SUSE

31 JULY 2022

Peeking into the BPF verifier

Shung-Hsi Yu
COSCUP 2022

About me

Shung-Hsi Yu @shunghsiyu

Kernel Engineer at SUSE

maintaining BPF in SLES & openSUSE kernel

Copyright © SUSE 2022

Disclaimers

Copyright © SUSE 2021

Goals of this Talk

- Focus on the verifier for **eBPF** runtime in the Linux Kernel
  - Mostly source code can be found in **kernel/bpf/verifier.c**
- Helps you understand why verifier rejected you program (though that should be quite rare these days)

Copyright © SUSE 2022

Non-goals

- Teach you all about the BPF verifier in 30 minutes, rather
  - Talk about a very, very, **small portion** of how the BPF verifier works
  - Provide **foundation** upon which audience can further use to understand the BPF verifier
- Provide the most up-to-date insight
  - By the time I'm talking about it, it's (probably) already outdated
  - Based on **v5.18** unless otherwise mentioned

Copyright © SUSE 2022

Background

Copyright © SUSE 2021

What is BPF

- In-kernel **Virtual Machine**
- Runs **user-provided** code
  - Inside the Linux Kernel privilege level
  - In a **safe** manner
- Variety of use-cases
  - tracing, profiling, networking, security, etc.

Copyright © SUSE 2022

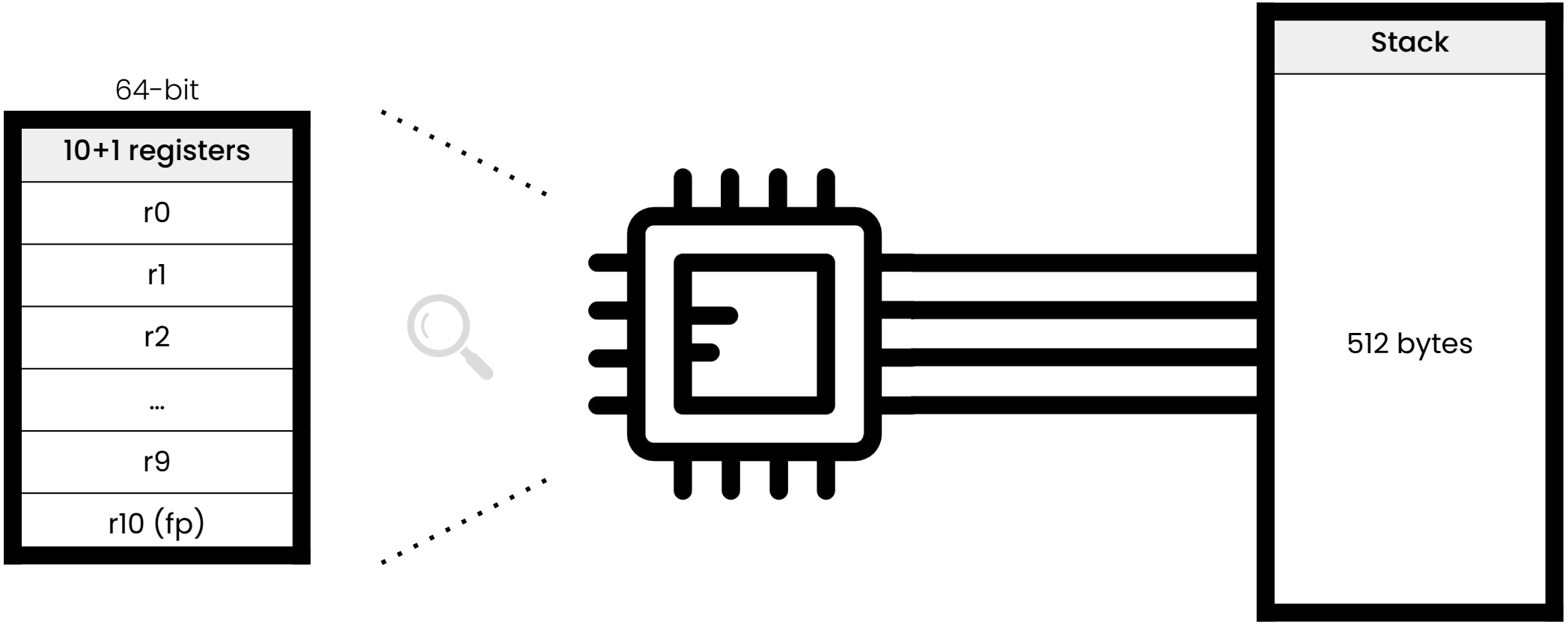
What is BPF

- In-kernel **Virtual Machine**
- Runs **user-provided** code
  - Inside the Linux Kernel privilege level
  - In a **safe** manner
- Variety of use-cases
  - tracing, profiling, networking, security, etc.

Copyright © SUSE 2022

BPF Virtual Machine

Virtual CPU & Memory

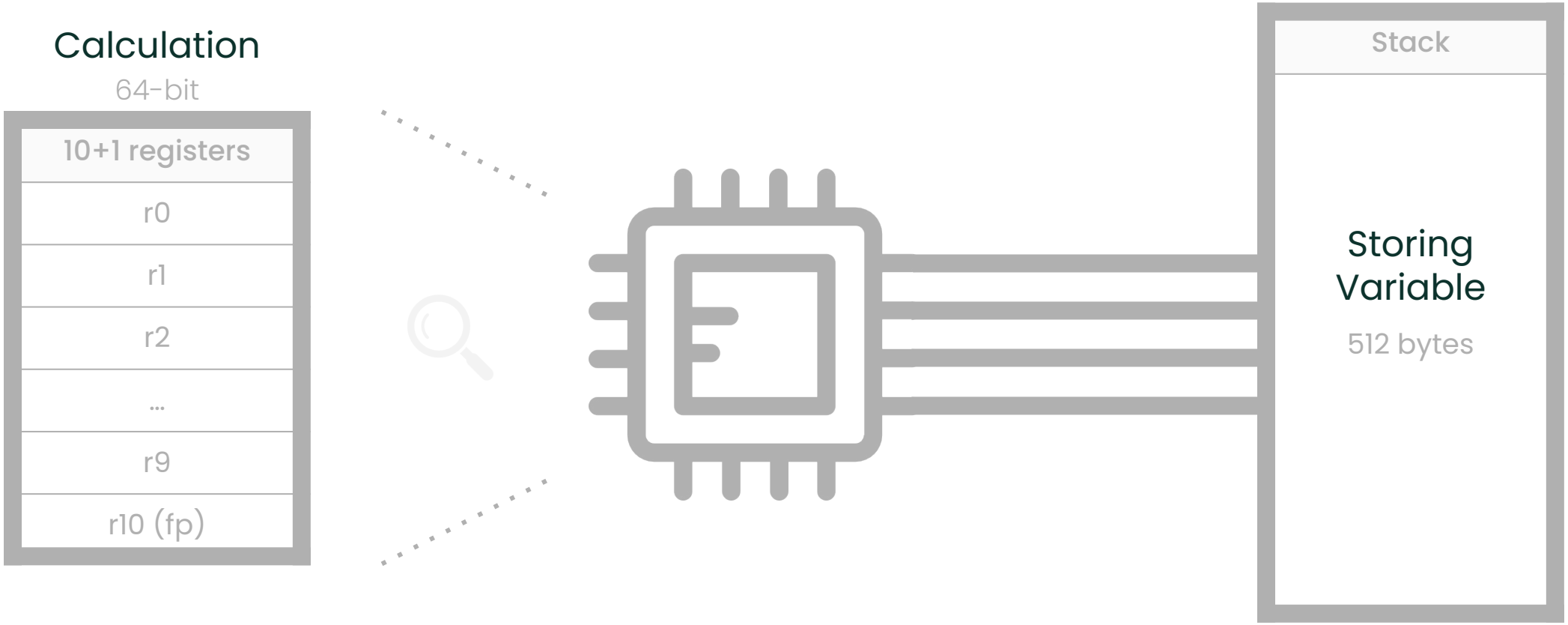


Copyright © SUSE 2022

Cpu icons created by Prosymbols - Flaticon. Magnifying glass icons created by Muhammad_Usman - Flaticon.

BPF Virtual Machine

Virtual CPU & Memory

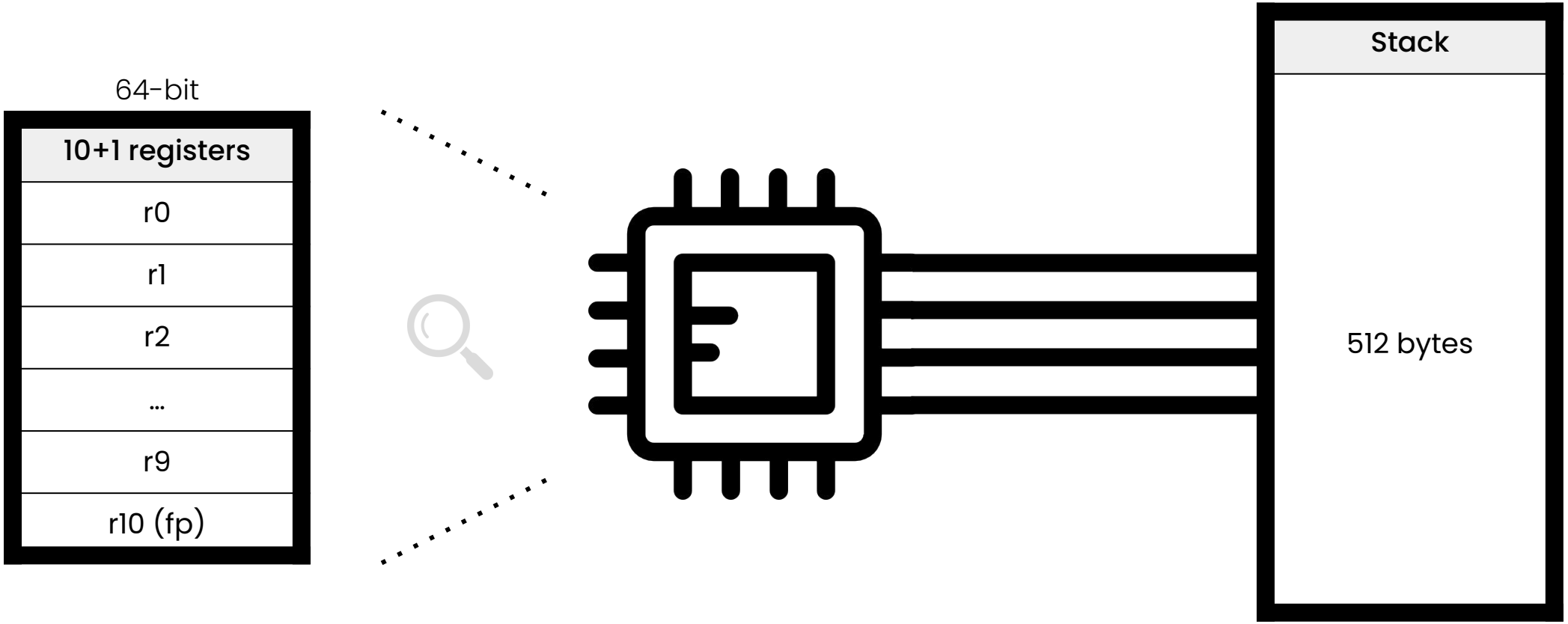


Copyright © SUSE 2022

Cpu icons created by Prosymbols - Flaticon. Magnifying glass icons created by Muhammad_Usman - Flaticon.

BPF Virtual Machine

Virtual CPU & Memory

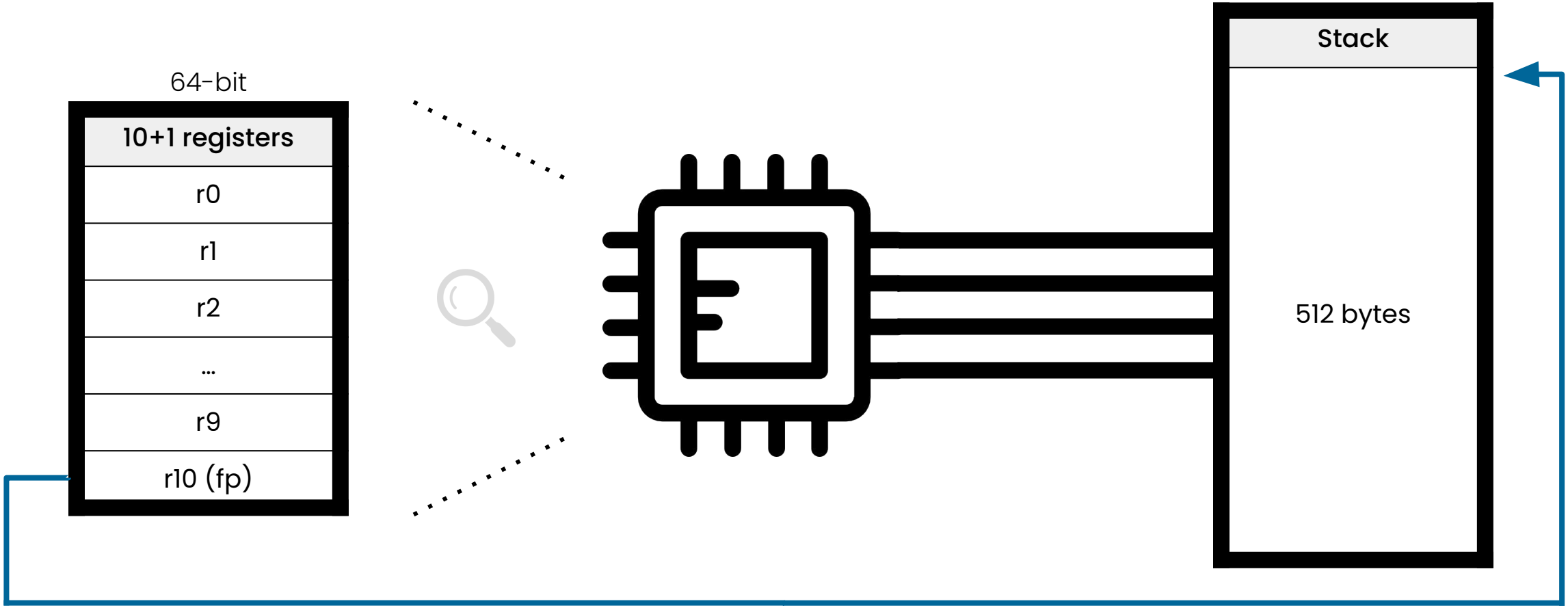


Copyright © SUSE 2022

Cpu icons created by Prosymbols - Flaticon. Magnifying glass icons created by Muhammad_Usman - Flaticon.

