README.md

Copyright (c) University of Luxembourg 2022.

Created by Enrico VIGANO, enrico.vigano@uni.lu, SnT, 2022.

DAMAt example

Introduction

This is an example of how to apply the *DAMAt* tool and procedure to improve an existing test suite. This procedure has been performed on Ubuntu 20.04.4 LTS.

We decided to use *libCSP*, a C library implementing the Cubesat Space Protocol (CSP), a small protocol stack written to facilitate communication between the components of a CubeSat. Additional information on *libCSP* can be found at this link https://github.com/libcsp/libcsp.

To apply the *DAMAt* procedure we need a Software Under Test and the relative test suite to evaluate. *LibCSP* does not come with a test suite, but it includes some examples; starting from one of these examples, we devised a small, intentionally flawed example of integration test suite. The test suite contains a single test case that emulates a server/client configuration over loop-back and verifies that a minimum number of packets is transmitted by the client and correctly received by the server in a given time.

Installation procedure

Once uncompressed, inside the *libcsp_workspace* directory you will find three sub-folders:

- *damat-pipeline*: this folder contains all the scripts and utilities necessary to apply *DAMAt*.
- *libcsp*: this folder contains the source code for libCSP.
- *test_suite*: this folder contains the test suite under test that we are going to try and improve.

To compile *libCSP* and its test cases you will need to install the following packages:

```
sudo apt-get python gcc pkg-config libsocketcan-dev libzmq3-dev
```

This version of *libCSP* has been slightly modified with the insertion of DAMAt's mutation probes (see Step 3: Inserting the mutation probes). To compile it, you will need to go to the *libcsp* folder and run the following command.

```
./waf distclean configure build --mutation-opt -1 --singleton TRUE \
--with-os=posix --enable-rdp --enable-promisc --enable-hmac --enable-dedup \
--enable-can-socketcan --with-driver-usart=linux --enable-if-zmqhub \
--enable-examples
```

The flag *--mutation-opt -1* will set the mutation probes to an inactive state.

To execute the original test case go to the *test_suite* folder and run the following commands:

```
make clean
make test_01
```

The test should pass.

Executing DAMAt

Step 1: Fault Model Specification

The first step for applying the *DAMAt* procedure is specifying a fault model. The fault model must be written in the *csv* format supported by DAMAt. You can find it here:

```
damat-pipeline/fault_model_libcsp.csv
```

This *csv* file contains two *Fault Models* implementing a total of 39 *Mutation Operators*, which will generate the mutants. Then you can make sure that the *DAMAt_configure.sh* scripts contain the correct path to the fault model file.

Another *csv* file, *test.csv* should contain the following:

```
test_01,10000
```

The first column should contain the names of the test cases and the second the normal maximum execution time.

Step 2: Generating the mutation API

To generate the mutation API, run the following commands:

```
cd damat-pipeline
./DAMAt_probe_generation.sh
```

You should see that a new file is generated in the *damat-pipeline* folder: *FAQAS_dataDrivenMutator.h*. This file will contain the implementation of the mutation probes that will be used to modify the SUT's buffer data and generate the mutants. Another newly generated file, *function_calls.out* contains the definitions of the functions implementing the mutation probes. There is a probe for every different fault model.

Step 3: Insert the mutation probes

We will insert the mutation probes in a specific *libCSP* source file:

```
libcsp/src/csp_io.c
```

This file contains the definitions of the two functions that we will target with the probe insertion strategy:

- *csp_send*
- *csp_read*

these functions take as input the data structure *csp_conn_t*, which we are going to mutate. By looking at the content of the file

```
libcsp/csp_io_mutated.c
```

you can see how the probes were embedded in the code. This is an example:

```
csp_packet_t * csp_read(csp_conn_t * conn, uint32_t timeout) {

        /* mutation probe */

        int damat_buffer_read[7];

        damat_buffer_read[0] = 0;
        damat_buffer_read[1] = conn->idin.pri;
        damat_buffer_read[2] = conn->idin.src;
        damat_buffer_read[3] = conn->idin.dst;
        damat_buffer_read[4] = conn->idin.dport;
        damat_buffer_read[5] = conn->idin.sport;
        damat_buffer_read[6] = conn->idin.flags;

        mutate_FM_Read(damat_buffer_read);

        conn->idin.pri = damat_buffer_read[1];
        conn->idin.src = damat_buffer_read[2];
        conn->idin.dst = damat_buffer_read[3];
        conn->idin.dport = damat_buffer_read[4];
        conn->idin.sport = damat_buffer_read[5];
        conn->idin.flags = damat_buffer_read[6];

        /* end of the probe */

        csp_packet_t * packet = NULL;

        if ((conn == NULL) || (conn->state != CONN_OPEN)) {
                return NULL;
        }

#if (CSP_USE_RDP)
        /* RDP: timeout can either be 0 (for no hang poll/check)
                or minimum the "connection timeout" */

        if (timeout && (conn->idin.flags & CSP_FRDP) && (timeout < conn->rdp.c
                timeout = conn->rdp.conn_timeout;
        }
#endif

        if (csp_queue_dequeue(conn->rx_queue, &packet, timeout) != CSP_QUEUE_C
                return NULL;
        }

#if (CSP_USE_RDP)
        /* Packet read could trigger ACK transmission */
        if ((conn->idin.flags & CSP_FRDP) && conn->rdp.delayed_acks) {
                csp_rdp_check_ack(conn);
        }
#endif

        return packet;
}
```

