Flying the nest - a BPF port of Doom

LPC 2024

Arpad Kiss (hello at akissxyz point com)

September 18, 2024

A beginner's perspective

- My first introduction to BPF.
- Undergraduate dissertation project.
- Goal: make BPF do strange things for fun.
  - Explore the boundaries of what BPF can do.
  - Report on the state of the ecosystem.

The long view

Where is BPF going?

- Growing use cases
- Larger, more complex programs
- Both in kernel & userspace

The 3 virtues of BPF

Performance

- JIT compilation (x64, ARM, RISC)
- Native calling convention
- No context switch

The 3 virtues of BPF

Performance

- JIT compilation (x64, ARM, RISC)
- Native calling convention
- No context switch

Portability

- CPU architectures
- Kernel versions
- Platforms (BPF on Windows)[1]

---

eBPF for Windows. Microsoft. https://github.com/microsoft/ebpf-for-windows

The 3 virtues of BPF

Performance

- JIT compilation (x64, ARM, RISC)
- Native calling convention
- No context switch

Portability

- CPU architectures
- Kernel versions
- Platforms (BPF on Windows)[1]

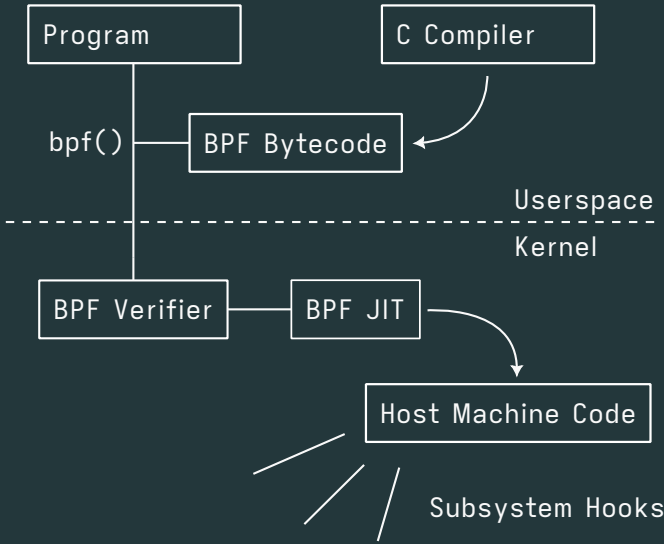
Proven safety

- Memory safety
- Bounded termination
- Division by zero

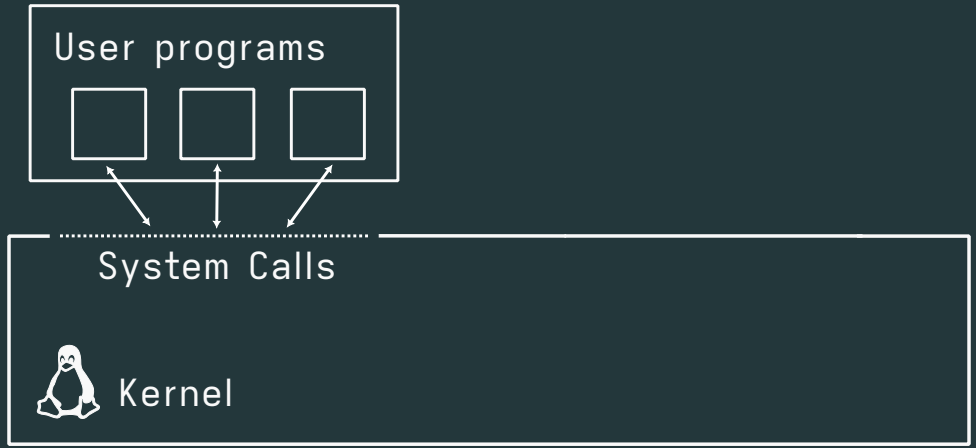
---

eBPF for Windows. Microsoft. https://github.com/microsoft/ebpf-for-windows

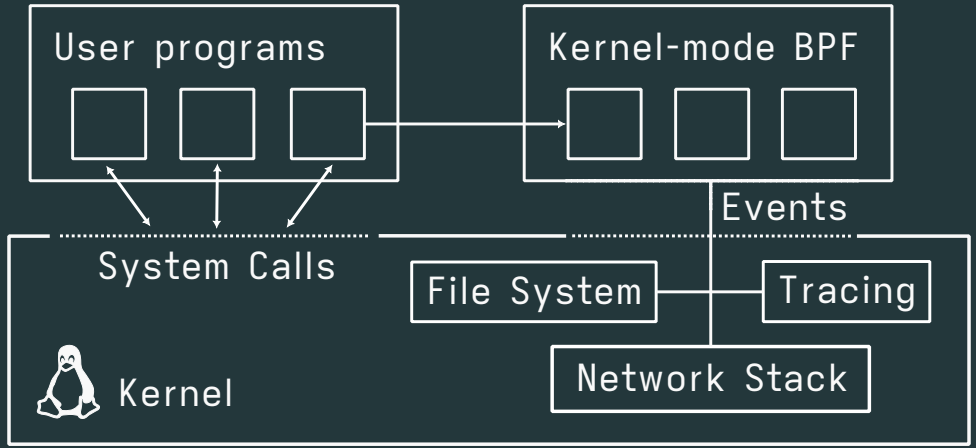
BPF in the kernel



BPF in the kernel



BPF in the kernel



Growing use cases

Cloudflare: ***Cloudflare architecture and how BPF eats the world*.** Marek Majkowski. Blogpost published 18th of May 2019. Accessed on the 6th of October 2023. Available:

https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/. Cloudflare

Growing use cases

Meta: ***BPF at Facebook.*** A talk given at Kernel Recipes 2019. Slide 8 states approx. 40 BPF programs on each server instance. Slides available at https://kernel-recipes.org/en/2019/talks/bpf-at-facebook/. Talk available at https://www.youtube.com/watch?v=bbHFg9IsTk8. Alexei Starovoitov, Kernel Recipes

Growing use cases

Google: ***Stories from BPF security auditing at Google – Brendan Jackman, Google.*** A talk given at the eBPF Summit 2021. Accessed 22nd of November 2023. https://www.youtube.com/watch?v=N4YKcMV8iaY. eBPF & Cilium Community, Google

