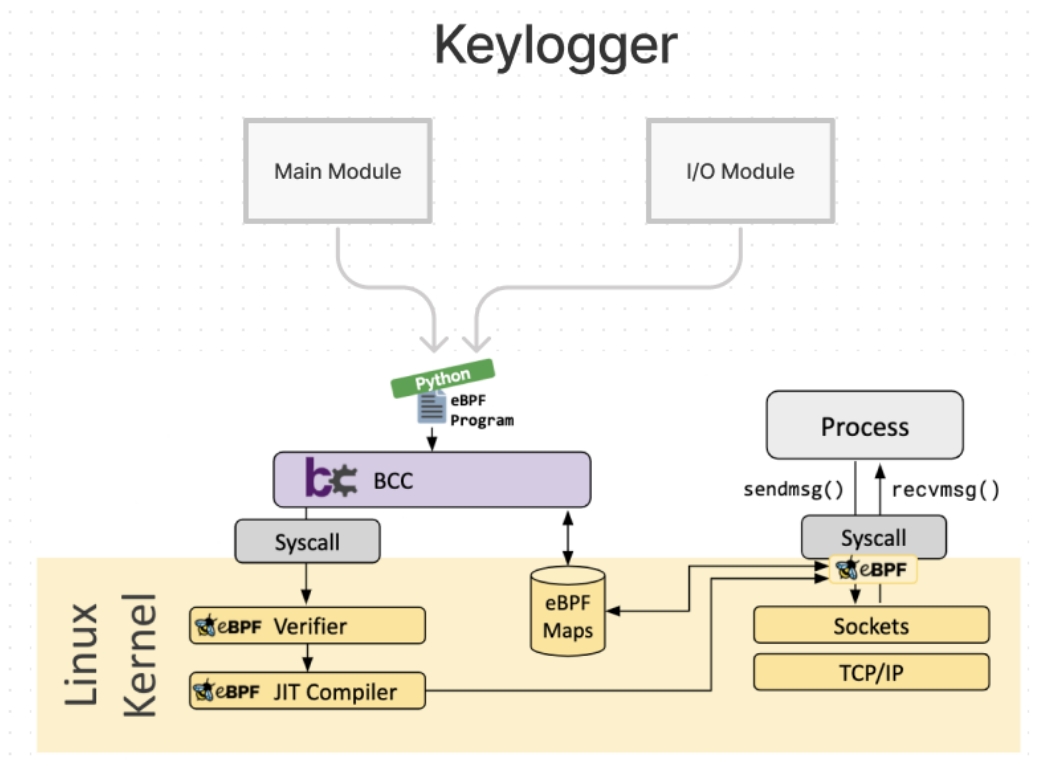
Overall Architecture

The project is divided into five modules, namely:

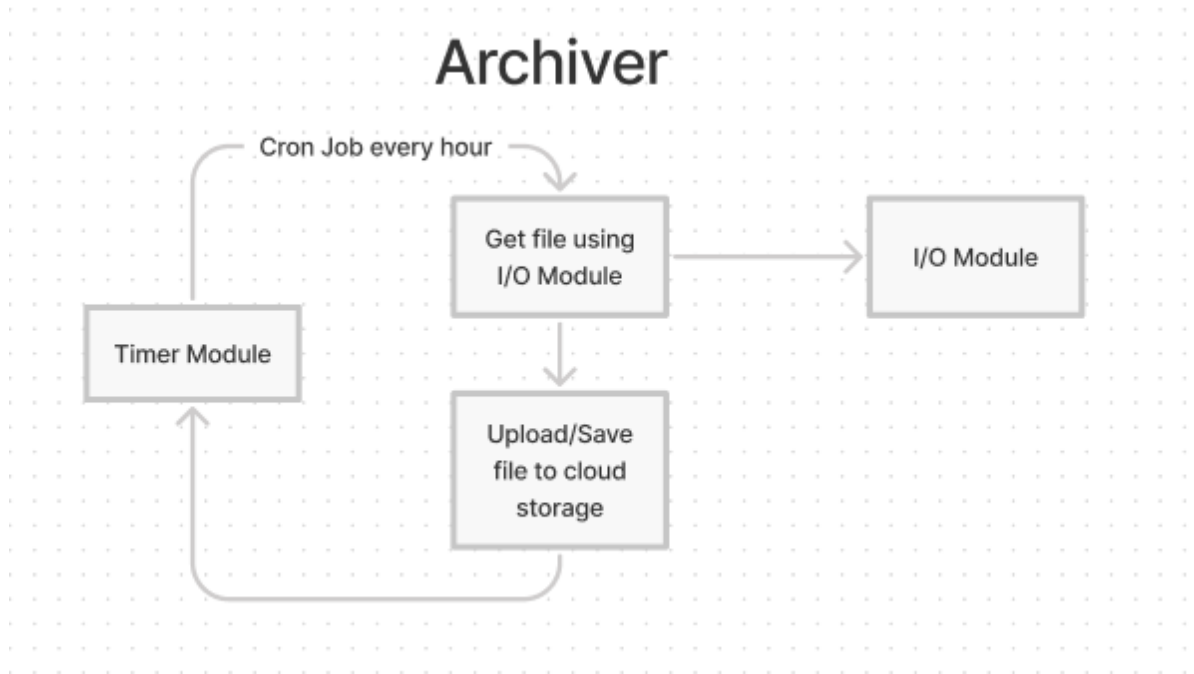
- **Main Module:** The main module is the module that holds the basic CLI and parses the instructions given by the user.



