eBPF Offload to Hardware cls_BPF and XDP

Nic Viljoen, DXDD (Based on Netdev 1.2 talk)
November 10th 2016

What is eBPF?

A “universal in-kernel virtual machine”

- 10 64-bit registers
- 512 byte stack
- Infinite size maps
- Well defined bytecode

Stable, in-kernel JIT compiler

- X86, ARM64, PowerPC etc…

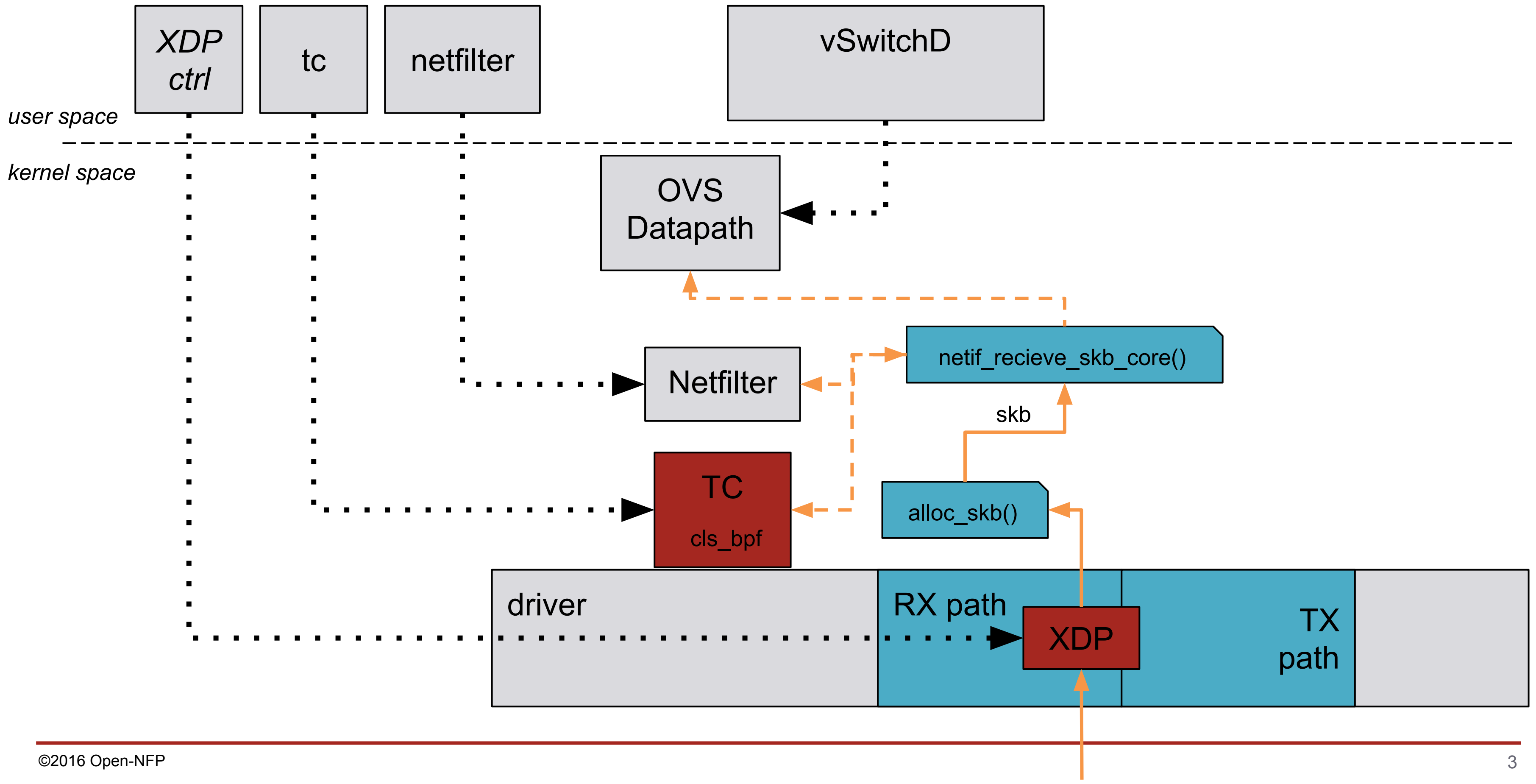
LLVM backend means that there is well optimised compiler support to produce bytecode (from C), tools exist for GO, P4 and others

eBPF allows backwards jumps if it is ensured that infinite loops are not present (this is checked by the bpf verifier)

In kernel helper functions ensure that any required additional functionality is present

- E.g working with cgroups

eBPF within the networking stack (XDP/TC)



Motivation: Avoiding Whack-a-mole

Fully programmable hardware **could** do anything:

- However fully custom code for custom purposes is not scalable or sustainable
- Offloading kernel code transparently is both
- Avoids vendor specific solutions, ensures flexibility and portability

Solving the problem of **how** to do programmable offloads, rather than **what** to offload avoids whack-a-mole

Why eBPF?

It is a well defined framework

- Parameters are well constrained-registers, memory use, instruction set, helpers etc.

Implementation is ideal for multi-core architectures

- Good for transparent offload to many-core NPU

Programming method used in XDP and TC

- The most likely targets for this type of general networking offload

Gaining Traction

Target architecture (NFP based SmartNIC)

A group of fully programmable Flow Processing Cores (FPCs)

