Polar Signals

Developing eBPF profilers for polyglot cloud-native applications

Vaishali Thakkar (@vthakkar_)
Javier Honduvilla Coto <javier@polarsignals.com>

Agenda

- Infrastructure-wide profilers
- Low level ecosystem
- Stack unwinding/walking in the Linux kernel
- Building profilers using BPF
- Walking user stacks (without frame pointers)
- Future work and questions

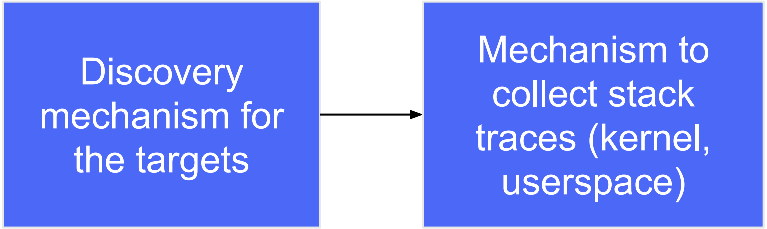
Profilers for the cloud native environment

- Developer machines != production systems
- Infrastructure-wide profilers
- Types of profilers
  - Tracing and sampling
- Raw data for sampling profilers
  - Different formats (pprof, folded etc)

Profilers for the cloud native environment

Discovery mechanism for the targets

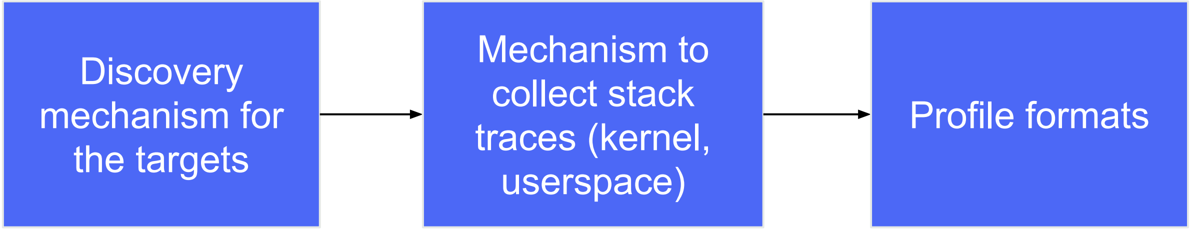
Profilers for the cloud native environment



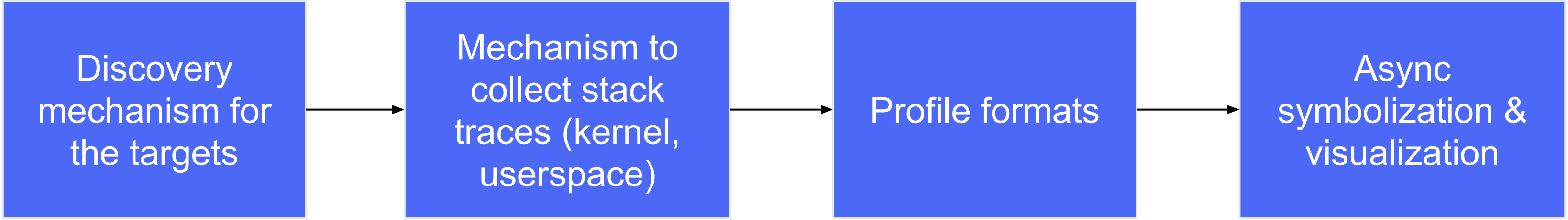
Discovery mechanism for the targets

Mechanism to collect stack traces (kernel, userspace)

Profilers for the cloud native environment


Discovery mechanism for the targets
Mechanism to collect stack traces (kernel, userspace)
Profile formats

Profilers for the cloud native environment


Discovery mechanism for the targets
Mechanism to collect stack traces (kernel, userspace)
Profile formats
Async symbolization & visualization

Low level ecosystem

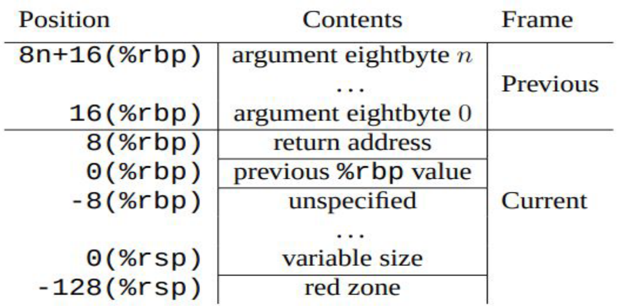
ELF and DWARF

- Executable Linkable format - ELF
  - For obj file, executable program, shared object etc
- DWARF - widely used debugging format
  - CIE - Common Information Entry
- Tools to read ELF and/or DWARF information
  - readelf, objdump, elfutils, llvm-dwarfdump
  - gcc also has -g option