BPF Virtual Machine

Virtual CPU & Memory



Copyright © SUSE 2022

Cpu icons created by Prosymbols - Flaticon. Magnifying glass icons created by Muhammad_Usman - Flaticon.

What is BPF

- In-kernel **Virtual Machine**
- Runs **user-provided** code
  - Inside the Linux Kernel privilege level
  - In a **safe** manner
- Variety of use-cases
  - tracing, profiling, networking, security, etc.

Copyright © SUSE 2022

User-provided Code

User usually right something like this

```
int x, y;

int add(void)
{
    return x+y;
}
```

Copyright © SUSE 2022

User-provided Code

It will then be compiled down to **BPF instructions** (aka BPF bytecode)

```
0000000000000000 <add>:
    0: r1 = 0 ll (&x);          BPF_LD_IMM64_RAW(BPF_REG_2, ...)
    2: r1 = *(u32 *)(r1 + 0);   BPF_LDX_MEM(BPF_W, BPF_REG_1, BPF_REG_1, 0)
    3: r2 = 0 ll (&y);          BPF_LD_IMM64_RAW(BPF_REG_2, ...)
    5: r0 = *(u32 *)(r2 + 0);   BPF_LDX_MEM(BPF_W, BPF_REG_0, BPF_REG_0, 0)
    6: r0 += r1;                BPF_ALU64_REG(BPF_ADD, BPF_REG_0, BPF_REG_1)
    7: exit;                    BPF_EXIT_INSN()
```

Copyright © SUSE 2022

User-provided Code

It will then be compiled down to **BPF instructions** (aka BPF bytecode)

```
0000000000000000 <add>:
    0: r1 = 0 ll (&x);          BPF_LD_IMM64_RAW(BPF_REG_2, ...)
    2: r1 = *(u32 *)(r1 + 0);   BPF_LDX_MEM(BPF_W, BPF_REG_1, BPF_REG_1, 0)
    3: r2 = 0 ll (&y);          BPF_LD_IMM64_RAW(BPF_REG_2, ...)
    5: r0 = *(u32 *)(r2 + 0);   BPF_LDX_MEM(BPF_W, BPF_REG_0, BPF_REG_0, 0)
    6: r0 += r1;                BPF_ALU64_REG(BPF_ADD, BPF_REG_0, BPF_REG_1)
    7: exit;                    BPF_EXIT_INSN()
```

Copyright © SUSE 2022

BPF Instruction

```
/* Conceptual representation of
 * a BPF_ADD instruction
 */
r0 += r1;    BPF_ALU64_REG(BPF_ADD, BPF_REG_0, BPF_REG_1)
```

Copyright © SUSE 2022

BPF Instruction

```
/* Conceptual representation of
 * a BPF_ADD instruction
 */
r0 += r1;   BPF_ALU64_REG(BPF_ADD, BPF_REG_0, BPF_REG_1)
```

Copyright © SUSE 2022

BPF Instruction

```
struct bpf_insn insn = {
        .code    = BPF_ALU64 | BPF_ADD | BPF_X,
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

Copyright © SUSE 2022

BPF Instruction

Building blocks of BPF program

8-bit	4-bit	4-bit	16-bit	32-bit
opcode	dst reg	src reg	signed offset	signed immediate

Copyright © SUSE 2022

BPF Instruction

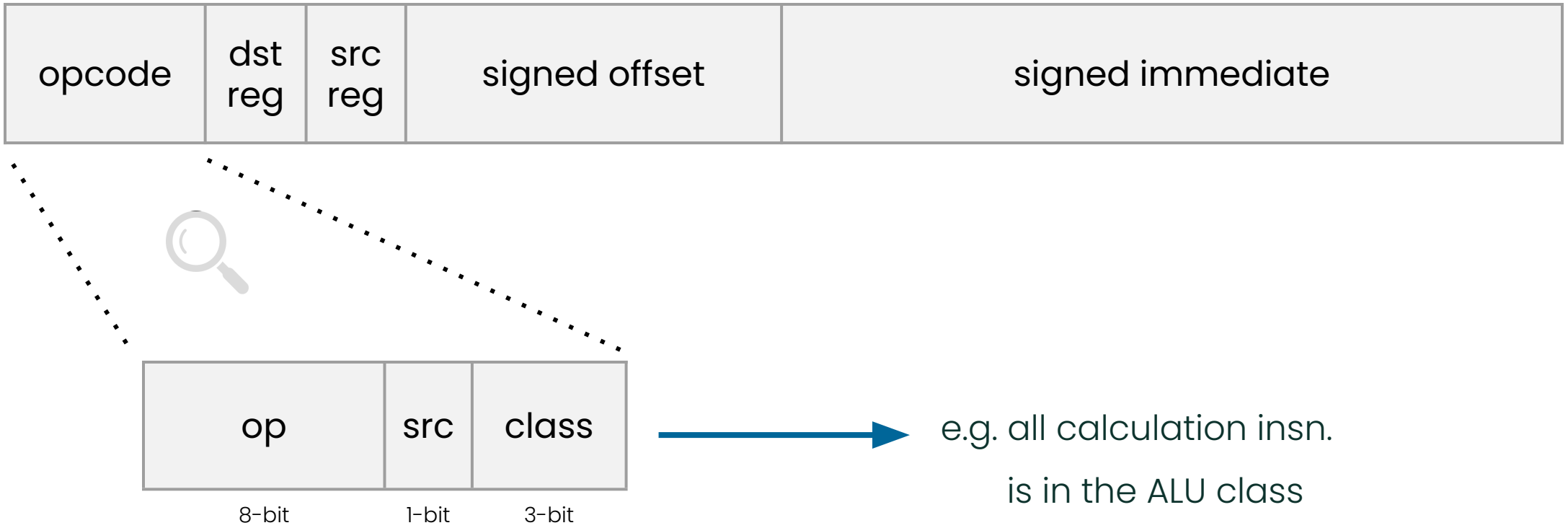
Building blocks of BPF program

Copyright © SUSE 2022

* opcode representation for ALU and JMP class

BPF Instruction

Building blocks of BPF program



e.g. all calculation insn. is in the ALU class

Copyright © SUSE 2022

* opcode representation for ALU and JMP class

BPF Instruction

op	src	class

```
struct bpf_insn insn = {
        .code    = BPF_ALU64 | BPF_ADD | BPF_X,
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

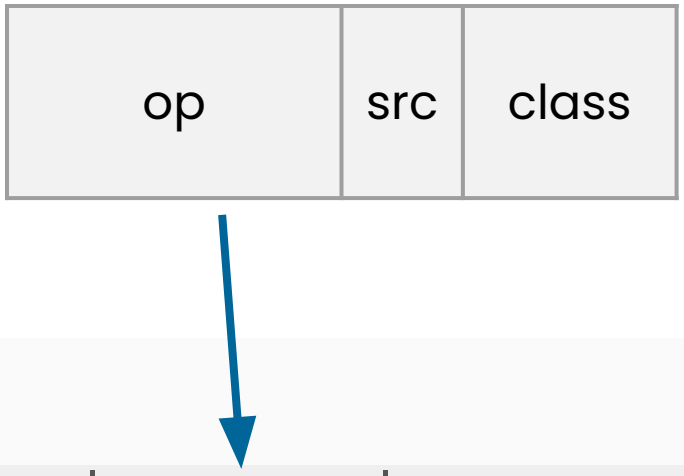
Copyright © SUSE 2022

BPF Instruction

```
struct bpf_insn insn = {
        .code    = BPF_ALU64 | BPF_ADD | BPF_X,
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

Copyright © SUSE 2022

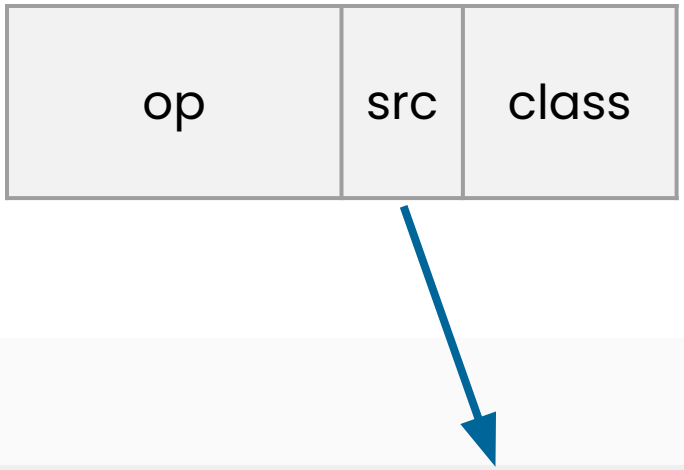
BPF Instruction




```
struct bpf_insn insn = {
        .code    = BPF_ALU64 | BPF_ADD | BPF_X,
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

Copyright © SUSE 2022

BPF Instruction




```
struct bpf_insn insn = {
        .code    = BPF_ALU64 | BPF_ADD | BPF_X,
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

Copyright © SUSE 2022

BPF Instruction

```
/* Defined in include/uapi/linux/bpf.h */
struct bpf_insn {
        __u8    code;           /* opcode */
        __u8    dst_reg:4;      /* dest register */
        __u8    src_reg:4;      /* source register */
        __s16   off;            /* signed offset */
        __s32   imm;            /* signed immediate constant */
};
```

Copyright © SUSE 2022

Safety

Copyright © SUSE 2021

Safety Guarantee

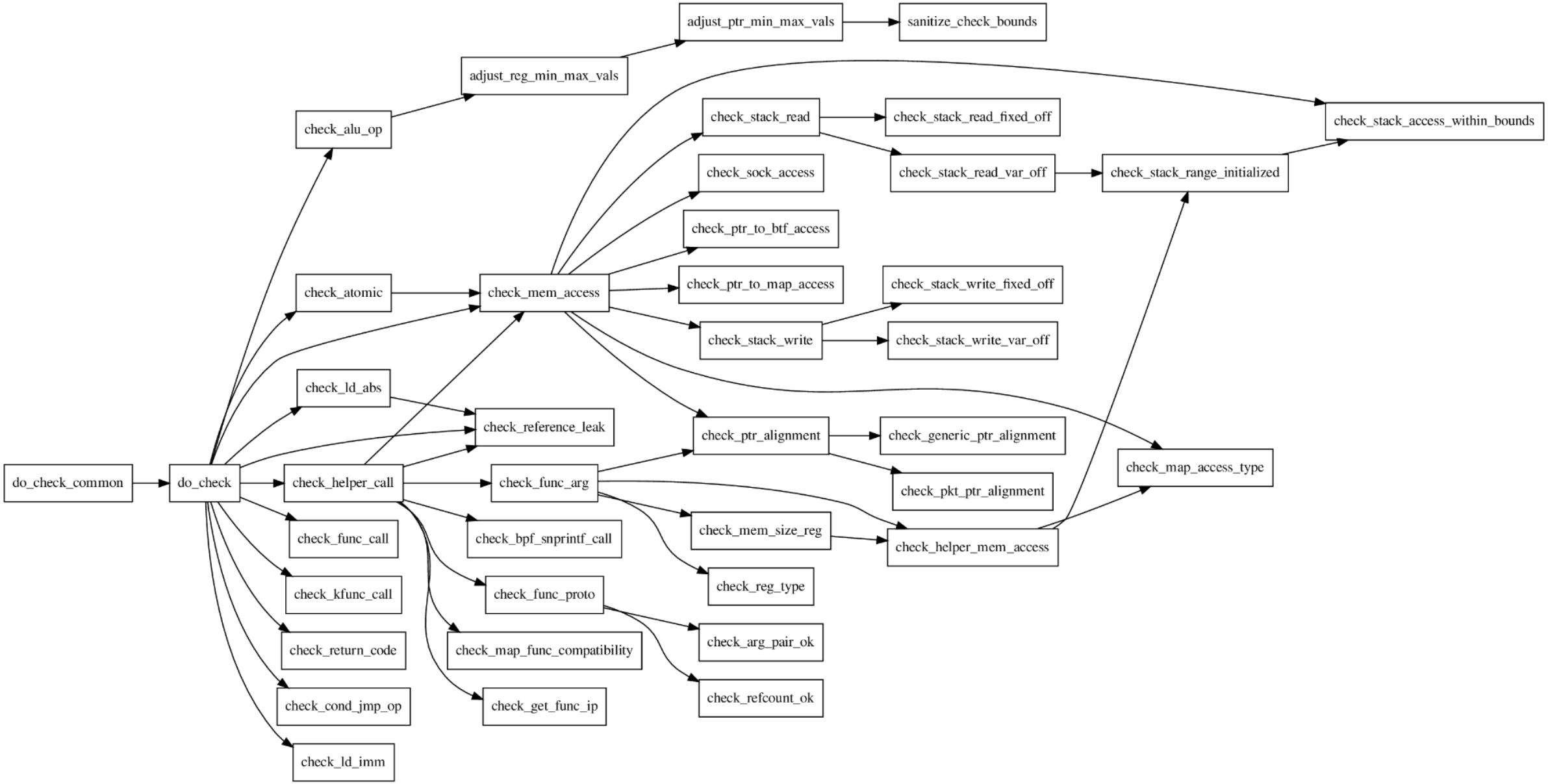
- Invalid memory use
  - Reading
  - Writing
- Address leakage
- Improper termination
- Violation of spec

Copyright © SUSE 2022

Safety Guarantee

- Invalid memory use
  - Reading
  - Writing
- Address leakage
- Improper termination
- Violation of spec

Copyright © SUSE 2022



Copyright © SUSE 2022

Tracking Value & Type

Copyright © SUSE 2021

Tracking Value & Type

For **each registers**, the verifier tracks

