D2.2

Collaboration of analyses intermediate release V1

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 / D2.2 / 1.0
Work package contributing to the deliverable:	WP2
Due date:	Dec 2018 – M24
Actual submission date:	16th January 2019

Responsible organisation:	SLAB
Editor:	Gergely Eberhardt
Dissemination level:	PU
Revision:	1.0

Abstract:	Companion report describing software delivered as D2.2
Keywords:	Combining static and dynamic analysis, AFLSCA, StaDy

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

Editor

Gergely Eberhardt (SLAB)

Contributors

Virgile Prevosto (CEA)

Ákos Köte (SLAB)

Bence Hambalkó (SLAB)

Disclaimer

Executive Summary

This document gives a brief overview of the initial 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 D2.2, and related to tasks T2.4 and T2.5. For the former, 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 the latter, the MarkDown Report is presented in Chapter 3. It provides two ways for presenting the results of the Eva plug-in of Frama-C into semi-structured formats that can be understood by external tools.

Chapter 2 Combining Static and Dynamic analysis (SLAB)

2.1 General Description

In the STANCE [4] 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 the Frama-C, the SANTE method is replaced by the STUDY 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 resolution, which is an NP-complete problem. Therefore even 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.

To establish the connection between STADY and AFL, we implement the AFLSCA plugin similarly to the PathCrawler plugin. The AFLSCA implements the Testgen interface, which is called during Frama-C execution at specific times.

2.2 Installation instructions

The AFLSCA tool is consisting of the following software modules:

- Modified StaDy plugin
- SCAAFL plugin
- AFL

2.2.1 Installing Frama-C v17.x – Chlorine

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

```
apt install opam
opam init
opam pin add frama-c 20180502 # depending on the opam version, might be 17.1[2]
opam install frama-c
```

[1] http://lcamtuf.coredump.cx/afl/

[2] In opam 1.2, the default on Debian/Ubuntu, Frama-C Chlorine is known 20180502, while on opam 2.x (the newest version of the tool), it is known as 17.1.

There are two Frama-C packages in OPAM:

- `frama-c-base`: minimal Frama-C installation, without GUI; few dependencies
- `frama-c`: includes all GUI-related dependencies, plus other recommended packages.

For installing missing Frama-C dependencies:

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

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

2.2.2 Adding StaDy plugin

Download StaDy plugin from the official GitHub repository[4]. The repo's chlorine branch must be used for Frama-C v17.x (chlorine). The building process is the following (taken from the repo's readme file):

```
autoconf
./configure
make
make install
```

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 Installing AFL

First download and unzip the source files from the AFL website[5]. Compiling the source files should be pretty straightforward with `make`. As the quick start guide states: *If the build fails, you should see `docs/INSTALL` for tips*. The targeted binary must be instrumented before it can be analysed with AFL. If the source code is available, then the program must be compiled with tools found in the root folder of AFL.

For C programs:

```
<path to afl>/afl-gcc main.cpp –o main <other gcc arguments>
```

For C++ programs:

```
<path to afl>/afl-g++ main.cpp –o main <other gcc arguments>
```

With makefile:

```
CC=<path to afl>/afl-g++
…
```

Finally call the instrumented program with:

```
<path to afl>/afl-fuzz -i <path to afl>/testcases -o findings_dir main
<program arguments>
```

The input folder should contain the test cases, the results should be found in the `findings_dir`. There is a sample directory for input named `testcases` in the root folder of AFL.

[3] https://frama-c.com/install-chlorine-20180501.html
[4] https://github.com/gpetiot/Frama-C-StaDy/tree/chlorine
[5] http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz

With the first usage of the AFL a typical program abort error could emerge:

```
[-] PROGRAM ABORT : Pipe at the beginning of 'core_pattern'
```

It means that the AFL could not get core dump notifications directly. It can be resolved with editing the following file:

```
sudo nano /proc/sys/kernel/core_pattern
```

The contents should be temporarily modified to:

```
echo core >/proc/sys/kernel/core_pattern
```