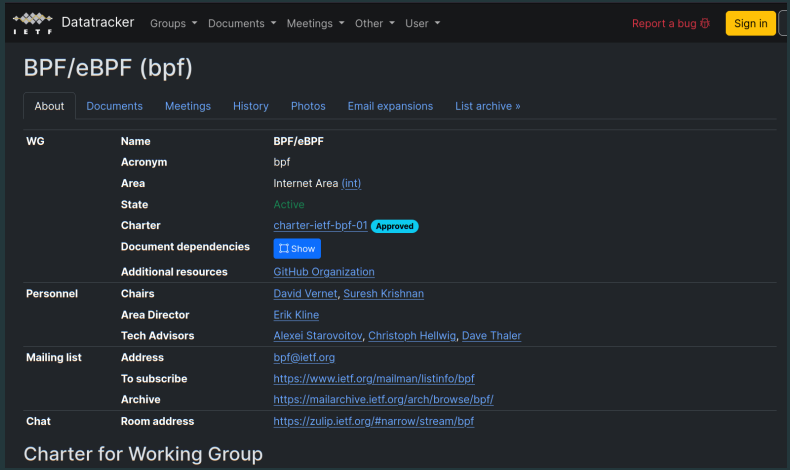
Growing use cases

Netflix: ***Netflix talks about Extended BPF - A new software type.*** A talk given the Ubuntu Masters Conference. At 8:18, mentions 14 BPF programs running on every Netflix instance. Published 28th of November 2019. Accessed 22nd of November 2023. Available: https://www.youtube.com/watch?v=7pmXdG8-7WU. Brendan Gregg, Canonical, Netflix

Growing use cases

Amazon: ***eBPF in Microservices Observability.*** A talk by Jaana Dogan at CloudNativeCon. Accessed: 22nd of November 2023. Available: https://www.youtube.com/watch?v=SKLA6n3TKL0. Amazon AWS

BPF IETF working group[2] adoption milestone



[2] ***Standards working group for BPF.*** https://datatracker.ietf.org/wg/bpf/about/. IETF.

BPF as a software platform?

Evaluate BPF as a platform for general software.

Technical goal: Attack BPF limitations

Recursion

```
Y = λf.(λx.f (x x))
        (λx.f (x x))
```

Limited C, no libc

Floating point

Unbounded Loops

```
while (1) {
  //  ..
}
```

No debuggers

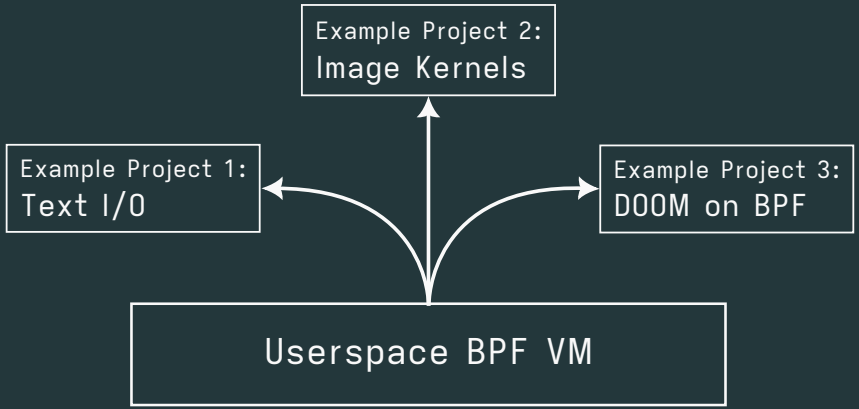
No signed division

```
(-2 /  3) = ?
```

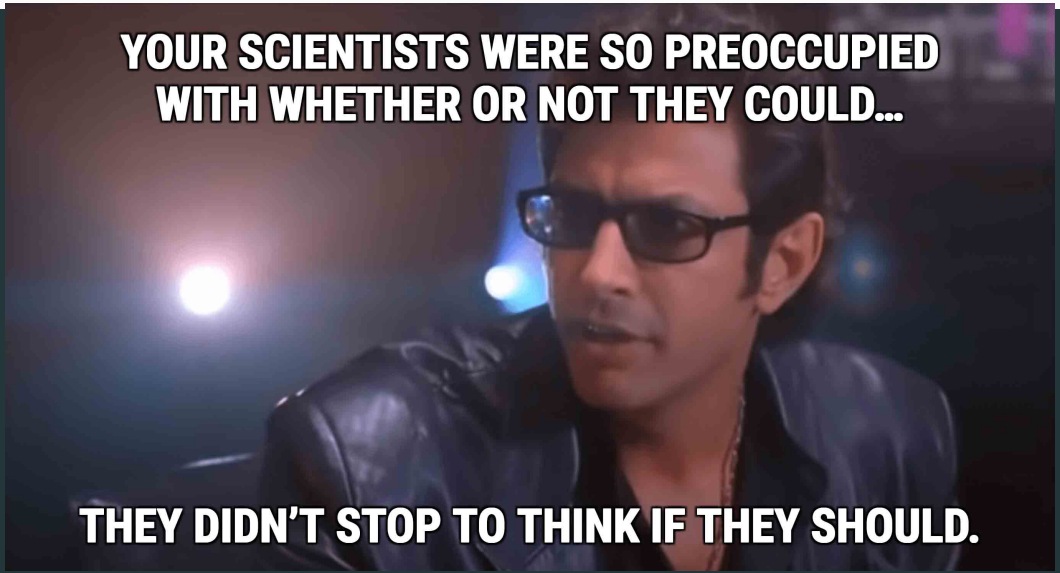
Agenda

Userspace BPF VM

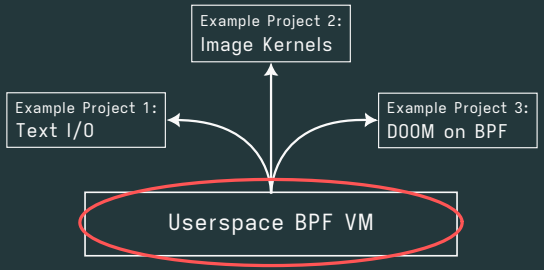
Agenda


Example Project 2:
Image Kernels
Example Project 1:
Text I/O
Example Project 3:
DOOM on BPF
Userspace BPF VM

Don’t try this at home.



BPF Virtual Machine



Custom BPF VM

- Why make a VM?

Custom BPF VM

- Why make a VM?
  - Enable custom features (intrinsics, debugger, Python interface)

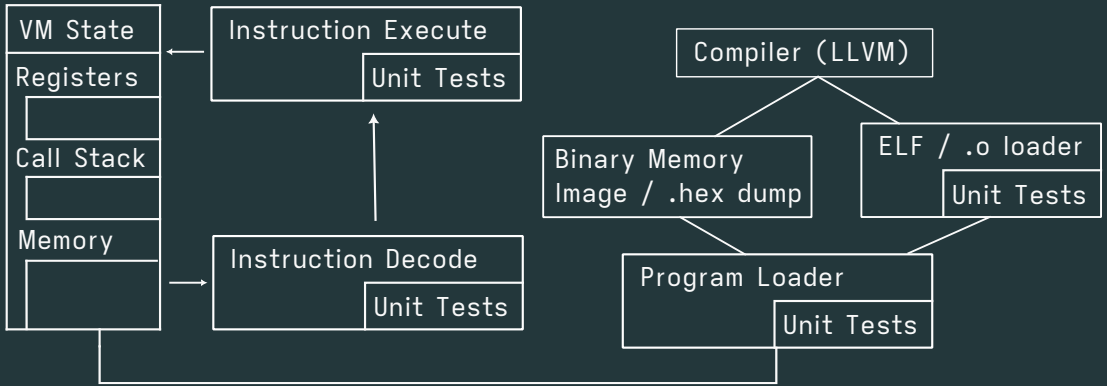
Custom BPF VM

- Why make a VM?
  - Enable custom features (intrinsics, debugger, Python interface)
  - Available options replicate the kernel environment

Custom BPF VM

- Why make a VM?
  - Enable custom features (intrinsics, debugger, Python interface)
  - Available options replicate the kernel environment
  - Don’t offer enough technical sophistication to run Doom

VM internal organisation

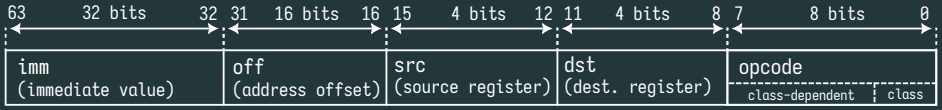


What does BPF code look like?