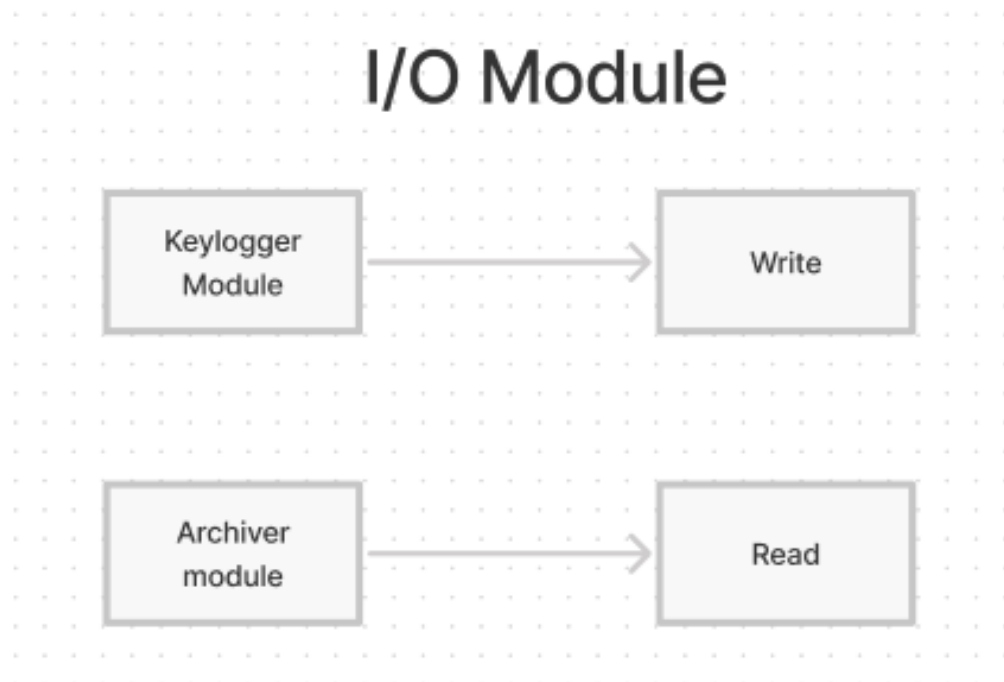
- **Keylogger Module:** Keylogger module is the core eBPF code for the keylogger to listen to keystrokes from the kernel and talk to the I/O Module to write to a file.



- **Timer Module:** Timer Module is a cron job used to run the Archiver Module at certain time intervals.
- **Archiver Module:** Archiver Module uses the I/O Module to read from the written file system and then save it in a cloud storage platform.



- **I/O Module:** I/O Module uses `bpf_trace_printk` to print and store the data in a new file and read or delete it when the Archiver Module uses it.

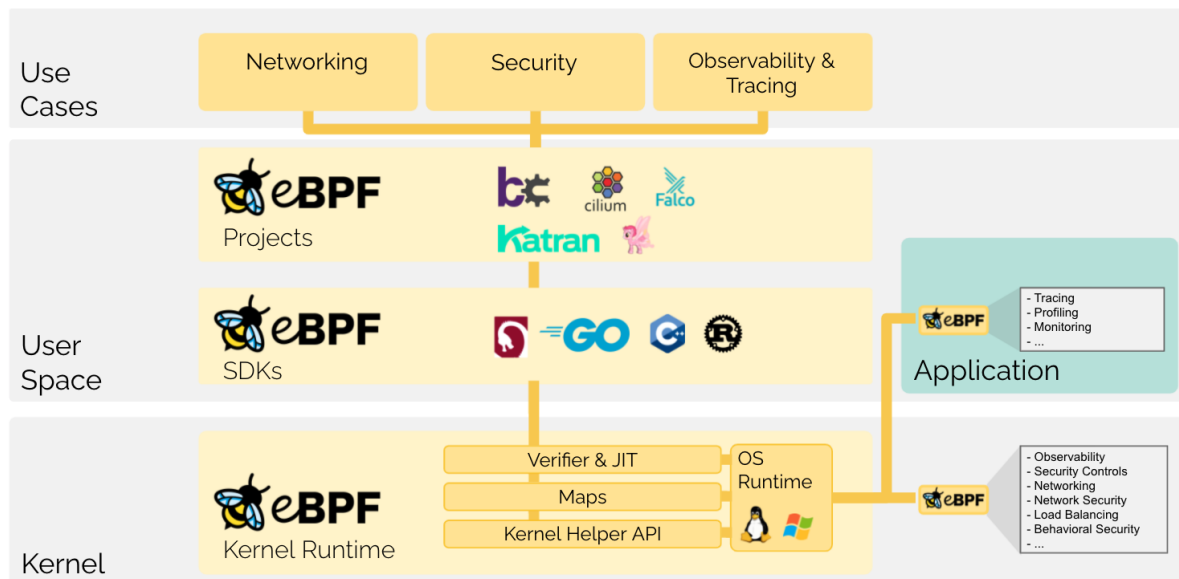


Proposed Methodology:

The proposed methodology is to make a kernel-level keylogger using eBPF.

eBPF:

eBPF is a revolutionary technology in the Linux kernel that can run sandboxed programs in an operating system kernel. It safely and efficiently extends the kernel's capabilities without changing kernel source code or loading kernel modules.



Today, eBPF is used extensively to drive a wide variety of use cases: Providing high-performance networking and load-balancing in modern data centres and cloud-native environments, extracting fine-grained security observability data at low overhead, helping application developers trace applications, providing insights for performance troubleshooting, preventive application and container runtime security enforcement, and much more. The possibilities are endless, and the innovation that eBPF is unlocked has only just begun.

