

# Linux вам не компилятор: ускоряем eBPF в полтора раза с помощью LLVM

Илья Гладышев

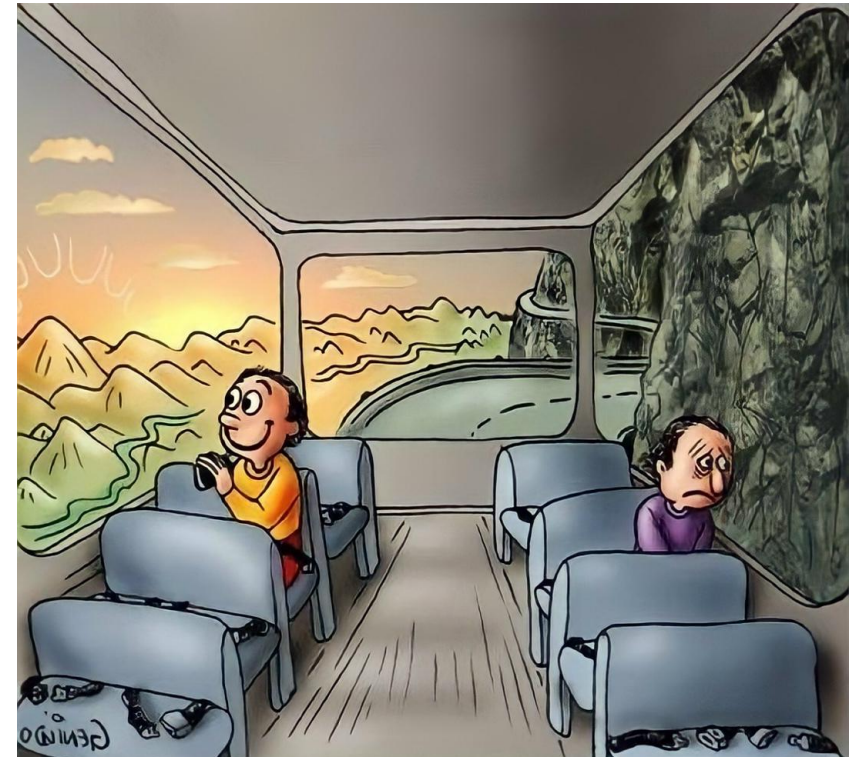
# В этом докладе

- Устройство eVRF на данный момент  
Байткод и его компиляция  
Где спрятаны неоптимальности



# В этом докладе

- Устройство eBPF на данный момент  
Байткод и его компиляция  
Где спрятаны неоптимальности
- Как притащить LLVM  
Архитектура и нюансы  
Результаты



# Что такое eBPF

- Виртуальная машина внутри ядра
  - “Реактивные” расширения (хуки, фильтры)
- Удобное взаимодействие с userspace (maps)
- Гарантируется отсутствие UB
  - Детерминированное исполнение
  - Изоляция багов



# Примеры eBPF

- sched\_ext

  - Планировщики CPU на eBPF

  - Эксперименты с topology-aware scheduling

- Cilium

  - Сетевой стек для Kubernetes

  - Pod-to-pod, load balancing, ...

  - Полностью в eBPF, “минуя” ядро

# Пишем XDP фильтр

```
const int MIN_SIZE = 100;
```

```
SEC("xdp")
```

```
int xdp_pass(struct xdp_md *ctx) {  
    void *end = (void *) (long) ctx->data_end;  
    void *data = (void *) (long) ctx->data;  
  
    if (end - data < MIN_SIZE) {  
        return XDP_DROP;  
    }  
  
    return XDP_PASS;  
}
```

# Пишем XDP фильтр

```
const volatile int MIN_SIZE = 100;
```

```
SEC("xdp")
```

```
int xdp_pass(struct xdp_md *ctx) {  
    void *end = (void *) (long) ctx->data_end;  
    void *data = (void *) (long) ctx->data;  
  
    if (end - data < MIN_SIZE) {  
        return XDP_DROP;  
    }  
  
    return XDP_PASS;  
}
```

MIN\_SIZE читается из  
.rodata

Можем менять без  
КОМПИЛЯЦИИ

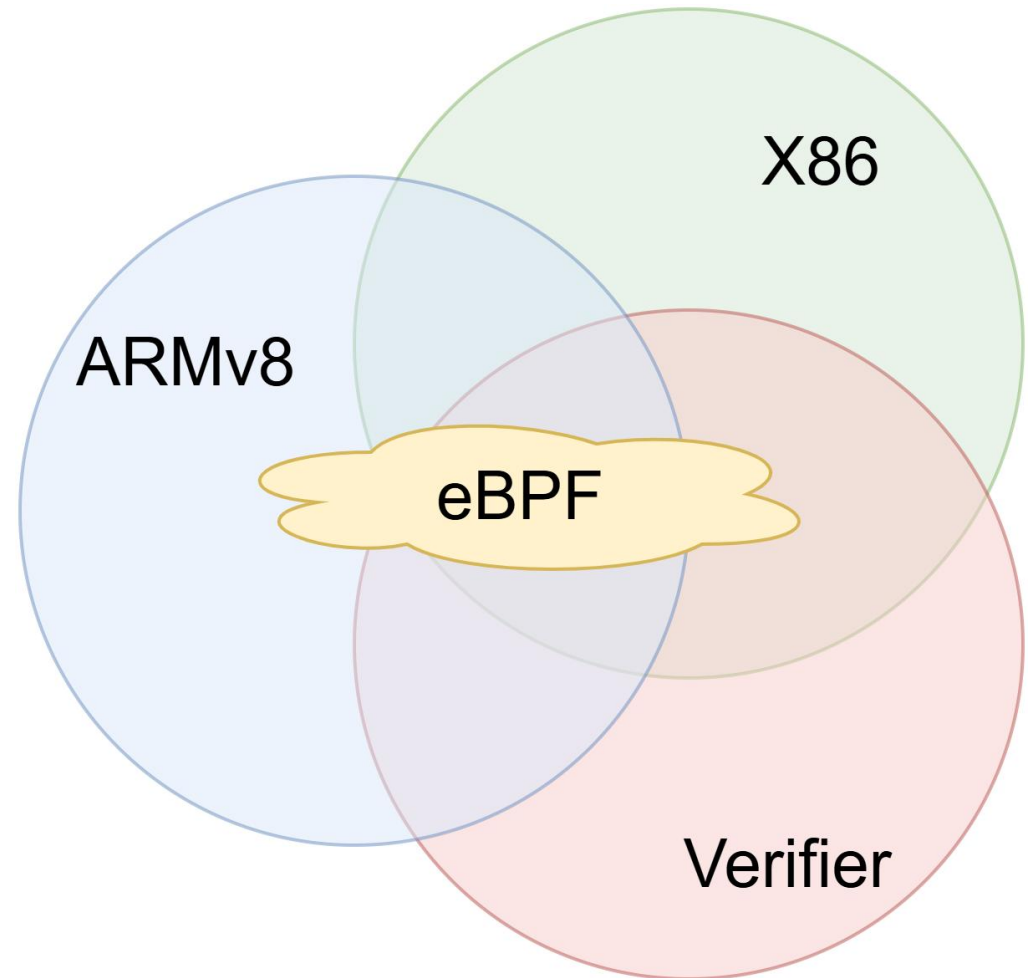
# Виртуальная машина eBPF

- 64-битная RISC архитектура
- 10 регистров + stack pointer
- Двухоперандные инструкции

R0	Возврат
R1-R5	Аргументы
R6-R9	Callee-saved
R10	Stack pointer

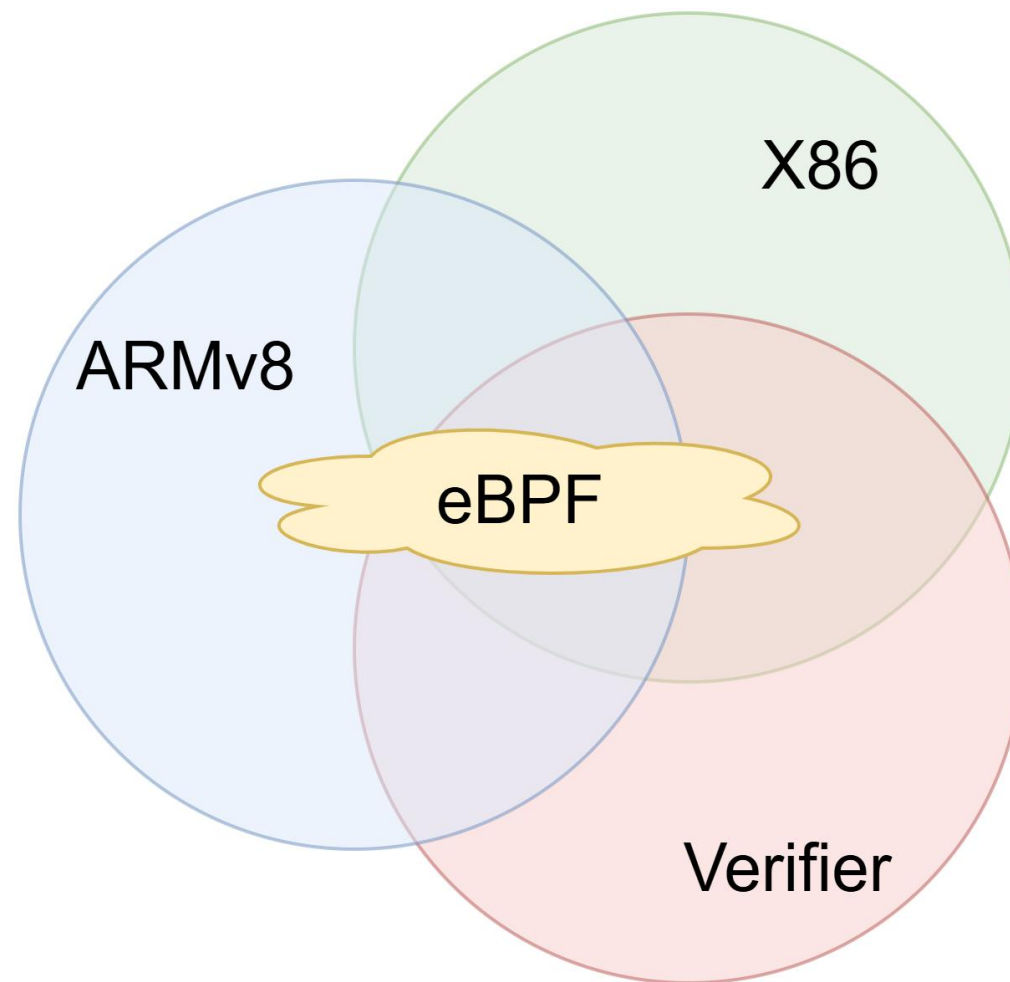
# Виртуальная машина eBPF

- ✓ ABI совместимо с основными архитектурами
  - Не требуется конвертация
- ✓ Просто верифицировать



