# JIT Code Generator for NetBSD

Alexander Nasonov <alnsn@NetBSD.org>

EuroBSDCon 2014, Sofia

# BPF JIT Project

- Started on Dec 26, 2011 as external project https://github.com/alnsn/bpfjit.
- Added to the NetBSD tree on Oct 27, 2012.
- NetBSD 7 is the first release with JIT support.
- Still work-in-progress!

# BPF JIT Project

| | Modular kernel | Monolithic kernel |
|---|---|---|
| Configure | # modload sljit | options SLJIT |
| | # modload bpfjit | options BPFJIT |
| | # sysctl -w net.bpf.jit=1 | # sysctl -w net.bpf.jit=1 |
| Run | # tcpdump tcp and host localhost and port http | |
| | # fstat \| grep jit | |

# BPF JIT Performance

- Good news: bpf interpreter is fast.
- Even better news: bpf jit is several times faster.
  - For a short filter program it's 4 times faster on amd64 and arm.
- There is a small overhead over C code.
  - About 15-20% slower.

# BPF - Berkeley Packet Filter

The BSD Packet Filter: A New Architecture for User-level Packet Capture by Steven McCanne and Van Jacobson, 1992.

---

- Raw interface (*/dev/bpfN*) to datalink layers that supports packet filtering.
- Machine language for the BPF virtual machine.
- Comes with high-level filter language in the *libpcap* library.
- Programs like *tcpdump* send filter programs to the kernel via raw device.
- Machine language is usually interpreted, but can be compiled!

# BPF Machine Language

- From outside, bpf program can be viewed as a pure leaf function: single entry, no nested calls and no side effects.
- *A* and *X* registers.
- Stack: 16 32bit cells *M[0]*...*M[15]*.
- Simple instructions (aka RISC) with one exception.
- Forward jumps: compare and jump, jump always.
- No backward jumps, thus, no loops.

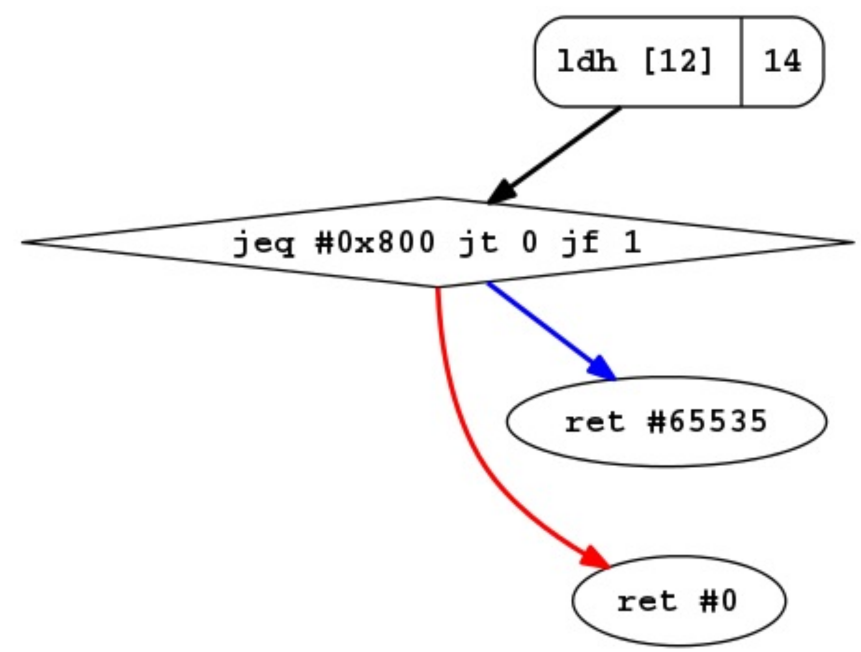
# BPF Machine Language

- Load byte, halfword and word from a packet: *ldb[9] ldh[9] ld[9]*.
- Indexed loads: *ldh[x+9]*.
- NetBSD doesn't allow wrap-around in indexed loads: *ld[x +0xffffffff]* doesn't do *ld[x-1]*, it aborts a program.
- Arithmetic operations: *add sub mul div neg and or lsh rsh*.
- Two extensions in NetBSD: coprocessor functions and external memory. Side effects. Only available in the kernel.

# Filter Programs

# tcpdump ip

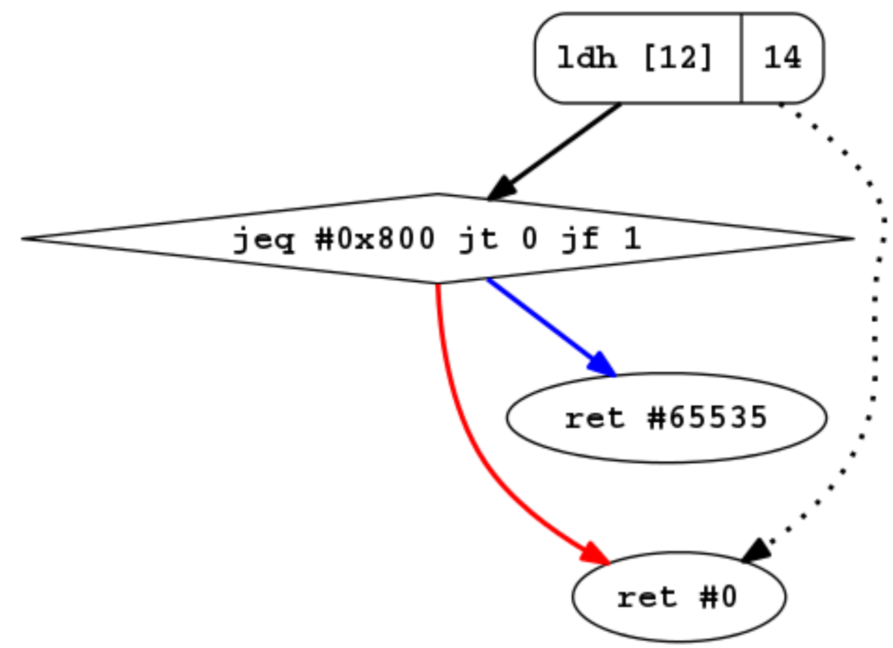
```
ldh [12]

jeq #0x800 jt 0 jf 1

ret #65535

ret #0
```