2.3 Usage / User stories

0. Follow the installation instructions.
1. Annotate the source code with ACSL to get a better result during static analysis.
2. It is recommended to use the GUI of Frama-C. You can run this via: `sudo frama-c-gui`
3. Load the source files in the Frama-C GUI with File->Source Files. Here you need to select the `*.c` files you want to analyze. After the files have been loaded you can start running the different plugins to analyze the source code. There is a way to do this in the console if automation needs to be in place using: `frama-c <file> -main <entry point function> -stady -stady-socket stdio`. However, a user will find the GUI much easier to use since the error messages are more friendly on there.
4. To run an analysis select `Analyse->Analyses` and in the upcoming dialog select the analysis you would like to run.

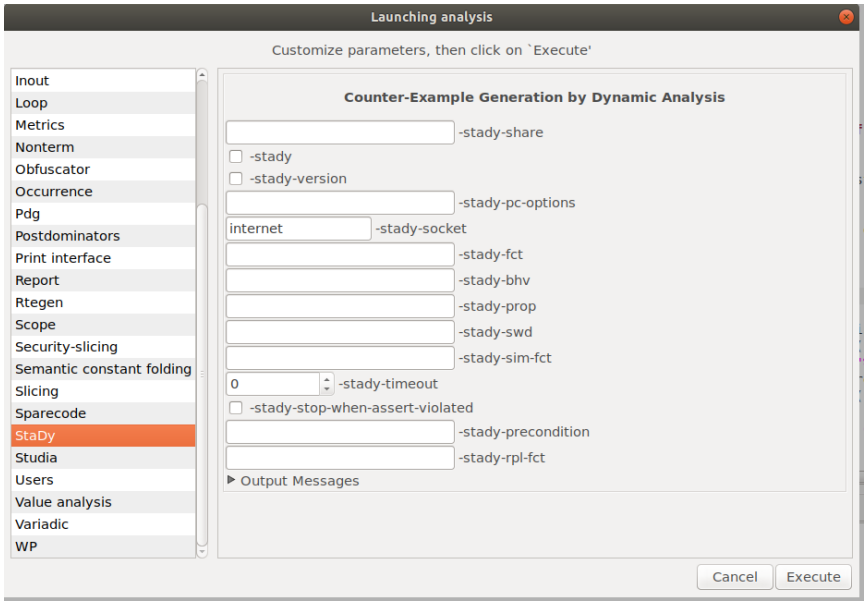

Figure 1: Analysis dialog

5. After pressing execute the analysis will begin and the results can be inspected.

2.4 Usage example

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.

The original example source code performs very simple authentication by comparing the received input with the secret password. The example code contains a very simple buffer overflow in the

`function`. If the attacker starts it with an argument larger than 8 characters, the `strcpy` function will overwrite the buffer, which is allocated as 8 bytes long.

```
#include "stdio.h"
#include "string.h"
#include "stdlib.h"

void function(char* input)
{
    unsigned int i=1;
    unsigned int j=2;
    char buffer[8];
    strcpy(buffer, input);
}

int main(int argc, char* argv[])
{
    char *line = NULL;
    size_t len = 0;
    size_t read = 0;
    read = getline(&line, &len, stdin);
    line = strtok(line, "\n");
    function(line);

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

Figure 2: Example source code

To help the static analyser we have to annotate the code with ACSL[6]. Annotation is required by Frama-C so it can understand what the code does.

```
#include "stdio.h"
#include "string.h"
#include "stdlib.h"

/*@ requires \valid(input);
*/
void function(char* input)
{
    unsigned int i=1;
    unsigned int j=2;
    char buffer[8];
    strcpy(buffer, input);
}

/* @ requires \valid(argv);
   @ ensures \result == 0;
*/
int main(int argc, char* argv[])
{
    char *line = NULL;
    size_t len = 0;
    size_t read = 0;
    read = getline(&line, &len, stdin);
    line = strtok(line, "\n");
    function(line);

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

Figure 3: ACSL annotated source code

[6] https://frama-c.com/download/acsl_1.13.pdf

After we have annotated the code we can load it into Frama-C.

