D2.4

Collaboration of analyses intermediate release V2

Project number:	731453
Project acronym:	VESSEDIA
Project title:	Verification engineering of safety and security critical dynamic industrial applications
Start date of the project:	1st January, 2017
Duration:	36 months
Programme:	H2020-DS-2016-2017

Deliverable type:	Other
Deliverable reference number:	DS-01-731453 / D2.4 / V1.0
Work package contributing to the deliverable:	WP2
Due date:	Aug 2018– M32
Actual submission date:	11th September 2019

Responsible organisation:	CEA
Editor:	Virgile Prevosto
Dissemination level:	PU
Revision:	1.0

Abstract:	Companion report describing software delivered as D2.4
Keywords:	Combining static and dynamic analysis, AFLSCA, StaDy, SARIF

The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

Editor

Virgile Prevosto (CEA)

Contributors

Balázs Berkes (SLAB)

Gergely Eberhardt (SLAB)

Disclaimer

Executive Summary

This document gives a brief overview of the final version of the tools developed to accomplish software analyser collaboration in WP2 of the VESSEDIA project. The main part of the deliverable consists in the software themselves, which are submitted separately.

Contents

List of Figures

Chapter 1 Introduction

This document presents the two prototypes composing D2.4, and related to tasks T2.4 and T2.5. It is the continuation of D2.2 [4], which presented an earlier version of the tools described in the current report. As in D2.2, regarding T2.4, the collaboration between analyzers, and more precisely between static and dynamic analysis is exemplified by the use of the AFL fuzzer to generate test cases that will falsify an ACSL property that could not be proved by the WP plugin (see Chapter 2). For T2.5, the MarkDown Report is presented in 3.1. It provides two ways for presenting the results of the Eva and WP plug-ins of Frama-C into semi-structured formats that can be understood by external tools thus enabling potential (further) coupling of analyzers and the incorporation of the results into broader security reports.

Chapter 2 Combining Static and Dynamic analysis (SLAB)

2.1 General Description

In the STANCE [6] project SLAB integrated the FLINDER fuzzer tool with the SANTE module. The combined result used value analysis, program slicing and structural testing for C program verification and used the FLINDER fuzzer to dynamically confirm each alarm (see section 2.2 in [1]). In the current version of Frama-C, the SANTE method is replaced by the StaDy plugin (section 2.3 in [1]), which generalized the SANTE method and made possible to combine any static analysis plugin with the dynamic analysis.

The original StaDy plugin uses the PathCrawler tool to generate test cases using constraint solving, which is an NP-complete problem. Therefore, even if PathCrawler can ensure coverage, the termination of the tool cannot be guaranteed within reasonable time in every case. To address this problem, we aimed to replace PathCrawler with AFL[1] (American Fuzzy Lop), which is a widely used fuzzer capable of increasing coverage with compile-time instrumentation and genetic algorithms. Although it can handle complex cases also, it cannot guarantee full coverage, so both the PathCrawler and the AFL test generation can be meaningful in different cases.

2.2 Installation instructions

The AFLSCA tool is consisting of the following software modules:

- Modified StaDy plugin
- AFLSCA plugin
- AFL

2.2.1 Installing Frama-C v19.x – Potassium

Using OPAM package manager, the basic installation can be done with the following steps on Ubuntu Linux:

```
wget https://frama-c.com/download/frama-c-19.0-Potassium.tar.gz
tar -xvf frama-c-19.0-Potassium.tar.gz
cd frama-c-19.0-Potassium
opam init
opam pin add --kind=path frama-c .
```

Note that in older version of Frama-C, there used to be a `frama-c-base` package, which is now obsolete. Only the frama-c package should be installed.

For installing missing Frama-C dependencies:

```
opam install depext
opam depext frama-c
```

For more detailed installation instructions see Frama-C chapter in [3] and [5] or the full installation guide at the Frama-C site[2].

[1] http://lcamtuf.coredump.cx/afl/
[2] https://frama-c.com/install-potassium-20190501.html

2.2.2 Adding StaDy plugin

Download StaDy plugin from the GitHub repository[3]. The repo's chlorine branch must be used for Frama-C v19.x (Potassium). The building process is the following:

```
git clone https://github.com/vprevosto/Frama-C-Stady
git checkout potassium
cd Frama-C-StaDy
 autoconf
./configure
make
make install
```

Note however that StaDy requires PathCrawler, which is not freely available.

If the installation was successful, then the StaDy plugin can be used with Frama-C:

```
frama-c <file> -main <entry point function> -stady -stady-socket stdio
```

2.2.3 Install AFLSCA plugin

The plugin has a makefile included with it which sets up everything needed including preparing AFL for usage the only prerequisite is python3 which, on Linux, can usually be installed using the system standard package manager, e.g. on Debian, Ubuntu and their derivatives:

```
sudo apt-get install python3
```

The plugin can be installed with the makefile as follows from the archive attached:

```
cd plugin
make
make install
```

alternatively, it is possible to simply launch the packaged install.sh script.

[3] https://github.com/vprevosto/Frama-C-StaDy/

2.3 AFLSCA design

First of all, the tool runs StaDy on the given file and entry point function. The generated __sd_*.c __sd_*.pl files are interpreted and translated into a json file. From the generated json file, the entry point's argument list, types, and ranges will be examined. For test cases random initial values are generated. These test cases contain only single lines, in which the initial variable values are separated with keywords (specified in an AFL dictionary). An automatically created (wrapper) source file will call the (entry point) function under test with the corresponding arguments. This created (wrapper) file and the original source file will be linked together (with afl-gcc). The tool then calls the AFL fuzzer, the initial values will be fuzzed, and then it looks for crashes reported. After a timeout, the tool closes the AFL fuzzer, then with the help of GDB the generated crashes will be examined and displayed.

