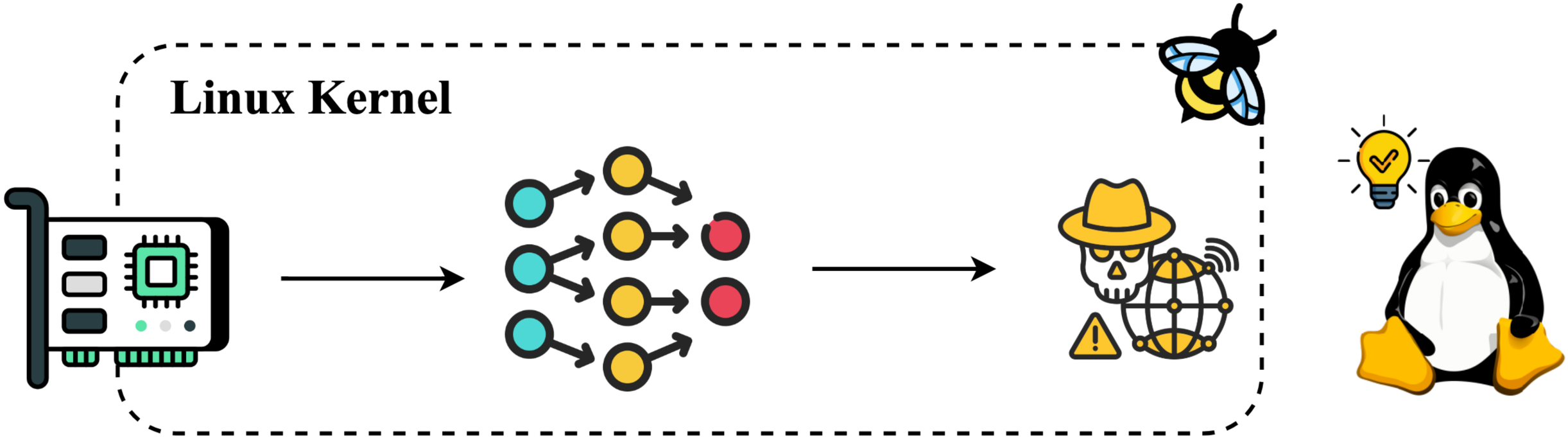
Real Time Intrusion Detection and Prevention With Neural Network in Kernel using eBPF

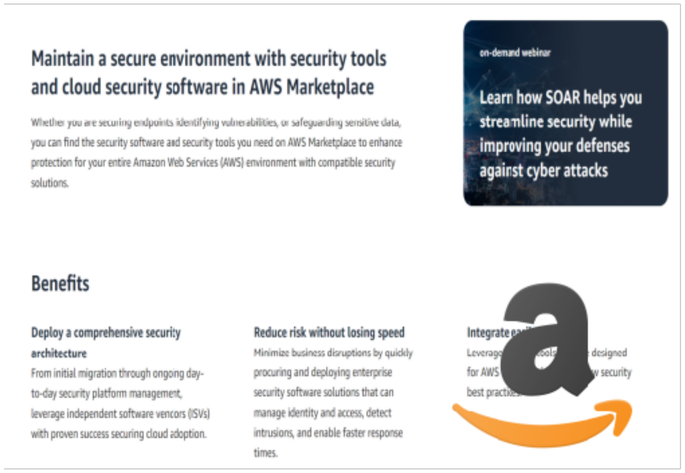
Junyu Zhang, Pengfei Chen, Zilong He, Hongyang Chen, Xiaoyun Li
School of Computer Science and Engineering, Sun Yat-sen University

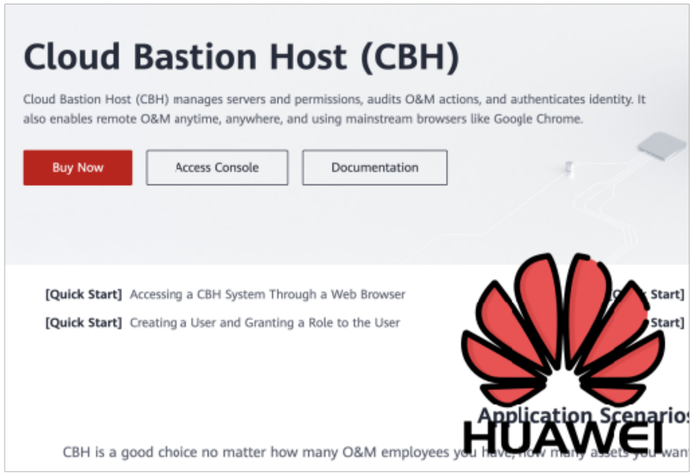
eBPF + Neural Network in Kernel = ?



