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

Anders Fischer-Nielsen

December 16th 2019

Contents

Contents **2**

1 Introduction **4**

2 Background **5**

3 Finding Double Unlock Bugs **7**
3.1 Checker Signature 7
3.2 Implementing a Double Unlock Checker 8
3.2.1 testP1 8
3.2.2 testQ1 8
3.2.3 testP2 8
3.2.4 testQ2_weak 8
3.3 May & Must Modalities 8

4 Results **9**
4.1 True Positives 9
4.1.1 `4dd75b33` 9
4.2 False Negatives 10

5 Future Work **11**

6 Conclusion **11**

References **12**

7 Appendix **13**
7.1 Confirmed Double Unlock Bugs 13
7.2 Evaluation Results 13

Todo list

1 Introduction

The Linux kernel supports a vast array of computer architectures and runs on a multitude of devices from embedded devices, through personal computers to large servers; on wireless access points, smart TVs, smartphones, refrigerators. Errors in the Linux kernel therefore affect a multitude of devices and can 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. Shared-memory concurrency and 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 consecutively 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." [4]

If a thread wanting to unlock a lock does not currently hold that lock, the lock has either been unlocked already or never been locked. This will in both instances lead to undefined behaviour at the 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. A program implemented on an assumption that the output of the undefined behaviour will always be within a certain interval might behave unexpectedly with changes in a later compiler update. 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.

A reduced example of a double unlock error found in the Linux kernel can be seen in Fig. 1. This example is based on patch `4dd75b33`[1].

[1] See `https://github.com/torvalds/linux/commit/4dd75b33`

```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);
}
```

Figure 1: An example of a double unlock found in the Linux kernel.

In this project, I want to answer the question:

"What is the performance of a double-unlock error checker implementation when validiation this implementation on the source code of Linux kernel components?"

I will accomplish this by developing an error checker able to find such double-unlock errors and, furthermore, analyzing its performance on Linux kernel components by running the implementation on a set of confirmed double unlock bugs manually found in the Linux kernel components.

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

Section 2 will present the background for double unlock errors and the EBA analyzer. Section 3 will present the design and implementation of the double unlock error checker — the key outcome of this work. Section 4 describes the evaluation of the efficiency of the error checker. Section 5 will describe where to go from here in regards to ways to develop future error checkers and extend the EBA analyzer. Section 6 will summarize and conclude the report.

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 shape-and-effect system in the EBA analyzer.

The same shape-and-effect analysis can be used to build more error checker. Extending the existing implementation allows building another error checker relatively fast, since the implementation is already able to run on the Linux kernel components.

The Linux kernel source code can be analyzed using existing static analysis tools in order to build a Control Flow Graph (CFG) of a given kernel component implementation. The control flow graph shows the possible execution paths of the component.)One can then formulate Computational Tree Logic to sepcify desirable and undesirable execution paths (errors).

The following example shows how to formulate the double *lock* error using CTL:

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

This example states that I am interested in finding a double lock locking the same memory region p. The formula describes execution traces in which, after a prefix, a lock is taken on p and a second lock is eventually taken on p without an unlock on p in between the two locks. This CTL formulation can be visualized as seen in Fig 2.

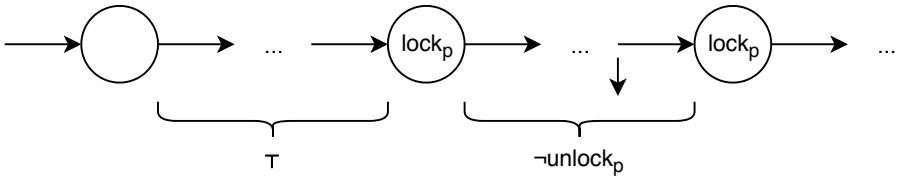

Figure 2: The CTL form of a double lock error visualized.

Computation tree logic models program executions in time as tree-like structures 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 the CTL notation is merely used to formalize which errors I want to find in the Linux kernel source code — in this case a double-unlock error.

