

# **INDIAN INSTITUTE OF TECHNOLOGY HYDERABAD**



## **Topics in Networks (CS6220) 2022**

A project report on

### **Port PUF based UAV authentication verifier logic to XDP**

**Under the guidance of Dr. Praveen Tammana**

**Mentor:** Ranjitha K

#### **Team**

Prem Kumar Saraf (cs21mtech11014)  
Sandeep Kumar Nallala(cs21mtech11002)  
(G10)

## Table of Contents

Sl. No	Content	Page No
1.	Abstract	3
2	Introduction	4
3	Existing Works	5
3.1	Digital signatures based solutions	5
3.2	Cryptographic security solutions	5
3.3	PUF based solutions	5
3.4	PUF based solution with verifier logic off-loaded to high speed programmable data planes	6
4	Extended Berkeley Packet Filters (eBPF)	6
5	Network hooks	7
6	eXpress Data Path	7
7	Experimental Setup	7
7.1	Topology	8
8	Implementation	8
9	Conclusion	13
10	Instructions for Execution	14
11	References	15

## ABSTRACT

Unmanned Aerial vehicles (eg. drones) are very resource-constrained and possess low energy sources and constrained processing capabilities. Existing security mechanisms use Cryptographic techniques which include encryption and authentication. But this method involves storing the keys in the memory of UAVs, which is very limited. It can also lead to some kinds of attacks from malicious people. Other works proposed to employ the PUF (Physical Unclonable Functions) circuits, which provide a unique way to identify integrated circuits. PUFs exploit this inherent variability in integrated circuit manufacturing to implement challenge-response functions whose output depends on the input and on the physical microstructure of the device. Addressing security issues to a good extent, it has overlooked a few important parameters like latency, reliability, throughput, etc. This method involves a 3 step mechanism where the authentication code is deployed in the Server and first UAV sends a request to the server, the server replies with a “challenge”, and then the UAV sends a “response”, which then needs to be authenticated at a general-purpose CPU server machine. eXpress Data Path(XDP) provides a high performance, programmable network data path in the Linux kernel. It involves hooking the packet processing logic at Network Interface Card(NIC) that decides where the packet has to be directed. The packet can be checked and passed to the kernel or it can be dropped. This decreases the latency and the overhead for the kernel. Here UAV authentication using PUF verifier logic is done at XDP.

**Key Words:** *PUF, XDP, challenge, response.*

# INTRODUCTION

The usage of Unmanned Aerial Vehicle (UAV) technology has not been very uncommon due to its wide variety of applications in various sectors. An increase in its usage has also paved the way for malicious use by cyber-criminals. Sometimes, the result of the attack can be very worse. So, an effective secure communication channel is needed in order to protect the integrity and privacy of applications and sensitive data from physical attacks. Existing security mechanisms use Cryptographic techniques which include encryption and authentication. But this method involves storing the keys in the memory of UAVs, which is very limited. Other works proposed to employ the PUF (physical unclonable functions) circuits, which provide a unique way to identify integrated circuits. Comparable in a simplistic way with a “unique fingerprint” of an IC that differentiates one IC from another (though apparently identical), PUFs exploit this inherent variability in integrated circuit manufacturing to implement challenge-response functions whose output depends on the input and on the physical microstructure of the device. Addressing security issues to a good extent, it has overlooked a few important parameters like latency, reliability, throughput, etc. in this method, the challenge-response pairs of each and every UAV device are stored at the server/verifier side. Then a 3 step mechanism is employed. The first step involves UAV sending a request to the server, the server replies with a “challenge”, from its stored pairs for that device and then the UAV sends a “response”, which then needs to be authenticated at a general-purpose CPU server machine.

Here, we study and explore the fast packet processing power of XDP, in the context of authentication of the UAV devices, thereby decreasing the latency and increasing the authentication rate. Our work includes hooking the XDP logic at the NIC of the verifier side. The client sends an authentication request message which is first caught at the Network interface Card level itself, before allowing it to go to the Linux kernel network stack, performing the check if it is indeed a request packet and then redirecting it accordingly to userspace. It helps in the reduction of time and thereby the latency.

## **EXISTING WORKS**

### **a. Digital Signatures based solutions**

Certificate-based Digital Signatures for the Internet of Drones have been worked on and failed to provide protection against physical attacks. Another signature-based authentication scheme has also been proposed which involves high communication and computational overheads. A blockchain-based authentication scheme has been worked on for security and privacy enhancements, but it results in huge space and computational complexity.