# Виртуальная машина eBPF

- ✓ ABI совместимо с основными архитектурами
  - Не требуется конвертация
- ✓ Просто верифицировать
- ✗ Все архитектуры отображены одинаково плохо



# Пишем XDP фильтр

```
const volatile int MIN_SIZE = 100;
```

```
SEC("xdp")
```

```
int xdp_pass(struct xdp_md *ctx) {  
    void *end = (void *) (long) ctx->data_end;  
    void *data = (void *) (long) ctx->data;  
  
    if (end - data < MIN_SIZE) {  
        return XDP_DROP;  
    }  
  
    return XDP_PASS;  
}
```

```
xdp_pass:
```

```
w2 = *(u32 *) (r1 + 0)
```

```
w1 = *(u32 *) (r1 + 4)
```

```
r1 -= r2
```

```
r2 = MIN_SIZE ll
```

```
w2 = *(u32 *) (r2 + 0)
```

```
r2 <<= 32
```

```
r2 s>>= 32
```

```
w0 = 1
```

```
if r1 s< r2 goto LBB0_2
```

```
w0 = 2
```

```
LBB0_2:
```

```
exit
```

```
clang -target ebpf xdp.c -o xdp.bpf.o
```

# Загрузка eBPF



ELF



libbpf



bpf()

XDP

```
int xdp_pass(ctx)
```

.rodata

```
int MINSIZE
```

*meta*

```
symtab / rel
```

# Загрузка eBPF



ELF



libbpf



bpf()

- Собирает функции в программы

xdp\_one

xdp\_two

static  
foo()

static  
bar()

# Загрузка eBPF



ELF



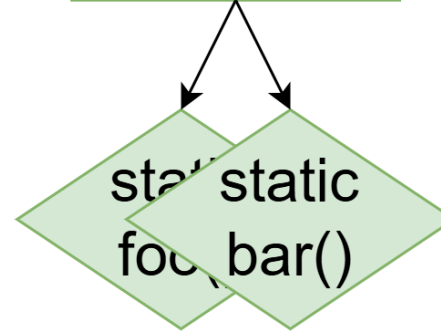
libbpf



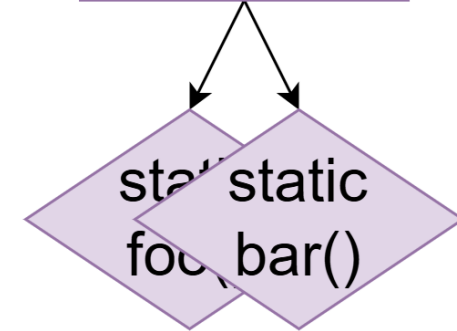
bpf()

