

THE BPF TARGET IN LLVM



ABOUT ME

- Michal Rostecki
- **vadorovsky** @ Github, Discord, Mastodon etc.
- Software Engineer @ **Light Protocol** (ZK on Solana)
- **Aya** maintainer (BPF library for Rust)
- Interests: Linux kernel, Rust, ZK cryptography

BPF

- "Berkeley Packet Filter"
- ...but that's not what it is, really.
- It's a virtual machine for running minimal, sandboxed programs.
- Restricted environment (instruction limit, verifier).
- Rules: execute fast, don't sleep*, don't allocate too much memory.

USE CASES

- **Linux kernel** - BPF originates from there
 - Network filtering
 - Tracing and profiling
 - Security policies
 - Extending scheduler
 - HID-BPF

USE CASES

- Solana
 - Smart contracts
- **IoT / microcontrollers**

STORING DATA

- Stack memory is limited (512 bytes).
- Linux kernel BPF
 - BPF maps (stored in memory)
- Solana BPF
 - Accounts (stored in memory of validator nodes)

PROJECTS

- **Cilium** - Container Network Interface for Kubernetes.
- **Falco** - Security alert engine.
- Pretty much every Solana project. Orca, Mango, Audius, Metaplex, Light Protocol etc.
- **Good stuff from Deepfence coming soon!** Security, but with focus on enforcement.

BPF AND LLVM

- First and main compiler providing BPF support.
- BPF has a backend in LLVM.
- Supported by Clang (C) and Rust.

ENABLING BPF BACKEND

```
-DLLVM_TARGETS_TO_BUILD="host;BPF"
```

e.g.

```
CC=clang CXX=clang++ cmake -DCMAKE_BUILD_TYPE=Debug \  
  -DLLVM_PARALLEL_LINK_JOBS=1 -DLLVM_BUILD_LLVM_DYLIB=1 \  
  -DLLVM_ENABLE_LLD=1 -DLLVM_ENABLE_PROJECTS="clang" \  
  -DLLVM_TARGETS_TO_BUILD="host;BPF"
```

LINKING

- BPF target is **not** supported by lld.
- Instead, BPF has its own linkers:
 - **bpftool-link** - created by libbpf community
 - **bpf-linker** - for Rust, created by Aya community
- Only static linking.

LLD

- There is a patch for lld to support BPF target, but it's not accepted.
 - <https://reviews.llvm.org/D101336>

REGISTERS

- **R0** - return value from functions, exit values for programs.
- **R1 - R5** - function arguments.
- **R6 - R9** - callee saved registers that function calls will preserve.
- **R10** - read-only frame pointer to access stack.

R0 - R5: scratch registers, programs need to spill them if necessary across calls.

BASIC INSTRUCTION ENCODING

32 bits
(MSB)

16
bits

4 bits

4 bits

8 bits
(LSB)

Integer
Immediate
Value

Offset

Source
register

Destination
register

Opcode

WIDE INSTRUCTION ENCODING

64 bits (MSB)

64 bits (LSB)

basic instruction

pseudo instruction

ARITHMETIC INSTRUCTIONS

Code	Description
BPF_ADD	dst += src
BPF_SUB	dst -= src
BPF_MUL	dst *= src
BPF_DIV	dst = (src != 0) ? (dst / src) : 0
BPF_OR	dst
BPF_AND	dst &= src

ARITHMETIC INSTRUCTIONS

Code	Description
BPF_LSH	$\text{dst} \ll= \text{src}$
BPF_RSH	$\text{dst} \gg= \text{src}$
BPF_NEG	$\text{dst} = \sim \text{src}$
BPF_MOD	$\text{dst} = (\text{src} \neq 0) ? (\text{dst} \% \text{src}) : \text{dst}$
BPF_XOR	$\text{dst} \wedge= \text{src}$

ARITHMETIC INSTRUCTIONS

Code	Description
BPF_MOV	dst = src
BPF_ARSH	sign extending shift right
BPF_END	byte swap operations

BYTE SWAP INSTRUCTIONS

Code	Description
BPF_TO_LE	host byte order -> little endian
BPF_TO_BE	host byte order -> big endian

