BPF Map Tracing: Hot Updates of Stateful Programs

Joe Burton
Google
Mountain View, United States
jevburton@google.com

***Abstract*—In this document we introduce BPF map tracing: the ability to execute BPF programs upon map access. This facility enables non-disruptive updates of sets of stateful BPF programs, which is needed in policy enforcement use cases and convenient in many more.**

***Index Terms*—eBPF, maps, tracing, updates**

I. Introduction

Extended Berkeley Packet Filter (eBPF) is a Linux kernel feature providing sandboxed execution of verifiably safe programs. Programs execute in response to events, and generally share data through maps. Programs may read and write to maps from kernel space, and usermode applications may also read and write to the same maps.

II. Motivation

At Google we frequently see sets of related BPF programs deployed together. These programs are expected to upgrade and downgrade as one unit. There are frequently data dependencies between these programs in the form of map accesses, as illustrated in Figure 1.

For some applications, map contents are critical and should not be lost on upgrade. It is not feasible to enforce that the data in the maps never changes its layout. Developers should be able to reorder, add and delete fields as they see fit. Many applications like this also cannot tolerate downtime. The problem then, is how to upgrade a stateful set of programs with zero downtime while also migrating state.

Let's walk through a theoretical example using our simple BPF application. We want to upgrade from v0 to v1, such that at any instant, any program which runs does not have a broken data dependency.

At the beginning of the transform, we have v0 loaded and attached. We also load v1. Because v1 is not attached, it has no effect. This is illustrated in Figure 2.

At the end of the transform, we will have v1 loaded and attached. v1's map will be populated with the migrated contents of v0's map. v0 may still be loaded, but because it is no longer attached, it has no effect. This is illustrated in Figure 7.

The rest of this paper details how such a migration can be done.

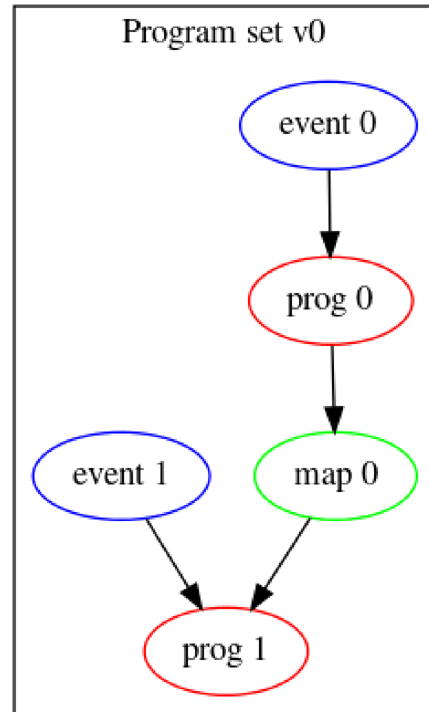

Fig. 1. Simple BPF application with a data dependency between two programs.

III. Theoretical Solution

Starting from the state shown in Figure 2, we attach a BPF program to map 0. This program runs whenever map 0 is modified. Because it is an arbitrary program, we can make it propagate those modifications to map 0'. This is illustrated in Figure 3. I will refer to this program as a copy-on-write handler, because its purpose is to give map 0 copy-on-write semantics.

Next, we can perform a bulk migration of the data in map0 to map0'. This migration should skip any cells already written by the copy-on-write handler to avoid overwriting newer data with older data. This can be achieved with a map iterator which acquires the same lock as the copy-on-write handler. See Figure 4.

Next, we can start swapping the programs attached to our

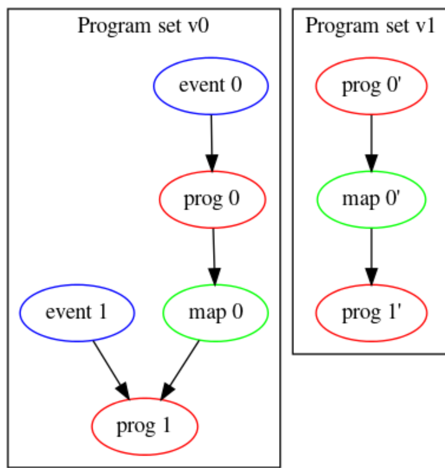

Fig. 2. System state before migrating.

