Exceptions in BPF

Kumar Kartikeya Dwivedi
<memxor@gmail.com>

EPFL

Agenda

1. Introduction
2. Example, Use cases
3. Design
4. Future Work
5. Questions

Introduction

- An exception aborts the program immediately and returns control back to the kernel.
- Similar to `panic!()` or `std::terminate` (in Rust and C++).
- `bpf_throw(cookie)` kfunc is used to generate an exception.
- By default, cookie is the return value when an exception is thrown.
- An ‘exception callback’ can be installed to take the passed cookie value as an input, and return a different return value back to the kernel.

- More importantly, the verifier will not follow a program path further once it encounters a `bpf_throw` call.

Example

```
SEC(“tc”)
__exception_cb(exception_callback)
int prog(struct __sk_buff *ctx)
{
    if (ctx->data + 4 > ctx->data_end)
        bpf_throw(TC_ACT_SHOT);
    …
    return TC_ACT_OK;
}
```

Use cases - Assertions

- bpf_assert_eq, bpf_assert_lt, bpf_assert_gt, bpf_assert_le, bpf_assert_ge, bpf_assert_range
- Update the verifier's knowledge about a certain scalar value.

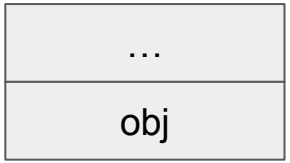
  Literally this sequence:

  ```
  if (reg <op> rhs) goto pc+2

  r1 = cookie

  call bpf_throw
  ```

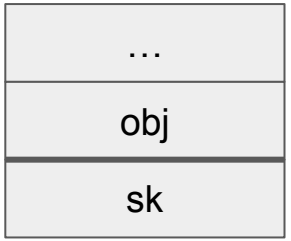
- Therefore, any improvements to the verifier's ordinary range and value tracking code will apply to assertions as well.

Design - High level

- At each program point which can throw (i.e. a `bpf_throw` kfunc call), for all frames in the call chain, prepare *frame descriptors*.
- These descriptors describe the state of the stack and registers (r6 - r9) for each frame.
- When unwinding, any resources tied to these stack slots or registers will be cleaned up before unwinding the frames.
- Finally, after all stack frames are unwound, the exception callback, if any, is invoked with the cookie value supplied to the `bpf_throw` call.
- Thus, the program is immediately aborted, and all resources are released through frame by frame unwinding.

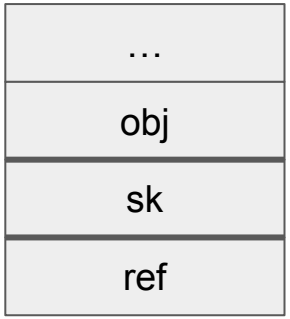



```
call bpf_obj_new
*(u64 *)(r10 - 16) = r0
...
call pc+X
```




```
call bpf_obj_new
*(u64 *)(r10 - 16) = r0
...
call pc+X
```

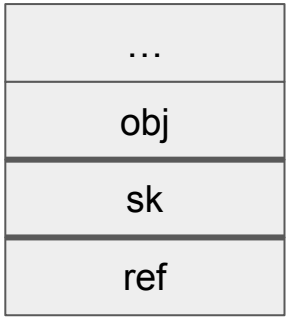
```
call bpf_sk_lookup
*(u64 *)(r10 - 8) = r0
...
call pc+X
```




```
call bpf_obj_new
*(u64 *)(r10 - 16) = r0
...
call pc+X
```

```
call bpf_sk_lookup
*(u64 *)(r10 - 8) = r0
...
call pc+X
```

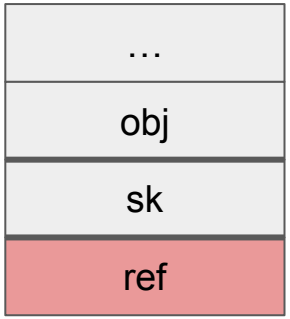
```
call bpf_refcount_acquire
*(u64 *)(r10 - 8) = r0
...
r1 = cookie
call bpf_throw
```




```
call bpf_obj_new
*(u64 *)(r10 - 16) = r0
...
call pc+X
```

```
call bpf_sk_lookup
*(u64 *)(r10 - 8) = r0
...
call pc+X
```

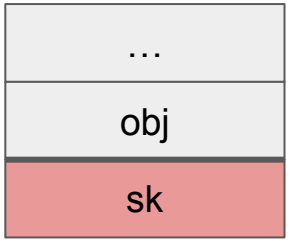
```
call bpf_refcount_acquire
*(u64 *)(r10 - 8) = r0
...
r1 = cookie
call bpf_throw
```




```
call bpf_obj_new
*(u64 *)(r10 - 16) = r0
...
call pc+X
```

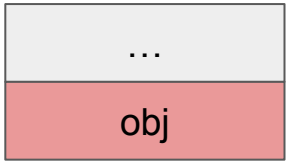
```
call bpf_sk_lookup
*(u64 *)(r10 - 8) = r0
...
call pc+X
```

```
call bpf_refcount_acquire
*(u64 *)(r10 - 8) = r0
...
r1 = cookie
call bpf_throw
```




```
call bpf_obj_new
*(u64 *)(r10 - 16) = r0
...
call pc+X
```

```
call bpf_sk_lookup
*(u64 *)(r10 - 8) = r0
...
call pc+X
```




```
call bpf_obj_new
*(u64 *)(r10 - 16) = r0
...
call pc+X
```

Resource cleanup on unwinding

- In frame 0 and frame 1, we have a frame descriptor for each call instruction that may potentially throw.
- We do a pass before verification to analyze all possible call chains and identify subprograms that may need generation of frame descriptors.