```
struct bpf_reg_state {
    /* What kind of value is inside? */
    enum bpf_reg_type type;
    /* What actual value are possible? */
    ...
};
```

Copyright © SUSE 2022

Tracking Type

Copyright © SUSE 2022

Possible Types

The main types

Not-init

Scalar

Pointer

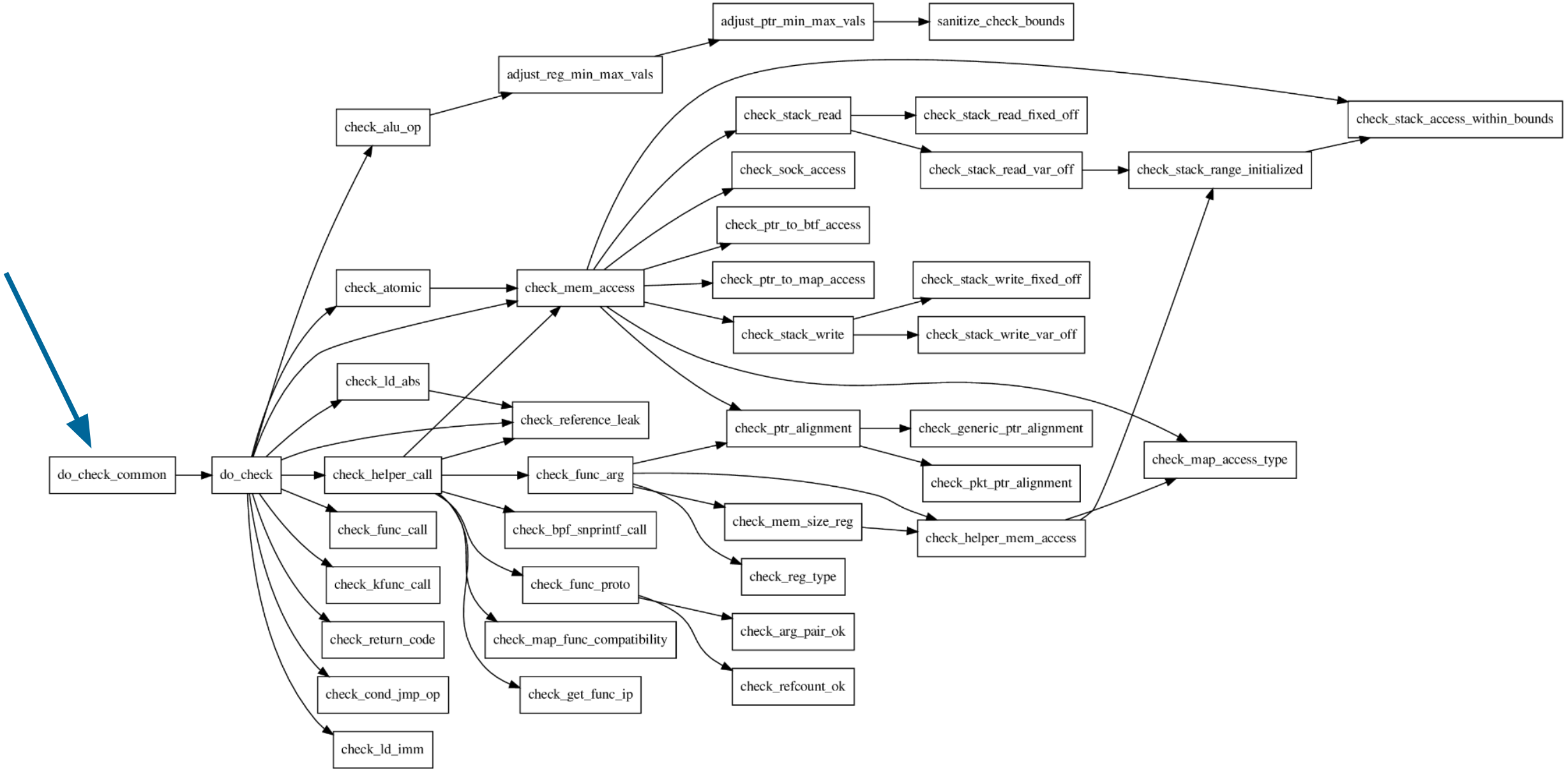
Copyright © SUSE 2022

Possible Types

Types defined

```
enum bpf_reg_type {
    NOT_INIT = 0,        /* nothing was written into register */
    SCALAR_VALUE,        /* reg doesn't contain a valid pointer */
    PTR_TO_CTX,          /* reg points to bpf_context */
    CONST_PTR_TO_MAP,    /* reg points to struct bpf_map */
    PTR_TO_MAP_VALUE,    /* reg points to map element value */
    PTR_TO_MAP_KEY,          /* reg points to a map element key */
    PTR_TO_STACK,        /* reg == frame_pointer + offset */
    PTR_TO_PACKET_META,      /* skb->data - meta_len */
    PTR_TO_PACKET,       /* reg points to skb->data */
    PTR_TO_PACKET_END,   /* skb->data + headlen */
    ...
};
```

Copyright © SUSE 2022



Copyright © SUSE 2022

Initial State

Of type tracking

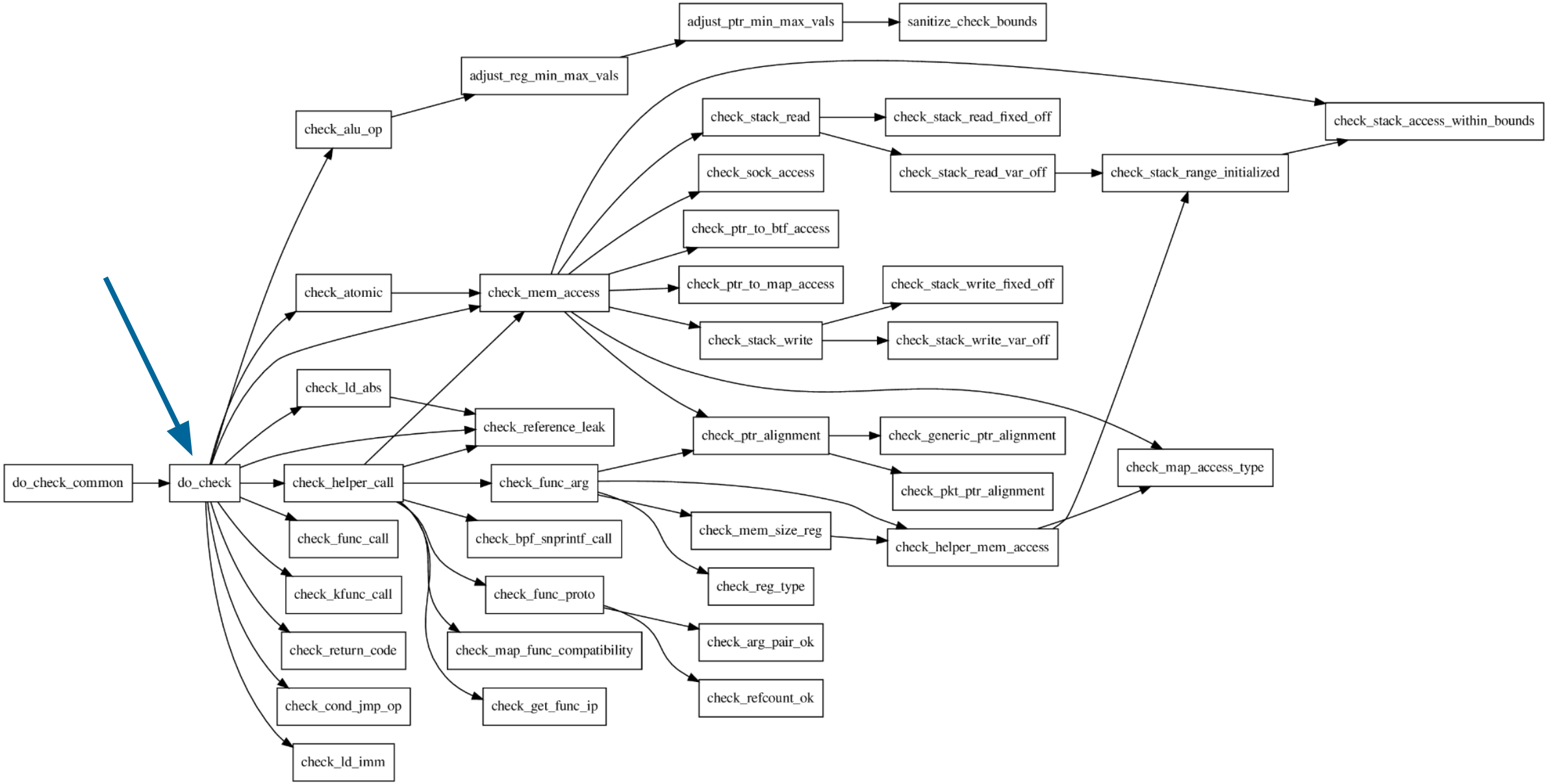
Registers	Type
r0	NOT_INIT
r1	PTR_TO_CTX
r2	NOT_INIT
r3	NOT_INIT
r4	NOT_INIT
...	...
r9	NOT_INIT
r10 (fp)	PTR_TO_STACK

Copyright © SUSE 2022

Diving Deeper

```
/* Heavily trimmed-down version */
static int do_check_common(struct bpf_verifier_env *env, int subprog) {
    init_func_state(env, ...); /* calls init_reg_state() */

    /* 1st arg to a function */
    regs[BPF_REG_1].type = PTR_TO_CTX;
    mark_reg_known_zero(env, regs, BPF_REG_1);

    ret = do_check(env);
    return ret;
}
```

Copyright © SUSE 2022



Copyright © SUSE 2022

Inferring Type & Value

```
static int do_check(struct bpf_verifier_env *env) { /* 332 lines */
	for (;;) {
		struct bpf_insn *insn = &insns[env->insn_idx];
		u8 class = BPF_CLASS(insn->code);

		if (class == BPF_ALU || class == BPF_ALU64) {
			...
		} else if (class == BPF_LDX) {
			...
		} else if (class == BPF_STX) {
			...
		} else if (class == BPF_ST) {
			...
		} else if (class == BPF_JMP || class == BPF_JMP32) {
			...
		} else if (class == BPF_LD) {
			...
		}

		env->insn_idx++;
	}
	return 0;
}
```

Copyright © SUSE 2022

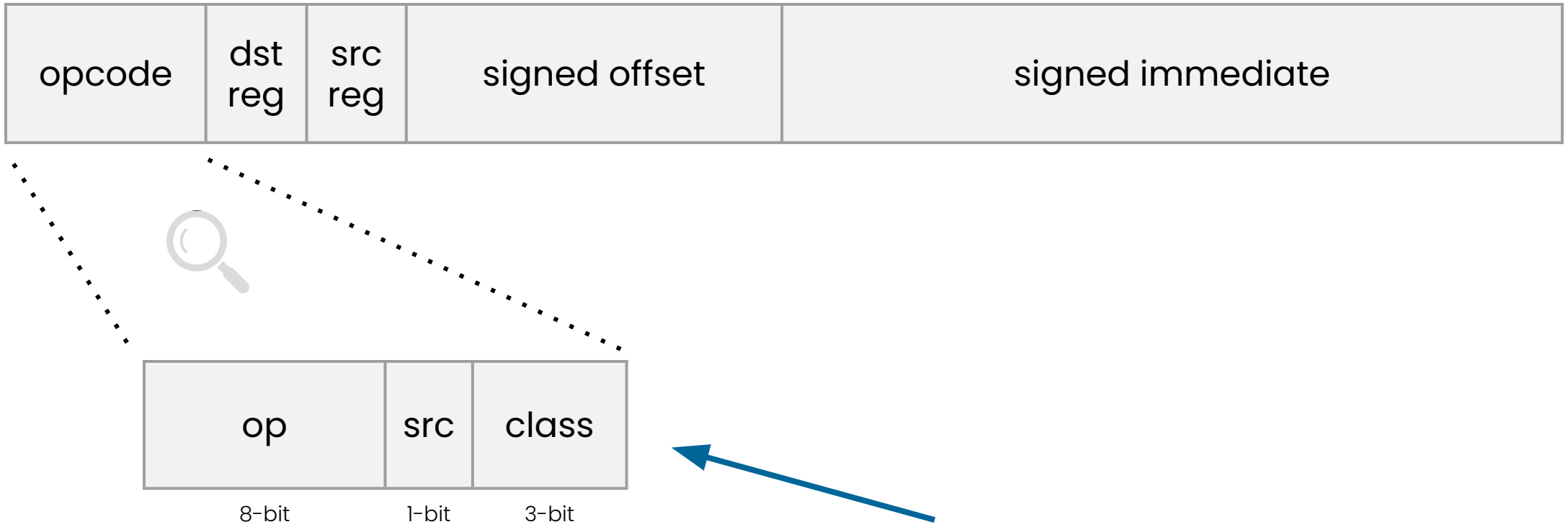
```
static int do_check(struct bpf_verifier_env *env) { /* 332 lines */
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
static int do_check(struct bpf_verifier_env *env) { /* 332 lines */
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

