Demystify eBPF JIT Compiler

Jiong Wang, Netronome

September 11, 2018

© 2018 NETRONOME

Agenda

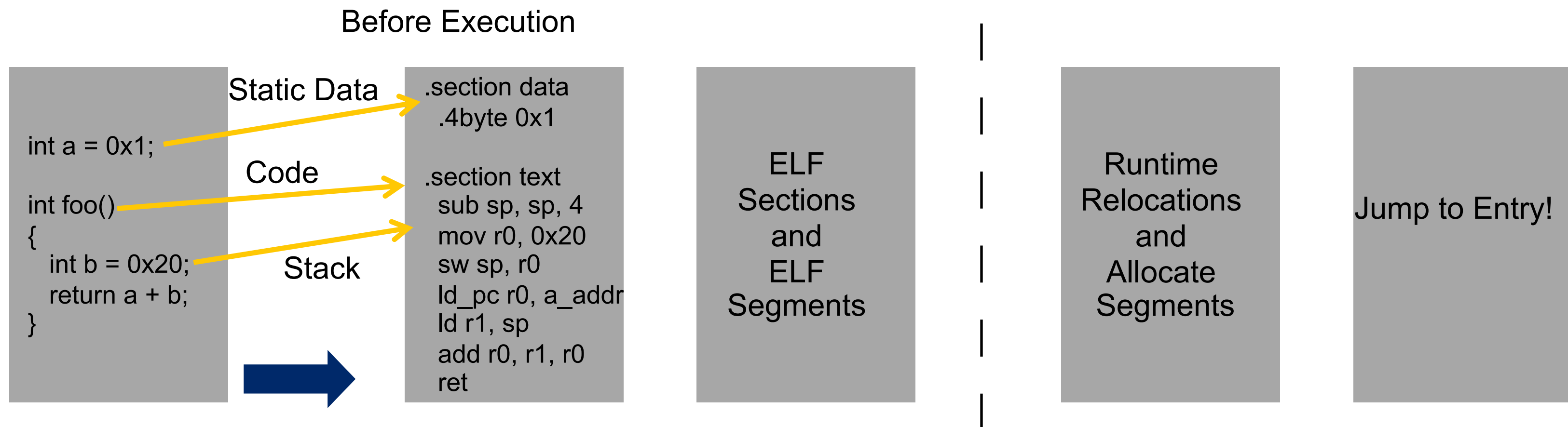
- What is JIT Compiler?
- What is eBPF JIT Compiler?
- eBPF JIT Compiler – Verification Stage
- eBPF JIT Compiler – Code Gen Stage
- Netronome Flow Processor (NFP) Code Gen Back-end
- eBPF JIT Compiler Emerging Features

© 2018 NETRONOME

NETRONOME

A Traditional Static Compiler

- Translates source language to machine instructions before execution
- Good fit for statically typed language
- When people mention compiler they might actually mean toolchain
- Compilation → Assembling → Linking → Loading → Execution

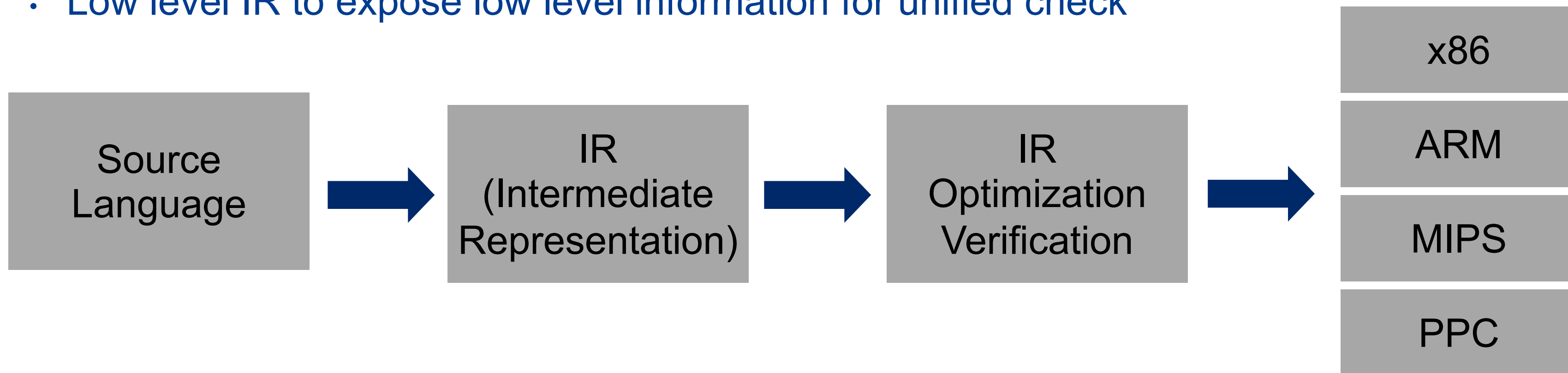


© 2018 NETRONOME

Types of Compilers (Continued)

JIT Compiler

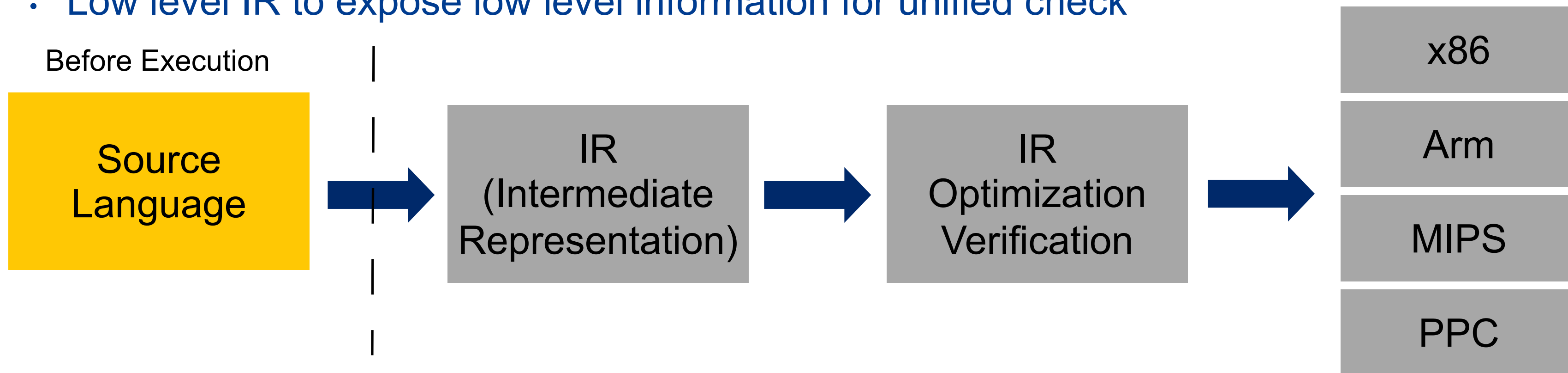
- Just In Time ⟷ During Execution ⟷ Runtime
- Machine instruction generation depends on runtime information
  - Dynamically typed language
  - Portable without re-compilation
  - Low level IR to expose low level information for unified check



© 2018 NETRONOME

JIT Compiler

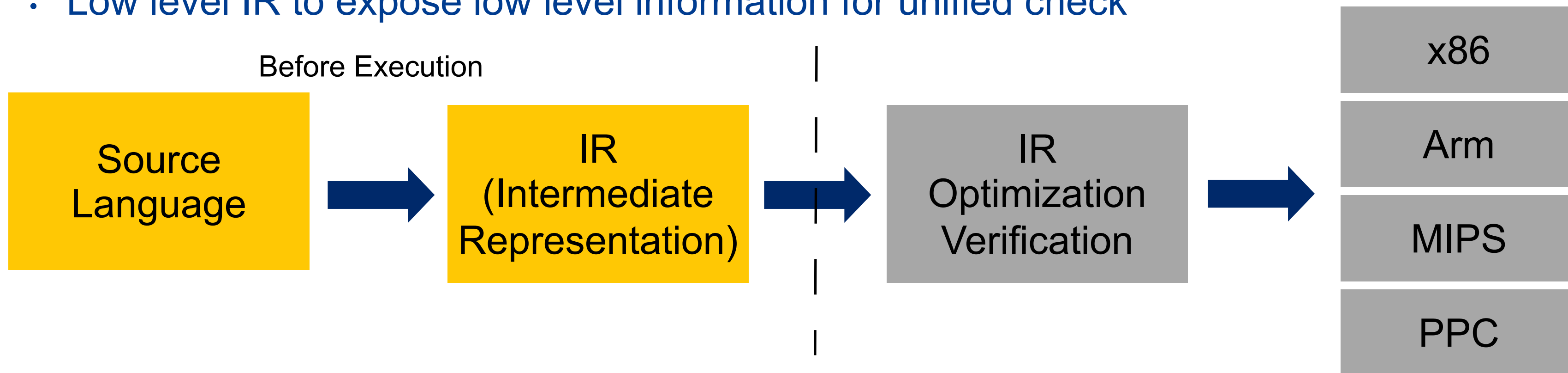
- Just In Time ⟷ During Execution ⟷ Runtime
- Machine instruction generation depends on runtime information
  - Dynamically typed language
  - Portable without re-compilation
  - Low level IR to expose low level information for unified check



© 2018 NETRONOME

JIT Compiler

- Just In Time ⟷ During Execution ⟷ Runtime
- Machine instruction generation depends on runtime information
  - Dynamically typed language
  - Portable without re-compilation
  - Low level IR to expose low level information for unified check



© 2018 NETRONOME

What is a JIT Compiler?

NETRONOME

- How does a JIT compiler really look like? Here is a very simple example.

```
int main(int argc, char *argv[]) {
  // mov eax, 0
  unsigned char mov[] = {0xb8, 0x00, 0x00, 0x00, 0x00};
  // ret
  unsigned char ret[] = {0xc3};

  int num = atoi(argv[1]);
  memcpy(&mov[1], &num, 4);

  void *mem = mmap(NULL, sizeof(mov) + sizeof(ret),
                PROT_WRITE | PROT_EXEC,
                MAP_ANON | MAP_PRIVATE, -1, 0);

  memcpy(mem, mov, sizeof(mov));
  memcpy(mem + sizeof(mov), ret, sizeof(ret));

  int (*func)() = mem;

  return func();
}
```

Modified from http://blog.reverberate.org/2012/12/hello-jit-world-joy-of-simple-jits.html

© 2018 NETRONOME

What is a JIT Compiler?

NETRONOME

- How does a JIT compiler really look like? Here is a very simple example.

```
int main(int argc, char *argv[]) {
  // mov eax, 0
  unsigned char mov[] = {0xb8, 0x00, 0x00, 0x00, 0x00};
  // ret
  unsigned char ret[] = {0xc3};

  int num = atoi(argv[1]);
  memcpy(&mov[1], &num, 4);

  void *mem = mmap(NULL, sizeof(mov) +sizeof(ret),
                PROT_WRITE | PROT_EXEC,
                MAP_ANON | MAP_PRIVATE, -1, 0);

  memcpy(mem, mov, sizeof(mov));
  memcpy(mem + sizeof(mov), ret, sizeof(ret));

  int (*func)() = mem;

  return func();
}
```

Any Program

Block of Memory

© 2018 NETRONOME

What is a JIT Compiler?

