Finding Double-Unlock Bugs with Shape-and-Effect Analysis

Anders Fischer-Nielsen

Why?

- The Linux kernel is everywhere (embedded devices, routers, TVs, phones, PCs & large servers)
- Bugs in the Linux kernel can therefore have a big impact on many devices

- Shared-memory concurrency and locks are used extensively in the source code of the kernel
  - Allows parallelization of subsystems while avoiding race conditions

- Static analysis of the Control Flow Graph (CFG) of the kernel allows detecting possible ressource manipulation errors

Double-Unlock Errors

- A thread holding a lock which then releases this lock more than once consecutively will result in **undefined behaviour**, according to the POSIX standard

> *"The results are undefined if the lock is not held by the calling thread. [...]*
> *The results are undefined if this function is called with an uninitialized thread spin lock."*

- Undefined behaviour at the kernel level possibly makes the operating system behave in unexpected ways
- A program depending on undefined behaviour might not break today, but could break in the future, since no assumptions can be made about undefined behaviour

- Detecting these errors allows developers to detect errors in their code, hopefully leading to safer programs --- provided the developers fix detected errors.

An Example

```c
#include "ubifs.h"

static void orphan_delete(struct ubifs_info *c, struct ubifs_orphan *orph)
{
    if (orph->del) {
        spin_unlock(&c->orphan_lock);
        return;
    }
    if (orph->cmt) {
        spin_unlock(&c->orphan_lock);
        return;
    }
}

void ubifs_delete_orphan(struct ubifs_info *c, ino_t inum)
{
    orphan_delete(c, orph);
    spin_unlock(&c->orphan_lock);
}
```

What I Wanted to Accomplish

- ***What is the performance of a double-unlock error checker prototype implementation when run on the source code of Linux kernel components?***

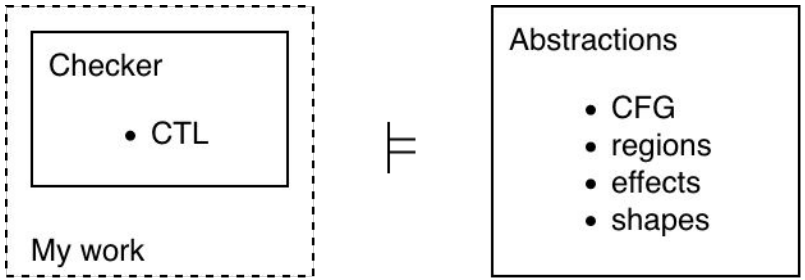
- Accomplished by:
  - Developing an error checker able to find these double-unlock errors
  - Analyzing the checker performance by running the implementation on confirmed double-unlock bugs found in the Linux kernel

Building Upon an Existing Implementation

- Building upon the shape-and-effect inference system described by Abal et. al. to develop a double-unlock checker relatively quickly
- Computational Tree Logic (CTL) is used by Abal et. al. to show which errors should be detected by a **double-lock** checker
- This specification can then be used to formulate a **double-unlock** checker
- CTL models program executions in time as tree-like structures where all paths can be the actual path executed at runtime

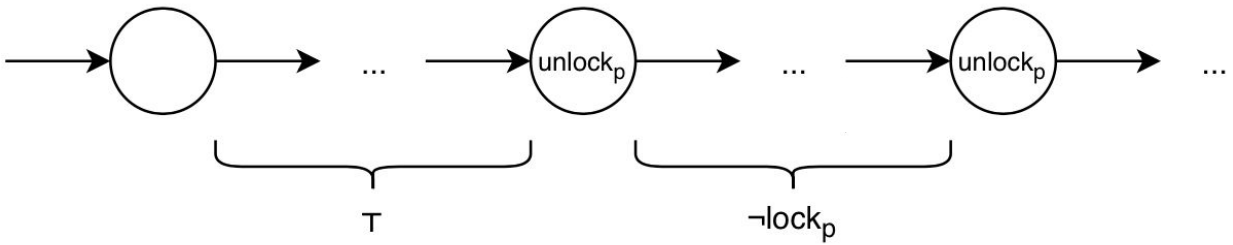
Scope

- Internal abstractions are merely used --- not modified --- in the project implementation