Inferring Type & Value



Copyright © SUSE 2022

```
static int do_check(struct bpf_verifier_env *env) { /* 332 lines */
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
static int do_check(struct bpf_verifier_env *env) { /* 332 lines */
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

Example Programs

Copyright © SUSE 2022

Correct Program

Starting with the simplest program

```
SEC("sock_filter")
int bpf_prog(struct __sk_buff *skb)
{
    return 1;
}
```

Copyright © SUSE 2022

Correct Program

But a correct problem is not fun

```
SEC("sock_filter")
int bpf_prog(struct __sk_buff *skb)
{
    return 1;
}
```

Copyright © SUSE 2022

Safety Guarantee

- Invalid memory use
  - Reading
  - Writing
- Address leakage
- Improper termination
- Violation of spec

Copyright © SUSE 2022

Leaking Kernel Address

What happens when we try to leak a pointer

```
SEC("socket1")
void *bpf_prog(struct __sk_buff *skb)
{
    /* Try to leak pointer 😈 */
    return (void*)skb;
}
```

Copyright © SUSE 2022

Leaking Kernel Address

What happens when we try to leak a pointer

```
SEC("socket1")
void *bpf_prog(struct __sk_buff *skb)
{
    /* Try to leak pointer 😈 */
    return (void*)skb;
}
```

Copyright © SUSE 2022

Leaking Kernel Address

We get a program with 2 instructions, a move instruction, and an exit instruction

```
0000000000000000 <bpf_prog>:
    0: r0 = r1;    BPF_ALU64_REG(BPF_MOV, BPF_REG_0, BPF_REG_1)
    1: exit;       BPF_EXIT_INSN()
```

Copyright © SUSE 2022

Leaking Kernel Address

BPF call convention is that **1st argument** (aka context) is passed through **r1**

```
SEC("socket1")
void *bpf_prog(struct __sk_buff *skb)
{
    /* Try to leak pointer 😈 */
    return (void*)skb;
}
```

Copyright © SUSE 2022

Leaking Kernel Address

BPF call convention is that **return value** is passed through **r0**

```
SEC("socket1")
void *bpf_prog(struct __sk_buff *skb)
{
    /* Try to leak pointer 😈 */
    return (void*)skb;
}
```

Copyright © SUSE 2022

Leaking Kernel Address

Let's now try to load this program into the kernel

```
0000000000000000 <bpf_prog>:
    0: r0 = r1;    BPF_ALU64_REG(BPF_MOV, BPF_REG_0, BPF_REG_1)
    1: exit;       BPF_EXIT_INSN()
```

Copyright © SUSE 2022

Leaking Kernel Address

Verifier rejects the program

```
libbpf: prog 'bpf_prog': BPF program load failed: Permission denied
libbpf: prog 'bpf_prog': -- BEGIN PROG LOAD LOG --
func#0 @0
0: R1=ctx(off=0,imm=0) R10=fp0
; void *bpf_prog(struct __sk_buff *skb)
0: (bf) r0 = r1                    ; R0_w=ctx(off=0,imm=0)
R1=ctx(off=0,imm=0)
; return skb;
1: (95) exit
R0 leaks addr as return value
```

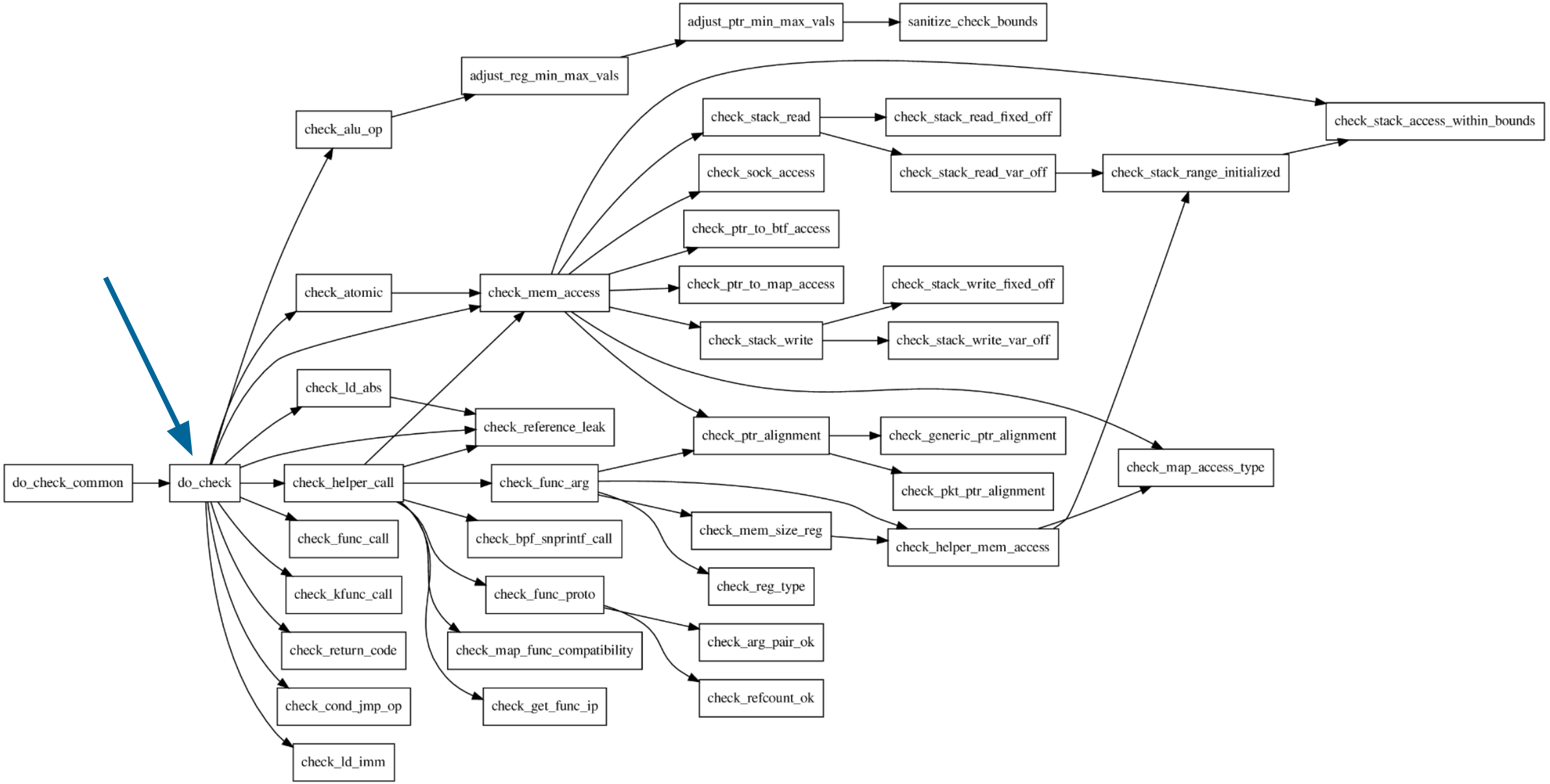
Copyright © SUSE 2022

Leaking Kernel Address

Let's look at the 1st instruction

```
0000000000000000 <bpf_prog>:
    0: r0 = r1;    BPF_ALU64_REG(BPF_MOV, BPF_REG_0, BPF_REG_1)
    1: exit;       BPF_EXIT_INSN()
```

Copyright © SUSE 2022



Copyright © SUSE 2022

```
/* defined in
 * include/linux/filter.h
 */
BPF_ALU64_REG(BPF_MOV,
              BPF_REG_0,
              BPF_REG_1)
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
/* defined in
 * include/linux/filter.h
 */
BPF_ALU64_REG(BPF_MOV,
              BPF_REG_0,
              BPF_REG_1)
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            err = check_alu_op(env, insn);
            if (err)
                return err;

        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            err = check_alu_op(env, insn);
            if (err)
                return err;

        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn)
{
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }

    return 0;
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn)
{
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }

    return 0;
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn)
{
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }

    return 0;
}
```

Copyright © SUSE 2022

```c
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```c
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn) {
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
        if (BPF_SRC(insn->code) == BPF_X) {
            ...
        } else { /* BPF_SRC(insn->code) == BPF_K */
            ...
        }
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn) {
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
        if (BPF_SRC(insn->code) == BPF_X) {
            ...
        } else { /* BPF_SRC(insn->code) == BPF_K */
            ...
        }
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn) {
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
        if (BPF_SRC(insn->code) == BPF_X) {
            ...
        } else { /* BPF_SRC(insn->code) == BPF_K */
            ...
        }
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        struct bpf_reg_state *src_reg =
            regs + insn->src_reg;
        struct bpf_reg_state *dst_reg =
            regs + insn->dst_reg;

        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            /* case: R1 = R2
             * copy register state to dest reg
             */
            *dst_reg = *src_reg;
        } else {
            ...
        }
    } else {
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        struct bpf_reg_state *src_reg =
            regs + insn->src_reg;
        struct bpf_reg_state *dst_reg =
            regs + insn->dst_reg;

        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            /* case: R1 = R2
             * copy register state to dest reg
             */
            *dst_reg = *src_reg;
        } else {
            ...
        }
    } else {
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        struct bpf_reg_state *src_reg =
            regs + insn->src_reg;
        struct bpf_reg_state *dst_reg =
            regs + insn->dst_reg;

        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            /* case: R1 = R2
             * copy register state to dest reg
             */
            *dst_reg = *src_reg;
        } else {
            ...
        }
    } else {
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        struct bpf_reg_state *src_reg =
            regs + insn->src_reg;
        struct bpf_reg_state *dst_reg =
            regs + insn->dst_reg;

        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            /* case: R1 = R2
             * copy register state to dest reg
             */
            *dst_reg = *src_reg;
        } else {
            ...
        }
    } else {
        ...
    }
```

Copyright © SUSE 2022

```c
struct bpf_reg_state {
    /* What kind of value is inside? */
    enum bpf_reg_type type;
    ...
};
```

```c
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```c
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        struct bpf_reg_state *src_reg =
            regs + insn->src_reg;
        struct bpf_reg_state *dst_reg =
            regs + insn->dst_reg;

        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            /* case: R1 = R2
             * copy register state to dest reg
             */
            *dst_reg = *src_reg;
        } else {
            ...
        }
    } else {
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_reg_state {
    /* What kind of value is inside? */
    enum bpf_reg_type type;
    ...
};
```

```
struct bpf_insn insn = {
        .code     = (BPF_ALU64
                     | BPF_MOV
                     | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off      = 0,
        .imm      = 0
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        struct bpf_reg_state *src_reg =
            regs + insn->src_reg;
        struct bpf_reg_state *dst_reg =
            regs + insn->dst_reg;

        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            /* case: R1 = R2
             * copy register state to dest reg
             */
            *dst_reg = *src_reg;
        } else {
            ...
        }
    } else {
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        struct bpf_reg_state *src_reg =
            regs + insn->src_reg;
        struct bpf_reg_state *dst_reg =
            regs + insn->dst_reg;

        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            /* case: R1 = R2
             * copy register state to dest reg
             */
            *dst_reg = *src_reg;
        } else {
            ...
        }
    } else {
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_X),
        .dst_reg = BPF_REG_0,
        .src_reg = BPF_REG_1,
        .off     = 0,
        .imm     = 0
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        struct bpf_reg_state *src_reg =
            regs + insn->src_reg;
        struct bpf_reg_state *dst_reg =
            regs + insn->dst_reg;

        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            /* case: R1 = R2
             * copy register state to dest reg
             */
            *dst_reg = *src_reg;
        } else {
            ...
        }            /* state of r0 = state of r1 */
    } else {
        ...
    }
```

Copyright © SUSE 2022

Leaking Kernel Address

Initial state

Registers	Type
r0	NOT_INIT
r1	PTR_TO_CTX
r2	NOT_INIT
...	...
r9	NOT_INIT
r10 (fp)	PTR_TO_STACK

Copyright © SUSE 2022

Leaking Kernel Address

Registers	Type
r0	NOT_INIT
r1	PTR_TO_CTX
r2	NOT_INIT
…	…
r9	NOT_INIT
r10 (fp)	PTR_TO_STACK

r0 = r1

BPF_ALU64_REG(BPF_MOV, BPF_REG_0, BPF_REG_1)

Registers	Type
r0	PTR_TO_CTX
r1	PTR_TO_CTX
r2	NOT_INIT
…	…
r9	NOT_INIT
r10 (fp)	PTR_TO_STACK

Copyright © SUSE 2022

Leaking Kernel Address

So far no rejection from the verifier yet

Now let's looks at the 2nd instruction

```
0000000000000000 <bpf_prog>:
    0: r0 = r1;     BPF_ALU64_REG(BPF_MOV, BPF_REG_0, BPF_REG_1)
    1: exit;        BPF_EXIT_INSN()
```

Copyright © SUSE 2022