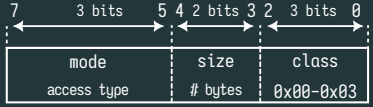
```c
// example.c: A very simple C program.

int times_three(int x) {
  return x * 3;
}

int begin(void* ctx, int x) {
  return 3 + times_three(x + 1);
}
```

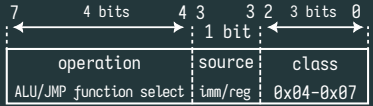
```
diss.o:    file format elf64-bpf

Disassembly of section .text:

0000000000000000 <times_three>:
    0:       bf 10 00 00 00 00 00 00 r0 = r1
    1:       27 00 00 00 03 00 00 00 r0 *= 0x3
    2:       95 00 00 00 00 00 00 00 exit

0000000000000018 <begin>:
    3:       bf 21 00 00 00 00 00 00 r1 = r2
    4:       07 01 00 00 01 00 00 00 r1 += 0x1
    5:       85 10 00 00 ff ff ff ff call -0x1
    6:       07 00 00 00 03 00 00 00 r0 += 0x3
    7:       95 00 00 00 00 00 00 00 exit
```

The BPF instruction encoding



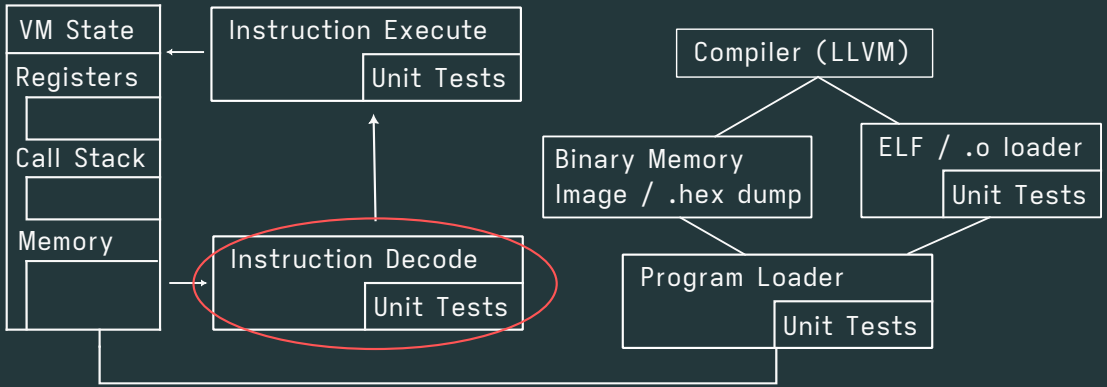
Opcode Class		Value
BPF_LD	non-standard load	0x00
BPF_LDX	register load	0x01
BPF_ST	store from imm	0x02
BPF_STX	store from reg	0x03
BPF_ALU	32-bit maths	0x04
BPF_JMP	64-bit jump	0x05
BPF_JMP32	32-bit jump	0x06
BPF_ALU64	64-bit maths	0x07





Let’s parse an instruction.

First stop: decode




```
r1 *= 3 | 0x00'00'00'03'00'00'01'27
```

The life of an instruction

```
// BPF assembly:
r1 *= 3
```

The life of an instruction

```
// BPF assembly:
r1 *= 3

// Corresponds to hex:
0x00'00'00'03'00'00'01'27
```

The life of an instruction: instruction format

```
# Instruction format:
|---------32b immediate -------| |--16b offset--| |--| |--| |opcode|
                                        source reg ^    ^ dest reg

# For 'r1 *= 3':
|---------32b immediate -------| |--16b offset--| |--| |--| |opcode|
00000000000000000000000000000011 0000000000000000 0000 0001 00100111
                                        Source Reg ^    ^ Dest Reg
```

`r1 *= 3`	`0x00'00'00'03'00'00'01'27`

The life of an instruction: understanding the opcode

```
#                              |opcode|
#                              00100111
                     Insn class ALU64  ^^^

#                        |------ opcode ------|
#                        0010       0       111
                multiply ^^^^                ^^^ ALU64
                                Source ^ (0 = immediate, 1 = register)
#                     =>  ALU64 multiply by immediate
```

```
r1 *= 3 | 0x00'00'00'03'00'00'01'27
```

The life of an instruction: putting the decode all together

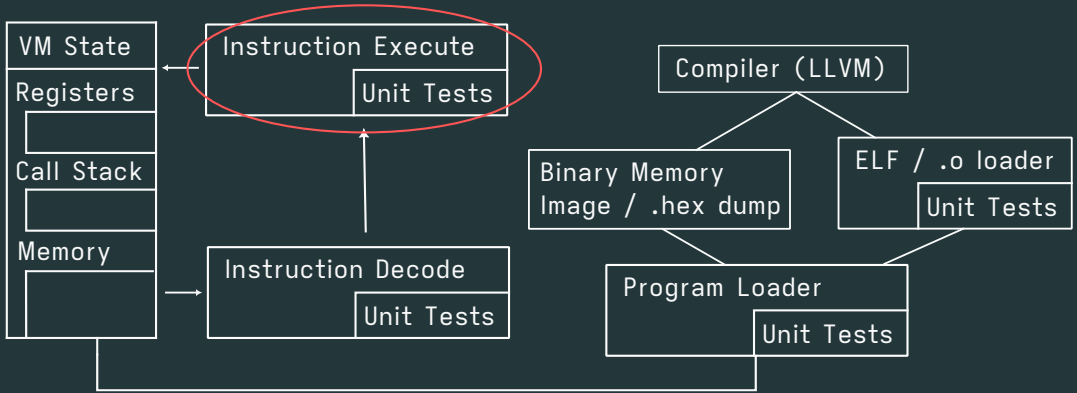
```
#                                   r1 *= 3

                          0x00'00'00'03'00'00'01'27

 |---------32b immediate -------| |--16b offset--| |--| |--| |opcode|
 00000000000000000000000000000011 0000000000000000 0000 0001 00100111

#       imm=3 | off=0 | src=0 | dst=1 | op = multiply by immediate

#                                   r1 *= 3
```

