

Code Verification

V Anuradha (04026014)
Chinmay Jain (04005004)
Srivathsan B (04d10022)

1 Introduction

The general problem of code verification, which involves asserting certain post conditions after the execution of a code snippet, is known to be undecidable. Various abstraction techniques have been developed which claim successful verification of many useful properties. These techniques have been implemented in tools thus enabling hands on verification of many programs. In this report, we discuss the issues and analyze the verification of certain chosen properties of a particular linux module.

2 Choice of the Code

Our hunt for a suitable code to be verified started with reading up many bug reports on the linux kernel, so that we get a code on which we could actually see the error being displayed. These were sufficiently incomprehensible that we had to resort to other simpler modules in the linux kernel.

Our first choice was to check for some process scheduling properties. We chose to verify *mutex.c*. With the introduction to tools BLAST and DAGGER by this time, we tried parsing the code with these tools. At this point of time, we understood the usage of both the tools. DAGGER required that all type definitions be placed in the same file. In BLAST, we were also introduced to the abstraction using *assume* for functions. We also felt the need to understand the code in more depth. Also, the problems with verification of pointer expressions was noticed.

With the hope to get a better verifiable code, that which we can understand better and so, do the abstractions in a better way, we explored the scheduling process of the IP Virtual Server. We could get a good overall understanding of the structure and behaviour of the modules. This contained a lot of interlinked modules again and after sufficient efforts to understand the details of the code, we had to discard it.

We finally have chosen the following code which duplicates a parent process' file structure on to the child's file structure during the process of forking. The following paragraphs give an explanation of the code used.

The linux kernel includes a program for forking of processes (*fork.c*). We used the function *dup_fd* inside this program which duplicates the file descriptors.

fork called at some point in a process, creates a child process identical to its parent except for its process ID. The process of forking involves duplicating the parent's page tables. A part of the forking process also involves copying the file structure of the parent process to the child. Every process has a *struct files_struct* which maintains an array of pointers to files that are currently open.

The main function which starts a new process, calls a function *copy_process* which copies all attributes of the parent process to a newly created child process structure. Within *copy_process* is another function *copy_files* which calls the chosen function *dup_fd*.

dup_fd duplicates the old file structure (which is passed as an argument) and returns a new file structure which is identical in all the fields of the older one. In the process it calls functions such as: *alloc_files* to allocate a new page and setting certain values. Then it calls *count_open_files* which counts the number of open files in the old structure. We have assumed that the maximum number of file descriptors for the new structure is same as the old structure, which will prevent the need of calling *expand_files*. Finally, the function *dup_fd* which copies the file descriptor array from old structure to the new structure. This is the most important step that we need to verify. Locking and unlocking of spinlocks is also done during the function for atomicity purposes.

3 Properties to be verified

We wanted to verify if the file structure copy is carried out properly. This involved checking certain fields of the *struct files_struct*. The parameter passed on to the function is *struct files_struct * oldf* and it returns a *struct files_struct * newf*. Our initial aim was to check if the two structures are the same and more specifically if the file descriptor array were being copied properly. This would involve checking the individual fields of the two structures. We also wanted to check if the lock-unlock is handled properly and if smaller code segments were doing what they were supposed to do which in many cases blast was not able to infer without our help.

4 Modifications done on the code for verification

We had to do the following modifications to the code for verification.

- The function *dup_fd* uses a lot of type definitions, structure definitions and functions which have been placed in other files that have been included in *fork.c*.
- We placed all the required typedefs, struct definitions, predefined constants and function definitions in the same file.
- For the functions, some of them were copied from the files where they were defined for e.g. *alloc_files* while other like *expand_files* were abstracted out. We abstracted those functions which were not very important for the verification and were calling various other functions.

- These modifications ensured that the code could be preprocessed and the .i file made. We were able to start verifying the code using *pblast.opt*.
- A part of the code handles the case when the number of open files in the old file structure is greater than the field in the new file structure which maintains the value of the maximum file descriptor, a function *expand_files* is called. We assumed that the program never takes this path. We added corresponding *assume* at these places.

5 Observations and Results

- We managed to successfully check temporal properties like the alternate use of *spin_lock* and *spin_unlock*. This involved specifying the temporal property in a specification file.
- We managed to verify various smaller code segments in the functions like: *alloc_files* and *count_open_files* which also involved dealing with pointers.
- We were not able to verify that the copying is being done successfully. The problem that we faced was related to the understanding of pointers by BLAST. BLAST although understands pointer assignments it does some very trivial mistakes while verification. We shall mention those in later section.
- We have also submitted a code in file *ex3.c* which shows the inability of BLAST while dealing with pointers.
- Finally, we can say that on the way of verifying the whole code we were unable to verify just the last segment of the code which would have completed our verification.

6 Difficulties and Problems

- Understanding the code was a daunting task.
- Building the code - involved getting all the required function definitions, typedefs, predefined constants and structs.
- Handling of arrays and pointers.
- Ambiguous results for the some piece of code.
- Handling of events given in the specification file. The matching of strings is not working up to the mark.
- We were using the foci server as the back end constraint solver and this turned out to be not working correctly. We then resorted to clp which gave expected results for the initial problems.

7 Some Unexpected Results

We shall now give examples of some very simple yet important problems with the verification done by BLAST.

Locking-Unlocking For checking the locking and unlocking of the spinlocks we made a specification file. The documentation mentions that we can use \$1 and \$? for matching any variable string. Thus, in order to verify *spin_lock(&oldf->file_lock)* and *spin_lock(&newf->file_lock)*. However, BLAST only checks the string outside the brackets i.e. the function name. It does not differentiate on the basis of the argument to the function. We thus had to make some change in the original code.

Predicate satisfiability Blast unexpectedly gives a counter example to be true which can easily be seen as unsatisfiable. One example that we encountered is:

$$\begin{aligned} &Pred(t@main == u@main) \\ &Pred(*(u@main)! = *(t@main)) \end{aligned}$$

Order Of assert We were surprised to see that BLAST changes the result of the verification of code as we change the order of the assert statements given. We were not able to find a logical way to explain such happening. We have the files ex1.c and ex2.c which differ only in the position of `assert(*u == 2)` and it turns out that blast gives fail for ex1 and success for ex2. We have added the file in the submission.

8 Conclusion and Experiences

- This experience revealed to us the difficulty that goes in, at present, to verify a real, non-toy code.
- It was nice to analyze counterexamples in which we could see the predicates being tracked and assertions done based on these predicates.
- The limitations of the tool played a major part in the difficulty experienced during verification and the other major part was the need for *understanding* the code for the *automated* verification.
- We sincerely appreciate the efforts to make the developed theory usable and applicable. But we are forced to maintain that formal methods is not yet for the common man and do wish that verification would at some time become as simple and as common as search.