CETIC
Faults injections framework : Documentation

Alex Vermeylen

May 31, 2015

Contents

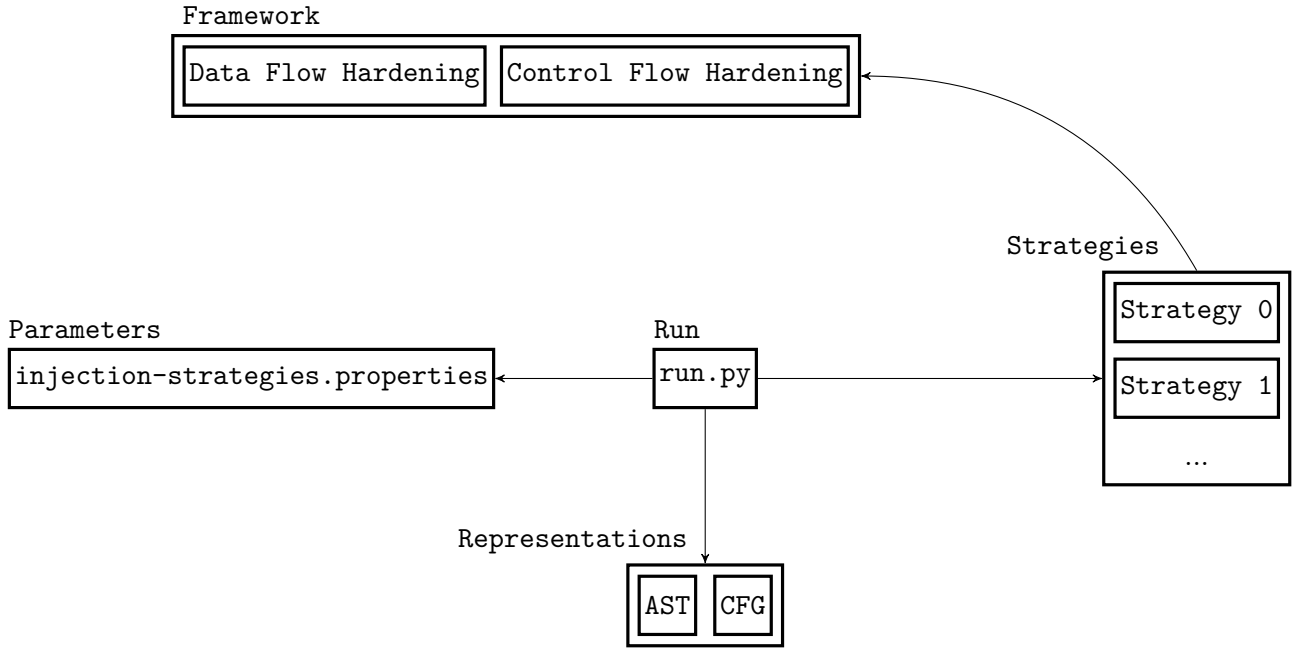
1 The tool

The philosophy of this testing tool is to check the validity of a hardened program. If a hardener has been used to make the tested program robust, this testing approach can also allow one to evaluate the hardener. In this case, the tested programs have been hardened using the CETIC's hardener, even if the tool is more generic and can be used with a wider variety of source code.

The generated tests are the last step in a testing process. Indeed, for it to work, a set of unit tests are required. The coverability of these tests are supposed to be maximum. They also prove the equivalence between a program and its hardened counterpart. Of course, they also prove the correctness of the program under test. The only role of this last step is to check the robustness capabilities of the program and, indirectly, to challenge the hardening rules.

The test strategies can be customized in order to be adapted to the code and/or to the hardening rules. Also, the number of test cases can be reduced using these strategies.

2 Architecture



`run.py` is the main script. It makes the link between all the other parts of the tool. It also generates the AST and CFG. Moreover it manages parameters from the properties file, and give them to the strategies.

The framework part contains all the function needed to generate tests scripts.