# Filter Programs

*# tcpdump ip*

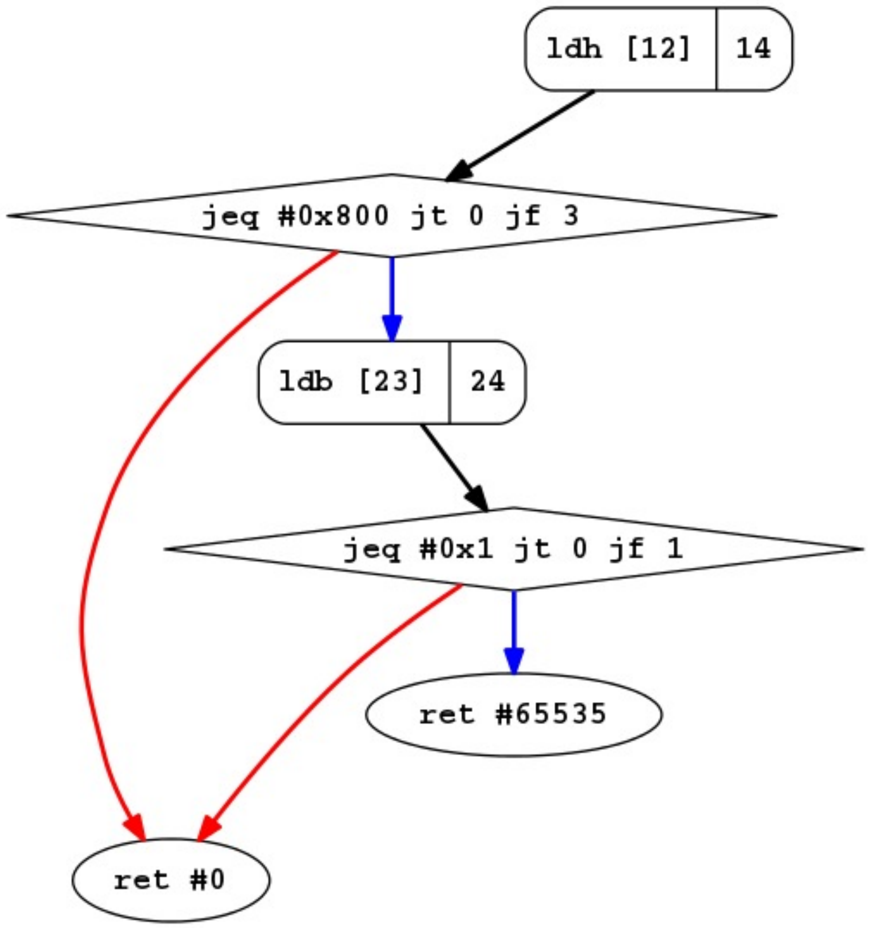
```
ldh [12]

jeq #0x800 jt 0 jf 1

ret #65535

ret #0
```



# Filter Programs

*# tcpdump icmp*

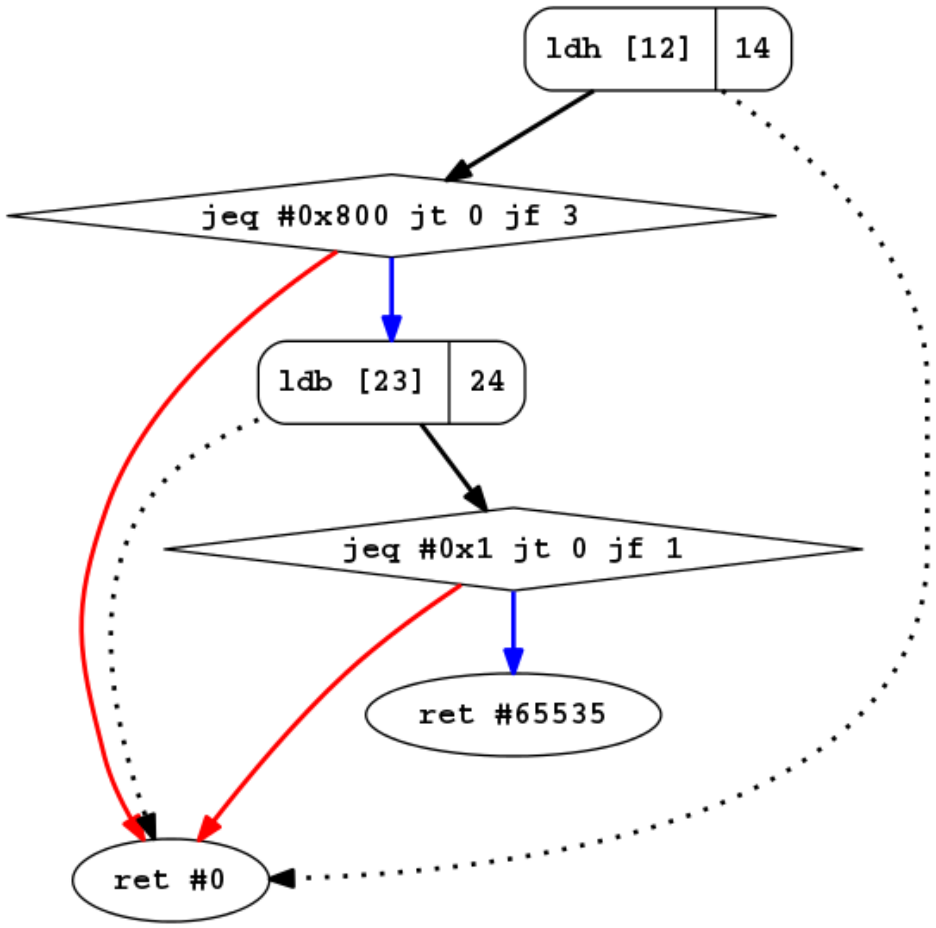
```
ldh [12]

jeq #0x800 jt 0 jf 3

ldb [23]

jeq #0x1 jt 0 jf 1

ret #65535

ret #0
```