- Собирает функции в программы

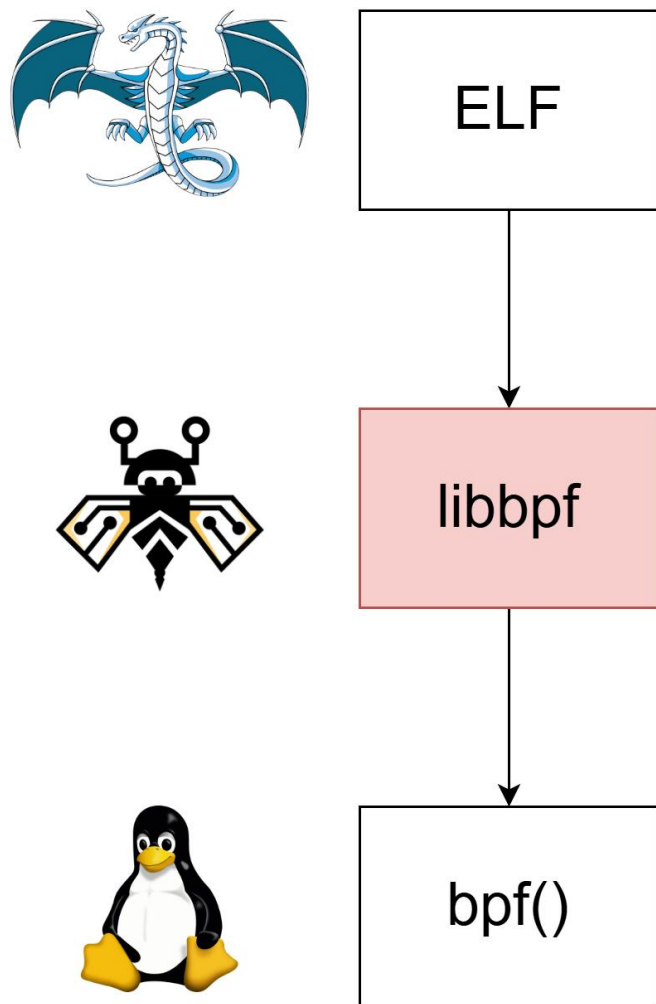
xdp\_one



xdp\_two



# Загрузка eBPF



- Собирает функции в программы
- Конвертирует rodata в eBPF map



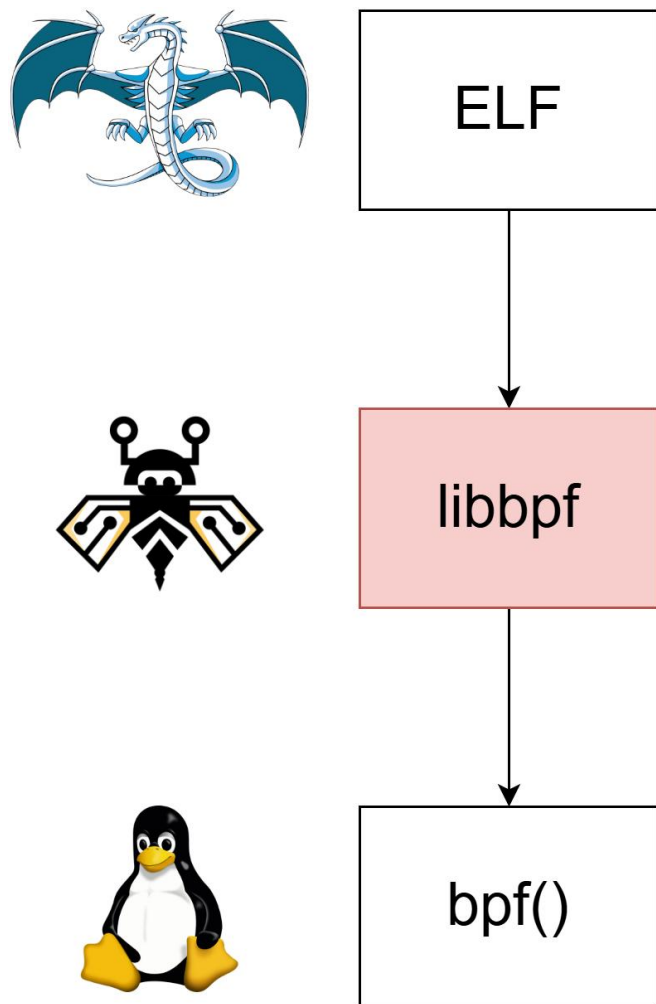
`.rodata`  
78 bytes



ArrayMap

size = 1  
[0] = <78 bytes>

# Загрузка eBPF



- Собирает функции в программы
- Конвертирует rodata в eBPF map



`.rodata`  
78 bytes

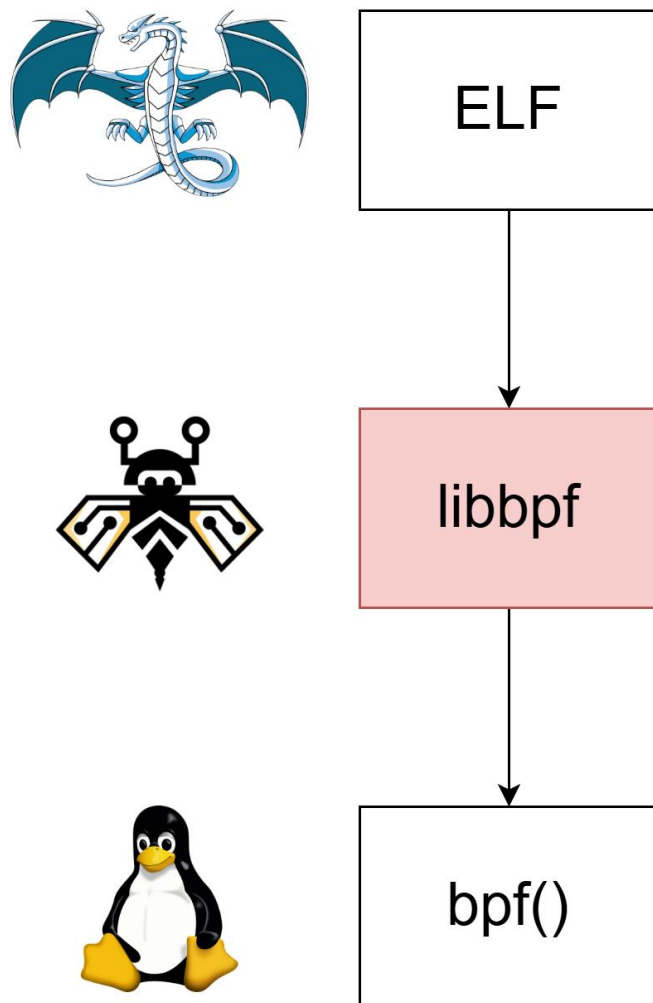


`ArrayMap`

`frozen`

`size = 1`  
`[0] = <78 bytes>`

# Загрузка eBPF



- Собирает функции в программы
- Конвертирует rodata в eBPF map



`.rodata`  
78 bytes

`r2=<.rodata+0x78>`



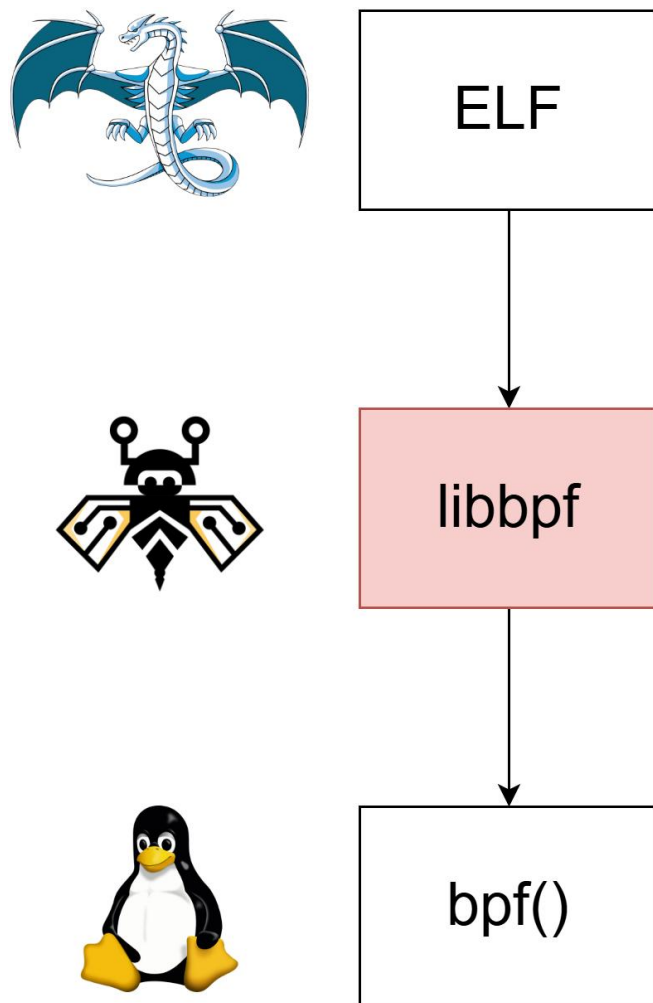
`ArrayMap`

`frozen`

`size = 1`  
`[0] = <78 bytes>`

`r2 = &map[0]+0x78`

# Загрузка eBPF



- Собирает функции в программы
- Конвертирует rodata в eBPF map



`.rodata`  
78 bytes

`r2=<.rodata+0x78>`



`ArrayMap`

frozen

`size = 1`  
`[0] = <78 bytes>`

`r2 = &map[0]+0x78`

`r2 = <const addr>`

# Загрузка eBPF



ELF



libbpf



bpf()

- Собирает функции в программы
- Конвертирует rodata в eBPF map
- Инициализирует ресурсы (map)

# Загрузка eBPF



ELF



libbpf



bpf()

- Собирает функции в программы
- Конвертирует rodata в eBPF map
- Инициализирует ресурсы (map)
- Передает байткод ядру

# Загрузка eBPF



ELF



libbpf



bpf()

- Проверяет корректность (отсутствие UB)
- Удаляет мертвый код
- Инлайн ядерных функций
- Транслирует в нативные инструкции

# Верификация

```
SEC("xdp")
```

```
int xdp_scream(struct xdp_md *ctx) {  
    void *pkt = (void *) (long) ctx->data;  
    void *pkt_end = (void *) (long) ctx->data_end;  
    u8 val = 0;  
  
    if (pkt_end > pkt + 0x36) {  
        val = *(u8 *) (pkt + 0x16);  
    }  
  
    return val == 0xAA ? XDP_PASS : XDP_DROP;  
}
```

# Верификация

xdp\_foo():

```
r2 = *(r1 + 4)
```

```
r1 = *(r1 + 0)
```

```
r3 = r1
```

```
r3 += 36
```

```
if r3 >= r2 jmp +3
```

```
r0 = *(r1 + 16)
```

# Верификация

xdp\_foo():

r1=cxt

```
r2 = *(r1 + 4)
```

```
r1 = *(r1 + 0)
```

```
r3 = r1
```

```
r3 += 36
```

```
if r3 >= r2 jmp +3
```

```
r0 = *(r1 + 16)
```

# Верификация

xdp\_foo():

r2 = \*(r1 + 4)

r1 = \*(r1 + 0)

r3 = r1

r3 += 36

if r3 >= r2 jmp +3

r0 = \*(r1 + 16)

r1=cxt

r2=pkt\_end

r1=pkt

# Верификация

`xdp_foo()` :

`r2 = *(r1 + 4)`

`r1 = *(r1 + 0)`

`r3 = r1`

`r3 += 36`

`if r3 >= r2 jmp +3`

`r0 = *(r1 + 16)`

`r1=cxt`

`r2=pkt_end`

`r1=pkt`

`r3=pkt`

`r3=pkt(off=36)`

# Верификация

`xdp_foo()` :

`r2 = *(r1 + 4)`

`r1 = *(r1 + 0)`

`r3 = r1`

`r3 += 36`

`if r3 >= r2 jmp +3`

`r0 = *(r1 + 16)`

`r1=cxt`

`r2=pkt_end`

`r1=pkt`

`r3=pkt`

`r3=pkt(off=36)`

`sizeof(pkt) > 36`

# Верификация

`xdp_foo()` :

`r2 = *(r1 + 4)`

`r1 = *(r1 + 0)`

`r3 = r1`

`r3 += 36`

`if r3 >= r2 jmp +3`

`r0 = *(r1 + 16)`

`r1=cxt`

`r2=pkt_end`

`r1=pkt`

`r3=pkt`

`r3=pkt(off=36)`

`sizeof(pkt) > 36`

# Верификация

`xdp_foo()` :

`r2 = *(r1 + 4)`

`r1 = *(r1 + 0)`

`r3 = r1`

`r3 += 36`

`if r3 >= r2 jmp +3`

`r0 = *(r1 + 50)`

`r1=cxt`

`r2=pkt_end`

`r1=pkt`

`r3=pkt`

`r3=pkt(off=36)`

`sizeof(pkt) > 36`

# Верификация

“Исполняет” код и следит за

- Типом регистров
- Диапазоном значений (min/max)
- Control Flow

# Верификация

“Исполняет” код и следит за

- Типом регистров
- Диапазоном значений (min/max)
- Control Flow

× Очень хрупкая конструкция

# Верификация

```
SEC("xdp")
```

```
int xdp_scream(struct xdp_md *ctx) {  
    void *pkt = (void *) (long) ctx->data;  
    void *pkt_end = (void *) (long) ctx->data_end;  
    u8 val = 0;  
  
    if (pkt_end > pkt + 0x36) {  
        val = *(u8 *) (pkt + 0x16);  
    }  
  
    return val == 0xAA ? XDP_PASS : XDP_DROP;  
}
```

# Верификация

```
SEC("xdp")
```

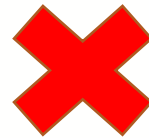
```
int xdp_scream(struct xdp_md *ctx) {  
    void *pkt = (void *) (long) ctx->data;  
    void *pkt_end = (void *) (long) ctx->data_end;  
    u8 val = 0;  
  
    int pkt_size = pkt_end - pkt;  
    if (pkt_size > 0x36) {  
        val = *(u8 *) (pkt + 0x16);  
    }  
  
    return val == 0xAA ? XDP_PASS : XDP_DROP;  
}
```

# Верификация

```
SEC("xdp")
```

```
int xdp_scream(struct xdp_md *ctx) {  
    void *pkt = (void *) (long) ctx->data;  
    void *pkt_end = (void *) (long) ctx->data_end;  
    u8 val = 0;
```

```
    int pkt_size = pkt_end - pkt;  
    if (pkt_size > 0x36) {  
        val = *(u8 *) (pkt + 0x16);  
    }
```



pkt\_size = scalar()  
pkt\_size = scalar(0x37, MAX)  
sizeof(pkt) = ??

```
    return val == 0xAA ? XDP_PASS : XDP_DROP;
```

```
}
```

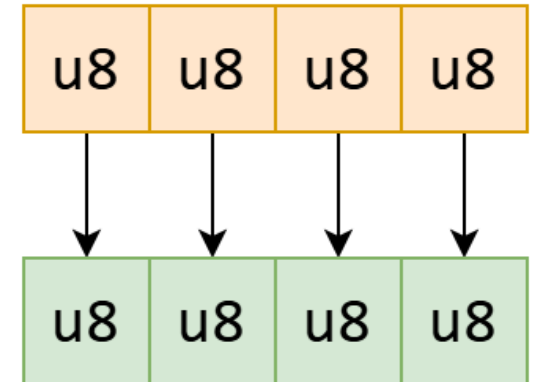
# Верификация

```
addr->p1 = ctx->dst_ip6[0];
```

```
addr->p2 = ctx->dst_ip6[1];
```

```
addr->p3 = ctx->dst_ip6[2];
```

```
addr->p4 = ctx->dst_ip6[3];
```