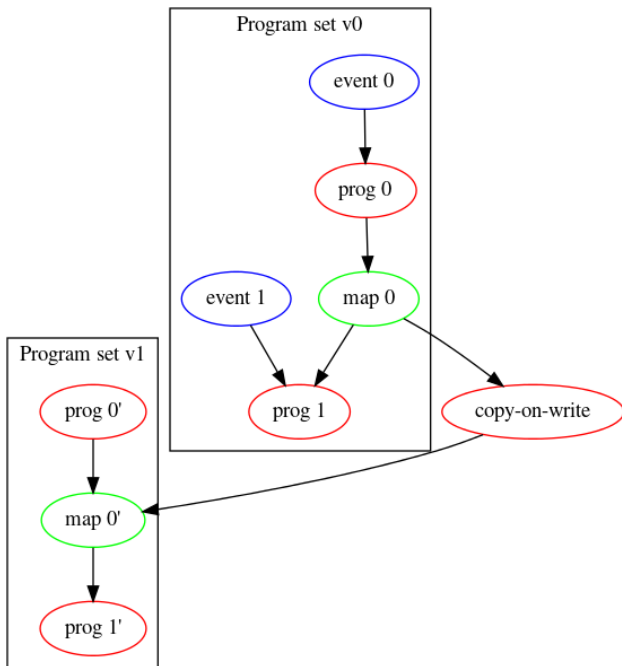

Fig. 3. Phase 1: install a copy-on-write handler on map 0.

events. The order of swapping depends on the topological sort of our programs with respect to their data dependencies. In other words, data consumers should be swapped before data providers. The first swap is shown in Figure 5, and the second swap is shown in Figure 6. After the first swap, the consuming program's data dependencies are clearly met: very old data is present due to the bulk transfer, and very recent data is present due to the copy-on-write handler. In addition, because the data it's accessing passed through either our copy-on-write handler or our bulk transfer, it has been migrated.

Finally, we can unload our copy-on-write handler and bulk transfer program in any order. We have transformed the state

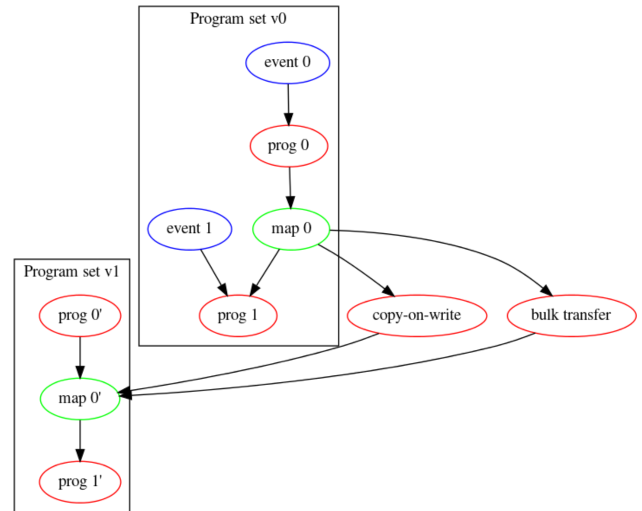

Fig. 4. Phase 2: perform bulk migration between the two maps.

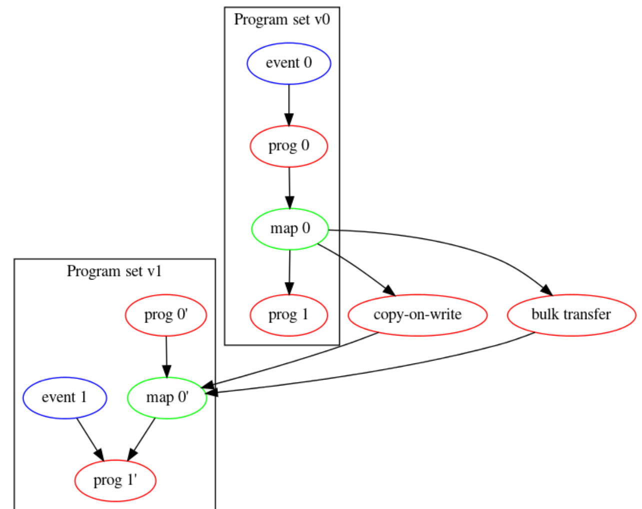

Fig. 5. Phase 3.a: atomically swap data consumer.

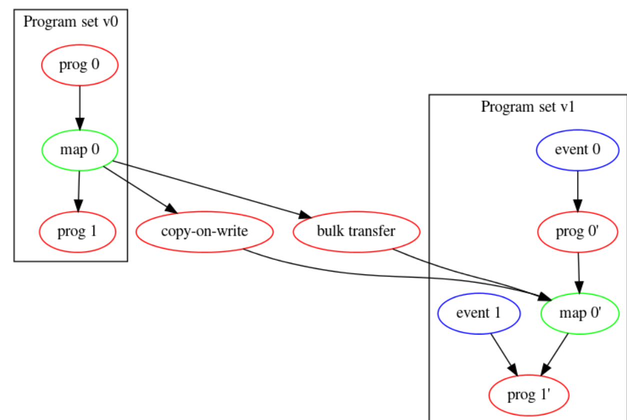

Fig. 6. Phase 3.b: atomically swap data producer.

of the system from Figure 2 to Figure 7.

