Google Cloud

BPF Map Tracing
Hot updates of stateful programs

20 Sep 2021

This presentation is both an RFC and a status update.

The ideas presented here have been prototyped and could be sent upstream soon.

There are open questions I would like feedback on.

Table of Contents

01

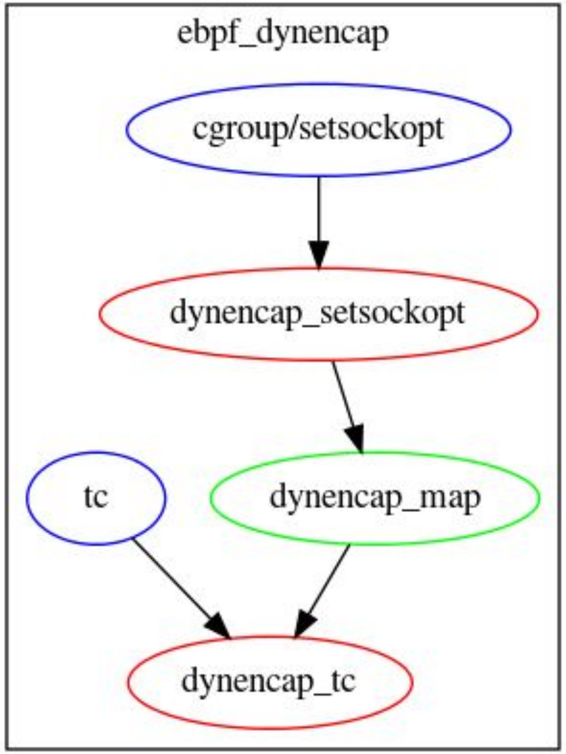
Motivation
An un-upgradable stateful program

ebpf_dynencap

ebpf_dynencap is a set of BPF programs developed at Google. At the highest level, it allows applications to dynamically specify what src and dst IPs to use when encapsulating the packets on a socket.

To opt in a socket, programs invoke a cgroup/setsockopt program. A TC program performs the encapsulation.

It uses three maps.



ebpf_dynencap

dynencap_map is an SK_STORAGE_MAP that records the IPs to use when encapsulating the packets from a socket.

It is populated a cgroup/setsockopt program, via bpf_sk_storage_get().

It is consumed by the TC program.

Losing the data in this map means that packets do not get encapsulated, breaking the contract with usermode.

ebpf_dynencap

syn_encap_map is an LRU_HASH. It stores the outer IPs of encapsulated SYN packets. The SYNACK response to such an encapsulated SYN is encapsulated with the same IPs.

It is mutated by a cgroup_skb/ingress program, via bpf_map_update_elem() and bpf_map_delete_elem().

It is consumed by the TC program.

Losing the data in this map causes the TC program to drop the SYNACK. This is not ideal, but not fatal.

ebpf_dynencap

reflection_status is an PERCPU_HASH which simply counts the number of errors of a certain type.

It is mutated by a cgroup_skb/ingress, via bpf_map_lookup_elem() and bpf_map_update_elem().

It is consumed by a userspace app.

It is informational, not critical for policy enforcement.

ebpf_dynencap

Symbol name	Type	Impact of state loss	Mutation helpers
dynencap_map	SK_STORAGE	contract breaks	sk_storage_get()
syn_encap_map	LRU_HASH	performance / availability hit	update_elem(), delete_elem()
reflection_status	PERCPU_HASH	doesn't matter	lookup_elem(), update_elem()

