Walking native stacks in BPF without frame pointers
Agenda

● Why the need for a DWARF-based stack walker in BPF
● Design of our stack walker
● Making it production ready
● Learnings so far
● Future plans
Native stack walker in BPF using DWARF: Why?

- Stack walking and history of frame pointers
- Current state of the world
  - How hyperscalers solve this problem
  - Recent discussions in Fedora mailing list - TL;DR: will be enabled Fedora 38, late-april release
  - Go runtime
  - Apple ecosystem
  - Simple Frame (previously known as CTF format)
- We want to support all the runtimes and distributions
Native stack walker in BPF using DWARF

• If not frame pointers then what?
  ○ .eh_frame/.debug_frame and DWARF CFI
  ○ How ORC does it?
Motivation

- If not frame pointers then what?
- Perf and libunwind
  - Security
  - Performance
Motivation

- If not frame pointers then what?
- Perf and libunwind
- BPF advantages
  - Higher safety
  - Lower barrier of entry
**.eh_frame**

- Call Frame Information (CFI)
- Space efficient and versatile
- Encoded unwind tables
- CFI opcodes
- Two main layers
  - State machine encoded in a VM - only need DW_CFA_remember_state and DW_CFA_restore_state
  - A special opcode that contains another set of opcode
Design

Unwind tables generation

BPF management
- Creating maps
- Loading program
- Writing in maps
- Reading output
- etc.

BPF maps

BPF program
Design

- Read the initial registers
  - Instruction pointer $rip
Design

• Read the initial registers
  ○ Instruction pointer $rip
  ○ Stack pointer $rsp
Design

- Read the initial registers
  - Instruction pointer $rip
  - Stack pointer $rsp
  - Frame pointer $rbp
Design

- Read the initial registers
  - Instruction pointer $rip
  - Stack pointer $rsp
  - Frame pointer $rbp

- While `unwind_frame_count <= MAX_STACKDEPTH`
  - Find the unwind table row for the PC
Design

- Read the initial registers
  - Instruction pointer $rip
  - Stack pointer $rsp
  - Frame pointer $rbp

- While `unwind_frame_count <= MAX_STACK_DEPTH`
  - Find the unwind table row for the PC
  - Add instruction pointer to the stack
Design

- Read the initial registers
  - Instruction pointer $rip
  - Stack pointer $rsp
  - Frame pointer $rbp

- While \textbf{unwind\_frame\_count} \leq \textbf{MAX\_STACK\_DEPTH}
  - Find the unwind table row for the PC
  - Add instruction pointer to the stack
  - Calculate the previous frame's stack pointer
Design

- Read the initial registers
  - Instruction pointer $rip
  - Stack pointer $rsp
  - Frame pointer $rbp

- While ```unwind_frame_count <= MAX_STACK_DEPTH```:
  - Find the unwind table row for the PC
  - Add instruction pointer to the stack
  - Calculate the previous frame's stack pointer
  - Update the registers with the calculated values for the previous frame
Design

- Read the initial registers
  - Instruction pointer $rip
  - Stack pointer $rsp
  - Frame pointer $rbp

- While unwind_frame_count <= MAX_STACK DEPTH
  - Find the unwind table row for the PC
  - Add instruction pointer to the stack
  - Calculate the previous frame's stack pointer
  - Updates the registers with the calculated values for the previous frame
  - Continue with the next frame - go back to adding instruction pointer
Storing the unwind information

- In-process, hijacking the process using `ptrace(2) + mmap(2) + mlock(2)`
  - Altering the execution flow of the program is a no-go
  - We must lock this memory
  - When to clean up?
  - Sharing of memory is harder, accounting for our overhead is also harder
Storing the unwind information

