

BPF: 16-byte values, exceptions, and Rust

Yonghong Song

Linux Plumbers Conference 2026

Agenda

1 $\leq$ 16-byte values across calls

by-value struct / `__int128` arguments; R0:R2 aggregate returns; arena pointers in returns; GCC vs. clang vs. rustc

2 Exception handling

Rust motivation; LLVM 23 cleanup / catch pads with `.bpf_cleanup` section; kernel runtime (`bpf_unwind()`) and verifier.

3 Future Work

≤16-byte values

By-value struct / `__int128` arguments and return values

Why 16-byte Value Is Useful in Kernel

- ▶ BPF calling convention: args in R1-R5, return in R0 — **one 8-byte register each.**
- ▶ Anything wider did not fit, so real code hit walls:

```
/* global func or kfunc taking a struct by value */
Arg#1 type STRUCT in tar() is not supported yet.
Unrecognized R2 type STRUCT
/* __int128 arg: compiler uses 2 regs, verifier counted 1 -> every later arg off by one */
/* returning an aggregate (clang < 23) */
error: aggregate returns are not supported
error: unable to allocate function return #1          (__int128 return)
```

- ▶ Rust makes this common: u128, small `#[repr(C)]` structs, tuples, `Option<T>/Result<T,E>` are all naturally passed and returned *by value*:

```
fn checked_div(a: u64, b: u64) -> Result<u64, i64> {    // 16 bytes
    if b == 0 {
        return Err(-22);                                // R0:R2 = (Err, -22)
    }
    Ok(a / b)                                            // R0:R2 = (Ok, a / b)
}

let q = checked_div(x, y)?;                            // caller: tag in R0, value in R2
```

≤16-byte Values Can Be Used for Arguments and Return Values

Arguments

- ▶ A value of ≤16 bytes takes 1 or 2 *consecutive* arg registers.
- ▶ `sub->args[]` is now indexed by **argument slot**, not BTF parameter (`arg_cnt` → `arg_slot_cnt`).
- ▶ Global funcs: all slots must fit in R1-R5 (no stack args).
- ▶ Members must be **scalars only** (aggregate or `__int128`).

Return values

- ▶ First 8 bytes in R0, second 8 bytes in R2.
- ▶ kfuncs and bpf-to-bpf subprogs (static and global).
- ▶ Main prog unchanged: its return is the exit code.
- ▶ Needs LLVM 23.

```
f(int a, struct pair p, int c)
```

R1=a	R2=p.lo	R3=p.hi	R4=c
------	---------	---------	------

```
struct pair g(...)
```

R0=lo	R2=hi
-------	-------

What LLVM 23 emits

```
struct pair { u64 lo, hi; };

u64 take_pair(u64 a, struct pair p,
              u64 c)
{ return a + p.lo + p.hi + c; }

struct pair make_pair(u64 a, u64 b)
{ return (struct pair){ a + b, a - b }; }

__int128 add128(__int128 x,
               __int128 y)
{ return x + y; }
```

clang 22: **error: aggregate returns are not supported**

```
take_pair:
    r0 = r2          ; p.lo
    r0 += r1         ; a
    r0 += r3         ; p.hi
    r0 += r4         ; c
    exit
make_pair:
    r0 = r2
    r0 += r1         ; lo in R0
    r1 -= r2
    r2 = r1          ; hi in R2
    exit
add128:
    ...              ; x=r1:r2 y=r3:r4
    ...              ; ret R0:R2
```

```
caller:
    call make_pair
    r3 = r2          ; p.hi
    r1 = 1           ; a
    r2 = r0          ; p.lo
    r4 = 2           ; c
    call take_pair
```

Argument calling convention: three compilers

argument	GCC	clang	rustc
<code>__int128</code>	<code>r1+r2</code>	<code>r1+r2</code>	<code>r1+r2</code>
<code>≤8B struct</code>	pointer in r1	<code>r1</code>	<code>r1</code>
<code>9–16B struct</code>	pointer in r1	by value <code>r1+r2</code>	by value <code>r1+r2</code>

- ▶ clang and rustc agree; the kernel follows them.
- ▶ GCC passes *any* aggregate by invisible reference $\Rightarrow$ the by-value struct selftests are clang-only.