Example instruction decode unit test

```
bpf_insn i;
bpf_insn_from_u64(0x00'00'00'03'00'00'01'27 , &i);

REQUIRE(i.dst_reg == BPF_REG_1);
REQUIRE(i.imm == 3);
REQUIRE(i.opcode == (BPF_ALU64 | BPF_K | BPF_MUL));
```

```
r1 *= 3 | 0x00'00'00'03'00'00'01'27
```

The life of an instruction: Execute




```
r1 *= 3 | 0x00'00'00'03'00'00'01'27
```

Let’s look again at the code we’re executing

```c
// example.c: A very simple C program.

int times_three(int x) {
  return x * 3;
}

int begin(void* ctx, int x) {
  return 3 + times_three(x + 1);
}
```

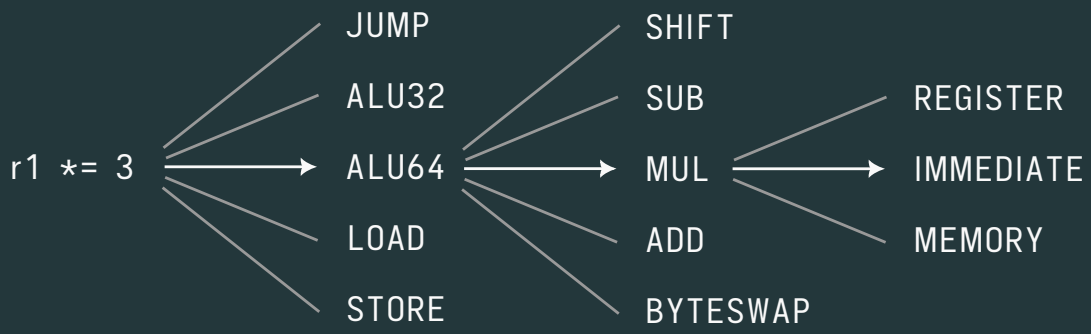
```
diss.o:    file format elf64-bpf

Disassembly of section .text:

0000000000000000 <times_three>:
    0:       bf 10 00 00 00 00 00 00 r0 = r1
    1:       27 00 00 00 03 00 00 00 r0 *= 0x3
    2:       95 00 00 00 00 00 00 00 exit

0000000000000018 <begin>:
    3:       bf 21 00 00 00 00 00 00 r1 = r2
    4:       07 01 00 00 01 00 00 00 r1 += 0x1
    5:       85 10 00 00 ff ff ff ff call -0x1
    6:       07 00 00 00 03 00 00 00 r0 += 0x3
    7:       95 00 00 00 00 00 00 00 exit
```

All* the instructions!

ALU code field	Value	Meaning
BPF_ADD	0x00	dst += src
BPF_SUB	0x10	dst -= src
BPF_MUL	0x20	dst *= src
BPF_DIV	0x30	dst /= src
BPF_OR	0x40	dst |= src
BPF_AND	0x50	dst &= src
BPF_LSH	0x60	dst <<= src
BPF_RSH	0x70	dst >>= src
BPF_NEG	0x80	dst = ~src
BPF_MOD	0x90	dst = dst mod src
BPF_XOR	0xa0	dst ^= src
BPF_MOV	0xb0	dst = src
BPF_ARSH	0xc0	sign extending shift right
BPF_END	0xd0	byte swap ops

JMP code field	Value	Meaning
BPF_JA	0x00	PC += off (jump)
BPF_JEQ	0x10	jump if dst == src
BPF_JGT	0x20	jump if dst > src
BPF_JGE	0x30	jump if dst ≥ src
BPF_JSET	0x40	jump if dst & src
BPF_JNE	0x50	jump if dst ≠ src
BPF_JSGT	0x60	jump if dst > src
BPF_JSGE	0x70	jump if dst ≥ src
BPF_CALL	0x80	function call
BPF_EXIT	0x90	function return program return
BPF_JLT	0xa0	jump if dst < src
BPF_JLE	0xb0	jump if dst ≤ src
BPF_JSLT	0xc0	jump if dst < src
BPF_JSLE	0xd0	jump if dst ≤ src

A naive interpreter




```
r1 *= 3 | 0x00'00'00'03'00'00'01'27
```

Finally, running code!

```
#include "bpf_vm.h"
#include "bpf_loading.h"
#include "bpf_exec.h"

SCENARIO("Test that example.c produces 15 when passed x=3.", "[example.c]")
{
    FILE* f = fopen("example.o", "rb");

    bpf_program prog = bpf_program_load_from_obj(f);
    bpf_vm vm = bpf_vm_init();

    // Set the argument x to 3.
    bpf_vm_reg_write(vm, BPF_REG_2, 3);

    bpf_execute_program(prog, vm);

    REQUIRE(bpf_vm_reg(vm, BPF_REG_RETURN) == 15);
}
```

Did you catch that?

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

   Disassembly of section .text:

0000000000000000 <times_three>:
    0:       bf 10 00 00 00 00 00 00 r0 = r1
    1:       27 00 00 00 03 00 00 00 r0 *= 0x3
    2:       95 00 00 00 00 00 00 00 exit