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

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

The formula describes execution traces in which, after a prefix, an unlock is performed on p and a second unlock is eventually performed on p without a lock taken on p in between the two unlocks. I am interested in finding a double unlock locking the same memory region, since this is where the undefined behaviour would occur. The visualization of this can be seen in Fig 3.

The existing implementation of the EBA analyzer explores the CFG of the Linux kernel while checking on the fly 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, and an error is raised.

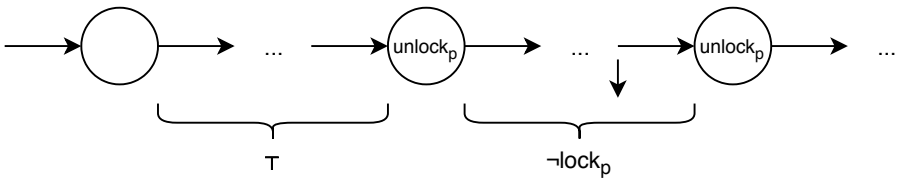

Figure 3: The CTL form of a double unlock error visualized.

The implementation of the EBA analyzer uses a concept of *may lock* and *must lock* internally similar to the concept described by Godefroid et. al. [3]. 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 module in the implementation source code and making this available to the user(s) of the analyzer using a command-line parameter. The following section will elaborate on the implementation details of my work which has been integrated into the EBA code base in this manner.

3 Finding Double Unlock Bugs

The implementation of my work has been integrated in the EBA code base and contributed to the project as a pull request as a prototype. This section describes the implementation of this double unlock checker.

3.1 Checker Signature

The EBA analyzer is implemented in a manner such that all checkers adhere to a unified interface — a signature. This signature mimics the CTL definition of the double locks in that the analyzer expects four functions to be implemented in order to satisfy $p_1 \,\mathrm{EU}\, q_1 \,\mathrm{X}(p_2 \,\mathrm{EU}\, q_2)$, matching the CTL shown in the previous section. All checkers implement four checks, p_1, q_1, p_2 and q_2 respectively, all returning a matching value. The reachability of evaluating the functions within the aforementioned CTL expression is then evaluated by the outer layers of the implementation of the EBA analyzer. An error has been detected in the program if a reachable node is found.

These four functions are defined in a signature, `Spec` in the EBA implementation, which defines the aforementioned checker functions and other helper functions a given checker must implement in order to interface with the outer parts of the analyzer. The implementation of the EBA analyzer has not been modified significantly instead the implementation of the double unlock checker of this project simply implements the signature of the analyzer in order to run.

3.2 Implementing a Double Unlock Checker

The aforementioned functions are named `testP1`, `testQ1`, `testP2`, `testQ2_weak` in the `Spec`, respectively. These functions take so-called `step`s in the CFG of the input program and evaluate to an `Option` value. The `Option` will be `None` if the function did not match, otherwise `Some(step)`.

The double unlock checker of this project implements these as follows:

3.2.1 testP1

This check is implemented as a lift of the input step value to an `Option`, thereby matching all input values and allowing the following checks to execute.

3.2.2 testQ1

A helper function is implemented within this check, `unlocks_and_not_locks`, which — as the name implies — checks whether an *unlock* is performed on a given region and no *lock* is taken in the same `step`. This helper function is evaluated across the `step`s in the CFG of the input program. If a `step` satisfying this is found, a value is returned for the remaining analyses.

3.2.3 testP2

This check verifies that no lock is taken on a region in a given `step`. If a step is found which satisfies this, a value is returned for the remaining analysis step.

3.2.4 testQ2_weak

This check verifies that an unlock is present in a region with associated effects in a given `step`. The implementation of this helper can be seen below.

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

3.3 May & Must Modalities

The implementation of the EBA analyzer implements the concept of *may* and *must* for effects of shapes. As mentioned previously, certain effects are not guaranteed to be executed and are therefore of the type *may*. In order to execute a more thorough analysis, the double unlock checker of this project performs both *may* an *must* checking in an attempt to detect more double unlock errors.