What We Want to Detect

$$\top \, EU \, (unlock_p \wedge EX \, (\neg lock_p \, EU \, unlock_p))$$



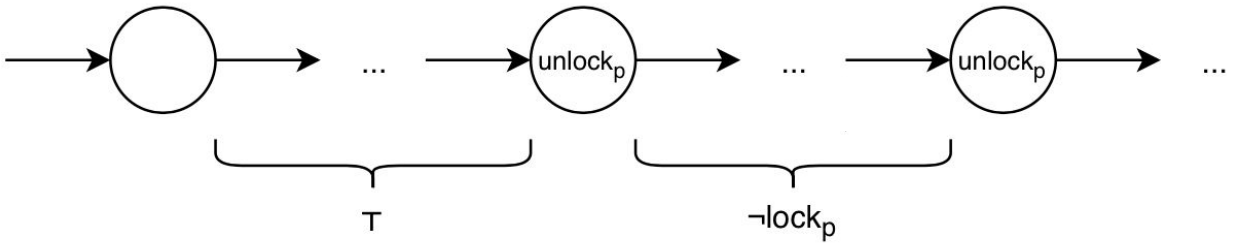
In other words; an unlock is performed on *p* without a lock on *p* being present before another unlock is performed on *p*, leading to a double-unlock

What We Want to Detect

$$\neg\, E\,U\,(unlock_p \wedge EX\,(\neg lock_p\; E\,U\; unlock_p))$$

$$E\,[\top\; U\; unlock_p \wedge EX\; E[\neg lock_p\; U\; unlock_p]]$$

$$EF\; unlock_p \wedge EX\; E[\neg lock_p\; U\; unlock_p]$$



In other words; an unlock is performed on *p* without a lock on *p* being present before another unlock is performed on *p*, leading to a double-unlock

Implementation

- EBA framework requires implementing a `Spec` OCaml module. If all checks return an Option value, an error has been found.
- The checks found in the `Spec` signature have been implemented using existing EBA infrastructure & helper functions
- Reachability of a node matching the checker definitions is tested internally in the EBA infrastructure

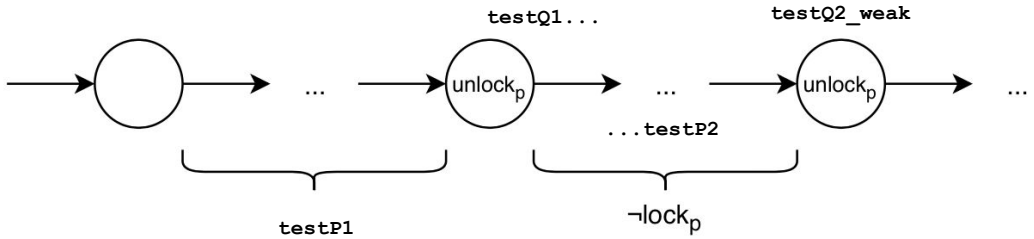
```
module type Spec = sig
   (** A name to identify the checker *)
   val name : string

   (** Checker's internal state, eg. memory regions to track. *)
   type st
   (** Selects initial contexts *)
   val select : AFile.t -> Cil.fundec -> shape scheme -> AFun.t -> st L.t
   (** Flags steps of interest for triaging. *)
   val trace : st -> Effects.t -> bool
   (** Tests *)
   val testP1 : st -> step -> st option
   val testQ1 : st -> step -> st option
   val testP2 : st -> step -> st option
   (** Q2 = P2 /\ Q2-weak *)
   val testQ2_weak : st -> step -> st option

   (** Bug data *)
   type bug
   val bug_of_st : st -> bug
   val doc_of_report : fn:Cil.varinfo -> bug -> loc1:Cil.location -> loc2:Cil.location -> trace:path -> PP.doc
end
```

```
module type Spec = sig
  (** A name to identify the checker *)
  val name : string

  (** Checker's internal state, eg. memory regions to track. *)
  type st
  (** Selects initial contexts *)
  val select : AFile.t -> Cil.fundec -> shape scheme -> AFun.t -> st L.t
  (** Flags steps of interest for triaging. *)
  val trace : st -> Effects.t -> bool
  (** Tests *)
  val testP1 : st -> step -> st option
  val testQ1 : st -> step -> st option
  val testP2 : st -> step -> st option
  (** Q2 = P2 /\ Q2-weak *)
  val testQ2_weak : st -> step -> st option

  (** Bug data *)
  type bug
  val bug_of_st : st -> bug
  val doc_of_report : fn:Cil.varinfo -> bug -> loc1:Cil.location -> loc2:Cil.location -> trace:path -> PP.doc
end
```