### **b. Cryptographic security solutions**

Many cryptographic frameworks and security solutions for authentication of UAV are being proposed. For UAV authentication both symmetric and asymmetric key exchange schemes were used. A commonly used method for UAV authentication is using AES algorithm. It is a symmetric algorithm, in this algorithm, the key is shared between both the UAV and ground station. And the cryptographic solutions provide integrity and authentication which tell that the message that the receiver has received is the same message that the sender has sent and it is sent only by the intended sender. Although cryptographic based solutions are secured, it needs high computational power. For example, even a cryptographic algorithm which has a probability of finding the solution by pure chance of one in a billion is not considered secured as modern computers can do billions of operations per second. So, to provide security computational complexity has to be increased.

### **c. PUF based solutions**

To overcome the challenges faced by the cryptographic solutions that are high computation, and resource constrained issue in UAV, another solution is being proposed, i.e; PUF based solution. PUF works based on challenge-response solution. That is, given a challenge input, the receiver has to produce an appropriate response to that challenge. PUF are unclonable. That is, two different PUFs manufactured with the same circuit and fabrication process would give different outputs for the same input, so these PUFs are used as unique identifiers analogues to fingerprints. There is no secret key stored in the UAV or ground station, so if an intruder gets access to the UAV, neither the key gets

exposed nor the PUF circuit gets cloned. This prevents from generating the same Challenge-Response pairs and attacks further. These authentication mechanisms are lightly weighted so that they are efficient for UAVs.

#### **d. PUF based authentication by off-loading verification logic to high-speed programmable data planes**

In this approach, the verifier logic from the general-purpose CPU machines is being off-loaded to a very high-speed packet processing data plane of programmable switches (eg. Tofino). It proved to be very efficient and optimal. But the problem is switches like Tofino are used mainly for academic and research purposes and may not be deployed anywhere and everywhere due to their limited availability.

## **EXTENDED BERKELEY PACKET FILTERS (eBPF)**

Extended Berkeley Packet Filter(eBPF) lies inside the Linux kernel. It is an instruction set and an execution environment inside the Linux Kernel. eBPF enables interaction, modification and kernel programmability at runtime. Express Data Path(XDP) is a kernel network layer which processes packets closer to the NIC fast. Developers can write programs in P4 or C languages and then compile them to eBPF instructions. These compiled instructions can be processed by the kernel or programmable devices (e.g. SmartNics). Since eBPF was introduced it has been adopted by many companies such as Facebook, Cloudflare, Netronome etc. It is used in network monitoring, load balancing, network traffic manipulation, system profiling etc.

An eBPF program is written in a high-level language (mainly restricted C). The clang compiler transforms it into an ELF/object code. An ELF eBPF loader can then insert it into the kernel using a special system call. During this process, the verifier analyzes the program and upon approval, the kernel performs the dynamic translation (JIT). The program can be offloaded to hardware, otherwise, it is executed by the processor itself. To ensure the integrity and security of the operating system, the kernel uses a verifier that performs static program analysis of eBPF instructions being loaded into the system. Its implementation is available at kernel/bpf/verifier.c in the kernel source code. some eBPF functions can only be called by programs with GPL compatible licenses. Because of that, the verifier checks whether the licenses of the functions used by a program and the program's license are compatible and rejects the program if they are not.

Last, the verifier does not allow memory access beyond the local variables and packet boundaries to ensure the integrity and security of the kernel. To access any bytes in the packet, it is always necessary to perform a border check. However, each byte only needs to be checked once, unless the storage space of the packet gets modified. This way, during the analysis of the program, the verifier guarantees that all memory accesses made to the packet are in checked addresses. If the eBPF program does not do this type of check, then the verifier rejects it, and so it cannot be loaded in the kernel.

### **Network Hooks:**

In Computer Networking, hooks are used for intercepting packets before the call or during execution in the operating system. The Linux kernel exposes several hooks to which eBPF programs can be attached, enabling data collection and custom event handling.