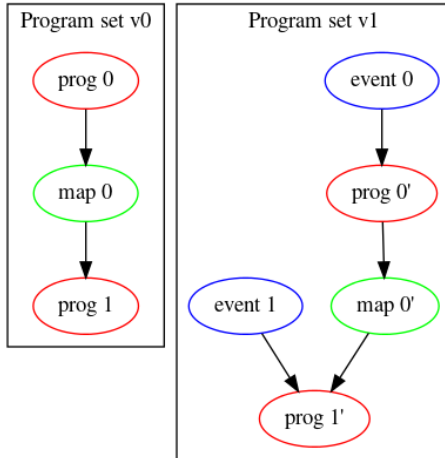

Fig. 7. System state after migrating.

IV. Kernel Implementation

The bulk transfer can be implemented using a map iterator and requires no kernel changes. The copy-on-write handler requires kernel changes: there's no existing facility to execute a program on map updates. The remainder describes the work we've done implementing such a facility.

A. A toy example

This toy program runs when `BPF_MAP_UPDATE_ELEM` is called on `traced_map`. We apply the collatz transform to the value to illustrate the intent of data transformation, but any BPF-verifiable transformation could be applied.

```
uint32_t collatz(uint32_t x)
{
        return x % 2 ? x * 3 + 1 : x / 2;
}

SEC("map_trace/traced_map/UPDATE_ELEM")
int tracer(struct
    bpf_map_trace_ctx__update_elem *ctx)
{
        uint32_t key = 0, val = 0;

        if (bpf_probe_read(&key, sizeof(key),
            ctx->key))
                return 1;
        if (bpf_probe_read(&val, sizeof(val),
            ctx->value))
                return 1;
        val = collatz(val);
        bpf_map_update_elem(&tracer_map, &key,
            &val, /*flags=*/0);
        return 0;
}
```

The context depends on the operation being applied to the map. It always mirrors the data being passed into a map modification function.

```
struct bpf_map_trace_ctx__update_elem {
        __bpf_md_ptr(void *, key);
        __bpf_md_ptr(void *, value);
        u64 flags;
};

struct bpf_map_trace_ctx__delete_elem {
        __bpf_md_ptr(void *, key);
};
```

B. Loading and attaching

Users load their `BPF_PROG_TYPE_TRACING` program and attach it to their chosen map with `BPF_LINK_CREATE`. Links are extended so that a program can be attached to a particular operation on a particular map:

```
enum bpf_map_trace_type {
        BPF_MAP_TRACE_UPDATE_ELEM = 0,
        BPF_MAP_TRACE_DELETE_ELEM = 1,

        MAX_BPF_MAP_TRACE_TYPE,
};

struct bpf_map_trace_link_info {
        __u32 map_fd;
        enum bpf_map_trace_type trace_type;
};
```

At link creation time, we have to handle the possibility that the program updates the same map that it's tracing. This could happen directly:

```
/* This traces traced_map and updates it,
   creating an (invalid) infinite loop.
 */
SEC("map_trace/traced_map/UPDATE_ELEM")
int tracer(struct
    bpf_map_trace_ctx__update_elem *ctx)
{
        uint32_t key = 0, val = 0;

        bpf_map_update_elem(&traced_map, &key,
            &val, /*flags=*/0);
        return 0;
}
```

... or indirectly:

```
SEC("map_trace/map0/UPDATE_ELEM")
int tracer0(struct
    bpf_map_trace_ctx__update_elem *ctx)
{
        uint32_t key = 0, val = 0;

        bpf_map_update_elem(&map1, &key, &val,
            /*flags=*/0);
        return 0;
}

/* Since this traces map1 and updates map0,
   it forms an infinite loop with
 * tracer0.
 */
```

```
SEC("map_trace/map1/UPDATE_ELEM")
int tracer1(struct
    bpf_map_trace_ctx__update_elem *ctx)
{
      uint32_t key = 0, val = 0;

      bpf_map_update_elem(&map0, &key, &val,
          /*flags=*/0);
      return 0;
}
```

We prevent this by checking the set of maps updated by the program we're attaching. If the map we're attaching the program to is already in that set, then an infinite loop could form, and thus the link creation should fail. Furthermore, each map associated with the program may be traced by a set of programs. So this routine has to recursively expand every map associated with the program. Doing this catches both the direct and indirect infinite loops outlined above.