# Верификация

```
addr->p1 = ctx->dst_ip6[0];
```

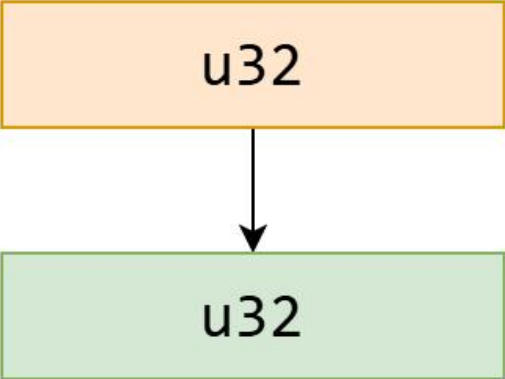
```
addr->p2 = ctx->dst_ip6[1];
```

```
addr->p3 = ctx->dst_ip6[2];
```

```
addr->p4 = ctx->dst_ip6[3];
```



[u8 x 4] = u32



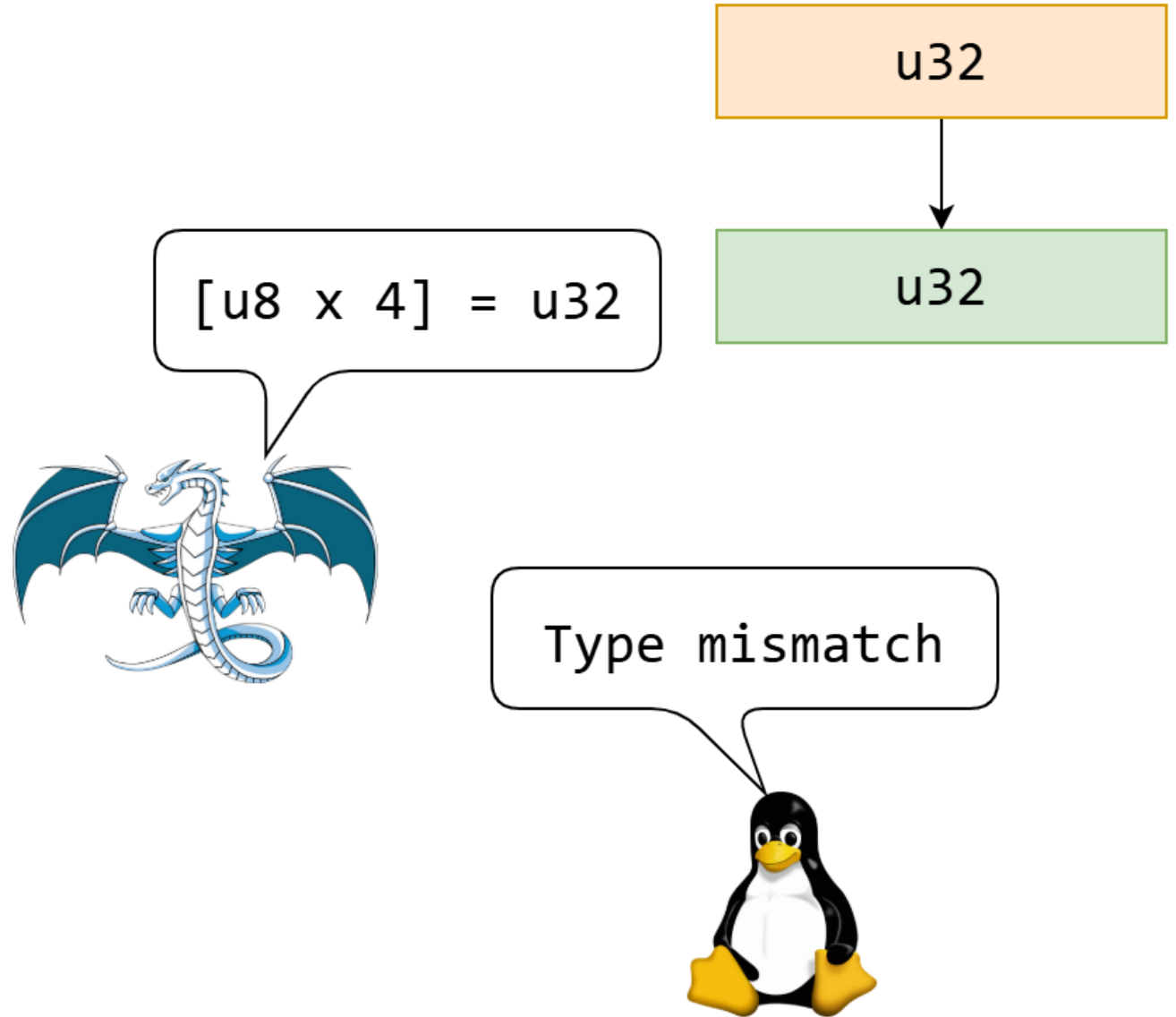
# Верификация

```
addr->p1 = ctx->dst_ip6[0];
```

```
addr->p2 = ctx->dst_ip6[1];
```

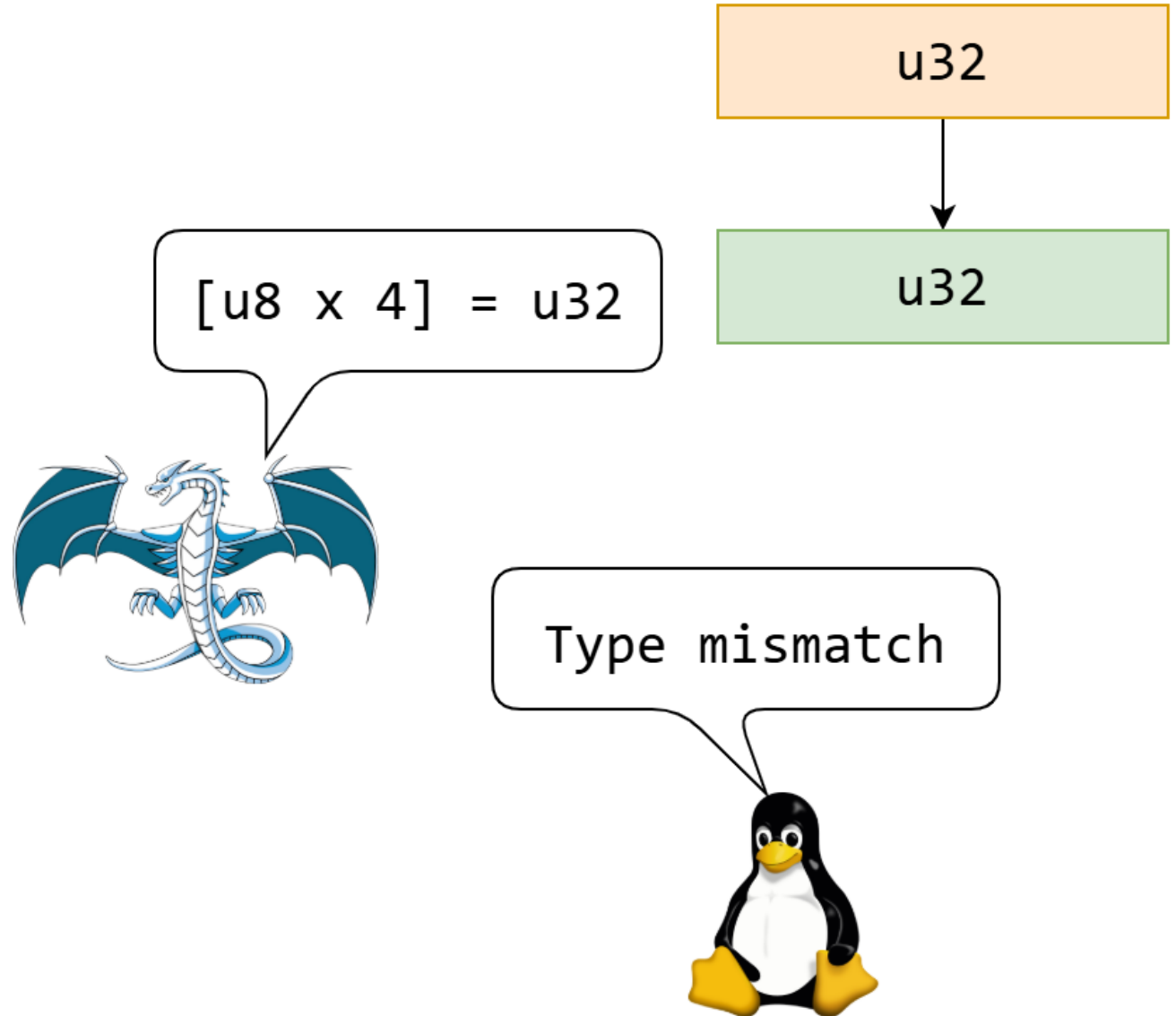
```
addr->p3 = ctx->dst_ip6[2];
```

```
addr->p4 = ctx->dst_ip6[3];
```



# Верификация

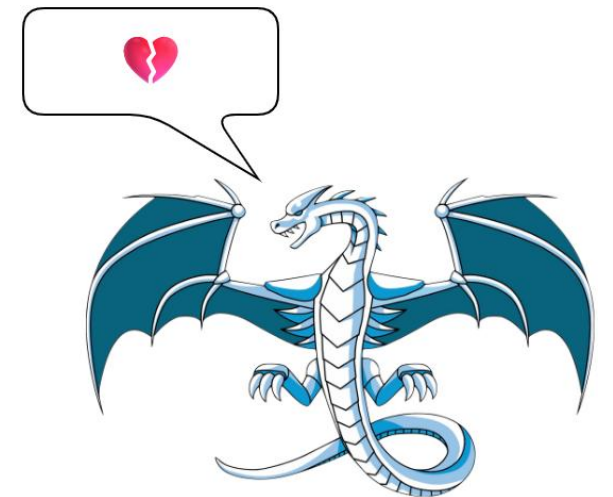
```
addr->p1 = ctx->dst_ip6[0];  
asm volatile("" ::: "memory");  
addr->p2 = ctx->dst_ip6[1];  
asm volatile("" ::: "memory");  
addr->p3 = ctx->dst_ip6[2];  
asm volatile("" ::: "memory");  
addr->p4 = ctx->dst_ip6[3];  
asm volatile("" ::: "memory");
```



# Верификация

В реальности борьба с верификатором приводит к

- Компиляторным барьерам  
Особенно внутри циклов
- Копипаст кода (через inline)
- volatile read
- volatile asm



# Загрузка eBPF



ELF



libbpf



bpf()

- ~~Верификация~~  
Вынуждает писать неоптимально
- Удаляет мертвый код
- Инлайн ядерных функций
- Транслирует в нативные инструкции

# Dead Code Elimination: Why

> Откуда вообще берется мертвый код?

