Lab 5: Bounded model checking C code

15-414: Automated program verification

Lab goals

In this lab, we will use a model checking tool called the C Bounded Model Checker (CBMC) to verify safety properties of C code up to a fixed execution depth. The lab is distributed with two source files, `queue.c` and `mutex.c`, each containing a number of implementation bugs. Your job is to use the model checker to identify these bugs, repair them, and then show that your repairs are sufficient using bounded model checking.

Lab instructions

This lab makes use of a new tool. Although it should not require as much time to become familiar with it as it did learning to use Why3, it is important to take some time before attempting the problems in this lab learning how to use the tool. The first part of this document walks through the basics of CBMC on an example covered in lecture, and then directs you to a brief tutorial written by the CBMC developers. Finish these first before moving on to the rest of the lab.

Download the file `cbmc.zip`, and verify that it contains `queue.c` and `mutex.c`. The lab is due at midnight on December 8. Because this is the end of the semester, there are no extensions, so please start early in case you experience difficulties using the tool. When you complete the lab, place all of the files requested in Parts 1 and 2 in a `zip` archive, and submit it to Autolab.

1 A brief introduction to the C Bounded Model Checker

1.1 Setting up the model checker

You do not need to install any software to complete this lab. The course staff has set up an installation of CBMC on AFS, which can be accessed on an SSH session to `linux.andrew.cmu.edu` or `linux.gp.cs.cmu.edu`, or on a CS cluster machine. The binary is located at `/afs/cs.cmu.edu/academic/class/15414-f17/cbmc/cbmc`, and should run as-is. We recommend adding `/afs/cs.cmu.edu/academic/class/15414-f17/cbmc` to your `PATH` variable, so that you can run the model checker by simply invoking `cbmc`.

Before getting started with the lab, please test the command. If you do not see the following output, let the course staff know immediately.

```
andrewid@linux1:~$ cbmc
CBMC version 5.8 64-bit x86_64 linux
Please provide a program to verify
```

Installing CBMC on your own machine. While it is possible to install CBMC on your own machine, we do not recommend doing so as this may lead to delays in your completing the assignment. Because an installation is available on AFS, the course staff will not be able to spend time helping you debug a failed installation on a personal machine. Binaries and installation instructions for Windows, MacOS, and Windows are available at `http://www.cprover.org/cbmc/`. However, if you attempt to use these and installation does not work on your machine immediately, please revert to using the AFS installation.

1.2 Using CBMC

We will illustrate the use of CBMC to find bugs or verify their absence by applying it to the toy example from Lecture 19, which is shown in Figure 1. The first thing to note is that we have placed the code in the `main` function. Being a C function, CBMC expects the entry point of the program to reside in `main`. If the file given to CBMC has no `main`, and an alternate entry point isn't provided with the `function` command-line argument, then CBMC will finish without verifying anything.

```
int N, x;
int main() {
  int i = N;
  while(0 <= x && x < N) {
    i = i - 1;
    x = x + 1;
  }
  __CPROVER_assert(0 <= i, "postcondition");
}
```

Figure 1: Toy example program from Lecture 19

The next thing to note is the call to `__CPROVER_assert`. This is the primary form of user-defined specification supported by CBMC. When the model checker is invoked, it will attempt to verify that the condition given as the first argument holds on all paths up to a specified bound. If no bound is given on the command line, CBMC will attempt to infer an upper bound on the program's execution depth, and verify the program after unwinding. The second argument to `__CPROVER_assert` is a diagnostic string that will be reported in the results if CBMC finds a counterexample for the assertion.

Let's run the model checker on this example. For now, we will not specify an unwinding bound, and let CBMC try to infer the bound on its own. The results are shown in Figure 2. Surprisingly, we see that CBMC concluded with a `VERIFICATION SUCCESSFUL` message! This is contrary to what we saw in class when we worked this example out, where we found a counterexample at $\texttt{N} = -1, \texttt{x} = 0$. Why didn't CBMC find this bug? Notice that `N` and `x` are not initialized. Because they are static globals, CBMC assumes that they are initialized to 0 by default. This is not necessarily a safe assumption to make, and there are two primary ways to address it.