The extra lines of code are because, in its current form, *DAMAt* can only mutate arrays. The file also contains the following inclusion:

```
#include "FAQAS_dataDrivenMutator.h"
```

To insert the probes, you just need to substitute *csp_io_mutated.c* to the original file and copy the *FAQAS_dataDrivenMutator.h* in the same folder.

Step 4: Compile mutants

To enable this step, which will be automatically performed by the *DAMAt* pipeline, the user will need to modify this script:

```
damat-pipeline/DAMAt_run_tests.sh
```

to include the commands to compile the SUT. By taking a look at it, you will see how it has been done for this particular case.

```
# here the user must invoke the compilation of the SUT

compilation_folder="/home/vagrant/libcsp_workspace/libcsp"

pushd $compilation_folder

echo "$deco"
echo "compiling test"
echo "$deco"

./waf distclean configure build --mutation-opt $mutant_id $EXTRA_FLAGS_SINGL \
--with-os=posix --enable-rdp --enable-promisc --enable-hmac --enable-dedup \
--enable-can-socketcan --with-driver-usart=linux --enable-if-zmqhub
```

Step 5: Execute the test suite

As with Step 4, Step 5 will be performed automatically by the pipeline, but the user must take a few preparatory steps. First, the user must provide the *csv* file containing the name of the test case and a normal execution time (useful to set a timeout in case a mutant should cause an infinite loop); You can find the one used in this example here:

```
damat-pipeline/tests.csv
```

Then the commands to run the test cases must be included in the proper space of the script

```
damat-pipeline/DAMAt_run_tests.sh
```

In this case, this is the code that has been added:

```
# here the user shall call the execution of the current test case,

  pushd /home/vagrant/libcsp_workspace/test_suite

  echo "$deco"
  echo "$(tput setaf 1) RUNNING THE TEST NOW! $(tput sgr0)"
  echo "$deco"

  # timeout 30 ./build/examples/csp_server_client -t
```

```
    make clean
    make $tst

    EXEC_RET_CODE=$?

    echo "$deco"
    echo "$(tput setaf 2) FINITO! $(tput sgr0)"
    echo "$deco"

    popd
```

The *$tst* variable is read from the first column of the csv file. When new test cases are added they must be included in that file to be executed during the procedure.

Step 6: Generate the results

At this point to execute the test suite against the mutants and gather the results you just need to run the pipeline by typing:

```
cd damat-pipeline
./DAMAt_mutants_launcher.sh
```

When the execution is complete you will see a message with the evaluation provided by *DAMAt*.

Improving on the existing test suite

After the first execution of *DAMAt*, the metrics describing the performance of the test suite are the following:

- *Fault Model Coverage* 100%
- *Mutation Operation Coverage* 98%
- *Mutation Score* 52%

At a first glance we can see that not all the input partitions have been covered, since the *MOC* is < 100%, and some oracles are missing or incomplete since the *MS* is < 100%.

The file

```
results/final_mutants_table.csv
```

contains information on the status of every single mutant. This file will allow us to see what fault models, data items, and input partitions, in particular, are not well tested due to the test suite's shortcomings. For example the *DataItem* column contains information on which data item in the buffer is targeted by the mutant. In particular:

- 1 = conn-><idin/idout>.pri
- 2 = conn-><idin/idout>.src
- 3 = conn-><idin/idout>.dst
- 4 = conn-><idin/idout>.dport
- 5 = conn-><idin/idout>.sport
- 6 = conn-><idin/idout>.flags

We will add test cases based on the data gathered by DAMAt to improve the test suite. A summary of this process, including metrics and tables is contained in the file

```
summary_of_the_results.xlsx
```

Improving the Mutation Operation Coverage

We can improve the *Mutation Operation Coverage* by adding new test cases that exercise partitions not covered by the test suite such as the ones targeted by mutants that were *NOT_APPLIED*.

By looking at the mutants that were *NOT_APPLIED* we can identify input partitions not covered by the test suite:

- there is no test case that covers a value of *conn->idin.pri* and *conn->idout.pri* > 3;

Improving the Mutation Score

We can improve the *Mutation Score* by adding new test cases that contain oracles on the values modified by *LIVE* mutants. In this case, this being an integration test suite, our primary focus is to check whether the different components (server and client) interact correctly and if the connection data contained in the structure *csp_conn_t* is correctly handled and preserved through these interactions.