Two passed are performed over a given input CFG in order to accomplish this. These two passes can be expressed as extensions to the aforementioned CTL as

$$\top \text{ EU } may_{unlock} \text{ X}(\neg may_{lock} \text{ EU } must_{unlock})$$
$$\text{and}$$
$$\top \text{ EU } must_{unlock} \text{ X}(\neg may_{lock} \text{ EU } may_{unlock}).$$

Together these functions satisfy $\top \text{EU} \mathit{unlocks} \text{X}(\neg \mathit{locks} \text{EU} \mathit{unlocks})$. If a reachable path satisfying this is found within the CFG of a Linux kernel component a possible error is found.

The source code of the implementation of the EBA analyzer is hosted publicly on GitHub[2] and the implementation of the double unlock checker of this project has been submitted to the EBA project as a Pull Request[3], extending the capabilities of the analyzer.

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 I am 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. 4.

[2] `https://github.com/iagoabal/eba`
[3] `https://github.com/IagoAbal/eba/pull/5`

```
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); /* If executed, double unlock */
        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); /* If executed, double unlock */
        dbg_gen("delete later ino %lu", orph->inum);
        return;
    }
}
```

Figure 4: The `orphan_delete` function containing a double unlock.

4.2 False Negatives

Several false negatives were found during evaluation of the proposed error checker. Out of evaluation on 15 true positives found in the Linux kernel source only 2 were found.

Debugging the implementation of the EBA analyzer shows that results in the true positives are lost on the $Q2$ in $p_2 \mathrm{EU} q_2$. This check has been made as general as possible in the implementation of the prototype by only verifying that a `step` must/may contain an unlock. This unlock check has been implemented using the internal constructs of the implementation of the EBA analyzer and this leads to a suspicion that the unlock check in the internals is imprecise.

Due to the false negatives containing confirmed double unlocks it seems that either the path exploration of the implementation of EBA does not evaluate all paths or that the unlock check does not detect the second unlock correctly.

5 Future Work

Extending the implementation of the EBA analyzer has proven to be a relatively fast approach for developing error checkers analyzing the Linux kernel, allowing building an error checker that is 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 way furthermore allows the user(s) of the analyzer to only check for specific error types, speeding up evaluation of source code if only specific checks are desired.

Implementing the shortcomings of the prototype of this project explained in the previous section to detect a larger number of true positives should be done by improving the ability of the implementation of the EBA analyzer to detect unlock statements in the source code. This would improve the accuracy and therefore the usefulness of the prototype.

Implementing the prototype of this project as a finite state machine could potentially result in an implementation with greater accuracy. The implementation of such an approach might prove easier to reason about and debug, though it is unknown if this is the case. I believe the CTL-inspired implementation of this project can be translated into a finite state machine, though this would have to be determined in a future project.

The `eba-cil` library used for generating the C intermediate language used by EBA does currently not support constructs found in the output of the GCC compiler heavily used by the Linux kernel developers. These include:

- *Static Assertions*, added in C11 which has been implemented since since GCC 4.6[4]
- *Assembler Instructions with C Expression Operands*, an extension available since GCC 3.1[5].

Currently the output from GCC on input files using these features results in a parse error in EBA and therefore need to be removed from the compiler output before analysis. This negatively impacts the soundness of the analysis due to the removal of source code. Supporting these features in the implementation of EBA would improve the soundness of the analysis.

6 Conclusion

[4]See [5]
[5]See [2]

References

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

[2] Inc. Free Software Foundation. Assembler instructions with c expression operands. `https://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Extended-Asm.html`, 2002. Accessed: 2019-11-25.

[3] 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.

[4] IEEE and The Open Group. pthread_spin_unlock - unlock a spin lock object. `https://pubs.opengroup.org/onlinepubs/9699919799/`, 2017. Accessed: 2019-11-25.

[5] ISO. *ISO/IEC 9899:1999(E) - Programming Languages - C*. Geneva, Switzerland, 1999.

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	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	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