NETRONOME

- How does a JIT compiler really look like? Here is a very simple example.

```
int main(int argc, char *argv[]) {
  // mov eax, 0
  unsigned char mov[] = {0xb8, 0x00, 0x00, 0x00, 0x00};
  // ret
  unsigned char ret[] = {0xc3};

  int num = atoi(argv[1]);
  memcpy(&mov[1], &num, 4);

  void *mem = mmap(NULL, sizeof(mov) +sizeof(ret),
                PROT_WRITE | PROT_EXEC,
                MAP_ANON | MAP_PRIVATE, -1, 0);

  memcpy(mem, mov, sizeof(mov));
  memcpy(mem + sizeof(mov), ret, sizeof(ret));

  int (*func)() = mem;

  return func();
}
```

Any Program

Block of Memory

© 2018 NETRONOME

What is a JIT Compiler?

NETRONOME

- How does a JIT compiler really look like? Here is a very simple example.

```
int main(int argc, char *argv[]) {
  // mov eax, 0
  unsigned char mov[] = {0xb8, 0x00, 0x00, 0x00, 0x00};
  // ret
  unsigned char ret[] = {0xc3};

  int num = atoi(argv[1]);
  memcpy(&mov[1], &num, 4);

  void *mem = mmap(NULL, sizeof(mov) +sizeof(ret),
                PROT_WRITE | PROT_EXEC,
                MAP_ANON | MAP_PRIVATE, -1, 0);

  memcpy(mem, mov, sizeof(mov));
  memcpy(mem + sizeof(mov), ret, sizeof(ret));

  int (*func)() = mem;

  return func();
}
```

Any Program

Block of Memory

© 2018 NETRONOME

What is a JIT Compiler?

NETRONOME

- How does a JIT compiler really look like? Here is a very simple example.

```
int main(int argc, char *argv[]) {
  // mov eax, 0
  unsigned char mov[] = {0xb8, 0x00, 0x00, 0x00, 0x00};
  // ret
  unsigned char ret[] = {0xc3};

  int num = atoi(argv[1]);
  memcpy(&mov[1], &num, 4);

  void *mem = mmap(NULL, sizeof(mov) +sizeof(ret),
                PROT_WRITE | PROT_EXEC,
                MAP_ANON | MAP_PRIVATE, -1, 0);

  memcpy(mem, mov, sizeof(mov));
  memcpy(mem + sizeof(mov), ret, sizeof(ret));

  int (*func)() = mem;

  return func();
}
```

Any Program

Jump & Execute

Block of Memory

© 2018 NETRONOME

What is a JIT Compiler - Summary

NETRONOME

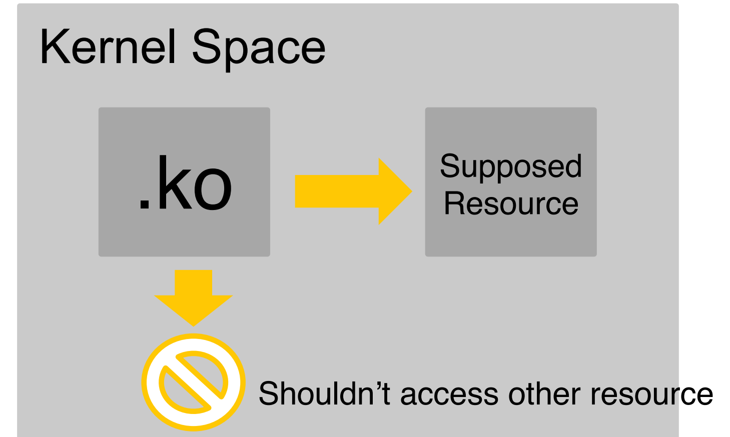
- Generate machine instructions at runtime, write them to executable memory and execute them there directly
  - No assembling stage, direct encoding generation
  - No linking stage, direct absolute address generation
- Memory region allocated should be protected
  - Very risky to leave it as both executable and writable
  - Classic buffer overflow writing to stack could be seen as JIT compilation
- Runtime overhead
  - Only JIT compiles hot code, trace compilation etc.

© 2018 NETRONOME

What is an eBPF JIT Compiler?

NETRONOME

- The whole idea is to run user-supplied programs inside kernel
- Why not kernel module .ko?
  - C is permissive, pointer is exploitable
- What's wrong with checking .ko?
  - It is compiled already, different architectures are different in ISA (instruction set architecture), we will end up with a bunch of .ko verifiers doing similar things and they can't be merged due to ISA differences
- We want to check on unified representation. It was BPF (Berkeley Packet Filter), now enhanced as eBPF (Extended BPF)



© 2018 NETRONOME

What is an eBPF JIT Compiler?

NETRONOME

eBPF representation

- Designed to be set of instructions and are close to x86-64, AArch64 etc.
- 64-bit instruction encoding and 64-bit register with 32-bit sub-register
- Usual data manipulation instructions and control transfer instructions
- Support both interpretation execution and JIT execution
- KERNEL/Documentation/bpf/bpf_design_QA.rst

```
35:    57 02 00 00 3f ff 00 00        r2 &= 65343  (AND)
36:    55 02 c9 02 00 00 00 00        if r2 != 0 goto +713 (JMP)
37:    71 82 17 00 00 00 00 00        r2 = *(u8 *)(r8 + 23) (LOAD)
38:    7b 3a a8 ff 00 00 00 00        *(u64 *)(r10 - 88) = r3 (STORE)
```

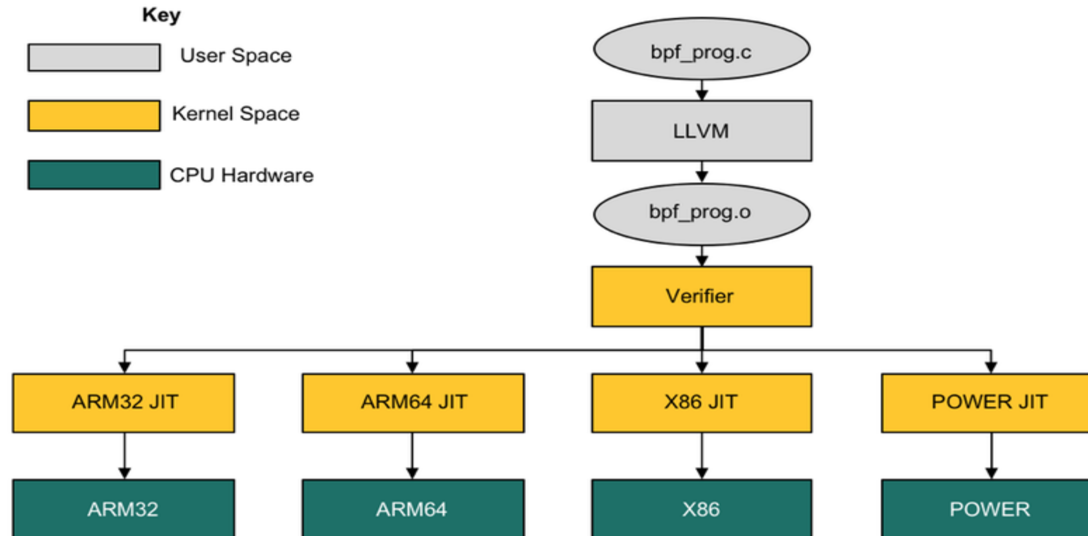
© 2018 NETRONOME

What is an eBPF JIT Compiler?

NETRONOME

eBPF software stack for JIT execution

- Compile C into eBPF sequence
- Check eBPF sequence
- JIT compile and execute the sequence

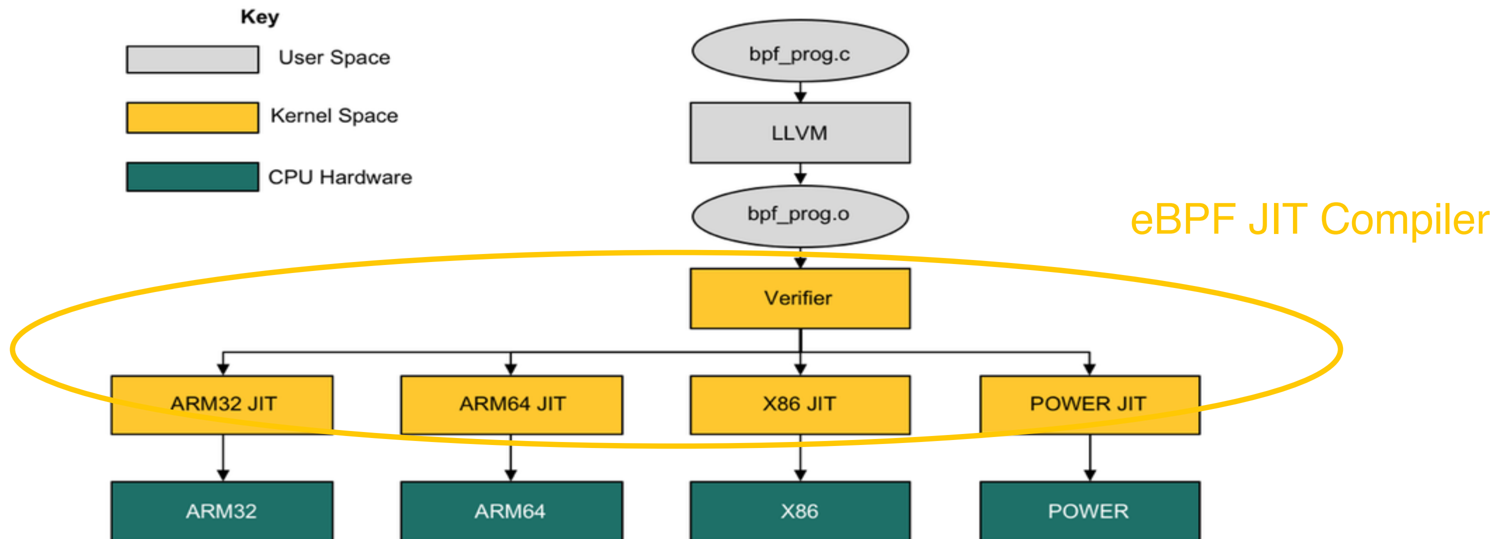


© 2018 NETRONOME

What is an eBPF JIT Compiler?