By looking at the mutants that were *APPLIED* but not *KILLED* by the test suite, we notice that they belong to some specific members of the *csp_conn_t* structure:

- *conn->idin.pri* and *conn->idout.pri*, which define the priority of the connection;
- *conn->idin.src*, *conn->idout.src*, *conn->idin.dst*, and *conn->idout.src*, which represent the source and destination;
- *conn->idin.sport*, which represents the source port.

Test 02: Priority (pri)

A test case containing an oracle that checks if the *conn->idin.pri* and *conn->idout.pri* coincide between server and client should detect eventual mismanagement of the priority in the connection interfaces, and kill the mutants emulating these kinds of faults, that were previously *APPLIED* but not *KILLED*. In the test the client will send 5 packages with the four different priorities defined by *libCSP*:

- CSP_PRIO_CRITICAL (0)
- CSP_PRIO_HIGH (1)
- CSP_PRIO_NORM (2)
- CSP_PRIO_LOW (3)

Then it will test what happens if the priority is not defined by *libCSP*, for example, if it is equal to 6, thus hopefully improving the *Mutation Operation Coverage*, leading to a more extensive test suite. The content of *conn->idin.pri* as received by the server will be checked against the priority established by the client when connecting. The test case is implemented in the file

```
test_suite/test_02/test_02.c
```

and can be executed by compiling *libCSP*, moving to the *test_suite* folder, and executing the following command:

```
make test_02
```

Then you can add the test to the *damat-pipeline/tests.csv* so that it will be recognized and executed by *DAMAt*: The file should now contain these lines:

```
test_01,10000
test_02,10000
```

After adding *test_02* to the test suite and re-executing *DAMAt*, the metrics should become the following:

- *Fault Model Coverage* 100%
- *Mutation Operation Coverage* 100%
- *Mutation Score* 61%

Test 03: Source (src) and Destination (dst)

A test case containing an oracle that checks the content of *conn->idin.dst*, *conn->idin.src*, *conn->idout.dst*, *conn->idout.src* should detect eventual alteration of these values between server and client, and kill the mutants emulating these kinds of faults, that were previously *APPLIED* but not *KILLED*.

The test consists of the client sending 5 messages to the server. For every message, the content of the aforementioned members of the *csp_conn_t* data structure shall be checked for discrepancies between server and client.

The test case is implemented in the file

```
test_suite/test_03/test_03.c
```

and can be executed by compiling *libCSP*, moving to the *test_suite* folder, and executing the following command:

```
make test_03
```

Then you can add the test to the *damat-pipeline/tests.csv* so that it will be recognized and executed by *DAMAt*: The file should now contain these lines:

```
test_01,10000
test_02,10000
test_03,10000
```

After adding *test_03* to the test suite and re-executing *DAMAt*, the metrics will become the following:

- *Fault Model Coverage* 100%
- *Mutation Operation Coverage* 100%
- *Mutation Score* 67%

Test 04: Source Port (sport) and Destination Port (dport)

A test case containing an oracle that checks the content of *conn->idin.sport* and *conn->idin.dport* should detect eventual alteration of these values between server and client, and kill the mutants emulating these kinds of faults, that were previously *APPLIED* but not *KILLED*.

The test consists of the client sending 4 messages to the server. For every message, the content of the aforementioned members of the *csp_conn_t* data structure shall be checked for discrepancies between server and client. In particular, the server-side *dport* shall coincide with the client-side *sport* and vice-versa.

The test case is implemented in the file

```
test_suite/test_04/test_04.c
```

and can be executed by compiling *libCSP*, moving to the *test_suite* folder, and executing the following command:

```
make test_04
```

Then you can add the test to the *damat-pipeline/tests.csv* so that it will be recognized and executed by *DAMAt*: The file should now contain these lines:

```
test_01,10000
test_02,10000
test_03,10000
test_04,10000
```

After adding *test_04* to the test suite and re-executing *DAMAt*, the metrics will become the following:

- *Fault Model Coverage* 100%
- *Mutation Operation Coverage* 100%
- *Mutation Score* 81%

Test 05: Flags (flags)

A test case containing an oracle that checks the content of *conn->idin.flags* and *conn->idin.flags* should detect eventual alteration of these values between server and client, and kill the mutants emulating these kinds of faults, that were previously *APPLIED* but not *KILLED*.

The test case is implemented in the file

```
test_suite/test_04/test_04.c
```

and can be executed by compiling *libCSP*, moving to the *test_suite* folder, and executing the following command:

```
make test_04
```

Then you can add the test to the *damat-pipeline/tests.csv* so that it will be recognized and executed by *DAMAt*: The file should now contain these lines:

```
test_01,10000
test_02,10000
test_03,10000
test_04,10000
```

After adding *test_05* to the test suite and re-executing *DAMAt*, the metrics will become the following:

- *Fault Model Coverage* 100%
- *Mutation Operation Coverage* 100%
- *Mutation Score* 95%

Conclusion

This example is intended to show that, with the help of DAMAt, a user can obtain valuable indication on how to improve a test suite. The new test suite for *libCSP*, while intentionally still very limited and simple, should be more capable of identifying problems in the SUT than it was at the start.