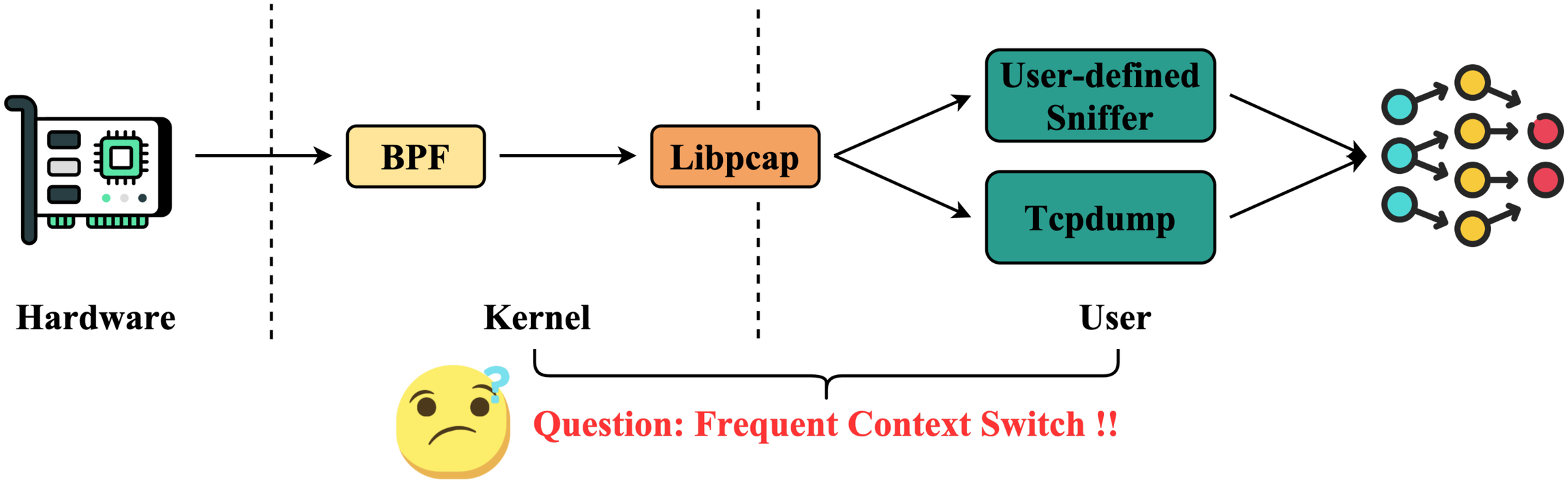
Real Time Detection & Prevention

- Rapid growth in the public cloud service
- Overhead in the real-time packet capturing is neglected
- Defense tools should balance the security level and performance