The representation part allows to represents an AST and CFG.

The strategies are defined by the tester and called by the main script. They handle the generation of tests scripts according to the desired parameters and the way they are implemented.

3 Sources

- **controlFlowGraph** : contains all scripts related to the generation and representation of basic blocks and control flow graph.
  - `cfg.py` : Contains the definition of `BasicBlock` and `CFG` objects.
  - `COntrolFlow.py` : Manage the generation of the CFG of a function based on its AST.
- **cunit** : contains the files related to the transformation of cunit tests to be adapted to hardened code.

  - `transform_cunit.py` : Manage the transformation.
  - **utils** : Contains all the `.h` files that need to be included in the transformed version of the cunit tests.

- **hardening** : Contains all files related to the generation of fault injection scripts.
  - **dataHardening** : contains all files related to the generation of GDB scripts to inject fault on data flow.
    * `tests_dataHardening.py` : Contains all functions ot generate GDB scripts and analyze AST.
    * **utils** : contains all custom GDB commands and listeners for data flow.
  - **controlFlowHardening** : contains all files related to the generation of GDB scripts to inject fault on control flow.
    * `tests_controlFlowHardening.py` : Contains all functions ot generate GDB scripts.
    * **utils** : contains all custom GDB commands and listeners for control flow.

    Strategies : Contains all definition of strategies.
  - **utils** : Contains all custom GDB commands and listener for both data and control flow. Also contains fake libs to include for pycparser and the xsl stylesheet needed to produce a HTML results file from the XML one.
  - `injection-strategies.properties` : Contains all parameters for generation of script and to parametrize strategies.
  - `run.py` : Main file that coordinates the generation. It is the central point between the properties file, the selected strategies and the framework.

4 Dependencies

- python 3 : Because the tool is written in python, the version 3 of python is required.
- scons : scons is used to build the source code in order to be executed with gdb and perform the fault injection. Note that if you just want to generate the fault injection script, scons is not necessary. scons can be installed following the instruction given on the project website : `http://www.scons.org/` . If you use a Linux distribution, the scons package can probably be installed through your package manager.
- pycparser : pycparser is the parser used to parse the C source code. It can be downloaded and installed following the instructions given on the github page of the project : `https://github.com/eliben/pycparser` .

5 Generation of faults injection scripts

This section will detail the generation of the GDB scripts that allows to inject faults. We will discuss here the use of each python file used for the generation of the tests and the relations between them.

5.1 Representations

5.1.1 Abstract Syntax Tree

To analyze each instruction of each function body one by one, we need to build a AST from the source code. To do so, the code need to be parsed in order to build the tree. In this context, pycparser was chosen.

pycparser use the grammar form the C99 dialect of the C language and handle the parsing of the library header in a convenient way (by the use of "fake headers", which allows better performance. see the pycparser website for more informations). The AST build by pycparser contains the necessary informations to achieve our goal. Moreover, it is written in python. Since GDB can be extended using the python language, using a parser written in the same language could allow us to use it directly with GDB. No such use has been used in the context of this work but this could allows some extensions.

5.1.2 Control Flow Graph

The Control Flow Graph is also essential to produce all kinds of CFEs. To build it, no adapted existing tool was found. So, a python script was written to construct the graph, based on the AST produced by pycparser. Two files are involved here : `cfg.py` and `ControlFlow.py`.

`cfg.py` contains the definition of two objects : `BasicBlock` and `CFG`.

As its name states, the first object represents a basic block. It has a unique id (the id 0 allows to recognize the root of the CFG containing the block), the coordinate of the first and last line of the block (coordinates contain the file name and line number), the list of all the statement contained by the block, a list of parents and children blocks and a condition (in the case of a conditional statement finishing the block, it is set to `None` otherwise). Several methods are needed to modify blocks during the visit of the AST in order to build the CFG. So, once a `BasicBlock` has been build, it is possible to add new statements to it (following the statements already stored in the block), add new child, get the first or the last statement of the block or set the condition in case we encounter a conditional instruction. Other methods are available to print a block and compare it ith other blocks, based on their id number.

