# RRA tutorial by an example

This document illustrates an end-to-end process running RRA on an example program. It uses aws\_array\_eq as the example benchmark.

## Clone and Build CBMC (with RRA extension)

Clone the git repo:

git clone git@github.com:zlfben/cbmc.git

git checkout rra-dynamic

Follow the instruction in “COMPILE.md” in the repo to compile CBMC into the cbmc/build folder.

Copy the executions to the system’s executable path. For example, in ubuntu, from the repo:

sudo cp build/bin/\* /usr/local/bin/

## Get the example benchmark “aws\_array\_eq”

Clone the AWS-C-Common repo:

git clone git@github.com:awslabs/aws-c-common.git

cd aws-c-common

git checkout e06caa32195fa124b970a6ad0be3a1d047d462e2

git submodule update --init

Go to the directory for the “aws-array-eq” proof:

cd verification/cbmc/proofs/aws\_array\_eq

## Re-write the C programs to prepare unbounded proof with RRA

In aws-c-common/verification/cbmc/proofs/aws\_array\_eq/aws\_array\_eq\_harness.c, replace all the boundaries for arrays because we want to get unbounded proof:

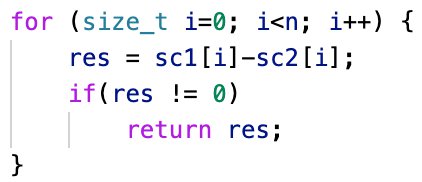
\_\_CPROVER\_assume(rhs\_len <= MAX\_BUFFER\_SIZE);

===>

\_\_CPROVER\_assume(rhs\_len <= UINT32\_MAX);

Note that this replace the MAX\_BUFFER\_SIZE (set to 20 in the proof) with another bound. We did not use maximum 64 bit integer as the bound because of the way CBMC handles pointers: top bits are used to maintain which memory object the pointer is referring to. In a 64-bit machine, this 32-bit bound is safe and will not touch these top bits.

In aws-c-common/verification/cbmc/stubs/memcmp\_override.c, replace the loop body with:



This is because RRA currently does not support two iterators iterating abstracted loops at the same time.

Now we are ready to build the original GOTO program. In aws-c-common/verification/cbmc/proofs/aws\_array\_eq/, run:

make goto

This will build the GOTO program in the goto/ folder under the current directory. Go to the folder where GOTO program “aws\_array\_eq\_harness.c.goto” exists to prepare for the next step.

Things not supported are:

1. Function calls inside assertions
2. Initialization of abstracted variables inside another function
3. Pointer iterators with a different width from the array entry (e.g. using uint64\_t \* to iterate through uint8\_t array)

## Write a specification file

We will need to write a specification file on which arrays should be abstracted and what variables are iterators of abstracted loops.

Assuming that you are under the directory that has the GOTO file “aws\_array\_eq\_harness.c.goto”. Create a file called “aws\_array\_eq.json” with the following content:



We need to identify the arrays to be abstracted: “aws\_array\_eq\_harness::1::lhs” and “aws\_array\_eq\_harness::1::lhs”. Normally length variables act as “guard”s for abstracted loops so they need to be concrete. We use “length” entity type to tell RRA that they need to be concrete. We also need to identify loop iterators for abstracted loops. In this case there is only one loop, which is the one in “memcmp”. We use “iterator\_scalar” to identify those loop iterators.

Note that we use full GOTO variable names with functions and block ids. If you are not sure about corresponding GOTO names for variables in C programs, you can run the following command to show all names of variables:

goto-instrument --list-symbols aws\_array\_eq\_harness.c.goto

## Abstract the program and run the proof

Under the same directory, run the following command to abstract the program into “aws\_array\_eq\_harness.c.abst.goto”:

goto-instrument --use-rra $(abspath aws\_array\_eq.json) aws\_array\_eq\_harness.c.goto aws\_array\_eq\_harness.c.abst.goto

Then run CBMC on the abstracted GOTO program:

cbmc aws\_array\_eq\_harness.c.abst.goto