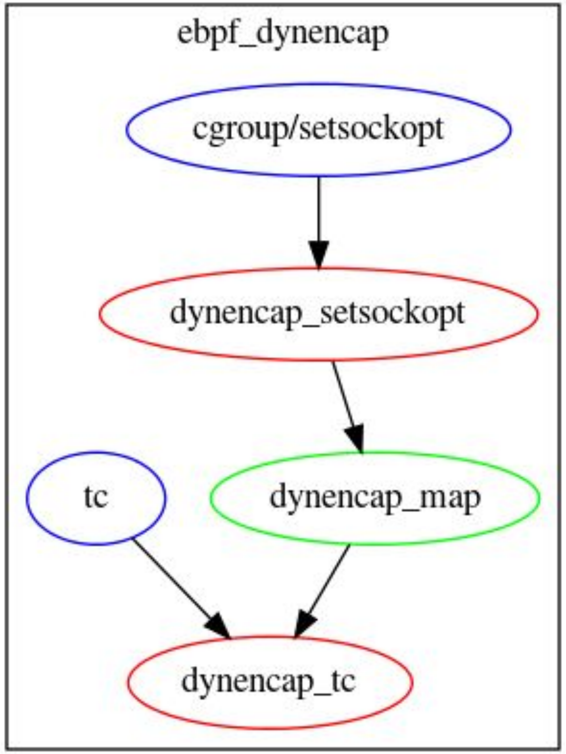
ebpf_dynencap

Symbol name	Type	Impact of state loss	Mutation helpers
dynencap_map	SK_STORAGE	contract breaks	sk_storage_get()
syn_encap_map	LRU_HASH	performance / availability hit	update_elem(), delete_elem()
reflection_status	PERCPU_HASH	doesn't matter	lookup_elem(), update_elem()

ebpf_dynencap simplified

Let’s consider how we might upgrade the critical subset of dynencap.

We have two programs, each attached to an event, and one map.



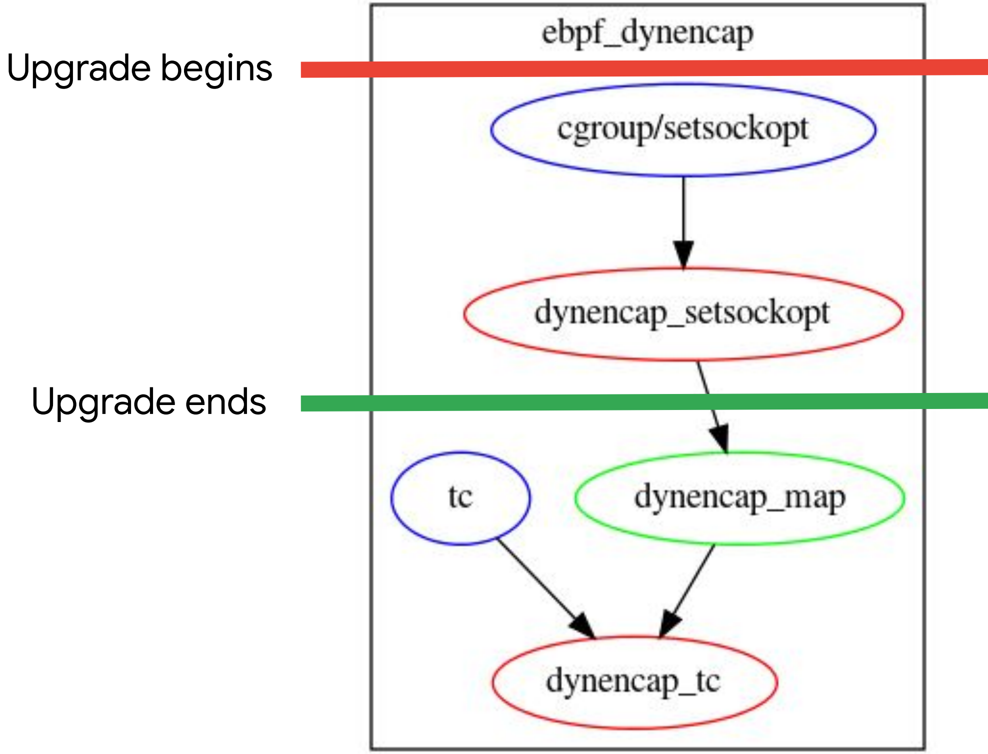
ebpf_dynencap simplified

Suppose we upgrade by loading a new copy of each program and map. We do a bulk copy of the data in dynencap_map, then swap each program.