# Dead Code Elimination: Why

> Откуда вообще берется мертвый код?

- `const volatile`

```
if (CONFIG(encryption_strict_ingress) && !ctx_is_decrypt(ctx)) {  
    ret = DROP_UNENCRYPTED_TRAFFIC;  
    goto out;  
}
```

# Dead Code Elimination: Why

> Откуда вообще берется мертвый код?

- `const volatile`

```
if (CONFIG(encryption_strict_ingress) && !ctx_is_decrypt(ctx)) {  
    ret = DROP_UNENCRYPTED_TRAFFIC;  
    goto out;  
}
```

```
#define CONFIG(name)  
(*({  
    void *out;  
    asm volatile("%0 = " __stringify(__config_##name) " 11"  
                : "=r"(out));  
    (typeof(__config_##name) *)out;  
}))
```

# Dead Code Elimination: Why

> Откуда вообще берется мертвый код?

- `const volatile`
- “Рантайм” информация

```
/*
 * Fallback to the old API if the kernel doesn't support
 * scx_bpf_select_cpu_and().
 *
 * This is required to support kernels <= 6.16.
 */
if (!bpf_ksym_exists(scx_bpf_select_cpu_and)) {
    bool is_idle = false;
```

# Dead Code Elimination: How

CONFIG\_TRUE ? 100 : 200

```
r1 = &CONFIG_TRUE
```

r1=const ptr

```
r1 = *(r1 + 0)
```

r1=scalar()

```
if r1 != 0 jmp +2
```

```
r2 = 200
```

```
jmp +1
```

```
r2 = 100
```

# Dead Code Elimination: How

CONFIG\_TRUE ? 100 : 200

```
r1 = &CONFIG_TRUE
```

```
r1 = *(r1 + 0)
```

```
if r1 != 0 jmp +2
```

```
r2 = 200
```

```
jmp +1
```

```
r2 = 100
```

r1=const ptr

r1=scalar(**true**)

# Dead Code Elimination: How

CONFIG\_TRUE ? 100 : 200

```
r1 = &CONFIG_TRUE
```

```
r1 = *(r1 + 0)
```

```
jmp +2
```

```
r2 = 200
```

```
jmp +1
```

```
r2 = 100
```

r1=const ptr

r1=scalar(**true**)

# Dead Code Elimination: How

CONFIG\_TRUE ? 100 : 200

```
r1 = &CONFIG_TRUE
```

```
r1 = *(r1 + 0)
```

```
r2 = 100
```

r1=const ptr

r1=scalar(**true**)

# Dead Code Elimination: Result

- Можем выкинуть до 80% кода  
слишком много опций  
сложная реализация совместимости
- Упрощаем Control Flow граф

# Dead Code Elimination: Result

- Можем выкинуть до 80% кода
- Упрощаем Control Flow граф

Изначально:

- × Неэффективная аллокация регистров
- × Неправильный расчет инлайна
- × Пессимистичный Control Flow

# Загрузка eBPF



ELF



libbpf



bpf()

- ~~Верификация~~  
Вынуждает писать неоптимально
- ~~Удаляет мертвый код~~  
Было скомпилировано пессимистично
- Инлайн ядерных функций
- Транслирует в нативные инструкции

# Inline: Что подставляется

- Интринсики
  - Адрес хештаблицы
- “Простые” функции  
map\_lookup(), get\_processor\_id(), bpf\_loop(), ...
- Трансформации для удобства JIT  
Деление как тернарный оператор

# Inline: Как реализовано

6 базовых блоков

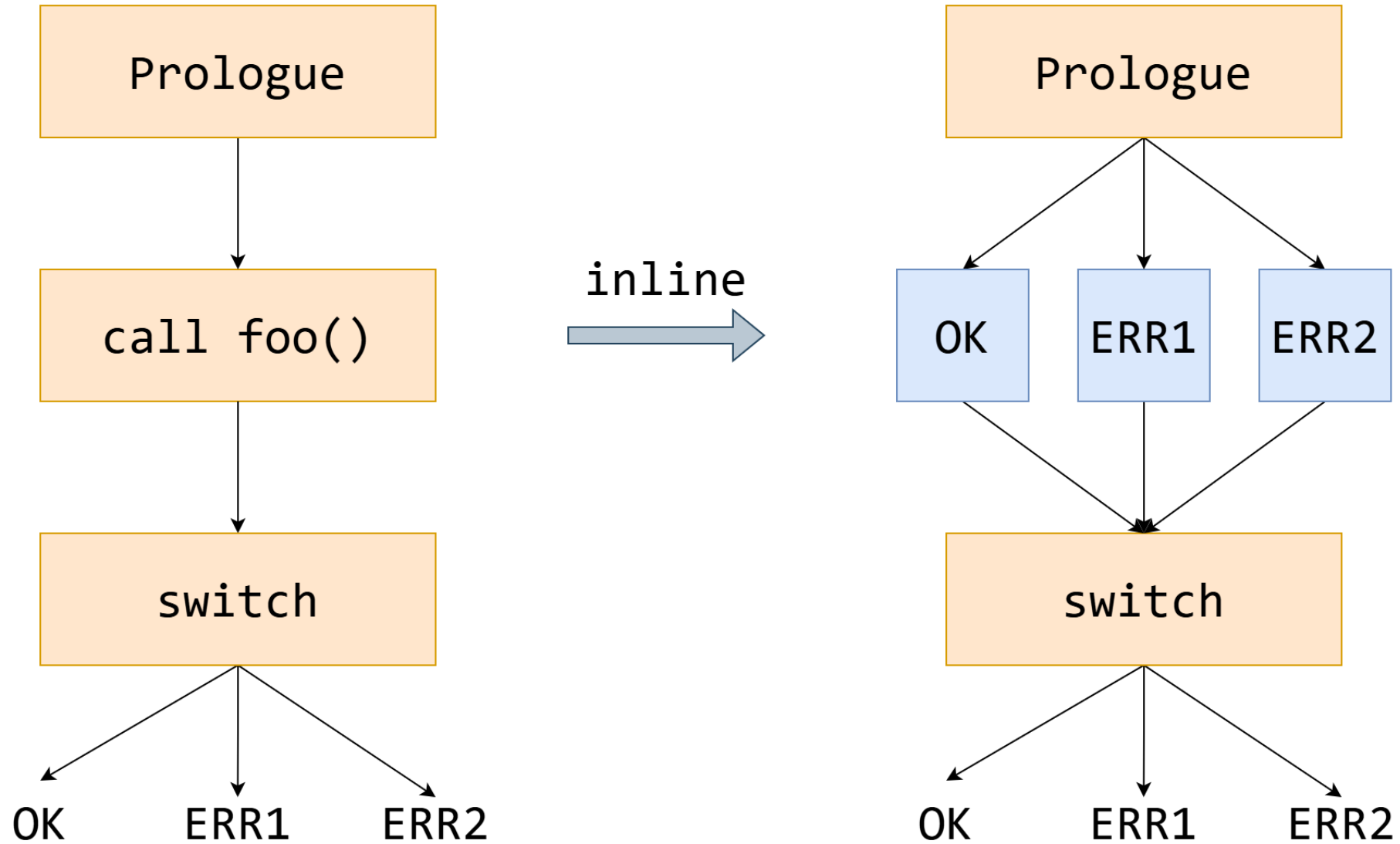
3 различных выхода

```
insn_buf[0] = BPF_JMP_IMM(BPF_JNE, BPF_REG_3, 0, 7);

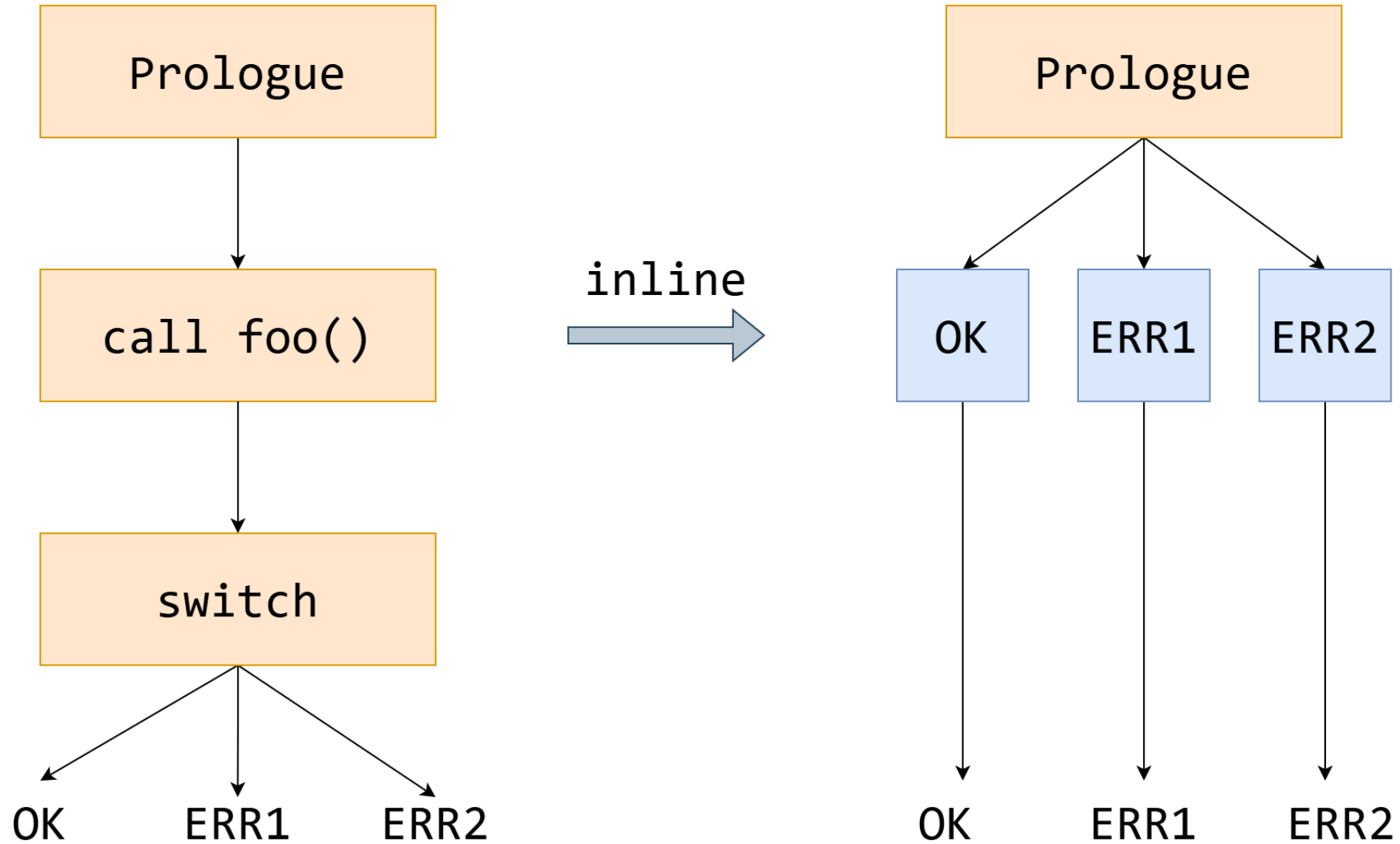
/* Transform size (bytes) into number of entries (cnt = size / 24). */
insn_buf[1] = BPF_MOV32_IMM(BPF_REG_0, 0xaaaaaaaaab);
insn_buf[2] = BPF_ALU64_REG(BPF_MUL, BPF_REG_2, BPF_REG_0);
insn_buf[3] = BPF_ALU64_IMM(BPF_RSH, BPF_REG_2, 36);

/* call perf_snapshot_branch_stack implementation */
insn_buf[4] = BPF_EMIT_CALL(static_call_query(perf_snapshot_branch_stack));
/* if (entry_cnt == 0) return -ENOENT */
insn_buf[5] = BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 4);
/* return entry_cnt * sizeof(struct perf_branch_entry) */
insn_buf[6] = BPF_ALU32_IMM(BPF_MUL, BPF_REG_0, br_entry_size);
insn_buf[7] = BPF_JMP_A(3);
/* return -EINVAL; */
insn_buf[8] = BPF_MOV64_IMM(BPF_REG_0, -EINVAL);
insn_buf[9] = BPF_JMP_A(1);
/* return -ENOENT; */
insn_buf[10] = BPF_MOV64_IMM(BPF_REG_0, -ENOENT);
```