Example: by-value struct argument (selftest)

Positive: scalars only

```
struct pair { __u64 lo; __u64 hi; };

__noinline __u64
take_pair_global(int a, struct pair p, int c)
{
    return (__u64)a + p.lo + p.hi + c;
}

SEC("tc")
__success __retval(0)
int aggregate_arg_global_struct_c_test(
    struct __sk_buff *skb)
{
    __u64 a = skb->len ^ MIX_A;
    __u64 b = skb->len ^ MIX_B;
    struct pair p = { .lo = a, .hi = b };

    if (take_pair_global(1, p, 2) != a + b + 3)
        return 1;
    return 0;
}
```

Negative: a pointer member

```
struct with_ptr {
    void *p;
    __u64 x;
};

__noinline __u64
take_with_ptr_global(struct with_ptr s)
{
    return s.x;
}

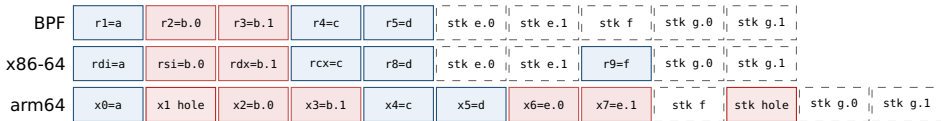
SEC("tc")
__failure __msg("type STRUCT in
take_with_ptr_global() is not
composed of scalars")
int aggregate_arg_ptr_member_fail(
    struct __sk_buff *skb)
{
    struct with_ptr s = {
        .p = skb, .x = skb->len };

    return take_with_ptr_global(s);
}
```

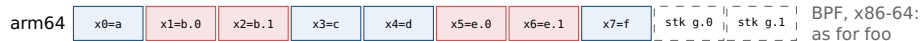
kfuncs: the JIT maps BPF slots onto the native ABI

BPF is *maximally dense*: pack every eightbyte, pad nothing, split at the register/stack boundary. The JIT moves args to the native ABI (`bpf_jit_plan_arg_moves()`).

```
foo(u64 a, __int128 b, u64 c, u64 d, __int128 e, u64 f, __int128 g)
```



```
bar(u64 a, struct pair b, u64 c, u64 d, struct pair e, u64 f, struct pair g)
```



- **x86-64**: e does not fit in r9, so all of e goes to the stack and f back-fills r9.
- **arm64**: an `__int128` takes an even register pair (x1 hole) and a 16-byte aligned stack slot (hole after f); struct pair { u64 lo, hi; } is 8-byte aligned, so no holes.

kfuncs: one ABI description per architecture

```
/* include/linux/filter.h */
struct bpf_jit_arg_abi {
    u8    nr_arg_regs;
    bool  even_reg_align;
    bool  even_stack_align;
    bool  split_at_boundary;
    bool  backfill_after_stack;
};
```

even reg / stk a 16-byte arg starts at an even register / stack slot

split an arg may straddle the last register and the stack

backfill a later arg may still take a free register

arch	regs	even reg	even stk	split	backfill
BPF	5	–	–	✓	n/a
x86-64	6	–	✓	–	✓
arm64	8	✓	✓	–	–
riscv64 LP64	8	–	✓	✓	n/a
ppc64le ELFv2	8	–	–	✓	n/a
grey: described, not wired up yet					

- ▶ An arch fills in a struct `bpf_jit_arg_abi` and its register table.
- ▶ From that description, the JIT places every argument correctly: one common implementation for all arches.
- ▶ Done for **x86-64 and arm64** (JIT + arm64 trampoline `save_args()`/`restore_args()`).

Commit 1e6c598b02e7 (“bpf: Prepare kfunc arguments for the JIT from an ABI description”).

Return value calling convention: three compilers

return	GCC	clang (LLVM 23)	rustc (LLVM 23)
__int128	r0+r1	r0+r2	r0+r2
≤8B struct	memory, sret ptr in r1	r0	r0
9–16B struct	memory, sret ptr in r1	r0+r2	r0+r2

- ▶ Kernel adopts **R0:R2**. clang and rustc get it from the LLVM 23 BPF backend.
- ▶ GCC returns aggregates through memory (hidden sret pointer) — works for static subprogs, but not a register-pair ABI a kfunc or global func can share.