The Power of Programmability:

The main feature eBPF brings to the table is The Power of Programmability.

Let's start with an analogy. On old websites like GeoCities, pages used to be almost exclusively written in a static markup language (HTML). A web page was basically a document with an application (browser) able to display it. Looking at web pages today, web pages have become full-blown applications, and web-based technology has replaced a vast majority of applications written in languages requiring compilation. What enabled this evolution?



The short answer is programmability with the introduction of JavaScript. It unlocked a massive revolution resulting in browsers evolving into almost independent operating systems.

Why did this evolution happen? Programmers were no longer as bound to users running particular browser versions. Instead of convincing standards bodies that a new HTML tag was needed, the availability of the necessary building blocks decoupled the pace of innovation of the underlying browser from the application running on top. This is, of course, a bit oversimplified as HTML did evolve and contribute to the success, but the evolution of HTML itself would not have been sufficient.

Before taking this example and applying it to eBPF, let's look at a couple of critical aspects that were vital in the introduction of JavaScript:

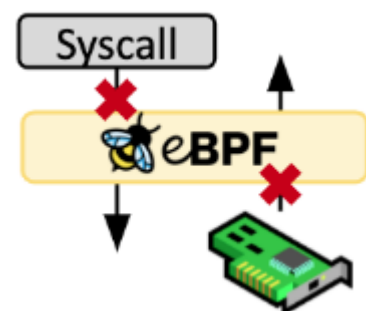
- Safety: Untrusted code runs in the browser of the user. This was solved by sandboxing JavaScript programs and abstracting access to browser data.
- Continuous Delivery: Evolution of program logic must be possible without constantly shipping new browser versions. This was solved by providing the right low-level building blocks sufficient to build arbitrary logic.
- Performance: Programmability must be provided with minimal overhead. This was solved with the introduction of a Just-in-Time (JIT) compiler.

For all of the above, exact counterparts can be found in eBPF for the same reason.

Uses of eBPF:

1. Security:

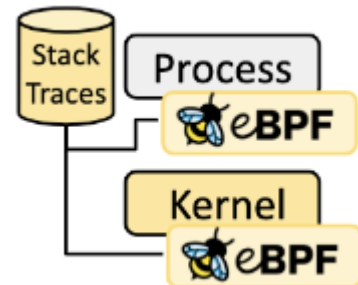
Building on the foundation of seeing and understanding all system calls and combining that with a packet and socket-level view of all networking operations allows for revolutionary new approaches to securing systems. While aspects of system call filtering, network-level filtering, and process context tracing have typically been handled by entirely independent systems, eBPF allows for combining the



visibility and control of all aspects to create security systems operating in more context with a better level of control.

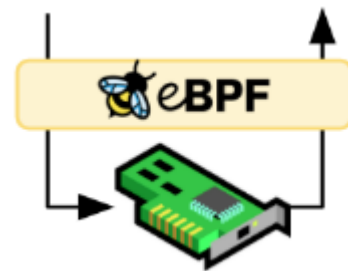
2. Tracing & Profiling:

The ability to attach eBPF programs to tracepoints as well as kernel and user application probe points allows unprecedented visibility into the runtime behaviour of applications and the system itself. By giving introspection abilities to both the application and system side, both views can be combined, allowing powerful and unique insights to troubleshoot system performance problems. Advanced statistical data structures allow extracting meaningful visibility data in an efficient manner, without requiring the export of vast amounts of sampling data as typically done by similar systems.



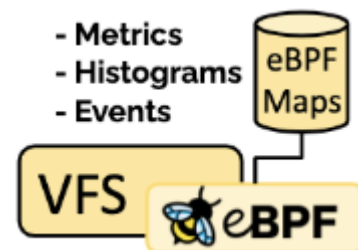
3. Networking:

The combination of programmability and efficiency makes eBPF a natural fit for all packet processing requirements of networking solutions. The programmability of eBPF enables adding additional protocol parsers and easily programming any forwarding logic to meet changing requirements without ever leaving the packet processing context of the Linux kernel. The efficiency provided by the JIT compiler provides execution performance close to that of natively compiled in-kernel code.



4. Observability & Monitoring:

Instead of relying on static counters and gauges exposed by the operating system, eBPF enables the collection & in-kernel aggregation of custom metrics and generation of visibility events based on a wide range of possible sources. This extends the depth of visibility that can be achieved as well as reduces the overall system overhead significantly by only collecting the visibility data required and by generating histograms and similar data structures at the source of the event instead of relying on the export of samples.

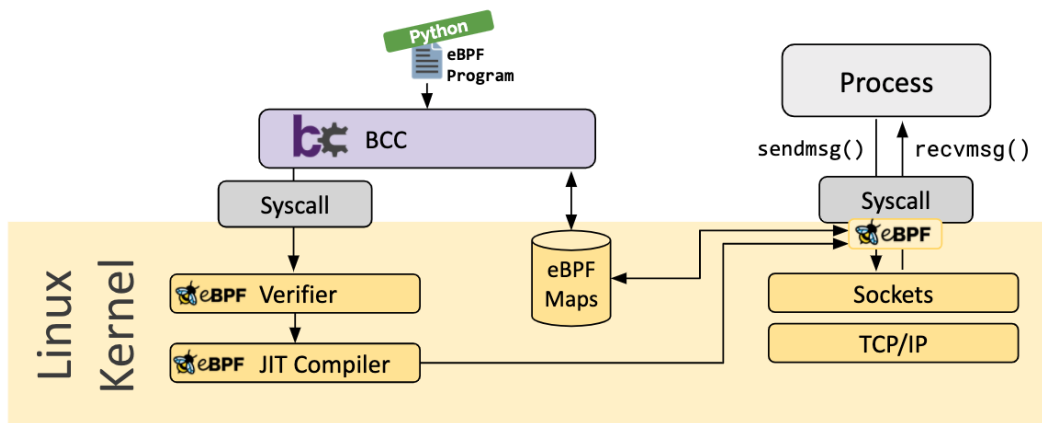


Development toolchain:

Several development toolchains exist to assist in the development and management of eBPF programs. All of them address different needs of users, like:

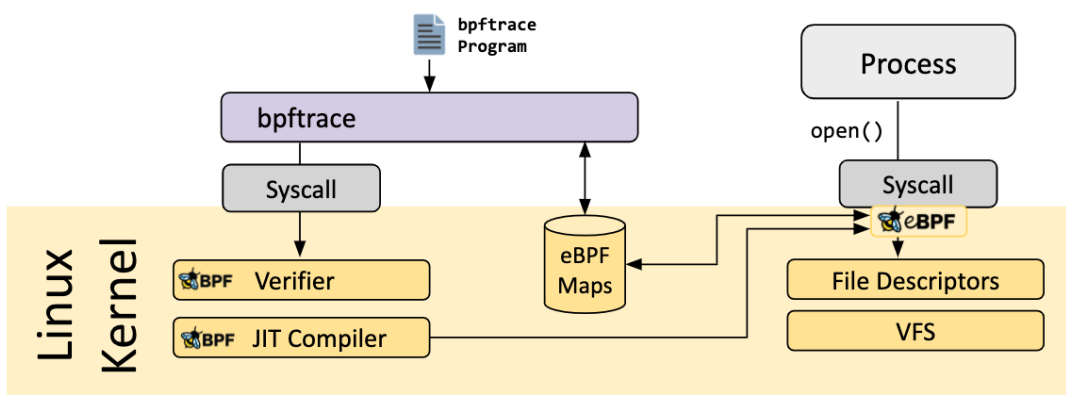
1. bcc Python library

BCC is a framework that enables users to write python programs with eBPF programs embedded inside them. The framework is primarily targeted for use cases that involve application and system profiling/tracing where an eBPF program is used to collect statistics or generate events, and a counterpart in userspace collects the data and displays it in a human-readable form. Running the python program will generate the eBPF bytecode and load it into the kernel.



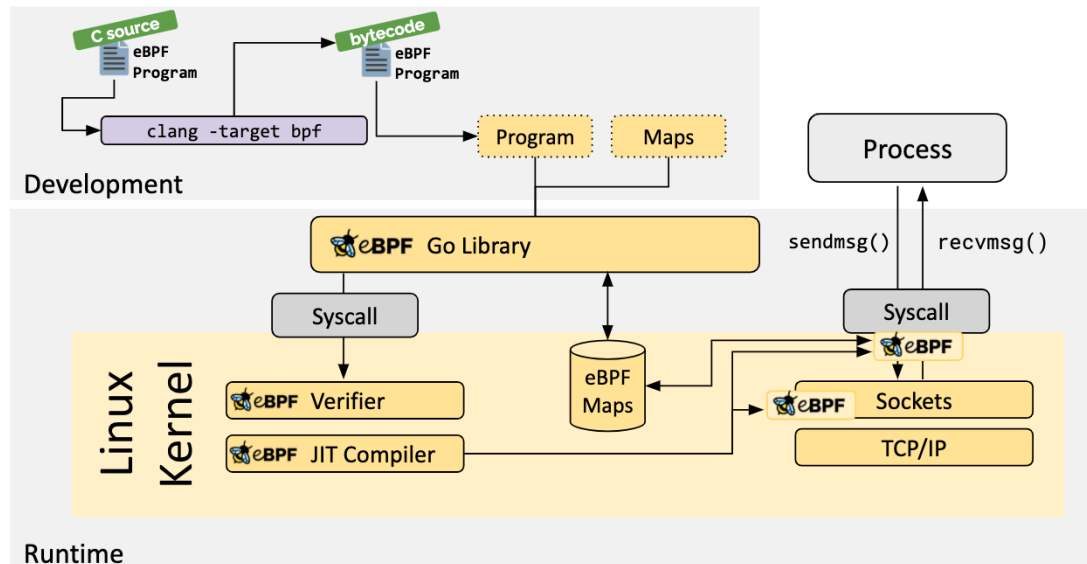
2. bpftool

bpftool is a high-level tracing language for Linux eBPF and is available in recent Linux kernels (4.x). bpftool uses LLVM as a backend to compile scripts to eBPF bytecode and uses BCC for interacting with the Linux eBPF subsystem and existing Linux tracing capabilities: kernel dynamic tracing (kprobes), user-level dynamic tracing (uprobes), and tracepoints. The bpftool language is inspired by awk, C and predecessor tracers such as DTrace and SystemTap.



