



D4.5

Quality tests & limits of VESSEDIA tools regarding security vulnerabilities detection

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

Deliverable type:	Report
Deliverable reference number:	DS-01-731453 / D4.5/ 1.0
Work package contributing to the deliverable:	WP 4
Due date:	December 2019 – M36
Actual submission date:	7 th January 2020

Responsible organisation:	AMO
Editor:	Florent SAUDEL
Dissemination level:	PUBLIC
Revision:	1.0

Abstract:	This report will present the application of the Frama-C static analyser to the VESSEDIA benchmarks presented in the D4.3 report. It also describes the problem faced, the approach taken to resolve them before comparing the Frama-C's results against others known and mature static analysers.
Keywords:	Security evaluation, C Source code, Code auditing Frama-C



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

Editor

Florent SAUDEL (AMO)

Contributors (ordered according to beneficiary numbers)

Virgile PREVOSTO (CEA)

Balázs Berkes (SLAB)

Cédric BERTHION (AMO)

Disclaimer

The information in this document is provided “as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The content of this document reflects only the author’s view – the European Commission is not responsible for any use that may be made of the information it contains. The users use the information at their sole risk and liability.

Executive Summary

This report presents the application of the Frama-C static analyser to the VESSEDIA benchmarks presented in the D4.3 report. It also describes the problem faced, the approach taken to resolve them before comparing the Frama-C's results against the Clang static analyser tool.

This document starts with the remainder of the test cases chosen and explained in the [D4.3] report.

It describes the practical issues faced when applying Frama-C on the benchmarks, due to the peculiarity of the DARPA CGC benchmark.

The next section describes the solution that enabled Frama-C to analyse successfully all the samples. The solution is based upon a compatibility layer and some minor modifications of the source code of the sample.

Then, the document gives the results for the Frama-C and Clang static analyser tools on the benchmarks before commenting them. Later, the statistics about the coverage of the analysis and the number of alarms raised by Frama-C are given.

Overall, Frama-C is able to warn the analyst of the presence of potential runtime errors. On the contrary, Clang static analyser is able to give a valid warning for only one sample. Yet Frama-C produces a lot more alarms. Furthermore, the information given by the tool is not always sufficient to easily identify if the alarm is spurious nor the genuine nature of the bug.

Finally, the summary of the experiments is presented with all the issues faced. The conclusion proposes some improvements to Frama-C which could enhance the user experience or facilitate the manual process of the alarms triaging.

Contents

Chapter 1	Introduction	1
1.1	Related Deliverables	1
Chapter 2	Methodology	2
2.1	Reminder about the benchmarks and the methodology	2
2.2	Process of evaluation	2
2.3	Particularity of the DARPA CGC test cases which hinder the Frama-C's analysis....	2
2.4	Corpus modification	3
2.5	The minimal compatibility layer from DECREE to Linux.....	4
2.6	Experimental results.....	5
Chapter 3	Presentation and analysis of the benchmarks' results	6
3.1	Vulnerability 1 – Stack Buffer Overflow	6
3.1.1	Basic sample	6
3.1.2	Modifications made to the sample	7
3.1.3	Frama-C's results	7
3.1.4	Clang static analyser's results	9
3.2	Vulnerability 2 – Heap buffer overflow.....	10
3.2.1	Basic sample	10
3.2.2	Modifications made to the sample	10
3.2.3	Frama-C's results	11
3.2.4	Clang static analyser's results	12
3.3	Vulnerability 3 – Null pointer dereference	13
3.3.1	Basic sample	13
3.3.2	Modifications made to the sample	13
3.3.3	Frama-C's results	14
3.3.4	Clang static analyser's results	14
3.4	Vulnerability 4 – Use after free	15
3.4.1	Basic sample	15
3.4.2	Modifications made to the sample	16
3.4.3	Frama-C's result	16
3.4.4	Clang static analyser's results	17
3.5	Vulnerability 5 – Uninitialised variable	18
3.5.1	Basic sample	18
3.5.2	Modifications made to the sample	19

3.5.3	Frama-C's results	19
3.5.4	Clang static analyser's results	20
3.6	Vulnerability 6 – Off by one	22
3.6.1	Basic sample	22
3.6.2	Modifications made to the sample	22
3.6.3	Frama-C's results	23
3.6.4	Clang static analyser's results	24
3.7	Vulnerability 7 – Double free	25
3.7.1	Basic sample	25
3.7.2	Modifications made to the sample	25
3.7.3	Frama-C's results	25
3.7.4	Clang static analyser's results	27
3.8	Vulnerability 8 – Format string	28
3.8.1	Basic sample	28
3.8.2	Modifications made to the sample	29
3.8.3	Frama-C's results	29
3.8.4	Clang static analyser's results	29
Chapter 4	Experiment results summary	31
4.1	Experiments on the patched version of the source code	33
Summary and Conclusion	37
Chapter 5	List of Abbreviations	38