NETRONOME

- Components in yellow are sitting inside kernel space
- Verifier needs to walk instructions, quite a few information are collected and shared with architecture code generation back-ends
- They work closely, and form an eBPF JIT compiler as a whole



© 2018 NETRONOME

eBPF JIT Compiler - Verification Stage

- Control flow check
  - No function call to an unknown function
  - No fall through from one function to the next one
  - No jump destination is out of range
  - No unreachable instruction
  - No loop

© 2018 NETRONOME

eBPF JIT Compiler - Verification Stage

NETRONOME

- Individual instruction check based on static information
  - Divide by zero
  - Shifts with invalid shifting amount
  - Invalid stack access (unaligned, out of range etc)

```
if ((opcode == BPF_LSH || opcode == BPF_RSH ||
     opcode == BPF_ARSH) && BPF_SRC(insn->code) == BPF_K) {
   int size = BPF_CLASS(insn->code) == BPF_ALU64 ? 64 : 32;
   if (insn->imm < 0 || insn->imm >= size) {
     verbose(env, "invalid shift %d\n", insn->imm);
      return -EINVAL;
    }
}
```

```
/* BPF program can access up to 512 bytes of stack space. */
#define MAX_BPF_STACK   512

if (off >= 0 || off < -MAX_BPF_STACK) {
  verbose(env, "invalid stack off=%d size=%d\n", off, size);
  return -EACCES;
}
```

© 2018 NETRONOME

eBPF JIT Compiler - Verification Stage

NETRONOME

- Individual instruction check based on dynamic information
  - Value range based checks. For example, out of range access to packet data
  - Register status based checks. For example, read from uninitialized register

```
mov r0, r2
exit
```

  - Stack status based checks. For example, a corruption spilled pointer on stack

```
*(u64 *)(r10, -8) = r1
/* mess up with R1 pointer on stack */
*(u8 *)(r10, -7) = 0x23
/* fill back into R0 should fail */
r0 = *(u64 *)(r10, -8)
exit
```

- These checks requires tracking data flow dynamically at an instruction level

© 2018 NETRONOME

eBPF JIT Compiler - Verification Stage

NETRONOME

- Kernel space pointer leak checks under unprivileged mode
  - Such information is highly useful to an attacker once leaked to userspace
  - No pointer arithmetic and comparison
  - No store back to user space accessible storage, like map, packet etc.
  - https://lwn.net/Articles/660331/ https://lkml.org/lkml/2015/10/5/687

"unpriv: return pointer",
"unpriv: add const to pointer",
"unpriv: add pointer to pointer",
"unpriv: neg pointer",
"unpriv: cmp pointer with const",
"unpriv: cmp pointer with pointer",
"unpriv: check that printk is disallowed",
"unpriv: pass pointer to helper function",
"unpriv: indirectly pass pointer on stack to helper function",
"unpriv: mangle pointer on stack 1",
"unpriv: mangle pointer on stack 2",
"unpriv: read pointer from stack in small chunks",
"unpriv: write pointer into ctx",
"unpriv: spill/fill of ctx",

"unpriv: spill/fill of ctx 2",
"unpriv: spill/fill of ctx 3",
"unpriv: spill/fill of ctx 4",
"unpriv: spill/fill of different pointers stx",
"unpriv: spill/fill of different pointers ldx",
"unpriv: write pointer into map elem value",
"unpriv: partial copy of pointer",
"unpriv: pass pointer to tail_call",
"unpriv: cmp map pointer with zero",
"unpriv: write into frame pointer",
"unpriv: spill/fill frame pointer",
"unpriv: cmp of frame pointer",
"unpriv: adding of fp",
"unpriv: cmp of stack pointer",

© 2018 NETRONOME

eBPF JIT Compiler - Verification Stage

NETRONOME

- Any list for all supported verifications?
  - KERNEL/toos/testing/selftests/bpf/test_verifier.c is a good reference
  - ~900 tests
  - Tests are categorized to some extent using prefix, for example verifications related with another major feature bpf-to-bpf function could be listed using

```
cat tools/testing//selftestsbpf/test_verifier.c  I grep "\"calls:"

    "calls: basic sanity",
    "calls: not on unpriviledged",
    "calls: div by 0 in subprog",
    "calls: multiple ret types in subprog 1",
    "calls: multiple ret types in subprog 2",
    "calls: overlapping caller/callee",
    …
```

  - “unpriv:”, “call:”, “XDP,” are interesting categories worth having a look

Verification Stage - Algorithms

NETRONOME

- Control flow checks
  - Function partition
  - Depth first walk to detect loop and unreachable instructions
- Data flow tracking
  - All-code-paths walker to collect information instruction by instruction and path by path

© 2018 NETRONOME

Verification Stage - Algorithms

NETRONOME

- Control flow checks
  - Function partition
  - Depth first walk to detect loop and unreachable instructions
- Data flow tracking
  - All-code-paths walker to collect information instruction by instruction and path by path

  - Code path prune for avoiding walking paths proven to be safe

© 2018 NETRONOME

Verification Stage - Algorithms

NETRONOME

- Control flow checks
  - Function partition
  - Depth first walk to detect loop and unreachable instructions
- Data flow tracking
  - All-code-paths walker to collect information instruction by instruction and path by path
  - Code path prune for avoiding walking paths proven to be safe
  - Path sensitive register liveness tracking to release more prune opportunities

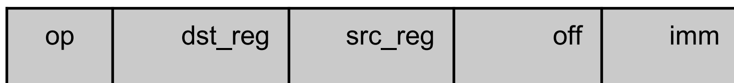
© 2018 NETRONOME

Algorithm - Function Partition

NETRONOME

- Input eBPF sequence is a sequence of functions
- Partition the sequence to make them explicit

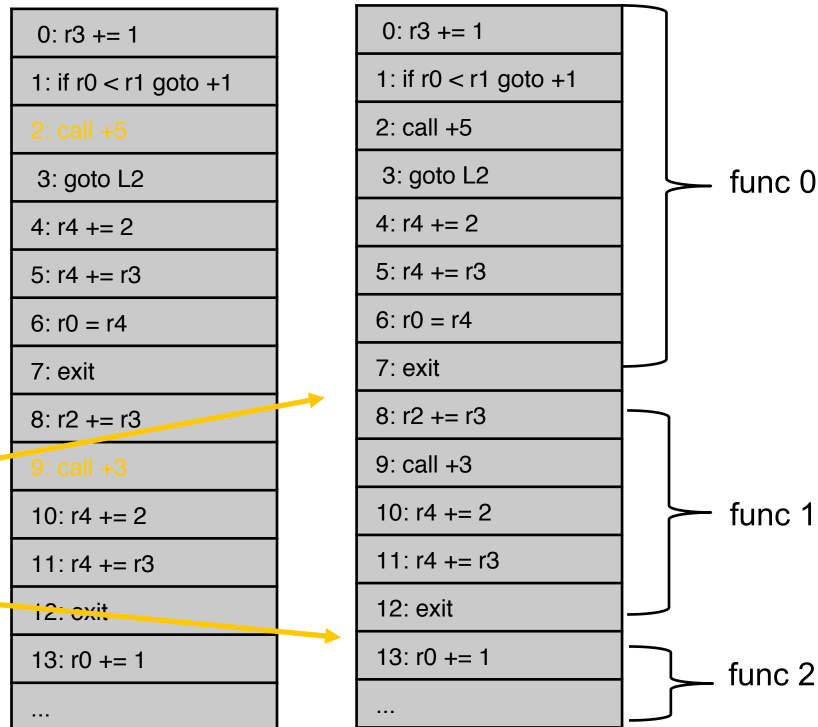
eBPF CALL instruction encoding



BPF_CALL

call destination = insn_index + imm + 1

func 1 start = 2 + 5 + 1 = 8
func 2 start = 9 + 3 + 1 = 13



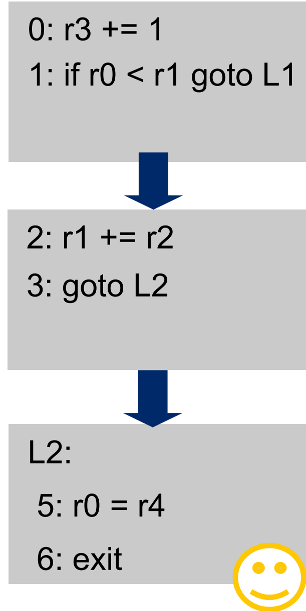
Algorithm - Depth First Walk

NETRONOME

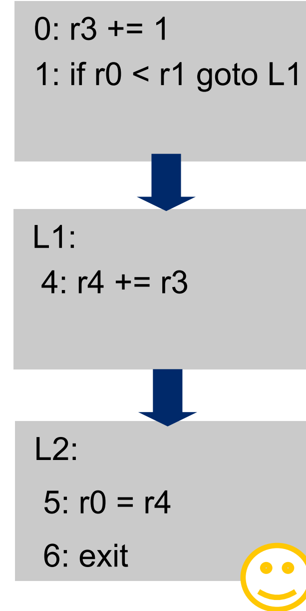
- Depth first walk is used to detect loop and unreachable instruction

```
0: r3 += 1
1: if r0 < r1 goto L1
2: r1 += r2
3: goto L2
L1:
4: r4 += r3
L2:
5: r0 = r4
6: exit
```

Branch not taken



Branch taken



Two auxiliary array to help the walk:
insn_stack [MAX_INSN_NUM]
insn_status[MAX_INSN_NUM]

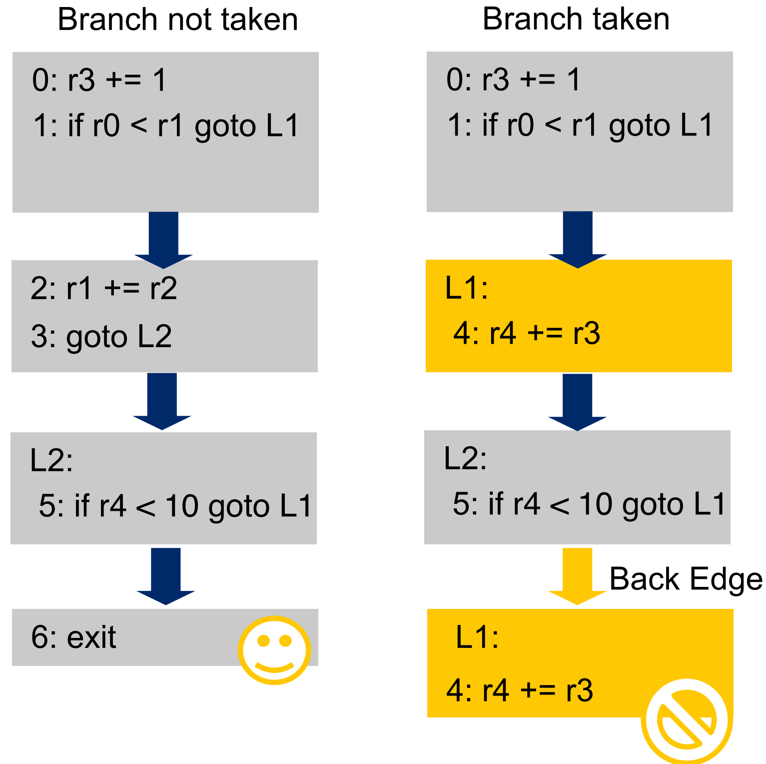
© 2018 NETRONOME

NETRONOME

Algorithm - Depth First Walk

- Depth first walk is used to detect loop and unreachable instruction

```
0: r3 += 1
1: if r0 < r1 goto L1
2: r1 += r2
3: goto L2
L1:
4: r4 += r3
L2:
5: if r4 < 10 goto L1
6: exit
```