By pure chance, an application does a setsockopt() in the middle of this operation, after we've finished the bulk copy of dynencap_map, but before we've swapped the programs.

We then swap our programs, but dynencap_map is missing the entry for that socket.

Oops! We've broken our contract with usermode.



Ramifications

First-order problems:

- We cannot upgrade our programs.
- We cannot roll back our agent arbitrarily, since it would need to know how to speak to newer programs.

Second-order problems:

- Before rolling out new programs, we need to roll out a new agent which can speak to them. This allows us to roll back our agent, but it adds latency to our releases.
- Our developers need to keep all of this complexity in their heads.

02

Abstract model

The problem, and a solution

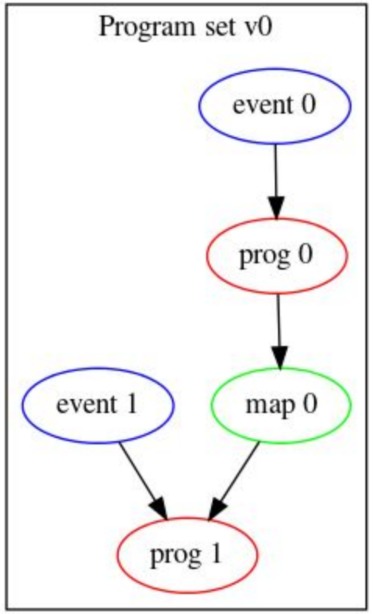
A model application

Let’s consider how to upgrade this simplified program set.

It is comprised of two programs, each attached to an event, and one map.

The final program makes a decision using the state in the map.

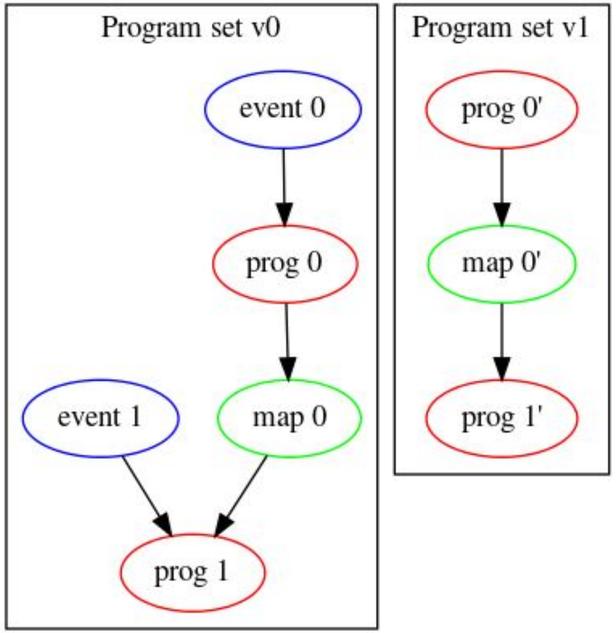
Let’s upgrade it.


$$\begin{tikzcd}
 & \text{event 0} \arrow[d] \\
 & \text{prog 0} \arrow[d] \\
\text{event 1} \arrow[dr] & \text{map 0} \arrow[d] \\
 & \text{prog 1}
\end{tikzcd}$$

Step 1: Load the new version

First we can load the new version of the program set.