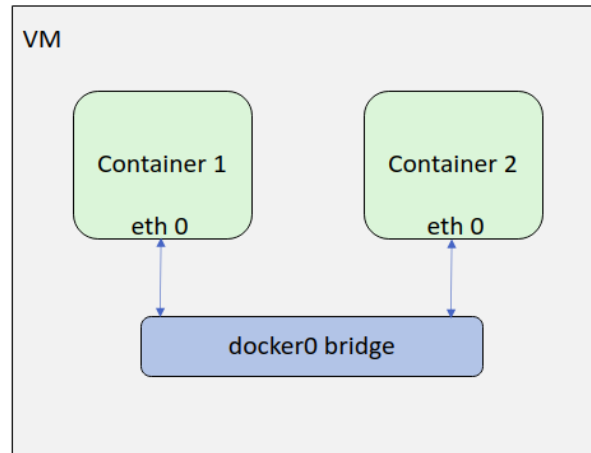
## **EXPRESS DATA PATH (XDP)**

eXpress Data Path (XDP) is the lowest layer of the Linux kernel network stack. It is present only on the RX path, inside a device's network driver, allowing packet processing at the earliest point in the network stack, even before memory allocation is done by the OS. It exposes a hook to which eBPF programs can be attached. In this hook, programs are capable of taking quick decisions about incoming packets and also performing arbitrary modifications on them, avoiding additional overhead imposed by processing inside the kernel. After processing a packet, an XDP program returns an action, which represents the final verdict regarding what should be done to the packet after program exit.

## **EXPERIMENTAL SETUP**

We have used a Virtual machine with an ubuntu 20.04 OS, with 8GB RAM and an intel i5 octa-core processor. Two docker containers are created which are connected through veth interfaces. One container is assumed as a client and the client program is run in that container whereas the other container is assumed to be the server.

## Topology:



## IMPLEMENTATION

For our purpose, to identify the packets with the required formats like authentication request, challenge, response and acknowledgement packets, we define a structure as follows

```
struct auth_header{
    uint8_t msgType;
    uint32_t challenge;
};
```

For different packets we set the msgType field as follows:

For authentication request , msgType = 0x00

For challenge packet, msgType = 0x01

For response packet, msgType = 0x02

For ack packet with successful authentication msgType = 0x03

For ack with failed authentication msgType = 0x04

For storing the “challenge-response” pairs, we need to use a data structure which mimics the “map” features. So for that we create another structure CR as follows.



```
struct CR{
    uint32_t ch;
    uint32_t resp;
};
```

In our implementation, we store only 3 challenge-response pairs. In practice, this can be extended as per requirements.

The first attempt of our implementation involves a client-server implementation of the authentication where the message flows will be as below.

Client to server -> auth request packet  
Server to client -> auth challenge packet  
Client to server -> auth response packet  
Server to client -> auth ack message

We ran the server in the first container and the client in the second container.  
For this first step implementation, the output screenshots are as below.