Arranged into islands containing 12 FPCs

Combined with specialist islands containing FPCs + hardware blocks

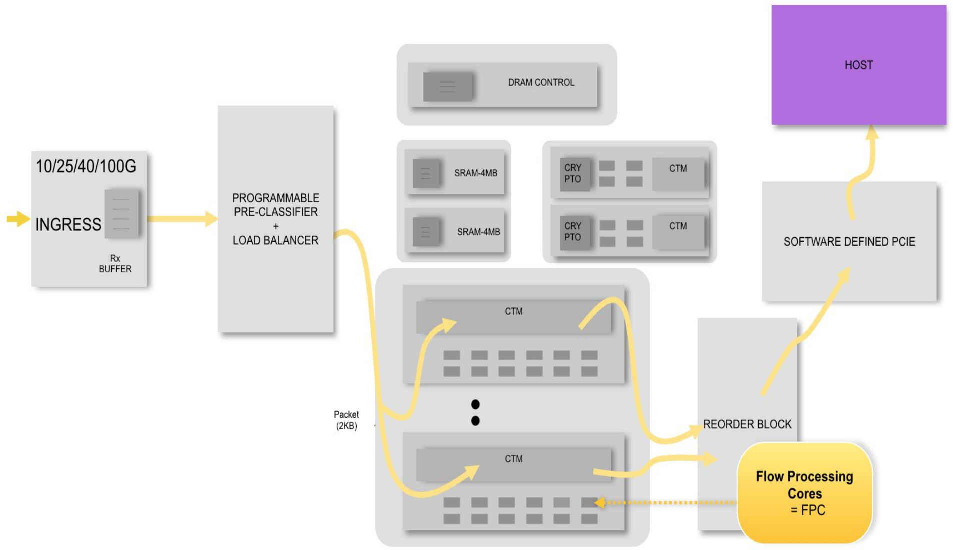
In total between 72-120 FPCs

320KB SRAM memory per island

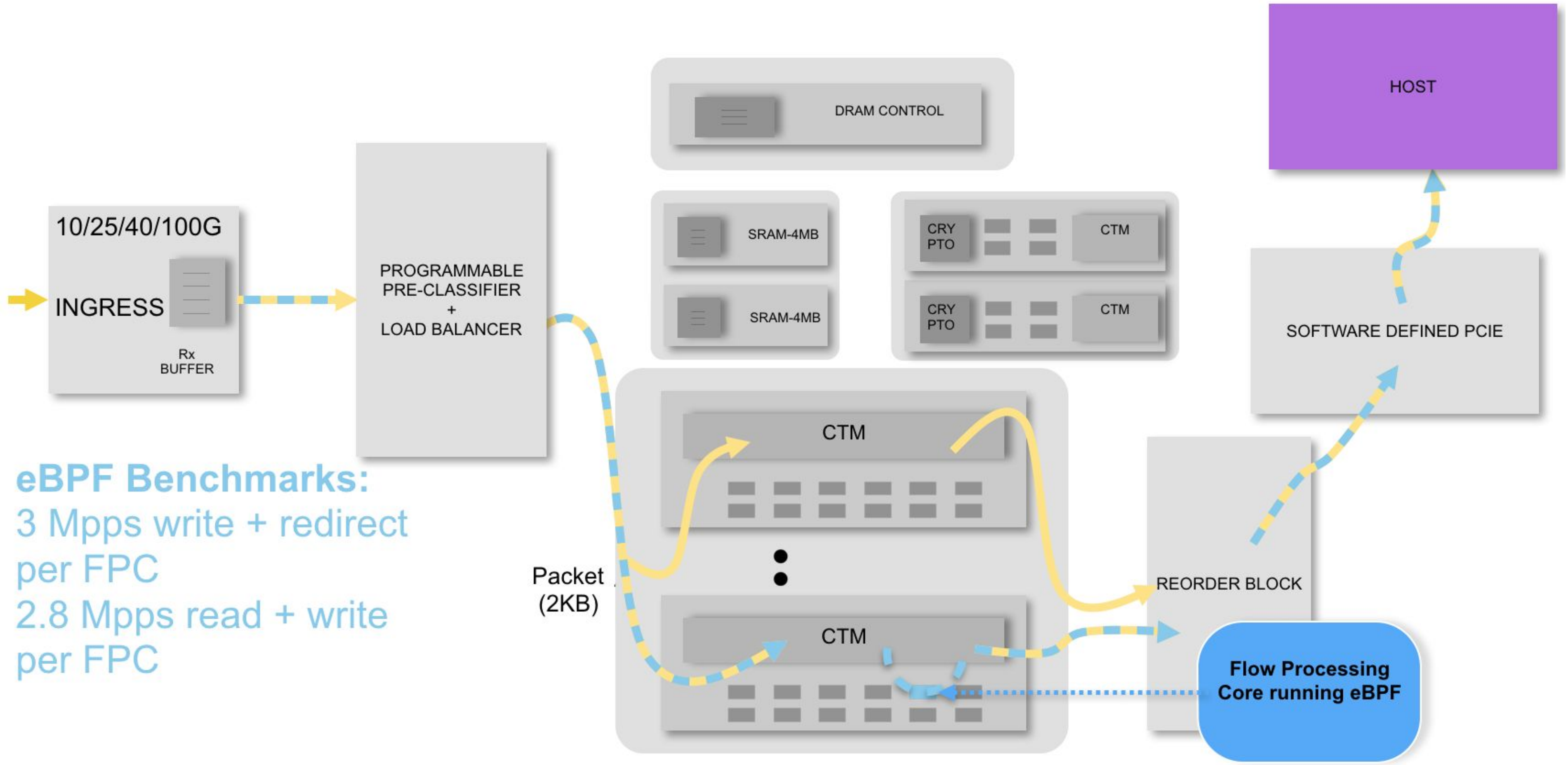
- 64 KB CLS-Cluster Local Scratch
- 256 KB CTM-Cluster Target Memory

8 MB of SRAM per NIC

2 GB of DRAM per NIC



Flow of eBPF Packet



eBPF Benchmarks:
3 Mpps write + redirect per FPC
2.8 Mpps read + write per FPC

The Flow Processing Core (Fully Programmable)