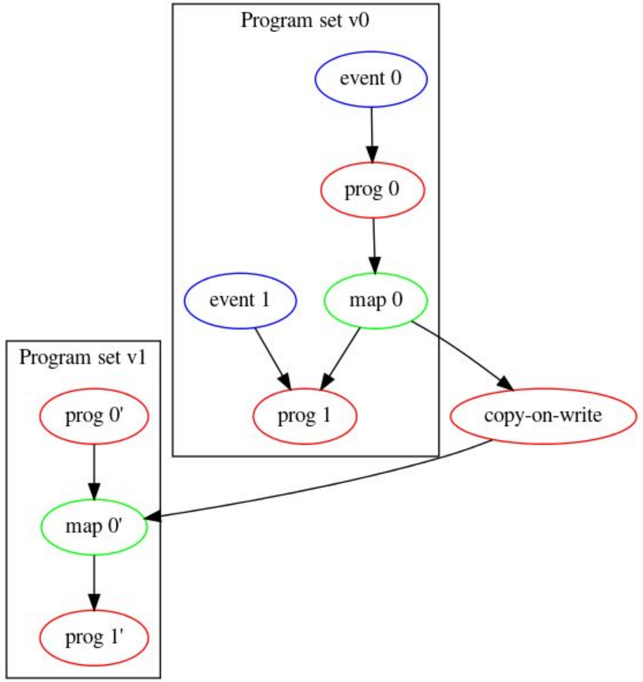
Since we haven't attached our programs to any events, this has no effect.



Step 2: Attach a copy-on-write handler to the map

Attach a new kind of tracing program to the old map. Any updates to the map invoke this program.

In this case, the program migrates the format of the data from old to new, then writes it to the new map.

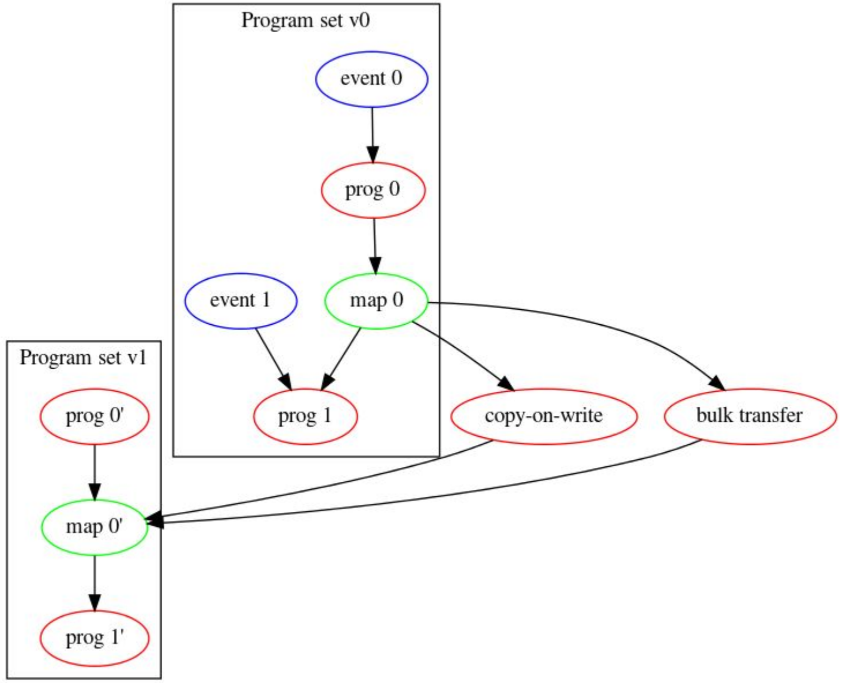


Step 3: Perform a bulk transfer of state

Now we can copy all of the state from the old map to the new one.

Suppose that the bulk transfer program is in the middle of migrating the last element of a large array when an earlier element is updated. Normally this change would not propagate to the new map. Because of the copy-on-write handler, the change does propagate.

To prevent the copy-on-write handler and bulk-transfer programs from clashing, we can use a spinlock.