# Filter Programs

# tcpdump icmp

```
ldh [12]
jeq #0x800 jt 0 jf 3
ldb [23]
jeq #0x1 jt 0 jf 1
ret #65535
ret #0
```

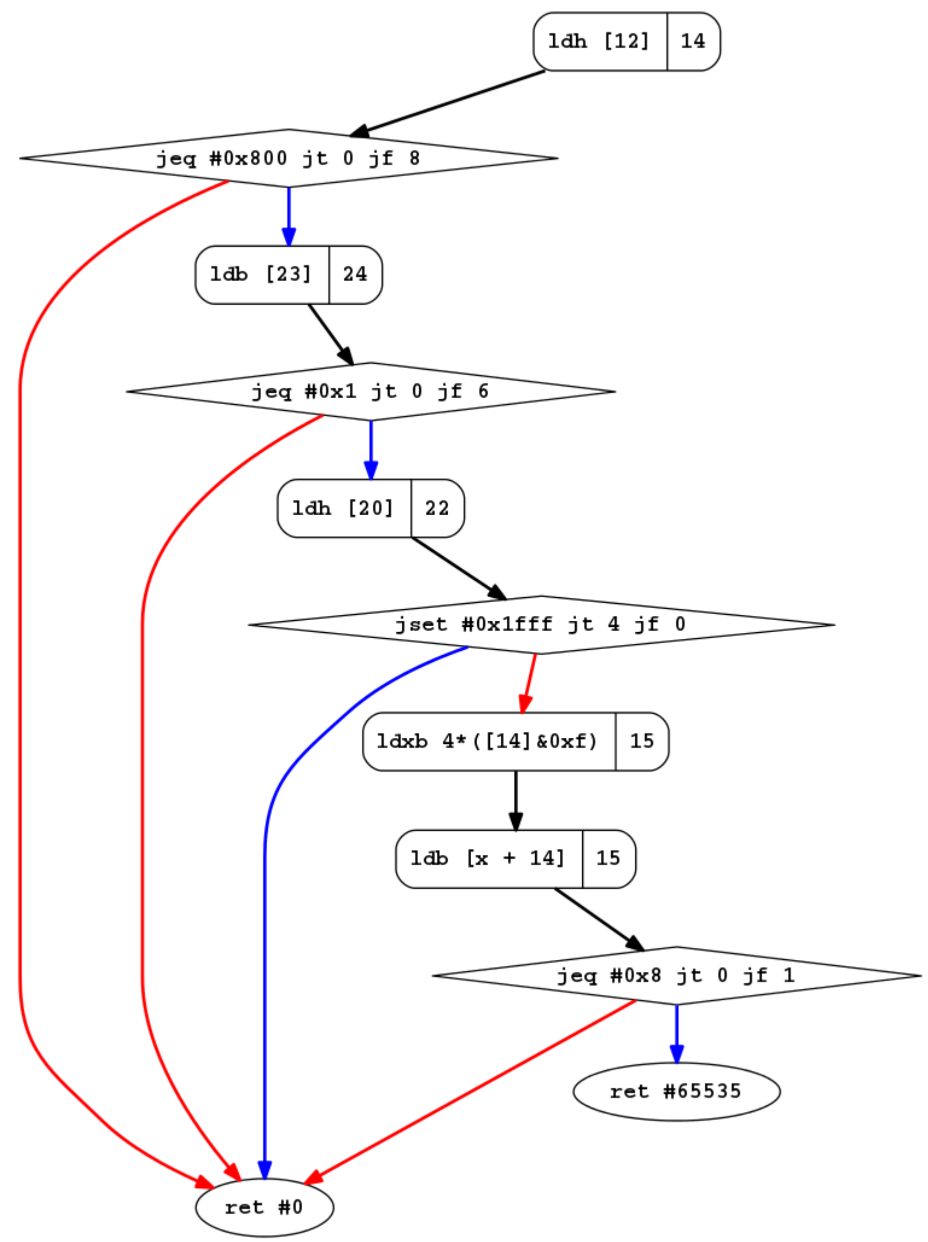


# Filter Programs

Ping requests:

*# tcpdump 'icmp[icmptype] = icmp-echo'*

```
ldh [12]
jeq #0x800 jt 0 jf 8
ldb [23]
jeq #0x1 jt 0 jf 6
ldh [20]
jset #0x1fff jt 4 jf 0
ldxb 4*([14]&0xf)
ldb [x + 14]
jeq #0x8 jt 0 jf 1
ret #65535
ret #0
```

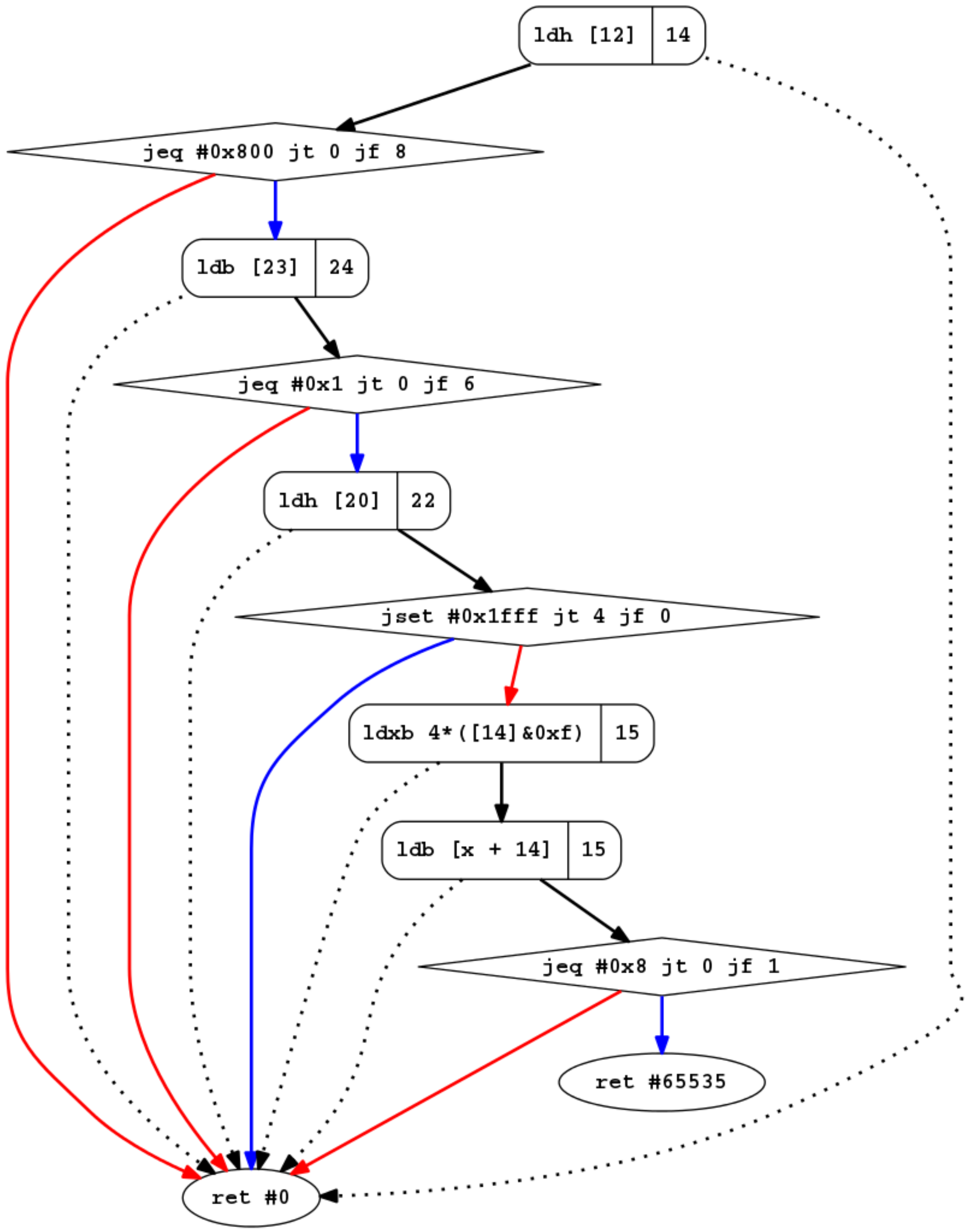


# Filter Programs

Ping requests:

*# tcpdump 'icmp[icmptype] = icmp-echo'*

```
ldh [12]
jeq #0x800 jt 0 jf 8
ldb [23]
jeq #0x1 jt 0 jf 6
ldh [20]
jset #0x1fff jt 4 jf 0
ldxb 4*([14]&0xf)
ldb [x + 14]
jeq #0x8 jt 0 jf 1
ret #65535
ret #0
```



# SLJIT Stack-less JIT

- Multi-platform BSD-licensed C library for code generation
  - Intel-x86 32, AMD-x86 64,
  - ARM 32 (ARM-v5, ARM-v7 and Thumb2), ARM 64,
  - PowerPC 32, PowerPC 64,
  - MIPS 32 (III, R1), MIPS 64 (III, R1),
  - SPARC 32,
  - Tilera TILE-Gx.
- Written by Zoltán Herczeg.
- TILE-Gx port by Jiong Wang on behalf of Tilera Corporation.