# Inline: Проблемы



# Inline: Проблемы



# Inline

- ✓ Экономим на call / ret
- × Не переаллоцируем регистры
- × Не упрощаем Control Flow
- × Исходный код был собран пессимистично

# Загрузка eBPF



ELF



libbpf



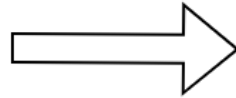
bpf()

- ~~Верификация~~  
Вынуждает писать неоптимально
- ~~Удаляет мертвый код~~  
Было скомпилировано пессимистично
- ~~Инлайн ядерных функций~~  
Экономия на спичках
- Транслирует в нативные инструкции

# Трансляция

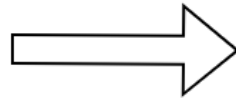
- > Регистры отображаются 1:1
- > Инструкции транслируются по одной

```
r6 += r3
```



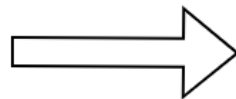
```
add %rdx, %rbx
```

```
r2 = *(r1 + 8)
```



```
mov 0x8(rdi), %rdx
```

```
if r3 >= r2  
goto pc+5
```

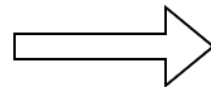


```
cmp %rdi, %rsi  
jae <+5>
```

# Трансляция

! Инструкции имеют разную семантику

```
r9 /= 100
```

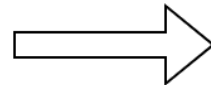


```
div 100, %r15
```

# Трансляция

- ! Инструкции имеют разную семантику  
В x64 зафиксирован source = rax:rdx

r9 /= 100

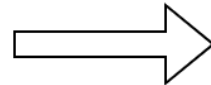


```
xor %rdx, %rdx
mov %r15, %rax
    div 100
mov %rax, %r15
```

# Трансляция

- ! Инструкции имеют разную семантику
- В x64 делить можно только на регистр

r9 /= 100

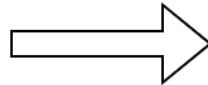


```
mov 100, %r11
xor %rdx, %rdx
mov %r15, %rax
    div %r11
mov %rax, %r15
```

# Трансляция

- ! Инструкции имеют разную семантику  
Нельзя инвалидировать eBPF регистры

`r9 != 100`



```
push %rax
push %rdx

mov 100, %r11
xor %rdx, %rdx

mov %r15, %rax

div %r11

mov %rax, %r15

pop %rdx

pop %rax
```

# Трансляция

- ✓ Зачастую эффективна
- ✓ Прямолинейная для большинства инструкций
- ✗ Проблемы с разной семантикой
- ✗ Ограничены выразительностью байткода

# Конвейер загрузки



ELF



libbpf



bpf()

- ~~Верификация~~  
Вынуждает писать неоптимально
- ~~Удаляет мертвый код~~  
Было скомпилировано пессимистично
- ~~Подставляет ядерные функции~~  
Экономия на спичках
- ~~Транслирует в нативные инструкции~~  
По одной инструкции неэффективно

# Хотим лучше

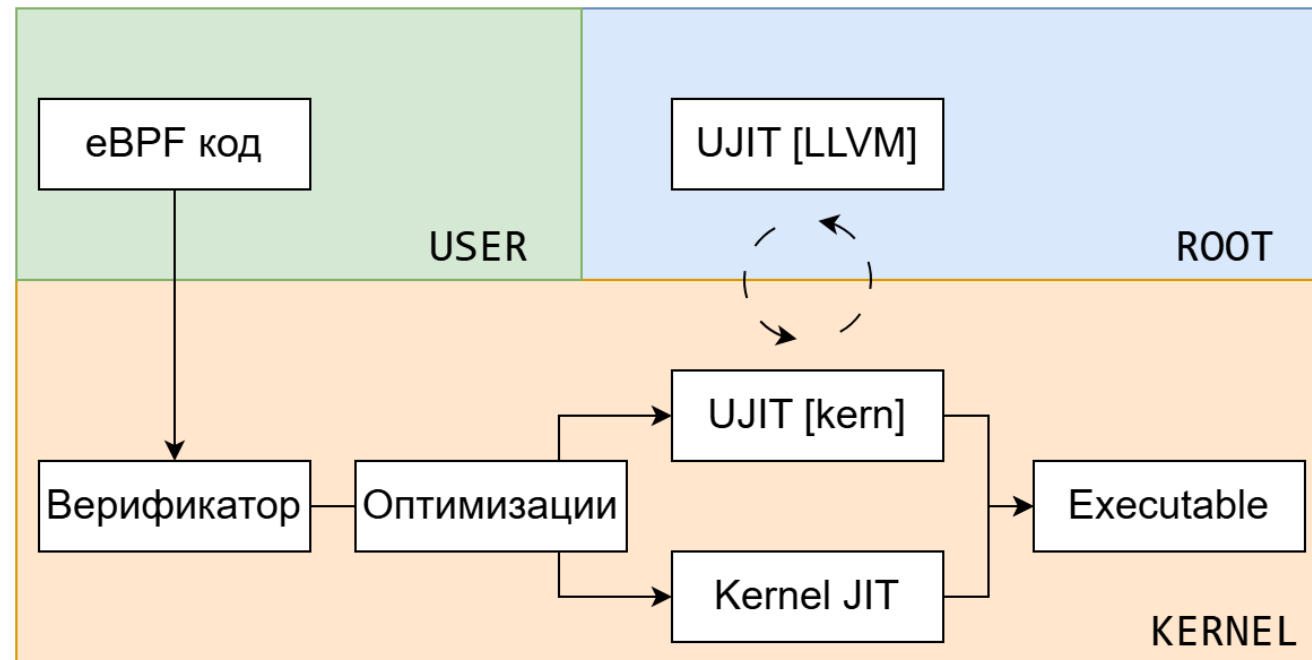
- > Не меняя интерфейс  
Иначе модули + Rust-for-Linux
- > Сохраняя гарантии корректности  
Не трогаем верификатор
- > Максимально простое решение  
Не изобретаем компилятор в ядре

# Хотим лучше

- > Не меняя интерфейс
  - > Сохраняя гарантии корректности
  - > Максимально простое решение
- ⇒ Существующий компилятор после верификации