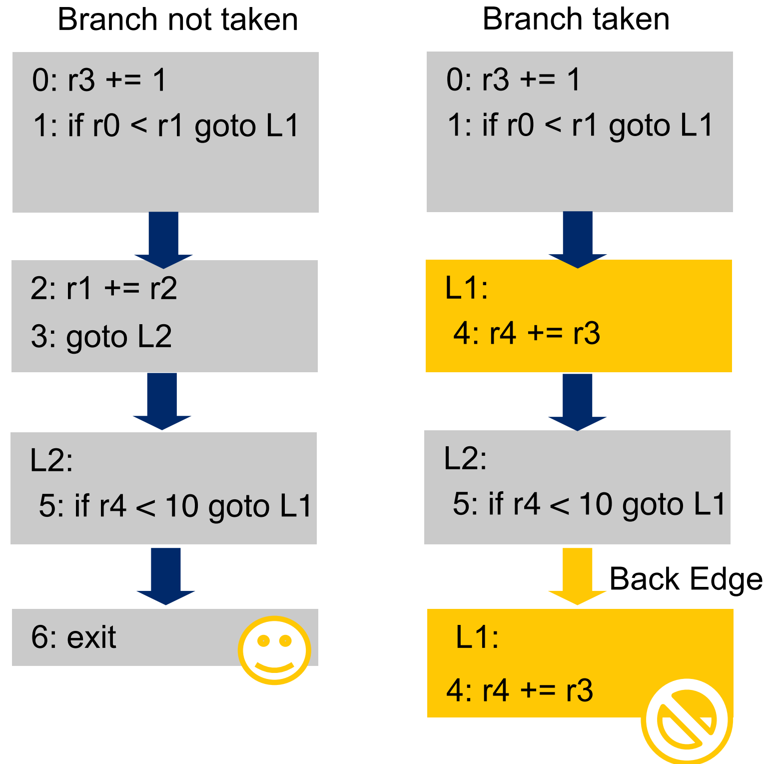
Two auxiliary array to help the walk:
insn_stack [MAX_INSN_NUM]
insn_status[MAX_INSN_NUM]

© 2018 NETRONOME

Algorithm - Depth First Walk

- Depth first walk is used to detect loop and **unreachable instruction**

```
0: r3 += 1
1: if r0 < r1 goto L1
2: r1 += r2
3: goto L2
L1:
4: r4 += r3
L2:
5: if r4 < 10 goto L1
6: exit
```



Two auxiliary array to help the walk:
insn_stack [MAX_INSN_NUM]
insn_status[MAX_INSN_NUM]

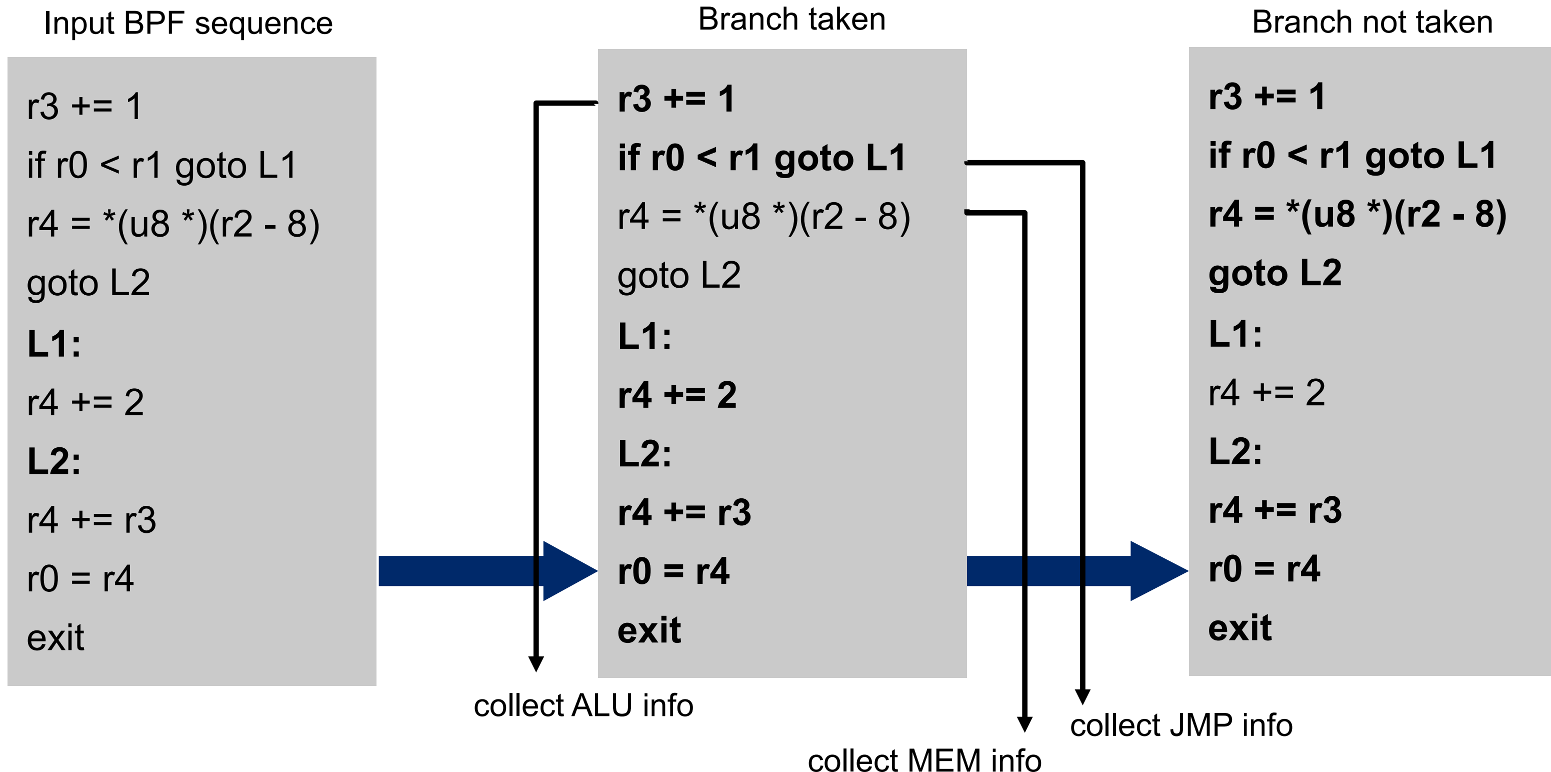
Algorithm - Code Path Walker

NETRONOME

- Execute all code path
  - Accurately knows program status (registers, memory etc.) at any point
  - Track and propagate scalar value range, pointer types etc.
  - Code path walker is time intensive
  - Program will be rejected if it is too complex (BPF_COMPLEXITY_LIMIT_INSNS)

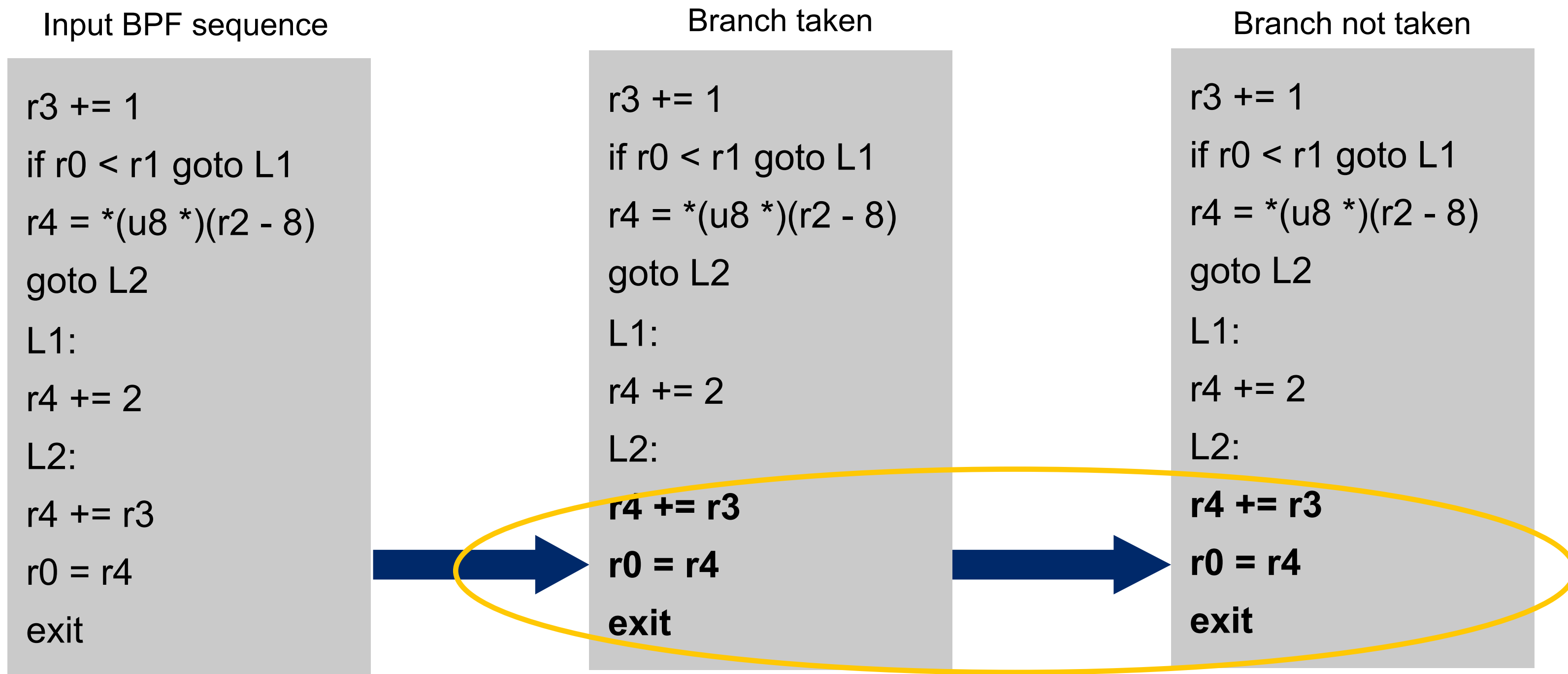
© 2018 NETRONOME

Algorithm - Code Path Walker



© 2018 NETRONOME

Algorithm - Code Path Walker

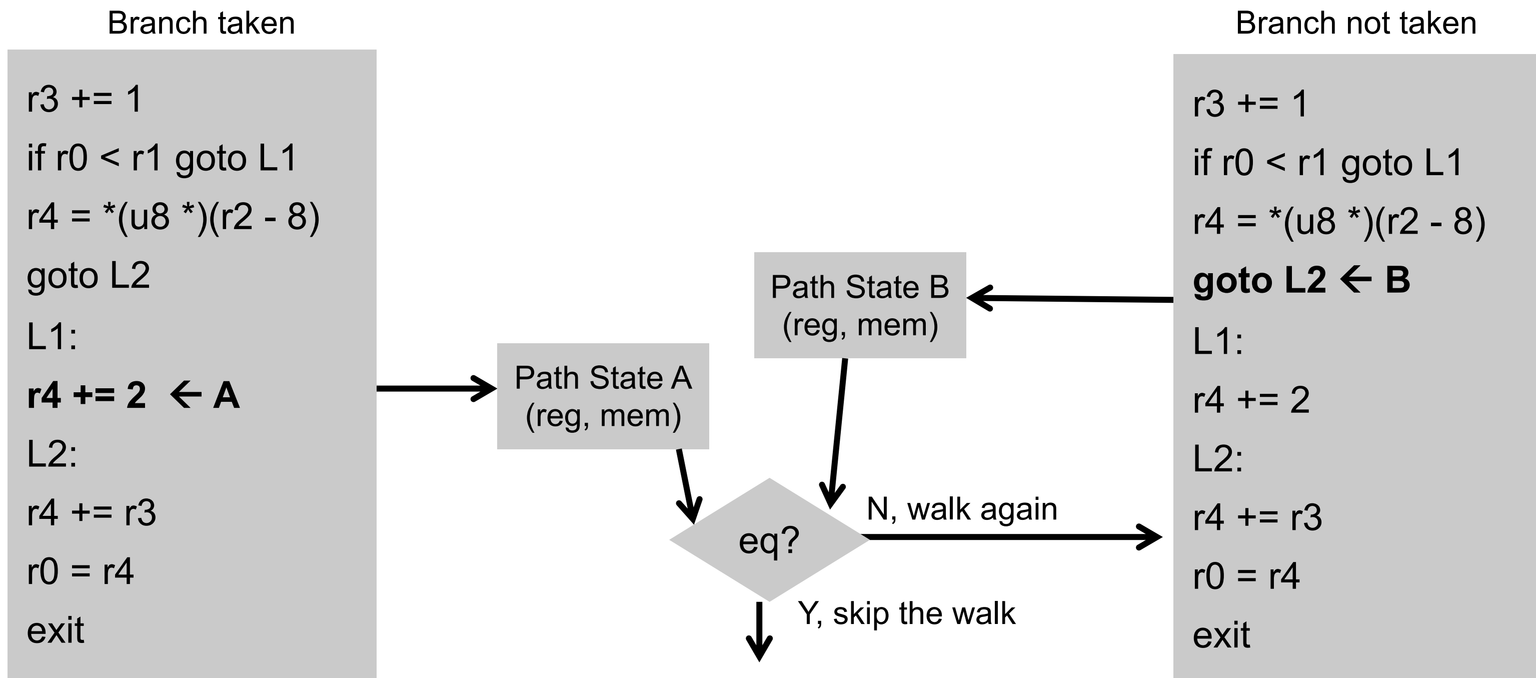


This sub-sequence starting from L2 towards the exit is executed for both.
Can we avoid walking through it again?