```
/* defined in
 * include/linux/filter.h
 */
BPF_EXIT_INSN()
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
/* defined in
 * include/linux/filter.h
 */
BPF_EXIT_INSN()
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```c
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```c
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
if (class == BPF_ALU || class == BPF_ALU64) {
    ...
} else if (class == BPF_LDX) {
    ...
} else if (class == BPF_STX) {
    ...
} else if (class == BPF_ST) {
    ...
} else if (class == BPF_JMP || class == BPF_JMP32) {
    u8 opcode = BPF_OP(insn->code);

    if (opcode == BPF_CALL) {
        ...
    } else if (opcode == BPF_JA) {
        ...
    } else if (opcode == BPF_EXIT) {
        ...
    } else {
        ...
    }
} else if (class == BPF_LD) {
    ...
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
if (class == BPF_ALU || class == BPF_ALU64) {
    ...
} else if (class == BPF_LDX) {
    ...
} else if (class == BPF_STX) {
    ...
} else if (class == BPF_ST) {
    ...
} else if (class == BPF_JMP || class == BPF_JMP32) {
    u8 opcode = BPF_OP(insn->code);

    if (opcode == BPF_CALL) {
        ...
    } else if (opcode == BPF_JA) {
        ...
    } else if (opcode == BPF_EXIT) {
        ...
    } else {
        ...
    }
} else if (class == BPF_LD) {
    ...
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
if (class == BPF_ALU || class == BPF_ALU64) {
    ...
} else if (class == BPF_LDX) {
    ...
} else if (class == BPF_STX) {
    ...
} else if (class == BPF_ST) {
    ...
} else if (class == BPF_JMP || class == BPF_JMP32) {
    u8 opcode = BPF_OP(insn->code);

    if (opcode == BPF_CALL) {
        ...
    } else if (opcode == BPF_JA) {
        ...
    } else if (opcode == BPF_EXIT) {
        ...
    } else {
        ...
    }
} else if (class == BPF_LD) {
    ...
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
} else if (class == BPF_JMP || class == BPF_JMP32) {
    u8 opcode = BPF_OP(insn->code);

    if (opcode == BPF_CALL) {
        ...
    } else if (opcode == BPF_JA) {
        ...
    } else if (opcode == BPF_EXIT) {
        /* Unreleased resources */
        err = check_reference_leak(env);
        if (err)
            return err;

        err = check_return_code(env);
        if (err)
            return err;
    } else {
        ...
    }
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
} else if (class == BPF_JMP || class == BPF_JMP32) {
    u8 opcode = BPF_OP(insn->code);

    if (opcode == BPF_CALL) {
        ...
    } else if (opcode == BPF_JA) {
        ...
    } else if (opcode == BPF_EXIT) {
        /* Unreleased resources */
        err = check_reference_leak(env);
        if (err)
            return err;

        err = check_return_code(env);
        if (err)
            return err;
    } else {
        ...
    }
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
static int
check_return_code(struct bpf_verifier_env *env)
{
    /* regs[BPF_REG_0].type == NOT_INIT ? -EACCES : 0*/
    err = check_reg_arg(env, BPF_REG_0, SRC_OP);
    if (err)
        return err;

    /* regs[BPF_REG_0].type != SCALAR ? true : false*/
    if (is_pointer_value(env, BPF_REG_0)) {
        verbose(env, "R0 leaks addr as return value\n");
        return -EACCES;
    }

    return 0;
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
static int
check_return_code(struct bpf_verifier_env *env)
{
    /* regs[BPF_REG_0].type == NOT_INIT ? -EACCES : 0*/
    err = check_reg_arg(env, BPF_REG_0, SRC_OP);
    if (err)
        return err;

    /* regs[BPF_REG_0].type != SCALAR ? true : false*/
    if (is_pointer_value(env, BPF_REG_0)) {
        verbose(env, "R0 leaks addr as return value\n");
        return -EACCES;
    }

    return 0;
}
```

Copyright © SUSE 2022

Registers	Type
r0	PTR_TO_CTX

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
static int
check_return_code(struct bpf_verifier_env *env)
{
    /* regs[BPF_REG_0].type == NOT_INIT ? -EACCES : 0*/
    err = check_reg_arg(env, BPF_REG_0, SRC_OP);
    if (err)
        return err;

    /* regs[BPF_REG_0].type != SCALAR ? true : false*/
    if (is_pointer_value(env, BPF_REG_0)) {
        verbose(env, "R0 leaks addr as return value\n");
        return -EACCES;
    }

    return 0;
}
```

Copyright © SUSE 2022

Registers	Type
r0	PTR_TO_CTX

```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
static int
check_return_code(struct bpf_verifier_env *env)
{
    /* regs[BPF_REG_0].type == NOT_INIT ? -EACCES : 0*/
    err = check_reg_arg(env, BPF_REG_0, SRC_OP);
    if (err)
        return err;

    /* regs[BPF_REG_0].type != SCALAR ? true : false*/
    if (is_pointer_value(env, BPF_REG_0)) {
        verbose(env, "R0 leaks addr as return value\n");
        return -EACCES;
    }

    return 0;
}
```

Copyright © SUSE 2022

Registers	Type
r0	PTR_TO_CTX

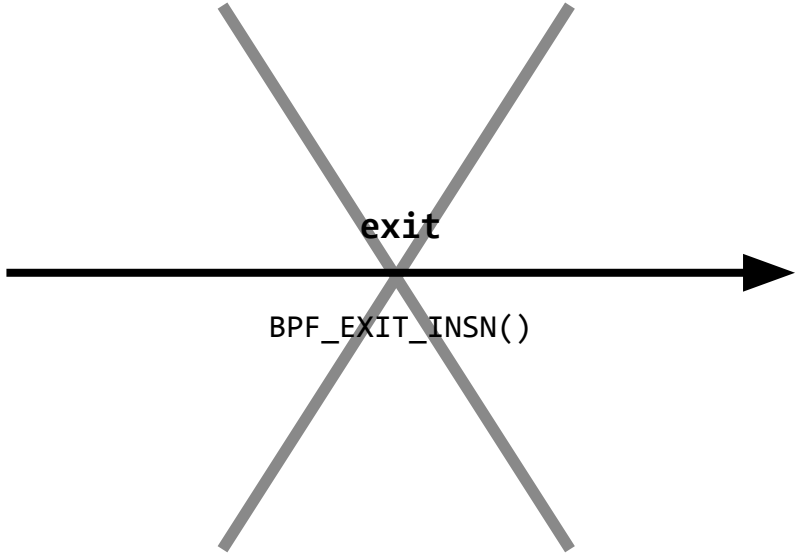
```
struct bpf_insn insn = {
        .code     = (BPF_JMP
                     | BPF_EXIT),
        .dst_reg = 0, /* unused */
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 0  /* unused */
}
```

```
static int
check_return_code(struct bpf_verifier_env *env)
{
    /* regs[BPF_REG_0].type == NOT_INIT ? -EACCES : 0*/
    err = check_reg_arg(env, BPF_REG_0, SRC_OP);
    if (err)
        return err;

    /* regs[BPF_REG_0].type != SCALAR ? true : false*/
    if (is_pointer_value(env, BPF_REG_0)) {
        verbose(env, "R0 leaks addr as return value\n");
        return -EACCES;
    }

    return 0;
}
```

Copyright © SUSE 2022

Inferring Type & Value

Registers	Type
r0	PTR_TO_CTX
r1	PTR_TO_CTX
r2	NOT_INIT
…	…
r9	NOT_INIT
r10 (fp)	PTR_TO_STACK



Copyright © SUSE 2022

Leaking Kernel Address

Verifier will reject the program

```
libbpf: prog 'bpf_prog': BPF program load failed: Permission denied
libbpf: prog 'bpf_prog': -- BEGIN PROG LOAD LOG --
func#0 @0
0: R1=ctx(off=0,imm=0) R10=fp0
; void *bpf_prog(struct __sk_buff *skb)
0: (bf) r0 = r1                    ; R0_w=ctx(off=0,imm=0)
R1=ctx(off=0,imm=0)
; return skb;
1: (95) exit
R0 leaks addr as return value
```

Copyright © SUSE 2022

Food for Thought

Will this pass?

Does the verifier rejects this program because it's returning a pointer?

```
SEC("socket1")
void *bpf_prog(struct __sk_buff *skb)
{
    return (void*)0;
}
```

Copyright © SUSE 2022

Tracking Value

Copyright © SUSE 2022

Safety Guarantee

- Invalid memory use
  - Reading
  - Writing
- Address leakage
- Improper termination
- Violation of spec

Copyright © SUSE 2022

Value Tracking

Will there be out-of-bound access?

```
unsigned int i;
char arr[4] = { 1, 2, 3, 4 };

SEC("sock_filter")
int bpf_prog(void)
{
    return arr[i]; /* Is i < 4 ? */
}
```

Copyright © SUSE 2022

Value Tracking

Verifier rejects the program

```
libbpf: prog 'bpf_prog': BPF program load failed: Permission denied
libbpf: prog 'bpf_prog': -- BEGIN PROG LOAD LOG --
0: R1=ctx(off=0,imm=0) R10=fp0
...
3: (67) r1 <<= 2                      ;
R1_w=scalar(umax=17179869180,var_off=(0x0;
0x3ffffffffc),s32_max=2147483644,u32_max=-4)
4: (18) r2 = 0x0                      ;
R2_w=map_value(off=0,ks=4,vs=16,imm=0)
6: (0f) r2 += r1
R2 unbounded memory access, make sure to bounds check any such access
R2 pointer arithmetic of map value goes out of range
```

Copyright © SUSE 2022

Value Tracking

How to do bound check?

```
unsigned int i;
char arr[4] = { 1, 2, 3, 4 };

SEC("sock_filter")
int bpf_prog(void)
{
    return arr[i]; /* Is i < 4 ? */
}
```

Copyright © SUSE 2022

Value Tracking

Adding bound check

Only after adding an if-statement will the program get pass the verifier

```
unsigned int i;
char arr[4] = { 1, 2, 3, 4 };

SEC("sock_filter")
int bpf_prog(void)
{
    if (i < 4) /* Bound check */
        return arr[i];
    return 0;
}
```

Copyright © SUSE 2022

Value Tracking

Adding bound check

Only after adding an if-statement will the program get pass the verifier

```
unsigned int i;
char arr[4] = { 1, 2, 3, 4 };

SEC("sock_filter")
int bpf_prog(void)
{
    if (i < 4) /* Bound check */
        return arr[i];
    return 0;
}
```

Copyright © SUSE 2022

Value Tracking

Adding bound check

Let's now see the compiled BPF instruction

```
unsigned int i;
char arr[4] = { 1, 2, 3, 4 };

SEC("sock_filter")
int bpf_prog(void)
{
    if (i < 4) /* Bound check */
        return arr[i];
    return 0;
}
```

Copyright © SUSE 2022

Value Tracking

```
0000000000000000 <bpf_prog>:
 0:                          BPF_LD_IMM64_RAW(BPF_REG_2, BPF_PSEUDO_MAP_VALUE...),
 2: r2 = i;                  BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_2, 0),
 3: r3 = arr;                BPF_LD_IMM64_RAW(BPF_REG_3, BPF_PSEUDO_MAP_VALUE...),
 5: if r2 > 3 goto 9;         BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5),
 6: r3 += r2;                BPF_ALU64_REG(BPF_ADD, BPF_REG_3, BPF_REG_2),
   /* ↑↑↑ this is the (arr + i) pointer arithmetic */
 7: r0 = *(r3 + 0);          BPF_LDX_MEM(BPF_DW, BPF_REG_0, BPF_REG_3, 0),
 8: exit;                    BPF_EXIT_INSN(),
 9: r0 = 0;                  BPF_MOV64_IMM(BPF_REG_0, 0),
10: exit;                    BPF_EXIT_INSN(),
```

Copyright © SUSE 2022

Value Tracking

Loading from memory **outside of BPF virtual machine**

Any value is possible

```
0000000000000000 <bpf_prog>:
 0:                          BPF_LD_IMM64_RAW(BPF_REG_2, BPF_PSEUDO_MAP_VALUE...),
 2: r2 = i;                  BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_2, 0),
 3: r3 = arr;                BPF_LD_IMM64_RAW(BPF_REG_3, BPF_PSEUDO_MAP_VALUE...),
 5: if r2 > 3 goto 9;         BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5),
 6: r3 += r2;                BPF_ALU64_REG(BPF_ADD, BPF_REG_3, BPF_REG_2),
   /* ^^^ this is the (arr + i) pointer arithmetic */
 7: r0 = *(r3 + 0);          BPF_LDX_MEM(BPF_DW, BPF_REG_0, BPF_REG_3, 0),
 8: exit;                    BPF_EXIT_INSN(),
 9: r0 = 0;                  BPF_MOV64_IMM(BPF_REG_0, 0),
10: exit;                    BPF_EXIT_INSN(),
```

Copyright © SUSE 2022

```
/* defined in
 * include/linux/filter.h
 */
BPF_LDX_MEM(BPF_DW,
            BPF_REG_2,
            BPF_REG_2, 0)
```

```
static int check_mem_access(...)
{
    if (reg->type == PTR_TO_MAP_KEY) {
        ...
    } else if (reg->type == PTR_TO_MAP_VALUE) {
        if (t == BPF_READ && value_regno >= 0) {
            mark_reg_unknown(env, regs, value_regno);
        } else {
        ...
        }
    } else if (base_type(reg->type) == PTR_TO_MEM) {
        ...
    } else if (reg->type == PTR_TO_CTX) {
        ...
    } else if (reg->type == PTR_TO_STACK) {
        ...
    } else if (reg_is_pkt_pointer(reg)) {
        ...
    } else if (reg->type == PTR_TO_FLOW_KEYS) {
        ...
    } else if (type_is_sk_pointer(reg->type)) {
        ...
    } else if (reg->type == PTR_TO_TP_BUFFER) {
```

Copyright © SUSE 2022

Value Tracking

Register	Type	Range
r2	NOT_INIT	-

r2 = i

```
BPF_LD_IMM64_RAW(BPF_REG_2, BPF_PSEUDO_MAP_VALUE...)
BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_2, 0)
```