When a program is attached, we add it to an array of linked lists in `struct bpf_map`:

```
struct bpf_map_trace_prog {
      struct list_head list;
      struct bpf_prog *prog;
      struct rcu_head rcu;
};

struct bpf_map_trace_progs {
      struct bpf_map_trace_prog __rcu
          progs[MAX_BPF_MAP_TRACE_TYPE];
      u32 length[MAX_BPF_MAP_TRACE_TYPE];
      struct mutex mutex; /* protects writes
          to progs, length */

struct bpf_map {
...
      struct bpf_map_trace_progs *trace_progs;
...
};
```

There is one list of tracing programs per type of traceable map update.

C. Helper functions

Tracing programs are executed by programs with helper functions. These helper functions have the exact same function signature as their corresponding map update function.

```
BPF_CALL_4(bpf_map_trace_update_elem, struct
    bpf_map *, map,
        void *, key, void *, value, u64, flags)
{
     bpf_trace_map_update_elem(map, key,
         value, flags);
     return 0;
}

const struct bpf_func_proto
    bpf_map_trace_update_elem_proto = {
     .func        = bpf_map_trace_update_elem,
     .ret_type    = RET_VOID,
     .arg1_type   = ARG_CONST_MAP_PTR,
      .arg2_type   = ARG_PTR_TO_MAP_KEY,
      .arg3_type   = ARG_PTR_TO_MAP_VALUE,
      .arg4_type   = ARG_ANYTHING,
};

BPF_CALL_2(bpf_map_trace_delete_elem, struct
    bpf_map *, map, void *, key)
{
      bpf_trace_map_delete_elem(map, key);
      return 0;
}

const struct bpf_func_proto
    bpf_map_trace_delete_elem_proto = {
      .func        = bpf_map_trace_delete_elem,
      .ret_type    = RET_VOID,
      .arg1_type   = ARG_CONST_MAP_PTR,
      .arg2_type   = ARG_PTR_TO_MAP_KEY,
};
```

D. Verifier extensions

We instrument the verifier to mechanically insert the aforementioned helper calls after `bpf_map_update_elem` and `bpf_map_delete_elem`. The return value from the tracing program is thrown away; error handling is assumed to be the responsibility of the tracing program.

```
int get_map_tracing_patchlet(
            void *map_func,
            void *map_trace_func,
            const int nregs,
            struct bpf_prog *prog,
            struct bpf_insn *insn_buf,
            int *extra_stack)
{
      const int stack_offset = -1 * (int16_t)
          prog->aux->stack_depth;
      const int reg_size_bytes = 8;
      int cnt = 0, i;

      /* push args onto the stack so that we
          can invoke the tracer after */
      for (i = 0; i < nregs; i++)
            insn_buf[cnt++] = BPF_STX_MEM(
                        BPF_DW, BPF_REG_FP,
                        BPF_REG_1 + i,
                        stack_offset - (i +
                            1) *
                            reg_size_bytes);

      insn_buf[cnt++] =
          BPF_EMIT_CALL(BPF_CAST_CALL(map_func));

      for (i = 0; i < nregs; i++)
            insn_buf[cnt++] = BPF_LDX_MEM(
                        BPF_DW, BPF_REG_1 + i,
                        BPF_REG_FP,
                        stack_offset - (i +
                            1) *
                            reg_size_bytes);

      /* save return code from map update */
      insn_buf[cnt++] = BPF_STX_MEM(BPF_DW,
          BPF_REG_FP, BPF_REG_0,
```

```
                              stack_offset -
                                  reg_size_bytes);

        /* invoke tracing helper */
        insn_buf[cnt++] =
            BPF_EMIT_CALL(BPF_CAST_CALL(map_trace_func));

        /* restore return code from map update
            */
        insn_buf[cnt++] = BPF_LDX_MEM(BPF_DW,
            BPF_REG_0, BPF_REG_FP,
                              stack_offset -
                                  reg_size_bytes);

        *extra_stack = max_t(int, *extra_stack,
            nregs * reg_size_bytes);
        return cnt;
}
```

V. Open questions and future work

It is not clear whether inserting map tracing helper calls in the verifier is acceptable for all users. We may have to introduce flags to control which (if any) helpers are inserted. It's also not clear whether the verifier is the best layer in the stack at which we can introduce these calls. E.g. we could insert them at the source code level so that they might benefit from compiler optimizations.

So far we have only prototyped tracing `map_update_elem` and `map_delete_elem`. Many, many applications use `map_lookup_elem` or analogous local storage APIs to perform map updates. We would like to trace these. Logically, this means chasing down the last modification to the returned pointer and calling a helper. Doing this in the verifier seems feasible, but it begs the question whether the verifier is the right layer in the stack to do this kind of transformation.

When running tracing programs, we unconditionally execute all of them and ignore errors. It's unclear whether this poses a problem for a large-scale production-grade state migration service, or what error handling should be done.

The infinite loop detection outlined earlier in this document may not be exhaustive.