© 2018 NETRONOME

Algorithm - Code Path Prune

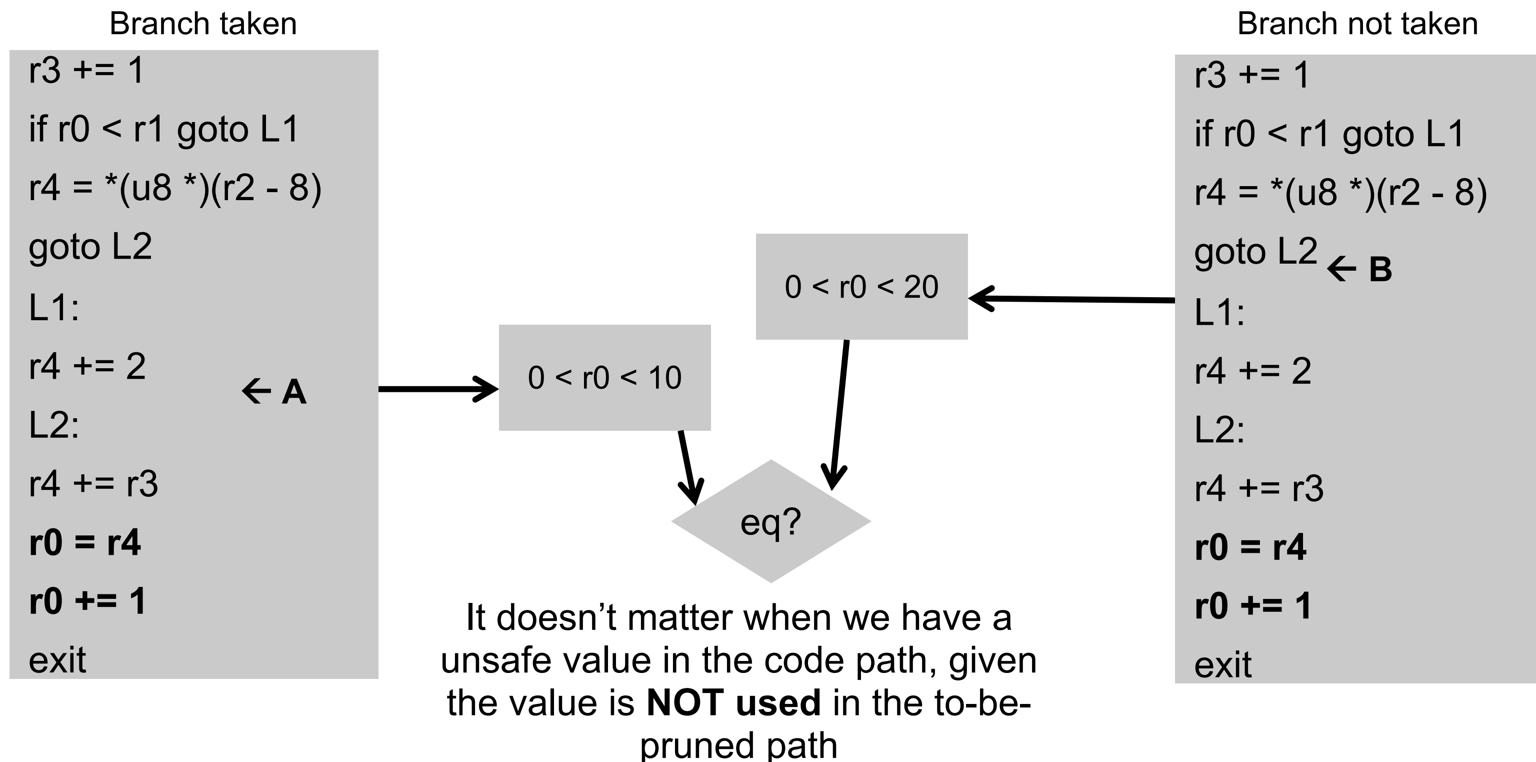
- No need to walk the sequence again given we have SAFER status when reaching the starting point of the sequence. Those points are where paths merged.



Algorithm - Register Liveness Tracking

NETRONOME

- The purpose of tracking register liveness is to make more path prune happen



© 2018 NETRONOME

Verification Stage - Challenges

NETRONOME

- Complex program will require too much verification resources
- Use classic static flow analysis techniques might lower resource consuming and making verifier more scalable, however would be challenging to guarantee the same level of securities.

© 2018 NETRONOME

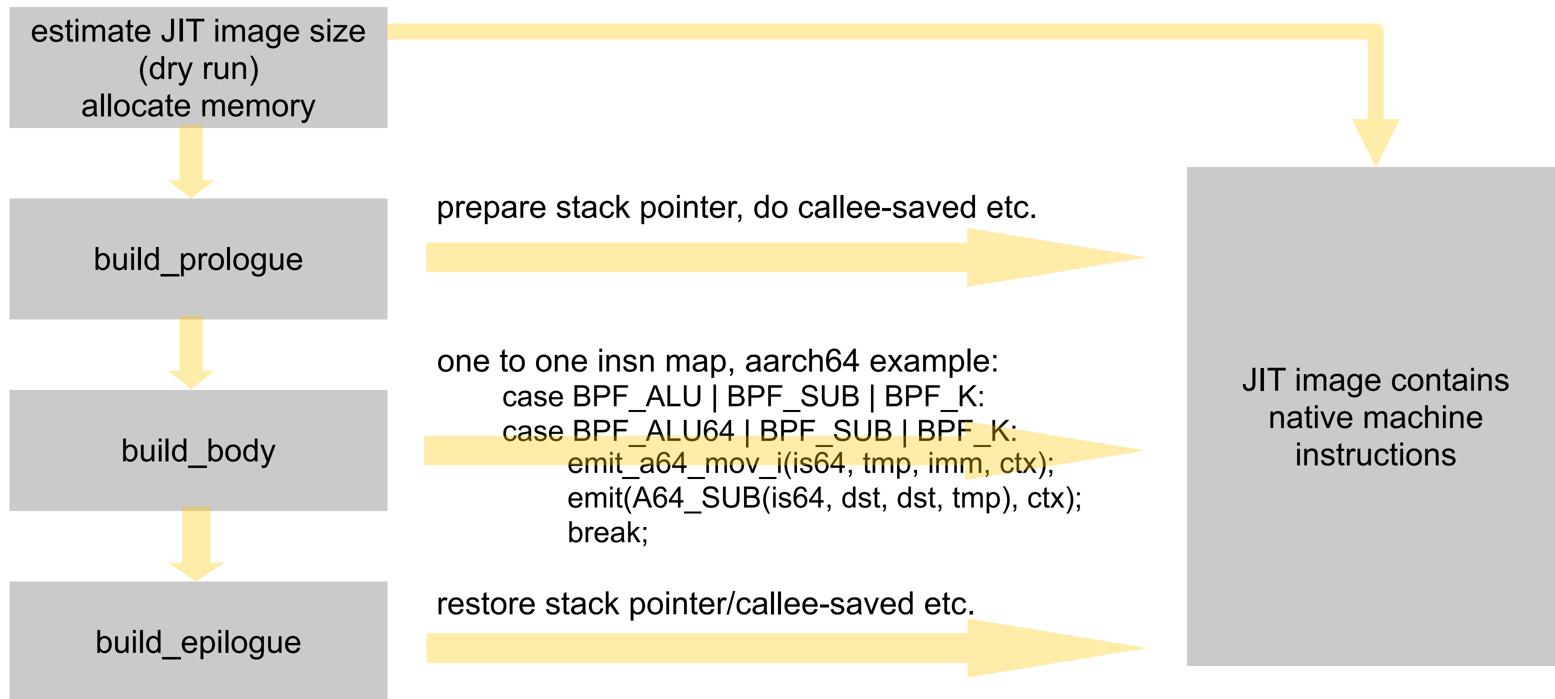
Code Gen Stage - Overview

NETRONOME

- After eBPF sequence passed verification, Code Gen stage start which translates eBPF into native machine instruction
- This is fairly straightforward as eBPF instructions could be one-to-one mapped to native machine instruction on most modern architectures
- Architecture Code Gen back-ends, for example x86-64/AArch64 etc., typically share the same Code Gen flow.