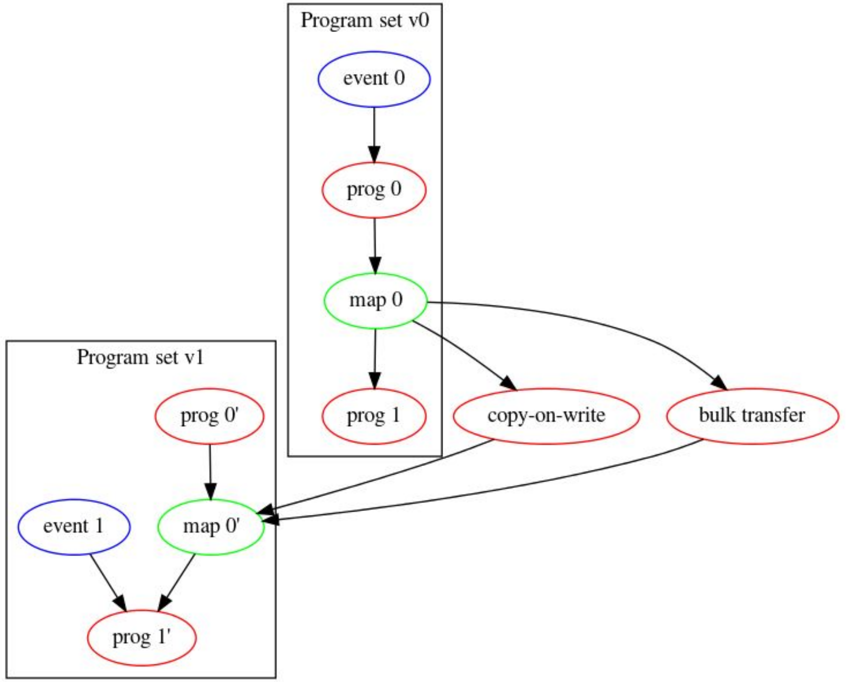


Step 4: Swap programs one by one

At this point we can start swapping programs one by one.

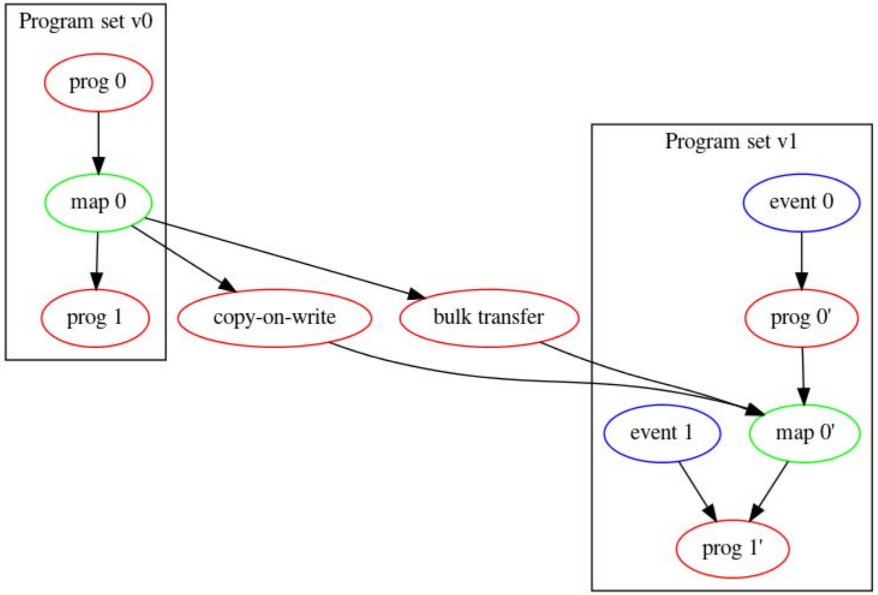
We have to do it in an order determined by the programs' data dependencies (simple topological sort).

If prog 0 fires while we're in this state, prog 1' has all of the state it needs due to either the copy-on-write or the bulk transfer programs.



