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

Anders Fischer-Nielsen

December 16th 2019

Contents

Todo list

1 Introduction

The Linux kernel supports a vast array of computer architectures and runs on a multitude of devices from personal computers to servers and embedded devices from everything from wireless access points to smart TVs, smartphones and refrigerators. Errors in the Linux kernel therefore affect a multitude of devices and therefore have a potential significant negative impact.

An important aspect of kernel programming is management and manipulation of resources, be it devices, file handles, memory blocks, and locks. Locks are used extensively in the C source code of the Linux kernel in order to allow parallelization of subsystems within the kernel while at the same time avoiding race conditions. Static analysers allow detection of errors in the C source code of the Linux kernel by reasoning about this resource manipulation. A control flow graph can be found for the components of the kernel, which can then in turn be statically analysed to detect possible ressource manipulation errors.

One such resource manipulation error is a *double unlock* error. A thread holding a lock and then releasing this lock more than once will result in undefined behaviour, according to the POSIX standard. This standard is an attempt to generate a standard version of UNIX to facilitate application portability and defines how C constructs should be implemented by UNIX OS vendors. The `pthread.h` file defines the spinlock constructs which are used in the Linux kernel and the accompanying specification is of note here, since this file describes how the structs found in the header should behave. The section describing `pthread_spin_unlock` defines the behaviour of the `spin_unlock` unlocking operation of a spinlock observed in the kernel code, and is defined as:

"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." [3]

If a thread wanting to unlock a lock does not currently hold that lock, the lock has either been unlocked already or has never been locked. This will in both instances lead to undefined behaviour at a kernel level, possibly making the operating system behave in unexpected ways.

Undefined behaviour is problematic since a program depending on undefined behaviour might not break today, but could break in the future. E.g. if a program depending on undefined behaviour has been implemented on an assumption that the output of the undefined behaviour will always be within a certain interval, but due to the nature of undefined behaviour this changes in a compiler update, the program suddenly breaks, leading to a software panic. Undefined behaviour is exactly that - *undefined* - and assumptions can therefore not be made about its output.

Developing a way to detect such errors is desirable in order to allow developers to detect errors in their code, leading to safer programs. An implementation for detecting such errors can furthermore be used as a tool for evaluating the performance of other code analysis tools, such as a tool for automated software repair. Patching the inverse error, a double lock error, could potentially introduce an unwanted double unlock error in the code. An implementation of a double unlock checker could serve as a correctness evaluation tool, and as a test harness for automated double lock repair.

Developing an error checker able to find such double-unlock errors and, furthermore, analyzing its performance on Linux kernel components will answer the question:

”What are the results of running a double-unlock error checker implementation on Linux kernel components?”

This report will detail our approach for detecting such double unlock errors based on previous work, give an overview of the concrete implementation of our approach and finally evaluate the results of validating an assembled set of files known to have bugs in the Linux kernel components.

2 Background

The shape-and-effect inference system described by Abal et. al. [1] enables

> "[...] efficient and scalable inter-procedural reasoning about resource manipulation"

Abal et. al. describe a method for detecting double-lock bugs in the kernel source code using the EBA analyzer.

Extending the approach used by Iago et. al. allows emplying the same shape and effect analysis concepts in addition to the CFL problem formulation used in their approach. Furthermore, extending the implementation of their tool allows building an error checker which is already able to run on the Linux kernel components.

The Linux kernel component source code can be analyzed using existing static analysis tools in order to find a Control Flow Graph (CFG) of a given component implementation. The control flow graph of a given Linux kernel component shows the possible execution paths of the component. These control flow graphs can be formulated in Computational Tree Logic, which in turn allows describing desirable or undesirable execution paths.