© 2018 NETRONOME

Code Gen Stage - General Flow



arch/x86,arm64/net/bpf_jit_comp.c are quite self-explained

© 2018 NETRONOME

Code Gen Stage - Linking

NETRONOME

- eBPF does not allow global variables and random function calls, no external symbol access in general which save linking jobs
- eBPF does allow call to special runtime helper functions (map), helper address needs fixup
  - Function ID is encoded in “imm” field of eBPF call instruction
  - ID needs to be mapped to address and call instruction should be relocated

verifier perpare absoluate address

```
case BPF_JMP | BPF_CALL:
{
  switch (insn->imm) {
    case BPF_FUNC_map_lookup_elem:
      insn->imm =
        BPF_CAST_CALL(ops->map_lookup_elem) - __bpf_call_base;
      continue;
    case BPF_FUNC_map_update_elem:
    ...
```

back-end relocate call instruction

```
case BPF_JMP | BPF_CALL:
{
  const u8 r0 = bpf2a64[BPF_REG_0];
  const u64 func = (u64)__bpf_call_base + imm;

  emit_a64_mov_i64(tmp, func, ctx);
  emit(A64_BLR(tmp), ctx);
```

64-bit address is too long to encode, keep offset instead

© 2018 NETRONOME

Code Gen Stage - Linking for BPF-to-BPF Call

NETRONOME

- bpf-to-bpf function call requires relocating function call address as well
- bpf-to-bpf function call requires JIT compile all eBPF functions first, then we know start address for each function, and need a second round compilation to relocate all call instructions.
- Architecture back-end doesn't know function partition, Verifier drives the whole Code Gen

```
for (i = 0; i < func_cnt; i++)
  arch_jit_hook_for_each_func

scan all insn, rewrite imm field of call insn to
   callee_start_address - __bpf_call_base

// redo the JIT
for (i = 0; i < func_cnt; i++)
  arch_jit_hook_for_each_func
```

© 2018 NETRONOME

Code Gen Stage - Summary

NETRONOME

- Architecture Code Gen hooks do instruction mapping
- Verifier do linking and all other jobs which are generic for all architectures
- Verifier and architecture back-ends work closely to form the whole eBPF JIT compiler

© 2018 NETRONOME

NFP Code Gen Back-end - NFP Introduction

NETRONOME

- Netronome Agilio® SmartNIC contains a powerful Network Flow Processor (NFP)
- eBPF aims to offer programmability to kernel network stack
- NFP and eBPF matches each other, perfect to offload eBPF to NFP, yes we can!
- NFP Code Gen back-end translates eBPF instructions into NFP instructions, write them to Agilio SmartNIC, and run them there on the fly
- In terms of the JIT compiler Code Gen, instruction set architecture is the most relevant part, so we will focus on this in the following slides
- NFP itself is a very powerful chip which offers many other capabilities, please see our website (Document Library) for details

© 2018 NETRONOME

NFP Instruction Set Architecture (ISA)

NETRONOME

- 32 x 32-bit general purpose registers for each eBPF context
- 8K instructions for each eBPF context
- Rich arithmetic and logic instructions
- Powerful memory unit allowing unaligned access and bulk access
- Global absolute jump within the whole instruction buffer
- Clustered register sets offers generous registers other than general purpose registers

© 2018 NETRONOME

NFP Code Gen Back-end - Overview

NETRONOME

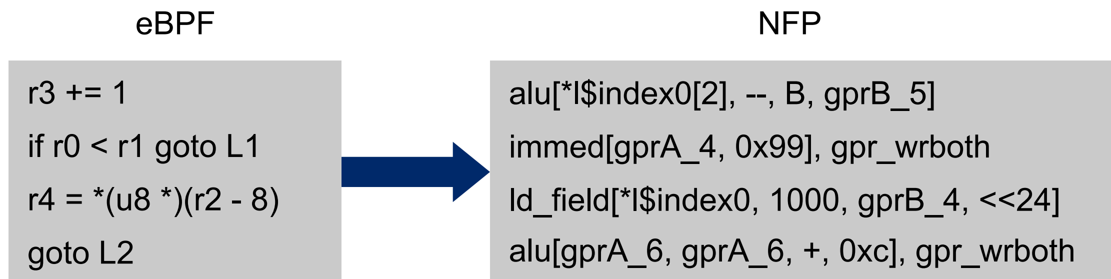
- Generally works the same way as the other eBPF arch back-ends.
  - Have verified eBPF sequence as input
  - Allocate JIT image on Agilio SmartNIC
  - One to one map eBPF instruction, and writes to JIT image
- However, there are some differences:
  - NFP has a unique and powerful memory unit which needs special support
  - NFP has other registers besides general purpose registers, need to utilize them
  - NFP has 32-bit general purpose register instead of 64-bit
  - NFP wants to do all linking jobs by itself

© 2018 NETRONOME

NFP Code Gen Back-end - Overview

NETRONOME

- Differences (continued):
  - More strict verification
  - Maps are now located on SmartNIC, need to redirect all map access. Call firmware functions and get results. Extra Code Gen about function call and a couple of instruction relocation.



© 2018 NETRONOME

NFP Code Gen Back-end - Memory Support

NETRONOME

Memcpy Optimization

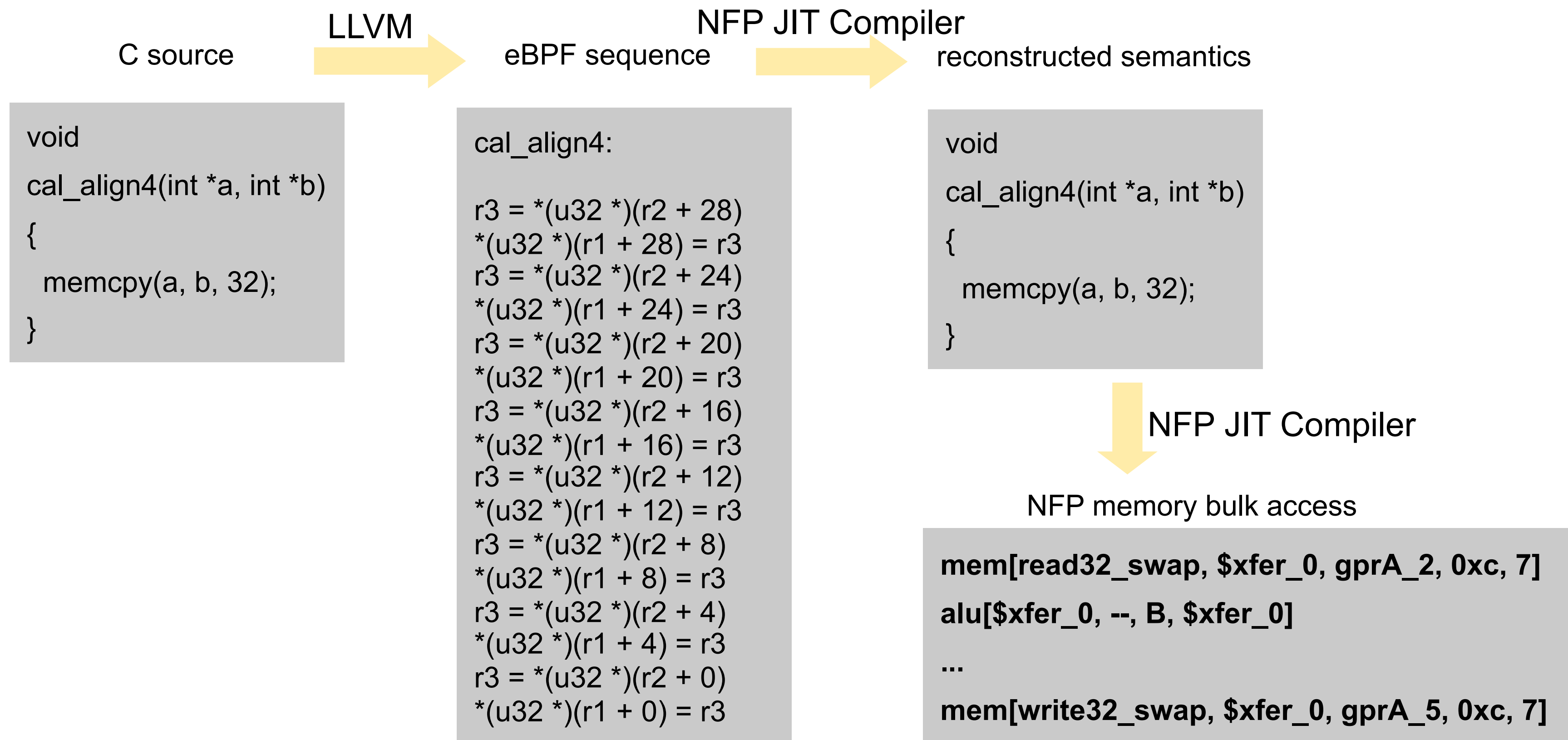
- Classic RISC arch use load/store model, can only access register width at maximum per instruction
- NFP has special internal bus allows bulk memory access and could access 128 bytes at maximum per instruction
- eBPF ISA follows RISC load/store model, memcpy function are expanded into load/store pair by LLVM compiler due to inline memcpy for small copy size is common techniques and eBPF does not allow call external function
- This is not good for NFP, or actually might be not good for other architectures. It is better to let the architecture back-ends decide how to expand memcpy

© 2018 NETRONOME

NFP Code Gen Back-end - Memory Support

NETRONOME

- Memcpy Optimization



© 2018 NETRONOME

NFP Code Gen Back-end - Memory Support

NETRONOME

Memcpy Optimization

Instruciton Scheduling

eBPF sequence → another sequence

eBPF sequence	another sequence
cal_align4:	cal_align4:
r3 = *(u32 *)(r2 + 28)	r3 = *(u32 *)(r2 + 28)
*(u32 *)(r1 + 28) = r3	*(u32 *)(r1 + 28) = r3
r3 = *(u32 *)(r2 + 24)	r3 = *(u32 *)(r2 + 24)
*(u32 *)(r1 + 24) = r3	*(u32 *)(r1 + 24) = r3
r3 = *(u32 *)(r2 + 20)	r3 = *(u32 *)(r2 + 16)
*(u32 *)(r1 + 20) = r3	*(u32 *)(r1 + 16) = r3
r3 = *(u32 *)(r2 + 16)	r3 = *(u32 *)(r2 + 12)
*(u32 *)(r1 + 16) = r3	*(u32 *)(r1 + 12) = r3
r3 = *(u32 *)(r2 + 12)	r3 = *(u32 *)(r2 + 20)
*(u32 *)(r1 + 12) = r3	*(u32 *)(r1 + 20) = r3
r3 = *(u32 *)(r2 + 8)	r3 = *(u32 *)(r2 + 4)
*(u32 *)(r1 + 8) = r3	*(u32 *)(r1 + 4) = r3
r3 = *(u32 *)(r2 + 4)	r3 = *(u32 *)(r2 + 0)
*(u32 *)(r1 + 4) = r3	*(u32 *)(r1 + 0) = r3
r3 = *(u32 *)(r2 + 0)	r3 = *(u32 *)(r2 + 8)
*(u32 *)(r1 + 0) = r3	*(u32 *)(r1 + 8) = r3

Instruction schduling done by compiler will make it hard for eBPF JIT compilers to reconstruct memcpy semantics, Netronome contributed new LLVM option

```
llc -bpf-expand-memcpy-in-order
```

to force LLVM generating unscheduled memcpy sequence.

© 2018 NETRONOME

NFP Code Gen Back-end - Memory Support

NETRONOME

Memcpy Optimization

Instruciton Scheduling

eBPF sequence → another sequence

```
cal_align4:

r3 = *(u32 *)(r2 + 28)
*(u32 *)(r1 + 28) = r3
r3 = *(u32 *)(r2 + 24)
*(u32 *)(r1 + 24) = r3
r3 = *(u32 *)(r2 + 20)
*(u32 *)(r1 + 20) = r3
r3 = *(u32 *)(r2 + 16)
*(u32 *)(r1 + 16) = r3
r3 = *(u32 *)(r2 + 12)
*(u32 *)(r1 + 12) = r3
r3 = *(u32 *)(r2 + 8)
*(u32 *)(r1 + 8) = r3
r3 = *(u32 *)(r2 + 4)
*(u32 *)(r1 + 4) = r3
r3 = *(u32 *)(r2 + 0)
*(u32 *)(r1 + 0) = r3
```

```
cal_align4:

r3 = *(u32 *)(r2 + 28)
*(u32 *)(r1 + 28) = r3
r3 = *(u32 *)(r2 + 24)
*(u32 *)(r1 + 24) = r3
r3 = *(u32 *)(r2 + 16)
*(u32 *)(r1 + 16) = r3
r3 = *(u32 *)(r2 + 12)
*(u32 *)(r1 + 12) = r3
r3 = *(u32 *)(r2 + 20)
*(u32 *)(r1 + 20) = r3
r3 = *(u32 *)(r2 + 4)
*(u32 *)(r1 + 4) = r3
r3 = *(u32 *)(r2 + 0)
*(u32 *)(r1 + 0) = r3
r3 = *(u32 *)(r2 + 8)
*(u32 *)(r1 + 8) = r3
```

Instruction schduling done by compiler will make it hard for eBPF JIT compilers to reconstruct memcpy semantics, Netronome contributed new LLVM option

llc -bpf-expand-memcpy-in-order

to force LLVM generating unscheduled memcpy sequence

No guarantee r3 is dead after this instruction
Make sure r3 has the same value as old sequence

© 2018 NETRONOME

NETRONOME

Memcpy Optimization

NFP memory bulk access

```
mem[read32_swap, $xfer_0, gprA_2, 0xc, 7]
alu[$xfer_0, --, B, $xfer_0]
...
mem[write32_swap, $xfer_0, gprA_5, 0xc, 7]
```

What's this?

