Modernizing bpftrace with libbpf

Viktor Malík
Kernel Engineer, Red Hat

LPC, September 19, 2024

bpftrace overview

“High-level tracing language and tool for Linux based on BPF.”

bpftrace overview

“High-level tracing language and tool for Linux based on BPF.”

Main bpftrace goals:

- Provide a powerful yet simple language for fast prototyping of tracing programs.
- Create an abstraction for the BPF layer.

bpftrace overview

"High-level tracing language and tool for Linux based on BPF."

Main bpftrace goals:

- Provide a powerful yet simple language for fast prototyping of tracing programs.
- Create an abstraction for the BPF layer.

Example: one-liner collecting numbers of VFS calls during one second:

```
# bpftrace -e 'kprobe:vfs_* { @[func] = count() } interval:s:1 { exit() }'
Attaching 2 probes...
@[vfs_readlink]: 4
@[vfs_fstatat]: 5
[...]
```

Problem: legacy architecture

Legacy bpftrace workflow

❶ Start with bpfscript program

Problem: legacy architecture

Legacy bpftrace workflow

1. Start with bpfscript program
2. Generate LLVM IR

Problem: legacy architecture

Legacy bpftrace workflow

1. Start with bpfscript program
2. Generate LLVM IR
3. Use LLVM to compile into BPF ELF object

Problem: legacy architecture

Legacy bpftrace workflow

1. Start with bpfscript program
2. Generate LLVM IR
3. Use LLVM to compile into BPF ELF object
4. For each map:
   - call `bpf_map_create` to obtain FD
   - manually fill FD to programs (relocate)

Problem: legacy architecture

Legacy bpftrace workflow

1. Start with bpfscript program
2. Generate LLVM IR
3. Use LLVM to compile into BPF ELF object
4. For each map:
   - call `bpf_map_create` to obtain FD
   - manually fill FD to programs (relocate)
5. For each program:
   - call `bpf_prog_load` to obtain FD
   - call libbpf/BCC to attach

Problem: legacy architecture

Drawbacks and limitations

- Cannot use “modern” BPF features which rely on BTF and relocations:
  - subprograms,
  - CO-RE,
  - global variables,
  - kfuncs,
  - ...
- Duplicating a lot of operations already performed by libbpf.

Migration to new architecture

Migration to new architecture

Architecture overview

1. Start with bpfscript program
2. Generate LLVM IR
3. Use LLVM to compile into BPF ELF object
4. For each map:
   - call `bpf_map_create` to obtain FD
   - manually fill FD to programs (relocate)
5. For each program:
   - call `bpf_prog_load` to obtain FD
   - call libbpf/BCC to attach

Migration to new architecture

Architecture overview

1. Start with bpfscript program
2. Generate LLVM IR
3. Use LLVM to compile into libbpf-compliant BPF ELF object
4. For each map:
   - call `bpf_map_create` to obtain FD
   - manually fill FD to programs (relocate)
5. For each program:
   - call `bpf_prog_load` to obtain FD
   - call libbpf/BCC to attach

Migration to new architecture

Architecture overview

1. Start with bpfscript program
2. Generate LLVM IR
3. Use LLVM to compile into libbpf-compliant BPF ELF object
4. Call `bpf_object__open` and `bpf_object__load`

Red Hat

Migration to new architecture

Architecture overview

❶ Start with bpfscript program

❷ Generate LLVM IR

❸ Use LLVM to compile into libbpf-compliant BPF ELF object

❹ Call `bpf_object__open` and `bpf_object__load`

❺ Call libbpf/BCC to attach

BPF ELF object

Requirements

- Programs are stored in TEXT sections and identified by function names
- Subprograms are stored in `.text` section
- Maps are stored in `.maps` section in BTF format
- License is stored in `license` section

There's an ongoing standardization effort:
`https://www.ietf.org/archive/id/draft-thaler-bpf-elf-00.html`

BPF ELF object

Programs and subprograms

- Each BPF (sub)program is represented as a separate function
    - Problem are bpftrace wildcarded probes – same program is attached to multiple attach points

BPF ELF object

Programs and subprograms

- Each BPF (sub)program is represented as a separate function
  - Problem are bpftrace wildcarded probes – same program is attached to multiple attach points

- Also add function info for (sub)programs:
  - Create LLVM debug info
  - Let LLVM generate BTF (`.BTF` and `.BTF.ext` sections)

BPF ELF object

Maps

- Maps are defined as global variables in the `.maps` DATA section
- Each map needs to have a corresponding BTF type entry
- Mandatory fields are:
    - `type` – e.g. `BPF_MAP_TYPE_HASH`
    - `max_entries`
    - `key` – key type
    - `value` – value type

BPF ELF object

Maps

- Maps are defined as global variables in the `.maps` DATA section
- Each map needs to have a corresponding BTF type entry
- Mandatory fields are:
  - `type` – e.g. `BPF_MAP_TYPE_HASH`
  - `max_entries`
  - `key` – key type
  - `value` – value type

- Integer values are represented by pointers to arrays of ints in which dimensionality of the array encodes the specified value.