Each FPC has the ability to run 4 or 8 cooperatively multiplexed threads

128 32bit A registers and 128 32bit B registers

- A bank registers can only interact with B bank registers

256 32-bit transfer registers

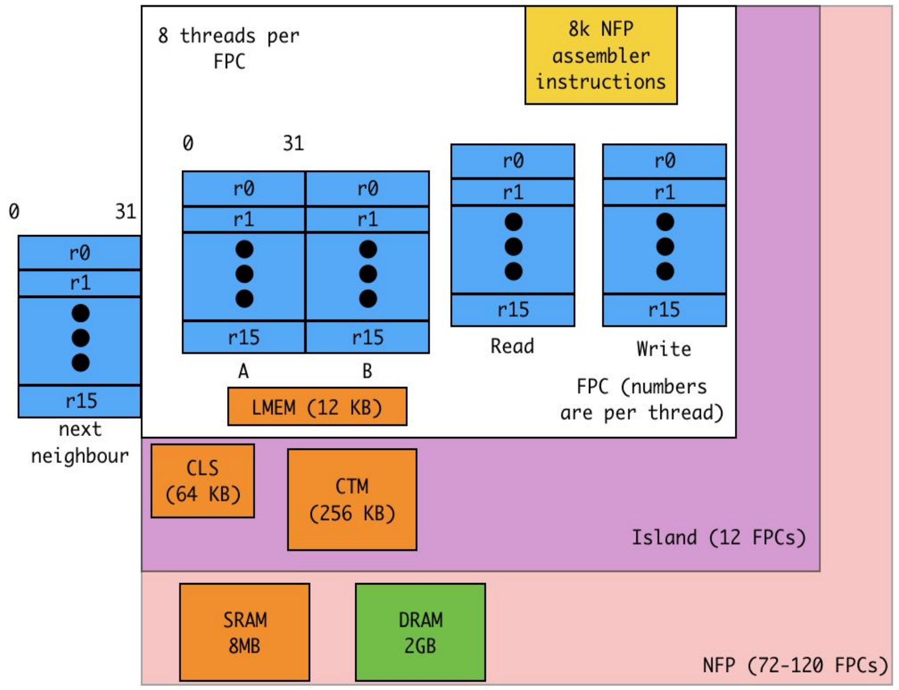
128 32-bit next neighbour registers

All of these registers are divided between the 4 or 8 contexts

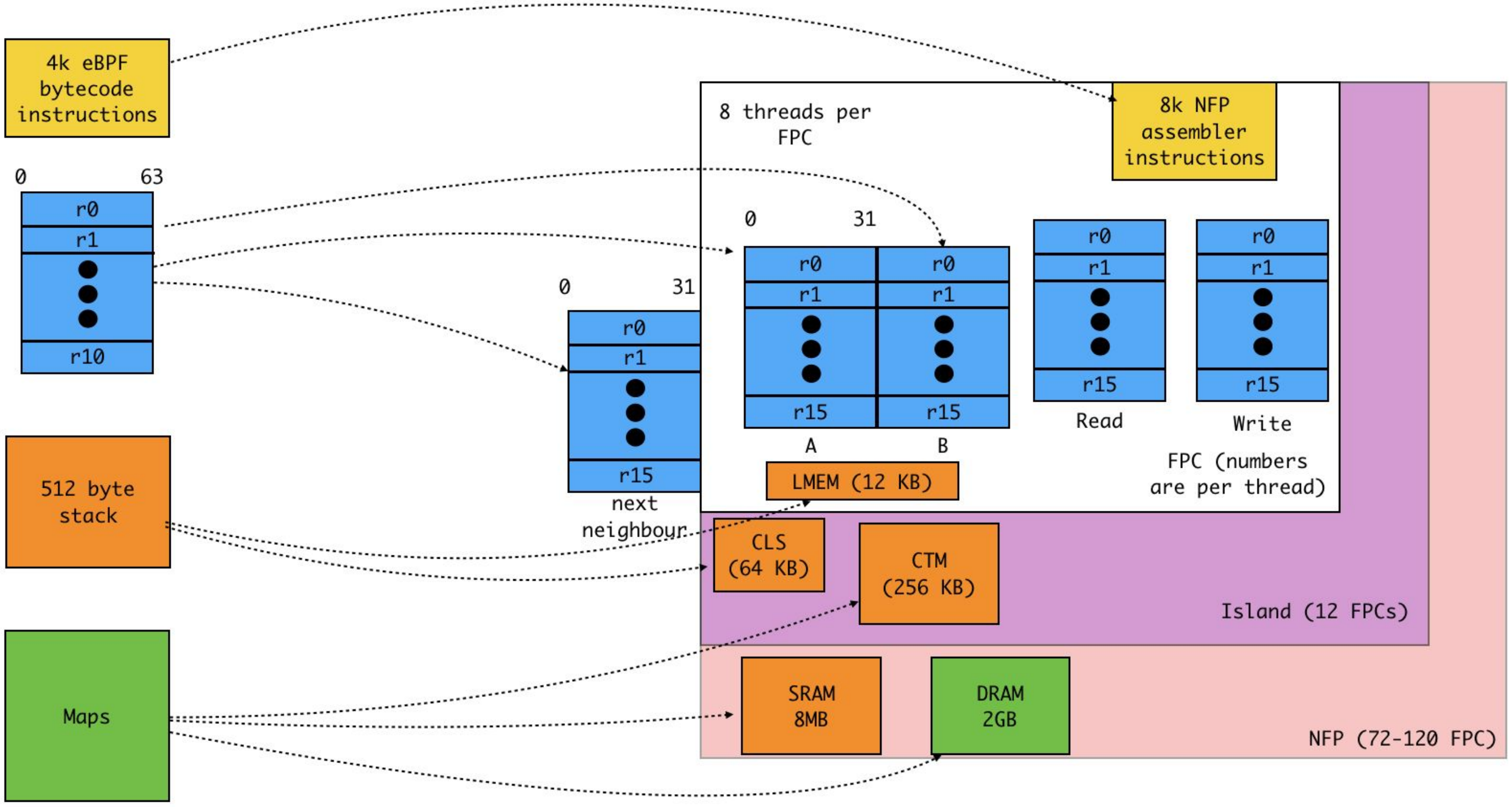
- so 8 context leads to 16 A registers, 16 B registers etc. per thread

8k of instructions per FPC

12 KB of SRAM per FPC



Mapping eBPF->NFP



Programming Model

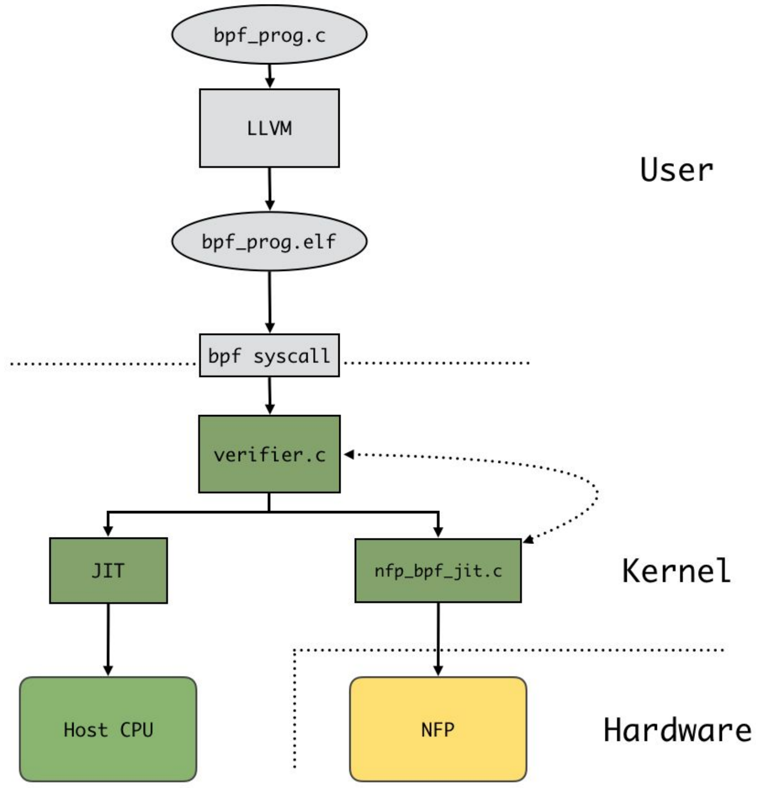
Program is written in standard manner

LLVM Compiled as normal

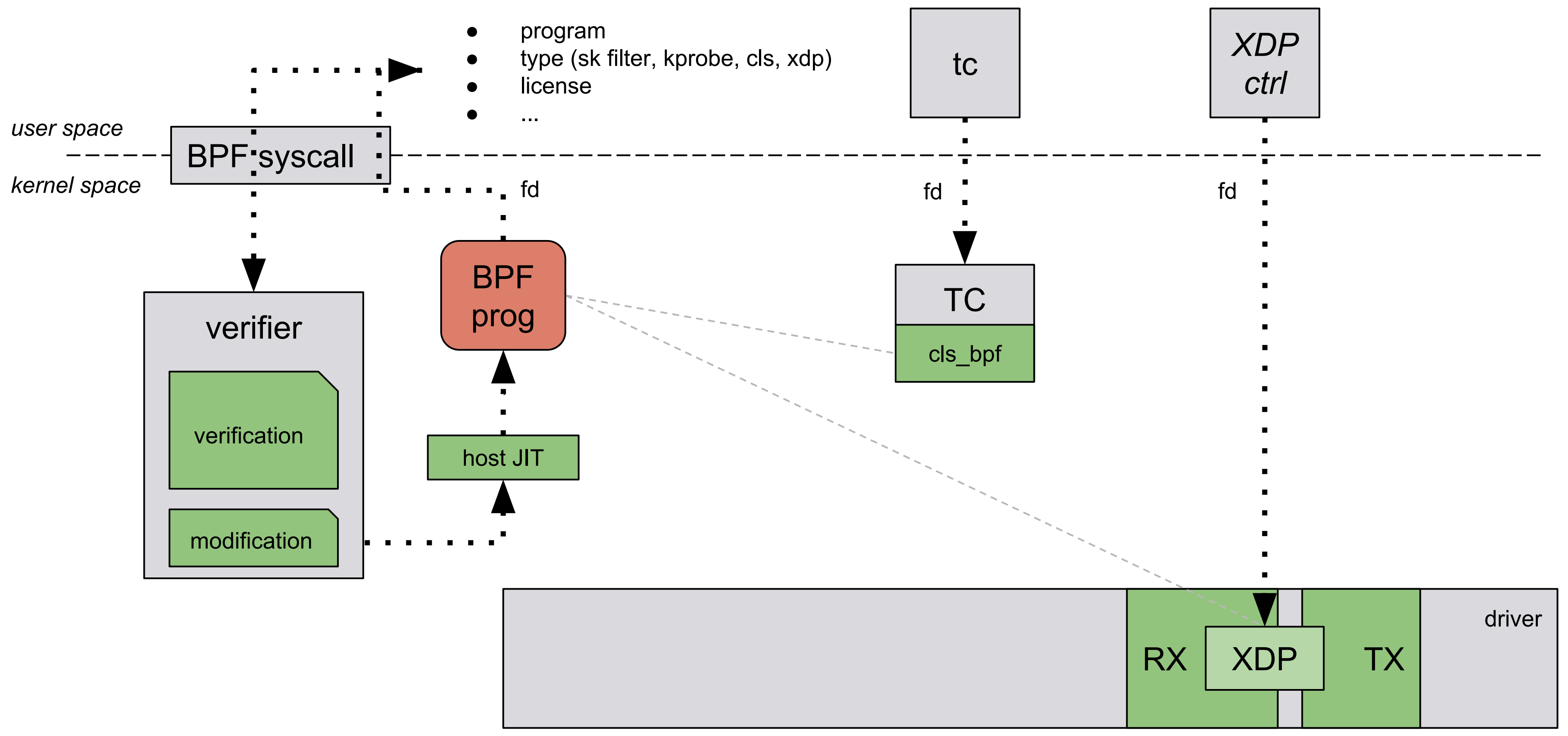
Then the *nfp_bpf_jit.c* converts the eBPF bytecode to NFP machine code