Step 4: Swap programs one by one

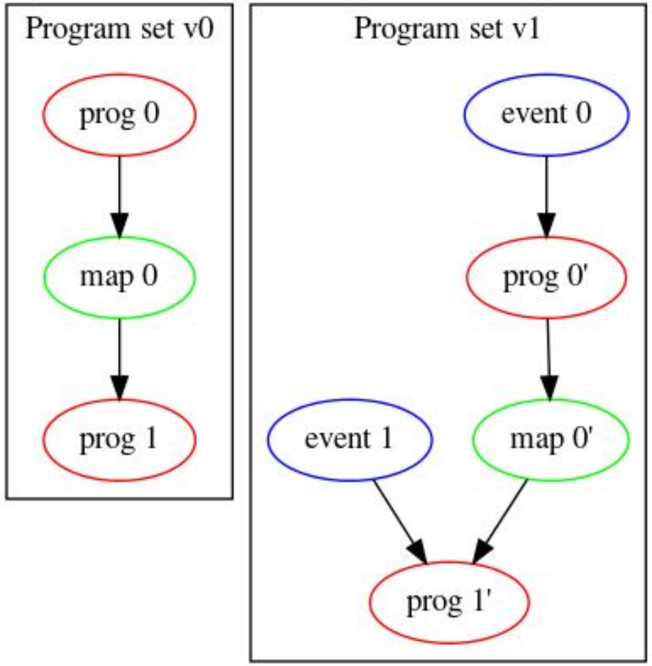
At this point we're essentially done. All that remains is to unload the migration and old programs.



Step 5: Unload migration programs

We have upgraded our stateful application without breaking a data dependency at any point in time, or making any assumptions about the nature of the change to the map.

The map's capacity, type, and data layout could all change arbitrarily.



03

Proposed solution
Source code

A toy program

```
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;
}
```

Program context

```
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);
};
```

Link API changes

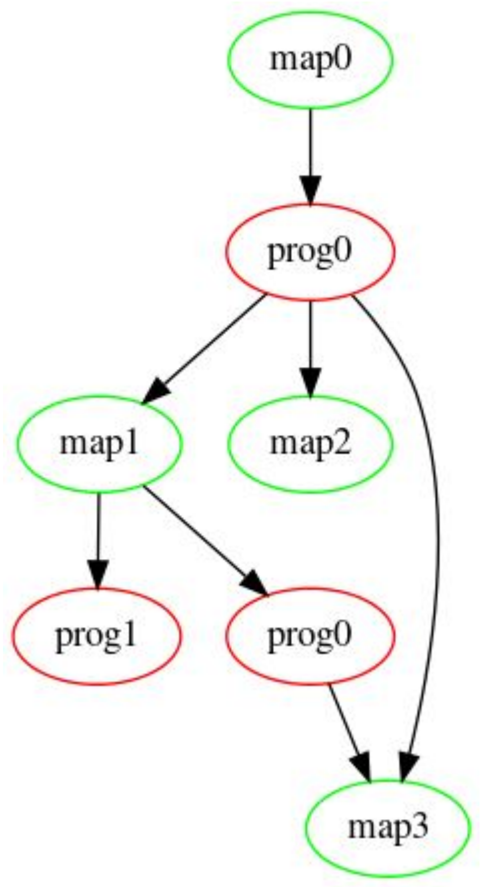
```
enum bpf_map_trace_type {
  BPF_MAP_TRACE_UPDATE_ELEM = 0,
  BPF_MAP_TRACE_DELETE_ELEM = 1,
  MAX_BPF_MAP_TRACE_TYPE,
};

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

struct { /* struct used by BPF_LINK_CREATE command */
...
  union {
    struct {
      __aligned_u64   map_trace_info;
      __u32           map_trace_info_len;
    };
...
};
```

Implications

Because we can attach a **list** of programs to a map, and a program can update multiple maps, updating one map can trigger a cascade of programs and maps.

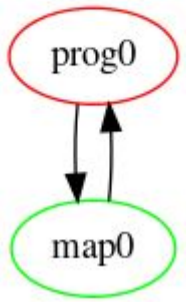
The cascade can take the form of a graph.