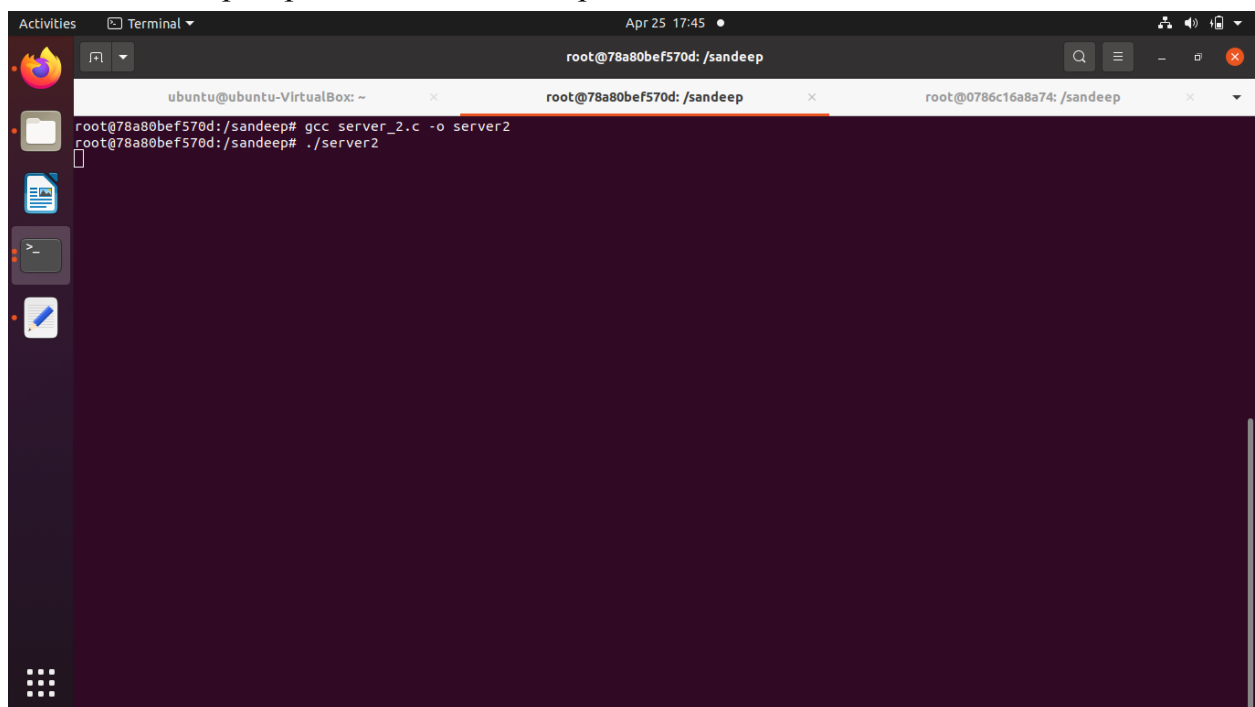
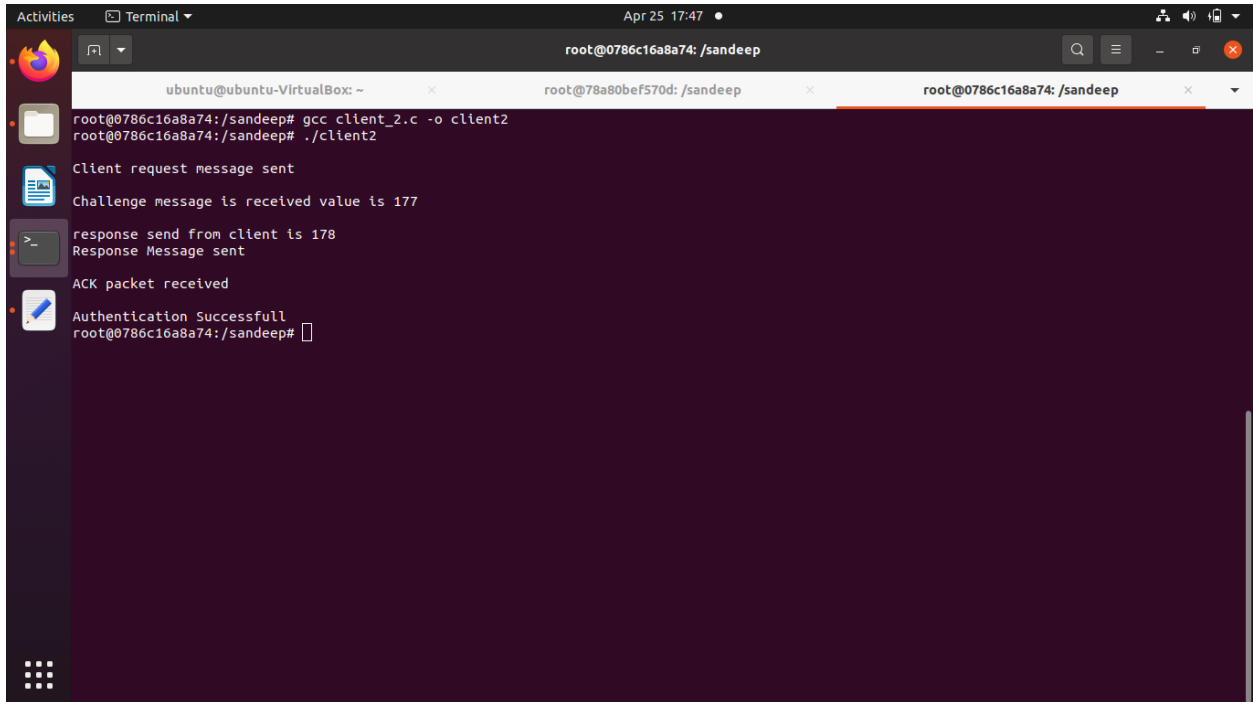