Stacktraces and x86_64 ABI

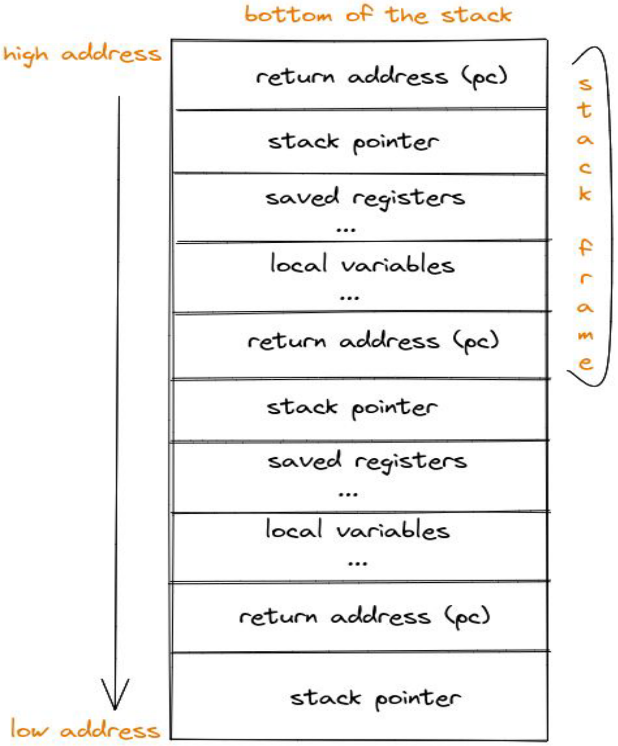
- What collecting stack traces involve
    - Kernel stacks
    - Application stacks
- Direction of stack growth
- So what are stack pointers, where do they come from

Figure 3.3: Stack Frame with Base Pointer



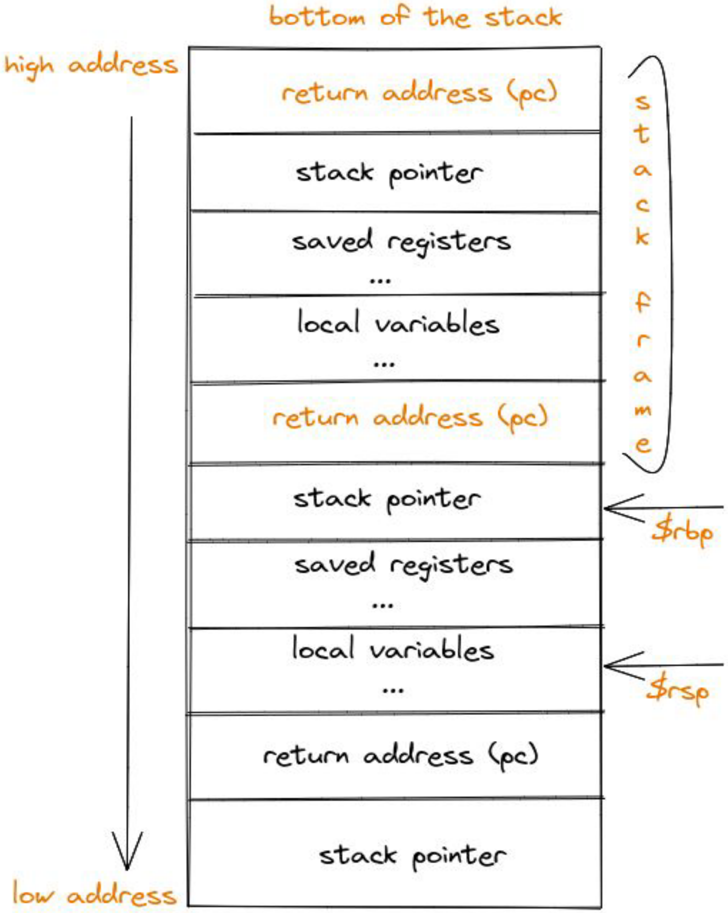
Position	Contents	Frame
8n+16(%rbp)	argument eightbyte n	Previous
	. . .	
16(%rbp)	argument eightbyte 0	
8(%rbp)	return address	Current
0(%rbp)	previous %rbp value	
-8(%rbp)	unspecified	
	. . .	
0(%rsp)	variable size	
-128(%rsp)	red zone	

From: x86_64 ABI specification



$rbp, $rsp & $rip registers

- $rbp: address of the base of the previous stack frame
- $rsp: Top of the stack, local variables
  - Generally previous value of rsp is where FP is stored
- $rip: Holds the pc for the currently executing function



Polar Signals

Frame pointers are often disabled

- Increased binary size → less i-cache hits
- 1 less register available

Cons of disabling frame pointers

- Walking stack traces becomes more expensive
- Less accuracy
- Way more work for compiler / debugger / profiler developers
- This information is large

The reality

- Great if you are hyperscaler

The ~~harsh~~ reality

- Great if you are hyperscaler
- But, for the rest of us…

Frame pointer believers

- Golang >=1.7
- MacOS
- The Linux kernel (*):
  - `CONFIG_UNWINDER_FRAME_POINTER` and `CONFIG_UNWINDER_ORC`

No frame pointers?