The first approach is to pass the command-line argument `nondet-static`, which tells CBMC to assume that any variable with static lifetime is initialized to a nondeterministic value. The second approach is to introduce the nondeterminism ourselves. We can do this by declaring a function with no body in the source file being analyzed, i.e., an external function. To deal with external code without making unwarranted assumptions, CBMC assumes that any values returned from such code can take any value. Figure 3 is updated

```
andrewid@linux1:~$ cbmc toy1.c
CBMC version 5.8 64-bit x86_64 macos
Parsing toy1.c
Converting
Type-checking toy1
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 38 steps
simple slicing removed 0 assignments
Generated 1 VCC(s), 0 remaining after simplification
VERIFICATION SUCCESSFUL
```

Figure 2: CBMC output on toy example program from Figure 1

```c
int nondet_int();
int N, x;
int main() {
  N = nondet_int();
  x = nondet_int();
  int i = N;
  while(0 <= x && x < N) {
    i = i - 1;
    x = x + 1;
  }
  __CPROVER_assert(0 <= i, "postcondition");
}
```

Figure 3: Toy example with nondeterministic initialization.

to reflect this approach, by declaring an external function `nondet_int` and calling it to initialize `N` and `x` at the beginning of `main`.

If we run the model checker again on the updated program, we see a large amount of output that fails to terminate.

```
andrewid@linux1:~$ cbmc toy1.c
CBMC version 5.8 64-bit x86_64 linux
...
Unwinding loop main.0 iteration 1785 file toy1.c line 7 function main thread 0
Unwinding loop main.0 iteration 1786 file toy1.c line 7 function main thread 0
Unwinding loop main.0 iteration 1787 file toy1.c line 7 function main thread 0
...
```

This is due to the fact that we did not specify an unwinding depth; CBMC attempts to find a bound on the depth of the loop, but is unable to do so because `N` is initialized nondeterministically to take any integer value. We address this by passing `--unwind 3` on the command line, telling CBMC to unroll the loop at most three times. We now see the following (note that some of the output has been omitted to save space).

```
andrewid@linux1:~$ cbmc toy1.c --unwind 3
Solving with MiniSAT 2.2.1 with simplifier
1019 variables, 3727 clauses
SAT checker: instance is SATISFIABLE
Runtime decision procedure: 0.006s
** Results:
[main.assertion.1] postcondition: FAILURE
** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
```

This is the result we expected to see, knowing that the assertion should not always hold. We see that CBMC generated a SAT instance with 1019 variables and 3727 clauses, and found it to be satisfiable. This corresponds to a violation of the assertion labeled `postcondition`, which is the property we wished to check.

In order to see a counterexample for this bug, we pass the command-line argument `--trace`. The output is shown below.

```
andrewid@linux1:~$ cbmc toy1.c --unwind 3 --trace
Trace for main.assertion.1:

State 20 file toy1.c line 4 function main thread 0
----------------------------------------------------
  N=-1073741824 (11000000000000000000000000000000)
State 21 file toy1.c line 5 function main thread 0
----------------------------------------------------
  x=-1073741825 (10111111111111111111111111111111)
State 22 file toy1.c line 6 function main thread 0
----------------------------------------------------
  i=0 (00000000000000000000000000000000)
State 23 file toy1.c line 6 function main thread 0
----------------------------------------------------
  i=-1073741824 (11000000000000000000000000000000)

Violated property:
  file toy1.c line 11 function main
  postcondition
  0 <= i
```

Among other things, the counterexample trace tells us that `N` is initialized to -1073741824 on line 4 in `main`, `i` is initialized to 0 on line 6, and subsequently updated to the same value as `N` on the same line. The trace then ends with the violated property, bypassing the loop entirely. Note that `i` is updated twice; the first instance corresponds to the declaration of `i`, where it is given the default value 0. The second corresponds to the initialization to `N`, which is the source of the bug.

Environment assumptions. Before moving on, we introduce the specification primitive `__CPROVER_assume(Q)`. Like `__CPROVER_assert`, the argument to `__CPROVER_assume` is a Boolean condition. CBMC interprets a call to this function slightly differently: any path that does not satisfy the condition passed to `__CPROVER_assume` is discarded from the analysis. Importantly, if such a path later contains a safety violation, it is not reported in the results. This can be useful when modeling assumptions about the environment, for example if we have reason to believe that the arguments passed to a function will always satisfy certain conditions.