BPF ELF object

Map BTF definition example

```
[1] PTR '(anon)' type_id=3
[2] INT 'int' size=4 bits_offset=0 nr_bits=32 encoding=SIGNED
[3] ARRAY '(anon)' type_id=2 index_type_id=4 nr_elems=1
[4] INT '__ARRAY_SIZE_TYPE__' size=4 bits_offset=0 nr_bits=32 encoding=(none)
[5] PTR '(anon)' type_id=6
[6] ARRAY '(anon)' type_id=2 index_type_id=4 nr_elems=4096
[7] PTR '(anon)' type_id=8
[8] INT 'int64' size=8 bits_offset=0 nr_bits=64 encoding=SIGNED
[9] STRUCT '(anon)' size=32 vlen=4
    'type' type_id=1 bits_offset=0
    'max_entries' type_id=5 bits_offset=64
    'key' type_id=7 bits_offset=128
    'value' type_id=7 bits_offset=192
```

BPF ELF object

Map BTF definition example

```
[1] PTR '(anon)' type_id=3
[2] INT 'int' size=4 bits_offset=0 nr_bits=32 encoding=SIGNED
[3] ARRAY '(anon)' type_id=2 index_type_id=4 nr_elems=1
[4] INT '__ARRAY_SIZE_TYPE__' size=4 bits_offset=0 nr_bits=32 encoding=(none)
[5] PTR '(anon)' type_id=6
[6] ARRAY '(anon)' type_id=2 index_type_id=4 nr_elems=4096
[7] PTR '(anon)' type_id=8
[8] INT 'int64' size=8 bits_offset=0 nr_bits=64 encoding=SIGNED
[9] STRUCT '(anon)' size=32 vlen=4
    'type' type_id=1 bits_offset=0
    'max_entries' type_id=5 bits_offset=64
    'key' type_id=7 bits_offset=128
    'value' type_id=7 bits_offset=192
```

BPF ELF object

Map BTF definition example

```
[1] PTR '(anon)' type_id=3
[2] INT 'int' size=4 bits_offset=0 nr_bits=32 encoding=SIGNED
[3] ARRAY '(anon)' type_id=2 index_type_id=4 nr_elems=1
[4] INT '__ARRAY_SIZE_TYPE__' size=4 bits_offset=0 nr_bits=32 encoding=(none)
[5] PTR '(anon)' type_id=6
[6] ARRAY '(anon)' type_id=2 index_type_id=4 nr_elems=4096
[7] PTR '(anon)' type_id=8
[8] INT 'int64' size=8 bits_offset=0 nr_bits=64 encoding=SIGNED
[9] STRUCT '(anon)' size=32 vlen=4
    'type' type_id=1 bits_offset=0
    'max_entries' type_id=5 bits_offset=64
    'key' type_id=7 bits_offset=128
    'value' type_id=7 bits_offset=192
```

BPF ELF object

Map BTF definition example

```
[1] PTR '(anon)' type_id=3
[2] INT 'int' size=4 bits_offset=0 nr_bits=32 encoding=SIGNED
[3] ARRAY '(anon)' type_id=2 index_type_id=4 nr_elems=1
[4] INT '__ARRAY_SIZE_TYPE__' size=4 bits_offset=0 nr_bits=32 encoding=(none)
[5] PTR '(anon)' type_id=6
[6] ARRAY '(anon)' type_id=2 index_type_id=4 nr_elems=4096
[7] PTR '(anon)' type_id=8
[8] INT 'int64' size=8 bits_offset=0 nr_bits=64 encoding=SIGNED
[9] STRUCT '(anon)' size=32 vlen=4
    'type' type_id=1 bits_offset=0
    'max_entries' type_id=5 bits_offset=64
    'key' type_id=7 bits_offset=128
    'value' type_id=7 bits_offset=192
```

BPF ELF object

License

- Necessary to allow usage of GPL-only helpers
- Defined by a global string inside the `license` section:

```
@LICENSE = global [4 x i8] c"GPL\00", section "license"
```

New enabled features

For-loops for maps

- Allow iteration over all map elements:

```
kprobe:vfs_* {
  @[func] = count();
}
END {
  for ($kv : @) {
    printf("%s called %d times\n", $kv.0, $kv.1);
  }
}
```

- use `bpf_for_each_map_elem` under the hood
- loop body is transformed into a callback function

New enabled features

bpfscript subprograms

- Allow splitting scripts into multiple functions:

```
fn get_path($ps: struct path *): string[64] {
  return str($ps->dentry->d_name.name);
}
kprobe:vfs_read {
  printf("read %s\n", get_path((struct path *)arg0));
}
```

- Currently WIP by Tomáš Glozar
- Simplifies the code, allows code reuse
- Opens up a way to bpftrace standard library

New enabled features

Calling external functions

- Idea: allow calling BPF functions from other ELF files
- Example usage: stack walkers written in pure BPF
- Currently WIP by Alastair Robertson

New enabled features

...and many more

- global variables (already used internally)
- kfuncs (WIP)
- CO-RE (ahead-of-time compiled programs – PoC already working)
- ...