Translation reuses a significant amount of verifier infrastructure

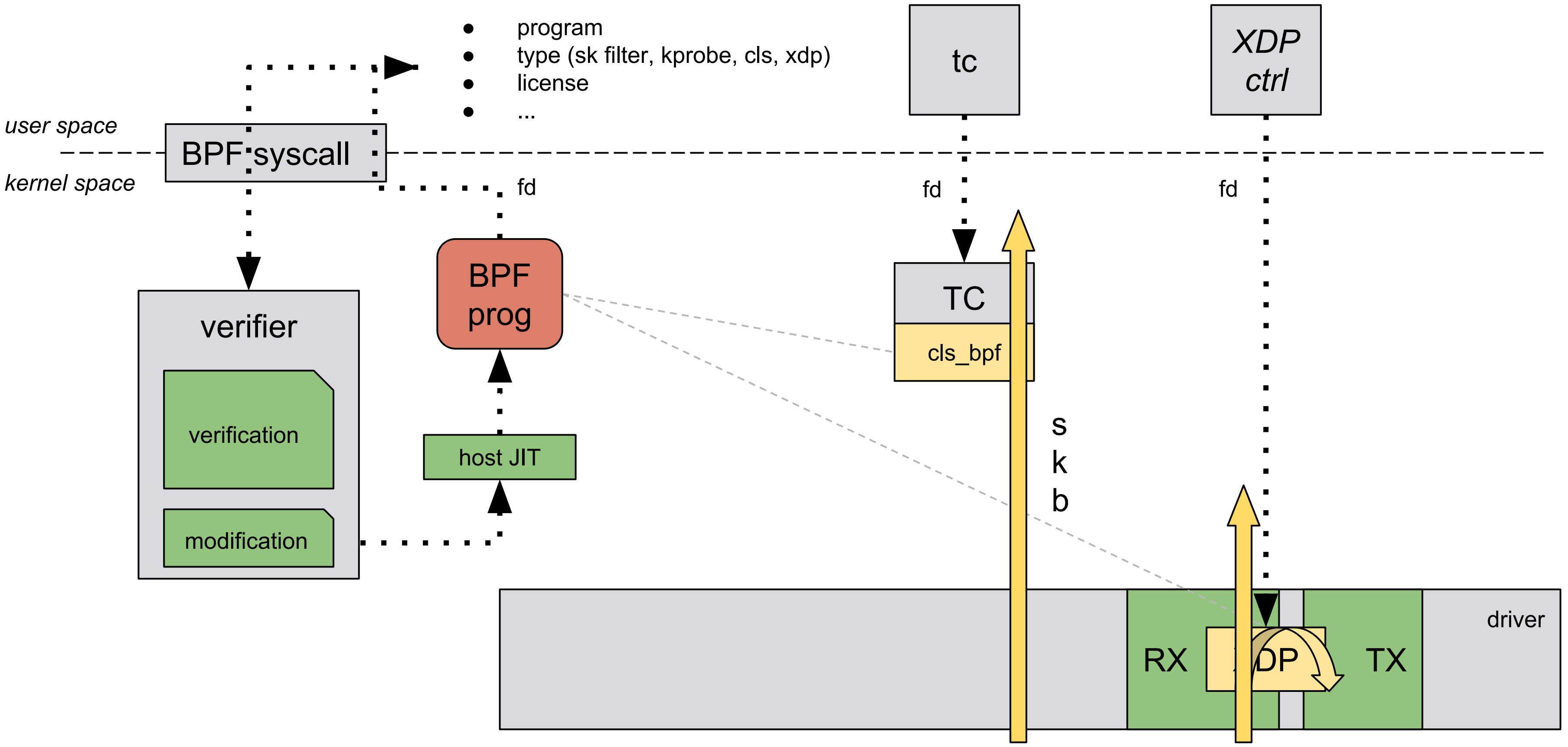
- This has motivated recent actions such as the creation of *bpf_verifier.h*



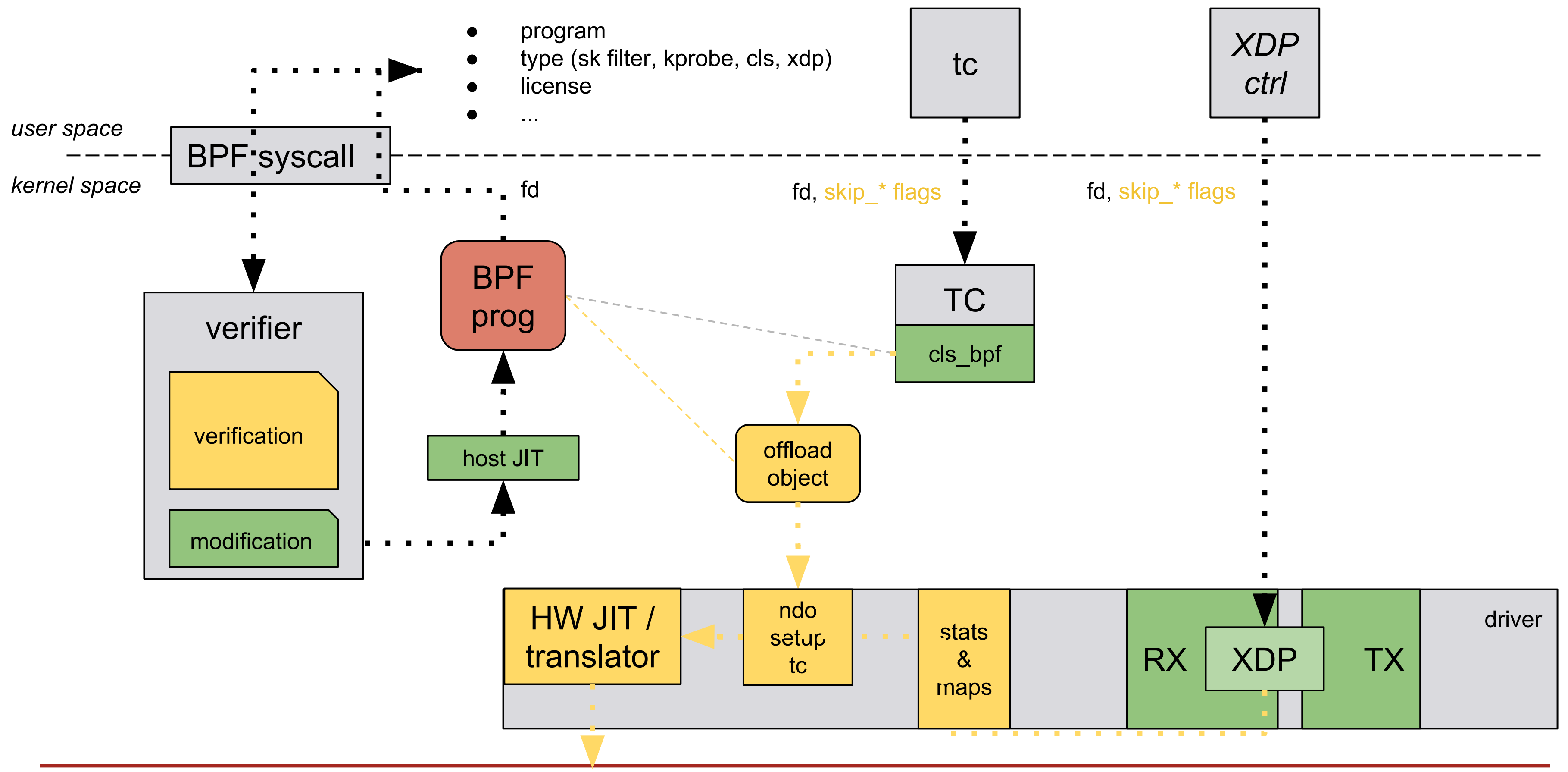
Kernel basics (before)