Stack unwinding in the Linux kernel w/o fp

- ORC (`CONFIG_UNWINDER_ORC` x86_64 only)
- Doesn’t rely on .debug_frame/.eh_frame
- Enabled by some of the major cloud vendors

Unwinding the stack without frame pointers

- DWARF unwind information
  - .eh_frame
  - .debug_frame
- Synthesizing them from object code
- Guessing which stack values are return addresses

.eh_frame – unwind tables

```
$ readelf -wF ./test_binary
      LOC           CFA      rbp   ra
   00000000004011f0 rsp+8    u     c-8
   00000000004011f1 rsp+16   c-16  c-8
   00000000004011f4 rbp+16   c-16  c-8
   0000000000401242 rsp+8    c-16  c-8
```

.eh_frame – generating unwind tables

```
$ readelf --debug-dump=frames ./test_binary
    DW_CFA_advance_loc: 1 to 00000000004011f1
    DW_CFA_def_cfa_offset: 16
    DW_CFA_offset: r6 (rbp) at cfa-16
    DW_CFA_advance_loc: 3 to 00000000004011f4
    DW_CFA_def_cfa_register: r6 (rbp)
    DW_CFA_advance_loc1: 78 to 0000000000401242
    DW_CFA_def_cfa: r7 (rsp) ofs 8
    DW_CFA_nop
```

Stack unwinding with eBPF

With frame pointers

```
stack_id = bpf_get_stackid(ctx, &user_stacks, BPF_F_USER_STACK);
```

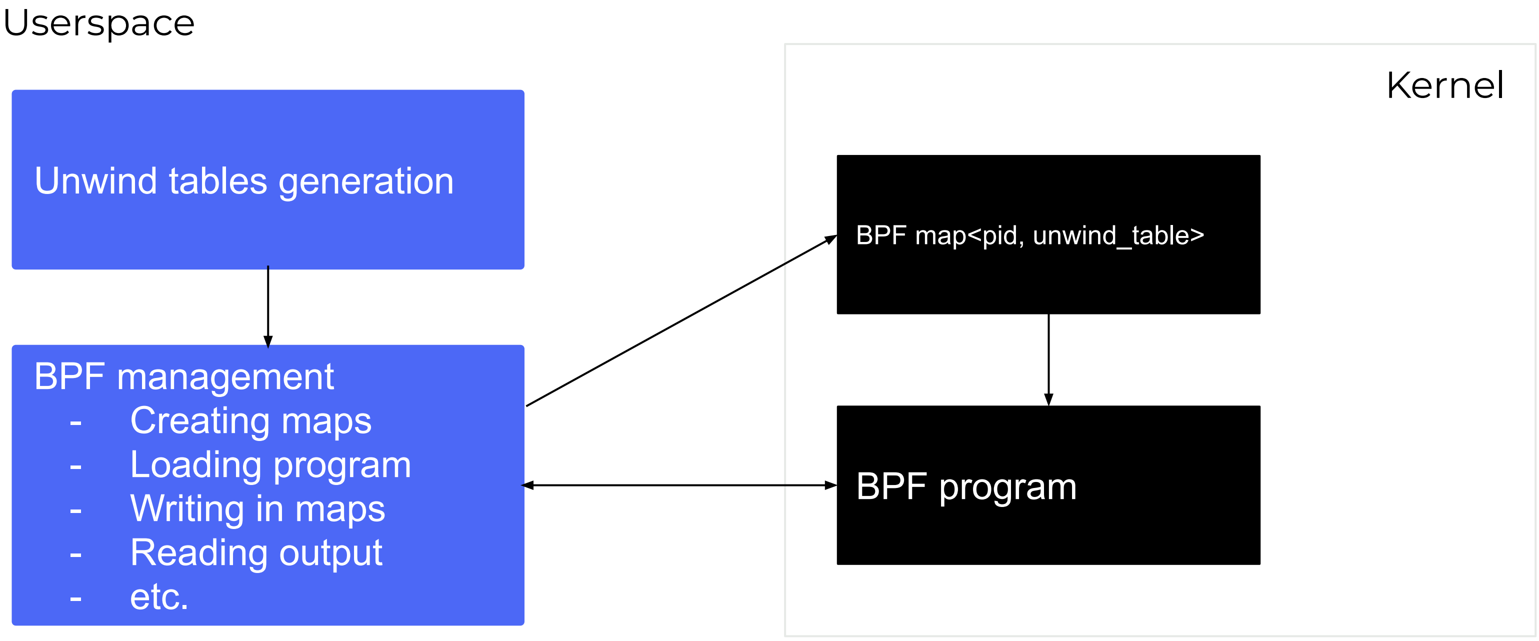
With frame pointers

```
stack_id = bpf_get_stackid(ctx, &user_stacks,
BPF_F_USER_STACK);

add_stack(stack_id);

// add_stack bumps map<stack_id, count_t>

// user_stacks = map<stack_id, array<addresses>>
```