Problems (and solutions)

Incomplete description of the BPF ELF format

- Current document is not complete
- Missing parts:
  - description of BTF map format
  - (in)valid characters in probe names
  - global variables format
  - ...and probably more
- It would be nice to proceed with the standardization effort or at least have a more complete documentation of the format
- Is or should the ELF format be considered ABI of libbpf?

Wildcarded probes

- Problem: bpftrace needs to attach the same code to potentially many targets

Wildcarded probes

- Problem: bpftrace needs to attach the same code to potentially many targets
- Possible solutions:
  - Multi-probes
    - available for certain program types only (kprobes, uprobes)
    - could be added for other program types (fentry/fexit, (raw) tracepoints?)

Wildcarded probes

- Problem: bpftrace needs to attach the same code to potentially many targets

- Possible solutions:
  - Multi-probes
    - available for certain program types only (kprobes, uprobes)
    - could be added for other program types (fentry/fexit, (raw) tracepoints?)
  - Duplicating programs in ELF object
    - quite space-inefficient (ELF can go from 9k to 60k)
    - sometimes inevitable (e.g. for tracepoint/USDT args)
    - currently implemented when multi-probes are unavailable

Wildcarded probes

- Possible solutions (cont.):
    - Manual cloning via `bpf_prog_load`
        - ELF contains just one instance of the function which is processed by libbpf
        - bpftrace clones the processed instructions by calling `bpf_prog_load` for each target
        - used by retsnoop

Wildcarded probes

- Possible solutions (cont.):
  - Manual cloning via `bpf_prog_load`
    - ELF contains just one instance of the function which is processed by libbpf
    - bpftrace clones the processed instructions by calling `bpf_prog_load` for each target
    - used by retsnoop
  - Using global subprograms
    - makes the ELF object considerably smaller
    - subprograms are still cloned in the kernel

Wildcarded probes

- Possible solutions (cont.):
  - Manual cloning via `bpf_prog_load`
    - ELF contains just one instance of the function which is processed by libbpf
    - bpftrace clones the processed instructions by calling `bpf_prog_load` for each target
    - used by retsnoop
  - Using global subprograms
    - makes the ELF object considerably smaller
    - subprograms are still cloned in the kernel
  - Using symbol aliasing
    - compiler allows to create multiple symbol table entries for the same address
    - each alias is interpreted as a different program by libbpf (and cloned)
    - needs libbpf changes in relocations and linker
    - too complicated

Wildcarded probes

Proposed solution

❶ Implement manual cloning via `bpf_prog_load`

- it's the simplest approach
- may be further simplified by new libbpf API
- also used by retsnoop

❷ Gradually add multi-probe support for more program types

- fentry/fexit
- BTF-enabled raw tracepoints
- normal tracepoints should work out-of-box

Error reporting

- Problem: when `bpf_object__load` fails, it is impossible to determine which program/map failed to load/create
    - `bpf_object__load` returns `-errno`
    - FDs are either -1 (uninitialized) of >0
    - libbpf log contains information on which program/map failed to load/create

Error reporting

- Problem: when `bpf_object__load` fails, it is impossible to determine which program/map failed to load/create
  - `bpf_object__load` returns `-errno`
  - FDs are either -1 (uninitialized) of >0
  - libbpf log contains information on which program/map failed to load/create

- Proposed solution: store `-errno` in program/map FDs in case of failure

Missing features and future work

Missing features

- Attachment via libbpf
  - BCC is currently used for: k(ret)probe, u(ret)probe, tracepoints, USDTs, perf events
  - Migrating these would allow bpftrace to drop dependency on BCC completely
  - Eventually, we could even use auto-attachment based on section names

Missing features

- Attachment via libbpf
    - BCC is currently used for: k(ret)probe, u(ret)probe, tracepoints, USDTs, perf events
    - Migrating these would allow bpftrace to drop dependency on BCC completely
    - Eventually, we could even use auto-attachment based on section names
- CO-RE relocations
    - Useful for AOT (ahead-of-time compilation)
    - Should be now easy to do as libbpf should take care of everything

Conclusion

- bpftrace underwent significant modernization by offloading a lot of program loading to libbpf
- May be a good inspiration to other projects striving to implement custom front-end for BPF, using libbpf as the back-end
- There’s still a lot of work to do, especially on bpftrace side but also on libbpf/community side

Conclusion

- bpftrace underwent significant modernization by offloading a lot of program loading to libbpf
- May be a good inspiration to other projects striving to implement custom front-end for BPF, using libbpf as the back-end
- There's still a lot of work to do, especially on bpftrace side but also on libbpf/community side
- Do you want to discuss bpftrace? Come to our BoF session on **Friday 5pm, Room 1.14**.

Conclusion

- bpftrace underwent significant modernization by offloading a lot of program loading to libbpf
- May be a good inspiration to other projects striving to implement custom front-end for BPF, using libbpf as the back-end
- There’s still a lot of work to do, especially on bpftrace side but also on libbpf/community side
- Do you want to discuss bpftrace? Come to our BoF session on **Friday 5pm, Room 1.14**.

Thank you for the attention!

Questions?

Red Hat