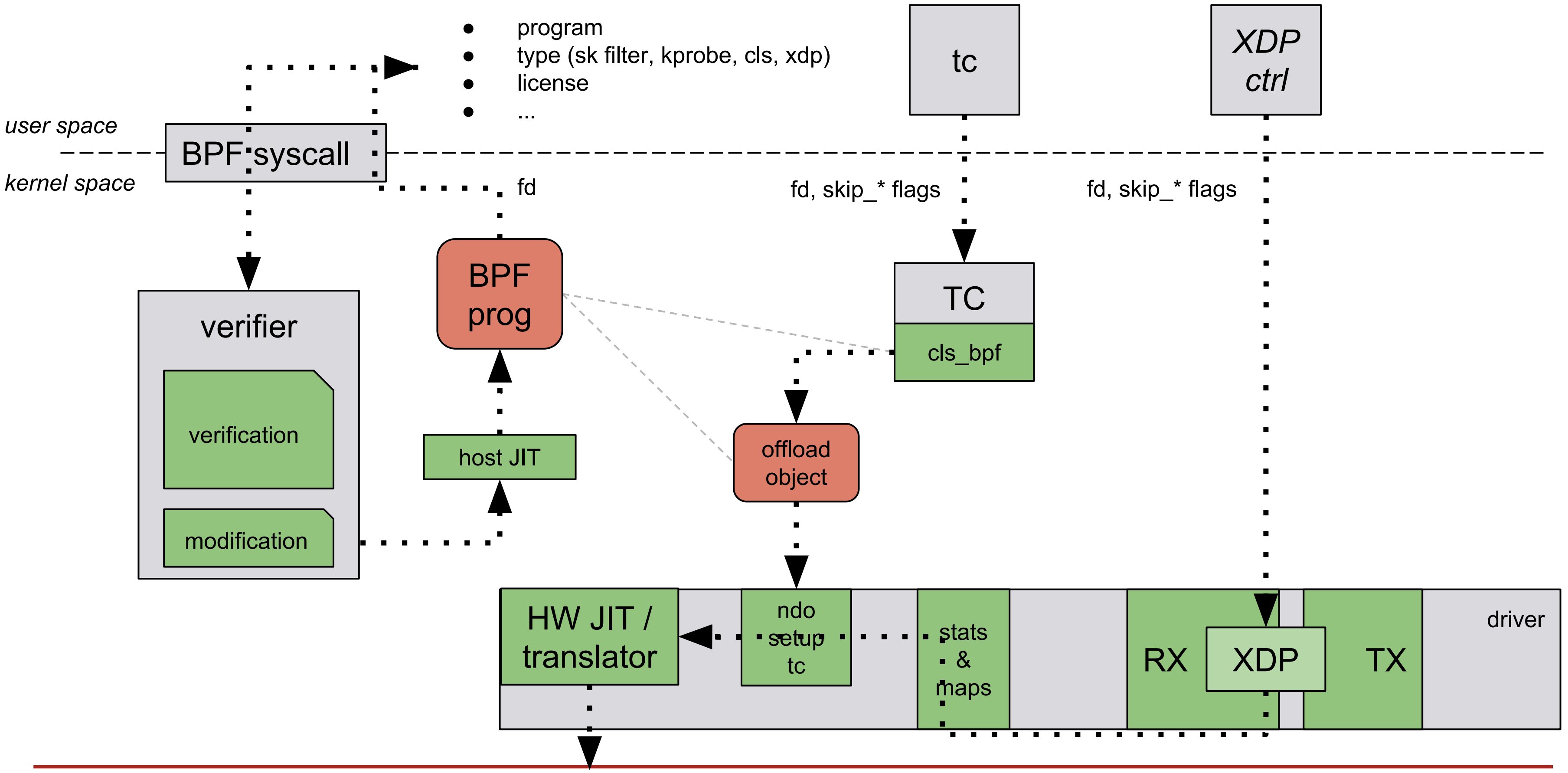
Kernel basics (flow no offload)



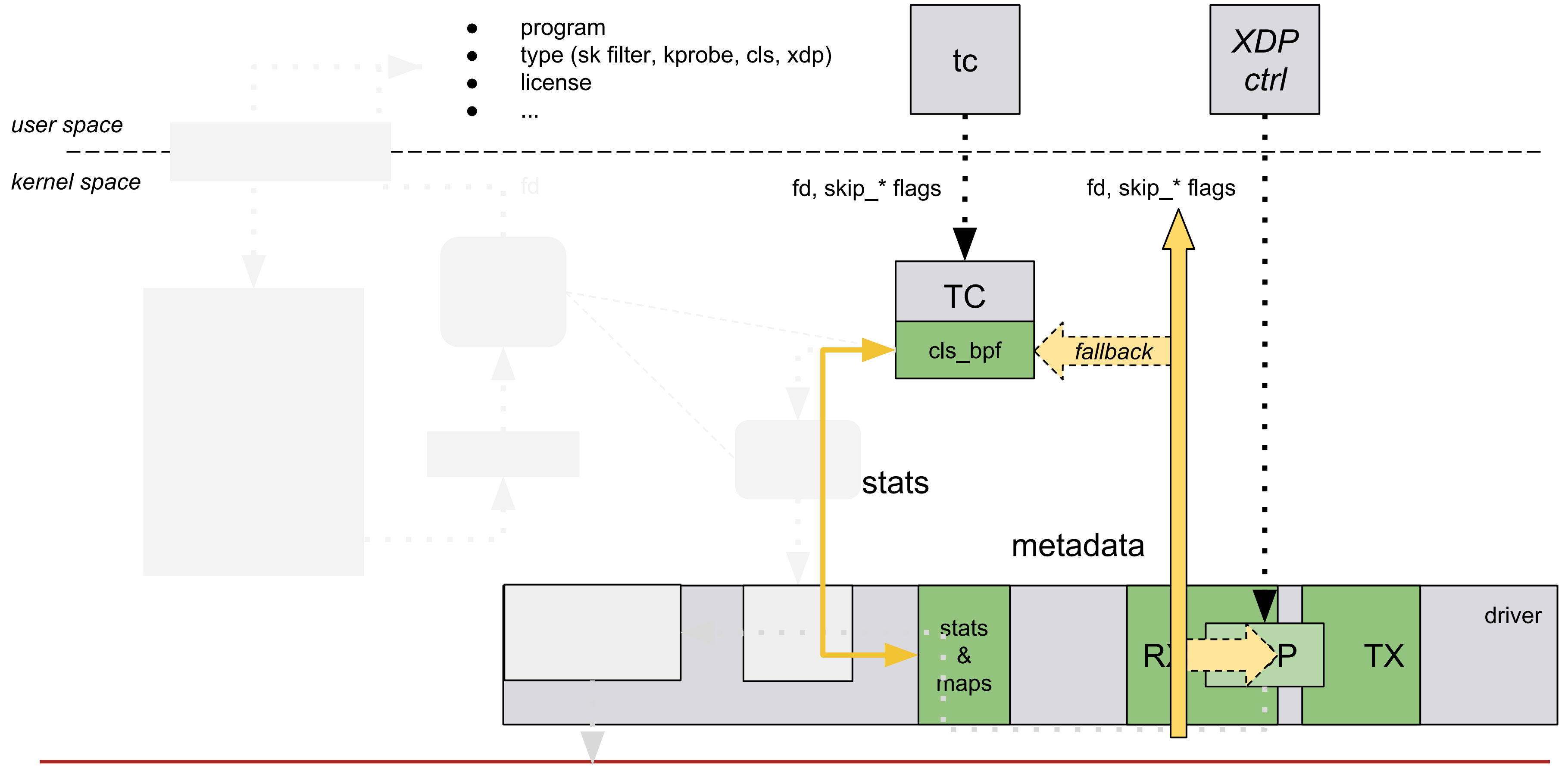
Kernel basics (after)