R0:R2 returns: make_pair()

C

```
struct pair { __u64 lo, hi; };

static __noinline struct pair
make_pair(__u64 a, __u64 b)
{
    struct pair p = { .lo = a + b,
                      .hi = a - b };
    return p;
}
```

BPF (clang, LLVM 23, -O2)

```
make_pair:
    r0 = r2
    r0 += r1          ; R0 = lo = a + b
    r1 -= r2
    r2 = r1           ; R2 = hi = a - b
    exit

caller:
    call make_pair
    r0 ^= r2          ; reads both halves
    exit
```

- ▶ A 16-byte aggregate comes back in **R0:R2** — from clang and rustc alike (LLVM 23).
- ▶ Verifier: R2 is a second return register; a callee that leaves it unset is refused.
- ▶ kfuncs: the JIT maps R0:R2 onto the native pair (x86-64 RAX:RDX), on x86-64, arm64 and riscv64.

Arena pointers in by-value returns

```
/* bpf_testmod.c */
__bpf_kfunc struct prog_test_ret_arena
bpf_kfunc_call_test_ret_arena(u64 addr)
{
    struct prog_test_ret_arena r = {
        .a = (void *)addr,
        .b = (void *) (addr + 4) };
    return r;
}

/* BPF program */
r = bpf_kfunc_call_test_ret_arena((u64)page);
/* both halves back in R0:R2 */
a = (u32 __arena *)r.a;
b = (u32 __arena *)r.b;
*a = 1; *b = 2;
```

- ▶ "Scalars only" was stricter than needed.
- ▶ An **arena pointer** is safe as raw register bits — and can already be returned on its own.
- ▶ Now allowed as struct member(s) for global funcs and kfuncs (not for the main prog).
- ▶ Any other pointer member still rejected.

Summary

series	content
aggregate return values up to 16 bytes	R0:R2 for kfuncs + subprogs; verifier, precision, liveness, JIT (x86-64/arm64/riscv64)
arena pointers in by-value returns	arena-pointer members in returned structs; better diagnostics
by-value struct and <code>__int128</code> arguments	slot-indexed args for global funcs + kfuncs; x86-64 / arm64 JIT arg moves; arm64 trampoline fix

Exception handling

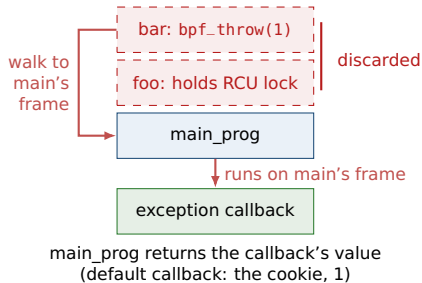
Rust Drop / LLVM/Kernel Exception Handling

bpf_throw() discards every frame up to the main program

```
static __noinline int bar(u32 len)
{
    if (len > 1500)
        bpf_throw(1);          /* cookie 1 */
    return 0;
}

static __noinline int foo(struct __sk_buff *skb)
{
    bpf_rcu_read_lock();
    bar(skb->len);              /* may throw */
    bpf_rcu_read_unlock();     /* skipped on a throw */
    return 0;
}

SEC("tc")
int main_prog(struct __sk_buff *skb)
{
    return foo(skb);
}
```



- ▶ Nothing in `foo` or `bar` runs again: the unlock is skipped.
- ▶ So the verifier refuses it, as it does for a held spin lock, reference, preempt or IRQ state:
bpf_throw cannot be used inside
bpf_rcu_read_lock-ed region

Releasing resources on a panic

- ▶ **Rust panics are everywhere** (bounds checks, `unwrap()`, `overflow`), and resources are released by **Drop**. What runs on a panic depends on `-C panic=?`, plus `catch_unwind`:

panic	on a panic	landing pads rustc emits	kernel
<code>panic=abort</code>	stop; no Drop runs	none	leak: refused
<code>panic=unwind</code>	each frame runs its Drop	cleanup pads (end in resume)	this series
<code>panic=unwind + catch_unwind</code>	Drops run, then the catching frame goes on	a catch pad (catch ptr null, ends in ret)	not yet

- ▶ With `panic=unwind`, rustc *already* emits the cleanup code (invoke / landingpad cleanup / resume); nothing new is needed from the frontend.
- ▶ With `panic=unwind + catch_unwind`, rustc also emits a catch pad for the call it wraps (landingpad catch ptr null, then a normal return): the unwind stops there.
- ▶ LLVM 23 writes the pads down in `.bpf_cleanup`; the kernel runs them on the way out, and the verifier follows every unwind to the pad it reaches.