0000000000000018 <begin>:
    3:       bf 21 00 00 00 00 00 00 r1 = r2
    4:       07 01 00 00 01 00 00 00 r1 += 0x1
    5:       85 10 00 00 ff ff ff ff call -0x1
    6:       07 00 00 00 03 00 00 00 r0 += 0x3
    7:       95 00 00 00 00 00 00 00 exit
```

ELF Relocations

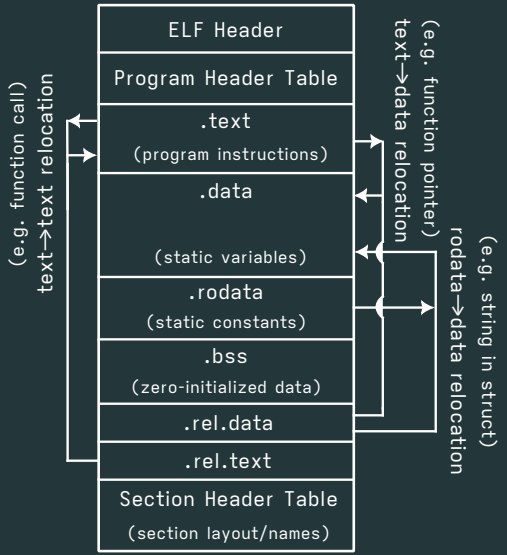
“There are only two hard things in Computer Science: relocations & naming things (and off-by-1 errors).”

ELF Relocations

Relocations are ‘todo notes’ left by the compiler.

Loaders/linkers resolve these by filling in the correct run-time addresses.

Internal layout of an ELF file



Common C constructs & the relocations they generate

Relocation	Example	Code Example
`text -> text`	Function calls	`sleep();`
`text -> data`	Reference to global	`printf("hi");`
`data -> text`	Global function pointer	`static void (*fp)(int) = a_func;`
`data -> data`	Pointers between data	`static int c = 5; int* c_ptr = &c;`

The 6 standard BPF relocation types.

ELF Relocation Type	Usage	Reloc. Addr.	Address Calc.
R_BPF_NONE	–	–	–
R_BPF_64_32	Call instructions	$R + 4$	$(S + A)/8 - 1$
R_BPF_64_64	Reloc for wide insn.	$R + 4$	$(S + A)$
R_BPF_ABS32	Plain data relocation	R	$(S + A)$
R_BPF_ABS64	Plain data relocation	R	$(S + A)$
R_BPF_NODYLD32	BTF-specific	R	$(S + A)$

Relocations are hard.

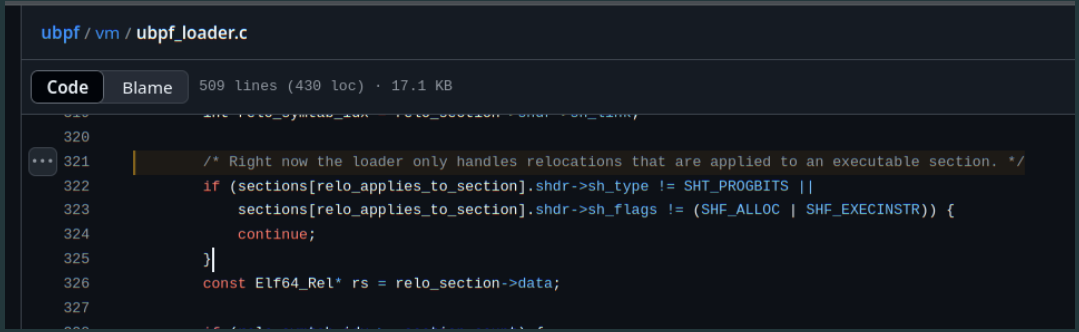
Relocations are not easy to implement & somewhat arcane.

Low hanging fruit: make better documentation for the BPF ELF format.

How does this compare?

	Kernel	Our VM	uBPF
text -> text	✔	✔	✔
text -> data	✔	✔	✔
data -> text	✔	✔	✖
data -> data	✔	✔	✖
CO-RE	✔	✖	✖

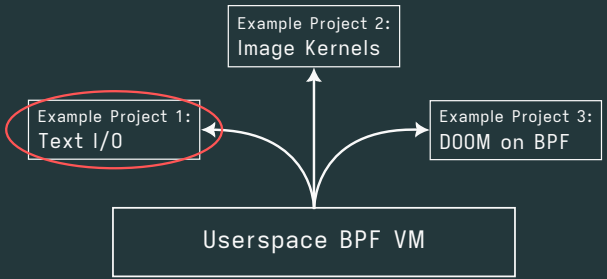
uBPF's ELF loader relocation support



Intrinsics based on relocations

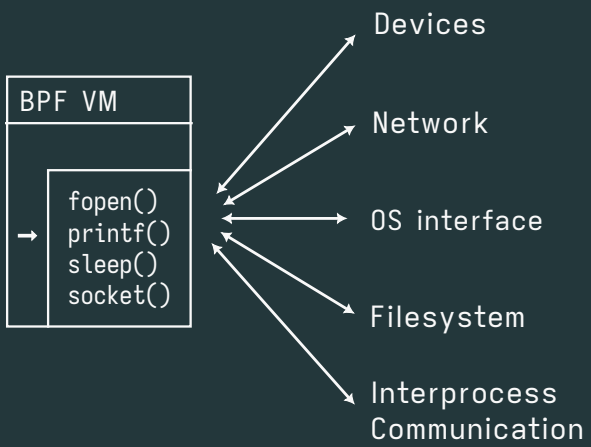
```
/// API function declared in C, but not defined.
/// This results in an undefined relocation in ELF.
char read();

/// Our VM reads these undefined relocations, checks if we've exposed
/// those symbols as intrinsics, and if so, replaces them.
void write(char c);
```

Example Project 1: Text I/O

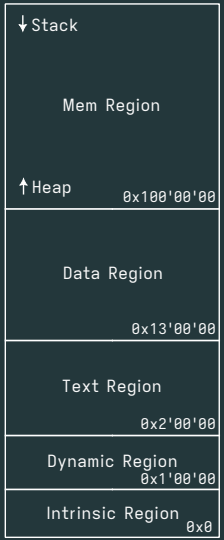


Uppercasing an input stream

Bridging the gap



VM design: Runtime memory organisation



To support complex, full-featured programs with state, we need to support:

- Memory allocation
- Global variables (.data, .rodata, .bss)
- Function pointers
- Intrinsics

Intrinsics in action

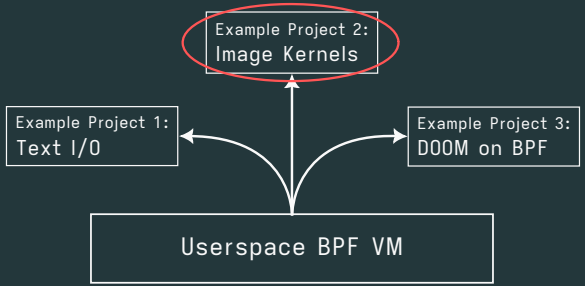
```
void read_intrin(bpf_vm vm) {
    char c = getc(stdin);
    // Place the value into R0
    // (return register in BPF)
    bpf_vm_reg_write(vm, BPF_REG_0,
    ↪  *(u64*)&c);
}

void write_intrin(bpf_vm vm) {
    auto v = bpf_vm_reg(vm, BPF_REG_1);
    putchar((char)v);
}
```

```
TEST_CASE("BPF uppercase", "[project-1]")
{
    // ...
    bpf_vm vm = bpf_vm_init();

    bpf_vm_add_intrinsic(vm,
        { "read", read_intrin });
    bpf_vm_add_intrinsic(vm,
        { "write", write_intrin });

    bpf_execute_program(prog, vm);
}
```

Example Project 1: Text Processing - Implementation

```
while (c ≠ -1) {
    bool is_lowercase_alpha = ('a' ⩽ c) && (c ⩽ 'z');
    bool is_uppercase_alpha = ('A' ⩽ c) && (c ⩽ 'Z');

    had_uppercase = had_uppercase || is_uppercase_alpha;

    if (is_lowercase_alpha && !had_uppercase) {
        c += 'A' - 'a';
    }

    write(c);
    c = read();
}
```

Example Project 2: Image Kernels



BPF's numeric limitations

Recursion

```
Y = λf.(λx.f (x x))
        (λx.f (x x))
```

Limited C, no libc

Floating point

Unbounded Loops