Register	Type	Range
r2	SCALAR_VALUE	$0-2^{64}$

Copyright © SUSE 2022

Value Tracking

```
0000000000000000 <bpf_prog>:
 0:                          BPF_LD_IMM64_RAW(BPF_REG_2, BPF_PSEUDO_MAP_VALUE...),
 2: r2 = i;                  BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_2, 0),
 3: r3 = arr;                BPF_LD_IMM64_RAW(BPF_REG_3, BPF_PSEUDO_MAP_VALUE...),
 5: if r2 > 3 goto 9;        BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5),
 6: r3 += r2;                BPF_ALU64_REG(BPF_ADD, BPF_REG_3, BPF_REG_2),
   /* ^^^ this is the (arr + i) pointer arithmetic */
 7: r0 = *(r3 + 0);          BPF_LDX_MEM(BPF_DW, BPF_REG_0, BPF_REG_3, 0),
 8: exit;                    BPF_EXIT_INSN(),
 9: r0 = 0;                  BPF_MOV64_IMM(BPF_REG_0, 0),
10: exit;                    BPF_EXIT_INSN(),
```

Copyright © SUSE 2022

Value Tracking

```
0000000000000000 <bpf_prog>:
 0:                          BPF_LD_IMM64_RAW(BPF_REG_2, BPF_PSEUDO_MAP_VALUE...),
 2: r2 = i;                  BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_2, 0),
 3: r3 = arr;                BPF_LD_IMM64_RAW(BPF_REG_3, BPF_PSEUDO_MAP_VALUE...),
 5: if r2 > 3 goto 9;        BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5),
 6: r3 += r2;                BPF_ALU64_REG(BPF_ADD, BPF_REG_3, BPF_REG_2),
   /* ↑↑↑ this is the (arr + i) pointer arithmetic */
 7: r0 = *(r3 + 0);          BPF_LDX_MEM(BPF_DW, BPF_REG_0, BPF_REG_3, 0),
 8: exit;                    BPF_EXIT_INSN(),
 9: r0 = 0;                  BPF_MOV64_IMM(BPF_REG_0, 0),
10: exit;                    BPF_EXIT_INSN(),
```

Copyright © SUSE 2022

Value Tracking

```
0000000000000000 <bpf_prog>:
 0:                          BPF_LD_IMM64_RAW(BPF_REG_2, BPF_PSEUDO_MAP_VALUE...),
 2: r2 = i;                  BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_2, 0),
 3: r3 = arr;                BPF_LD_IMM64_RAW(BPF_REG_3, BPF_PSEUDO_MAP_VALUE...),
 5: if r2 > 3 goto 9;        BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5),
 6: r3 += r2;                BPF_ALU64_REG(BPF_ADD, BPF_REG_3, BPF_REG_2),
   /* ^^^ this is the (arr + i) pointer arithmetic */
 7: r0 = *(r3 + 0);          BPF_LDX_MEM(BPF_DW, BPF_REG_0, BPF_REG_3, 0),
 8: exit;                    BPF_EXIT_INSN(),
 9: r0 = 0;                  BPF_MOV64_IMM(BPF_REG_0, 0),
10: exit;                    BPF_EXIT_INSN(),
```

Copyright © SUSE 2022

Value Tracking

```
0000000000000000 <bpf_prog>:
 0:                          BPF_LD_IMM64_RAW(BPF_REG_2, BPF_PSEUDO_MAP_VALUE...),
 2: r2 = i;                  BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_2, 0),
 3: r3 = arr;                BPF_LD_IMM64_RAW(BPF_REG_3, BPF_PSEUDO_MAP_VALUE...),
 5: if r2 > 3 goto 9;        BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5),
 6: r3 += r2;                BPF_ALU64_REG(BPF_ADD, BPF_REG_3, BPF_REG_2),
   /* ^^^ this is the (arr + i) pointer arithmetic */
 7: r0 = *(r3 + 0);          BPF_LDX_MEM(BPF_DW, BPF_REG_0, BPF_REG_3, 0),
 8: exit;                    BPF_EXIT_INSN(),
 9: r0 = 0;                  BPF_MOV64_IMM(BPF_REG_0, 0),
10: exit;                    BPF_EXIT_INSN(),
```

Copyright © SUSE 2022

```
/* defined in
 * include/linux/filter.h
 */
BPF_JMP_IMM(BPF_JGT,
            BPF_REG_2,
            3,
            5)
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
/* defined in
 * include/linux/filter.h
 */
BPF_JMP_IMM(BPF_JGT,
            BPF_REG_2,
            3,
            5)
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```c
struct bpf_insn insn = {
        .code    = (BPF_JMP
                   | BPF_JGT
                   | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```c
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                   | BPF_JGT
                   | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
if (class == BPF_ALU || class == BPF_ALU64) {
    ...
} else if (class == BPF_LDX) {
    ...
} else if (class == BPF_STX) {
    ...
} else if (class == BPF_ST) {
    ...
} else if (class == BPF_JMP || class == BPF_JMP32) {
    u8 opcode = BPF_OP(insn->code);

    if (opcode == BPF_CALL) {
        ...
    } else if (opcode == BPF_JA) {
        ...
    } else if (opcode == BPF_EXIT) {
        ...
    } else {
        ...
    }
} else if (class == BPF_LD) {
    ...
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
if (class == BPF_ALU || class == BPF_ALU64) {
    ...
} else if (class == BPF_LDX) {
    ...
} else if (class == BPF_STX) {
    ...
} else if (class == BPF_ST) {
    ...
} else if (class == BPF_JMP || class == BPF_JMP32) {
    u8 opcode = BPF_OP(insn->code);

    if (opcode == BPF_CALL) {
        ...
    } else if (opcode == BPF_JA) {
        ...
    } else if (opcode == BPF_EXIT) {
        ...
    } else {
        ...
    }
} else if (class == BPF_LD) {
    ...
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
if (class == BPF_ALU || class == BPF_ALU64) {
    ...
} else if (class == BPF_LDX) {
    ...
} else if (class == BPF_STX) {
    ...
} else if (class == BPF_ST) {
    ...
} else if (class == BPF_JMP || class == BPF_JMP32) {
    u8 opcode = BPF_OP(insn->code);

    if (opcode == BPF_CALL) {
        ...
    } else if (opcode == BPF_JA) {
        ...
    } else if (opcode == BPF_EXIT) {
        ...
    } else {
        ...
    }
} else if (class == BPF_LD) {
    ...
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
} else if (class == BPF_ST) {
    ...
} else if (class == BPF_JMP || class == BPF_JMP32) {
    u8 opcode = BPF_OP(insn->code);

    if (opcode == BPF_CALL) {
        ...
    } else if (opcode == BPF_JA) {
        ...
    } else if (opcode == BPF_EXIT) {
        ...
    } else {
        err = check_cond_jmp_op(env,
                                insn,
                                &env->insn_idx);
        if (err)
            return err;
} else if (class == BPF_LD) {
    ...
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
} else if (class == BPF_ST) {
    ...
} else if (class == BPF_JMP || class == BPF_JMP32) {
    u8 opcode = BPF_OP(insn->code);

    if (opcode == BPF_CALL) {
        ...
    } else if (opcode == BPF_JA) {
        ...
    } else if (opcode == BPF_EXIT) {
        ...
    } else {
        err = check_cond_jmp_op(env,
                                insn,
                                &env->insn_idx);
        if (err)
            return err;
} else if (class == BPF_LD) {
    ...
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
static int check_cond_jmp_op(struct bpf_verifier_env
*env,
                    struct bpf_insn *insn, int *insn_idx)
{
    other_branch = push_stack(env,
                              *insn_idx + insn->off + 1,
                              *insn_idx, false);
    other_branch_regs =
     other_branch->frame[other_branch->curframe]->regs;

    if (BPF_SRC(insn->code) == BPF_X) {
        ...
    } else if (dst_reg->type == SCALAR_VALUE) {
        reg_set_min_max(
                &other_branch_regs[insn->dst_reg],
                dst_reg, insn->imm, (u32)insn->imm,
                opcode, is_jmp32);
    }
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
static int check_cond_jmp_op(struct bpf_verifier_env
*env,
                             struct bpf_insn *insn, int *insn_idx)
{
    other_branch = push_stack(env,
                              *insn_idx + insn->off + 1,
                              *insn_idx, false);
    other_branch_regs =
     other_branch->frame[other_branch->curframe]->regs;

    if (BPF_SRC(insn->code) == BPF_X) {
        ...
    } else if (dst_reg->type == SCALAR_VALUE) {
        reg_set_min_max(
                &other_branch_regs[insn->dst_reg],
                dst_reg, insn->imm, (u32)insn->imm,
                opcode, is_jmp32);
    }
}
```

Copyright © SUSE 2022

Value Tracking

Register	Type	Range
r2	SCALAR_VALUE	$0-2^{64}$

```
if r2 > 3
goto 9
```

```
BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5)
```

```
r2 > 3 == false
proceed
```

```
r2 > 3 == true
jump
```

Register	Type	Range
r2	?	?

Register	Type	Range
r2	?	?

Copyright © SUSE 2022

```c
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```c
static int check_cond_jmp_op(struct bpf_verifier_env
*env,
                             struct bpf_insn *insn, int *insn_idx)
{
    other_branch = push_stack(env,
                              *insn_idx + insn->off + 1,
                              *insn_idx, false);
    other_branch_regs =
     other_branch->frame[other_branch->curframe]->regs;

    if (BPF_SRC(insn->code) == BPF_X) {
        ...
    } else if (dst_reg->type == SCALAR_VALUE) {
        reg_set_min_max(
                &other_branch_regs[insn->dst_reg],
                dst_reg, insn->imm, (u32)insn->imm,
                opcode, is_jmp32);
    }
}
```

Copyright © SUSE 2022

```c
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```c
static void reg_set_min_max(
                struct bpf_reg_state *true_reg,
                struct bpf_reg_state *false_reg,
                u64 val, u32 val32,
                u8 opcode, bool is_jmp32)
{
    switch (opcode) {
    case BPF_JEQ:
    case BPF_JNE:
        ...
    case BPF_JSET:
        ...
    case BPF_JGT:
        ...
    case BPF_JSGE:
    case BPF_JSGT:
        ...
    case BPF_JLE:
    case BPF_JLT:
        ...
    }
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                   | BPF_JGT
                   | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
static void reg_set_min_max(
                struct bpf_reg_state *true_reg,
                struct bpf_reg_state *false_reg,
                u64 val, u32 val32,
                u8 opcode, bool is_jmp32)
{
    switch (opcode) {
    case BPF_JEQ:
    case BPF_JNE:
        ...
    case BPF_JSET:
        ...
    case BPF_JGT:
        ...
    case BPF_JSGE:
    case BPF_JSGT:
        ...
    case BPF_JLE:
    case BPF_JLT:
        ...
    }
}
```

Copyright © SUSE 2022

```c
struct bpf_insn insn = {
        .code    = (BPF_JMP
                   | BPF_JGT
                   | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```c
switch (opcode) {
case BPF_JEQ:
case BPF_JNE:
    ...
case BPF_JSET:
    ...
case BPF_JGT:
    false_reg->umax_value =
        min(false_reg->umax_value, val);

    true_reg->umin_value =
        max(true_reg->umin_value, val + 1);

    break;
case BPF_JSGE:
case BPF_JSGT:
    ...
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
switch (opcode) {
case BPF_JEQ:
case BPF_JNE:
    ...
case BPF_JSET:
    ...
case BPF_JGT:
    false_reg->umax_value =
        min(false_reg->umax_value, val);

    true_reg->umin_value =
        max(true_reg->umin_value, val + 1);

    break;
case BPF_JSGE:
case BPF_JSGT:
    ...
}
```

Copyright © SUSE 2022

Value Tracking

Register	Type	Range
r2	SCALAR_VALUE	$0-2^{64}$

```
if r2 > 3
goto 9
```

```
BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5)
```

```
r2 > 3 == false
proceed
```

```
r2 > 3 == true
jump
```

Register	Type	Range
r2	?	?-3

Register	Type	Range
r2	?	?

Copyright © SUSE 2022

Value Tracking

Register	Type	Range
r2	SCALAR_VALUE	$0-2^{64}$

```
if r2 > 3
goto 9
```

```
BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5)
```

```
r2 > 3 == false
proceed
```

```
r2 > 3 == true
jump
```

Register	Type	Range
r2	SCALAR_VALUE	0-3

Register	Type	Range
r2	?	?