2.4 Usage

For the following usage example we will use this c code, which contains a stack-based buffer overflow if the input string larger than 7 bytes:

```
void function(char* input)
{
     int i=1;
     int j=2;
     char buffer[8];
     strcpy(buffer, input);
     printf("%x %x %s\n",i, j, buffer);
}

int stackof(char* input)
{
  int k=3;
  function(input);

  if (strcmp(input, "secret"))
  {
    puts("Access denied!");
    return -1;
  }
  else
  {
    puts("Access granted!");
  }
  return 0;
}
```

First we load the frama-c-gui and import the c file. After that we fill in the parameters of the plugin:

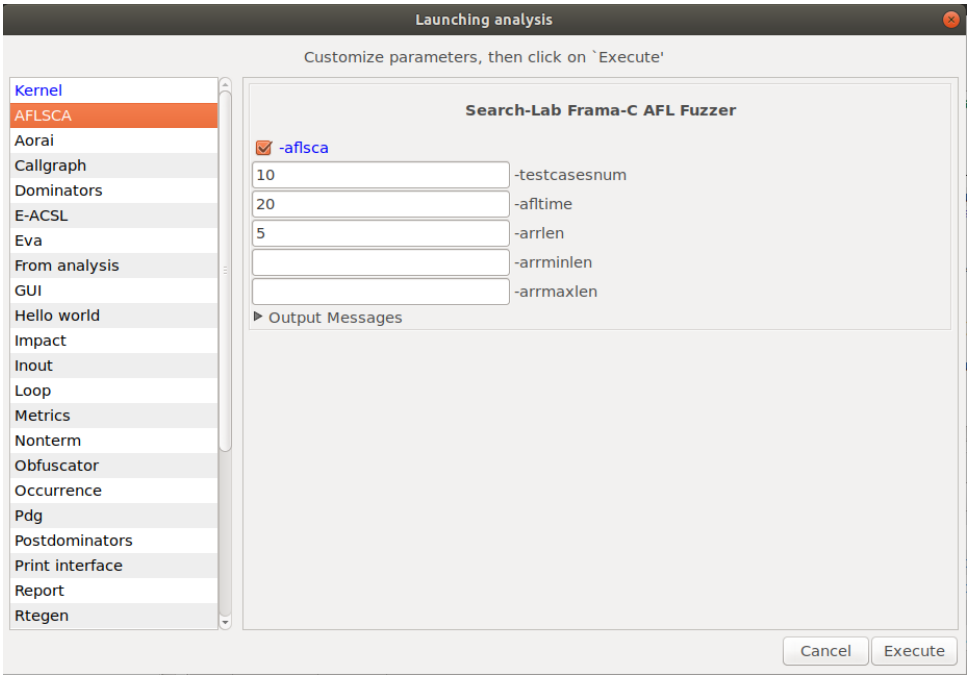

Figure 1. The plugin's settings

Then we execute the plugin. The AFLSCA calls the StaDy plugin, which generates a c and a pl file. The c file contains the original source with pre-processed variable names.

```
void function(char *input)
{
  char * const old___fc_p_strsignal;
  char * const old_ptr___fc_p_strsignal;
  char * const old___fc_p_strerror;
  char * const old_ptr___fc_p_strerror;
  char *old___fc_strtok_ptr;
  char *old_ptr___fc_strtok_ptr;
  unsigned short *old___fc_p_random48_counter;
  unsigned short *old_ptr___fc_p_random48_counter;
  int old___fc_random48_init;
  unsigned long const old___fc_rand_max;
  struct __fc_FILE * const old___fc_p_fopen;
  struct __fc_FILE * const old_ptr___fc_p_fopen;
  struct __fc_FILE *old___fc_fopen;
  struct __fc_FILE *old_ptr___fc_fopen;
  char *old_input;
  char *old_ptr_input;
  old___fc_p_strsignal = __fc_p_strsignal;
  old_ptr___fc_p_strsignal = __fc_p_strsignal;
  old___fc_p_strerror = __fc_p_strerror;
  old_ptr___fc_p_strerror = __fc_p_strerror;
  old___fc_strtok_ptr = __fc_strtok_ptr;
  old_ptr___fc_strtok_ptr = __fc_strtok_ptr;
  old___fc_p_random48_counter = __fc_p_random48_counter;
  old_ptr___fc_p_random48_counter = __fc_p_random48_counter;
  old___fc_random48_init = __fc_random48_init;
  old___fc_rand_max = __fc_rand_max;
  old___fc_p_fopen = __fc_p_fopen;
  old_ptr___fc_p_fopen = __fc_p_fopen;
  old___fc_fopen = __fc_fopen;
  old_ptr___fc_fopen = __fc_fopen;
  old_input = input;
  old_ptr_input = input;
  {
    char buffer[8];
    int i = 1;
    int j = 2;
    strcpy(buffer,(char const *)input);
    printf_va_1("%x %x %s\n",(unsigned int)i,(unsigned int)j,buffer);
    return;
  }
}
```