```
while (1) {
  //  ..
}
```

No debuggers

No signed division

```
(-2 /  3) = ?
```

Why image processing?

- Algorithms are numerical in nature
- Familiarity with topic
- Illustrates potential use case of BPF VM as an extension library

Image filter selection

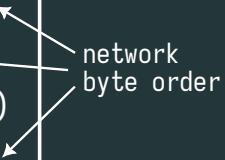
1. Greyscale filter
2. Contrast boost
3. Generic color space conversion (RGB→YUV)
4. Kernel convolution
    - 4.1 Blur
    - 4.2 Sharpen
    - 4.3 Sobel Edge Detection
    - 4.4 Unsharp Masking

Image pipeline using Farbfeld[3]

[3] ***Farbfeld.*** A lossless image format which is easy to parse, pipe and compress. https://tools.suckless.org/farbfeld/. The Suckless Community.

Farbfeld[4] in 1 slide

Offset	Description
0 - 7	magic 8 bytes
8 - 11	width (uint32)
12 - 15	height (uint32)
16 - ..	pixels (row-major)
	[RGBA] pixels 16 bit per color



[4] ***Farbfeld.*** A lossless image format which is easy to parse, pipe and compress. https://tools.suckless.org/farbfeld/. The Suckless Community.

Example Project 2: Image Processing - ‘Parsing’ Farbfeld

```
typedef struct ff_pix_s
{
    u16 r; u16 g; u16 b; u16 a;
} ff_pix;

typedef struct ff_image_s
{
    u64 ff_magic_value;
    u32 width;
    u32 height;
    ff_pix buf[];
} ff_image;
```

Example Project 2: Image Processing - Contrast boost

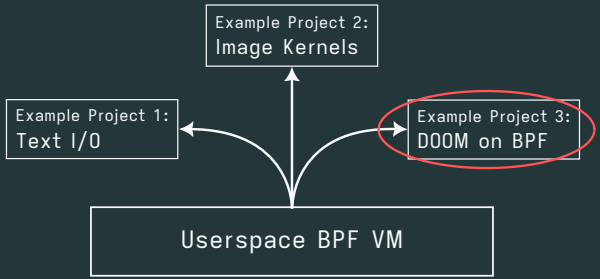
```
void begin(void* ctx, ff_image* img, u32 alpha, u16 beta)
{
    if (!ff_image_magic_value_ok(img->ff_magic_value))
        return;
    for (u32 y = 0; y < ntohl(img->height); y++) {
        for (u32 x = 0; x < ntohl(img->width); x++) {
            ff_pix* p = &img->buf[y * img_width + x];
            ff_pix_ntohs(p);

            p->r = ((p->r * alpha) >> frac_bits) + beta;
            p->g = ((p->g * alpha) >> frac_bits) + beta;
            p->b = ((p->b * alpha) >> frac_bits) + beta;

            ff_pix_htons(p);
        }
    }
}
```

Example Project 2: Image Processing - Greyscale

```
void begin(void* ctx, ff_image* img, u32 alpha, u16 beta)
{
    if (!ff_image_magic_value_ok(img->ff_magic_value))
        return;
    for (u32 y = 0; y < ntohl(img->height); y++) {
        for (u32 x = 0; x < ntohl(img->width); x++) {
            ff_pix* p = &img->buf[y * img_width + x];
            ff_pix_ntohs(p);

            u16 luma = ff_pix_luma(p);
            p->r = luma;
            p->g = luma;
            p->b = luma;

            ff_pix_htons(p);
        }
    }
}
```

Example Project 2: Image Processing - Demo

Example Project 3: DOOM



Example Project 3: DOOM

Example Project 3: DOOM - Motivation

Porting DOOM to new platforms is a long-standing movement.

Example Project 3: DOOM - Motivation

Practical hardware/platform analogue of ’Turing Completeness’

Example Project 3: DOOM - Technical Motivation

- Demonstrate the viability of porting large codebases to BPF.
- Highly interactive application: rendering, file I/O, user input.
- Shows massive level of technical sophistication in the VM.

https://github.com/ozkl/doomgeneric

README GPL-2.0 license

doomgeneric

The purpose of doomgeneric is to make porting Doom easier. Of course Doom is already portable but with doomgeneric it is possible with just a few functions.

To try it you will need a WAD file (game data). If you don't own the game, shareware version is freely available (doom1.wad).

porting

Create a file named doomgeneric_yourplatform.c and just implement these functions to suit your platform.

- DG_Init
- DG_DrawFrame
- DG_SleepMs
- DG_GetTicksMs
- DG_GetKey

Functions	Description
DG_Init	Initialize your platfrom (create window, framebuffer, etc...).
DG_DrawFrame	Frame is ready in DG_ScreenBuffer. Copy it to your platform's screen.
DG_SleepMs	Sleep in milliseconds.
DG_GetTicksMs	The ticks passed since launch in milliseconds.
DG_GetKey	Provide keyboard events.
DG_SetWindowTitle	Not required. This is for setting the window title as Doom sets this from WAD file.

Example Project 3: DOOM - Technical Challenges: Compilation

Even compiling DOOM to BPF is a challenge.

Example Project 3: DOOM - Technical Challenges: Compilation

LLVM crash on signed division.

```
Error at line 300: Unsupport signed division for DAG: 0x55e5513193f0: i64 = sdiv
↪  0x55e551318f20, Constant:i64<-2>, am_map.c:300:15
Please convert to unsigned div/mod.
fatal error: error in backend: Cannot select: 0x55e5513193f0: i64 = sdiv
↪  0x55e551318f20, Constant:i64<-2>, am_map.c:300:15
  0x55e551318f20: i64 = sra 0x55e55131af80, Constant:i64<32>, am_map.c:298:11
...
In function: AM_activateNewScale
PLEASE submit a bug report to https://github.com/llvm/llvm-project/issues/ and
↪  include the crash backtrace, preprocessed source, and associated run script.
Stack dump:
...
```

Example Project 3: DOOM - Technical Challenges: Compilation

Same bug, but the crash reporting breaks - reporting wrong location.

```
[Compiling d_main.c]
Error at line 0: Unsupport signed division for DAG: 0x5559088f6da0: i64 = sdiv
↪  0x55590891fae0, 0x555908942b0, d_main.c:0Please convert to unsigned div/mod.
fatal error: error in backend: Cannot select: 0x5559088f6da0: i64 = sdiv
↪  0x55590891fae0, 0x555908942b0, d_main.c:0
  0x55590891fae0: i64 = sra 0x555908942080, Constant:i64<32>, d_main.c:0
    0x555908942080: i64 = add 0x55590898d1f0, Constant:i64<4294967296>, d_main.c:0
...
```

Example Project 3: DOOM - Technical Challenges: Compilation

Even worse, signed division is required for pointer differences!

```
[Compiling g_game.c]
Error at line 1236: Unsupport signed division for DAG: 0x563f481af5e0: i64 = sdiv
↪  exact 0x563f481f1e10, Constant:i64<10>, g_game.c:1236:31Please convert to
↪  unsigned div/mod.
fatal error: error in backend: Cannot select: 0x563f481af5e0: i64 = sdiv exact
↪  0x563f481f1e10, Constant:i64<10>, g_game.c:1236:31
...
In function: G_DeathMatchSpawnPlayer
PLEASE submit a bug report to https://github.com/llvm/llvm-project/issues/ and
↪  include the crash backtrace, preprocessed source, and associated run script.
Stack dump:
...
```

Example Project 3: DOOM - Technical Challenges: Compilation

There is no BPF linker.

Example Project 3: DOOM - Technical Challenges: Compilation

Create a unity build for DOOM (single translation unit).

Example Project 3: DOOM - Technical Challenges: Compilation

LLVM crash on use of builtin memset.