If this graph has a loop, we're in trouble.



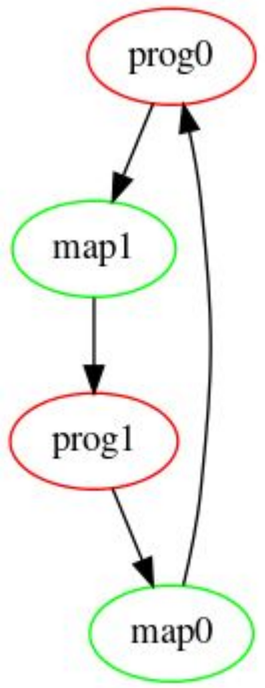
Infinite loops - direct

```
/* 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;
}
```



Infinite loops - indirect

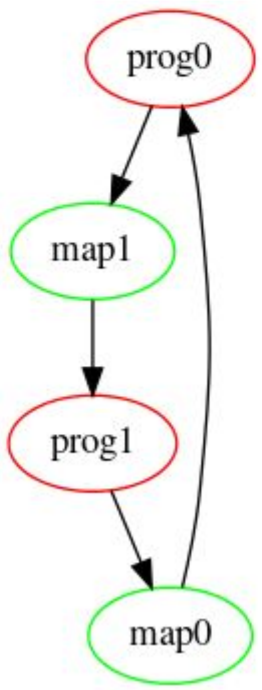
```
SEC("map_trace/map0/UPDATE_ELEM")
int tracer0(struct bpf_map_trace_ctx__update_elem *ctx) {
  uint32_t key = 0, val = 0;
  bpf_map_update_elem(&map1, &key, &val, /*flags=*/0);
  return 0;
}

/* 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;
}
```



Infinite loop detection

At link creation time, we detect infinite loops with a simple depth-first search on this graph.

```
def would_loop(tracing_prog, traced_map):
    for each map in tracing_prog.used_maps:
        if map == traced_map:
            return True;
        for each program attached to this map:
            if (would_loop(program, traced_map)):
                return True;
    return False;
```



struct bpf_map changes

```
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; <= No mutex!
  ...
};
```

Tracing helper functions

```
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) {
  ...
```

verifier changes

```
/* 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;
```

04

Open questions

Supporting bpf_map_lookup_elem() and local storage

Many applications use bpf_map_lookup_elem() to get a pointer to a map entry, then modify the entry through that pointer.

So far, our approach revolves around mechanically tracing helper calls. This doesn't work for pointer based updates because of situations like this:

```
int* x = bpf_map_lookup_elem(...);

bpf_map_trace_lookup_elem(...);

*x++; /* The tracer never saw this! */
```

Supporting bpf_map_lookup_elem() and local storage

After some light discussion internally, we concluded that invoking tracer calls at program exit time may be the best solution.

```
int* x = bpf_map_lookup_elem(...);

*x++;

...

bpf_map_trace_lookup_elem(x);

return 0;
```

This transformation would be done at the source code level to avoid issues with register allocation.

Verifier-inserted tracing calls

Our prototype mechanically expands each bpf_map_update_elem(), appending a bpf_map_trace_update_elem() call right after it.

Should it be done conditionally, according to a flag in BPF_PROG_LOAD?

Should it be done at the source code level instead of at the byte code?

What interesting use cases have we not thought of?

We conceived of the map tracing facility to ease the issues around upgrading stateful programs, and to allow these programs to couple tightly with applications.

What else is this good for?

05

Summary

Summary

We hope to solve the problem of un-upgradable stateful programs by temporarily imbuing maps with copy-on-write semantics.

A set of helper functions are defined which can invoke these programs. They may be mechanically added to programs either in the verifier or in source code.

Open questions:

1. How should we support pointer-based updates, e.g. local storage?
2. Is unconditionally inserting helper calls in the verifier acceptable?
3. What other problems might this solve?

Thank you.

Google Cloud