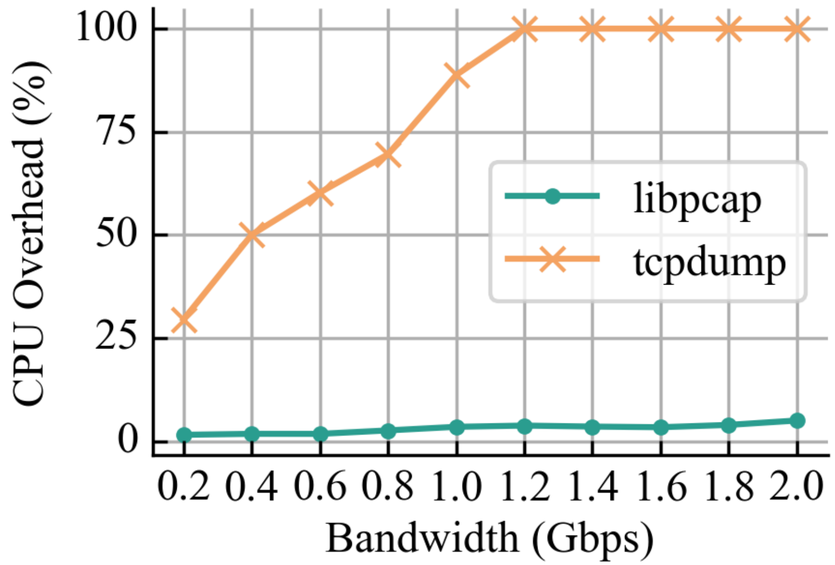


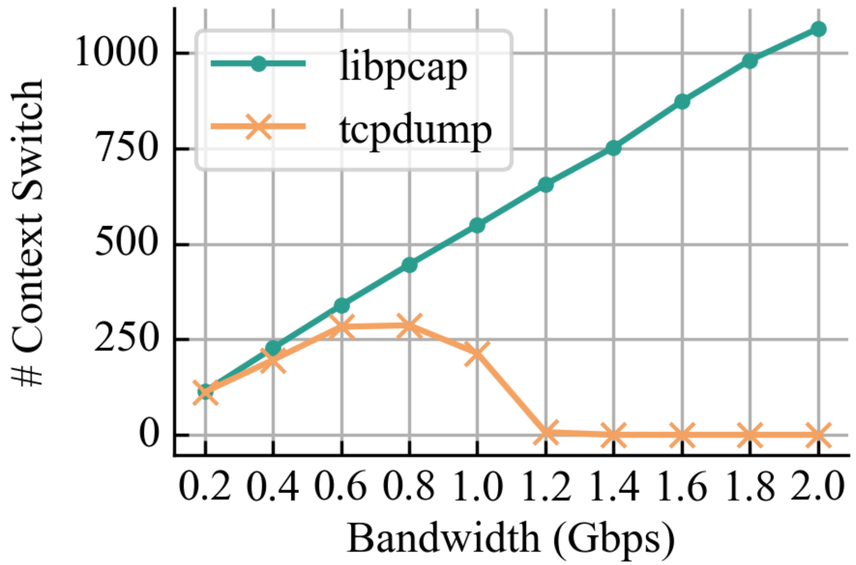


Real Time Detection in User Space



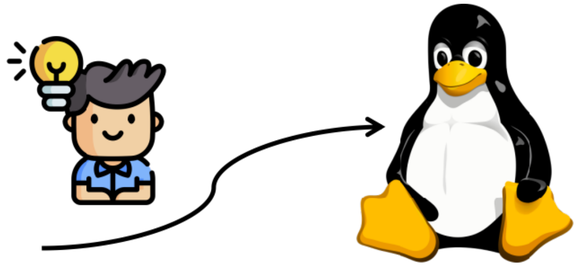
Overhead in Real Time Packet Capturing





- CPU：tcpdump (29-100%)，libpcap (2-5%)
- Context Switch: increases linearly with bandwidth, leading to packet loss

Offloading Detection into Kernel ?

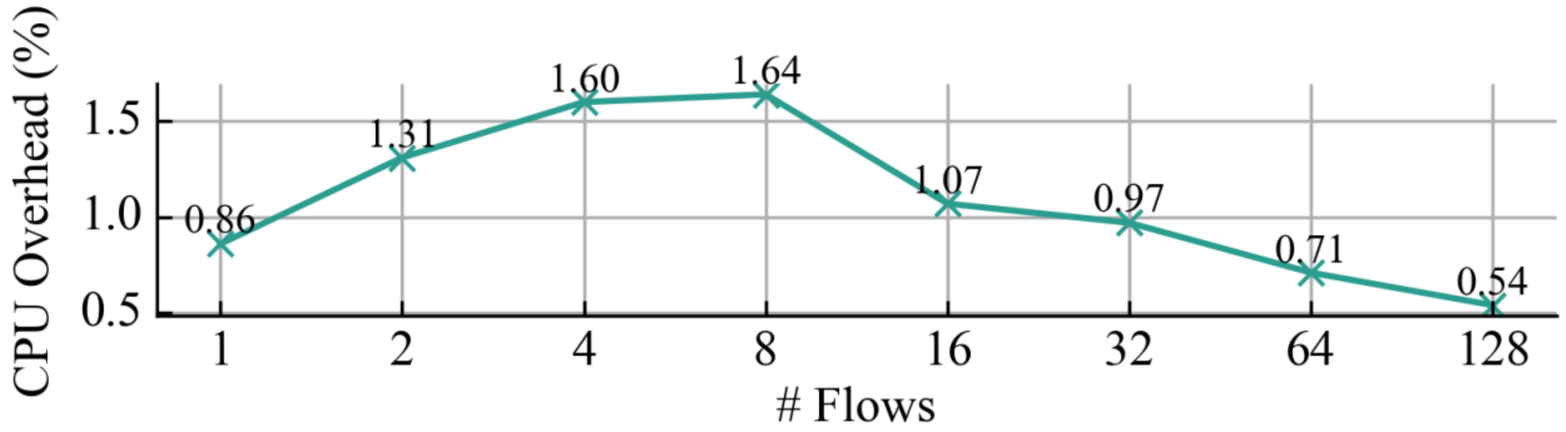


- Benefits:
  - Detection in-place in the kernel upon receipt of a packet
  - Reducing the overhead of user-space packet capturing

eBPF enables user-defined and sandboxed execution of programs in kernel

  - eBPF is widely used in network packet filtering (Google, Netflix, …)
  - A Step Further: detection in kernel after packet filtering

eBPF + Neural Network in Kernel



- 1-128 concurrent flows, transmitting at maximum rate
- CPU: 0.54-1.64%, including packet capturing, feature extraction, and inference
- Memory: 5KB, including neural network parameters