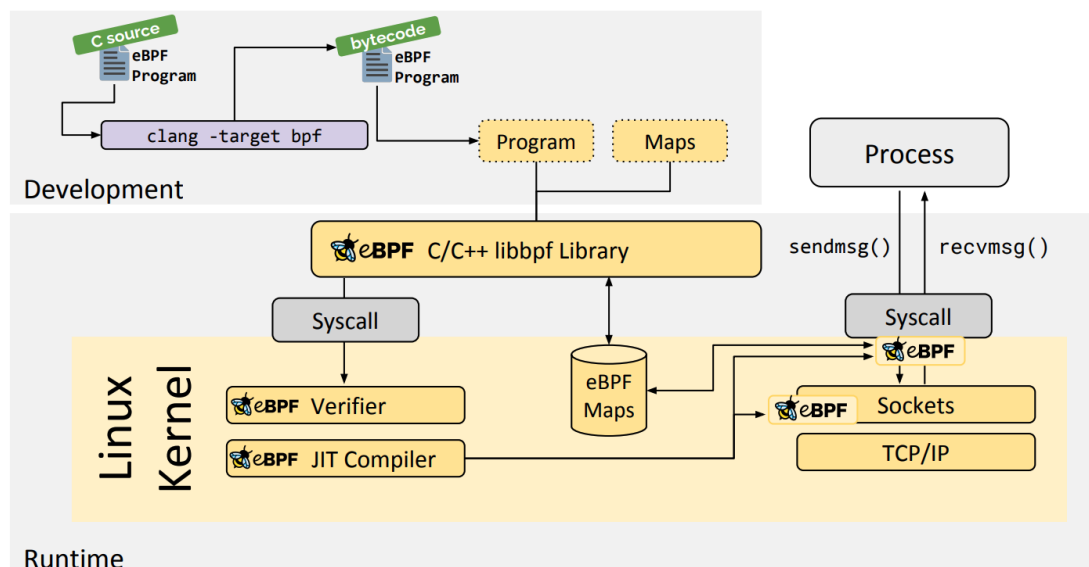
3. eBPF Go Library

The eBPF Go library provides a generic eBPF library that decouples the process of getting to the eBPF bytecode and the loading and management of eBPF programs. eBPF programs are typically created by writing a higher-level language and then using the clang/LLVM compiler to compile to eBPF bytecode.



4. libbpf C/C++ Library

The libbpf library is a C/C++-based generic eBPF library that helps to decouple the loading of eBPF object files generated from the clang/LLVM compiler into the kernel and generally abstracts interaction with the BPF system call by providing easy to use library APIs for applications.



Keylogger:

The file structure of the `src/` directory is:

```
> tree src
src
├── bpf
│   ├── bpf_program.c
│   ├── bpf_program.h
│   └── helpers.h
├── python
│   ├── archiver.py
│   ├── bpf_program.py
│   ├── defs.py
│   ├── keys.py
│   ├── main.py
│   ├── __pycache__
│   │   ├── bpf_program.cpython-38.pyc
│   │   ├── defs.cpython-38.pyc
│   │   ├── keys.cpython-38.pyc
│   │   └── utils.cpython-38.pyc
│   └── utils.py
└── 3 directories, 13 files
```

The `bpf/` directory holds the c code that is injected into the kernel and it has the kprobe that runs inside the kernel. This kprobe triggers the uprobe that runs the code inside `bpf_program.py`.

`kprobe__input_handle_event()` function is the trigger for every key keystroke in the kernel that runs the uprobe.

The `main.py` holds the starter code that triggers the kprobe to start.

`keys.py` holds the conversion of the [input-event-codes.h](https://www.kernel.org/doc/Documentation/input-event-codes.txt) from torvalds/linux repository in an interpretable form for the python code to pretty-print the key logs.

To run the archiver module, `main.py` runs a separate thread to execute `archiver.py` script every x seconds and does not hinder the keylogger's progress.

Results

The code repository can be found on Github <https://github.com/sloorush/ebpf-keylogger>

Here are the results of the project run in every mode:

- `sudo ./ebpf-keylogger -h` Running the help command tells us about all the possible arguments that the software can do:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

> sudo ./ebpf-keylogger -h
[sudo] password for rush:
usage: bpf-keylogger [-h] [--debug] [-t] [-o OUTFILE] [-u]

A keylogger written in eBPF.

optional arguments:
  -h, --help            show this help message and exit
  --debug               Print debugging info.
  -t, --timestamp       Print time stamps.
  -o OUTFILE, --outfile OUTFILE
                        Output trace to a file instead of stdout.
  -u, --upload          upload the files to google drive

~ /usr/p/ebpf-keylogger  master
```

- `sudo ./ebpf-keylogger` Running the keylogger in normal mode runs the keylogger to log all keystrokes using the kprobe to detect keystrokes and log them to stdout of the terminal:

```
> sudo ./ebpf-keylogger
Namespace(debug=False, outfile=None, timestamp=False, upload=False)
Logging key presses... ctrl-c to quit
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
CAPSLOCK
CAPSLOCK
SHIFT
SHIFT
SHIFT+Z
CTRL
ALT
META
CTRL
CTRL+L
CTRL
CTRL+C
^C

~ /usr/p/ebpf-keylogger  master
```


- `sudo ./ebpf-keylogger -t` Running it with the `-t` flag adds more information to the output adding timestamps of every keystroke:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

> sudo ./ebpf-keylogger -t
Namespace(debug=False, outfile=None, timestamp=True, upload=False)
Logging key presses... ctrl-c to quit
[2021/12/02 20:00:35] A
[2021/12/02 20:00:36] B
[2021/12/02 20:00:36] C
[2021/12/02 20:00:36] D
[2021/12/02 20:00:36] H
[2021/12/02 20:00:37] E
[2021/12/02 20:00:37] W
[2021/12/02 20:00:38] CTRL
[2021/12/02 20:00:38] SHIFT
[2021/12/02 20:00:38] CTRL
[2021/12/02 20:00:39] CTRL+Z
[2021/12/02 20:00:41] CTRL
[2021/12/02 20:00:41] CTRL+C
^C
~/.ebpf-keylogger master
```