Iago et. al. formulate the double *lock* error as using Computational Tree Logic (CTL) as follows:

$$\top \text{ EU } (lock_\rho \wedge \text{ EX } (\neg unlock_\rho \text{ EU } lock_\rho))$$

Computation tree logic is in a class of temporal logics that includes linear temporal logic (LTL), modeling program paths in time as a tree-like structure where all paths can be the actual path executed at runtime. The specifics of the CTL formulation is not the focus of this project, instead CTL notation is merely used to formulate which errors are desired found in the Linux kernel components. CTL be used to model all possible executions of a program avoid some undesirable condition - in this case a double-unlock error.

p in the above specifies that we are interested in finding a double lock locking the same memory region, since this is where an infinite spinlock would occur.
This CTL formulation can be visualized as follows:

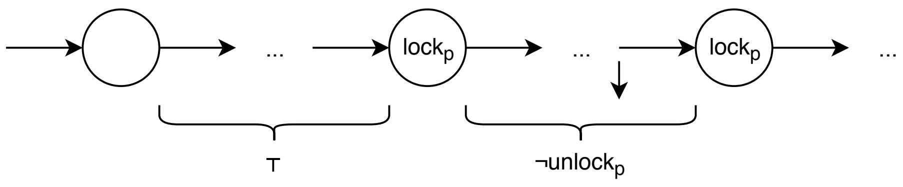

Figure 1: The CTL form of a double lock error visualized.

The inverse problem - the case of two unlocks being present with no locks in between them - can therefore be formulated in CTL as:

$$\top \text{ EU } (unlock_\rho \wedge \text{ EX } (\neg lock_\rho \text{ EU } unlock_\rho))$$

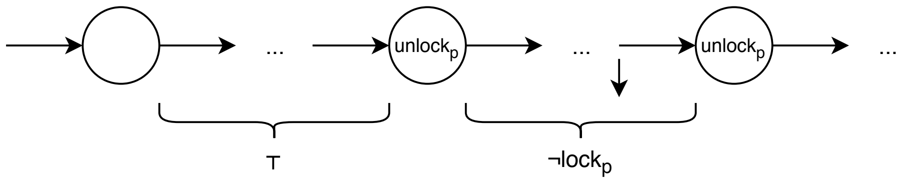

Figure 2: The CTL form of a double unlock error visualized.

p in the above again specifies that we are interested in finding a double unlock locking the same memory region, since this is where the undefined behaviour would occur.
Visualized, this looks as follows:
The existing implementation of the EBA analyzer explores the CFG of the Linux kernel components while checking in order to validate that a given undesired execution path is not present in the source code. If such an execution path is found, a bug is possibly present in the analyzed component.

The implementation of the EBA analyzer uses a concept of *may lock* and *must lock* internally similar to the concept described in [2]. A lock within an `if` statement on a dynamic value is represented as a *may lock*, since it is uncertain whether the `if` statement will evaluate to `true` at runtime. On the other hand, a *must lock* is an explicit lock within the code which will always be executed. This concept of *may* and *must* also applies to the double unlock problem, where a *may unlock* is an unlock in an execution branch which might not be executed at runtime. A *must unlock* is an explicit unlock where a lock will always be unlocked at runtime.

The implementation of the EBA analyzer supports implementing separate error checkers in a plug-and-play nature. This allows implementing a double unlock checker as a separate file in the implementation source code and making this available to the user(s) of the analyzer using a command-line parameter. The implementation of our approach has been added to the EBA code base in this manner, which the following section will detail.

3 Finding Double Unlock Bugs

4 Results

Given the open source nature of the Linux kernel, the project sees contributions from many developers. Contributions are integrated into the code base using a Distributed Version Control System (DVCS), namely *git*.

The log of contributions, *commits*, can be queried in order to find patches to errors discovered in the code base. Here we are interested in identifying patches that fix double unlock bugs, accomplished by querying the complete log of all commits to the code base.

This allows extracting a set of confirmed bugs, that is bugs which have been fixed by developers. Furthermore, picking a point in time before the patch was merged with the code base allows finding the bug in the code base before it was fixed, allowing verification of our proposed analyzer on an implicitly *confirmed* bug. If a patch has been submitted and accepted in the log, it must implicitly have been confirmed as being an actual bug by the maintainers of the code base.

The accuracy of our proposed code analyzer can be determined by extracting n actual double unlock bugs and seeing whether these true positives are detected by the analyzer.

The following sections will detail the evaluation of the proposed analyzer and highlight interestring cases of bugs that either were or were not detected.