We are not the first…

Related Work: eBPF + ? In kernel

On Practicality of Kernel Packet Processing
Empowered by Lightweigh
Decision

Takanori Hara
Graduate School of Science and Technology
Nara Institute of Science and Technology
Ikoma, Japan
hara@ieee.org

***Abstract*—Kernel packet processing such as extended Berkeley Packet Filter (eBPF) and eXpress Data Path (XDP) is a promising framework that can speedily/efficiently process packets without passing them to conventional packet processing software running on the user space. Several studies pointed out the possibility of eBPF empowered by simple machine learning techniques (e.g., decision tree (DT)) to realize intelligent packet processing (e.g., intrusion detection) in the kernel space. Note that the quantitative evaluation of both packet processing and detection performance has not been conducted sufficiently. In addition, to ensure the kernel stability and safety, the eBPF program must process packets under strict constraints such as prohibition of floating-point number, which is usually used in neural networks (NNs). In this paper, we examine the possibility of NN-empowered eBPF/XDP based packet processing. More specifically, we first train a floating-point NN and quantize it as a fixed-point NN using 8-bit integers in the user space. Then, we implement the lightweight NN in the eBPF/XDP program to achieve fast packet processing with integer-arithmetic-only inference in the**

Research Article

High-performance Intrusion Detection Systemusing eBPF with Machine Learning algorithms

NEMALIKANTI ANAND, SAIFULLA M A, F

https://doi.org/10.21203/rs.3.rs-3140072/

Abstract

Denial of Service (DoS) and Distribut
on network services face. Detecting
damage caused. This paperproposes
ley Packet Filter (eBPF) with machine
Support Vector Machine (SVM), and
without modifyingthe kernel source
security, and networking. Socket filte
allows for efficient filtering and manip

A flow-based IDS using Machine Learning in eBPF

Maximilian Bachl, Joachim Fabini, Tanja Zseby
Technische Universität Wien
firstname.lastname@tuwien.ac.at

***Abstract*—eBPF is a new technology which allows dynamically loading pieces of code into the Linux kernel. It can greatly speed up networking since it enables the kernel to process certain packets without the involvement of a userspace program. So far eBPF has been used for simple packet filtering applications such as firewalls or Denial of Service protection. We show that it is possible to develop a flow-based network intrusion detection system based on machine learning entirely in eBPF. Our solution uses a decision tree and decides for each packet whether it is malicious or not, considering the entire previous context of the network flow. We achieve a performance increase of over 20% compared to the same solution implemented as a userspace program.**

I. Introduction

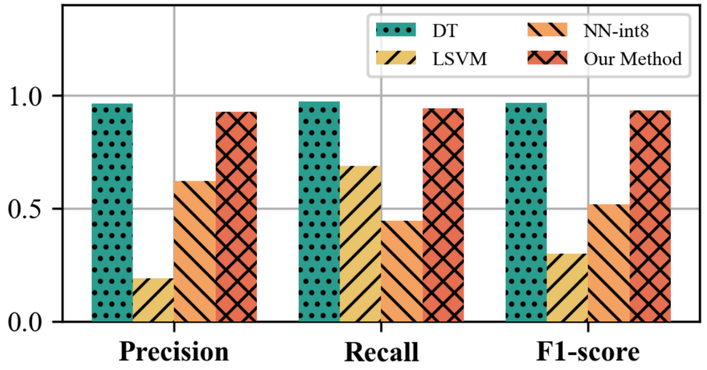
A. eBPF

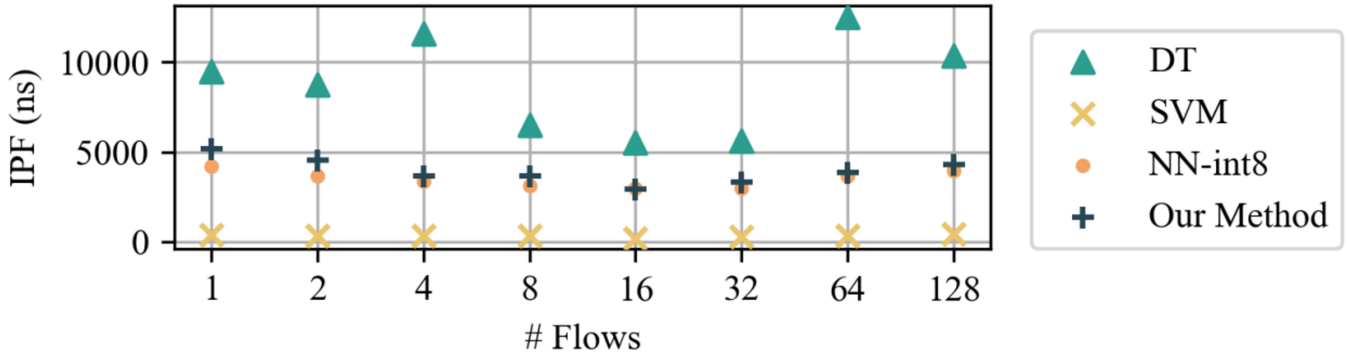
A drawback, however, is that because eBPF is verified, it can only use certain data structures. For example, an eBPF program cannot use normal C arrays since they allow out-of-bounds accesses. For example, in an array of length 10, C would allow accessing the 15th element even though the array only has 10 elements. Thus, eBPF programs make use of special data structures which are safe. However, this can potentially be a performance penalty since checking the bounds of an array each time it is accessed requires extra work to be done by the CPU.