The pl file describes the type and range of function input parameters.

```
:- module(test_parameters).
:- import create_input_val/3 from substitution.
:- export dom/4.
:- export create_input_vals/2.
:- export unquantif_preconds/2.
:- export quantif_preconds/2.
:- export strategy/2.
:- export precondition_of/2.

dom(0,0,0,0).
dom('stackof', cont('input',_), [], int([-128..127])).
dom('stackof', cont(cont('__fc_stderr',_),0), [], int([0..4294967295])).
dom('stackof', cont(cont('__fc_stderr',_),1), [], int([0..4294967295])).
dom('stackof', cont(cont('__fc_stdin',_),0), [], int([0..4294967295])).
dom('stackof', cont(cont('__fc_stdin',_),1), [], int([0..4294967295])).
dom('stackof', cont(cont('__fc_stdout',_),0), [], int([0..4294967295])).
dom('stackof', cont(cont('__fc_stdout',_),1), [], int([0..4294967295])).
dom('stackof', cont(cont('__fc_fopen',_),0), [], int([0..4294967295])).
dom('stackof', cont(cont('__fc_fopen',_),1), [], int([0..4294967295])).
dom('stackof', cont(cont('__fc_p_fopen',_),0), [], int([0..4294967295])).
dom('stackof', cont(cont('__fc_p_fopen',_),1), [], int([0..4294967295])).
dom('stackof', cont('__fc_random48_counter',_), [], int([0..4294967295])).
dom('stackof', cont('__fc_p_random48_counter',_), [], int([0..4294967295])).
dom('stackof', dim(cont('__fc_env',_)), [], int([0..4294967295])).
dom('stackof', cont(cont('__fc_env',_),_), [], int([-128..127])).
dom('stackof', cont('__fc_strtok_ptr',_), [], int([-128..127])).
dom('stackof', cont('__fc_strerror',_), [], int([-128..127])).
dom('stackof', cont('__fc_p_strerror',_), [], int([-128..127])).
dom('stackof', cont('__fc_strsignal',_), [], int([-128..127])).
dom('stackof', cont('__fc_p_strsignal',_), [], int([-128..127])).
dom('pathcrawler__stackof_precond',A,B,C) :-
  dom('stackof',A,B,C).
create_input_vals('stackof', Ins):-
  create_input_val('__fc_random_counter', int([-2147483648..2147483647]),Ins),
  create_input_val('__fc_random48_init', int([-2147483648..2147483647]),Ins),
  create_input_val(dim('__fc_p_random48_counter'), int([0..4294967295]),Ins),
  create_input_val(dim('__fc_stdin'), int([0..4294967295]),Ins),
  create_input_val('__fc_errno', int([-2147483648..2147483647]),Ins),
  create_input_val(dim('__fc_p_fopen'), int([0..4294967295]),Ins),
  create_input_val(dim('__fc_strtok_ptr'), int([0..4294967295]),Ins),
  create_input_val('__fc_heap_status', int([-2147483648..2147483647]),Ins),
  create_input_val(dim('__fc_p_strerror'), int([0..4294967295]),Ins),
  create_input_val('__fc_mblen_state', int([-2147483648..2147483647]),Ins),
  create_input_val(dim('input'), int([0..4294967295]),Ins),
  create_input_val(dim('__fc_p_strsignal'), int([0..4294967295]),Ins),
  create_input_val('__fc_wctomb_state', int([-2147483648..2147483647]),Ins),
  create_input_val('__fc_mbtowc_state', int([-2147483648..2147483647]),Ins),
  create_input_val(dim('__fc_stderr'), int([0..4294967295]),Ins),
  create_input_val('__fc_rand_max', int([0..4294967295]),Ins),
  create_input_val(dim('__fc_stdout'), int([0..4294967295]),Ins),
  true.
create_input_vals('pathcrawler__stackof_precond',Ins) :-
  create_input_vals('stackof',Ins).
quantif_preconds('stackof',
  [

  ]
).
quantif_preconds('pathcrawler__stackof_precond',A) :-
  quantif_preconds('stackof',A).
unquantif_preconds('stackof',
  [

  ]
).
unquantif_preconds('pathcrawler__stackof_precond',A) :-
  unquantif_preconds('stackof',A).
strategy('stackof',[]).
```

```
strategy('pathcrawler__stackof_precond',A) :-
  strategy('stackof',A).
precondition_of('stackof','pathcrawler__stackof_precond').
```

After parsing the pl and c files, the AFLSCA starts the AFL in the console:

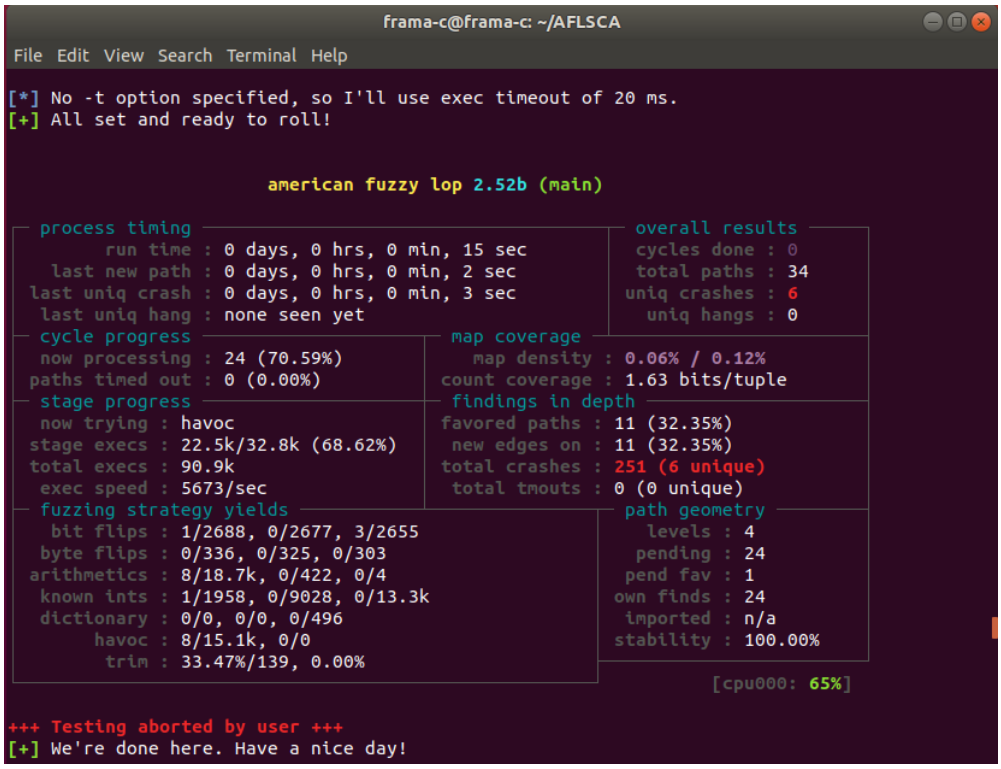

Figure 2. AFL running in the console

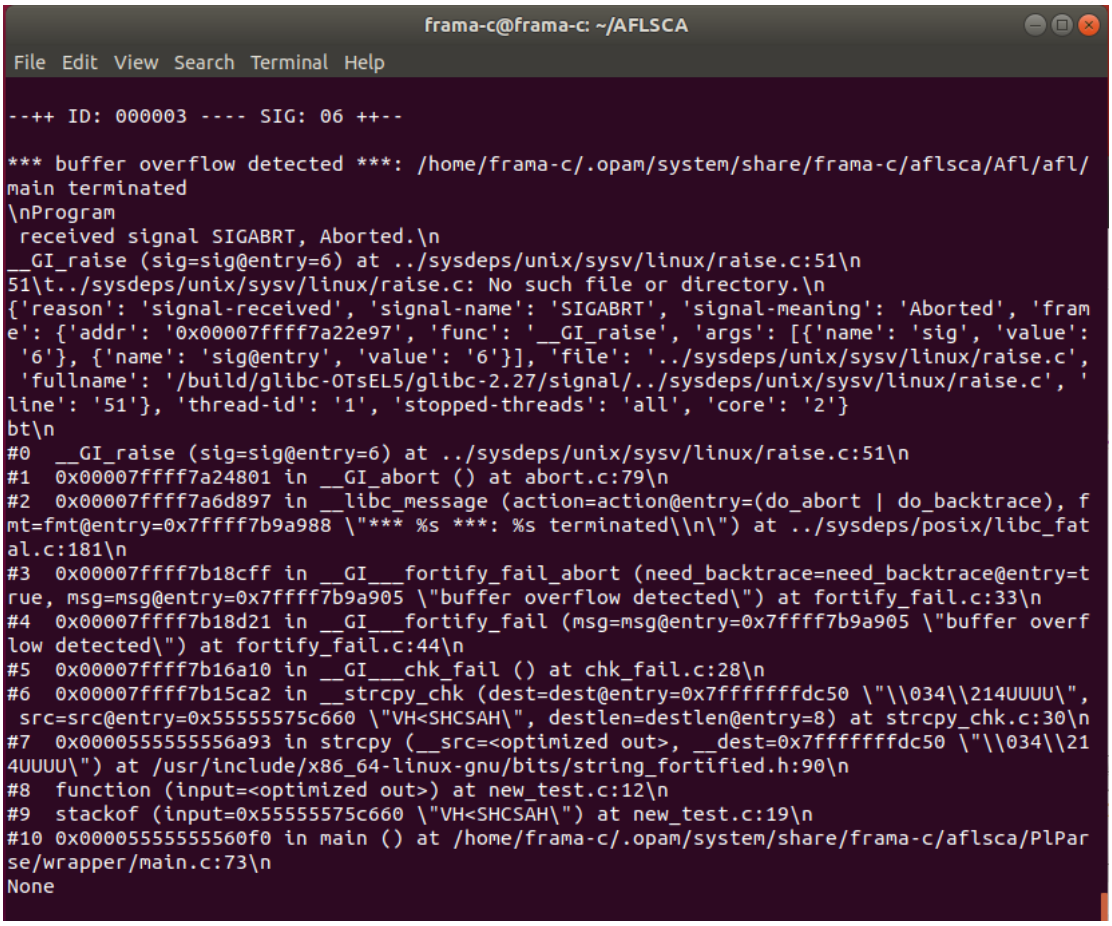

Figure 3. The crash results in GDB

After the plugin has completed execution we can see the results in the GUI in the messages tab:

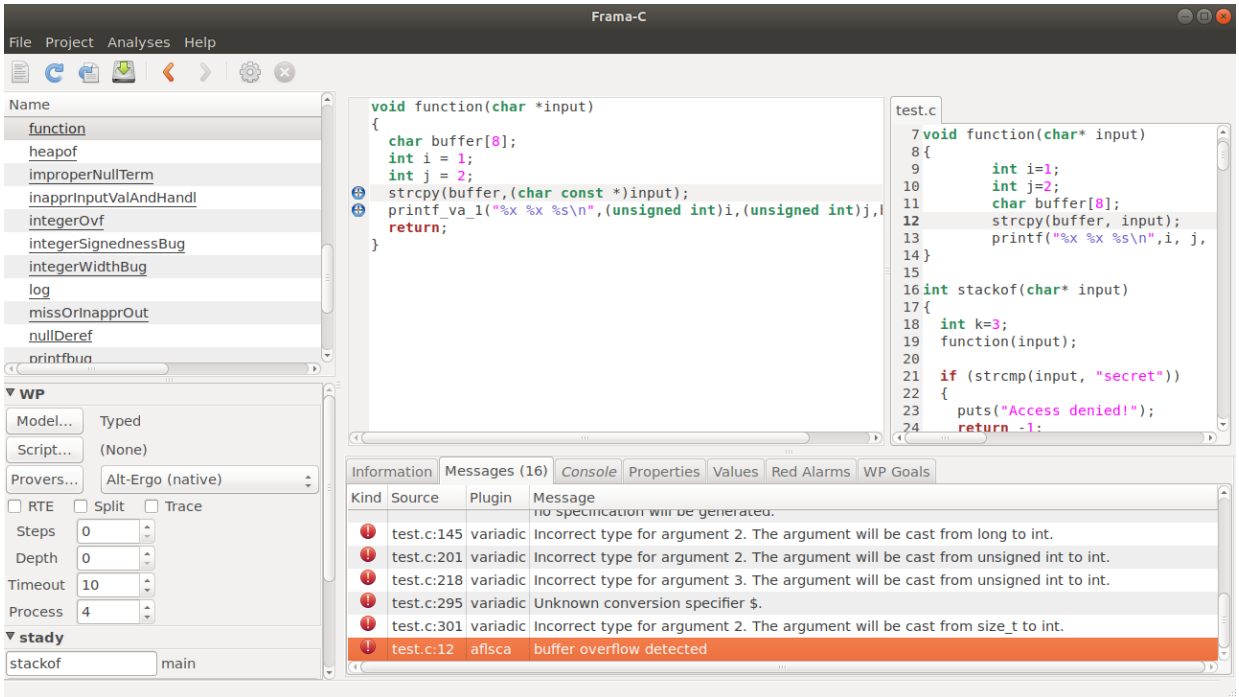

Figure 4. The plugin results

According to the AFL crash log, one of the input string, which caused crash was “hP&<>w>w&<”.

2.5 Usage examples

In this section we will demonstrate the usage of the analysis for C code on a given example from the VESSEDIA vulnerability taxonomy [2] and analyse it with a set of plugins to demonstrate how static and dynamic analysis can be combined using the existing and newly developed tools.

2.5.1 Example 1

Code:

```
int getValueFromArray(int *array, int len, int index)
{
  int value;

  // check that the array index is less than the maximum
  // length of the array
  if (index < len) {

    // get the value at the specified index of the array
    value = array[index];
  }
  // if array index is invalid then output error message
  // and return value indicating error
  else {
    printf("Error! The array size is only %d!\n",len);
    value = -1;
  }

  return value;
}

int arridxerr(int n)
{
  int array[] = {1337, 1336, 1338, 2014};
  int val = getValueFromArray(array,4,n-1);
  if (val!= -1)
    printf("The value of index %d is %d.\n\n",n,val);
  return 1;
}
```

Plugin setup:

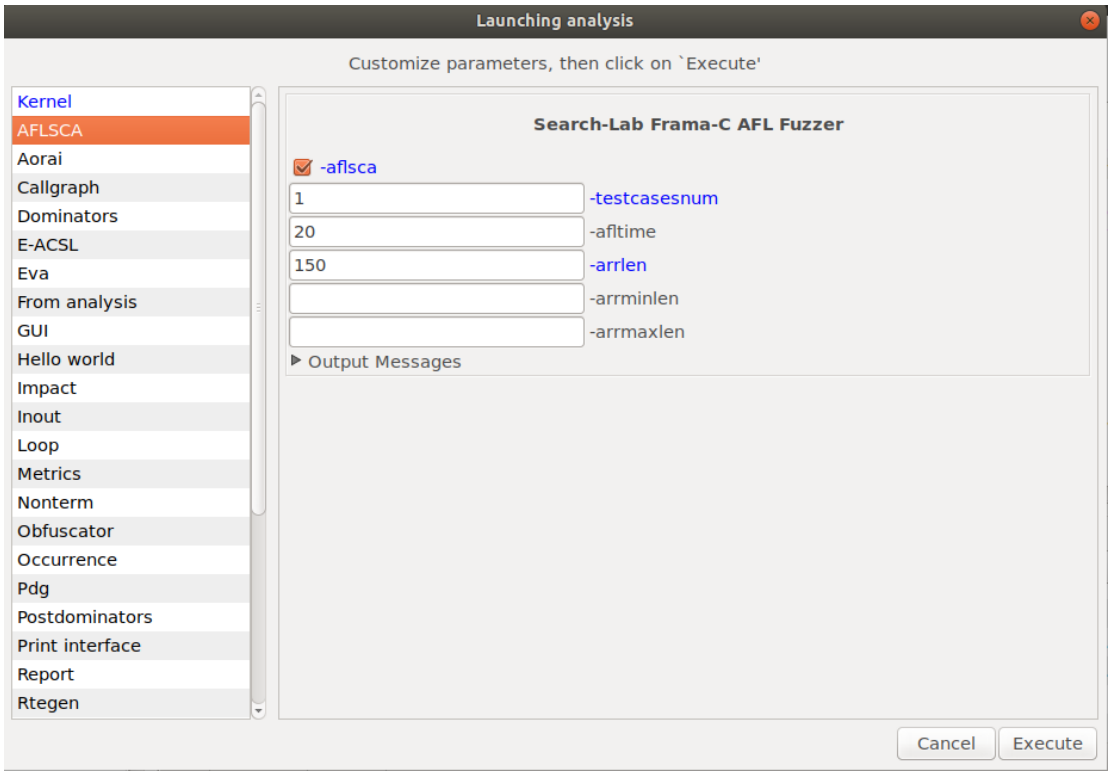

Figure 5: Plugin setup

The results in the frama-c-gui:

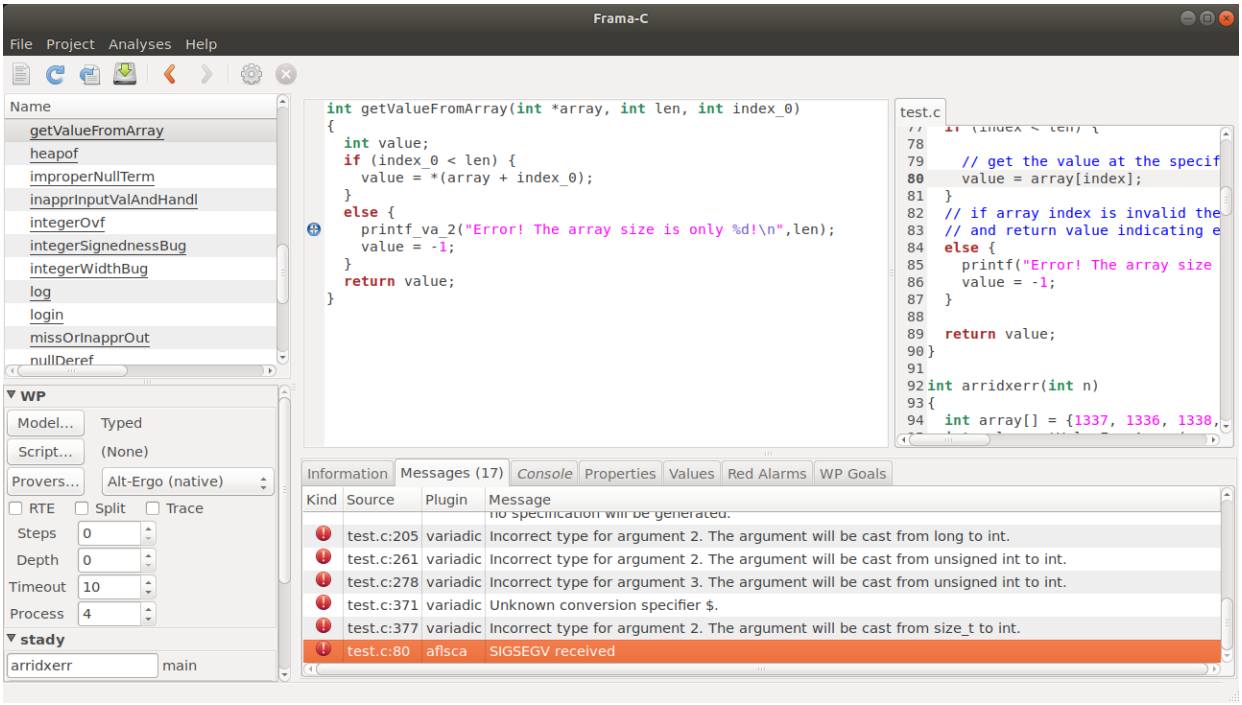

Figure 6. The results in the GUI

Description of the vulnerability:

The above code has an array indexing error [4] which can cause memory related problems if the right circumstances present.

This example demonstrates that AFL can detect array indexing errors.

The crash was happened for the following inputs:

- array = [1337, 1337, 1338, 2014]
- len = 4
- index_0 = -8

2.5.2 Example 2

Code:

```
void log(char* msg) {
  printf("LOG.INFO: %s \n", msg);
}

int missOrInapprOut(int argc, char *argv[])
{
  if(argc != 2) {
    printf("Usage: prog integer_value\n");
    return -1;
  }
  int value = strtol(argv[1], NULL, 10);
  if(!value) {
    char* new_cmd = (char*)malloc(strlen("Failed to parse val =
")+strlen(argv[1])+1);
    strcat(new_cmd, "Failed to parse val = ");
    strcat(new_cmd, argv[1]);
    log(new_cmd);
    free(new_cmd);
  }
  printf("Done\n");
  return 0;
}
```

[4] https://cwe.mitre.org/data/definitions/129.html

Plugin setup:

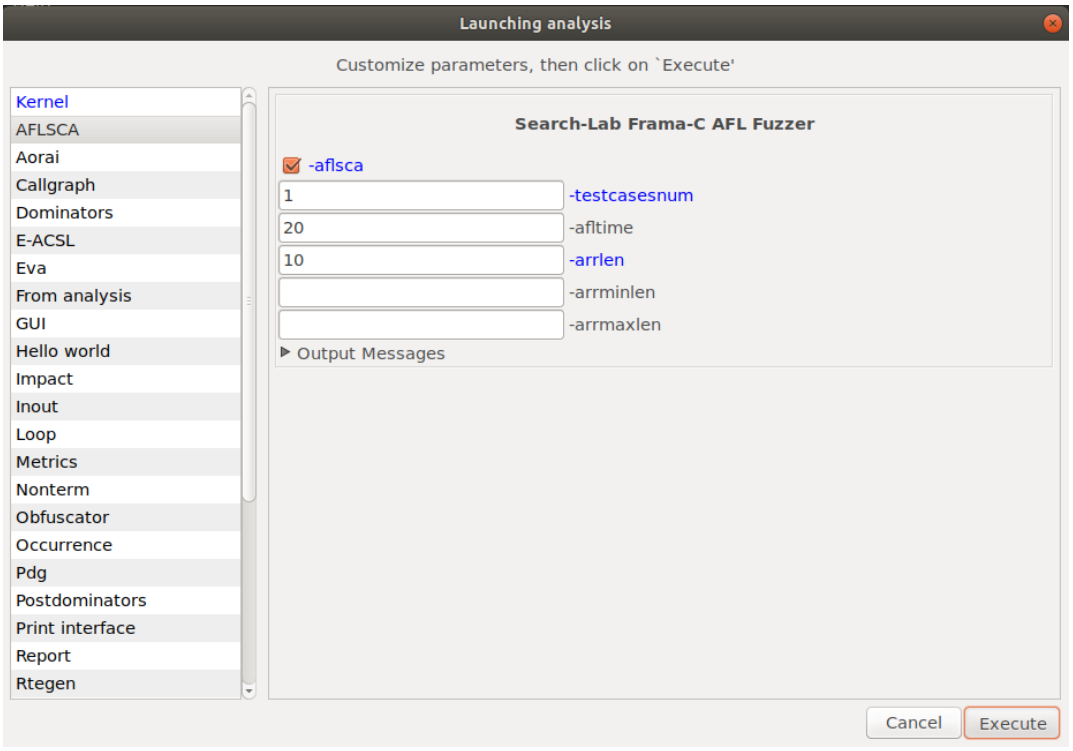

Figure 7. Plugin parameters

The results in the frama-c-gui:

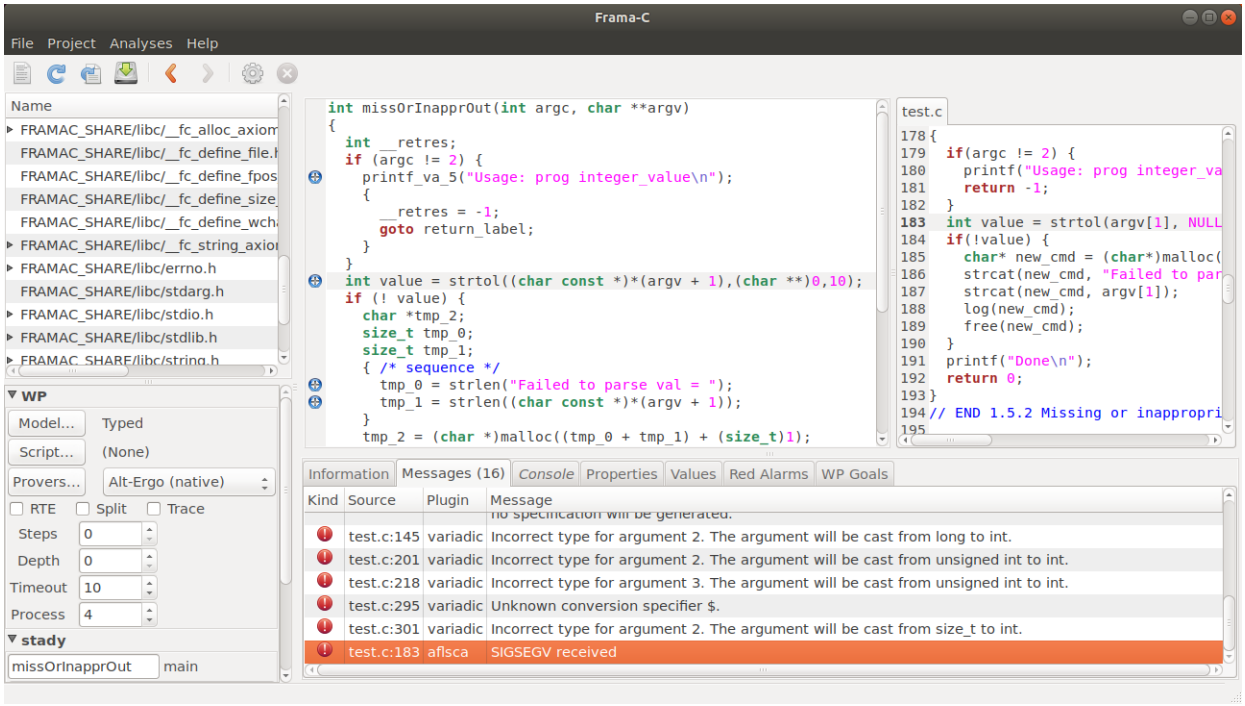

Figure 8. The results in the GUI

Description of the vulnerability:

The above code demonstrates missing or inappropriate output encoding[5]. When this code is executed and the result is processed by some other program the attacker can poison the input of that program due to the encoding problems.

The crash was happened for the following inputs:

- argc = 2
- argv = [null, “SW !A”]

2.6 Conclusion and future work

In the above we showed how AFLSCA can combine the static and dynamic analysis to find vulnerabilities in functions using fuzzing technique with the help of the static analysis provided by the StaDy plugin. The current state of the AFLSCA is capable of demonstrating the composite analysis in case of basic input types. In the future, the plugin can be improved to

- handle more complex input types such as structures and objects, which StaDy can analyse,
- instrument the function to save and restore the program state (using StaDy analysis result) at the function start and
- convert the AFL crash file to a human readable format.

[5] https://cwe.mitre.org/data/definitions/116.html

Chapter 3 Verification Artefact Management (CEA)

3.1 General Description

As part of D2.4, the *MarkDown Report (MDR)* plug-in contains the work done on T2.5 relative to the integration of the results of Frama-C analyzers into more general verification activities. In its current version, it targets analyses done with the Eva abstract interpretation plug-in and WP deductive verification plug-in. It has two possible outputs: either Markdown, a markup language that can easily be edited by hand, or SARIF, the Static Analyzers Results Interchange Format, a json schema proposed as a standard in the OASIS foundation for providing any kind of information about source code (see https://github.com/oasis-tcs/sarif-spec/). The generated markdown document can be used to produce standalone documents or be integrated into larger reports thanks to the use of the Pandoc tool (https://pandoc.org). Furthermore, as described below, the generated markdown can be completed with remarks made by the user, for instance to indicate why a given reported issue is in fact a false alarm.

3.2 Installation instructions

In order to facilitate the usage of VESSEDIA artefacts, the version of MDR included in this deliverable is meant to be compiled against Frama-C 19.0 Potassium. Hence, the first step of the installation is the same as in the previous chapter: installing Frama-C 19.0 Potassium, preferably through `opam`.