In the present example, if we place a call to `__CPROVER_assume(0 <= N && N < 3);` immediately after the intitialization of `N` on line 4, then CBMC will return `VERIFICATION SUCCESSFUL`. Furthermore, if we tell CBMC to insert unwinding assertions by passing the command-line argument `--unwinding-assertions`, then we can conclude that there are no bugs up to the given unwinding depth, and that the unwinding depth is sufficient for exhaustive verification.

```
andrewid@linux1:~$ cbmc toy1.c --unwind 3 --unwinding-assertions
** Results:
[main.assertion.1] postcondition: SUCCESS
[main.unwind.0] unwinding assertion loop 0: SUCCESS
** 0 of 2 failed (1 iteration)
```

Array bounds and pointer checking. This lab will have you verify the absence of memory errors in two C programs. It is possible to do this by inserting appropriate calls to `__CPROVER_assert` before array and pointer accesses, as in the following example.

```
char buf[100];
int i = get_index();
__CPROVER_assert(0 <= i && i < 100, "array bounds check");
printf("%d", buf[i]);
```

However, CBMC will automatically insert these checks for you when given the command-line arguments `--bounds-check` and `--pointer-check`. Before starting the lab, please read the tutorial at `http://www.cprover.org/cprover-manual/cbmc.shtml`, which provides more information about the use of these arguments. Further documentation is available at `http://www.cprover.org/cprover-manual/`.

2 Lab instructions

2.1 Part 1: Buggy data structures

In the file `queue.c` distributed with the lab, you will find an implementation of two data structures: and "unbounded" array, and a queue that uses the unbounded array to store its elements. Your job is to use CBMC to find the memory safety errors in this code, fix them, and verify your fixes up to a reasonable bound. You are free to use the command-line arguments `--bounds-check` and `pointer-check` rather than writing your own assertions to check for bugs, but you may find it useful to use `__CPROVER_assert` along the way to check your understanding of bugs discovered by CBMC.

Unbounded array. Begin by verifying the unbounded array implementation on its own. Note that a preprocessor constant `MAX_ARRAY_BOUND` has been defined and set to 8. This is the value that the graders will use to determine the success of any fixes you implement, so you should make sure that you are able to verify the absence of bugs when arrays of this size are used. However, in the intermediate stages of your work, you may find it useful to temporarily decrease this value to speed up the model checker.

You should proceed as follows.

1. Because no `main` function is provided, and the implementation of the unbounded array is intended to be used as a general-purpose library, to verify this code you will need to provide an entry function that invokes all of the available functionality of the unbounded array implementation. Place this in the function `ubarray_harness`, and direct CBMC to use this as the entry point with `--function`. In order to ensure that the verifier explores all of the possible ways in which the implementation could be used, you should use nondeterminism to make the harness as general as possible. You may find it helpful to consult the notes for lecture 14; note that `nondet_int` and `nondet_char` have been declared at the top of `queue.c`.

2. Use CBMC to find as many bugs as you can by increasing the unwinding depth until it converges. Note that you may save time in the model checking step by passing the argument `--slice-formula`. Save the command line arguments that reached convergence in the number of identified bugs in a file named `ubarray-2-1-2-args.txt`, and save the output of CBMC with these arguments in `ubarray-2-1-2-output.txt`. Finally, make a copy of `queue.c` with your finished harness in `ubarray-2-1-2-harness.c`.

3. Now fix any bugs that you identified in the previous step. Note that CBMC may generate many error reports for the same underlying programming mistake, so you should not need to make as many modifications as the error report from step 2 suggest. You should use the `--trace` argument to gain more information about the bugs found by CBMC.

   Your fixes can make sparing use of the `handle_error_en` macro defined at the top of the file (see `uba_resize` for an example). When invoked, this will cause the program to exit with an error message. Note that if used improperly, this will result in a non-functional implementation of an unbounded array. Your fixes should aim to satisfy the following informal specification: whenever the array structure given to a function satisfies `valid_ubarray`, then the function should behave as expected (without terminating the program), returning a correct result as expected and a `valid_ubarray` where applicable. Rather than terminating the program, your implementation should return an appropriate error code whenever possible, or NULL where appropriate.

   Verify that your fixes are sufficient to rule out memory errors by running CBMC on the result; you do not need to verify full functional correctness. Report the largest unwinding depth that you were able to verify in a reasonable amount of time (i.e., less than one minute) in `ubarray-2-1-3-report.txt`. Additionally, report whether you are able to use `--unwinding-assertions` to conclude that your bound is sufficient, and explain why. Save the resulting fixed code, in addition to your harness, in `ubarray-2-1-3-fixed.c`.

Queue. Now continue by verifying the queue implementation that appears later in the source file. You should proceed as you did before with unbounded arrays.