The `CFG` object allows to represents the entire Control Flow Graph consisting of the basic blocks described above. A `CFG` object consists of a count value that hold the number of blocks contained int the CFG and a dictionnary containing all these blocks. The keys of the dictionnary are the line number of the coordinate of the first statement of the basic block. That allows us, thanks to a `getBlock(self, lineNbr)` method, to get the block in which an instruction is by giving its line number to this method. Other methods are also available to add blocks, get the root block and print the graph or generate a dot file that can be tranformed to a png image (mainly used for test purpose).

In the context of the generation of the fault injection scripts, `ControlFlow.py` is used to visit the entire body of a given function and create the CFG of this function. The principle is quite simple. For each statement form the function body, we had it to the current block and modify the block according to the type of this statement. For example, if the statement is an "if statement", we had the condition to the block, we close the current block and create a new one for the following statements ; we also continue with the statements from the "if body" before continuing with the following instructions. Eaxh time a block is closed, it needs to be added to the CFG and the links between it and the already existing blocks need to be defined. Of course, the actions to make depends strongly on the type of the statement and a lot of special cases need to be taken into account.

5.2 Framework

5.2.1 Data flow

The `tests_dataHardening.py` file is used to generate all scripts related to the simulation of data flow errors. It also contains a special function to generate a dictionnay that contains all statements (or part of statements) on which an injection could be performed.

The statements dictionnary (`stmtsDico`) is produced by `getStatementsDictionnary` based on the part of the AST that represents a function body. It is divided into three smaller dictionnaries : `stmtsDico['left']` that contains statements which should be corrupted *a posteriori*

(e.g. left part of an assignation), `stmtsDico['right']` that contains statements which should be corrupted *a priori* (e.g. right part of an assignation) and `stmtsDico['op']` that contains statements updated with the use of an operator. `stmtsDico['left']` and `stmtsDico['right']` contains a special entry : `stmtsDico['right']['variablesNames']` which is a list of all different variables that are contained in the sub-dictionnary. Based on that, the statements list for a specific variable can be obtained by selecting the corresponding entry. For example `stmtsDico['left']['a']` for a variable named 'a'. For the operators, it is almost the same except that `stmtsDico['right']['variablesNames']` is replaced by `stmtsDico['op']['operators']`.

Other functions in this file are used to generate each kind of GDB script regarding the corruption of the data flow. The common parameters are the statement (or list of statements) on which to perform the injection, the bits to flip and the "function directory" which is the folder in which the test will be written. This is a mandatory parameter for strategies and it is automatically given so the user does not need to do something special about it. Also, the functions related to data storage have 'next'. If it is set to True, the injection is made *a posteriori*, else it is performed *a priori*. Finally, for the permanent fault scripts generation, the variable or operator need to be given.

Other utils functions, used by the functions described above are presents.

5.2.2 Control flow

The `tests_controlFlowHardening.py` file is used to generate all scripts related to the simulation of control flow errors. There is one function for each kind of CFE that takes one or two basic block(s) as parameter depending on the type of CFE, and as for the data hardening part, the "function directory". Also, for CFE of type 3, 4 and 5, the percentage of lines inside the blocks that need to be jumped from on on is given.

Other utils functions, used by the functions described above are presents.

5.3 Strategies

the framework can be used to generate several different kinds of GDB test scripts. The way these tests are generated, the parametrization of the generation and the number or kind of tests that produced are defined by a strategy.

5.3.1 Define a strategy

A strategy is no more than a python function inside a python file put into the "strategies" folder. This function has three mandatory parameters : stmtsDico, cfg and funDir. They are automatically given to the strategy function when the generation is launched. The user can also define other arguments.

5.3.2 Parameters

The optional parameters need to be specified into the `injection-strategies.properties` file, in JSON format. Just add your strategy name (which must be the same as the name of the python file defining your strategy, without the extension) into the "strategies" JSON object. You will then crate a new JSON object containing all your parameters.