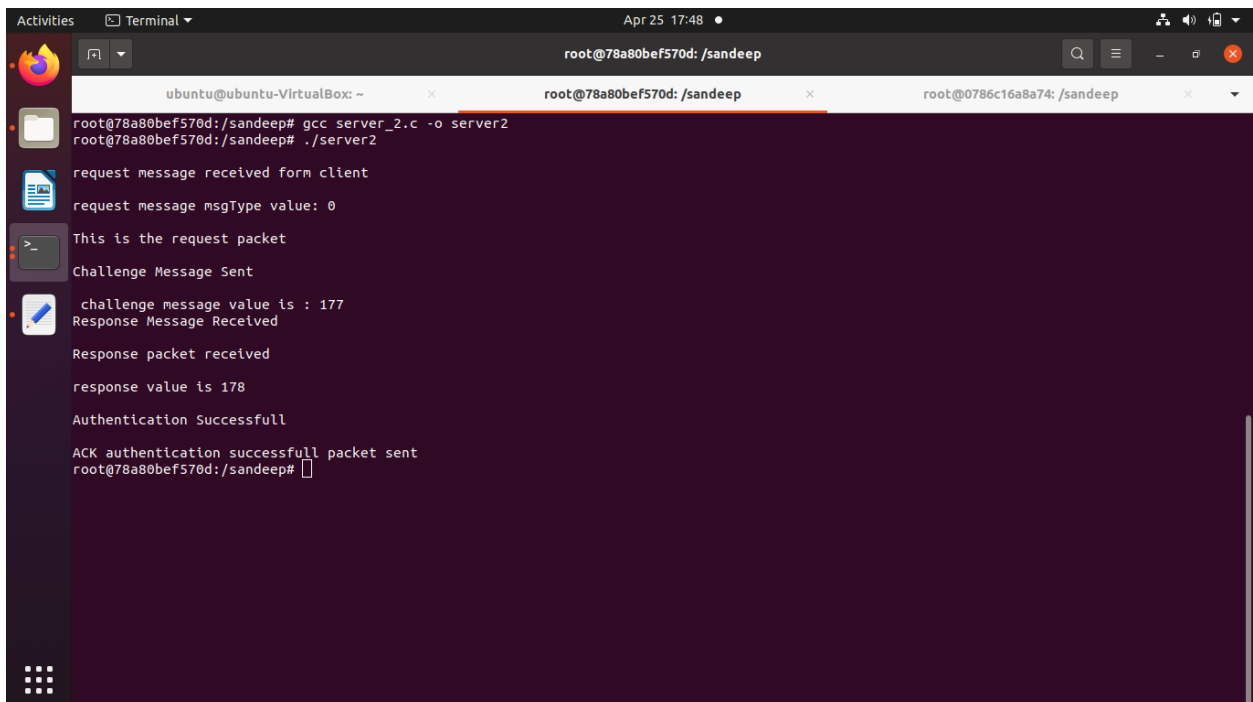
Fig 1. The server is ready and waiting for the client to send the request message



A terminal window titled 'Terminal' with a date and time of 'Apr 25 17:47'. The window shows the execution of a client program. The user is at the prompt 'root@0786c16a8a74: /sandeep'. The commands and output are as follows:

```
root@0786c16a8a74: /sandeep# gcc client_2.c -o client2
root@0786c16a8a74: /sandeep# ./client2
Client request message sent
Challenge message is received value is 177
response send from client is 178
Response Message sent
ACK packet received
Authentication Successfull
root@0786c16a8a74: /sandeep#
```

Fig 2. Client sent the request message and successfully authenticated.



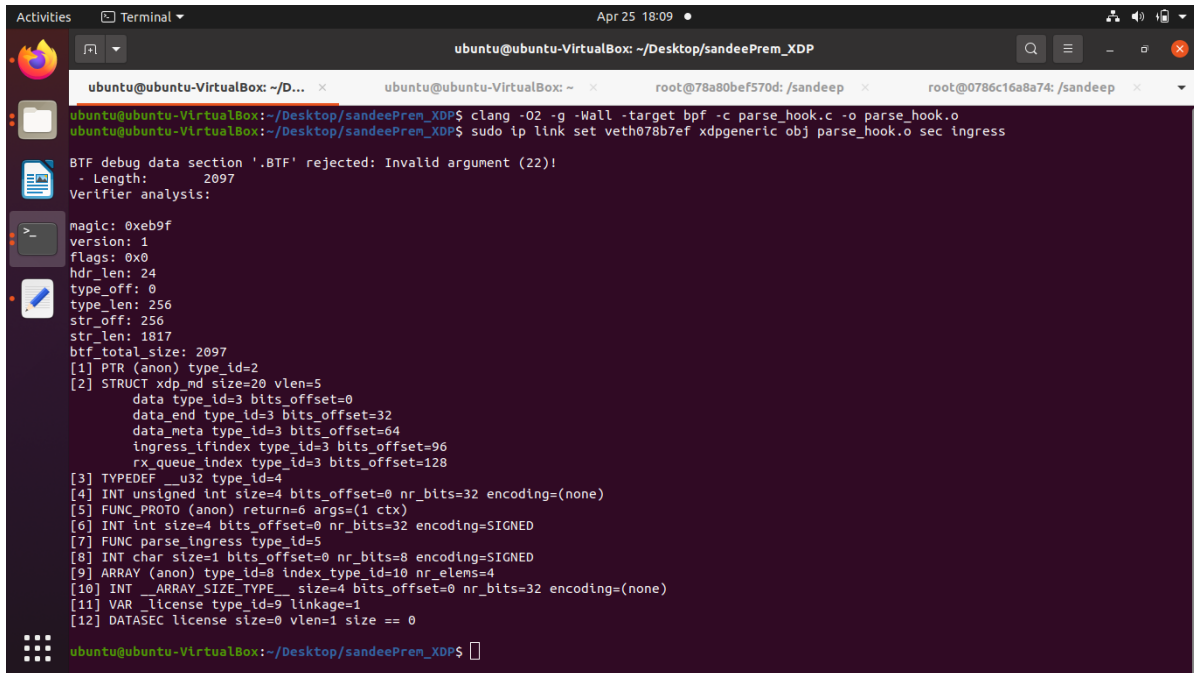
A terminal window titled 'Terminal' with a date and time of 'Apr 25 17:48'. The window shows the execution of a server program. The user is at the prompt 'root@78a80bef570d: /sandeep'. The commands and output are as follows:

```
root@78a80bef570d: /sandeep# gcc server_2.c -o server2
root@78a80bef570d: /sandeep# ./server2
request message received form client
request message msgType value: 0
This is the request packet
Challenge Message Sent
challenge message value is : 177
Response Message Received
Response packet received
response value is 178
Authentication Successfull
ACK authentication successfull packet sent
root@78a80bef570d: /sandeep#
```