- Frame descriptor information comes *directly* from the symbolic execution state of a frame at a program point.
- The information is then looked up using the program counter at runtime.

Pre-verification pass

- Why a pre-pass? Global subprograms are not descended during symbolic execution of the main program. Moreover, global subprograms are verified before the main program.
- We would need to know whether frame descriptors need to be generated for a subprogram at some point, without knowing whether its callee can throw.
- main -> static subprog 1 -> global subprog 1 -> static subprog 2 -> bpf_throw
- During verification, verifier sees:
  - global subprog -> static subprog 2 -> bpf_throw
  - main -> static subprog 1 -> global subprog 1
- We need to know global subprog 1 will cause an exception to be thrown, to generate frame descriptor for main and static subprog 1.

Future Work - Catching exceptions within frames

- Right now, we have a catch-all callback that can be installed.
- Invoked after complete unwinding is finished.
- Rather, could we have the equivalent of catch blocks per-frame?
- Perform user-defined cleanup logic which the verifier cannot do.
- Decide whether to continue throwing or stop exception propagation.

- First option: Dedicated compiler generated landing pads and metadata with program.
- Second option: __constructor, __destructor tags for program BTF types.
- This might allow us to integrate other BPF frontends in the future: Rust with catch_unwind, C++ with catch blocks.

Future Work

- Assertions just scratch the surface of what’s possible.
- Rather, exceptions allow you to do fundamentally rethink how the verifier enforces it’s safety properties.
- We can shed a lot of complexity in the verifier by offloading correctness burden to the runtime.
- This can be both “fast” and “correct”.
- Go back to the original goal: The verifier should only concern itself with protecting the kernel, not the BPF programmer from himself.
- Lately, we have been ‘tricked’ into doing the latter.

Cancellation

- Whether a program calls `bpf_throw` or not, how about we still generate frame descriptors?
- A bit expensive memory wise, but we gain:
- The ability to interrupt a running program*, and know what unwinding would require at this program point for all frames.
- Possibly steer execution to bpf_throw, which ends up aborting the program (regs->pc fixup).

* doesn’t work for NMI context programs.

Cancellation

- Once we have the ability to cancel a program:
- We have the ability to write loops which seemingly do not terminate for the verifier.
- Catch: These loops should be conformant to the `bpf_iter` style, i.e. the iterations must converge to a fixpoint, so that we can reason about program state at each iteration.
- Runtime cleanup only cares about kernel resources, so other data is irrelevant.

“Infinite” Loops

- Being able to pass through loops which seemingly never terminate is very powerful.
- Can express all kinds of things:
  - Iteration logic for user-managed linked lists
  - Spin loops for custom spin locks.
- Here, we rather enforce this by putting a limit on program execution, ‘cancelling’ it if it gets stuck.
- Depending on the configuration, the limits could vary. Some setups could tolerate programs running for 1ms, some for 5000ns. You need a worst case bound.
- Either do ‘cancellation’, or have a high bound on the loop and throw when breached.

Example

```
bpf_for (i, 0, BPF_MAX_LOOPS) {
     if (elem->next == NULL)     ← Linked list built out of array map nodes
          break;
     task = bpf_task_from_pid(elem->pid);
     if (!task)
          continue;
     …
     bpf_task_release(task);
     elem = elem->next;
}
```

Example

```
bpf_for (i, 0, BPF_MAX_LOOPS) {
     if (elem->next == NULL)      ← No guarantee of acyclicity
          break;
     task = bpf_task_from_pid(elem->pid);
     if (!task)
          continue;
     …
     bpf_task_release(task);
     elem = elem->next;
}
```

Example

```
bpf_for (i, 0, BPF_MAX_LOOPS) {  ← BPF_MAX_LOOPS is too big (8 million)
     if (elem->next == NULL)
          break;
     task = bpf_task_from_pid(elem->pid);
     if (!task)
          continue;
     …
     bpf_task_release(task);
     elem = elem->next;
}
```

Example

```
bpf_for (i, 0, BPF_MAX_LOOPS) {  ← We would rather ‘cancel’ the loop if stuck
     if (elem->next == NULL)
          break;
     task = bpf_task_from_pid(elem->pid);
     if (!task)
          continue;
     …
     bpf_task_release(task);
     elem = elem->next;
}
```

bpf_obj_new, Lists, RB-Trees

- Since we hand over *kernel memory* to the program, we need to ensure we get it back.
- Once it is used in a shared fashion across programs, we need to understand lifetime and ownership semantics.
- Unique ownership vs Shared ownership - The latter is difficult to reason about, and restrictive.
- Once this memory is used for an object part of a shared data structure, we need to understand synchronization invariants.
- Instead, we could remove all this complexity in favor of a simpler scheme.
    - Let programs build synchronization primitives.
    - Let programs build their own data structures and iteration logic.
    - All memory comes from a program managed “heap” (e.g. BPF array map, but something more dynamic) rather than kmalloc.

What have I tried so far?

- Spin locks (with multiple levels, and deadlock detection).
- Data structures (Hash maps, Linked Lists, RB-Trees, Skip List, QP-Trie).

- Exceptions (through cancellation) underpin the safety argument for all of these. They allow enforcing the termination property, without compromising program safety.
- Use maps as heap to manage memory.
- Unburden the verifier from reasoning from about concurrency, synchronization, liffetimes, and memory management.
- Only enforce safety properties: resource safety, memory safety, termination.

Questions?