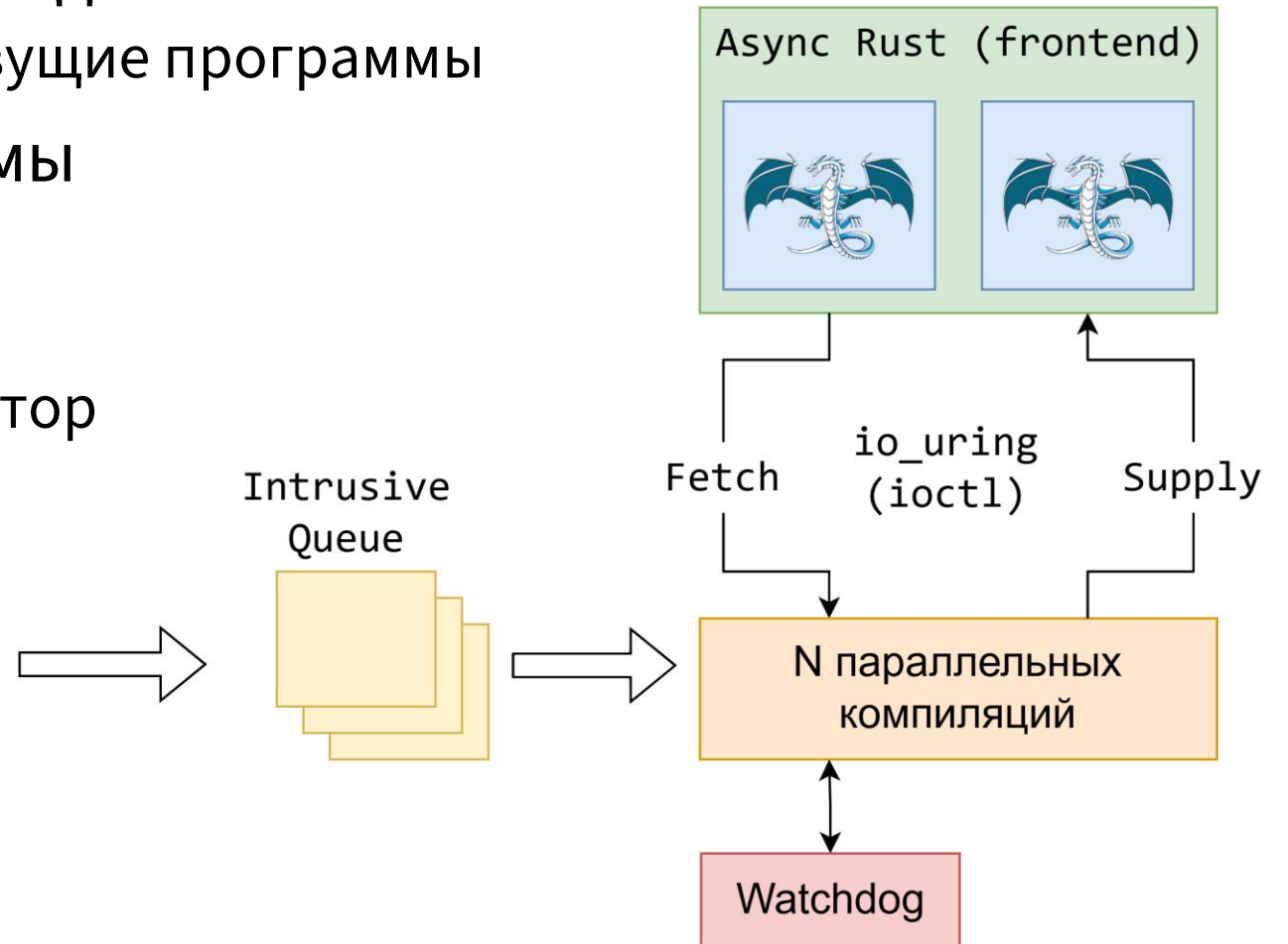
# UJIT

- > Демон в userspace + интеграция в ядре
- > Мы доверяем компилятору (root)



# UJIT: Детали

- Время компиляции от 30ms до 500ms  
Ориентировано на долгоживущие программы
- Программа и подпрограммы компилируются вместе  
Позволяет переделать inline  
Знаем больше чем верификатор
- `march=native*`  
`jump-таблицы`, `mov`  
`ropcnt`



# UJIT: Трансляция

- Разбиваем на базовые блоки
- Регистры и стек – локальные переменные  
ABI не использует стек
- Транслируем по одной инструкции  
O2/O3 оптимизирует

```
u64 ebpf_function(u64 arg1) {
    u64 r0, r1 = arg1, r2, /*... ,*/ r10;
    u8 stack[STACK_SIZE];

    /* Generate code for each instruction */
    {
        r1 = 0xFFFFDEAD;
    }
    {
        u64 (*kfuncptr)(u64) = 0xDEADC0DE;
        r0 = kfuncptr(r1);
    }

    return r0;
}
```

# UJIT: Трансляция

- Разбиваем на базовые блоки
- Регистры и стек – локальные переменные
  - ABI не использует стек
- Транслируем по одной инструкции
  - O2/O3 оптимизирует
- ! Надо знать сигнатуру функций
  - Достаем в ядре из верификатора и инлайнера
- ! Сложно обнаруживать баги
  - У eBPF хорошее покрытие тестами

```
u64 ebpf_function(u64 arg1) {
    u64 r0, r1 = arg1, r2, /*... ,*/ r10;
    u8 stack[STACK_SIZE];

    /* Generate code for each instruction */
    {
        r1 = 0xFFFFDEAD;
    }
    {
        u64 (*kfuncptr)(u64) = 0xDEADC0DE;
        r0 = kfuncptr(r1);
    }

    return r0;
}
```

# UJIT: Нюансы

- Бинарная совместимость с in-kernel JIT
  - Невыгодно: счетчик tail call вызовов передается через стек
  - Опасно: отслеживать версии ядра с точностью до коммита

# UJIT: Нюансы

- Отсутствует бинарная совместимость с in-kernel JIT  
Нельзя смешивать программы от разных JIT

# UJIT: Нюансы

- Отсутствует бинарная совместимость с in-kernel JIT  
Нельзя смешивать программы от разных JIT
- Исключения (unwind)
  - В ядре регистры сохраняются при входе в каждую функцию
  - У UJIT больше набор регистров => неприменимо

# UJIT: Нюансы

- Отсутствует бинарная совместимость с in-kernel JIT  
Нельзя смешивать программы от разных JIT
- Не поддерживаем исключения (пока что)  
Нельзя вызывать `bpf_throw`

# UJIT: Нюансы

- Отсутствует бинарная совместимость с in-kernel JIT  
Нельзя смешивать программы от разных JIT
- Не поддерживаем исключения (пока что)  
Нельзя вызывать `bpf_throw`
- Байткод не содержит всей информации
  - Например, теряем атрибуты `likely` или `null checks`

# UJIT: Нюансы

- Отсутствует бинарная совместимость с in-kernel JIT  
Нельзя смешивать программы от разных JIT
- Не поддерживаем исключения (пока что)  
Нельзя вызывать `bpf_throw`
- Байткод не содержит всей информации  
Редко используются + static branch “predictor”

# UJIT: Нюансы

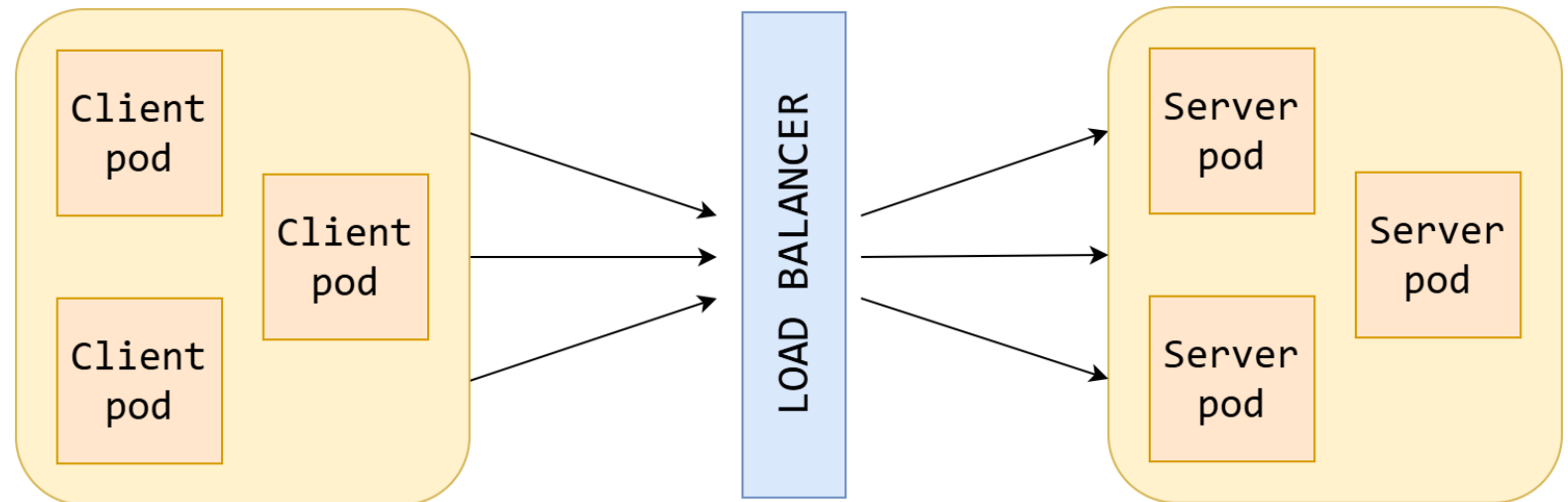
- Отсутствует бинарная совместимость с in-kernel JIT
  - Нельзя смешивать программы от разных JIT
- Не поддерживаем исключения (пока что)
  - Нельзя вызывать `bpf_throw`
- Байткод не содержит всей информации
  - Редко используются + static branch “predictor”
- Конфликт с будущей моделью памяти eBPF
  - LLVM умнее процессора в aliasing
  - Распространяется только на странный eBPF ассемблер

# UJIT: Нюансы

- Отсутствует бинарная совместимость с in-kernel JIT  
Нельзя смешивать программы от разных JIT
- Не поддерживаем исключения (пока что)  
Нельзя вызывать `bpf_throw`
- Байткод не содержит всей информации  
Редко используются + `static branch` “predictor”
- Конфликт с будущей моделью памяти eBPF  
Под ответственность пользователя (пока что)