```
[Compiling am_map.c]
am_map.c:836:5: error: A call to built-in function 'memset' is not supported.
  836 |     memset(fb, color, f_w*f_h);
      |     ^
...
```

Example Project 3: DOOM - Technical Challenges: Compilation

When replacing memset with a manual definition,

```
am_map.c:832:24: error: A call to built-in function 'memset' is not supported.
  832 |         ((int*)buf)[i] = c;
      |                        ^
```

Example Project 3: DOOM - Technical Challenges: Compilation

Not only that, memset is often inferred for regular assignments.

```
[Compiling r_plane.c]
In file included from r_plane.c:25:
In file included from ./i_system.h:23:
In file included from ./d_ticcmd.h:24:
r_plane.c:187:17: error: A call to built-in function 'memset' is not supported.
  187 |         ceilingclip[i] = -1;
```

Example Project 3: DOOM - Technical Challenges: Compilation

Pass by value not supported - structure is too large.

```
[Compiling am_map.c]
am_map.c:1326:2: error: pass by value not supported 0x557a1c4d3d80: i64 =
↪  GlobalAddress<ptr @AM_drawLineCharacter> 0, am_map.c:1326:2
 1326 |         AM_drawLineCharacter
      |         ^
am_map.c:1283:6: error: pass by value not supported 0x557a1c4d2ab0: i64 =
↪  GlobalAddress<ptr @AM_drawLineCharacter> 0, am_map.c:1283:6
 1283 |             AM_drawLineCharacter
      |             ^
am_map.c:1295:6: error: pass by value not supported 0x557a1c4d4790: i64 =
↪  GlobalAddress<ptr @AM_drawLineCharacter> 0, am_map.c:1295:6
 1295 |             AM_drawLineCharacter
      |             ^
```

Example Project 3: DOOM - Technical Challenges: Compilation

Too many arguments - LLVM is unable to spill to the stack in many cases.

```
[Compiling d_main.c]
d_main.c:323:9: error: too many args to 0x55b995bacc70: i64 = GlobalAddress<ptr
↪  @wipe_ScreenWipe> 0, d_main.c:323:9
  323 |         done = wipe_ScreenWipe(wipe_Melt
      |                ^
Error at line 0: Unsupport signed division for DAG: 0x55b995bdf000: i64 = sdiv
↪  0x55b995bad6f0, 0x55b995bf6fb0, d_main.c:0Please convert to unsigned div/mod.
fatal error: error in backend: Cannot select: 0x55b995bdf000: i64 = sdiv
↪  0x55b995bad6f0, 0x55b995bf6fb0, d_main.c:0
```

Example Project 3: DOOM - Technical Challenges: Compilation

Variadic arguments are not supported.

```
i_system.c:359:6: error: functions with VarArgs or StructRet are not supported
  359 | void I_Error (char *error, ...)
      |      ^
fatal error: error in backend: Cannot select: 0x55fe5c47a530: ch = vastart
↪  0x55fe5c47a450:1, FrameIndex:i64<2>, SrcValue:ch<0x55fe5c2db730>,
↪  i_system.c:387:5
  0x55fe5c305b60: i64 = FrameIndex<2>
In function: I_Error
PLEASE submit a bug report to https://github.com/llvm/llvm-project/issues/ and
↪  include the crash backtrace, preprocessed source, and associated run script.
Stack dump:
...
```

Example Project 3: DOOM - Technical Challenges: Runtime

Not enough time to discuss all resolved so far.

One significant challenge:

DOOM relies on a large part of the C standard library.

Example Project 3: DOOM - Technical Challenges: Runtime

The VM cannot simply link against a standard library - one main reason being that they could not share an address space.

So, we determined the exact list of symbols DOOM needs to compile, and implemented them as intrinsics.

Example Project 3: DOOM - Technical Challenges

In short: DOOM is hard.

Example Project 3: DOOM - Demo

”The best excuse is that you just f—ing did it.”

Jonathan Blow

Doom works!

1 : sh*Z   2 : emacsclient   3 : bash-   lapytopy

```
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 1
 * set_window_title DOOM Shareware
I_InitGraphics: DOOM screen size: w x h: 320 x 200
I_InitGraphics: Auto-scaling factor: 2
,
ft 1710253448s, 5.84709e-10fps
```

Doom works!

```
~~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 0
~~~~ Snoozing for ms = 1, us = 1000
Waiting on inet stuff... I_GetTime() = 1
 * set_window_title DOOM Shareware
I_InitGraphics: DOOM screen size: w x h: 320 x 200
I_InitGraphics: Auto-scaling factor: 2
f
ft 1710253448s, 5.84709e-10fps


 * Received frame: 0x7f841bd51028
OOOOO~~~~ Snoozing for ms = 1, us = 1000
ft 2s, 0.34904fps


 * Received frame: 0x7f841bd51028
~~~~ Snoozing for ms = 1, us = 1000
ft 2s, 0.486174fps


 * Received frame: 0x7f841bd51028
~~~~ Snoozing for ms = 1, us = 1000
ft 1s, 0.514933fps


 * Received frame: 0x7f841bd51028
OOOOOft 2s, 0.480769fps


 * Received frame: 0x7f841bd51028
OOOOOft 2s, 0.497018fps
```

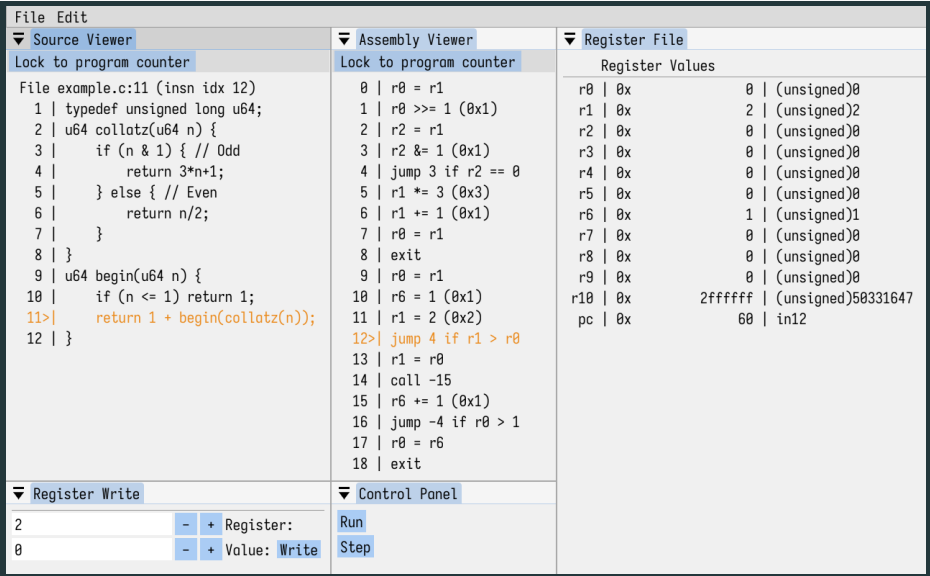
Userspace Debugger

Userspace Debugger

Step debugger for BPF built on our VM.