Fig 3. Server after successful authentication of the client.

Now we bring the XDP into the picture.

We hook the xdp program to the veth created for the server.



```
ubuntu@ubuntu-VirtualBox: ~/Desktop/sandeePrem_XDP
ubuntu@ubuntu-VirtualBox:~/Desktop/sandeePrem_XDP$ clang -O2 -g -Wall -target bpf -c parse_hook.c -o parse_hook.o
ubuntu@ubuntu-VirtualBox:~/Desktop/sandeePrem_XDP$ sudo ip link set veth078b7ef xdpgeneric obj parse_hook.o sec ingress

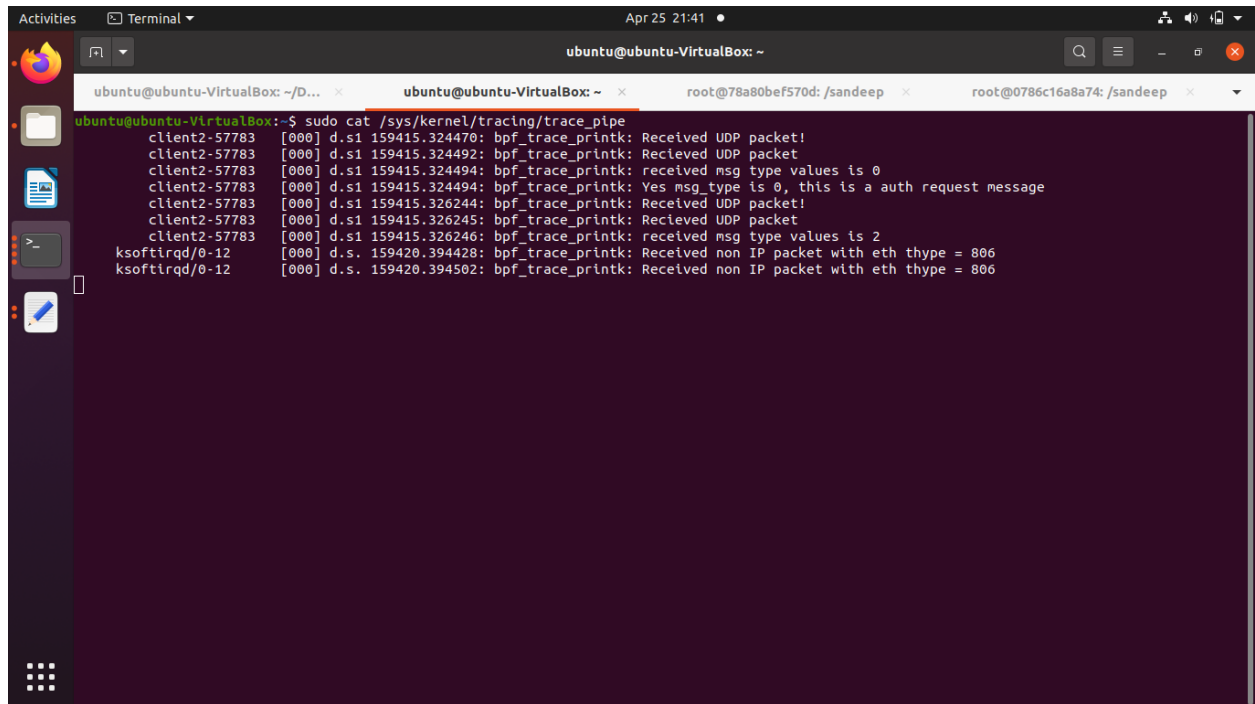
BTF debug data section '.BTF' rejected: Invalid argument (22)!
- Length: 2097
Verifier analysis:
magic: 0xeb9f
version: 1
flags: 0x0
hdr_len: 24
type_off: 0
type_len: 256
str_off: 256
str_len: 1817
btf_total_size: 2097
[1] PTR (anon) type_id=2
[2] STRUCT xdp_md size=20 vlen=5
    data type_id=3 bits_offset=0
    data_end type_id=3 bits_offset=32
    data_meta type_id=3 bits_offset=64
    ingress_ifindex type_id=3 bits_offset=96
    rx_queue_index type_id=3 bits_offset=128
[3] TYPEDEF __u32 type_id=4
[4] INT unsigned int size=4 bits_offset=0 nr_bits=32 encoding=(none)
[5] FUNC_PROTO (anon) return=6 args=(1 ctx)
[6] INT int size=4 bits_offset=0 nr_bits=32 encoding=SIGNED
[7] FUNC parse_ingress type_id=5
[8] INT char size=1 bits_offset=0 nr_bits=8 encoding=SIGNED
[9] ARRAY (anon) type_id=8 index_type_id=10 nr_elems=4
[10] INT __ARRAY_SIZE_TYPE__ size=4 bits_offset=0 nr_bits=32 encoding=(none)
[11] VAR license type_id=9 linkage=1
[12] DATASEC license size=0 vlen=1 size == 0

ubuntu@ubuntu-VirtualBox:~/Desktop/sandeePrem_XDP$
```