- `sudo ./ebpf-keylogger --debug` Running the tool in debug mode with `--debug` flag gives the developers more data to analyse, as it logs the data that the kprobe gets without pretty printing it, this logs several data objects like when there is a key-down and a key-up with notification of idle mouse clicks with accurate UNIX timestamp:

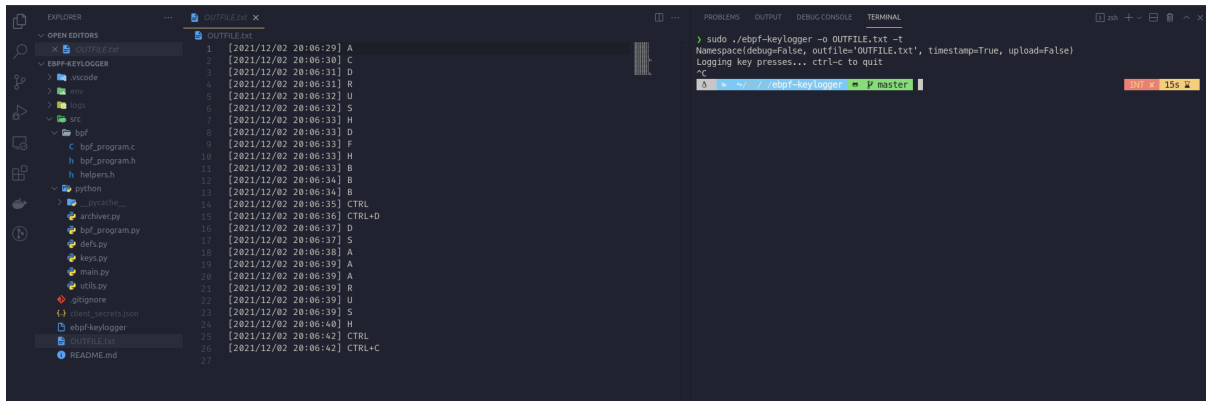
```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

> sudo ./ebpf-keylogger --debug
Namespace(debug=True, outfile=None, timestamp=False, upload=False)
Logging key presses... ctrl-c to quit
b'      <idle>-0      [006] d.h. 17903.191707: 0: key up 272'
b'      <idle>-0      [006] d.h. 17904.758708: 0: key down 272'
b'      <idle>-0      [006] d.h. 17904.758712: 0: value 1'
b' GpuMemoryThread-246629 [006] d.h. 17904.849716: 0: key up 272'
b'      threaded-ml-241761 [004] d.h. 17905.874769: 0: key down 33'
b'      threaded-ml-241761 [004] d.h. 17905.874773: 0: value 1'
b'      threaded-ml-241761 [004] d.h. 17905.874781: 0: key up 464'
b'      <idle>-0      [004] d.h. 17905.965775: 0: key up 33'
b'      <idle>-0      [004] d.h. 17905.965781: 0: key up 464'
b'      <idle>-0      [004] dNh. 17907.207774: 0: key down 17'
b'      <idle>-0      [004] dNh. 17907.207777: 0: value 1'
b'      <idle>-0      [004] dNh. 17907.207784: 0: key up 464'
b'      <idle>-0      [004] d.h. 17907.320771: 0: key up 17'
b'      <idle>-0      [004] d.h. 17907.320776: 0: key up 464'
b'      <idle>-0      [004] d.h. 17908.388768: 0: key down 16'
b'      <idle>-0      [004] d.h. 17908.388771: 0: value 1'
b'      <idle>-0      [004] d.h. 17908.388777: 0: key up 464'
b'      <idle>-0      [004] d.h. 17908.502775: 0: key up 16'
b'      <idle>-0      [004] d.h. 17908.502780: 0: key up 464'
b'      <idle>-0      [004] d.h. 17909.930772: 0: key down 38'
b'      <idle>-0      [004] d.h. 17909.930775: 0: value 1'
b'      <idle>-0      [004] d.h. 17909.930781: 0: key up 464'
b'      <idle>-0      [004] d.h. 17910.022777: 0: key up 38'
b'      <idle>-0      [004] d.h. 17910.022782: 0: key up 464'
b'      <idle>-0      [004] d.h. 17910.177771: 0: key down 37'
b'      <idle>-0      [004] d.h. 17910.177774: 0: value 1'
b'      <idle>-0      [004] d.h. 17910.177782: 0: key up 464'
b'      <idle>-0      [004] d.h. 17910.290771: 0: key up 37'
b'      <idle>-0      [004] d.h. 17910.290776: 0: key up 464'
b'      <idle>-0      [004] d.h. 17910.637770: 0: key down 33'
b'      <idle>-0      [004] d.h. 17910.637773: 0: value 1'
b'      <idle>-0      [004] d.h. 17910.637780: 0: key up 464'
b'      <idle>-0      [004] d.h. 17910.749776: 0: key up 33'
b'      <idle>-0      [004] d.h. 17910.749781: 0: key up 464'
b' also-source-ALC-2097 [004] d.h. 17911.383777: 0: key down 32'
b' also-source-ALC-2097 [004] d.h. 17911.383781: 0: value 1'
b' also-source-ALC-2097 [004] d.h. 17911.383788: 0: key up 464'
b'      <idle>-0      [004] d.h. 17911.474778: 0: key up 32'
b'      <idle>-0      [004] d.h. 17911.474785: 0: key up 464'
b'      <idle>-0      [004] d.h. 17912.915776: 0: key down 18'
b'      <idle>-0      [004] d.h. 17912.915779: 0: value 1'
b'      <idle>-0      [004] d.h. 17912.915787: 0: key up 464'
b'      <idle>-0      [004] d.h. 17912.984772: 0: key up 18'
b'      <idle>-0      [004] d.h. 17912.984776: 0: key up 464'
b'      <idle>-0      [006] d.h. 17914.030709: 0: key down 272'
b'      <idle>-0      [006] d.h. 17914.030712: 0: value 1'
b' ThreadPoolForeg-205601 [006] d.h. 17914.099697: 0: key up 272'
b'      <idle>-0      [004] d.h. 17914.442779: 0: key down 29'
b'      <idle>-0      [004] d.h. 17914.442782: 0: value 1'

