Evolution of stack trace capture with BPF

Andrii Nakryiko

Meta

BPF_MAP_TYPE_STACK_TRACE

```
struct {
     __uint(type, BPF_MAP_TYPE_STACK_TRACE);
     __uint(max_entries, MAX_STACK_TRACE_CNT);
     __uint(key, u32);
     __uint(value, u64[PERF_MAX_STACK_DEPTH]);
} stacks SEC(".maps");
```

```
int bpf_get_stackid(void *ctx, void *map, __u64 flags);
```

BPF side

```
id = bpf_get_stackid(ctx, &stacks, BPF_F_USER_STACK);
if (id < 0) {
    /* failure */
}

sample.ustack_id = id;

bpf_perf_event_output(ctx, …, &sample, sizeof(sample));
```

User space side

```
u64 addrs[PERF_MAX_STACK_DEPTH];

err = bpf_map_lookup_elem(map_fd, &sample.ustack_id, &addrs);
if (err) {
    /* error handling */
}

/* first N elements of addrs[] contain captured addresses */
```

Build ID support

- Possible to capture **`(build ID + file offset)`** instead of absolute address
- `.map_flags = BPF_F_USER_BUILD_ID`
- Special per-stack frame type:

```
#define BPF_BUILD_ID_SIZE 20
struct bpf_stack_build_id {
    __s32         status;
    unsigned char   build_id[BPF_BUILD_ID_SIZE];
    union {
            __u64   offset;
            __u64   ip;
    };
};
```

Quirks of STACK_TRACE API

- Returns 32-bit stack ID (*convenient!*)
- Captures user space stack trace (BPF_F_USER_STACK)
- … or kernel stack trace (**omit** BPF_F_USER_STACK)
  - can’t capture both (and no one complained so far!)
- actual number of captured addresses is **implicit** (!)
- automatic stack **deduplication**

Implementation: the good

Specialized hash map implementation.

Stacks deduplication can save space.

Design favors space efficiency and performance.

Does not support *hash collisions*.

Implementation: the bad

Hash collisions are *pretty frequent* and **unavoidable**!

Hash collision handling and tradeoffs controlled through flags:

- `BPF_F_FAST_STACK_CMP` – compare **only hashes**
- `BPF_F_REUSE_STACKID` – **overwrite** previous stack trace

Implementation: the ugly

Choice between two bad options:

- Lose data
  - Without `BPF_F_REUSE_STACKID` – drop stack trace even if there is space available
- Corrupt data
  - With `BPF_F_REUSE_STACKID` – corrupt all previous references for same stack ID

Our production *never* uses `BPF_F_REUSE_STACKID`!

Implementation: the ugly

- Stack dedup makes removal from STACK_TRACE **inherently racy**.
- While user space deletes element, BPF side might use that stack ID.
- Can't free up space as soon as user space consumed stack trace (!)
- STACK_TRACE is not well suited for longer-running sessions.

Making it work in practice

"Double buffering" approach:

- two `STACK_TRACE` maps, one active at a time
- the other is read and cleared by user space

Cons:

- wastes memory
- complicates setup
- a small transition window:
  - user space consumes stack traces
  - while BPF side completes writing into it

Observations from production

CPU profiling didn't benefit much from deduplication of stacks.

Stack traces are pretty unique, overall.

Let users manage memory.

Evolution: bpf_get_stack()

```
int bpf_get_stack(void *ctx, void *buf, __u32 size, __u64 flags);
```

- captures stack trace into user-supplied buffer
- returns amount of actual data
    - clears the tail, making it usable as part of hash map key (!)
- up to user how to use it afterwards:
    - dedup as part of `HASH` map key
    - send to user space with BPF ringbuf
    - analyze in BPF code (*but I'm not aware of anyone doing it*)
- All **but one** use cases at Meta switched to `bpf_get_stack()`!

Are done yet?

Not quite.

There are still problems.

Synchronous API: assumptions

- stacks are captured **synchronously**
- assume worst case (i.e., NMI context)
- no page faults allowed, memory has to be physically present

Synchronous API: consequences

- user stack traces capture can be **unreliable**
- build ID support is restricted and unreliable
  - again, worst-case NMI assumptions;
  - fails if build ID ELF note is not physically present
  - fails if build ID is not **within first 4KB of ELF file** (!)
  - there were attempts to add build ID caching (NACKed, though)
- kernel stack traces are **oblivious** to this (reliable!)

Synchronous API: limitations

- (*fundamentally*) incompatible with SFrame or .eh_frame (DWARF) stack unwinding approaches
- can’t wait for necessary data to be paged in

We need a new API

This time, **asynchronous**!

Asynchronous API: kernel stacks

- can't be done for kernel stack traces
- they are needed here and now (perf_event, kprobe, tracepoint)
- **good news**: it already works well even with synchronous API

Asynchronous API: user stacks

- *Key observation*: user stacks can be **postponed**
- requested in NMI – captured just before returning to user space
- user stack trace is still the same (user thread is frozen)
- do it in faultable (a.k.a. "sleepable") context
  - means we can wait for ELF data to be paged in, if necessary

API design: overview

- `bpf_get_stackid()`-like API, returning 32-bit stack ID
- ID is **a reservation,** stable and can be recorded upfront
- kernel stack trace is captured synchronously
- user stack trace is scheduled until return to user space
- `bpf_map_lookup_elem()` returns `-EAGAIN` if stack is not ready

API design: ~~deduplication~~

- `STACK_TRACE` map is notoriously hard to use reliably
- Stack deduplication **has to go** as part of public API.
- One ID – one unique stack.
- Makes `bpf_map_delete_elem()` race-free (no risk of reusing ID)

API design: ~~deduplication~~

- internal dedup is *possible* (but hidden from user)
  - Internal refcounting
  - `bpf_map_delete_elem()` drops refcount of underlying stack trace memory
- CPU vs memory trade off
  - complexity and CPU overhead with dedup
  - race-free deletes allow fast memory reuse!

Opinion: seems not worth it to bother.

API design: notifications

How to notify user that stack trace is ready?

API design: notifications

- trivial: no notification
    - (+) no code is best code
    - (-) user code forced to periodically retry (but maybe that's ok?)

API design: notifications

- easy: map-wide epoll notification whenever **any** stack trace is ready
    - (+) cheap and simple
    - (-) might be wasteful for user, causing many retries

API design: notifications

- wasteful: each slot supports epoll
  - (+) user can poll on each stack ID
  - (-) need to create FD for each ID
  - (-) each slot embeds `wait_queue_head_t`

API design: notifications

- (?) efficient: BPF ringbuf as an efficient delivery mechanism
  - (+) IDs are sent as they become "ready"
  - (+) Very efficient notification and consumption
  - (-) What to do if BPF ringbuf is full?
    - (?) User problem
    - (?) Some map stats
- (?) **Send entire stack trace?**
  - (+) variable-length data is possible, no space waste
  - (+) extensible way (BPF ringbuf record size is reported to user)

API design: customization

- should we allow custom BPF program for stack unwinding?
  - `bpf_wq` should be flexible and sufficient for that?
- good built-in kernel support is important
  - uretprobe "corrupting" stack trace
  - kernel can fix this up ([0])
- SFrame is coming?
- is limited `.eh_frame` (DWARF) support feasible?

[0] https://lore.kernel.org/all/20240508212605.4012172-3-andrii@kernel.org/

Thank you!