Rust source: a guard released by Drop

```
impl Drop for PreemptGuard {
    fn drop(&mut self) {
        unsafe { bpf_preempt_enable() };
        pad_ran(RAN_F003_PREEMPT);
    }
}

#[inline(never)]
#[unsafe(no_mangle)]
pub extern "C-unwind" fn foo3(val: u64) -> u64 {
    let _preempt = PreemptGuard::disable();
    if val > 100 {
        panic!("foo3: value out of range");
    }
    black_box(val) ^ 1
}

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    unsafe { bpf_unwind() }
}
```

- ▶ `_preempt` is live across the panic! call
⇒ rustc emits `invoke` with an unwind edge to a cleanup pad that runs `drop()`.
- ▶ Build needs:
 - `-C panic=unwind`
 - `panic-strategy=unwind` in target JSON
 - `"C-unwind"` on every frame

LLVM 23: invoke, cleanup landing pad, .bpf_cleanup

IR (test cleanup-section.ll: a BTreeMap insert that may fail)

```
define void @stat_inc(ptr %map, ptr %key, i64 %val)
    personality ptr @rust_personality {
        %node = call ptr @bpf_alloc(i64 256, i64 0)
        ...
    insert:
        invoke void @btree_insert(ptr %map, ptr %node,
                                   i64 %val)
            to label %done unwind label %cleanup
    done:
        ret void

    cleanup:                                ; cleanup landing pad
        %lp = landingpad { ptr, i32 } cleanup
        call void @bpf_free(ptr %node)      ; Drop
        resume { ptr, i32 } %lp
    }
```

BPF asm

```
.Ltmp0:                                # begin
    call btree_insert
.Ltmp1:                                # end
    exit
.Ltmp2:                                # landing pad
    r1 = r7
    call bpf_free
    call _Unwind_Resume

.section .bpf_cleanup,"a",@progbits
.long .Ltmp0
.long .Ltmp1
.long .Ltmp2
```

- ▶ One {begin, end, pad} per invoke region; R_BPF_64_NODYLD32 relocs.
- ▶ resume → call _Unwind_Resume.

LLVM commit 9d51c891b719 (llvm-project#192164), by Alexei Starovoitov.

LLVM 23: catch pad

IR (test cleanup-section.ll, modelled on catch_unwind)

```
define i64 @with_catch_unwind(ptr %map, i64 %val)
    personality ptr @rust_personality {
    %node = call ptr @bpf_alloc(i64 128, i64 0)
    invoke void @btree_insert(ptr %map, ptr %node,
                              i64 %val)
        to label %ok unwind label %caught
ok:
    ret i64 0
caught:
    ; catch pad
    %lp = landingpad { ptr, i32 }
        catch ptr null      ; catch-all
    call void @bpf_free(ptr %node)
    ret i64 1                ; no resume
}
```

BPF asm

```
.Ltmp9:                                # begin
    call btree_insert
.Ltmp10:                               # end
    r0 = 0
    exit
.Ltmp11:                               # catch pad
    call bpf_free
    r0 = 1
    exit                                # not _Unwind_Resume

.section .bpf_cleanup,"a",@progbits
.long .Ltmp9
.long .Ltmp10
.long .Ltmp11
```

- ▶ catch ptr null (catch-all; typed catches are a compile error) and a normal return: the unwind **stops** in this frame. Same {begin, end, pad} record as a cleanup pad.
- ▶ Rust's catch_unwind lowers to exactly this; the kernel refuses it for now (an exit in a pad).