5.4 Generation

A main file, named `run.py` following the common convention, manage all the generation. It can be run like that :

```
1   python run.py --strategies name_strategy_1 name_strategy_2
```

You can combien as much strategies as you want. The other parameters are : the files on which the injections will be performed, the folder in which generate the tests, the other files that need to be compiled to run the test and the Cunit file to tranform. Only the first is mandatory. If no folder is given, the tests will be written in `$HOME/FaultInjectionTests`. These parameters are defined in the `injection-strategies.properties` file.

`run.py` file has several roles :

- it generates the necessary representations to be used by the strategies to generate the tests. These representations are the Control Flow Graph, the statements dictionnary used by the dataflow part of the framework and the Abstract Syntax Tree, necessary to create the statements dictionnary.
- it creates all the directories based on the 'folder' argument given in the properties file.
- based on other parameters, it creates a Scons files. A Scons file has approximatively the same use as a makefile and will be used to compile the necessary files in order to run the tests.
- it initializes the run scripts specific to each kind of faults. These run scripts will be completed when the tests are generated and launched by the mai run script when one wants to run the tests.
- it goes through the Abstract Syntax Tree, extract the functions for which GDB script will be written and creates the statements dictionnary needed by the strategies. Then, for every one of these functions, it run the selected strategies.

5.5 Unit tests

The fault injection framework is designed to work with unit tests in order to be able to test that the fault detection mechanisms have worked properly for a given injection but also to check the value returned by the function under test. A file containing a `main` function performing only one unit test can be provided for this purpose. However, it is common to have a single file grouping several unit tests. An option has been designed to handle such file. However, this solution is very specific to the hardener developped by the CETIC.

The tester can provide a file containing unit tests written using the CUnit framwork through the properties parameter. These tests should have been designed for the non-hardened version of the code under test. The `transform_cunit.py` file will then manage to tranform the unit tests to make it adapted to the corresponding hardened code and also add some features to make him compatible with the fault injection test framework. This new unit test file will be called `cunit_test.c` and will be added to the root of the directory in which the tests will be writte. Also, to make this tranformation, some C function has been needed to create in order to add them to the tranformed CUnit test file.

Note that due to confidentiality reasons, this part cannot be described on a public repository. Thereby, as this feature has been removed, the cunit option should not be used with the version provided on this repository.

6 Tests scripts

This section will explain how the GDB test scripts are launched once they have been generated. The general organization and operation will also be explained.

6.1 Organization

To contain and organize the tests scripts, some folders are created. The root directory contains the general run script and the Scons file. The root directory also contains one subdirectory for each function on which fault injections will be performed. In each of these subfolders, there is a run script to launch test for a particular function. There are also three subdirectories : one for transient faults on the data flow, one for permanent faults on the data flow and one for Control Flow Errors. These directories contains subdirectories to organize the tests scripts. For the data flow folders, there are two subdirectories that differentiates between tests on data stroage and processing. For the Control flow directories, there are five subdirectories, one for each type of CFE. These subfolders are the lasts of the tree and holds the GDB tests scripts and a run script that allows to run all the tests contained in the directory.

6.2 Run the tests

To run the tests, one just need to run the main run script from the root directory (the directory chosen to generate the test in). This script will compile the program through the Scons script, and run the run scripts specific to each function folders. These scritps will then run the ones specific to each kind of faults for the functions the folder they are in represents. After running the tests, the root directory will contains the results files and the compiled version of the program to test. The functions subfolder will also contain a results file. Finally, the subdirectories containing the GDB scripts will contain one result file per test.