```

- `sudo ./ebpf-keylogger -o outfile.txt -t` Running the tool with the `-o` flag with a target file, directs the stdout to a file and saves all key logs there:



The screenshot shows a VS Code editor with a file explorer on the left and a terminal window on the right. The file explorer shows a project structure with files like `ebpf-keylogger`, `outfile.txt`, and `README.md`. The terminal window shows the command `sudo ./ebpf-keylogger -o outfile.txt -t` being executed, followed by the output: `Namespace(debug=False, outfile='outfile.txt', timestamp=True, upload=False)` and `Logging key presses... ctrl-c to quit`. The terminal also shows a list of key presses: `A`, `C`, `D`, `R`, `U`, `S`, `H`, `D`, `F`, `H`, `B`, `B`, `B`, `CTRL`, `D`, `S`, `A`, `A`, `R`, `U`, `H`, `CTRL`, and `CTRL+C`.

- `sudo ./ebpf-keylogger -o OUTFILE.txt -t -u` Running the tool in upload mode with the `-u` flag, uploads the `outfile.txt` to the set google drive folder. For demonstration purposes, we have set it to upload every 5 seconds to the directed folder and log the title of the folder with the google drive file id:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
> sudo ./ebpf-keylogger -o OUTFILE.txt -t -u
Namespace(debug=False, outfile='OUTFILE.txt', timestamp=True, upload=True)
Uploading to gdrive as well!
/home/rush/Desktop/projects/ebpf-keylogger/client_secrets.json
Logging key presses... ctrl-c to quit
title: /home/rush/Desktop/projects/ebpf-keylogger/OUTFILE.txt, id: 1ofCnesjntT5RjjVXPEY5YpsLcZsHJ8_y
/home/rush/Desktop/projects/ebpf-keylogger/client_secrets.json
title: /home/rush/Desktop/projects/ebpf-keylogger/OUTFILE.txt, id: 1_VVhwFdFdIIErW9FhTWw6JUiXIKuaa9_
/home/rush/Desktop/projects/ebpf-keylogger/client_secrets.json
title: /home/rush/Desktop/projects/ebpf-keylogger/OUTFILE.txt, id: 1UNI6vw3lkSalyVpj12s_hJLhENGxN_IC

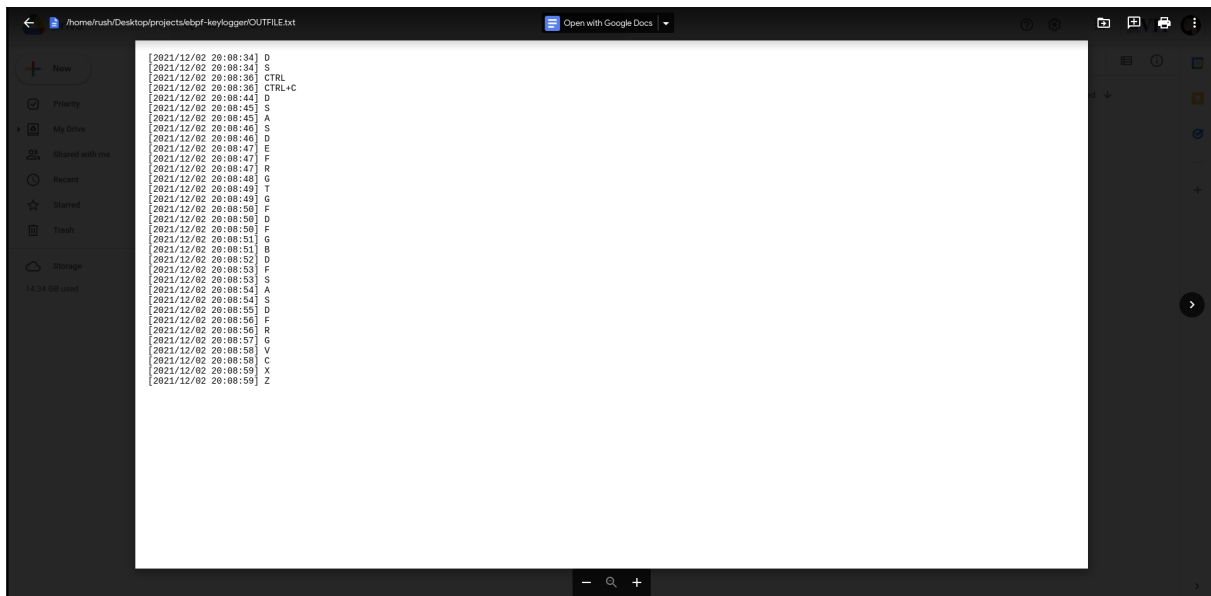
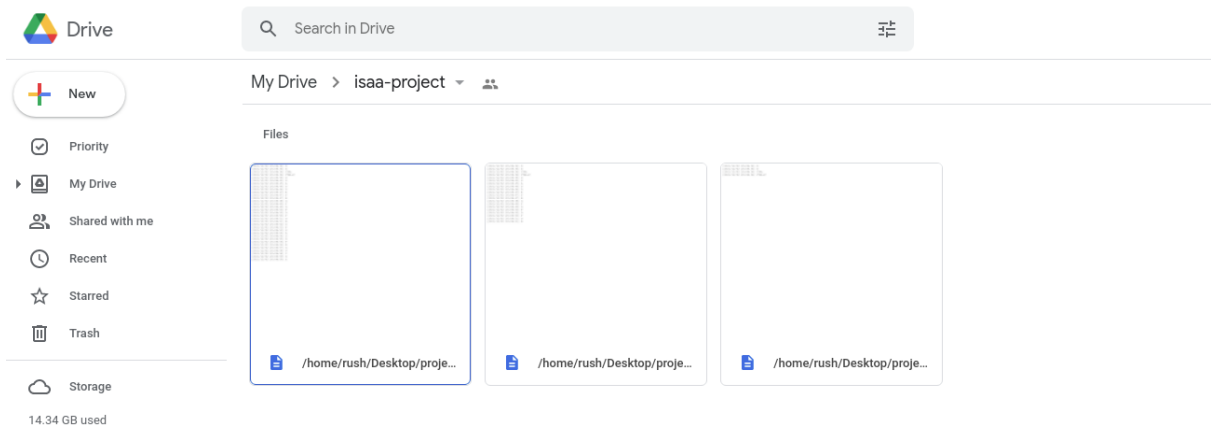
```