One alternative to using eBPF is using kernel modules. However, a drawback of kernel modules is that they usually cannot be verified for stability and that they have to be compiled for a specific kernel version. Moreover, developing

Use Case: Intrusion Detection

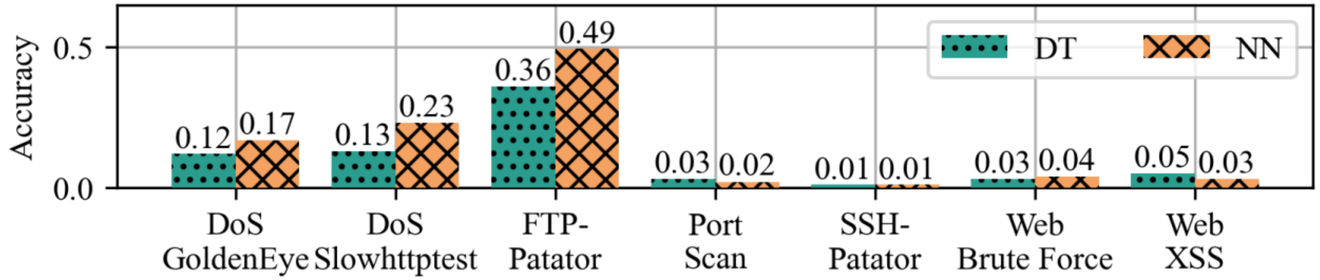
Our methods: low overhead + high performance





- Related Works: eBPF + ? In Kernel
  - Decision Tree: exponential memory overhead, uncertain inference time (IPF)
  - Support Vector Machine: insufficient detection ability
  - Neural Network with int8 quantization: quantization significantly reduces model performance

More Ignored Challenges…



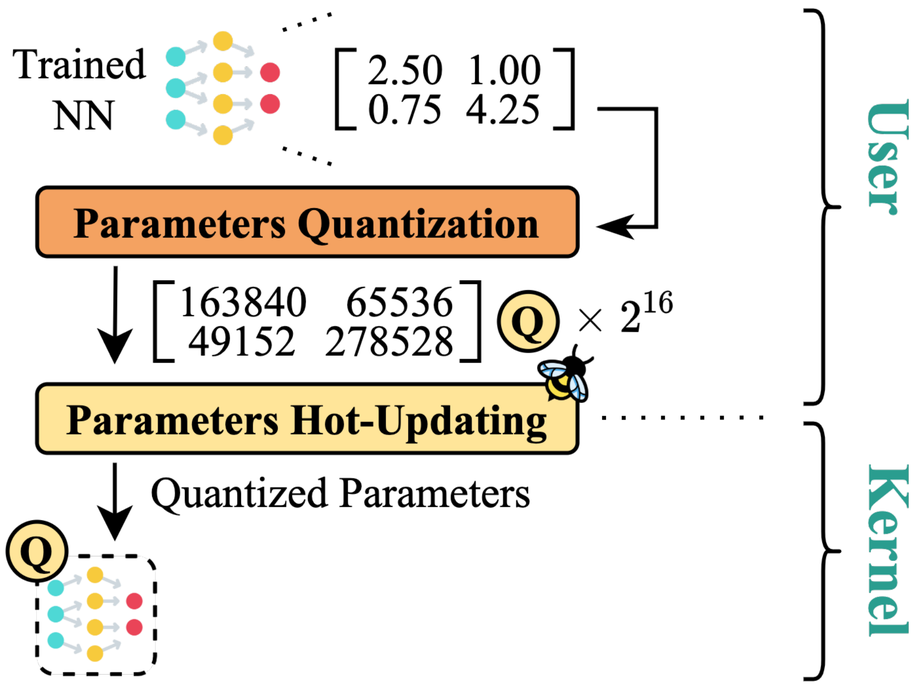
- Concept Drift:
  - Online model generates critical errors
  - Data race during online updating (hot updating)
  - Limitations and overhead of eBPF Spin Lock
- Programming limitations of eBPF:
  - 1M maximum instruction limit
  - 512B stack space

- When the lock is taken, calls (either BPF to BPF or helpers) are not allowed.
- The **BPF_LD_ABS** and **BPF_LD_IND** instructions are not allowed inside a spinlock-ed region.
- The BPF program MUST call **bpf_spin_unlock**() to release the lock, on all execution paths, before it returns.
- The BPF program can access **struct bpf_spin_lock** only via the **bpf_spin_lock**() and **bpf_spin_unlock**() helpers. Loading or storing data into the **struct bpf_spin_lock** *lock*; field of a map is not allowed.
- To use the **bpf_spin_lock**() helper, the BTF description of the map value must be a struct and have **struct bpf_spin_lock** *anyname*; field at the top level. Nested lock inside another struct is not allowed.