# SLJIT Stack-less JIT

- Like asm but each instruction is API function.
- At least 10 registers (some emulated on some platforms).
- Scratch registers (R0-R9), Saved registers (S0-S9).
- Stack-less: no stack for temporaries when sljit emulates instructions.
- Stack is available via *SLJIT_SP* register and *sljit_get_local_base*() function.
- Common instructions: mov, arithmetic, logical, bitops, comparisons.
- Labels and jumps.
- Jumps and constants can be updated after the code is generated.

# SLJIT Stack-less JIT

- *SLJIT_MOV*: move data between registers, register and memory.
- Load/store width: byte, half (16bit), int (32bit), word (32bit or 64bit).
- Addressing modes: [imm], [reg+imm], [reg+(reg<<imm)].
- *SLJIT_INT_OP*: 32bit mode on 64bit platforms.
- 3-operand instructions.
- Double and single-precision floating point.
- Call external functions with up to three arguments.
- Fast calls.

# Example: Fast 32bit Division

```
uint32_t mul;
uint8_t sh1, sh2;
fast_divide32_prepare(17, &mul, &sh1, &sh2);


uint32_t fast_divide32(uint32_t v) {
    uint64_t v64 = v;
    uint32_t t = (v64 * mul) >> 32;
    return (t + ((v - t) >> sh1)) >> sh2;
}
```

# Fast Division - 64bit Arch

```lua
local mul, sh1, sh2 = ...

return sljit.create_compiler()

    :emit_enter{args=1, saveds=1, scratches=1}

    :emit_op2('MUL', 'R0', 'S0', sljit.imm(mul))

    :emit_op2('LSHR', 'R0', 'R0', sljit.imm(32))

    :emit_op2('ISUB', 'S0', 'S0', 'R0')

    :emit_op2('ILSHR', 'S0', 'S0', sljit.imm(sh1))

    :emit_op2('IADD', 'R0', 'R0', 'S0')

    :emit_op2('ILSHR', 'R0', 'R0', sljit.imm(sh2))

    :emit_return('MOV_UI', 'R0')
```

```c
compiler = sljit_create_compiler();
if (compiler == NULL)
    goto fail;
status = sljit_emit_enter(compiler,
    0, 1, 1, 1, 0, 0, 0);
if (status != SLJIT_SUCCESS)
    goto fail;
status = sljit_emit_op2(compiler,
    SLJIT_MUL, SLJIT_R0, 0,
    SLJIT_S0, 0, SLJIT_IMM, mul);
if (status != SLJIT_SUCCESS)
    goto fail;
status = sljit_emit_op2(compiler,
    SLJIT_LSHR, SLJIT_R0, 0,
    SLJIT_R0, 0, SLJIT_IMM, 32);
```

# Fast Division - 64bit Arch

```
local mul, sh1, sh2 = ...

return sljit.create_compiler()
    :emit_enter{args=1, saveds=1, scratches=1}
    :emit_op2('MUL', 'R0', 'S0', sljit.imm(mul))
    :emit_op2('LSHR', 'R0', 'R0', sljit.imm(32))
    :emit_op2('ISUB', 'S0', 'S0', 'R0')
    :emit_op2('ILSHR', 'S0', 'S0', sljit.imm(sh1))
    :emit_op2('IADD', 'R0', 'R0', 'S0')
    :emit_op2('ILSHR', 'R0', 'R0', sljit.imm(sh2))
    :emit_return('MOV_UI', 'R0')
```

```
compiler = sljit_create_compiler();
if (compiler == NULL)
    goto fail;
status = sljit_emit_enter(compiler,
    0, 1, 1, 1, 0, 0, 0);
if (status != SLJIT_SUCCESS)
    goto fail;
status = sljit_emit_op2(compiler,
    SLJIT_MUL, SLJIT_R0, 0,
    SLJIT_S0, 0, SLJIT_IMM, mul);
if (status != SLJIT_SUCCESS)
    goto fail;
status = sljit_emit_op2(compiler,
    SLJIT_LSHR, SLJIT_R0, 0,
    SLJIT_R0, 0, SLJIT_IMM, 32);
```

# Fast Division - 64bit Arch

```lua
local mul, sh1, sh2 = ...

return sljit.create_compiler()

    :emit_enter{args=1, saveds=1, scratches=1}

    :emit_op2('MUL', 'R0', 'S0', sljit.imm(mul))

    :emit_op2('LSHR', 'R0', 'R0', sljit.imm(32))

    :emit_op2('ISUB', 'S0', 'S0', 'R0')

    :emit_op2('ILSHR', 'S0', 'S0', sljit.imm(sh1))

    :emit_op2('IADD', 'R0', 'R0', 'S0')

    :emit_op2('ILSHR', 'R0', 'R0', sljit.imm(sh2))

    :emit_return('MOV_UI', 'R0')
```

```c
compiler = sljit_create_compiler();
if (compiler == NULL)
    goto fail;
status = sljit_emit_enter(compiler,
    0, 1, 1, 1, 0, 0, 0);
if (status != SLJIT_SUCCESS)
    goto fail;
status = sljit_emit_op2(compiler,
    SLJIT_MUL, SLJIT_R0, 0,
    SLJIT_S0, 0, SLJIT_IMM, mul);
if (status != SLJIT_SUCCESS)
    goto fail;
status = sljit_emit_op2(compiler,
    SLJIT_LSHR, SLJIT_R0, 0,
    SLJIT_R0, 0, SLJIT_IMM, 32);
```

# Fast Division - 64bit Arch

```lua
local mul, sh1, sh2 = ...

return sljit.create_compiler()

    :emit_enter{args=1, saveds=1, scratches=1}

    :emit_op2('MUL', 'R0', 'S0', sljit.imm(mul))

    :emit_op2('LSHR', 'R0', 'R0', sljit.imm(32))

    :emit_op2('ISUB', 'S0', 'S0', 'R0')

    :emit_op2('ILSHR', 'S0', 'S0', sljit.imm(sh1))

    :emit_op2('IADD', 'R0', 'R0', 'S0')

    :emit_op2('ILSHR', 'R0', 'R0', sljit.imm(sh2))

    :emit_return('MOV_UI', 'R0')
```

```c
compiler = sljit_create_compiler();
if (compiler == NULL)
    goto fail;
status = sljit_emit_enter(compiler,
    0, 1, 1, 1, 0, 0, 0);
if (status != SLJIT_SUCCESS)
    goto fail;
status = sljit_emit_op2(compiler,
    SLJIT_MUL, SLJIT_R0, 0,
    SLJIT_S0, 0, SLJIT_IMM, mul);
if (status != SLJIT_SUCCESS)
    goto fail;
status = sljit_emit_op2(compiler,
    SLJIT_LSHR, SLJIT_R0, 0,
    SLJIT_R0, 0, SLJIT_IMM, 32);
```

# SLJIT vs GCC

```c
// Push Lua program to the stack
lua_pushinteger(L, mul);
lua_pushinteger(L, sh1);
lua_pushinteger(L, sh2);
lua_call(L, 3, 1);
compiler = luaSljit_tocompiler(L, -1);
fn = sljit_generate_code(compiler);
```

```c
uint32_t mul;
uint8_t sh1, sh2;
uint32_t fast_divide32(uint32_t v) {
    uint64_t v64 = v;
    uint32_t _t = (v64 * mul) >> 32;
    return (t + ((v - t) >> sh1)) >> sh2;
}
```

```c
uint32_t div17(uint32_t value) {
    return value / 17;
}
```