The command to run, from the root directory is :

```
python run_all_tests.py
```

6.3 Results

The global and function specific results files conrtains the following information :

- Test name
- How much breakpoints have been hit
- The unit test result
- the fault injection results

The results file are in XML format. However, a more user-friendly HTML verison of the global results file is also available. Here is an example :

Test	Breakpoint	Test result	FI Result
transientDataStorage_a_8_0	1	1	FI detected
transientDataProcessing_b_28_0	1	0	Failed
permanentDataStorage_c	0	0	No injection
CFE3_0-to-0_10	1	0	IGSEG

As you can see, there are four different results type for the fault injection results :

- Green means that the test passed, i.e. an injection has been permformed and detected.
- Red means that the test failed, i.e. an injection has been permformed but not detected.
- Orange means that no injection has been performed.
- Yellow is the default color if anythin else happens. In general, as in this example, a signal has interrupted the execution.

7 GDB python extensions

In order to add functionnalities to our generated fault injection scripts, we used GDB to define custom GDB commands, GDB functions and event listeners.

We will begin by explaining the different convenience variables used by the custom python extensions. A convenience variable is a variable that is declared and initialized into GDB and which can only be used by GDB without interfering with the C program. After that, we will describe the custom GDB command used here. Finally, the implemented event listener and their use will be discussed.

7.1 Convenience variables

Some convenience variable are used inside the injection scripts, mostly to retain informations that will be needed to write the results file. The specificity of such a variable is that its name alway begin with the dollar sign. These variables and their use are detailed below :

- `$bphit` : this variable is set to 0 at the begining of the script. Every time a breakpoint is hit and the injection is performed, it is incremented by one. That is finally used when the resutls are written. Indeed, if no error detection mechanisms have been triggered, that could be because no injection has been performed. So, in addition to writte the number of time an injection has been performed in the result, we can also write that no injection has been performed if the value held by `$bphit` is zero.
- `$bpnum` : every time a breakpoint is set, GDB assign it a unique breakpoint number. Because the number of the current breakpoint cannot be known inside our custom commands, we use the `$bpnum` variable, which is set to the current breakpoint number as soon as a breakpoint is hit. This information is then used with the internal GDB command "disable" that allows to disable breakpoint in order to simulate transient faults. For this purpose, temporary breakpoint cannot be used because, in the case of a CUnit test file the breakpoint needs to be reenabled between every unit test. Through this technique, the breakpoint is "manually" disabled once the injection has been performed and can be reeanbled before another test is executed.
- `$xml` : contain the name of the xml result file that need to be written in the results files. More informations about tests will be given into the corresponding section.
- `$fun` : Hold the name of the function on which the test inject a fault. This is used in the results files.
- `$testresult` : this variable is updated to the result of the test once it has finished executing. It is used to write the result value into the results files.

7.2 GDB Commands

The custom GDB commands that are used by the generated script will be decribed here. First, the custom commands for the control flow script will be detailed, followed by the ones used in the data flow fault injection script. Then, the general commands used by the two kind of script will be discussed.

Before going into details, an important thing to know is how a GDB command manage the parameters given to it. Imagine a `sum` command that return the sum of its arguments. It will be called like that :

```
sum 1 2 3
```

In this case, one can think that the sum command is called with three arguments. However, it is actually called with one argument wich is the "1 2 3" string. The python script handling the command will have to split the string to separate the three numbers and cast them before doing the sum.

7.2.1 MakeJump

The `MakeJump` command was implemented in order to write the scripts injecting faults into the control flow. Once a jump need to be written into a script, it is this command that is used. The string that serves as paramter contains two informations : the arrival coordinate, in the form "fileName:lineNumber", and the "next" string. The first thing done by the python script is to split the string on the space characters to isolate the two distincts parameters. The first is used as such to execute the internal GDB command "jump" that allows to jump to the desired instruction. The second argument, the "next" string, is used as a flag. If it is present, the script will execute the internal GDB command next that allows to execute the current line before making the jump. Indeed, when the script is generated, the breakpoint is set on a specific line, but we want the jump to be performed after this specific line has been executed. For example, in the case of the type 1 CFEs, the start basic block must be executed entirely, including the last statement, even if the breakpoint is set on this last instruction. A "next" should be performed. Moreover, this "next" command must be executed inside the python script and cannot be added to the `commands` block associated to the breakpoint because, once the next command is executed inside a breakpoint, GDB stop the executions of other commands in this block and continue the execution of the program. If we wrap the "next" inside the custom command, GDB will not act like that, and will allow the python command to continue.