JUMP INSTRUCTIONS

Code	Description
BPF_JA	PC += off
BPF_JEQ	PC += off if dst == src
BPF_JGT	PC += off if dst > src
BPF_JGE	PC += off if dst >= src

JUMP INSTRUCTIONS

Code	Description
BPF_JSET	PC += off if dst & src
BPF_JNE	PC += off if dst != src
BPF_JSGT	PC += off if dst > src
BPF_JSGE	PC += off if dst >= src

JUMP INSTRUCTIONS

Code	Description
BPF_CALL	function call
BPF_EXIT	program return

JUMP INSTRUCTIONS

Code	Description
BPF_JLT	PC += off if dst < src
BPF_JLE	PC += off if dst <= src
BPF_JSLT	PC += off if dst < src
BPF_JSLE	PC += off if dst <= src

ATOMIC OPERATIONS

Code	Description
BPF_ADD	atomic add
BPF_OR	atomic or
BPF_AND	atomic and
BPF_XOR	atomic xor

ATOMIC OPERATIONS

These atomic operations work only on 64 and 32 bit types.

Atomic CAS is **not** supported.

EXAMPLE (C CODE)

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int xdp_prog_simple(struct xdp_md *ctx)
{
    return XDP_PASS;
}

char _license[] SEC("license") = "GPL";
```

BUILDING (CLANG)

```
clang -O2 -target bpf -c -g xdp_pass_kern.c \  
-o xdp_pass_kern.o
```

LLVM-READELF

```
$ llvm-readelf -S xdp_pass_kern.o
```

Section Headers:

[Nr]	Name	Type	Address	Off	S
[0]		NULL	0000000000000000	000000	
[1]	.strtab	STRTAB	0000000000000000	000b5a	
[2]	.text	PROGBITS	0000000000000000	000040	
[3]	xdp	PROGBITS	0000000000000000	000040	
[4]	license	PROGBITS	0000000000000000	000050	
[5]	.debug_abbrev	PROGBITS	0000000000000000	000054	
[6]	.debug_info	PROGBITS	0000000000000000	000111	
[7]	.rel.debug_info	REL	0000000000000000	0008b8	
[8]	.debug_str_offsets	PROGBITS	0000000000000000	0001de	0
[9]	.rel.debug_str_offsets	REL	0000000000000000	0008f8	0
[10]	.debug_str	PROGBITS	0000000000000000	000246	
[11]	.debug_addr	PROGBITS	0000000000000000	0003cd	
[12]	.debug_line	PROGBITS	0000000000000000	000070	

LLVM-OBJDUMP

```
$ llvm-objdump -S xdp_pass_kern.o
```

```
xdp_pass_kern.o:          file format elf64-bpf
```

```
Disassembly of section xdp:
```

```
0000000000000000 <xdp_prog_simple>:
```

```
;          return XDP_PASS;
```

```
0:          b7 00 00 00 02 00 00 00 00 r0 = 0x2
```

```
1:          95 00 00 00 00 00 00 00 00 exit
```

EMITTING IR

```
clang -O2 -target bpf -c -g xdp.c -S -emit-llvm -o xdp.ll
```

IR

```
define dso_local i32 @xdp_prog_simple(ptr nocapture readnone
    %ctx) #0 section "xdp" !dbg !24 {
entry:
    call void @llvm.dbg.value(metadata ptr poison, metadata !40,
        metadata !DIExpression()), !dbg !41
    ret i32 2, !dbg !42
}
```

EXAMPLE (RUST CODE)

```
#![no_std]
#![no_main]

use aya_bpf::{bindings::xdp_action, cty::c_long, macros::xdp, procs::xdp};

#[xdp(name = "xdp")]
pub fn xdp(ctx: XdpContext) -> u32 {
    match try_xdp_hello(ctx) {
        Ok(ret) => ret,
        Err(_) => xdp_action::XDP_ABORTED,
    }
}

fn try_xdp_hello(_ctx: XdpContext) -> Result<u32, c_long> {
    Ok(xdp_action::XDP_PASS)
}
```

BUILDING (RUST)

```
cargo +nightly build --target=bpfel-unknown-none \  
-Z build-std=core
```


LLVM-READELF