- Dear ImGui UI
- Source-level debugging using debug info
- BPF disassembler

Simple debugging



Python Interface

BCC is amazing.

iovisor's BCC is amazing.

Their Python front-end lowers the barrier to entry massively.

Our very own Python interface.

Idea: use our VM as a shared library.

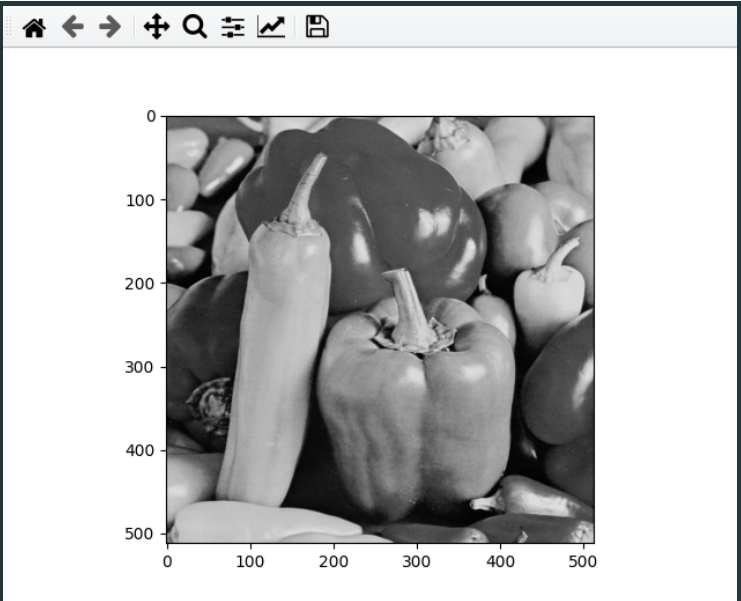
Expose the full VM API to Python.

Ease of use comparison - low-level Python interface

```python
from bcc import BPF

with open("input.bpf.c", "r") as f:
    prog = f.read()

# Create the BPF from the source
b = BPF(text=prog)
# Attach to a dummy event
ev = b.get_syscall_fnname("clone")
b.attach_kprobe(event=ev,
↪  fn_name="bpf_main")

# Trace one result
(_, _, _, _, _, msg) = b.trace_fields()

print(str(msg)[2:-1])
```

```python
# Low-level Python intf - directly exposed C API
from bpfvm import Bpf

bpf = Bpf(LIB_PATH)

with open(CODE_PATH, "rb") as code:
    prog = bpf.load_from_obj_fd(code.fileno())

# Create a BPF virtual machine.
vm = bpf.vm_init()

# Execute the program
bpf.execute_program(prog, vm)

# Print the contents of the output register
print(f'Result = {bpf.reg(vm, 0)}')
```

Doing real work with the Python interface - image processing



Intrinsics from Python & high-level Pythonic interface

```c
// C source for math.o
//  $ clang -target bpf -c math.c

// Intrinsics.
// These jump back out to Python!
int add(int x, int y);
int get();

int begin() {
    return add(2,3) * get();
}
```

```python
from bpf_vm import *

vm = BpfVm()

# Define intrinsics as *plain Python functions*!
def get(regs, vm):
    print(f"Intrinsic 'get' called at {vm.pc=}")
    return 3

vm.intrinsics.get = get
vm.intrinsics.sum = lambda reg: reg[1]+reg[2]

vm.execute_program("math.o")

print(f"{vm.regs=}")
```

Conclusions

Limitations overcome

✓✓ Recursion

```
Y = λf.(λx.f (x x))
        (λx.f (x x))
```

✓ C, partial libc

Floating point

✓✓ Unbounded Loops

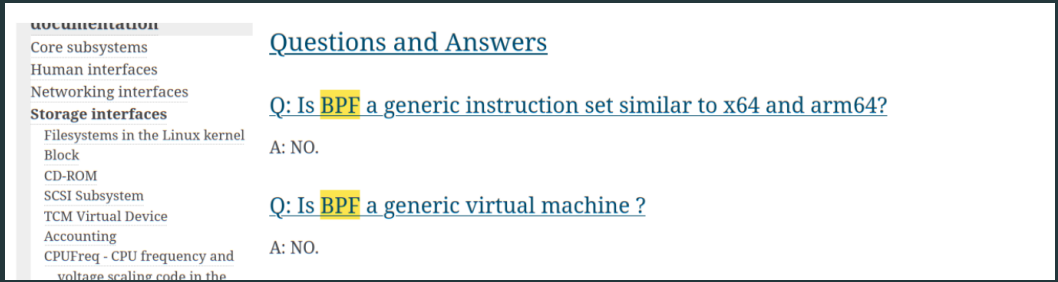
```
while (1) {
  // ...
}
```

✓✓ Source debugger

✓ No signed division

(-2 / -3) = ?

Linux Kernel Q&A on BPF[5]



[5] ***BPF Questions and Answers.*** https://www.kernel.org/doc/html/latest/bpf/bpf_design_QA.html. Linux Development Community.

More limitations overcome

Q: Can BPF programs access instruction pointer or return address?

A: NO.

Q: Can BPF programs access stack pointer ?

A: NO.

Only frame pointer (register R10) is accessible. From compiler point of view it's necessary to have stack pointer. For example, LLVM defines register R11 as stack pointer in its BPF backend, but it makes sure that generated code never uses it.

Q: Does C-calling convention diminishes possible use cases?

A: YES.

BPF design forces addition of major functionality in the form of kernel helper functions and kernel objects like BPF maps with seamless interoperability between them. It lets kernel call into BPF programs and programs call kernel helpers with zero overhead, as all of them were native C code. That is particularly the case for JITed BPF programs that are indistinguishable from native kernel C code.

Q: Does it mean that 'innovative' extensions to BPF code are disallowed?

A: Soft yes.

At least for now, until BPF core has support for bpf-to-bpf calls, indirect calls, loops, global variables, jump tables, read-only sections, and all other normal constructs that C code can produce.

Q: Can loops be supported in a safe way?

A: It's not clear yet.

BPF developers are trying to find a way to support bounded loops.

Yes, all of these

Q: Does it mean that 'innovative' extensions to BPF code are disallowed?

A: Soft yes.

At least for now, until BPF core has support for bpf-to-bpf calls, indirect calls, loops, global variables, jump tables, read-only sections, and all other normal constructs that C code can produce.

Is BPF a viable platform for software?

Evaluating BPF

Not yet.

Future Work

Next steps

- Release BPF VM source code. (WIP, stay in touch if interested)
- Release Python interface.

Stretch goals

- Contribute patches to the LLVM BPF backend (e.g. signed div)?
- Build a linker based on our current ELF loading code?

Flying the nest - a BPF port of Doom

LPC 2024

Arpad Kiss (hello at akissxyz point com)

September 18, 2024