# SLJIT vs GCC

```c
uint32_t div17(uint32_t value) {
    return value / 17;
}
```

```c
// Push Lua program to the stack
lua_pushinteger(L, mul);
lua_pushinteger(L, sh1);
lua_pushinteger(L, sh2);
lua_call(L, 3, 1);
compiler = luaSljit_tocompiler(L, -1);
fn = sljit_generate_code(compiler);
```

```c
uint32_t mul;
uint8_t sh1, sh2;
uint32_t fast_divide32(uint32_t v) {
    uint64_t v64 = v;
    uint32_t _t = (v64 * mul) >> 32;
    return (t + ((v - t) >> sh1)) >> sh2;
}
```

# SLJIT vs GCC

```
// Push Lua program to the stack

lua_pushinteger(L, mul);

lua_pushinteger(L, sh1);

lua_pushinteger(L, sh2);

lua_call(L, 3, 1);

compiler = luaSljit_tocompiler(L, -1);

fn = sljit_generate_code(compiler);
```

```
uint32_t div17(uint32_t value) {
    return value / 17;
}
```

```
uint32_t mul;
uint8_t sh1, sh2;
uint32_t fast_divide32(uint32_t v) {
    uint64_t v64 = v;
    uint32_t _t = (v64 * mul) >> 32;
    return (t + ((v - t) >> sh1)) >> sh2;
}
```

# SLJIT vs GCC

```c
uint32_t div17(uint32_t value) {
    return value / 17;
}
```

```c
uint32_t mul;
uint8_t sh1, sh2;
uint32_t fast_divide32(uint32_t v) {
    uint64_t v64 = v;
    uint32_t _t = (v64 * mul) >> 32;
    return (t + ((v - t) >> sh1)) >> sh2;
}
```

```c
// Push Lua program to the stack
lua_pushinteger(L, mul);
lua_pushinteger(L, sh1);
lua_pushinteger(L, sh2);
lua_call(L, 3, 1);
compiler = luaSljit_tocompiler(L, -1);
fn = sljit_generate_code(compiler);
```

# SLJIT vs GCC

```c
uint32_t div17(uint32_t value) {
    return value / 17;
}
```

```c
// Push Lua program to the stack
lua_pushinteger(L, mul);
lua_pushinteger(L, sh1);
lua_pushinteger(L, sh2);
lua_call(L, 3, 1);
compiler = luaSljit_tocompiler(L, -1);
fn = sljit_generate_code(compiler);
```

```c
uint32_t mul;
uint8_t sh1, sh2;
uint32_t fast_divide32(uint32_t v) {
    uint64_t v64 = v;
    uint32_t _t = (v64 * mul) >> 32;
    return (t + ((v - t) >> sh1)) >> sh2;
}
```

# SLJIT vs GCC

```
// Push Lua program to the stack
lua_pushinteger(L, mul);
lua_pushinteger(L, sh1);
lua_pushinteger(L, sh2);
lua_call(L, 3, 1);
compiler = luaSljit_tocompiler(L, -1);
fn = sljit_generate_code(compiler);
```

```
uint32_t mul;
uint8_t sh1, sh2;
uint32_t fast_divide32(uint32_t v) {
    uint64_t v64 = v;
    uint32_t _t = (v64 * mul) >> 32;
    return (t + ((v - t) >> sh1)) >> sh2;
}
```

```
uint32_t div17(uint32_t value) {
    return value / 17;
}
```

# SLJIT vs GCC

| | |
|---|---|
| push %rbx | |
| mov %rdi,%rbx | mov %edi,%eax |
| sub $0x10,%rsp | |
| movabs $0xe1e1e1e2,%rdi | mov 0x200ac0(%rip),%edx # 0x601928 <mul> |
| mov %rbx,%rax | |
| imul %rdi,%rax | imul %rdx,%rax |
| shr $0x20,%rax | shr $0x20,%rax |
| sub %eax,%ebx | sub %eax,%edi |
| | movzbl 0x200ab4(%rip),%ecx # 0x60192d <sh1> |
| shr %ebx | shr %cl,%edi |
| add %ebx,%eax | add %edi,%eax |
| | movzbl 0x200aa8(%rip),%ecx # 0x60192c <sh2> |
| shr $0x4,%eax | shr %cl,%eax |
| mov %eax,%eax | |
| add $0x10,%rsp | |
| pop %rbx | |

```
mov    %edi,%eax
mov    $0xf0f0f0f1,%edx
mul    %edx
shr    $0x4,%edx
mov    %edx,%eax
```

# Chekhov's Gun

If you see flow graphs in the first part of a presentation, the second or the third part will be about optimisations.

# BPF JIT Optimisations

- More careful about optimisations because it runs in the kernel.
- Don't optimise when the same optimisation can be achieved by changing a filter program.
    1. Exception: unreachable instructions.
    2. Exception: init A and X if they might be used uninitialised.
- Fixed number of passes through a filter program.
- Rule of thumb: optimise if it positively affects real programs, or if optimisation is "for free", e.g. a side effect of some other optimisation.

# Optimisation Passes

1. Initialisation pass: set members of auxiliary optimisation related data.
2. Flow pass
    - trivial hints (e.g. *X* is never used),
    - find use-before-init among registers and memwords,
    - detect unreachable instructions,
    - setup jump lists (without additional memory allocations).
3. Array Bounds Check (ABC) elimination backward pass.
4. ABC forward pass.