These functions model parts of the CTL specification

Implementation

- **`testP1`** - Lifts a given CFG node to an Option type
- **`testQ1`** - Checks whether an unlock is performed without a lock in a step
- **`testP2`** - Checks that no lock is performed in a given step
- **`testQ2_weak`** - Checks whether an unlock is present in a step



Implementation

https://github.com/IagoAbal/eba/pull/5

```
let testQ1 st step =
      let unlocks_and_not_locks r ef =
          let unlocks r ef :bool = E.(mem (unlocks ~r) ef) in
          unlocks r ef && not_locks r ef in

      unlocks_and_not_locks st.reg step.effs =>?
          let unlock_opt = find_unlock_object step in
          let krs = Option.Infix.(
              (unlock_opt >>= Lenv.kregions_of st.fna)
              |? Regions.empty
          ) in
          {st with unlock = unlock_opt; kreg = krs}
```

```
let testP2 st step =
      let res = not_locks st.reg
step.effs =>? st in
      res

   let testQ2_weak st step =
      let must_unlock = E.(mem (unlocks
~r:st.reg) step.effs) in
      if must_unlock
      then Some st
      else None
```

Implementation Caveats

- CTL formulations do not always match implementation in existing checkers
- The implementation of reachability checking in EBA is hard to reason about
- Seemingly sound checks in functions do not always give expected results
    - Due to parsing and may/must handling in EBA

More on this later...

Results

File	Status
drivers/block/drbd/drbd_main.c	Detected
fs/ubifs/orphan.c	Detected
drivers/gpu/drm/nouveau/nouveau_svm.c	Undetected
fs/btrfs/file.c	Undetected
drivers/staging/wilc1000/wilc_wlan.c	Detected
drivers/staging/kpc2000/kpc_dma/fileops.c	Detected
fs/nfs/client.c	Undetected
fs/btrfs/file.c	Undetected
drivers/media/dvb-core/dvbdev.c	Stack overflow
mm/memory_hotplug.c	Undetected
sound/soc/codecs/pcm512x.c	Detected
drivers/target/target_core_user.c	Uncaught exception
drivers/rpmsg/qcom_smd.c	Compilation error (GCC)
drivers/scsi/aacraid/commsup.c	Compilation error (GCC)
drivers/staging/rtl8188eu/os_dep/usb_intf.c	Compilation error (GCC)
block/blk-cgroup.c	Compilation error (GCC)

Technically all of these result in an EBA `ParseError` without mitigations

Results

- EBA abstractions do not support all lock types found in the Linux kernel

```
if Opts.all_lock_types()
    then begin
        add_axiom tbl ax_mutex_lock;
        add_axiom tbl ax_mutex_lock_nested;
        add_axiom tbl ax_mutex_lock_interruptible_nested;
        add_axiom tbl ax_mutex_unlock;
        add_axiom tbl ax_raw_read_lock;
        add_axiom tbl ax_raw_read_unlock;
        add_axiom tbl ax__raw_read_unlock;
    end
```

- EBA parsing does not support all C constructs found in the Linux kernel
- EBA implementation of the CTL subset does not allow expressing desired checks

Where to Go From Here?

- Extending capabilities of EBA framework
  - Develop more checkers
  - Supporting lock types
  - Extend C parsing

- Reimplementing existing and future checkers as monitor state machines
  - Existing CTL implementation of EBA might limits expressiveness in checkers (e.g. may/must)
  - Gives greater “expressibility” of checker specifications (e.g. `if/else if`)
- Reachability of EBA can be modelled as state machines
- Current CTL can be modelled as state machines