Kernel: the table and two kfuncs

UAPI

```
struct bpf_cleanup_info {
    __u32 begin_off;      /* insn index */
    __u32 end_off;
    __u32 landing_pad_off;
};
/* BPF_PROG_LOAD */
__aligned_u64 cleanup_info;
__u32 cleanup_info_rec_size;
__u32 cleanup_info_cnt;
```

kfuncs

```
void bpf_unwind(void);
void bpf_unwind_resume(void *ptr__ign);
```

C selftests write the table in inline asm

```
    "call bpf_rcu_read_lock;"
"1:" "call foo3;"           /* cleanup region */
"2:"
    ... normal path, ends in bpf_rcu_read_unlock ...
"6:"                        /* landing pad */
    "call bpf_rcu_read_unlock;"
    "call bpf_unwind_resume;"
    CLEANUP_REC("1b", "2b", "6b")
```

- ▶ `bpf_unwind()` is new; `bpf_throw()` keeps its meaning (discard the frames).
- ▶ `libbpf`: `_Unwind_Resume` → `bpf_unwind_resume`; also through the light skeleton and the static linker.

The design: Alexei's suggestion (v5 review)

imo the following is cleaner:

- do not repurpose `bpf_throw()` for this new thing. Introduce new kfunc that will do the unwind, let's name it `bpf_unwind()`?*
- replace all `call bpf_unwind_resume` with NOP by the verifier (or may with `bpf_exit`. tbd)*
- In `bpf_unwind()` walk all exception tables and replace return addresses in corresponding frames to `landing_pad_ip`-s.*
- just return from `bpf_unwind()`. restoring callee saved registers will happen automatically by corresponding frames and there is no need to search exception tables at each step. [...] For Rust unwinds we don't need all that.*

Alexei Starovoitov, v5 review: lore.kernel.org/bpf/DLN0K7231LF2.1UNIBGA546C90@gmail.com

Runtime example: the program

main $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C. A holds an RCU lock across its call to B; only that call is covered (record [1, 2) $\rightarrow$ P). C raises the unwind.

```
main:  r6 = r1          ; before
       call A
       r0 = 0          ; after: normal return
       exit

A:     call bpf_rcu_read_lock
       r6 = 1          ; before
1:     call B           ; covered: [1, 2)  $\rightarrow$  P
2:     r6 += 1          ; after: normal return
       call bpf_rcu_read_unlock
       exit

P:     call bpf_rcu_read_unlock ; landing pad
       call bpf_unwind_resume
```

```
B:     r7 = r1          ; before
       call C           ; no record
       r0 = r7          ; after: normal return
       exit

C:     r8 = 2           ; before
       if r1 > 100 goto 9f
       r0 = 0
       exit
9:     call bpf_unwind   ; no record
       r0 = r8          ; after
       exit
```

Runtime example: fill return address

main: ...

call A

r0 = 0; exit (*skipped*)

main's epilogue

A: call bpf_rcu_read_lock

1: call B

2: r6 += 1; ... (*skipped*)

P: call bpf_rcu_read_unlock

r0 = 0; exit (*resume*)

B: ...

call C

r0 = r7; exit (*skipped*)

B's epilogue

C: ...

call bpf_unwind

r0 = 0; exit (*patched in*)

3

A's slot: return address:
main's epilogue

2

B's slot: return address:
P, A's pad

1

C's slot: return address:
B's epilogue

0 bpf_unwind() in C only writes slots (saved return addresses):

1 B's epilogue → C's slot

2 **P** → B's slot

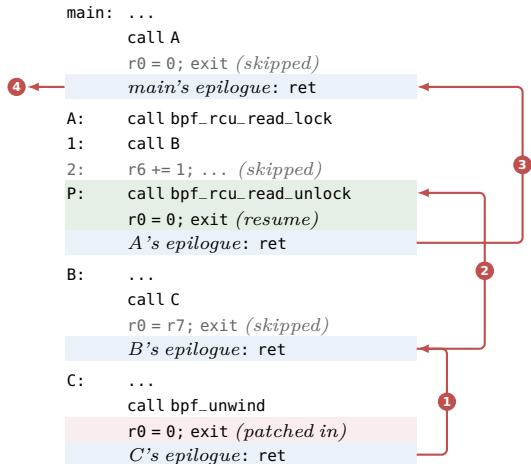
3 main's epilogue → A's slot

then returns into C.

Nothing has run yet: each frame's return is only redirected, an epilogue's address (or a pad's) written into its callee's slot. The next slide follows the returns.

Epilogue: what a frame runs to return; it puts back its caller's r6-r9.