# UJT: Производительность

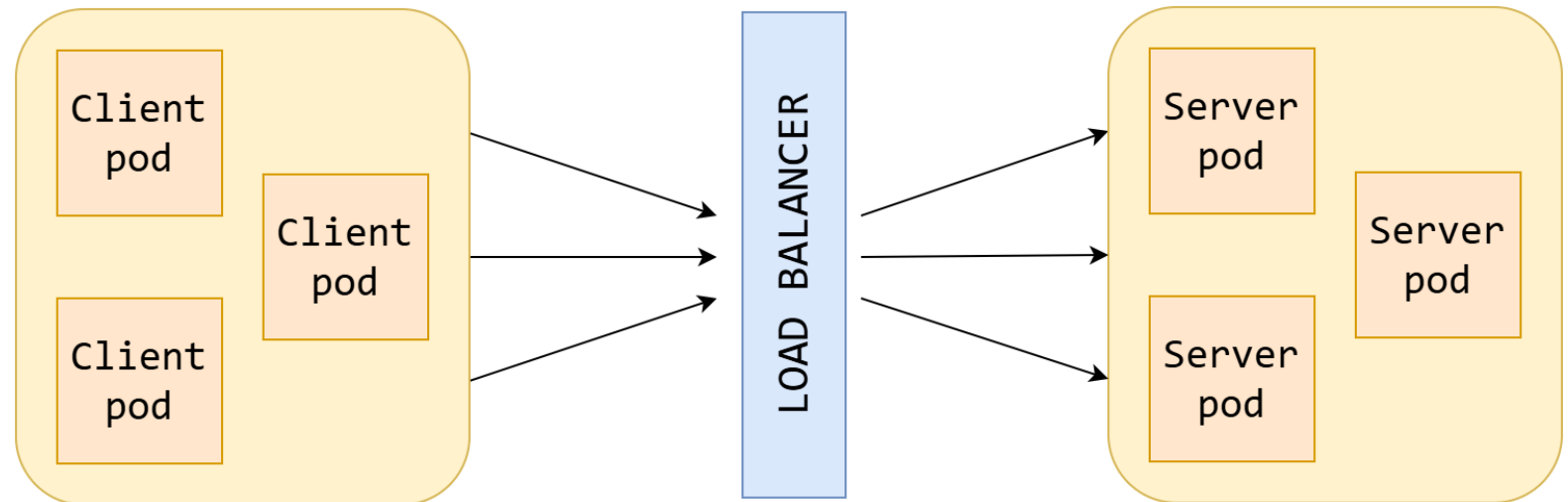
- 40% меньше инструкций  
Выигрыш по размеру меньше
- ~1.5x ускорение eBPF
- XX% уменьшение latency в Cilium





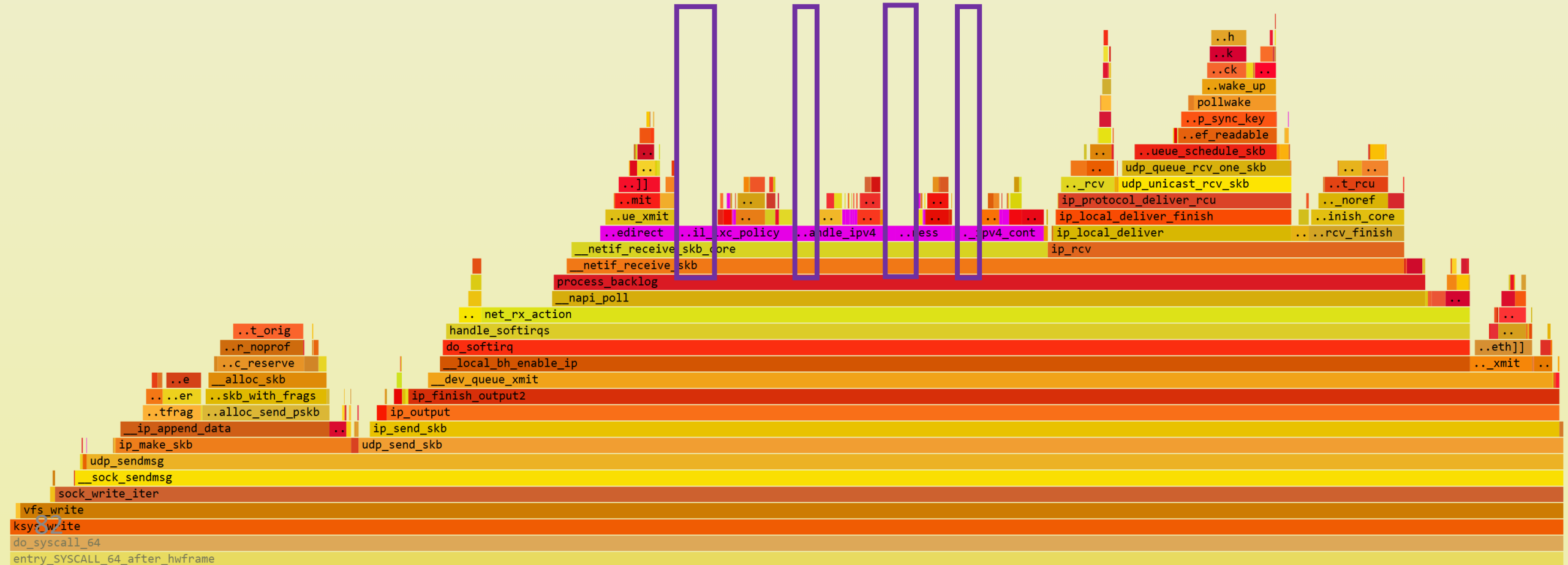
# UJT: Производительность

- 40% меньше инструкций  
Выигрыш по размеру меньше
- ~1.5x ускорение eBPF
- 5% уменьшение latency в Cilium



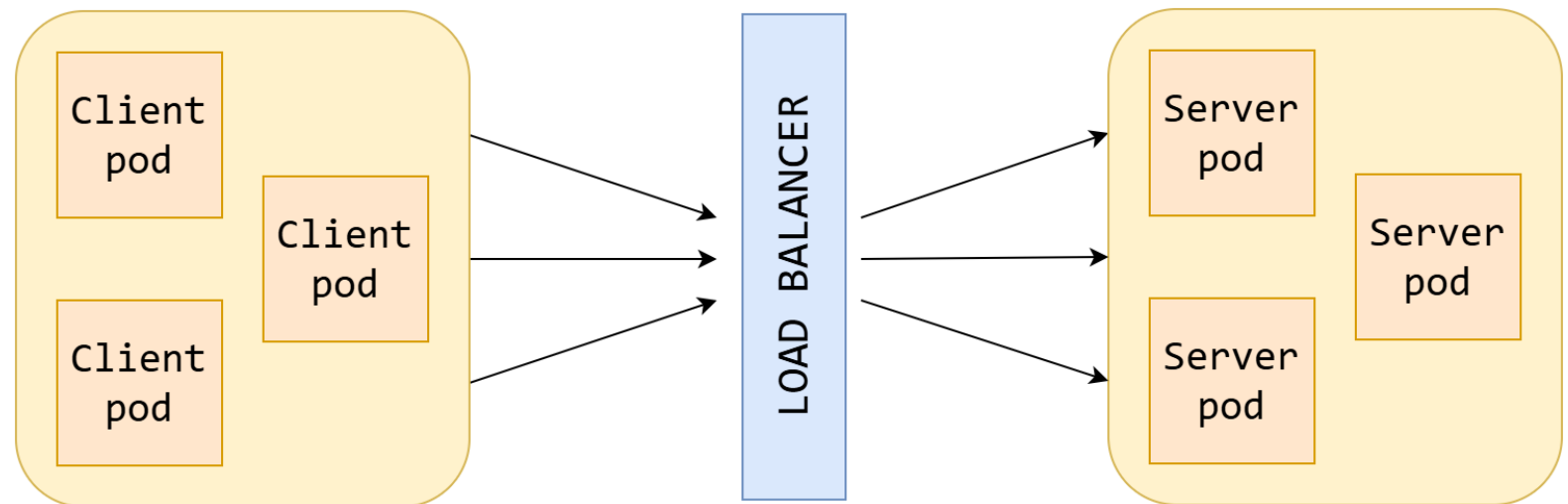
# UJT: Производительность

28% of ksys\_write



# UJT: Производительность

- 40% меньше инструкций  
Выигрыш по размеру меньше
- ~1.5x ускорение eBPF
- 5% уменьшение latency в Cilium  
**Сам байткод занимает мало времени**



# UJT: Будущее

- Программы раздроблены из-за ограничений сложности
  - Оптимизация с помощью “LTO”

# UJT: Будущее

- Программы раздроблены из-за ограничений сложности
  - Оптимизация с помощью “LTO”
- Проверки корректности в ядерных функциях
  - Сделать inline проверок

# UJT: Будущее

- Программы раздроблены из-за ограничений сложности
  - Оптимизация с помощью “LTO”
- Проверки корректности в ядерных функциях
  - Сделать inline проверок
- LLVM не знает типов
  - Экспорт типов из верификатора

# UJT: Будущее

- Интеграция в OpenEuler Linux
  - В экспериментальную ветку
- Обратная связь
  - Нужны первые экспериментаторы!
- No upstream...

# No upstream

- Конфликт с моделью памяти
- Хотят писать компилятор в ядре  
Аллокатор регистров через 2 года
- “Доверяете userspace – пишите модули на Rust”

# No upstream

- Конфликт с моделью памяти
  - Хотят писать компилятор в ядре
    - Аллокатор регистров через 2 года
  - “Доверяете userspace – пишите модули на Rust”
- RFC будет отправлена в любом случае

# Заключение

- Текущий дизайн eVPF приводит к низкой производительности
- Но часто этого достаточно
- Разработчик не может на это повлиять
  - Либо переписать на модули
- В ядре нет оптимизаций
- Легкими усилиями можно ускорить
  - Но развитие идеи потребует более инвазивных изменений