Without frame pointers

- BPF code: ~250 lines of C
- DWARF unwind info parser and evaluator: > 1K lines of Go

Unwinding w/o frame pointers – architecture



Unwinding w/o frame pointers – unwind table

```
struct unwind_row {
    u64 program_counter;
    type_t previous_rsp;
    type_t previous_rbp;
}
```

Unwinding w/o frame pointers – unwind table gen

- `.eh_frame` / `.debug_frame`
  - Parse
  - Evaluate

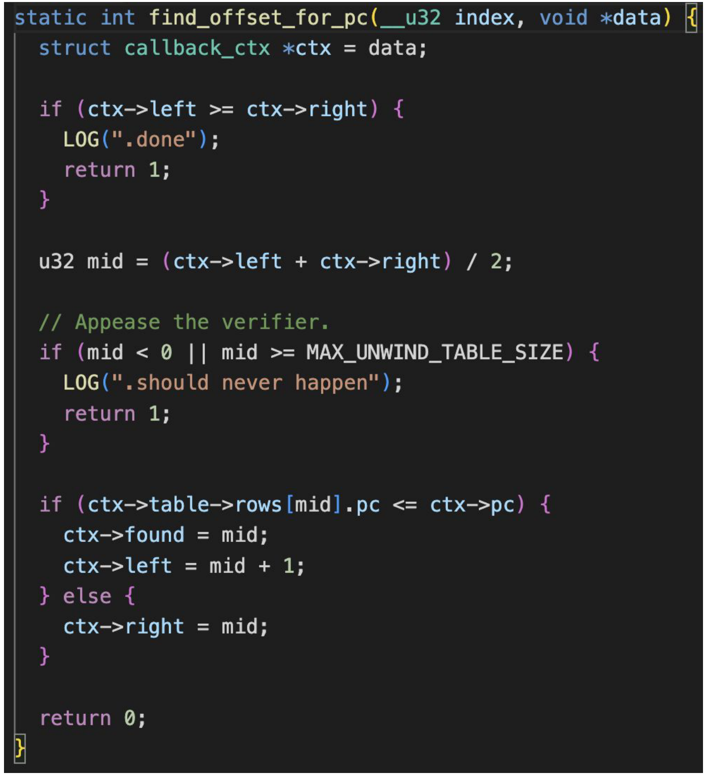
Unwinding w/o frame pointers – BPF (1)

- Find the unwind table for the current process
- While `main` isn't reached:
  - Append the program counter (`$rip`) to the walked stack
  - Find the unwind row for the current program counter
  - Restore registers for the previous frame
    - Return address `$rip`
    - Stack pointer `$rsp`
    - And `$rbp`, too

Unwinding w/o frame pointers – BPF (2)

- Efficiently finding the unwind data for a program counter
- Fun to implement in BPF :)

Unwinding w/o frame pointers – BPF (3)



```c
static int find_offset_for_pc(__u32 index, void *data) {
  struct callback_ctx *ctx = data;

  if (ctx->left >= ctx->right) {
    LOG(".done");
    return 1;
  }

  u32 mid = (ctx->left + ctx->right) / 2;

  // Appease the verifier.
  if (mid < 0 || mid >= MAX_UNWIND_TABLE_SIZE) {
    LOG(".should never happen");
    return 1;
  }

  if (ctx->table->rows[mid].pc <= ctx->pc) {
    ctx->found = mid;
    ctx->left = mid + 1;
  } else {
    ctx->right = mid;
  }

  return 0;
}
```

Unwinding w/o frame pointers – Future work

- Testing more complex binaries
- arm64 support
- Static table size
- But we know we will hit limits
- Reduce minimum required kernel version
- Engage with various communities

Polar Signals

Thank you!