After that, installing the plugin is simply a matter of extracting the archive, and compiling it. More precisely, the following sequence of commands should perform the installation.

```
tar xzvf markdown-report.tar.gz
cd markdown-report
make
make install
```

To ensure that the installation succeeded, type

```
frama-c -mdr-h
```

This should output the list of available options for the MarkDown Report plug-in. The most important options are described in the next section. Please refer to the README.md file in the source directory of the plug-in for more information.

3.3 Usage

We will use as an example the file `cwe126.c`, which is included in the `examples/` subdirectory in the sources of the plug-in. This small example comes from the Juliet test suite of NIST (https://samate.nist.gov/SRD/view_testcase.php?tID=76270) to show an example of CWE 126 (buffer overflow) and is available in the public domain. For convenience, the code of the function where the flaw is supposed to be found is included below:

```
void CWE126_Buffer_Overread__malloc_char_loop_64b_badSink(void * dataVoidPtr)
{
    /* cast void pointer to a pointer of the appropriate type */
    char * * dataPtr = (char * *)dataVoidPtr;
    /* dereference dataPtr into data */
    char * data = (*dataPtr);
    {
        size_t i, destLen;
        char dest[100];
        memset(dest, 'C', 100-1);
        dest[100-1] = '\0'; /* null terminate */
        destLen = strlen(dest);
        /* POTENTIAL FLAW: using length of the dest where data
         * could be smaller than dest causing buffer overread */
        for (i = 0; i < destLen; i++)
        {
            dest[i] = data[i];
        }
        dest[100-1] = '\0';
        free(data);
    }
}
```

First, we analyse the file with Eva. As we use some functions from the standard library (namely `memset`), we add the file `string.c` from Frama-C standard library as input file. We also use the `-slevel` option to increase precision, and save the results into file `cwe126.sav`, that we will use from now on. All in all, the command line for the analysis itself is the following:

```
frama-c -val -slevel 100 cwe126.c $(frama-c -print-share-path)/libc/string.c \
        -save cwe126.sav
```

We see in the output of the analysis that we only have one alarm, corresponding to the flaw we are supposed to detect, which means that our parameterization of Eva is appropriate. We can now go to the next step, generating a draft of the markdown report. For that, we will of course load the results of our analysis, and use `-mdr-gen draft` to indicate that we want to generate a draft document. In addition, `-mdr-stubs` allows us to say that `string.c` contains stub functions, i.e. functions that are used by the code under analysis but are not themselves in the perimeter of the analysis. In other words, they emulate some trusted third-party library and it is out of the scope of Frama-C to decide whether this emulation is faithful or not. Our command line for generating the draft report then becomes:

```
frama-c -load cwe126.sav -mdr-gen draft -mdr-stubs \
   $(frama-c -print-share-path)/libc/string.c -mdr-out cwe126.remarks.md
```

The draft markdown is generated in `cwe126.remarks.md`. This file is composed of a certain number of sections, each of which can be annotated by some markdown content to give additional information. Instructions on how to proceed are given directly in the file as markdown comments.

The sections are the following:

1. Introduction: empty by default, can be used to describe the purpose of the analysis
2. Context: describes input files and the parameters used by Eva
   2.1. Input files
   2.2. Configuration
      2.2.1. Eva Domains that have been used
      2.2.2. Stubbed functions
3. Coverage information
4. Warnings emitted by Frama-C
5. Alarms emitted by Eva
6. Conclusion

Once all remarks have been written (or copied from the file `cwe126.remarks-sample.md`), it is possible to generate the final version of the report in markdown or as a SARIF json object. For the former case, the command line is the following:

```
frama-c -load cwe126.sav -mdr-gen md -mdr-out cwe126.md \
   -mdr-remarks cwe126.remarks.md
```

For the latter, we only have to change the kind of output and the name of the file that should be created:

```
frama-c -load cwe126.sav -mdr-gen sarif -mdr-out cwe126.sarif \
   -mdr-remarks cwe126.remarks.md
```

Note however that SARIF support is currently in a very early stage of development, and that the obtained json object contains much less information than the markdown file.

Chapter 4 Summary and Conclusion

This deliverable contains two developments that illustrate how Frama-C can cooperate with other tools in order to achieve better results regarding software verification. On the one hand, the use of AFL shows how to leverage a state-of-the-art fuzzer in order to generate test cases that falsifies a given ACSL property. On the other hand, the MarkDown Report plug-in provides ways to integrate the results of Eva into broader verification tasks.

Chapter 5 List of Abbreviations

Abbreviation	Translation
AFL	American Fuzzy Lop
ACSL	ANSI/ISO C Specification Language
OASIS	Organization for the Advancement of Structured Information Standards
SARIF	Static Analyzers Results Interchange Format

Chapter 6 Bibliography

[1] VESSEDIA DS-01-731453 / D3.3 report: Guidelines for combination of static and dynamic analyses
[2] VESSEDIA DS-01-731453 / D1.5 report: Analyses choice methodology report
[3] VESSEDIA DS-01-731453 / D2.1 report: Basic Analyzers – Intermediate Release
[4] VESSEDIA DS-01-731453 / D2.2 report: Collaboration of analyses intermediate release V1
[5] VESSEDIA DS-01-731453 / D2.3 report: Basic analyzers final release
[6] Kiss B., Kosmatov N., Pariente D., Puccetti A. (2015) Combining Static and Dynamic Analyses for Vulnerability Detection: Illustration on Heartbleed. In: Piterman N. (eds) Hardware and Software: Verification and Testing. Lecture Notes in Computer Science, vol 9434. Springer, Cham.