© 2018 NETRONOME

Memcpy Optimization

NFP memory bulk access

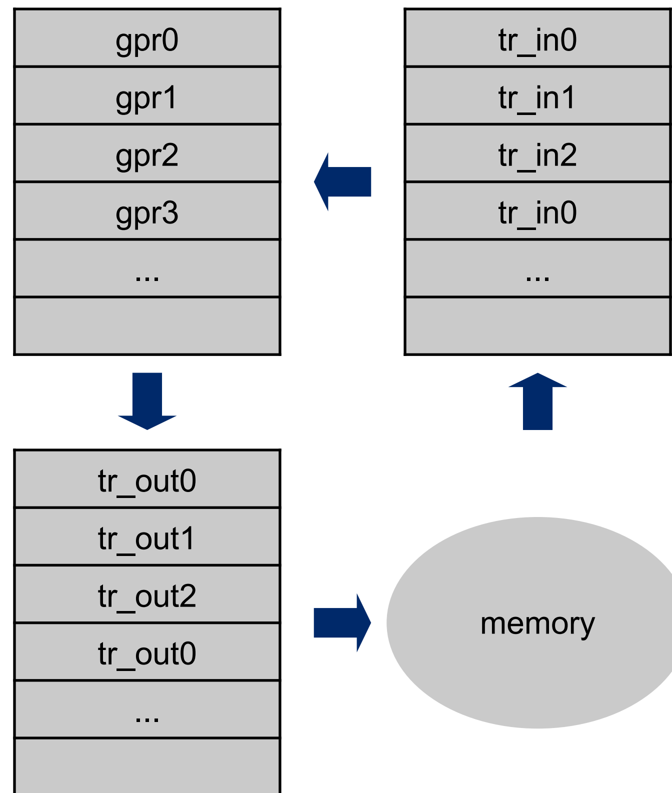
```
mem[read32_swap, $xfer_0, gprA_2, 0xc, 7]
alu[$xfer_0, --, B, $xfer_0]
...
mem[write32_swap, $xfer_0, gprA_5, 0xc, 7]
```

NFP has clustered register sets and have other registers other than general purpose registers

Memory read are into **transfer in registers** first, then moved to general purpose registers. The content is not clobbered if there is no other memory read

NFP eBPF JIT Compiler can use 32 transfer in registers, meaning could cache 128 bytes memory contents there

Packet data is frequently visited, so we use transfer in registers to cache packet data.



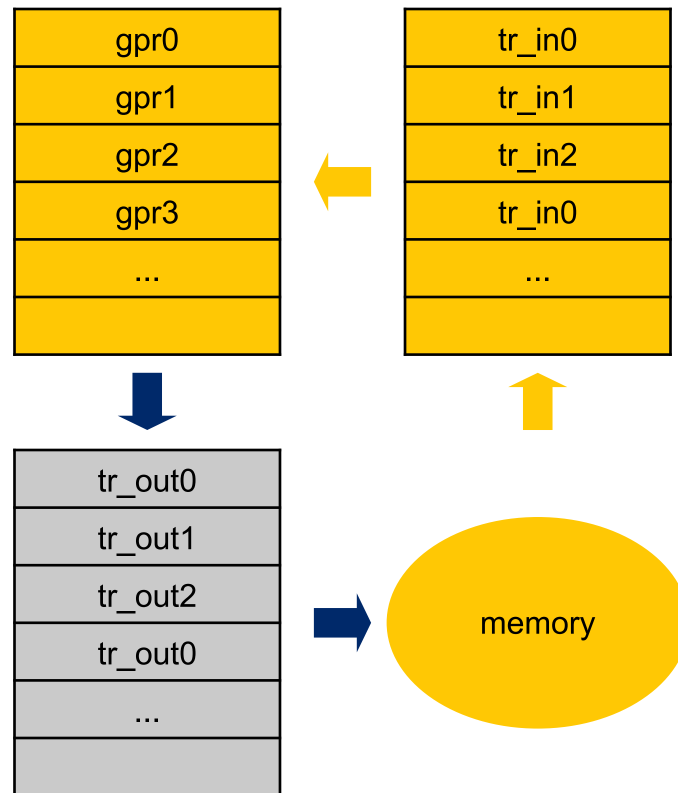
© 2018 NETRONOME

Memcpy Optimization

NFP memory bulk access

```
mem[read32_swap, $xfer_0, gprA_2, 0xc, 7]
alu[$xfer_0, --, B, $xfer_0]
...
mem[write32_swap, $xfer_0, gprA_5, 0xc, 7]
```

Read reasonable size from memory into transfer in register **during the first memory read**



© 2018 NETRONOME

NFP Code Gen Back-end - Memory Support

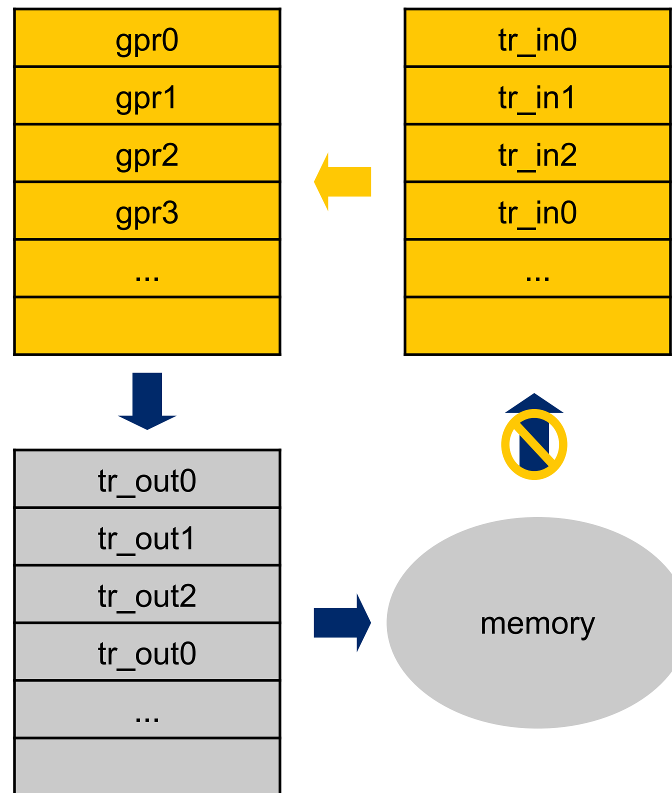
NETRONOME

- Memcpy Optimization

NFP memory bulk access

```
mem[read32_swap, $xfer_0, gprA_2, 0xc, 7]
alu[$xfer_0, --, B, $xfer_0]
...
mem[write32_swap, $xfer_0, gprA_5, 0xc, 7]
```

The follow up memory read use cache in transfer in register directly, **no need of memory read**



© 2018 NETRONOME

NFP Code Gen Back-end - 32-bit Optimization

NETRONOME

- eBPF ISA defined 32-bit sub-register and associated instructions
  - eBPF register is 64-bit, the lower half could be used as 32-bit sub-register
  - Any write to the lower half must zero the higher half. This is to match x86-64 and AArch64 ISA feature
  - A set of ALU32 instructions will operate on 32-bit sub-register

program operating on 32-bit type

```
void cal(unsigned int *a,
            unsigned int *b,
            unsigned int *c)
{
  unsigned int sum = *a + *b;

  *c = sum;
}
```

default eBPF code gen from LLVM

```
cal:
  r1 = *(u32 *)(r1 + 0)
  r2 = *(u32 *)(r2 + 0)
  r2 += r1
  *(u32 *)(r3 + 0) = r2
  exit
```

JITed AArch64 sequence

```
cal:
  ld4u r1, [r1, 0]
  ld4u r2, [r2, 0]
  addu r2, r2, r1
  st4 [r3, 0], r2
  ret
```

© 2018 NETRONOME

NFP Code Gen Back-end - 32-bit Optimization

NETRONOME

- 32-bit optimization
  - NFP (or ARM etc.) has 32-bit register, must use register pair to model eBPF register
  - NFP data manipulation instructions operates on 32-bit data. Operating on register pair needs two instructions

program operating on 32-bit type

```
void cal(unsigned int *a,
             unsigned int *b,
             unsigned int *c)
{
  unsigned int sum = *a + *b;

  *c = sum;
}
```

default eBPF code gen from LLVM

```
cal:
  r1 = *(u32 *)(r1 + 0)
  r2 = *(u32 *)(r2 + 0)
  r2 += r1
  *(u32 *)(r3 + 0) = r2
  exit
```

JITed NFP sequence (pseudo code)

```
cal:
  ld4 r2, [r2, 0]
  mov r3, 0
  ld4 r4, [r4, 0]
  mov r5, 0
  add r4, r4, r2
  addc r5, r5, r3
  st4 [r6, 0], r4
  ret
```

extra instructions to zero high half
extra instruction to operate on high half

Are they really necessary?