1. Write a harness for the queue implementation in `queue_harness`. Use a similar approach is you did for the unbounded array, utilizing nondeterminism to explore as much functionality as possible.

2. Use CBMC to find as many bugs as you can by increasing the unwinding depth until it converges. Save the command line arguments that reached convergence in the number of identified bugs in a file named `queue-2-1-2-args.txt`, and save the output of CBMC with these arguments in `queue-2-1-2-output.txt`. Finally, make a copy of `queue.c` with your finished harness in `queue-2-1-2-harness.c`.

3. As before, fix any bugs tht you find using `--trace` to gather information. Use the same guiding principles as in the case of unbounded arrays to inform your fixes. Verify that your fixes are sufficient to rule out memory errors by running CBMC on the result; you do not need to verify full functional correctness. Report the largest unwinding depth that you were able to verify in a reasonable amount of time (i.e., less than one minute) in `queue-2-1-3-report.txt`. Additionally, report whether you are able to use `--unwinding-assertions` to conclude that your bound is sufficient, and explain why. Save the resulting fixed code, in addition to your harness, in `queue-2-1-3-fixed.c`.

4. Write a harness for `queue_copy` in `queue_copy_harness` to test the functional correctness of this function. The harness should test that after copying a queue `Q` to `Qp`, the contents of `Qp` accessible through the queue library functions are identical. What is the largest unwinding bound that you are able to verify in under one minute? Copy the arguments that you used to verify at this depth to `queuecopy-2-1-4-args.txt`, and your source with this harness to `queuecopy-2-1-4-harness.c`.

2.2 Part 2: Dealing with threads

In the file `mutex.c`, you will find an implementation of a threaded program. It begins in `main` by spawning two new threads on functions `producer` and `consumer`.

```
pthread_create(&tid1, NULL, producer, (void *)0);
pthread_create(&tid2, NULL, consumer, (void *)0);
```

For the purposes of this lab, you do not need to be familiar with the details of these calls to `pthread_create` beyond the fact that after each call completes, the operating system will have spawned a new concurrent thread at the beginning of the function passed in as the third argument. When both complete, the progam will consist of three threads: one running inside `main`, one inside `producer`, and one in `consumer`. As the program executes, instructions executed by the threads can interleave in *any* possible order as the operating system decides which to schedule. CBMC accounts for this by unwinding the code executed by each thread in all possible orderings, treating the scheduler's choice nondeterministically.

Each thread has its own local call stack, so local variables are accessible only to the thread running in that process. However, global variables are accessible by any thread. This can create problems as the threads operate on shared data if they do not take steps to ensure mutual exclusion.

In this code, `producer` and `consumer` attempt mutual exclusion through the use of three global variables: `try_producer`, `try_consumer`, and `turn`. When either thread wants to enter the critical section, it signals its intention by setting `try_producer` (alternately, `try_consumer`) to 1. It then gives the other thread the opportunity to enter the critical section first, by setting `turn` to a value corresponding to the other thread (i.e., either `PRODUCER` or `CONSUMER`). It then waits to enter the critical section for as long as the other thread is trying to enter the critical section, and `turn` denotes that it is the other thread's turn to enter (i.e., see the `while` loops on lines 50 and 74). Once a thread enters its critical section, it does whatever work is necessary on shared global data, and when finished exits the critical section by setting `try_producer` or `try_consumer` to 0.

In this part of the lab, you will use CBMC to check that the code implements this functionality correctly and is free of memory errors.

1. First check for memory safety as in Part 1 using `--bounds-check` and `pointer-check`. Note that memory safety might be violated if mutual exclusion is not satisfied, due

to the threads' use of a shared array. Save the command line arguments that reached convergence in the number of identified bugs in a file named `mutex-2-2-1-args.txt`, and save the output of CBMC with these arguments in `mutex-2-2-1-output.txt`.

2. Fix any memory safety violations that you found in Part 1, and verify your fix by unwinding to a reasonable depth (i.e., you do not need to unwind further than it takes CBMC to verify in one minute). Save the command line arguments you used to achieve successful verification in `mutex-2-2-2-args.txt`, and your fixed implementation in `mutex-fixed.c`.

3. Verify that when `producer` and `consumer` terminate, all of the data in `data_source` has been processed by `consumer`. You may need to modify `main` slightly to achieve this, and make use of CBMC's custom specification facilities described earlier in the document. Save the command line arguments that you used to successfully verify this property in `mutex-2-2-3-args.txt`, and the source file with any additional specification code in `mutex-spec.c`.