# ABC Optimisation

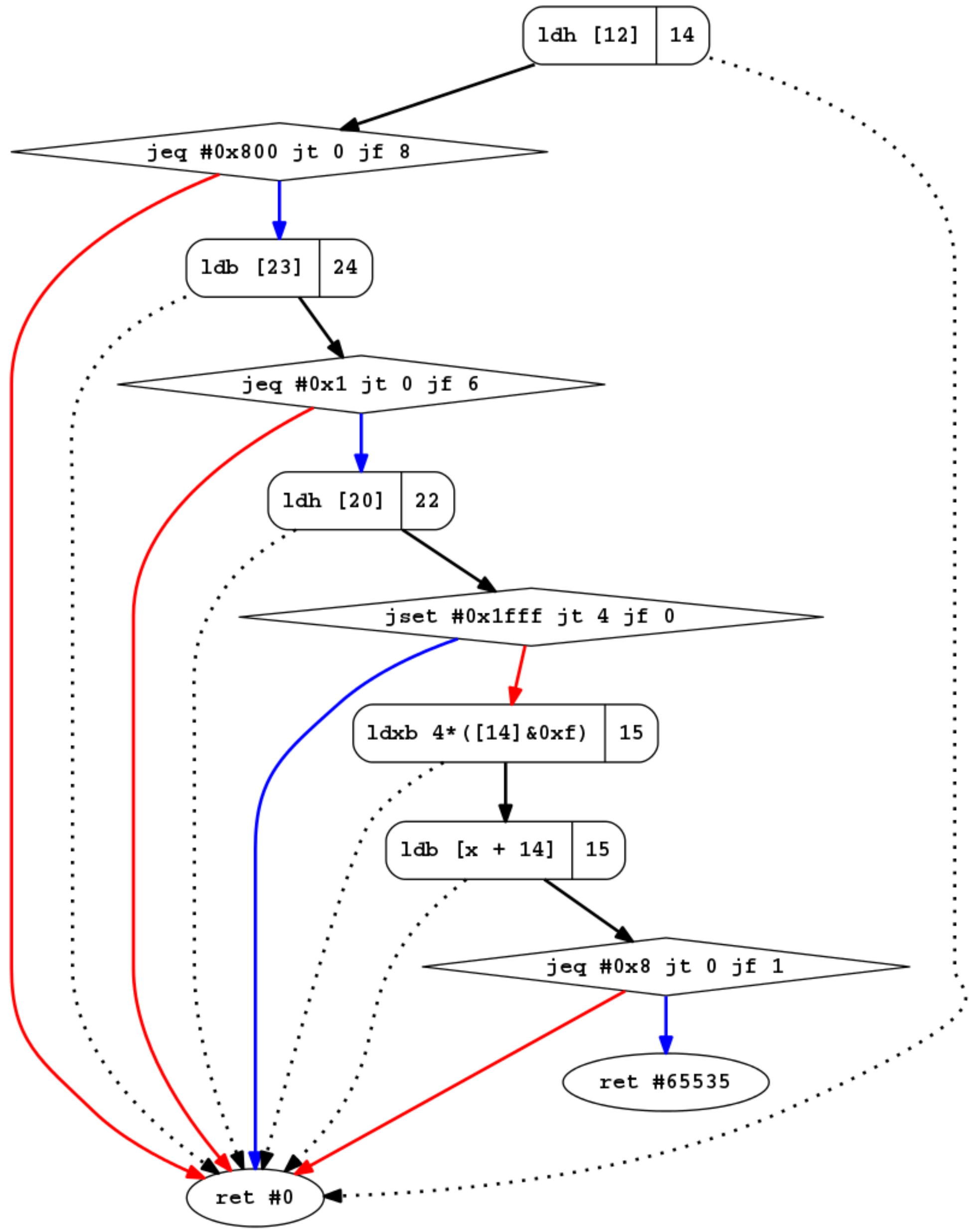
- ABC is Array Bounds Check elimination.
- In a context of bpf, it applies to packet reads.
- Filter programs often read packet bytes at increasing offsets,
  - for instance, when going through protocol layers.
- If program is going to read packet bytes at higher offsets later, why not check the packet length early?
- If there are side effects (there are none in classical bpf), the optimisation doesn't apply.

# ABC Optimisation

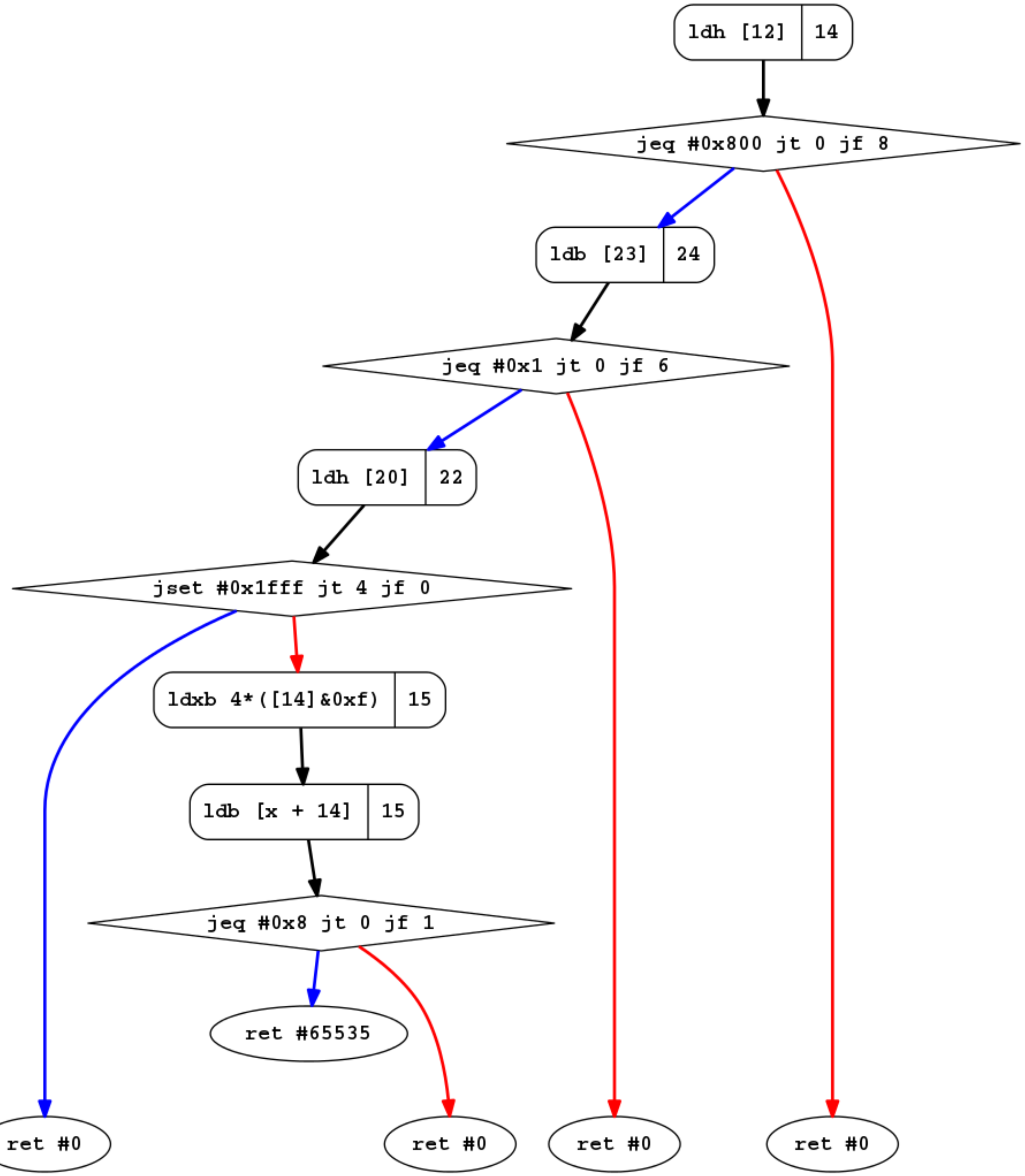
Ping requests:

*# tcpdump 'icmp[icmptype] = icmp-echo'*

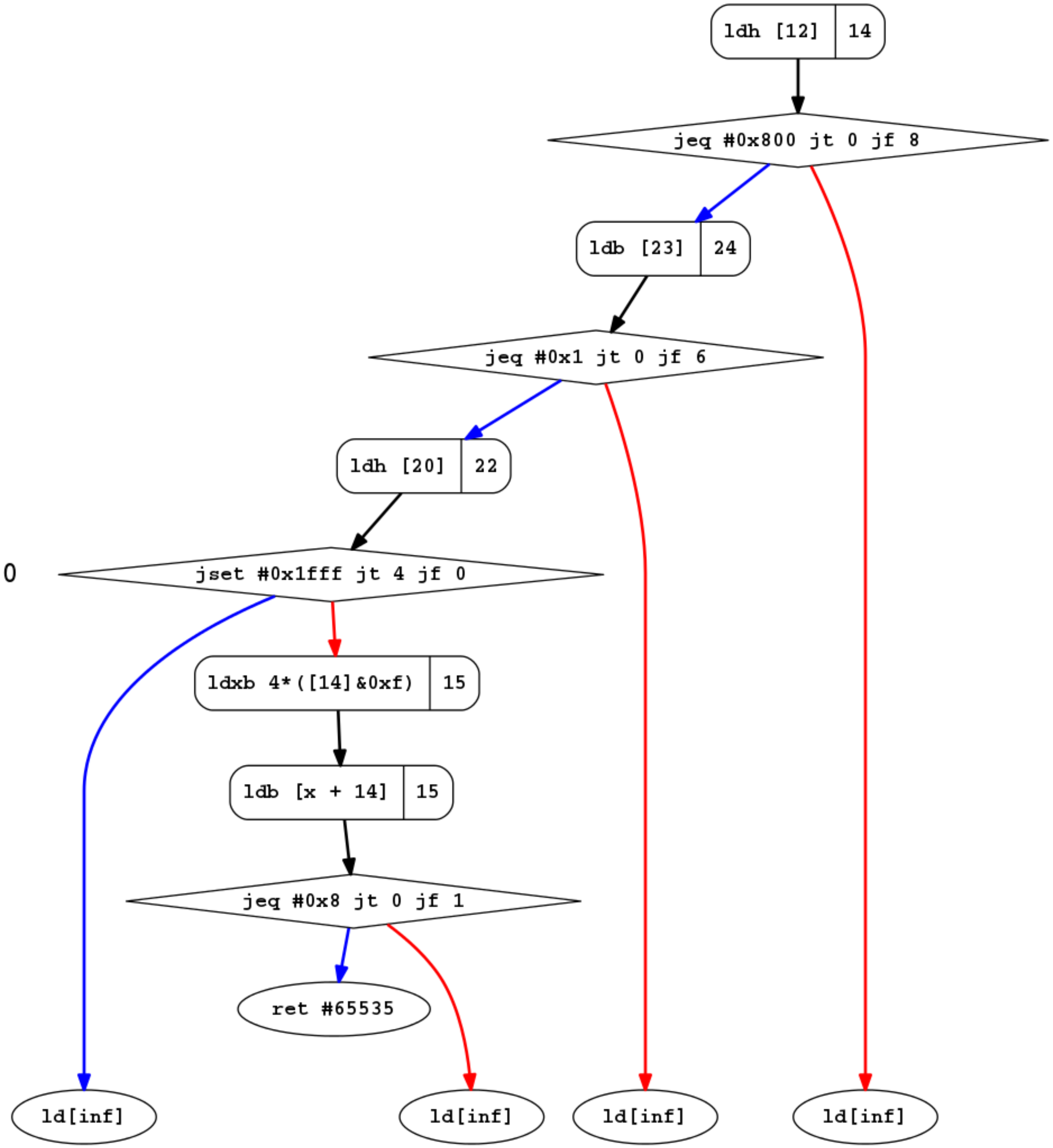
```
ldh [12]
jeq #0x800 jt 0 jf 8
ldb [23]
jeq #0x1 jt 0 jf 6
ldh [20]
jset #0x1fff jt 4 jf 0
ldxb 4*([14]&0xf)
ldb [x + 14]
jeq #0x8 jt 0 jf 1
ret #65535
ret #0
```



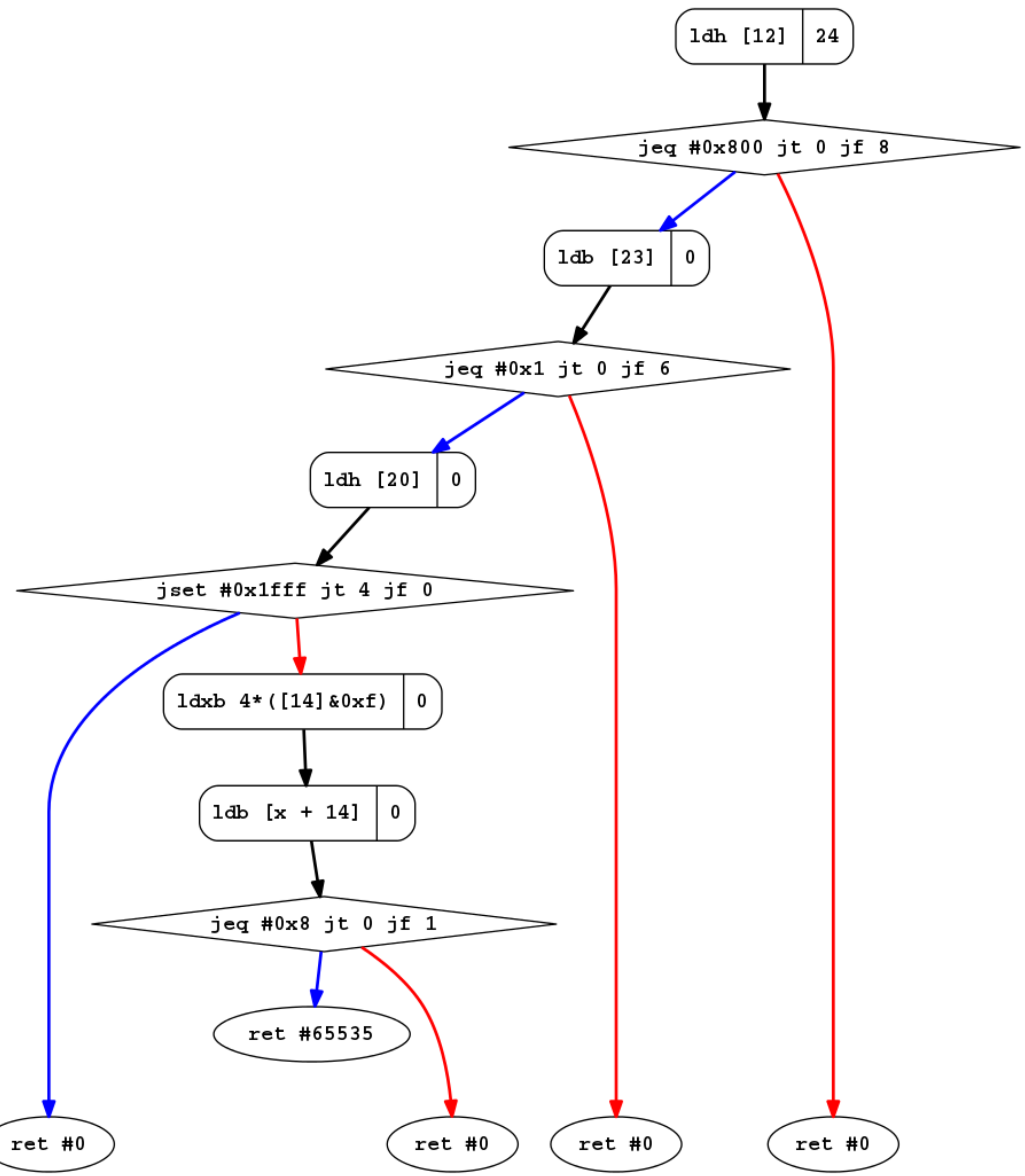
```
ldh [12]
jeq #0x800 jt 0 jf 8
ldb [23]
jeq #0x1 jt 0 jf 6
ldh [20]
jset #0x1fff jt 4 jf 0
ldxb 4*([14]&0xf)
ldb [x + 14]
jeq #0x8 jt 0 jf 1
ret #65535
ret #0
```