Once the next command has been called, the jump is performed. Then, the convenience variable `$bphit` is incremented. Afterwards, the current breakpoint is disabled. As stated before, a specific jump is only performed once, simulating only transient faults, because a permanent one could eassily lead to an infinite loop (e.g. we jump to a previous instruction). Finally, the internal GDB command "continue" is run in order to exit the breakpoint and continue the execution.

7.2.2 corrupt

The `Corrupt` command is used inside script affecting the data flow. It allows to flip one or more bits from the variable on which the fault is injected. It needs to be called like that :

```
1  corrupt variable_name nbrBits bits_to_corrupt [next] [disable]
```

As for the `MakeJump` command the parameter string is plitted in order to isolate all the arguments. The first is obviously the name of the variable that needs to be corrupted. The second is the number of bits that will be flipped. The third is actually more than one argument, it is the list of all specific bits that will be flipped (e.g. with "1 4 8" , the first, fourth and eighth bits will be flipped during the injection). If the user wants tthe bits to be chosen at random, this argument corresponds to "-1". Afterwards, the optionnal "next" argument has the same use as for the `MakeJump` command. Finnaly, the optionnal "disable" parameter is given if the breakpoint need to be temporary.

Once the arguments have been splitted, the `corrupt` command look at them and build the bits lists from the chosen bits to corrupt. Then it executes the "next" commands if the corresponding argument was set. Then, it performs the injection by calling the `corrupt` **function**. This function takes three argument, the name of the variable to corrupt, the number of bits to flip and the list of these bits. It simply transform the variable into a string representing it in binary form. The variable is converted to a long python type, because it is the biggest type given by python. This leads to a too high number of bits in the binary representation. However, as we

know the number of bits used to reprents the C type of the variable thanks to the GDB python API, we are sure that the unused bits will never be flipped. Then, this binary representation will be used to find the bit(s) to flip and modify them. The string is then transformed into the long type and return by the function. After the call to this function, we return to the `corrupt` command which will increments the `$bphit` variable, disable the breakpoint if the corresponding argument was given and finally execute the "continue" command to exit the breakpoint and resume the program execution.

7.2.3 XMLInit

This simple command is launched before any fault injection has been performed and initialize the result file specific for the current test.

7.2.4 WriteResult

This function is used to write relevant information in the general and individual results files once the test has finished. Its argument should be the string that correspond to the result (like "failed", "passed", etc.). See the next section for more informations about the results files.

7.3 events listeners

The GDB python API allows one to define special listener that are triggered when a special event occur. To do so, one need to define a function which speifications specific to the event he wants to catch. Then this function needs to be connected to the desired event. Once this event occurs, the function will be launched and the needed parameters will be given to it. Note that this function will be executed before anything else.

7.3.1 bp_listener

This listener is triggered when a breakpoint is hit. The associated function must take a `gdb.Breakpoint` object as parameter. From this paramter, it is then possible to access the breakpoint number, used to update the `$bpnum` variable. Once it is done, the instructions inside the `commands` block related to the breakpoint are executed.

7.3.2 exit_listener

This listener is triggered when the program has finished its execution. From the parameter passed to the associated function, it is possible to have the return code of the program. Based on its value and the value of the `$bphit` variable, the `WriteResult` command is launched with the string corresponding to the test result to write into the results files. Also, the `$testresult` variable is updated. That will allow `WriteResult` to use this value to put it into the results files.

Note that it cannot be used with a CUnit test file, because the program does not stop between each test.

7.3.3 signal_listener

This listener is triggered when the program has been interrupted by a signal. First, the signal is identified. Then the `WriteResult` is called with the signal type as argument to report the interruption into the results files.