List of Figures

Figure 1 - Frama-C's result for the basic stack overflow sample	6
Figure 2 - <code>main</code> function calling the vulnerable <code>fill</code> function.....	7
Figure 3 - Missing alarms on the Frama-C's analysis output for the stack buffer overflow sample in the <code>src/service.c</code> source file	8
Figure 4 – Warning about the unsigned downcast when the switch <code>-warn-unsigned-downcast</code> is enabled.....	8
Figure 5 - Vulnerable call to <code>strcat</code> function in the stack buffer overflow sample	9
Figure 6 – Frama-C's result for the basic heap overflow sample	10
Figure 7 - Excerpt of the Frama-C's analysis of the Heap Buffer overflow sample	11
Figure 8 - Frama-C's result for the basic null pointer dereference sample.....	13
Figure 9 - Red alarm for the null pointer dereference vulnerability	14
Figure 10 - Code path leading to the Null pointer dereference vulnerability.....	15
Figure 11 - Frama-C's result for the basic use-after-free sample.....	15
Figure 12 - Alarms related to dangling pointers in the use-after-free sample.....	17
Figure 13 – Frama-C's output for the basic uninitialised variable sample	18
Figure 14 - Red alarm spotting the uninitialised variable vulnerability.....	19
Figure 15 - <code>h_state</code> abstract value computed by the EVA plugin	20
Figure 16 - Code path to trigger the use of uninitialized <code>h_state</code> variable	21
Figure 17 – Frama-C's result for the basic off-by-one sample	22
Figure 18 - Alarms raised for the call to the <code>strcpy</code> vulnerable to an off-by-one buffer overflow ..	23
Figure 19 - EVA plugin outputs for the <code>ENV</code> and <code>i</code> variables in the off-by-one sample	24
Figure 20 – Frama-C's output for the basic double free sample	25
Figure 21 - Frama-C's result for the double free sample	26
Figure 22 - Undetected stack buffer overflow in the heap buffer overflow sample	26
Figure 23 - Guard making the stack buffer overflow unreachable in the <code>nyan</code> function	26
Figure 24 – Frama-C's output for the basic format string sample	28

List of Tables

Table 1: Related Deliverables	1
Table 2 - Coverage obtained by Frama-C's analysis	31
Table 3 - Metrics about the alarms raised by Frama-C.....	32
Table 4 - Metrics about the alarms raised by the Clang static analyser	33
Table 5 - Coverage obtained by Frama-C's analysis on the patched source code	34
Table 6 - Coverage obtained by Frama-C on the patched version of the source code	34
Table 7 - Metrics about the alarms raised by Frama-C on the patched version of the samples	35
Table 8 - Results of the Clang static analysers on the patched version of each samples	36

Chapter 1 Introduction

1.1 Related Deliverables

Deliverable Number	Deliverable Title	Type	Dissemination level
[D1.7]	Vulnerability discovery methodology	Report	Public
[D4.2]	VESSEDIA approach for security evaluation	Report	Public
[D4.3]	Benchmarks for evaluating VESSEDIA tools	Report	Public
[D43.zip]	Source code of samples selected for the benchmarks	Archive	Public
[D45.zip]	Modified source code of samples selected for the benchmarks	Archive	Public

Table 1: Related Deliverables

Chapter 2 Methodology

2.1 Reminder about the benchmarks and the methodology

This section sums up briefly the benchmarks and the methodology described in the [D4.3] document. The benchmarks are composed of eight categories. Each category contains two samples: one simple and another complex sample chosen from the DARPA CGC samples corpus. The two samples illustrate one kind of vulnerability:

The [D4.3] document present all the details about the vulnerabilities we are looking for in the benchmarks. Therefore, only the vulnerabilities in the basic samples are kept in this section.

- Vulnerability 1 – Stack Buffer Overflow
- Vulnerability 2 – Heap buffer overflow
- Vulnerability 3 – Null pointer dereference
- Vulnerability 4 – Use after free
- Vulnerability 5 – Uninitialised variable
- Vulnerability 6 – Off by one
- Vulnerability 7 – Double free
- Vulnerability 8 – Format string

2.2 Process of evaluation

The [D4.3] document has chosen two well-known static analysers to compare against Frama-C.

Between this document and [D4.3], the list of chosen static analysers received a major update. Thus, we decided to use the most up to date version of each selected static analysers.

The selected tools for this analysis are:

- Frama-C version 19.0 (Potassium)¹
- Clang Static Analyzer version 8.0.0²

Initially, it was planned to compare with the results provided by CodeSonar version 5 but due to a licensing issue, this tool was removed from the selection.