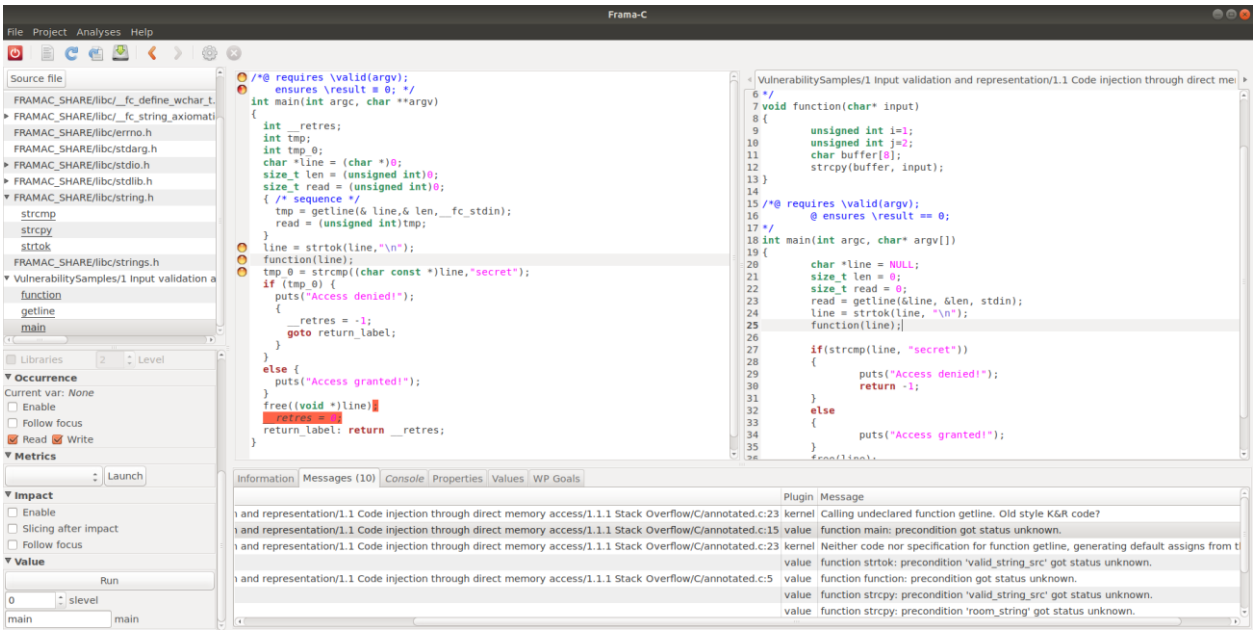

Figure 4: Example code loaded into Frama-C GUI

After running the plugins you can see the results under the messages tab:

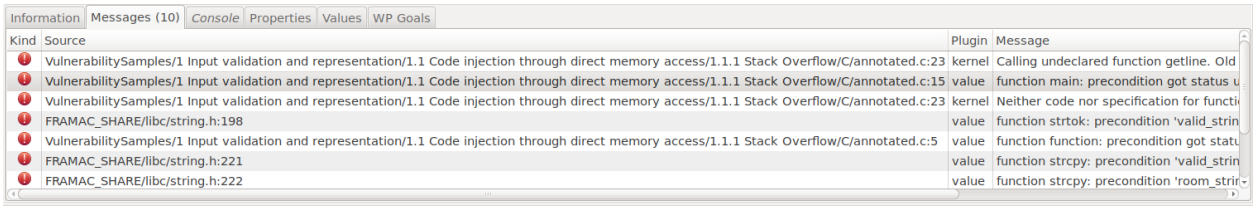

Figure 5: Result after executing the Value plugin for the example code

2.4.1 Using AFL

The dynamic analysis part uses AFL, so to test the code showed in the previous example, we have to compile it (instrument) with the following command:

```
afl-2.52b/afl-g++ main.cpp -o main
```