Fig 4. Compiling and Successfully hooking the xdp program to the veth interface for server.

In the xdp program, we parsed the packet one by one headers and reached the udp payload of the packet. We then compared the msgType in the request packet with the actual type for request packet that we gave in the program, if they match then we need to send the packet to the userspace program, where it will send the challenge accordingly to the client, and the rest of the process continues from there.

We could not do the project to the fullest, tried some concepts and below we are presenting the Screenshots of the same.



The screenshot shows a terminal window with the following content:

```
ubuntu@ubuntu-VirtualBox: ~$ sudo cat /sys/kernel/tracing/trace_pipe
client2-57783 [000] d.s1 159415.324470: bpf_trace_printk: Received UDP packet!
client2-57783 [000] d.s1 159415.324492: bpf_trace_printk: Received UDP packet
client2-57783 [000] d.s1 159415.324494: bpf_trace_printk: received msg type values is 0
client2-57783 [000] d.s1 159415.324494: bpf_trace_printk: Yes msg_type is 0, this is a auth request message
client2-57783 [000] d.s1 159415.326244: bpf_trace_printk: Received UDP packet!
client2-57783 [000] d.s1 159415.326245: bpf_trace_printk: Received UDP packet
client2-57783 [000] d.s1 159415.326246: bpf_trace_printk: received msg type values is 2
ksoftirqd/0-12 [000] d.s. 159420.394428: bpf_trace_printk: Received non IP packet with eth thype = 806
ksoftirqd/0-12 [000] d.s. 159420.394502: bpf_trace_printk: Received non IP packet with eth thype = 806
```

Fig 5. Debug statement outputs from the program.