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_JMP
                    | BPF_JGT
                    | BPF_K),
        .dst_reg = BPF_REG_2,
        .src_reg = 0, /* unused */
        .off     = 5, /* jump insn */
        .imm     = 3
}
```

```
switch (opcode) {
case BPF_JEQ:
case BPF_JNE:
    ...
case BPF_JSET:
    ...
case BPF_JGT:
    false_reg->umax_value =
        min(false_reg->umax_value, val);

    true_reg->umin_value =
        max(true_reg->umin_value, val + 1);

    break;
case BPF_JSGE:
case BPF_JSGT:
    ...
}
```

Copyright © SUSE 2022

Value Tracking

Register	Type	Range
r2	SCALAR_VALUE	$0-2^{64}$

```
if r2 > 3
goto 9
```

```
BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5)
```

```
r2 > 3 == false
proceed
```

```
r2 > 3 == true
jump
```

Register	Type	Range
r2	SCALAR_VALUE	0-3

Register	Type	Range
r2	?	4-?

Copyright © SUSE 2022

Value Tracking

Register	Type	Range
r2	SCALAR_VALUE	$0-2^{64}$

```
if r2 > 3
goto 9
```

```
BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5)
```

```
r2 > 3 == false
proceed
```

```
r2 > 3 == true
jump
```

Register	Type	Range
r2	SCALAR_VALUE	0-3

Register	Type	Range
r2	SCALAR_VALUE	$4-2^{64}$

Copyright © SUSE 2022

Value Tracking

Register	Type	Range
r2	SCALAR_VALUE	$0-2^{64}$

```
if r2 > 3
goto 9
```

```
BPF_JMP_IMM(BPF_JGT, BPF_REG_2, 3, 5)
```

```
r2 > 3 == false
proceed
```

```
r2 > 3 == true
jump
```

Register	Type	Range
r2	SCALAR_VALUE	0-3

Register	Type	Range
r2	SCALAR_VALUE	$4-2^{64}$

Copyright © SUSE 2022

Value Tracking

Now continue the analysis for both of the states

Register	Type	Range
r2	SCALAR_VALUE	0-3

Register	Type	Range
r2	SCALAR_VALUE	$4-2^{64}$

Copyright © SUSE 2022

Difficulty

Value tracking is **easy in concept**, but **difficult in practice** due to

- Sígness
- Overflow/Underflow
- Alignment and size (64-bit vs 32-bit)
- Different types
  - Pointer arithmetics
- Algorithmic complexity due to branching

Copyright © SUSE 2022

Value Tracking

```
struct bpf_reg_state {
    enum bpf_reg_type type;
    s64 smin_value; /* minimum possible (s64)value */
    s64 smax_value; /* maximum possible (s64)value */
    u64 umin_value; /* minimum possible (u64)value */
    u64 umax_value; /* maximum possible (u64)value */
    s32 s32_min_value; /* minimum possible (s32)value */
    s32 s32_max_value; /* maximum possible (s32)value */
    u32 u32_min_value; /* minimum possible (u32)value */
    u32 u32_max_value; /* maximum possible (u32)value */
    struct tnum var_off;
    ...
};
```

Copyright © SUSE 2022

Tristate Numbers

Value Tracking++

Copyright © SUSE 2022

Tristate Numbers

How BPF verifier tracks values

Representing **integer 1** with tristate number (far right)

$$\{1\} = \{01_2\} \rightarrow 01_t$$

Copyright © SUSE 2022

Tristate Numbers

How BPF verifier tracks values

Representing **integer that's either 1 or 3** with tristate number

$$\{1, 3\} = \{01_2, 11_2\} \rightarrow ?1_t$$

Copyright © SUSE 2022

Tristate Numbers

How BPF verifier tracks values

Representing **integer that's either 0 or 2** with tristate number

$$\{0, 2\} = \{00_2, 10_2\} \rightarrow ?0_t$$

Copyright © SUSE 2022

Tristate Numbers

$$
\begin{aligned}
\{0\} &= \{00_2\} & &\rightarrow 00_t \\
\{0,1\} &= \{00_2, 01_2\} & &\rightarrow 0?_t \\
\{1,2\} &= \{01_2, 10_2\} & &\rightarrow ??_t \\
\{2,3\} &= \{10_2, 11_2\} & &\rightarrow 1?_t \\
\{1,2,3\} &= \{01_2, 10_2, 11_2\} & &\rightarrow ??_t \\
\{0,1,2,3\} &= \{00_2, 01_2, 10_2, 11_2\} & &\rightarrow ??_t
\end{aligned}
$$

Copyright © SUSE 2022

Tristate Numbers

$$
\begin{aligned}
\{0\} &= \{00_2\} & &\rightarrow 00_t \\
\{0,1\} &= \{00_2, 01_2\} & &\rightarrow 0?_t \\
\{1,2\} &= \{01_2, 10_2\} & &\rightarrow ??_t \\
\{2,3\} &= \{10_2, 11_2\} & &\rightarrow 1?_t \\
\{1,2,3\} &= \{01_2, 10_2, 11_2\} & &\rightarrow ??_t \\
\{0,1,2,3\} &= \{00_2, 01_2, 10_2, 11_2\} & &\rightarrow ??_t
\end{aligned}
$$

precision loss

Copyright © SUSE 2022

Addition

Arithmetic operations on tristate numbers

Value we're tracking can undergo arithmetic operations

- These must be tracked as well!

Copyright © SUSE 2022

Addition

Example: 1 plus (0 or 1)

$$\{1\} + \{0, 1\} = \left\{ \begin{array}{c} 1 + 0 \\ 1 + 1 \end{array} \right\} = \{1, 2\}$$

Copyright © SUSE 2022

Addition

Example: 1 plus (0 or 1)

$$\{1\} + \{0, 1\} = \left\{ \begin{array}{c} 1 + 0 \\ 1 + 1 \end{array} \right\} = \{1, 2\}$$

$$\{01_2\} + \{00_2, 01_2\} = \left\{ \begin{array}{c} 01_2 + 00_2 \\ 01_2 + 01_2 \end{array} \right\} = \{01_2, 10_2\}$$

Copyright © SUSE 2022

Addition

Example: 1 plus (0 or 1)

$$\{1\} + \{0, 1\} = \left\{ \begin{array}{c} 1 + 0 \\ 1 + 1 \end{array} \right\} = \{1, 2\}$$

$$\{01_2\} + \{00_2, 01_2\} = \left\{ \begin{array}{c} 01_2 + 00_2 \\ 01_2 + 01_2 \end{array} \right\} = \{01_2, 10_2\}$$

$$01_t + 0?_t = ??_t$$

Copyright © SUSE 2022

Addition

How Linux Kernel implements it

```
struct tnum tnum_add(struct tnum a, struct tnum b)
{
    u64 sm, sv, sigma, chi, mu;

    sm = a.mask + b.mask;
    sv = a.value + b.value;
    sigma = sm + sv;
    chi = sigma ^ sv;
    mu = chi | a.mask | b.mask;
    return TNUM(sv & ~mu, mu);
}
```

Copyright © SUSE 2022

Addition

How Linux Kernel implements it

```
struct tnum tnum_add(struct tnum a, struct tnum b)
{
    u64 sm, sv, sigma, chi, mu;

    sm = a.mask + b.mask;
    sv = a.value + b.value;
    sigma = sm + sv;
    chi = sigma ^ sv;
    mu = chi | a.mask | b.mask;
    return TNUM(sv & ~mu, mu);
}
```

¯_(ツ)_/¯

Copyright © SUSE 2022

Addition

How Linux Kernel implements it

```
struct tnum tnum_add(struct tnum a, struct tnum b)
{
    u64 sm, sv, sigma, chi, mu;

    sm = a.mask + b.mask;
    sv = a.value + b.value;
    sigma = sm + sv;
    chi = sigma ^ sv;
    mu = chi | a.mask | b.mask;
    return TNUM(sv & ~mu, mu);
}
```

Formally verified

- Model Checking (a very small part) of BPF Verifier
- Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers

Copyright © SUSE 2022

Tip and Tricks

Copyright © SUSE 2021

Source Code Browsing

Vim/Neovim + Cscope

- Add cscope_maps.vim as Vim plugin
- Run `make cscope`
- Start browsing
  - Ctrl+] to jump to definition
  - Ctrl+I and Ctrl+O to jump forward and back

Copyright © SUSE 2022

Commit Hunting

Git

- `git blame`
- `git log --no-merges --oneline --grep "$KEYWORD" -- kernel/bpf/ include/linux/b{p,t}f*`
  - To search within source code, replace `--grep` with `-S` or `-G`

Google

- $KEYWORD site:lore.kernel.org

Copyright © SUSE 2022

Dumping BPF Instructions

Two main method

If it's a `*.o` ELF object file:

- `llvm-objdump -dr $OBJ` to show BPF instruction it contains
- `llvm-objdump -Sr $OBJ` to show to **source** as well (is debuginfo available)

Another way is to dump the BPF program that's already loaded into the kernel

- BPF verifier can print the instruction (see next slide)
- `bpftool prog dump xlated id $PROG_ID` (the program has to **pass verifier first** though)

Copyright © SUSE 2022

BPF Verifier Log

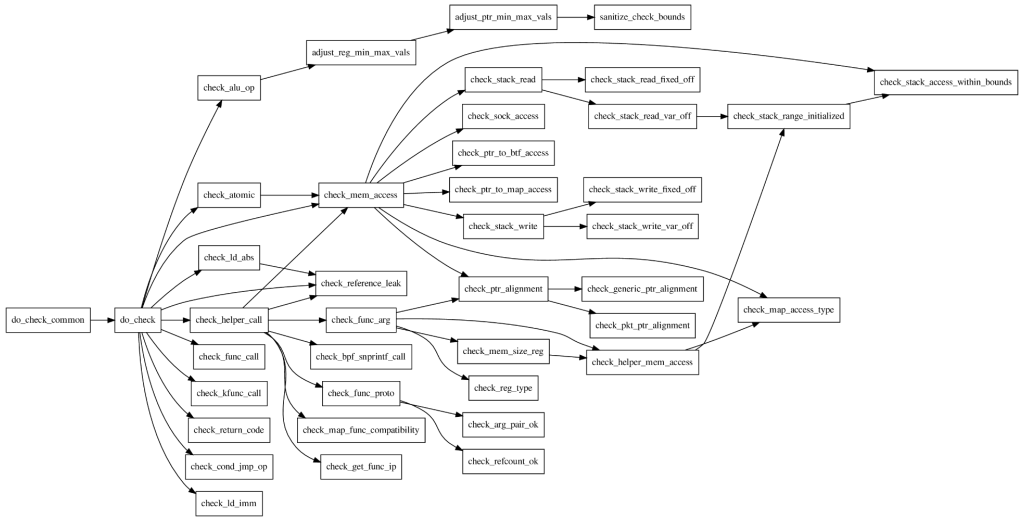
Set log_level to 3 `== (BPF_LOG_LEVEL1 | BPF_LOG_LEVEL2)` by either

- Set `bpf_attr.log_level` and call `printf("%s", bpf_attr.log_buf)` after program load or
- Set `bpf_object_open_opts.kernel_log_level` and set callback with `libbpf_set_print(libbpf_print_fn)`

Copyright © SUSE 2022

Function Call Diagram

The generation of the call graph is done with cflow, graphviz and some ad-hoc gvpr (also part of graphviz) script for post-processing.



Copyright © SUSE 2022

Syntax Highlighting

Pygments

```
wl-paste |
    pygmentize \
    -l c \
    -O style=manni,noclasses=True,nobackground=True,prestyles='font-family:
Consolas, monospace' \
    -f html |
    wl-copy -t text/html
```

Copyright © SUSE 2022

Other Interesting Topics

Copyright © SUSE 2021

Instruction patching (can be seen in patched form with `bpftool prog dump xlated`)

- Code sanitization
  - Spectre mitigation
  - Bound limiting
  - Make divide-by-zero impossible

Copyright © SUSE 2022

How tracing is done (unrelated to verifier)

- Making memory read/write safe

Copyright © SUSE 2022

Resources

Copyright © SUSE 2021

CVE Writeups

- Zero Day Initiative — CVE-2020-8835: Linux Kernel Privilege Escalation via Improper eBPF Program Verification
- Kernel Pwning with eBPF: a Love Story — Grapl
- CVE-2021-34866 Writeup — HexRabbit's Blog

Copyright © SUSE 2022

Talks

LISA21 - BPF Internals

Safe Programs The Foundation of BPF

Copyright © SUSE 2022

Documentation & Posts

eBPF Verifier — The Linux Kernel Documentation

bpf/verifier.c

BPF Verifier Overview — XDP Newbie

Copyright © SUSE 2022

Commits

Rework value tracking

Bounded loop

Propagate scalar ranges

Copyright © SUSE 2022

Research

Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers

Simple and Precise Static Analysis of Untrusted Linux Kernel Extension

Copyright © SUSE 2022

Courses and Topics

Linux Kernel & C

- 「你所不知道的 C 語言」系列講座 & 「Linux 核心設計」系列講座

Static Analysis

Abstract Interpretation

- Interval domain, Bitfield domain

Symbolic Execution

Copyright © SUSE 2022

SUSE

Thank you

For more information, contact SUSE at:

+1 800 796 3700 (U.S./Canada)

Frankenstrasse 146

90461 Nürnberg

www.suse.com

© 2022 SUSE LLC. All Rights Reserved. SUSE and the SUSE logo are registered trademarks of SUSE LLC in the United States and other countries. All third-party trademarks are the property of their respective owners.

Feel free to reach out to me on Twitter @shunghsiyu

Appendix

Copyright © SUSE 2021

Usual workflow of BPF

1. Write BPF program in C
2. Compile C to BPF instructions (usually with LLVM)
3. Load BPF instruction into the kernel ← this is when **BPF verifier runs**
4. Kernel JIT-compiles BPF instruction into native instructions
5. Attach BPF program to in-kernel hooks
6. BPF program runs when hooks are triggered

Copyright © SUSE 2022

Verifier Admission

When a program passes the verifier, it means

- Verifier guarantees that the program is safe

When verifier let an **unsafe program pass**, it's a **BUG**
(and usually security vulnerability as well)

Copyright © SUSE 2022

Verifier Rejection

Verifier rejects a program, when it can't guarantee it's safety

Which can mean either

- The program have unsafe behavior and is correctly rejected, or
- The program **does not** actually **exhibit unsafe behavior**, but the verifier wasn't advance enough to know it (not a bug, more like an **unimplemented feature**)

Copyright © SUSE 2022

Instruction Check

Copyright © SUSE 2022

Instruction Check

```
static int resolve_pseudo_ldimm64(struct bpf_verifier_env *env) {
        struct bpf_insn *insn = env->prog->insnsi;
        int insn_cnt = env->prog->len;
        ...
        for (i = 0; i < insn_cnt; i++, insn++) {
                ...

                /* Basic sanity check before we invest more work here. */
                if (!bpf_opcode_in_insntable(insn->code)) {
                        verbose(env, "unknown opcode %02x\n", insn->code);
                        return -EINVAL;
                }
        }

        return 0;
}
```

Copyright © SUSE 2022

Instruction Check

```
static int resolve_pseudo_ldimm64(struct bpf_verifier_env *env) {
        struct bpf_insn *insn = env->prog->insnsi;
        int insn_cnt = env->prog->len;
        ...
        for (i = 0; i < insn_cnt; i++, insn++) {
                ...