```
$ llvm-readelf -S target/bpfel-unknown-none/debug/xdp
There are 5 section headers, starting at offset 0x220:
```

Section Headers:

[Nr]	Name	Type	Address	Off	S
[0]		NULL	0000000000000000	000000	
[1]	.strtab	STRTAB	0000000000000000	0001c0	
[2]	.text	PROGBITS	0000000000000000	000040	
[3]	xdp/xdp	PROGBITS	0000000000000000	0000d8	
[4]	.symtab	SYMTAB	0000000000000000	0000e8	

LLVM-OBJDUMP

```
$ llvm-objdump -S target/bpfel-unknown-none/debug/xdp-hello  
[...]  
Disassembly of section xdp/xdp:
```

```
0000000000000000 <xdp>:  
      0:      b7 00 00 00 02 00 00 00 00 r0 = 0x2  
      1:      95 00 00 00 00 00 00 00 00 exi
```

EMITTING IR

```
cargo +nightly rustc --target=bpfel-unknown-none \  
-Z build-std=core -- --emit=llvm-ir
```

IR

```
define dso_local noundef i32 @xdp(ptr nocapture noundef  
    readnone %ctx) unnamed_addr #0 section "xdp/xdp" {  
start:  
    ret i32 2  
}
```

BPF TYPE FORMAT (BTF)

- Debug info format for BPF, way more lightweight than DWARF.
- Used for offsets across Linux kernel versions.
- Used for stack traces in BPF verifier.
- But there is no debugger (yet).

EXAMPLE: C CODE

```
struct foo {  
    __u32 a;  
    __u64 b;  
};
```

EXAMPLE: LLVM DEBUG INFO

```
!49 = distinct !DICompositeType(tag: DW_TAG_structure_type,  
    name: "foo", file: !3, line: 21, size: 128, elements: !50)  
!50 = !{!51, !54}  
!51 = !DIDerivedType(tag: DW_TAG_member, name: "a", scope: !49,  
    file: !3, line: 22, baseType: !52, size: 32)  
!52 = !DIDerivedType(tag: DW_TAG_typedef, name: "__u32",  
    file: !53, line: 27, baseType: !12)  
!54 = !DIDerivedType(tag: DW_TAG_member, name: "b", scope: !49,  
    file: !3, line: 23, baseType: !55, size: 64, offset: 64)  
!55 = !DIDerivedType(tag: DW_TAG_typedef, name: "__u64",  
    file: !53, line: 31, baseType: !56)
```

EXAMPLE: BTF

```
[8] STRUCT 'foo' size=16 vlen=2  
    'a' type_id=9 bits_offset=0  
    'b' type_id=10 bits_offset=64
```


EXAMPLE: RUST CODE

```
pub struct Foo {  
    a: u32,  
    b: u64,  
}
```

EXAMPLE: LLVM DEBUG INFO

```
!42 = !DIBasicType(name: "u32", size: 32,  
    encoding: DW_ATE_unsigned)  
!60 = !DICompositeType(tag: DW_TAG_structure_type, name: "Foo",  
    scope: !2, file: !5, size: 128, align: 64, elements: !61,  
    templateParams: !65,  
    identifier: "63dcf8d9f7a7a7ed6f05eaed70c4b12f")  
!61 = !{!62, !63}  
!62 = !DIDerivedType(tag: DW_TAG_member, name: "a", scope: !60,  
    file: !5, baseType: !42, size: 32, align: 32, offset: 64)  
!63 = !DIDerivedType(tag: DW_TAG_member, name: "b", scope: !60,  
    file: !5, baseType: !64, size: 64, align: 64)  
!64 = !DIBasicType(name: "u64", size: 64,  
    encoding: DW_ATE_unsigned)
```

EXAMPLE: BTF

```
[20] STRUCT 'Foo' size=16 vlen=2  
      'a' type_id=14 bits_offset=64  
      'b' type_id=21 bits_offset=0
```

LOCAL BTF

Each modern Linux kernel comes with BTF info:

```
# bpftool btf dump file /sys/kernel/btf/vmlinux
[...]
[335] STRUCT 'pid' size=112 vlen=8
      'count' type_id=330 bits_offset=0
      'level' type_id=59  bits_offset=32
      'lock'  type_id=324 bits_offset=64
      'tasks' type_id=336 bits_offset=128
      'inodes' type_id=134 bits_offset=384
      'wait_pidfd' type_id=328 bits_offset=448
      'rcu' type_id=139 bits_offset=640
      'numbers' type_id=337 bits_offset=768
```

BTF RELOCATIONS

- BPF programs are adjusted to read type fields at the offset specified in **local** BTF info.
- Regardless of the memory layout of the type.
- Types with BTF-based access are annotated with `llvm.preserve.*.access.index` intrinsics.

CHALLENGES WITH RUST

- BPF support introduced later than in Clang.
- BTF emission not supported, but close to be done!
- BTF relocations not supported.

WHAT'S THE PROBLEM?

- Kernel expects specific BTF layout.
- It's very C-specific.
 - BPF maps definitions have to be anonymous structs (which Rust doesn't support).
 - Complex Rust types (e.g. data carrying enums) are not supported.

SOLUTIONS

- Temporary: modify DI in bpf-linker.
- Long-term: # [btf_export] macro in Rust.

BPF-LINKER

Currently working PoC. Transforms DI to meet kernel expectations:

- Removes names from pointer types and BTF map structs.
- Tweaks the DI of Rust-specific types to be C-compatible.

BTF (FROM RUST) AFTER MODIFICATIONS

```
[10] STRUCT '(anon)' size=40 vlen=5
      'type' type_id=1 bits_offset=0
      'key' type_id=5 bits_offset=64
      'value' type_id=5 bits_offset=128
      'max_entries' type_id=6 bits_offset=192
      'map_flags' type_id=8 bits_offset=256
```

```
[6] PTR '(anon)' type_id=7
```

DEBUG INFO INCLUDED

```
; let parent_pid: i32 = unsafe {  
    ctx.read_at(PARENT_PID_OFFSET)? };  
9: 63 1a f4 ff 00 00 00 00 *(u32 *) (r10 - 0xc) = r1  
[...]  
; let child_pid: i32 = unsafe {  
    ctx.read_at(CHILD_PID_OFFSET)? };  
19: 63 1a f8 ff 00 00 00 00 *(u32 *) (r10 - 0x8) = r1  
20: bf a2 00 00 00 00 00 00 r2 = r10
```

LLVM CHANGES

- Already merged, but to be released in LLVM 17:
 - `LLVMGetDINodeTag` function to get the tag of DI Node.
 - `LLVMReplaceMDNodeOperandWith` function to modify DI.

IF YOU WANT TO TRY IT OUT

- github.com/vadorovsky/aya-btf-map - structs and macros for BTF maps.
- github.com/vadorovsky/aya-btf-maps-experiments - example project using it.
- Requires LLVM and bpf-linker patches.

#[BTF_EXPORT]

- Decoupled from `-C debuginfo`.
- Generates DI, which produces correct BTF for annotated types.
- Raises a compiler error when used on BTF-incompatible type.

VERIFIER

- (So far) only in Linux kernel (Solana, rBPF and other user space implementations don't have it).
- Ensuring safe memory access - kinda, like, making C Rusty (◡‿◡).
- Descending all possible instruction paths, observing the change of registers and stack.

REGISTER STATE TYPES

- NOT_INIT - was never written to.
- SCALAR_VALUE - value not usable as a pointer.
- PTR_TO_CTX - program context.
- CONST_PTR_TO_MAP - pointer to BPF map.
- PTR_TO_MAP_VALUE - pointer to BPF map value.
- PTR_TO_MAP_VALUE_OR_NULL - pointer to BPF map value or NULL.
- PTR_TO_STACK - frame pointer.

REGISTER STATE TYPES

- PTR_TO_PACKET - pointer to network packet data.
- PTR_TO_PACKET_END - pointer to end of network packet data.
- PTR_TO_SOCKET - pointer to bpf_sock_ops.
- PTR_TO_SOCKET_OR_NULL - pointer to bpf_sock_ops or NULL.

WHAT DOESN'T PASS VERIFIER

- Reading from registers which were never written to.
- Instructions which are unreachable.
- Loops without bounds (to a constant).
- Memory access without bounds (to a constant).
- Direct packet access outside the end of packet.

DEMO

Let's write a simple LSM program which restricts filesystem access!

THANK YOU

- aya-rs.dev
 - github.com/aya-rs/aya
 - [Discord](#)
- lightprotocol.com