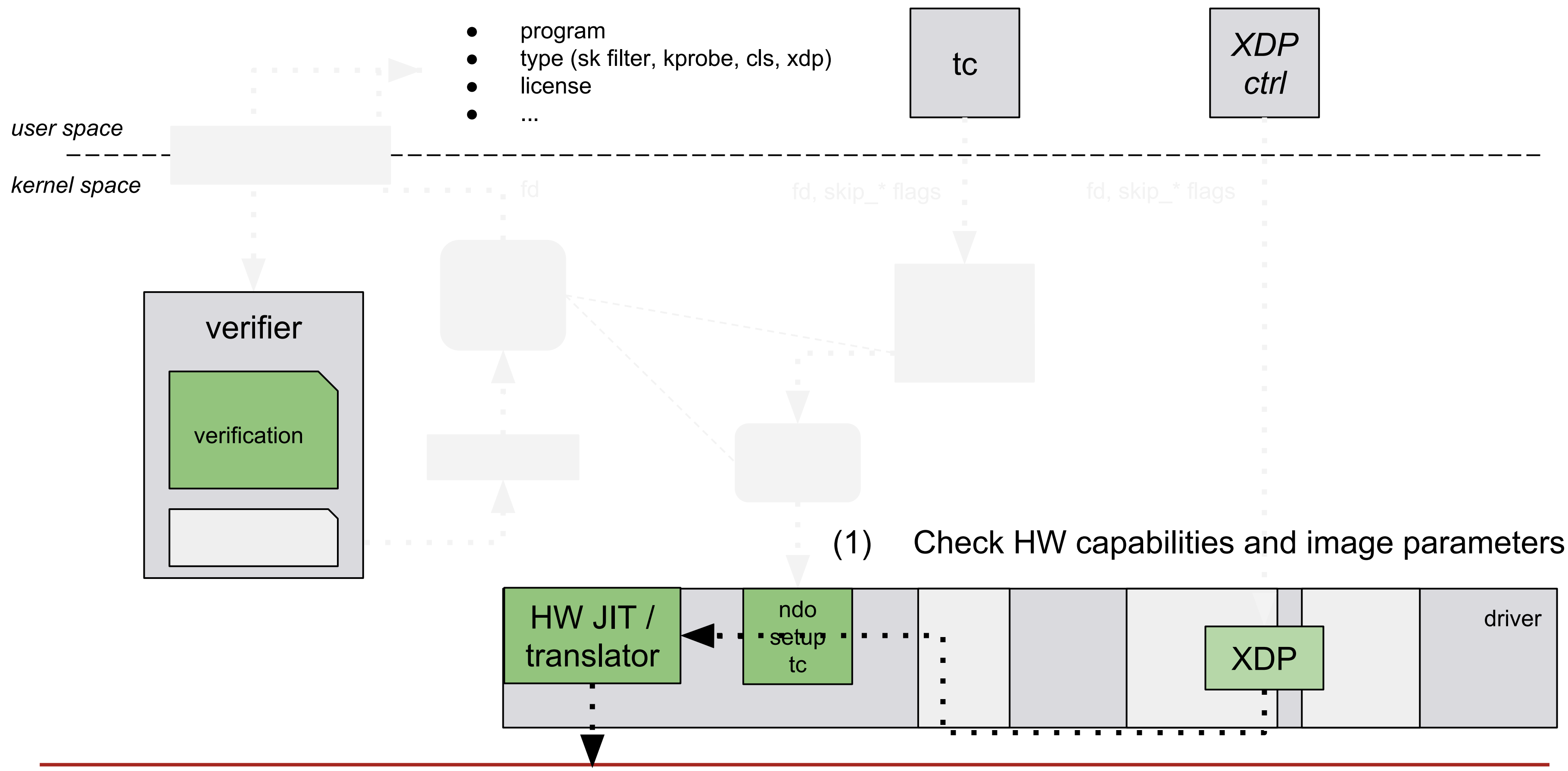
Kernel basics (after)



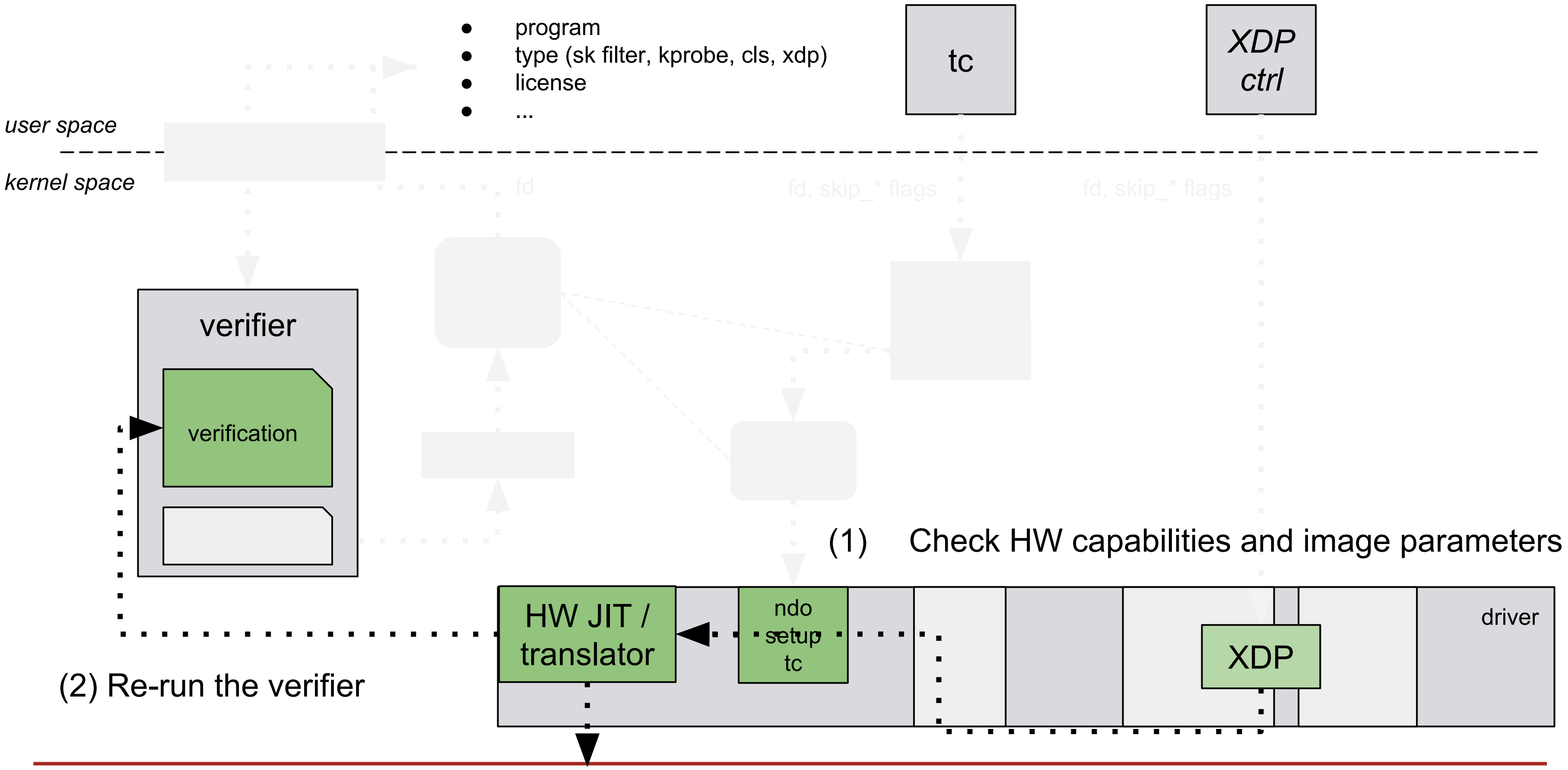
Kernel basics (flow with offload)