- BPF maps
  - A `<bytes, bytes>` hash-table
  - Always locked in memory, `BPF_F_NO_PREALLOC` is forbidden in tracing programs
  - We can reuse the same tables for multiple processes that share the same mappings
Storing the unwind information

libc    mysql    zlib    systemd    (unused)
Storing the unwind information – sharding

shard 0

shard 1

shard 2

shard 3
Storing the unwind information – sharding

systemd

shard 0

shard 1

shard 2

shard 3
Storing the unwind information – sharding

```
shard 0
shard 1
shard 2
shard 3
```

```
chunk1
c2
```

systemd
Storing the unwind information – sharding

(The above are stored in BPF maps)
Making our unwinder scale

- Unwind table for each executable mapping
  - Skip table generation most of the time (~0.9% of our CPU cycles in prod)
- This is suspiciously similar to a bump allocator
The unwinding process – in-depth

- pid
The unwinding process – in-depth

- pid
  - Do we have unwind information?
The unwinding process – in-depth

- pid
  - Do we have unwind information?
  - Find mapping with our current instruction pointer
The unwinding process – in-depth

• pid
  ○ Do we have unwind information?
  ○ Find mapping with our current instruction pointer
  ○ Find chunk
The unwinding process – in-depth

- **pid**
  - Do we have unwind information?
  - Find mapping with our current instruction pointer
  - Find chunk
  - We have the shard information
The unwinding process – in-depth

- pid
  - Do we have unwind information?
  - Find mapping with our current instruction pointer
  - Find chunk
  - We have the shard information
  - Let’s find the unwind info
The unwinding process – in-depth

- pid
  - Do we have unwind information?
  - Find mapping with our current instruction pointer
  - Find chunk
  - We have the shard information
  - Let's find the unwind info
  - Binary search in the table of up to 250k entries (~8 iterations)
The unwinding process – in-depth

- pid
  - Do we have unwind information?
  - Find mapping with our current instruction pointer
  - Find chunk
  - We have the shard information
  - Let's find the unwind info
  - Binary search in the table of up to 250k entries (~8 iterations)
  - Apply unwind action, add frame to stack-trace, continue with next frame
The unwinding process – in-depth

- If the stack is “correct”
  - We hash the addresses
  - Add the hash to a map
  - Bump a counter
BPF challenges

- Memlock, being aware of memory usage
- BPF verifier woes
  - Stack size: we rely on BPF maps to store state
  - Program size:
    - BPF tail calls to have bigger programs
    - Bounded loops (and bpf_loop) if you don't need to support older kernels 😊
Performance in userspace

- Many Go APIs aren’t designed with performance in mind
  - DWARF and ELF library in the stdlib
  - `binary.Read` & `binary.Write` allocate in the fast path (!!!)
- Profiling our profiler
  - Lots of found opportunities
  - But there’s more work to do!
Testing

- Thorough unit testing coverage for most of the core functions
- Snapshot testing for unwind tables ❤
Testing – snapshot testing

testdata @ c0d23d5

=> Function start: 2b450, Function end: 2b809
  pc: 2b450 cfa_type: 2 rbp_type: 0 cfa_offset: 8  rbp_offset: 0
  pc: 2b451 cfa_type: 2 rbp_type: 1 cfa_offset: 16 rbp_offset: -16
  pc: 2b454 cfa_type: 1 rbp_type: 1 cfa_offset: 16 rbp_offset: -16
  pc: 2b461 cfa_type: 1 rbp_type: 1 cfa_offset: 16 rbp_offset: -16
  pc: 2b6f2 cfa_type: 2 rbp_type: 1 cfa_offset: 8  rbp_offset: -16
  pc: 2b6f8 cfa_type: 1 rbp_type: 1 cfa_offset: 16 rbp_offset: -16
Testing – snapshot testing

```bash
write-dwarf-unwind-tables: build
    make -C testdata validate EH_FRAME_BIN=..,/dist/eh-frame
    make -C testdata validate-compact EH_FRAME_BIN=..,/dist/eh-frame

test-dwarf-unwind-tables: write-dwarf-unwind-tables
    $(CMD_GIT) diff --exit-code testdata/
```
Takeaways