© 2018 NETRONOME

NFP Code Gen Back-end - 32-bit Optimization

NETRONOME

32-bit optimization - Solution 1

- 32-bit sub-register and ALU32 instructions carries semantics, safe to generate native instructions operating on sub-register only
- Netronome has contributed 32-bit sub-register and ALU32 enablement to LLVM

program operating on 32-bit type

```
void cal(unsigned int *a,
         unsigned int *b,
         unsigned int *c)
{
  unsigned int sum = *a + *b;

  *c = sum;
}
```

-mattr=+alu32 code gen from LLVM

```
cal:
  w1 = *(u32 *)(r1 + 0)
  w2 = *(u32 *)(r2 + 0)
  w2 += w1
  *(u32 *)(r3 + 0) = w2
  exit

(previous 64-bit code gen)
cal:
  r1 = *(u32 *)(r1 + 0)
  r2 = *(u32 *)(r2 + 0)
  r2 += r1
  *(u32 *)(r3 + 0) = r2
  exit
```

JITed NFP sequence (pseudo code)

```
cal:
  ld4 r2, [r2, 0]
  ld4 r4, [r4, 0]
  add r4, r4, r2
  st4 [r6, 0], r4
  ret
```

© 2018 NETRONOME

NFP Code Gen Back-end - 32-bit Optimization

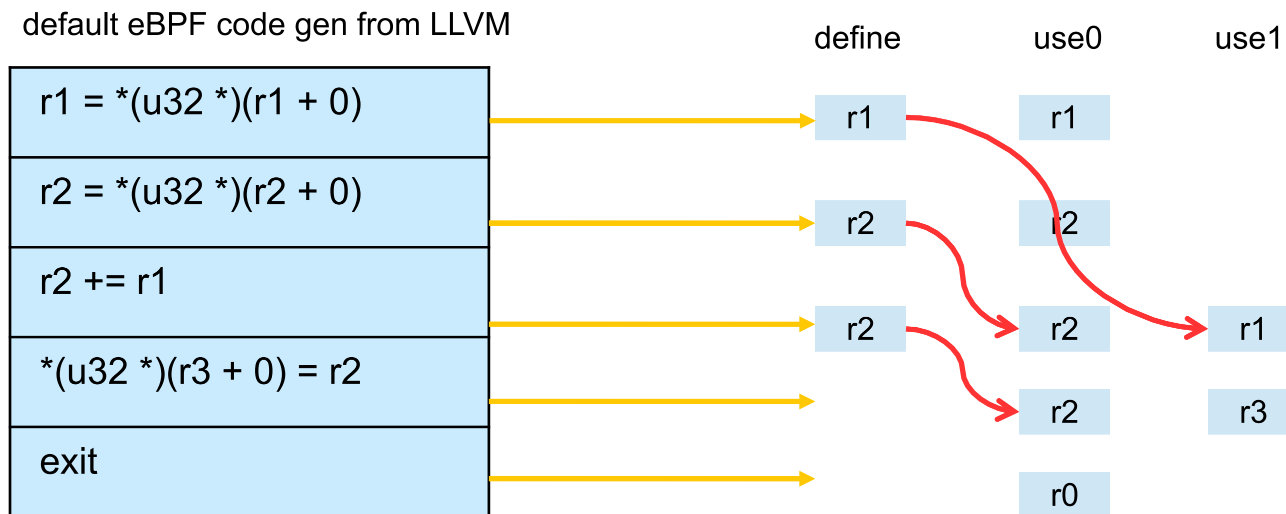
NETRONOME

- Can we rely on semantics from registers and instructions? NO
  - eBPF sequence can come from manual written assembly which doesn't conform to such semantics
  - LLVM could set ELF flags to tell the consumer the sequence is from LLVM therefore must conform to the semantics. But ELF flag could be faked even though we don't know whether a faked 32-bit flag is making the program easier or harder to exploit
  - Also for some instructions, operations on high half can’t be safely omitted without accurate usage information

© 2018 NETRONOME

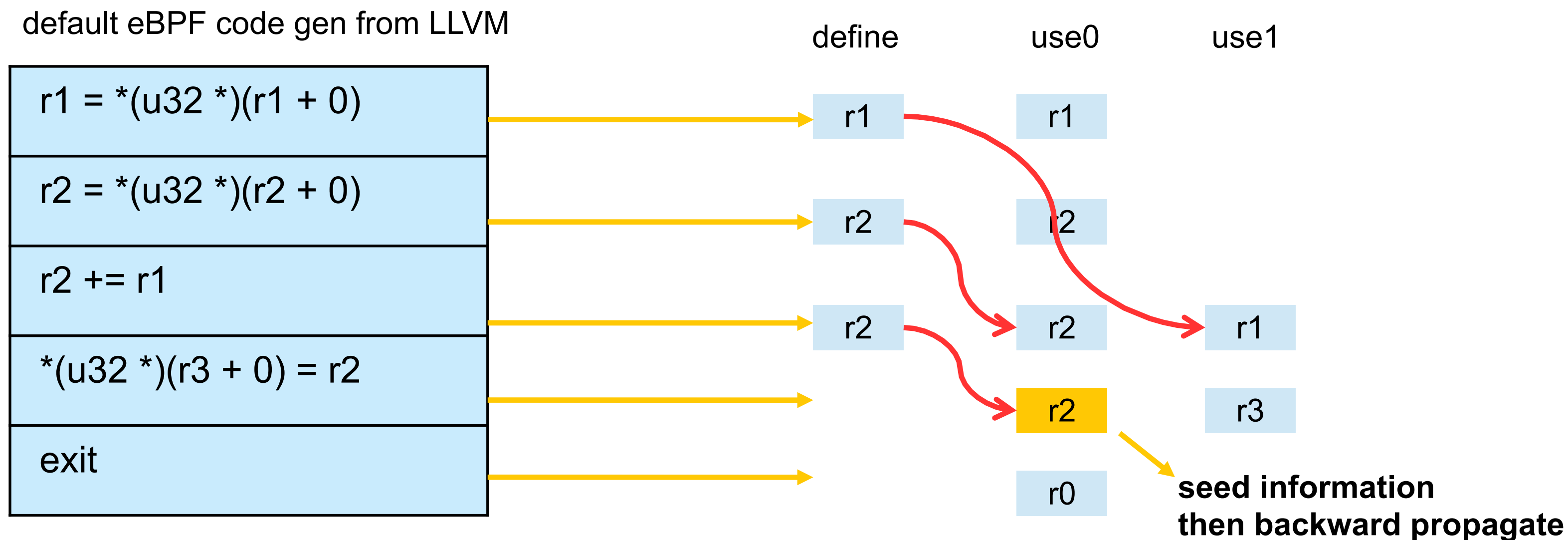
32-bit optimization - Solution 2

- JIT compiler figures out data flow from scratch, no rely on external information.
- Netronome has done initial support on this. Define-Use chain will be built for input eBPF sequence, a series of analysis could be done along the chain.



32-bit optimization - Solution 2

- JIT compiler figures out data flow from scratch, no rely on external information.
- Netronome has done initial support on this. Define-Use chain will be built for input eBPF sequence, a series of analysis could be done along the chain.



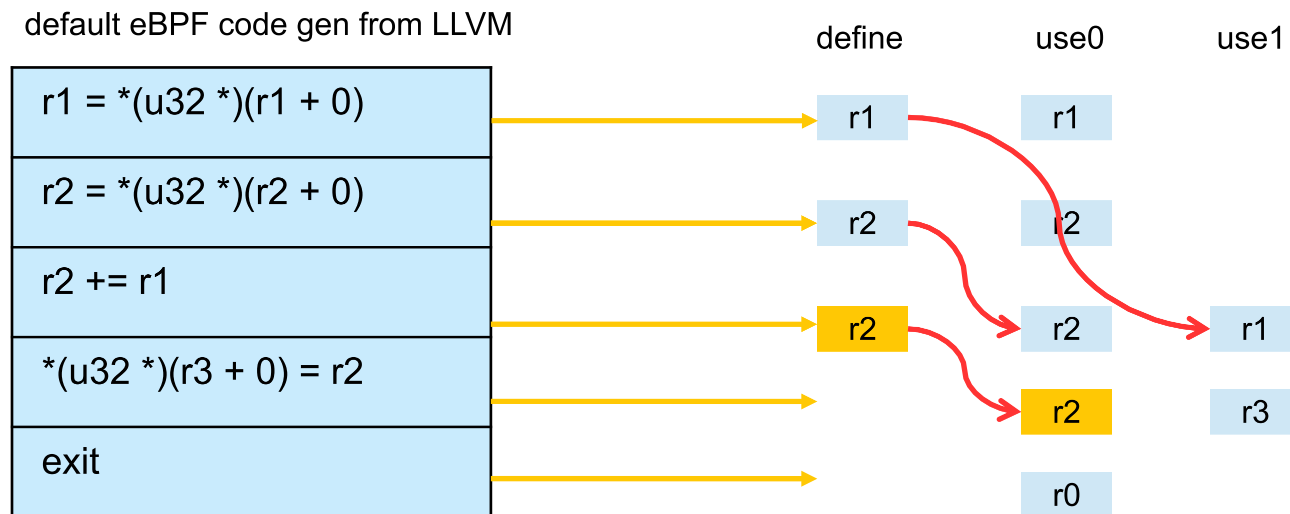
© 2018 NETRONOME

NFP Code Gen Back-end - 32-bit Optimization

NETRONOME

32-bit optimization - Solution 2

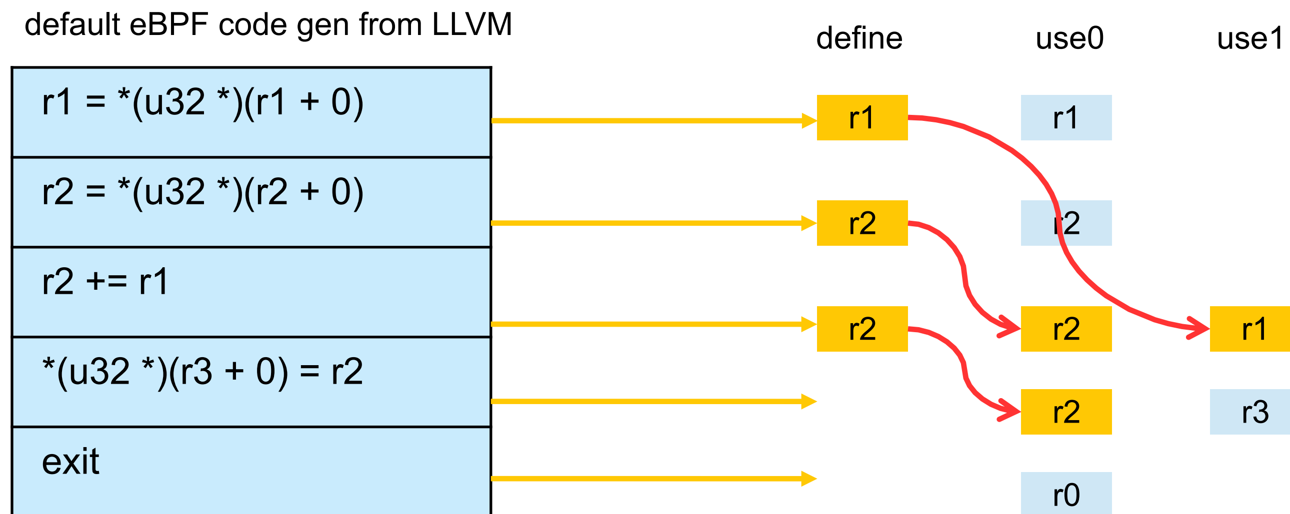
- JIT compiler figures out data flow from scratch, no rely on external information.
- Netronome has done initial support on this. Define-Use chain will be built for input eBPF sequence, a series of analysis could be done along the chain.



© 2018 NETRONOME

32-bit optimization - Solution 2

- JIT compiler figures out data flow from scratch, no rely on external information.
- Netronome has done initial support on this. Define-Use chain will be built for input eBPF sequence, a series of analysis could be done along the chain.



© 2018 NETRONOME

NFP Code Gen Back-end - Summary

NETRONOME

- Share the same flow with other host back-ends
- Has special optimization for NFP architecture features
- Targets offload execution environment

© 2018 NETRONOME

eBPF JIT Compiler - Emerging Features

NETRONOME

- Verification Stage
  - Strong requirement of a more scalable verification analysis infrastructure
  - Support bounded loops. Netronome had some proof of concept work with community to bring modern Control Flow Graph (CFG) infrastructure to eBPF verifier, then we could build a bounded loop detector on top of it
- Programming Model
  - Shared library support (dynamic/runtime linking)
- Debuggability
  - Debug JITed image through DWARF annotations. Debug information could be saved separately from executable image

© 2018 NETRONOME

Thank You!

jiong.wang@netronome.com

© 2018 NETRONOME