```

OUTFILE.txt
1 [2021/12/02 20:08:34] D
2 [2021/12/02 20:08:34] S
3 [2021/12/02 20:08:36] CTRL
4 [2021/12/02 20:08:36] CTRL+C
5 [2021/12/02 20:08:44] D
6 [2021/12/02 20:08:45] S
7 [2021/12/02 20:08:45] A
8 [2021/12/02 20:08:46] S
9 [2021/12/02 20:08:46] D
10 [2021/12/02 20:08:47] E
11 [2021/12/02 20:08:47] F
12 [2021/12/02 20:08:47] R
13 [2021/12/02 20:08:48] G
14 [2021/12/02 20:08:49] T
15 [2021/12/02 20:08:49] G
16 [2021/12/02 20:08:50] F
17 [2021/12/02 20:08:50] D
18 [2021/12/02 20:08:50] F
19 [2021/12/02 20:08:51] G
20 [2021/12/02 20:08:51] B
21 [2021/12/02 20:08:52] D
22 [2021/12/02 20:08:53] F
23 [2021/12/02 20:08:53] S
24 [2021/12/02 20:08:54] A
25 [2021/12/02 20:08:54] S
26 [2021/12/02 20:08:55] D
27 [2021/12/02 20:08:56] F
28 [2021/12/02 20:08:56] R
29 [2021/12/02 20:08:57] G
30 [2021/12/02 20:08:58] V
31 [2021/12/02 20:08:58] C
32 [2021/12/02 20:08:59] X
33 [2021/12/02 20:08:59] Z
34 [2021/12/02 20:09:00] S
35 [2021/12/02 20:09:01] D
36 [2021/12/02 20:09:01] F
37 [2021/12/02 20:09:02] CTRL
38 [2021/12/02 20:09:04] CTRL
39 [2021/12/02 20:09:04] CTRL+C
40

```

- Screenshots from google drive:



Conclusion and Future Work

Making a non-intrusive monitoring tool like a KeyLogger seems like a must-have thing in the security space. eBPF is a relatively new field in tech, has excellent potential but very limited resources and proven work to experiment with. The tech is revolutionary in the Linux kernel that can run sandboxed programs in an operating system kernel. This project also paves a route to explore eBPF's possibility in the field of security with the help of a kernel-level Keylogger.

The power of adding new custom features to an operating system makes eBPF a unique tool. Earlier, getting a feature approved and working on a Linux distro would take around 4-5 years, which proved useless as the user's requirement would change in the duration. With eBPF in the picture, this process is instant and would satisfy user requirements in quick succession.

In future, we plan to implement more strict security measures on our Keylogger, including modern AI techniques, to predict any malicious activity on the server.

References

1. [The rise of eBPF for non-intrusive performance monitoring](#)
2. [eBPF-based Content and Computation-aware Communication for Real-time Edge Computing](#)
3. [\(PDF\) eBPF - From a Programmer's Perspective](#)
4. [\(PDF\) Survey of Keylogger Technologies](#)
5. [\(PDF\) Keyloggers: silent cyber security weapons](#)
6. [Investigating the keylogging threat in android — User perspective \(Regular research paper\)](#)
7. [Combining System Visibility and Security Using eBPF](#)
8. [Accelerating Linux Security with eBPF iptables](#)
9. [A Framework for eBPF-Based Network Functions in an Era of Microservices](#)
10. [bpffbox: Simple Precise Process Confinement with eBPF](#)
11. [What is eBPF? An Introduction and Deep Dive into the eBPF Technology](#)
12. [Keyloggers: How they work and how to detect them \(Part 1\)](#)
13. [Linux: easy keylogger with eBPF](#)
14. [A Gentle Introduction to eBPF](#)
15. [Andrea Righi - Spying on the Linux kernel for fun and profit](#)
16. [Making eBPF work on Windows](#)
17. <https://www.refog.com>

Appendix

Aarush Bhat(19BCE0564): The mind behind the ideation, eBPF implementation

Jyotir Aditya(19BCE0846): Keylogger and Documentation