Runtime example: bpf_unwind()



Verifier: Compatible between Runtime and Verifier

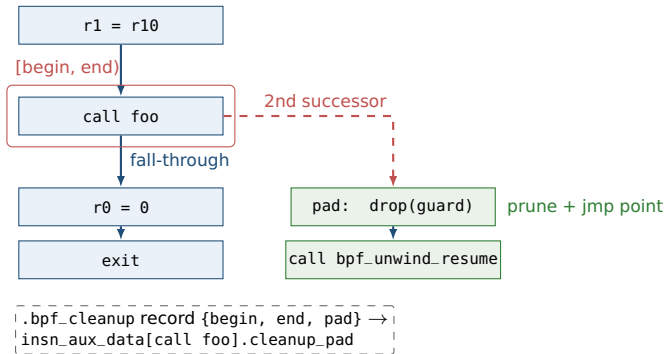
step	runtime	verifier
C unwinds	-	no record over the call: <code>unwind_frames()</code>
C	<code>r0 = 0; exit</code>	frame 3 popped
B	<code>exit</code>	frame 2 popped, nothing of B runs again
A	<code>pad runs; exit</code>	continue at P, then popped
main	<code>r0 = 0; exit</code>	<code>process_bpf_exit_full()</code> , <code>r0 = 0</code>

► After `bpf_unwind()`, both sides follow the same rules.

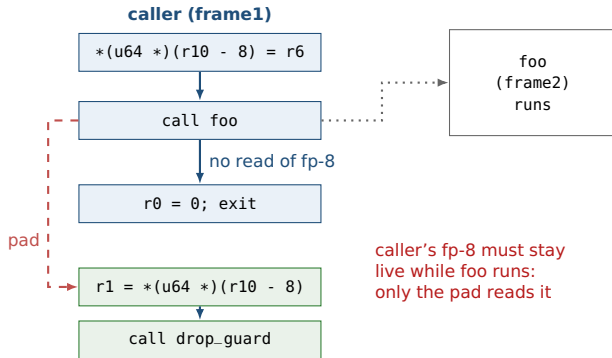
Verifier: Coverage

- ▶ **CFG**: covered call → the edge with landing pad
- ▶ **Liveness**: slots only a pad reads
- ▶ **unwind_frames()**: pop like the runtime
- ▶ **Global calls**: two successors, one normal return and another unwind
- ▶ **Precision**: backtrack from a pad into the frames the unwind left
- ▶ **Refusals**: what cannot be dispatched, e.g.
 - an exit in a pad (no catch pads)
 - a 2nd unwind while one is in flight
 - `bpf_throw()` alongside an unwind
 - `fexit` / `fmod_ret` / `fsession` on a subprog that can unwind
 - resources not cleaned up right before the main program returns

Verifier: CFG

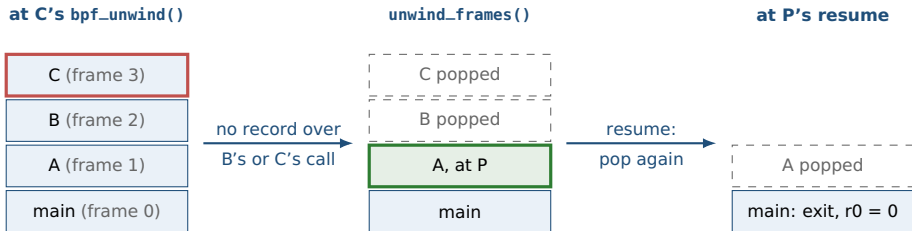


Verifier: liveness



Verifier: `unwind_frames()` pops like the runtime

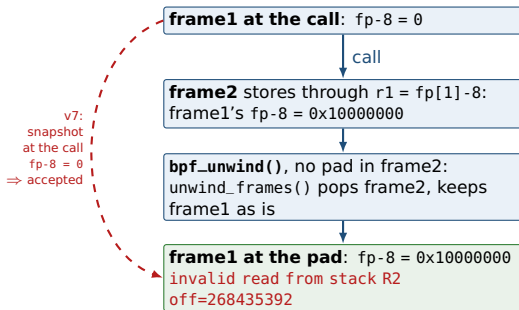
Same example, verified: the frames the verifier holds at three points.