They are present at the location `sys/kernel/tracing/trace_pipe`

## RESULTS

Without hooking the XDP, the total time taken from requesting to establishing the connection i.e., acknowledgement is 2.561 ms.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	172.17.0.3	172.17.0.2	UDP	47	47938 → 8080 Len=5
2	0.001165472	172.17.0.2	172.17.0.3	UDP	47	8080 → 47938 Len=5
3	0.002178973	172.17.0.3	172.17.0.2	UDP	47	47938 → 8080 Len=5
4	0.002561675	172.17.0.2	172.17.0.3	UDP	47	8080 → 47938 Len=5
5	5.222194594	02:42:ac:11:00:02	02:42:ac:11:00:03	ARP	42	Who has 172.17.0.3? Tell 172.17.0.2
6	5.222204897	02:42:ac:11:00:03	02:42:ac:11:00:02	ARP	42	Who has 172.17.0.2? Tell 172.17.0.3
7	5.22230284	02:42:ac:11:00:03	02:42:ac:11:00:02	ARP	42	172.17.0.3 is at 02:42:ac:11:00:03
8	5.22232984	02:42:ac:11:00:02	02:42:ac:11:00:03	ARP	42	172.17.0.2 is at 02:42:ac:11:00:02

Frame 4: 47 bytes on wire (376 bits), 47 bytes captured (376 bits) on interface docker0, id 0
Ethernet II, Src: 02:42:ac:11:00:02 (02:42:ac:11:00:02), Dst: 02:42:ac:11:00:03 (02:42:ac:11:00:03)
Internet Protocol Version 4, Src: 172.17.0.2, Dst: 172.17.0.3
User Datagram Protocol, Src Port: 8080, Dst Port: 47938
Data (5 bytes)

0000	32 42 ac 11 00 03 02 42	ac 11 00 02 08 00 45 00	.....R.....R.....F.....
0010	30 21 44 57 40 03 40 11	9e 4d ac 11 00 02 ac 11	..!Dw@.@..M.....
0020	30 03 1f 90 bb 42 00 0d	58 46 03 ef 00 00 00	.....B...XF.....

docker0: <live capture in progress>      Packets: 8 · Displayed: 8 (100.0%)      Profile: Default

Fig 6. Wireshark capture without XDP hooking

## CONCLUSION

XDP is a new technology and can be explored to a great extent and applied in many areas like DDos mitigation, Firewalling, Load balancing etc with very high packet processing speeds. We tried to implement the UAV authentication by porting the UAV verifier logic to the xdp. Since it is not completely done, we think it is a good place to start XDP, and accomplish the current project and look into the latency and the throughput metrics.

## **Instructions for execution:**

1. Open Terminal and check for the current list of containers running using the command  
    `sudo docker ps`.
2. Check the saved states of the containers using the command  
    `sudo docker images`
3. From the list of saved images, open the containers that are saved latest using the command  
    `sudo docker run -it <repository>:<tag> bin/bash`  
    it opens a container (for server/verifier)
4. Type `ifconfig` in the new container. Get its ip address details.
5. Type `ifconfig` in the host terminal window and see the virtual ethernet interface added  
    Let it be `vethserver`.
6. Using the same command in the 3, open a new container for the client also.
7. Type `ifconfig` in the client container and get its ip address.
8. Type `ifconfig` in the host terminal window and see the virtual ethernet interface newly added  
    Let it be `vethclient`.
9. Compile the server program from the server container, and client program from the client container using the `gcc <filename> -o <executable_file_name>`.
10. Compile the `xdp` program from the host terminal window using the following command.

```
clang -O2 -g -Wall -target bpf -c parse_hook.c -o parse_hook.o
```

Note: parse\_hook.c is the xdp program and parse\_hook.o is the executable generated.

11. After compiling, we need to hook it to the veth interface of the server using the following command.

```
sudo ip link set vethserver xdpgeneric obj parse_hook.o sec ingress
```

Note: ingress is the section in the xdp program.

12. To unhook a xdp program, use the below command.

```
sudo ip link set vethserver xdpgeneric off.
```

13. Run the server in its container, and client in the client's container.

14. We can see the debug statements output by using the below command in the host terminal window.

```
sudo cat /sys/kernel/tracing/trace_pipe.
```

## REFERENCES

1. Accelerating PUF-based UAV Authentication Protocols Using Programmable Switch by Praveen Tammana, Antony Franklin, Ranjitha K, Divya Pathak, Tejasvi Alladi. 2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS).

2. [https://homepages.dcc.ufmg.br/~mmvieira/so/papers/Fast\\_Packet\\_Processing\\_with\\_eBPF\\_and\\_XDP.pdf](https://homepages.dcc.ufmg.br/~mmvieira/so/papers/Fast_Packet_Processing_with_eBPF_and_XDP.pdf)

**Link to the repo:** [https://github.com/vijaysandeep114/TiN\\_Project](https://github.com/vijaysandeep114/TiN_Project)

**Link to the ppt slides:**

<https://docs.google.com/presentation/d/1Q2DFRuJ0RiwzrXQ04Ilnb29mBayeSAx9GylToElZqQI/edit?usp=sharing>