                /* Basic sanity check before we invest more work here. */
                if (!bpf_opcode_in_insntable(insn->code)) {
                        verbose(env, "unknown opcode %02x\n", insn->code);
                        return -EINVAL;
                }
        }

        return 0;
}
```

Copyright © SUSE 2022

Instruction Check

```
bool bpf_opcode_in_insntable(u8 code) {
#define BPF_INSN_2_TBL(x, y)    [BPF_##x | BPF_##y] = true
#define BPF_INSN_3_TBL(x, y, z) [BPF_##x | BPF_##y | BPF_##z] = true
        static const bool public_insntable[256] = {
                [0 ... 255] = false,
                /* Now overwrite non-defaults ... */
                BPF_INSN_MAP(BPF_INSN_2_TBL, BPF_INSN_3_TBL),
                /* UAPI exposed, but rewritten opcodes. cBPF carry-over. */
                [BPF_LD | BPF_ABS | BPF_B] = true,
                ...
        };
#undef BPF_INSN_3_TBL
#undef BPF_INSN_2_TBL
        return public_insntable[code];
}
```

Copyright © SUSE 2022

Instruction Check

```
bool bpf_opcode_in_insntable(u8 code) {
#define BPF_INSN_2_TBL(x, y)    [BPF_##x | BPF_##y] = true
#define BPF_INSN_3_TBL(x, y, z) [BPF_##x | BPF_##y | BPF_##z] = true
        static const bool public_insntable[256] = {
                [0 ... 255] = false,
                /* Now overwrite non-defaults ... */
                BPF_INSN_MAP(BPF_INSN_2_TBL, BPF_INSN_3_TBL),
                /* UAPI exposed, but rewritten opcodes. cBPF carry-over. */
                [BPF_LD | BPF_ABS | BPF_B] = true,
                ...
        };
#undef BPF_INSN_3_TBL
#undef BPF_INSN_2_TBL
        return public_insntable[code];
}
```

Copyright © SUSE 2022

Instruction Check

```
bool bpf_opcode_in_insntable(u8 code) {
        static const bool public_insntable[256] = {
                [0 ... 255] = false,
                /* Now overwrite non-defaults ... */
                BPF_INSN_MAP(BPF_INSN_2_TBL, BPF_INSN_3_TBL),
        };
        return public_insntable[code];
}
```

Copyright © SUSE 2022

Instruction Check

```
bool bpf_opcode_in_insntable(u8 code) {
        static const bool public_insntable[256] = {
                [0 ... 255] = false,
                /* Now overwrite non-defaults ... */
                BPF_INSN_MAP(BPF_INSN_2_TBL, BPF_INSN_3_TBL),
        };
        return public_insntable[code];
}
```

Copyright © SUSE 2022

Instruction Check

```
#define BPF_INSN_MAP(INSN_2, INSN_3)    \
    INSN_3(ALU, ADD,  X),        \
    INSN_3(ALU, SUB,  X),        \
    INSN_3(ALU, AND,  X),        \
    INSN_3(ALU, OR,   X),        \
    ...
    INSN_2(JMP, CALL),       \
    INSN_2(JMP, EXIT),       \
    ...
```

Copyright © SUSE 2022

Instruction Check

```
static const bool public_insntable[256] = {
        [0 ... 255] = false,
        /* Now overwrite non-defaults ... */
        BPF_INSN_MAP(BPF_INSN_2_TBL, BPF_INSN_3_TBL),
};
```

Copyright © SUSE 2022

Instruction Check

```
static const bool public_insntable[256] = {
    [0 ... 255] = false,
    /* Now overwrite non-defaults ... */
    INSN_3(ALU, ADD,  X),
    INSN_3(ALU, SUB,  X),
    INSN_3(ALU, AND,  X),
    INSN_3(ALU, OR,   X),
    ...
    INSN_2(JMP, CALL),
    INSN_2(JMP, EXIT),
    ...
};
```

Copyright © SUSE 2022

Instruction Check

```
bool bpf_opcode_in_insntable(u8 code) {
        static const bool public_insntable[256] = {
                [0 ... 255] = false,
                /* Now overwrite non-defaults ... */
                BPF_INSN_MAP(BPF_INSN_2_TBL, BPF_INSN_3_TBL),
        };
        return public_insntable[code];
}
```

Copyright © SUSE 2022

Instruction Check

```
static const bool public_insntable[256] = {
    [0 ... 255] = false,
    /* Now overwrite non-defaults ... */
    INSN_3(ALU, ADD,  X),
    INSN_3(ALU, SUB,  X),
    INSN_3(ALU, AND,  X),
    INSN_3(ALU, OR,   X),
    ...
    INSN_2(JMP, CALL),
    INSN_2(JMP, EXIT),
    ...
};
```

Copyright © SUSE 2022

Instruction Check

```
static const bool public_insntable[256] = {
    [0 ... 255] = false,
    /* Now overwrite non-defaults ... */
    BPF_INSN_3_TBL(ALU, ADD,  X),
    BPF_INSN_3_TBL(ALU, SUB,  X),
    BPF_INSN_3_TBL(ALU, AND,  X),
    BPF_INSN_3_TBL(ALU, OR,   X),
    ...
    BPF_INSN_2_TBL(JMP, CALL),
    BPF_INSN_2_TBL(JMP, EXIT),
    ...
};
```

Copyright © SUSE 2022

Instruction Check

```
bool bpf_opcode_in_insntable(u8 code)
{
#define BPF_INSN_2_TBL(x, y)    [BPF_##x | BPF_##y] = true
#define BPF_INSN_3_TBL(x, y, z) [BPF_##x | BPF_##y | BPF_##z] = true
        static const bool public_insntable[256] = {
        ...
}
```

Copyright © SUSE 2022

Instruction Check

```
static const bool public_insntable[256] = {
    [0 ... 255] = false,
    /* Now overwrite non-defaults ... */
    [BPF_ALU | BPF_ADD | BPF_X] = true,
    [BPF_ALU | BPF_SUB | BPF_X] = true,
    [BPF_ALU | BPF_AND | BPF_X] = true,
    [BPF_ALU | BPF_OR  | BPF_X] = true,
    ...
    [BPF_JMP | BPF_CALL] = true,
    [BPF_JMP | BPF_EXIT] = true,
    ...
};
```

Copyright © SUSE 2022

Instruction Check

```
static const bool public_insntable[256] = {
    [0 ... 255] = false,
    /* Now overwrite non-defaults ... */
    [BPF_ALU | BPF_ADD | BPF_X] = true,
    [BPF_ALU | BPF_SUB | BPF_X] = true,
    [BPF_ALU | BPF_AND | BPF_X] = true,
    [BPF_ALU | BPF_OR  | BPF_X] = true,
    ...
    [BPF_JMP | BPF_CALL] = true,
    [BPF_JMP | BPF_EXIT] = true,
    ...
};
```

Lookup Table

Copyright © SUSE 2022

Instruction Check

```
bool bpf_opcode_in_insntable(u8 code)
{
        static const bool public_insntable[256] = {
                ...
        };
        return public_insntable[code]; /* Indexed by opcode */
}
```

Copyright © SUSE 2022

Example Programs

Copyright © SUSE 2022

Analyzing Program

Starting with the simplest program

```
SEC("sock_filter")
int bpf_prog(struct __sk_buff *skb)
{
    return 1;
}
```

Copyright © SUSE 2022

Analyzing Program

You'll get program with 2 instructions

```
0000000000000000 <bpf_prog1>:
    0: r0 = 1;      BPF_ALU64_IMM(BPF_MOV, BPF_REG_0, 1)
    1: exit;        BPF_EXIT_INSN()
```

Copyright © SUSE 2022

Analyzing Program

You'll get program with 2 instructions

```
0000000000000000 <bpf_prog1>:
    0: r0 = 1;      BPF_ALU64_IMM(BPF_MOV, BPF_REG_0, 1)
    1: exit;        BPF_EXIT_INSN()
```

Copyright © SUSE 2022

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

```
/* defined in
 * include/linux/filter.h
 */
BPF_ALU64_IMM(BPF_MOV, BPF_REG_0, 1)
```

Copyright © SUSE 2022

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

```
/* defined in
 * include/linux/filter.h
 */
BPF_ALU64_IMM(BPF_MOV, BPF_REG_0, 1)
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0, /* unused */
        .off     = 0, /* unused */
        .imm     = 1
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm     = 1
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm     = 1
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            ...
        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }

        env->insn_idx++;
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm     = 1
}
```

```
static int do_check(struct bpf_verifier_env *env) {
    for (;;) {
        struct bpf_insn *insn = &insns[env->insn_idx];
        u8 class = BPF_CLASS(insn->code);

        if (class == BPF_ALU || class == BPF_ALU64) {
            err = check_alu_op(env, insn);
            if (err)
                return err;

        } else if (class == BPF_LDX) {
            ...
        } else if (class == BPF_STX) {
            ...
        } else if (class == BPF_ST) {
            ...
        } else if (class == BPF_JMP || class == BPF_JMP32) {
            ...
        } else if (class == BPF_LD) {
            ...
        }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm     = 1
}
```

```
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn)
{
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }

    return 0;
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm     = 1
}
```

```
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn)
{
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }

    return 0;
}
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm     = 1
}
```

```
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn)
{
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }

    return 0;
}
```

Copyright © SUSE 2022

```c
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm     = 1
}
```

```c
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn) {
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
        if (BPF_SRC(insn->code) == BPF_X) {
            ...
        } else { /* BPF_SRC(insn->code) == BPF_K */
            ...
        }
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm     = 1
}
```

```
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn) {
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
        if (BPF_SRC(insn->code) == BPF_X) {
            ...
        } else { /* BPF_SRC(insn->code) == BPF_K */
            ...
        }
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code    = (BPF_ALU64
                    | BPF_MOV
                    | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm     = 1
}
```

```
static int check_alu_op(struct bpf_verifier_env *env,
                        struct bpf_insn *insn) {
    u8 op = BPF_OP(insn->code);

    if (op == BPF_END || op == BPF_NEG) {
        ...
    } else if (op == BPF_MOV) {
        ...
        if (BPF_SRC(insn->code) == BPF_X) {
            ...
        } else { /* BPF_SRC(insn->code) == BPF_K */
            ...
        }
    } else if (op > BPF_END) {
        ...
    } else {   /* all other ALU ops: and, sub,
                * xor, add, ... */
        ...
    }
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_ALU64
                     | BPF_MOV
                     | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm      = 1
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        ...
    } else { /* BPF_SRC(insn->code) == BPF_K */

        /* case: register = imm */
        mark_reg_unknown(env, regs, insn->dst_reg);
        regs[insn->dst_reg].type = SCALAR_VALUE;
        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            __mark_reg_known(regs + insn->dst_reg,
                             insn->imm);
        } else {
            __mark_reg_known(regs + insn->dst_reg,
                             (u32)insn->imm);
        }
    }
} else if (op > BPF_END) {
    ...
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_ALU64
                     | BPF_MOV
                     | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off      = 0,
        .imm      = 1
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        ...
    } else { /* BPF_SRC(insn->code) == BPF_K */

        /* case: register = imm */
        mark_reg_unknown(env, regs, insn->dst_reg);
        regs[insn->dst_reg].type = SCALAR_VALUE;
        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            __mark_reg_known(regs + insn->dst_reg,
                             insn->imm);
        } else {
            __mark_reg_known(regs + insn->dst_reg,
                             (u32)insn->imm);
        }
    }
} else if (op > BPF_END) {
    ...
```

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_ALU64
                     | BPF_MOV
                     | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off     = 0,
        .imm      = 1
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        ...
    } else { /* BPF_SRC(insn->code) == BPF_K */

        /* case: register = imm */
        mark_reg_unknown(env, regs, insn->dst_reg);
        regs[insn->dst_reg].type = SCALAR_VALUE;
        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            __mark_reg_known(regs + insn->dst_reg,
                             insn->imm);
        } else {
            __mark_reg_known(regs + insn->dst_reg,
                             (u32)insn->imm);
        }
    }
} else if (op > BPF_END) {
    ...
```

Copyright © SUSE 2022

Inferring Type & Value

Registers	Type
r0	NOT_INIT
r1	PTR_TO_CTX
r2	NOT_INIT
…	…
r9	NOT_INIT
r10 (fp)	PTR_TO_STACK

r0 = 1

→

`BPF_ALU64_IMM(BPF_MOV, BPF_REG_0, 1)`

Registers	Type
r0	SCALAR_VALUE
r1	PTR_TO_CTX
r2	NOT_INIT
…	…
r9	NOT_INIT
r10 (fp)	PTR_TO_STACK

Copyright © SUSE 2022

```
struct bpf_insn insn = {
        .code     = (BPF_ALU64
                     | BPF_MOV
                     | BPF_K),
        .dst_reg = BPF_REG_0,
        .src_reg = 0,
        .off      = 0,
        .imm      = 1
}
```

```
if (op == BPF_END || op == BPF_NEG) {
    ...
} else if (op == BPF_MOV) {
    ...
    if (BPF_SRC(insn->code) == BPF_X) {
        ...
    } else { /* BPF_SRC(insn->code) == BPF_K */

        /* case: register = imm */
        mark_reg_unknown(env, regs, insn->dst_reg);
        regs[insn->dst_reg].type = SCALAR_VALUE;
        if (BPF_CLASS(insn->code) == BPF_ALU64) {
            __mark_reg_known(regs + insn->dst_reg,
                             insn->imm);
        } else {
            __mark_reg_known(regs + insn->dst_reg,
                             (u32)insn->imm);
        }
    }
} else if (op > BPF_END) {
    ...
```

Copyright © SUSE 2022