Our Methods

eBPF + Neural Network (NN) in kernel

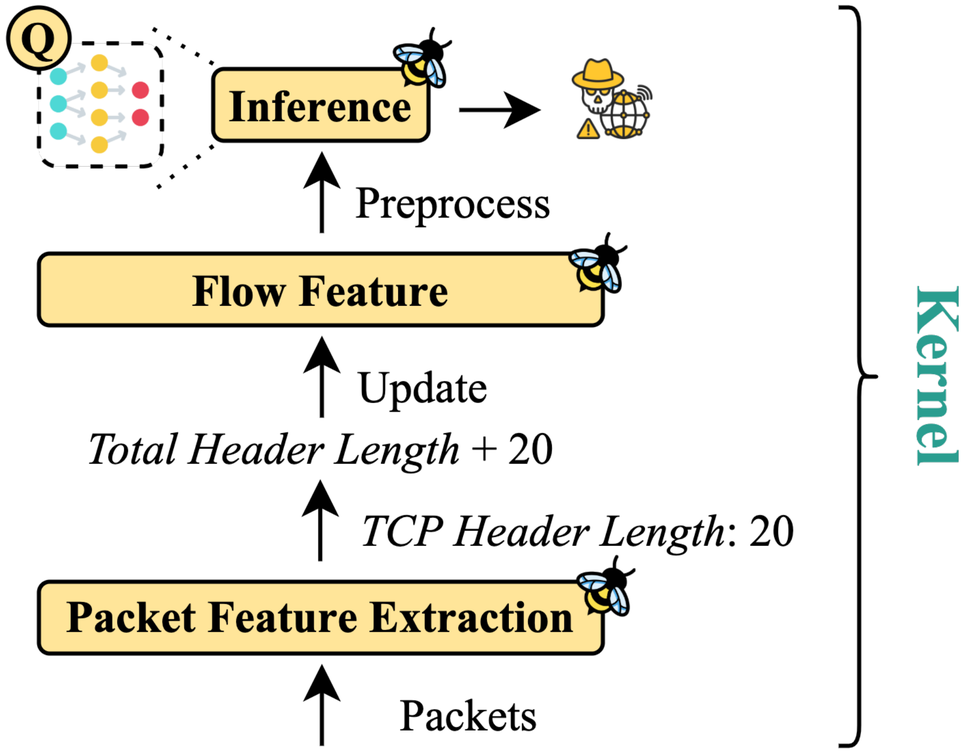
- NN Training & Parameters Quantization:
  - Training: use frameworks such as Pytorch to train NN
  - Quantization: convert floating-point numbers to integers
- Parameters Hot Updating:
  - Solve the read and write race condition problem in hot updating parameters from user to kernel



Our Methods

eBPF + Neural Network (NN) in kernel

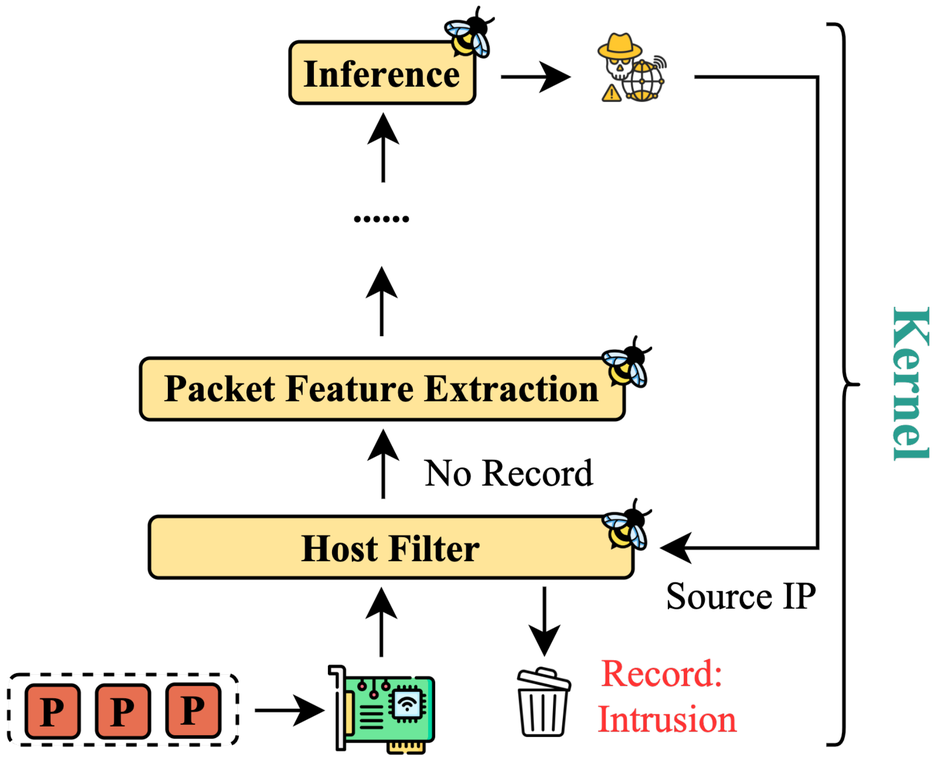
- Packet Feature Extraction & Flow Feature
  - Extract packet features
  - Update flow features
- NN Inference:
  - After a flow ends
  - Implementation using only integers