Verifier: a pad starts from the state the callee left

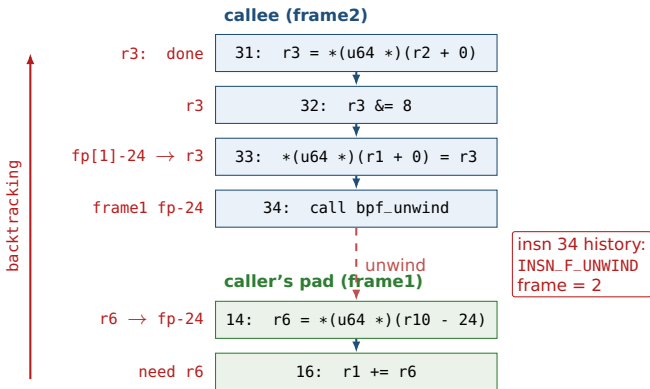
```
caller:                /* frame1 */
    *(u64 *) (r10 - 8) = 0
    r1 = r10
    r1 += -8
1:  call callee          /* covered */
    ...
3:  r1 = *(u64 *) (r10 - 8) /* pad */
    r2 = r10
    r2 += -64
    r2 += r1
    r0 = *(u8 *) (r2 + 0)

callee:                /* frame2 */
    r2 = 0x100000000
    *(u64 *) (r1 + 0) = r2
    call bpf_unwind
```



The callee's epilogue restores only r6–r9 and r10. Dynptr / iterator slots it reinitialised and packet pointers it invalidated reach the pad the same way. A **global** subprog is not walked: its call gets a second successor, the unwind, taken from the state the call returns in.

Verifier: precision

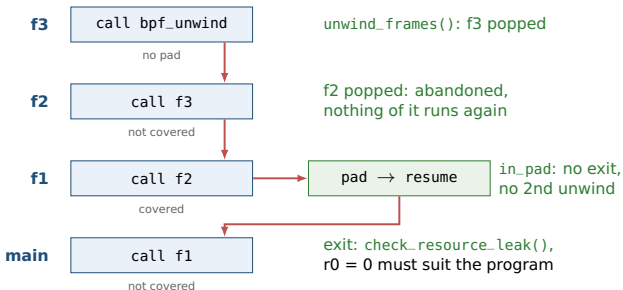


- ▶ Red: what backtracking needs precise, walking up from the pad.
- ▶ On entering a landing pad, a jump-history entry is recorded for backtracking.

```
/* backtrack_insn() */  
} else if (hist &&  
           (hist->flags & INSN_F_UNWIND)) {  
    bt_clear_reg(bt, BPF_REG_0);  
    bt->frame = hist->frame;  
    return 0;  
}
```

A global call's pad stays in its frame:
pad_of_call(idx) == subseq_idx.

Verifier: unwinding paths



Locks, RCU, preempt, IRQ and references are the *program's* state, not a frame's: the unwind carries them through every pad to main's exit.

- ▶ A pad may drop what its caller took: a Rust move, `consume(rec)`, gives the callee's pad `rec` to drop.
- ▶ Dropping twice (the callee's pad, then the caller's) is refused like any double release: the first drop invalidates the pointer.

Future Work

≤16-byte ABI

- ▶ GCC: converge on the clang/rustc calling convention (by-value args, **R0:R2** returns).
- ▶ Other arches (s390x / ppc64 etc.): support **R0:R2** returns for kfuncs.
- ▶ Other arches (riscv64 / ppc64 / s390x etc.): support 2-register (16-byte) kfunc arguments, via `bpf_jit_arg_abi`.
- ▶ BPF trampoline: support 16-byte by-value args on x86-64 (arm64 already does, `save_args()`/`restore_args()`) — in my to-do list.

Exceptions

- ▶ Support catch pads? When to use a catch pad, and when a cleanup pad?
- ▶ `bpf_throw()` and `bpf_unwind()` cannot be used in one program today. Too limited? `bpf_throw()` has use cases Rust does not express (e.g. assertions)?
- ▶ Private stack is disabled with cleanup (and catch?) pads: x86-64 is short of registers; arches with enough registers seem fine.
- ▶ More cases to cover: trampolines, others?