● De-risking the project
● Invest early and often in automated testing
● BPF programs **must** have kernel tests
● Measure, profile, test...
  ○ but make sure to do it in prod do it in prod, too!
Testing in multiple kernels

Test results:

- ✔ 5.4
- ✔ 5.10
- ✔ 5.18
- ✔ 5.19

Test summary: 4 passed, 0 failed
Takeaways – different environments

- Different environments can radically change the performance profile
  - Different hardware
  - Different configuration (pprof...)
Different hardware – slow disks
Different configuration – signals in prod

Do not enable pprof profiling until BPF program is loaded #1276

Merged javierhonduco merged 1 commit into main from fix-sigprofs-interrupting-bpf-loading 2 days ago
Different configuration – signals in prod

- Go’s signal-based profiler uses SIGPROF
- Which interrupts our process’ execution
- Our BPF program is loaded and verified by the kernel
- Gets interrupted
- Libbpf retries up to 5 times
- And then we crash!
Other considerations

- Short-lived processes
- DWARF CFI vs our format
- Benchmarking the BPF code
Other considerations – DWARF CFI vs our format

typedef struct {
    u64 pc;
    u16 _reserved_do_not_use;
    u8 cfa_type;
    u8 rbp_type;
    s16 cfa_offset;
    s16 rbp_offset;
} stack_unwind_row_t;
Other considerations – DWARF CFI vs our format

typedef struct {
    u64 pc; // 😞
    u16 _reserved_do_not_use; // 😞
    u8 cfa_type;
    u8 rbp_type;
    s16 cfa_offset;
    s16 rbp_offset;
} stack_unwind_row_t;
Other considerations – DWARF CFI vs our format
Other considerations – DWARF CFI vs our format

- We support parsing every DWARF CFI opcode
- Only can unwind if
  - Previous frame stack pointer (CFA) is based off the current stack pointer or frame pointer + offsets
  - DWARF expressions in Procedure Linkage Tables (PLT) for CFA
  - We are working on:
    - CFA := any_register + offset
    - Frame pointer defined by an known expression
Other considerations – DWARF CFI vs our format

- 2 DWARF expressions account for the ~50% of what we’ve seen in the wild ([https://github.com/parca-dev/parca-agent/pull/1058](https://github.com/parca-dev/parca-agent/pull/1058))
- CFA based off not $rbp or $rsp rarely happens
- Some other instances that very rarely occur
Other considerations – BFP performance

- Walking stacks of a host running Postgres, CPython, Ruby (MRI) applications (some with >90 frames)
  - P50: 285ns
  - P90: 370ns
  - Max: 428ns

(kernel 6.0.18 with Intel i7-8700K (late '17) )
Profiling Ruby with BPF – rbperf

- Knowledge of the ABI of each interpreter version
- Stack walker implemented in BPF
  - Directly extract the function names and other information off Ruby's memory
What’s coming in Parca

- Mixed unwinding mode
- arm64 support
- Enabling this feature by default
- Support for other runtimes (JVM, Ruby, etc)
We ❤ OSS – contributors welcome!

- Everything we’ve talked about here is fully OSS
  - Userspace: Apache 2.0
  - BPF: GPL
References

- Our project website: https://www.parca.dev/
  - Agent: https://github.com/parca-dev/parca-agent
  - BPF code: https://github.com/parca-dev/parca-agent/tree/main/bpf/cpu
- Previous talk at Linux Plumbers conference: https://www.youtube.com/watch?v=GrIrrSyzvqfg
- rbperf: https://github.com/javierhonduco/rbperf
Thank you!

Vaishali <vaishali.thakkar@polarsignals.com>
Mastodon: @vaishali@hachyderm.io

Javier <javier@polarsignals.com>
Mastodon: @javierhc@hachyderm.io