Our Methods

eBPF + Neural Network (NN) in kernel

- Host Filter
  - Drop packets from hosts previously identified as anomalous
  - NN detection results are recorded in Host Filter



Our Methods

NN Inference in Kernel

- Multilayer Perceptron (Linear + ReLU)
- eBPF does not support floating-point numbers:
  - Basic Idea: use fixed-point numbers to store floating-point numbers
  - Model Parameters:

$$\boldsymbol{W}_E^{(k)} \triangleq round(s \cdot \boldsymbol{W}^{(k)}) = [round(s \cdot w_{ij}^{(k)})]$$

Our Methods

NN Inference in Kernel

- Preprocessing Quantization: eBPF does not support signed division

$$x_j^{(1)} = \begin{cases} 0 & \sigma_j = 0 \\ -round(\frac{s \cdot (\mu_j - x_j)}{\sigma_j}) & x_j < \mu_j, \sigma_j \neq 0 \\ round(\frac{s \cdot (x_j - \mu_j)}{\sigma_j}) & x_j \geq \mu_j, \sigma_j \neq 0 \end{cases}$$

Our Methods

NN Inference in Kernel

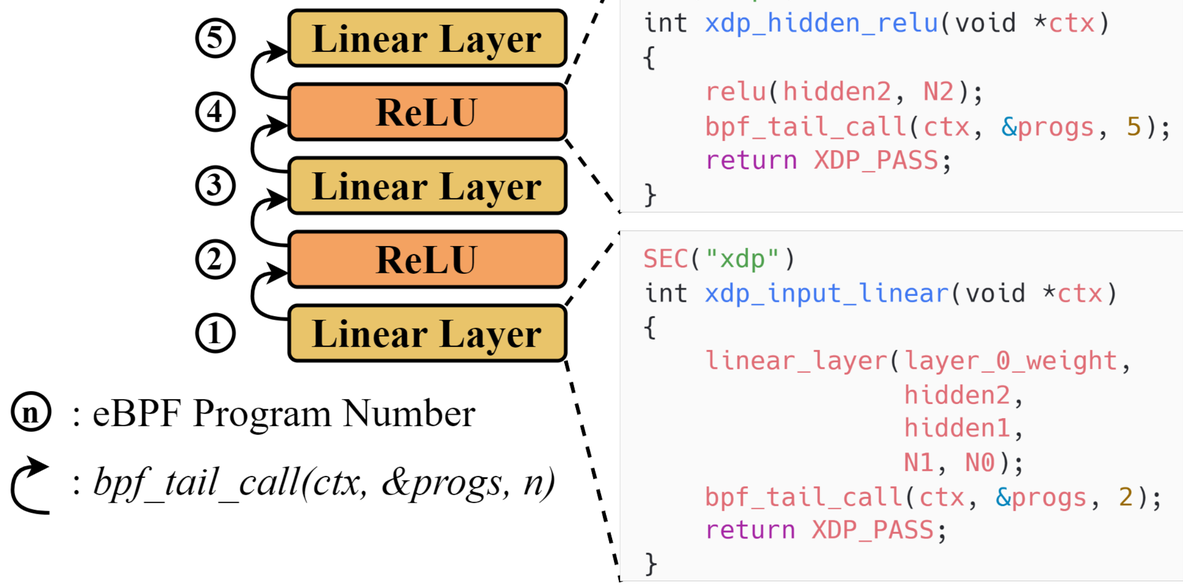
- Linear Layer: When $s = 2^b$, division can be achieved by shifting right by b bits.