4.1 True Positives

4.1.1 4dd75b33

The patch 4dd75b33 patches a double unlock error in the file fs/ubifs/orphan.c, part of the file system components of the kernel.

The function orphan_delete contains two if statements which, if evaluating to true will result in a double unlock of &c->orphan_lock, since the function ubifs_delete_orphan calls orphan_delete and then unlock the same pointer value.

The full code snippet for this function can be seen in Fig. 3.

```
void ubifs_delete_orphan(struct ubifs_info *c, ino_t inum)
{
    [...]

    orphan_delete(c, orph);

    spin_unlock(&c->orphan_lock);
}


static void orphan_delete(struct ubifs_info *c,
    struct ubifs_orphan *orph)
{
    if (orph->del) {
        spin_unlock(&c->orphan_lock);
        dbg_gen("deleted twice ino %lu", orph->inum);
        return;
    }
    if (orph->cmt) {
        orph->del = 1;
        orph->dnext = c->orph_dnext;
        c->orph_dnext = orph;
        spin_unlock(&c->orphan_lock);
        dbg_gen("delete later ino %lu", orph->inum);
        return;
    }
}
```

Figure 3: The orphan_delete function containing a double unlock.

4.2 False Negatives

Several false negatives were found during evaluation of the proposed error checker.

5 Future Work

Extending using the implementation of the EBA analyzer has proven to provide a foundation for analyzing the Linux kernel components, allowing building an error checker able to process these files with relative ease. Building error checkers for other error types would allow the EBA analyzer to validate the source code for Linux components while possibly detecting more error types.

Extending the analyzer in this plugin-style furthermore allows the user(s) of the analyzer to only check for specific error types, speeding up evaluation of source code.

`sed` replacements should be fixed in future versions

6 Conclusion

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. Effective bug finding in c programs with shape and effect abstractions. In *VMCAI*, 2017.

[2] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. *SIGPLAN Not.*, 45(1):43–56, January 2010.

[3] IEEE and The Open Group. pthread_spin_unlock - unlock a spin lock object, 2017.

7 Appendix

7.1 Confirmed Double Unlock Bugs

File	Present in	Patched in
drivers/block/drbd/drbd_main.c	b0814361	8e9c5230
fs/ubifs/orphan.c	7542c6de	4dd75b33
drivers/gpu/drm/nouveau/nouveau_svm.c	5fbcf501	de4ee728
fs/btrfs/file.c	78e03651	f49aa1de
drivers/staging/wilc1000/wilc_wlan.c	ca641bae	fea69916
drivers/staging/kpc2000/kpc_dma/fileops.c	d4c596eb	c85aa326
fs/nfs/client.c	a46126cc	c260121a
fs/btrfs/file.c	2b90883c	8fca9550
drivers/media/dvb-core/dvbdev.c	ded71626	122d0e8d
mm/memory_hotplug.c	6376360e	e3df4c6e
sound/soc/codecs/pcm512x.c	fd270fca	28b698b7
drivers/target/target_core_user.c	807cf197	f0e89aae
drivers/rpmsg/qcom_smd.c	fb416f69	c3388a07
drivers/scsi/aacraid/commsup.c	09624645	d844752e
drivers/staging/rtl8188eu/os_dep/usb_intf.c	612e1c94	23bf4042
block/blk-cgroup.c	e0223003	bbb427e3

7.2 Evaluation Results

File	Status
drivers/block/drbd/drbd_main.c	Undetected
fs/ubifs/orphan.c	Detected
drivers/gpu/drm/nouveau/nouveau_svm.c	Undetected
fs/btrfs/file.c	Undetected
drivers/staging/wilc1000/wilc_wlan.c	Undetected
drivers/staging/kpc2000/kpc_dma/fileops.c	Undetected
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	Undetected
drivers/target/target_core_user.c	Uncaught exception
drivers/rpmsg/qcom_smd.c	Compilation error
drivers/scsi/aacraid/commsup.c	Compilation error
drivers/staging/rtl8188eu/os_dep/usb_intf.c	Compilation error
block/blk-cgroup.c	Compilation error