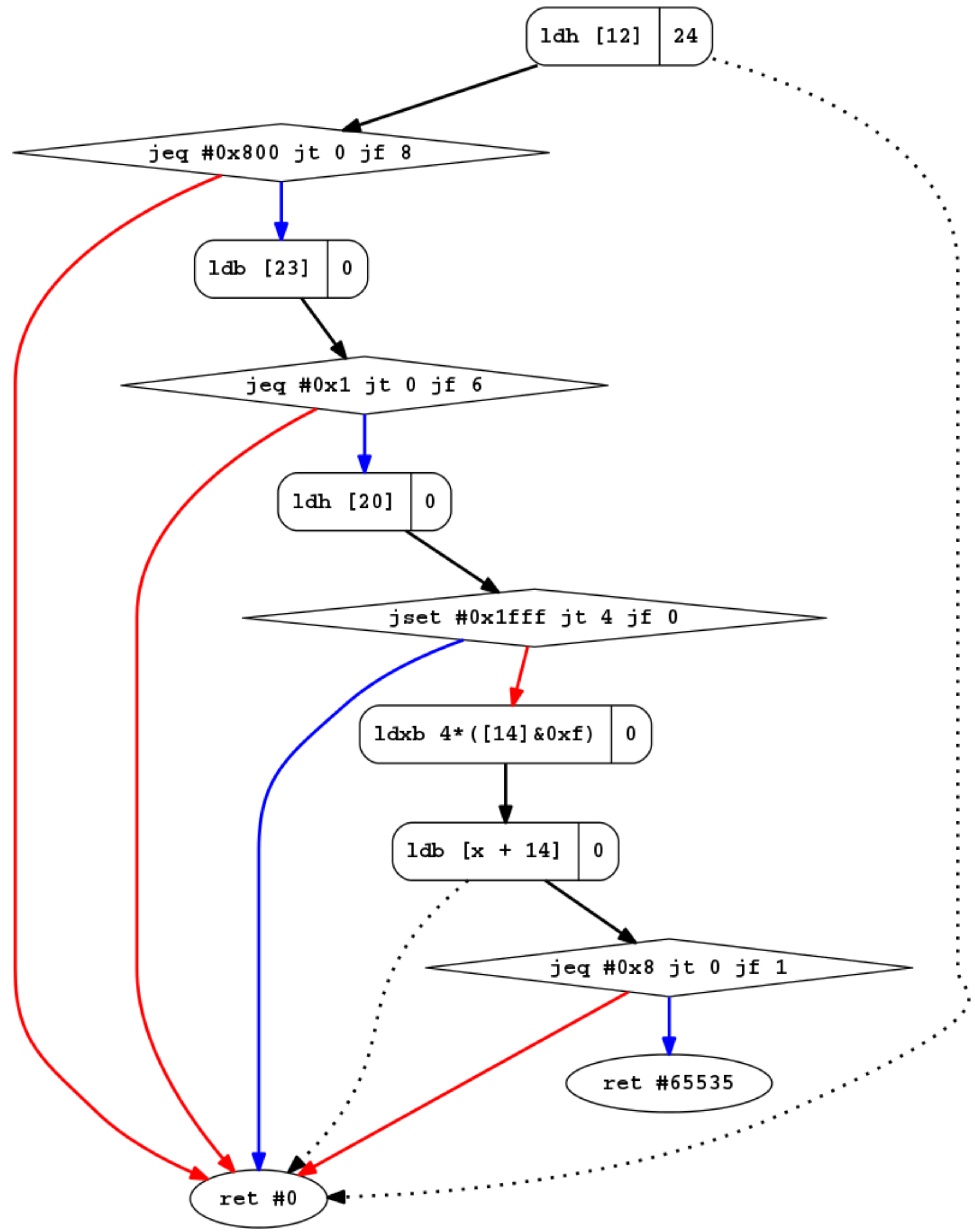
```
ldh [12]
jeq #0x800 jt 0 jf 8
ldb [23]
jeq #0x1 jt 0 jf 6
ldh [20]
jset #0x1fff jt 4 jf 0
ldxb 4*([14]&0xf)
ldb [x + 14]
jeq #0x8 jt 0 jf 1
ret #65535
ret #0
```



```
ldh [12]
jeq #0x800 jt 0 jf 8
ldb [23]
jeq #0x1 jt 0 jf 6
ldh [20]
jset #0x1fff jt 4 jf 0
ldxb 4*([14]&0xf)
ldb [x + 14]
jeq #0x8 jt 0 jf 1
ret #65535
ret #0
```



```
ldh [12]

jeq #0x800 jt 0 jf 8

ldb [23]

jeq #0x1 jt 0 jf 6

ldh [20]

jset #0x1fff jt 4 jf 0

ldxb 4*([14]&0xf)

ldb [x + 14]

jeq #0x8 jt 0 jf 1

ret #65535

ret #0
```



# Future Optimisations

- Merge two instructions into one.
    - Loads are often followed by jumps.
    - Both branches often load new values into *A* or don't use *A* at all.
    - Some instruction sets accept memory operand in comparisons.
- Assign sljit registers to BPF registers dynamically.
    - Some sljit registers are more expensive,
    - E.g. simulated registers or registers with simulated access modes.
    - Works best when applied after instruction pairs are merged. See the previous slides.

# Future Optimisations

- Packets are always contiguous in userspace but the kernel stores bigger packets in *mbuf* chains.
    - To access *mbuf* data, special *m_xbyte()*, *m_xhalf()* and *m_xword()* functions are called.
    - Those functions always check packet length.
    - Therefore, ABC checks are redundant.
- Majority of filter programs check packet bytes at low offsets that point to the first chunk.
- Fast path: if the first chunk of *mbuf* chain is big enough, load it.
    - No *m_xbyte()*, *m_xhalf() and m_xword()* calls for absolute loads.
    - Indexed loads may call those functions if the *X* register stores a big value.
- Slow path: call those functions for all loads.

# Testing Notes

- It's hard to write unit tests when observable result is a single number.
- Testing of optimisations is especially hard.
- Consider exposing intermediate representation and testing it.
- Graphs generated from intermediate representations was an important milestone.
- Would be nice to have tests in Lua.
- Userspace tests only cover contiguous buffers.
- How to run unit-tests in the kernel?

# Testing Notes

- Rump is a modular framework designed to run parts of the kernel in userspace.
- It's possible to configure a simple network between two rump processes, send a single packet and detect a leak (*bpfwriteleak* test).
- Most unit-tests in bpfjit are even more modular: they only borrow *mbuf* from the network stack and use rump versions of *sljit* and *bpfjit*.
- 114 unit tests in userspace.
- The same set of tests for rump kernel plus 20 *mbuf* related tests.

Questions?