$$\boldsymbol{y}^{(k)} = \frac{1}{s} \cdot \boldsymbol{W}_E^{(k)} \cdot \boldsymbol{x}^{(k)} = \frac{1}{s} \cdot [\sum_{j=1}^{n^{(k-1)}} w_{E,ij}^{(k)} \cdot x_j^{(k)}] \overset{s=2^b}{=} ars(\sum_{j=1}^{n^{(k-1)}} w_{E,ij}^{(k)} \cdot x_j^{(k)}, b)$$

- ReLU:

$$\boldsymbol{x}^{(k)} = relu(\boldsymbol{y}^{(k-1)})$$

Our Methods

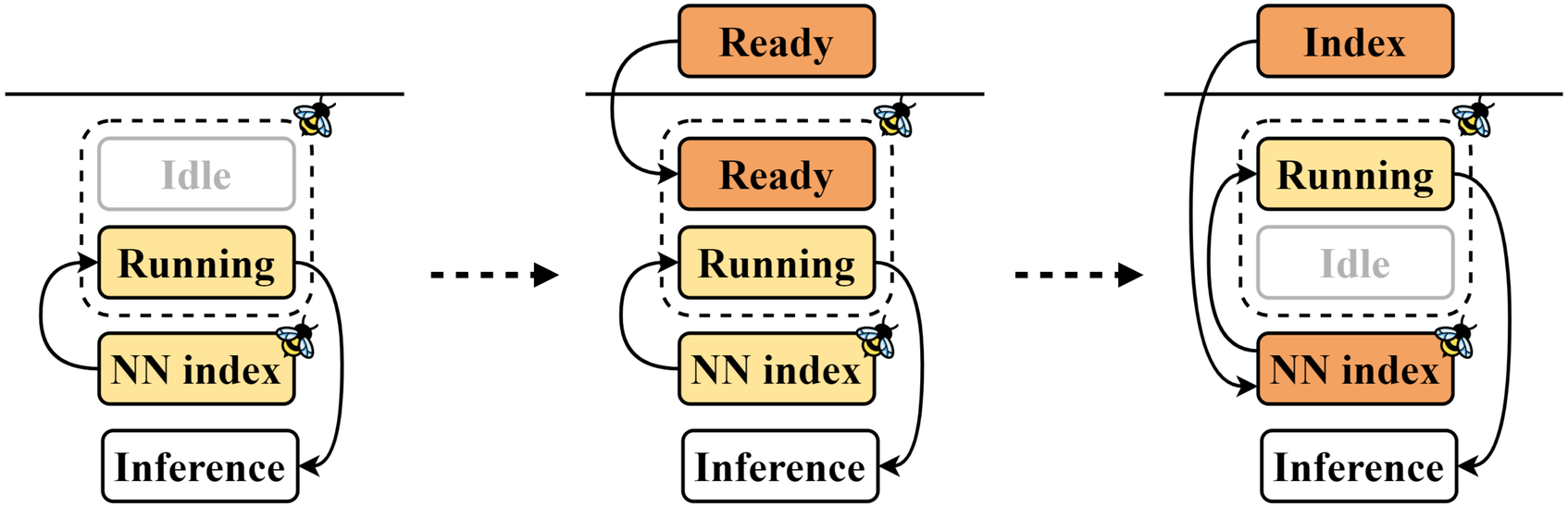



```c
SEC("xdp")
int xdp_hidden_relu(void *ctx)
{
    relu(hidden2, N2);
    bpf_tail_call(ctx, &progs, 5);
    return XDP_PASS;
}
```

```c
SEC("xdp")
int xdp_input_linear(void *ctx)
{
    linear_layer(layer_0_weight,
                 hidden2,
                 hidden1,
                 N1, N0);
    bpf_tail_call(ctx, &progs, 2);
    return XDP_PASS;
}
```

- Three-Layer MLP of [6,32,32,2] is rejected by the eBPF verifier
- Instruction Numbers Limit : MLP = Linear Layer + ReLU chained tail calls
  - Each layer is an eBPF program.
  - Each layer is called by the previous layer through eBPF tail call.

Our Methods



- Hot Updating: two-phase loading, without explicit eBPF Spin Lock
  1. NN Updating: loading new parameters (Ready) from User to Kernel (Idle)
  2. Index Updating: update the pointer to the parameters in use (Running) to new parameters (Ready)

Conclusion

- Offloading the detection model into kernel can greatly reduce the overhead of user-space packet capturing
- eBPF limitations can be addressed using fixed-point numbers for inference, two-phase loading strategies, and chained tail calls
- NN in kernel can reduce detection time and memory usage while maintaining the same performance as the state-of-the-art methods

Thank You !

Follow up questions

nelsoncheungsmile@gmail.com