For AFL testing run the `afl-fuzz` tool with the following command. As an initial test case we provided the "`secret`" string in the test case directory.

```
./afl-fuzz -i testcases2 -o findings_dir2 ./main
```

```
frama-c@frama-c:~/Downloads/afl-2.52b$ ./afl-fuzz -i testcases2 -o findings_dir2 ./main
afl-fuzz 2.52b by <lcamtuf@google.com>
[+] You have 1 CPU core and 6 runnable tasks (utilization: 600%).
[*] Checking core_pattern...
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'testcases2'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:test.txt'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 7, map size = 4, exec speed = 257 us
[+] All test cases processed.

[+] Here are some useful stats:

    Test case count : 1 favored, 0 variable, 1 total
       Bitmap range : 4 to 4 bits (average: 4.00 bits)
        Exec timing : 257 to 257 us (average: 257 us)

[*] No -t option specified, so I'll use exec timeout of 20 ms.
[+] All set and ready to roll!
```

Figure 6: Initialization of AFL

Since the example code contains a trivial vulnerability, after the initialization AFL finds the buffer overflow very quickly.

```
                american fuzzy lop 2.52b (main)

┌─ process timing ─────────────────────┬─ overall results ─┐
│        run time : 0 days, 0 hrs, 0 min, 1 sec │  cycles done : 1  │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec │  total paths : 2  │
│ last uniq crash : 0 days, 0 hrs, 0 min, 0 sec │ uniq crashes : 1  │
│  last uniq hang : none seen yet               │   uniq hangs : 0  │
├─ cycle progress ─────┬─ map coverage ─────────┴───────────────────┤
│  now processing : 0 (0.00%)  │    map density : 0.01% / 0.01%     │
│ paths timed out : 0 (0.00%)  │ count coverage : 1.00 bits/tuple   │
├─ stage progress ─────┼─ findings in depth ────────────────────────┤
│  now trying : havoc  │ favored paths : 2 (100.00%)                │
│ stage execs : 504/768 (65.62%)  │  new edges on : 2 (100.00%)     │
│ total execs : 3793   │ total crashes : 770 (1 unique)             │
│  exec speed : 1913/sec  │  total tmouts : 0 (0 unique)            │
├─ fuzzing strategy yields ──────────┬─ path geometry ──────────────┤
│   bit flips : 1/88, 0/86, 0/82     │    levels : 2                │
│  byte flips : 0/11, 0/9, 0/5       │   pending : 0                │
│ arithmetics : 0/616, 0/48, 0/0     │  pend fav : 0                │
│  known ints : 1/61, 0/252, 0/220   │ own finds : 1                │
│  dictionary : 0/0, 0/0, 0/0        │  imported : n/a              │
│       havoc : 0/1792, 0/0          │ stability : 100.00%          │
│        trim : 21.43%/2, 0.00%      ├──────────────────────────────┘
└────────────────────────────────────┘                [cpu:427%]
```

Figure 7: AFL runtime results

As it seen in the figure above, AFL reported 1 unique crash, which was caused by a 9 bytes long input string. Although AFL could not give information about the root cause of the crash, there are several ways (crash dump, executing the program in a controlled environment such as Valgrind with the input found by the AFL) how such information can be extracted in an automatic way.

During the next phase of T2.4, we will continue to implement the AFLSCA plugin, which will be able to use AFL fuzzer based on the StuDy plugin. The final version of the AFLSCA plugin will be presented in D2.4.

Chapter 3 Verification Artefact Management (CEA)

3.1 General Description

As part of D2.2, 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 only targets analyses done with the Eva abstract interpretation 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 markdown version is currently more advanced, but work on the SARIF output will continue and be made available in D2.4. 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 artifacts, the version of MDR included in this deliverable is meant to be compiled against Frama-C 17.1 Chlorine (also known as 20180502)[7]. Hence, the first step of the installation is the same as in the previous chapter: installing Frama-C 17.1 Chlorine, 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:

[7] A version compatible with Frama-C 18.0 Argon is also available upon request.

```
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 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 to 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 in 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.

While the first results are promising, the tools provided in this deliverable can be extended in several ways. In particular, MarkDown Report could take into account other plugins, notably WP, and provide a much better support of SARIF. The final release of the tools, in D2.4, will present the improvements done in the upcoming year.

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] 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.