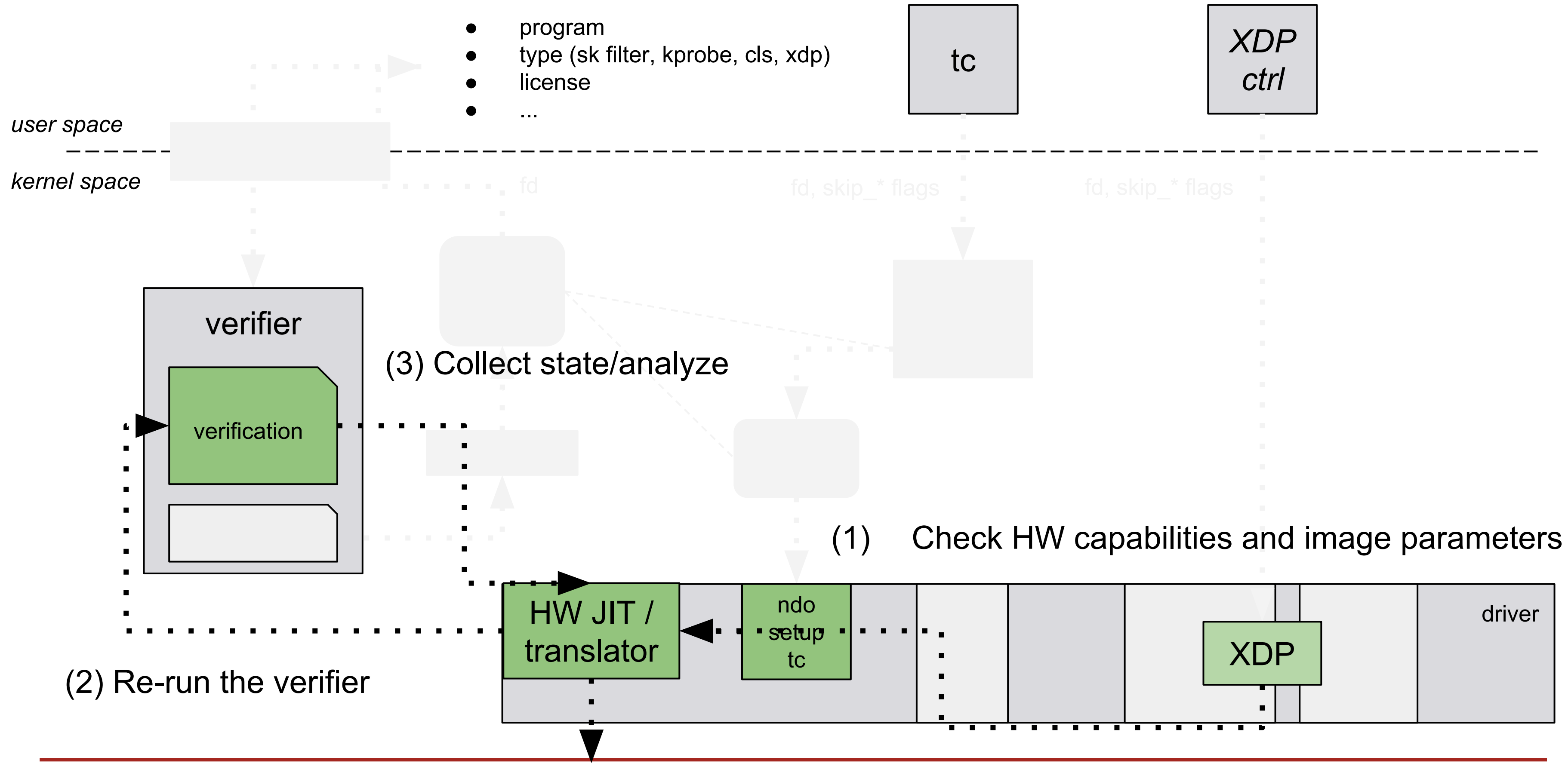
Translation and loading



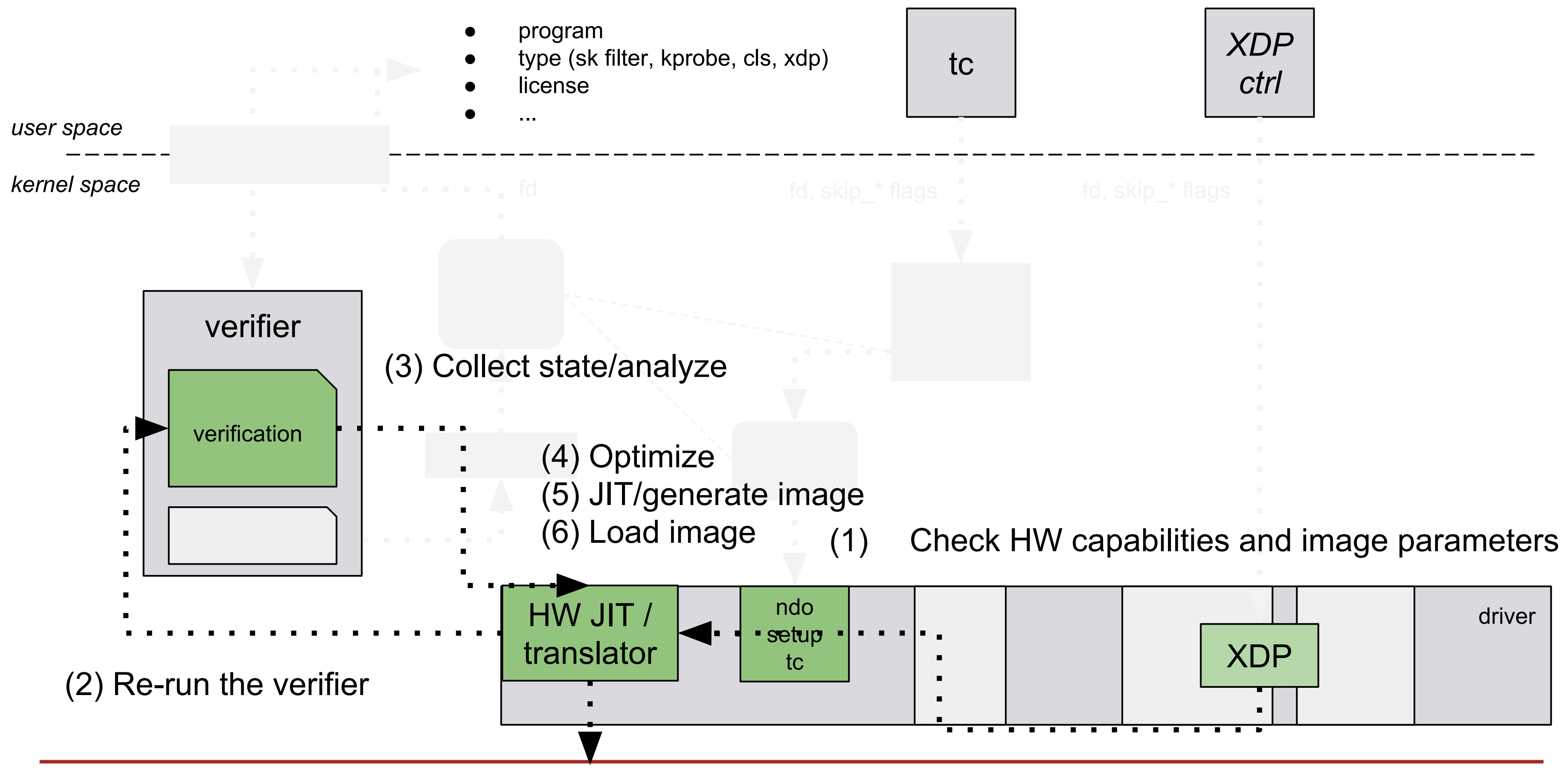
Translation and loading



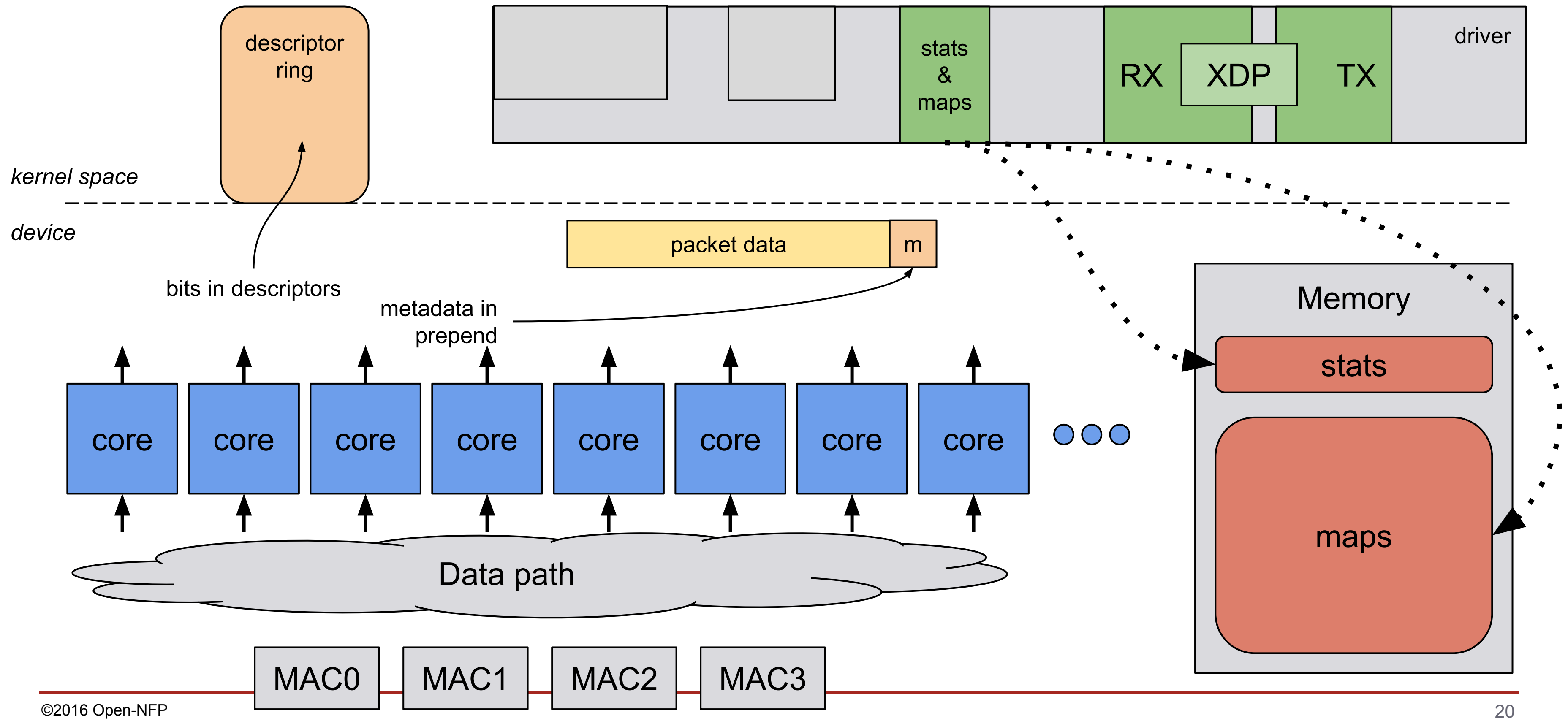
Translation and loading



Translation and loading



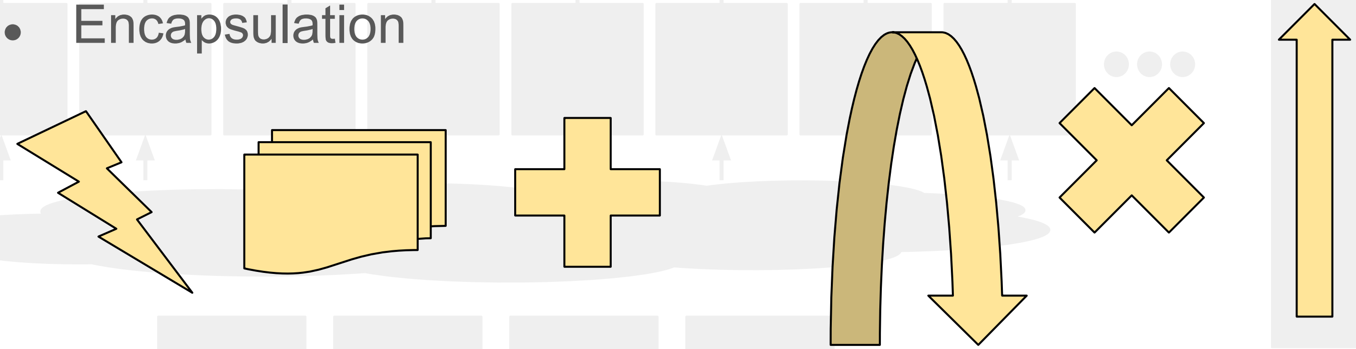
Quick peek at the device path



Operations and actions (done or near term)

- ALU instructions except multiply and divide
- Packet Modification (header or payload)
- Basic maps
- Operations on relevant packet metadata fields
- Encapsulation

- Redirection
- Drop
- Pass (with modification and metadata)



Map plans and ideas

Use of map by offloaded program	Map location	Mechanism needed
Read only	Host + device copy	update interception
Read/write	Device only	update/read interception lock out map in kernel space
Read + statistics	*gather*	update/read interception

- use verifier to check access types;
- add hooks in map code;
- add netdevice for binding the map to the device;
- *read + statistics* require further investigation;
- only allow *read/write* offload for skip-sw programs.

Optimizations and verifier work

32 bit state tracking (in kernel or LLVM machine type?).
Instruction merging (3-operand assembler, shift/mask/alu instructions).

```
BPF_ALU | BPF_SHR | BPF_K,0,0,0,5
BPF_ALU | BPF_AND | BPF_K,0,0,0,0xff  →  alu_shf[gpr1, 0xff, AND, gpr0, >>5]
BPF_ALU | BPF_MOV | BPF_X,1,0,0,0
```

Better register allocation-liveness analysis.
Clever placement/caching of registers and data (maps).

- Island SRAM access accessed in 20 cycles whereas DRAM is 150-500 (though hidden by multithreading)

Demo

Links to eBPF Webinar

Start of Webinar: https://www.youtube.com/watch?v=apU5sg0Ui5U

Start of Demo: https://youtu.be/apU5sg0Ui5U?t=2003

Also Check out: http://open-nfp.org/the-classroom/

Summary

cls_bpf and XDP are fast and efficient classifiers

However as time goes on, efficient use of CPU will become more important as networking workloads scale relative to CPU

To ensure that networking is able to cope without exponential increases in CPU usage requires the implementation of an efficient and transparent general offload infrastructure in the kernel

We believe this work is a step in the right direction

Thank You

Questions?