2.3 Particularity of the DARPA CGC test cases which hinder the Frama-C's analysis

At the beginning of the evaluation, Frama-C was only able to process and analyse the simple samples. For the complex ones, Frama-C could not process the source code without stumbling on an error.

Indeed, the DARPA CGC samples were developed for a non-standard operating system called DECREE. No standard library (libc) was given to the sample developers, so each sample contains its own implementation of a subset of the libc. The DARPA CGC corpus respects convention when

¹ Frama-C can be downloaded at <https://frama-c.com>

² Clang static analyser is available for download at <https://clang-analyzer.lvm.org>

it comes to the architecture of a sample repository. This minimal subset of the libc is always stored in the `lib/` sub-repository. The main part of the source code is in the `src/` sub-repository.

To analyse a code base, Frama-C makes the hypothesis that the code uses a POSIX standard library. Thus, Frama-C has its own annotated libc implementation. Also, thanks to its custom libc, Frama-C is able to analyse a code base without any annotations in it. Each annotated function in the libc plays the role of a starting point for the analysis. Furthermore, the “stubs” function and their annotations accelerate the analysis of the source code of the program because Frama-C does not need to reanalyse the source code of the libc, which is costly and redundant.

Fortunately, the re-implemented subset of the libc functions in each sample has the same name and signature than the ones in the POSIX libc. Hence, it seemed possible to abstract this subset and use the default standard library shipped with Frama-C. Moreover, the vulnerabilities which Frama-C should detect are never in the libc subset. All the buggy code of the DARPA CGC samples is always in the `src/` sub-repository.

With these two elements in mind, the best approach to handle these samples with Frama-C seemed to exclude all the code of subset libc which is mostly POSIX compliant and let Frama-C uses its own libc instead.

For the few discrepancies due to the DECREE operating system, a minimal compatibility layer was developed for each sample. Trail of Bits³ has chosen this same method to port the DARPA CGC corpus from the DECREE platform to the Windows, macOS, and Linux operating systems. This solution is directly inspired by their work. It tries to be the least intrusive possible and avoids to modify the source code of the sample.

2.4 Corpus modification

In the [D4.3] document, the selection of the samples was made over the Trail of Bits’ repository⁴. Their compatibility layer is useful to test the samples on widespread computers, however, it was a work overload to remove this layer and replace it with our own. Thus for this analysis, we started over from the original DARPA CGC repository⁵.

Each complex sample repository contains three sub-repositories:

- `src/` which contains the main source code of the sample and where the vulnerability is present;
- `lib/`: it contains the dependencies of the sample and its own implementation of its own subset of the libc;
- `pov/`: it contains the xml files that represent an input which trigger the vulnerability.

Our approach was to try to analyse with Frama-C only the portion of the code under `src/`. We proceeded incrementally by successively running an analysis using Frama-C and then fixing the reported errors, until we obtained a complete compatibility layer for Frama-C. The compatibility layer implementation is made of “stub” functions, functions that mimic the behaviour of the DARPA syscalls. The compatibility layer’s source code is in a `libcgc.c` file in the top repository of each sample.

The resulting source code should still compile correctly with a standard compiler like GCC or Clang and the program output should run as expected. This constraint was necessary to be able to apply the Clang Static Analyser to the modified samples.

³ <https://www.trailofbits.com/>

⁴ <https://github.com/trailofbits/cb-multios>

⁵ <https://github.com/lungetech/cgc-corpus>

The next sections describe the modifications needed for each complex sample and the results obtained by the Frama-C analysis.

2.5 The minimal compatibility layer from DECREE to Linux

The DECREE platform build for the CGC competition is an operating system which exposes only five “syscalls”: `receive`, `transmit`, `allocate`, `deallocate`, and `terminate`.

Most of the samples we faced during this analysis do not depend directly on these “syscalls” but use higher-level functions like `printf` to output result and `malloc/free` to allocate and deallocate memory.

However, the `receive` and `terminate` “syscalls” seems to be preferred against their standard counterparts `read` and `exit`.

To be able to analyse and compile the samples correctly, these five non-standard “syscalls” must be defined. Our approach was to give an equivalent implementation based upon the standard POSIX function known by Frama-C and the compliant compiler.

For instance, the `receive` “syscall” is implemented over the standard `read` function.

```
1 int receive(int fd, void *buf, size_t count, size_t *rx_bytes) {
2     const ssize_t ret = read(fd, buf, count);
3     if (ret < 0) {
4         return errno;
5     } else if (rx_bytes != NULL) {
6         *rx_bytes = ret;
7     }
8     return 0;
9 }
```

2.6 Experimental results

Our experiment repository is available in the [D45.zip] archive. This archive contains all the modified complex samples, two build scripts: one for Frama-C and one for GCC and the output for each static analyser. The simple samples analysed are also given in the archive as examples of expected results for each category of vulnerabilities.

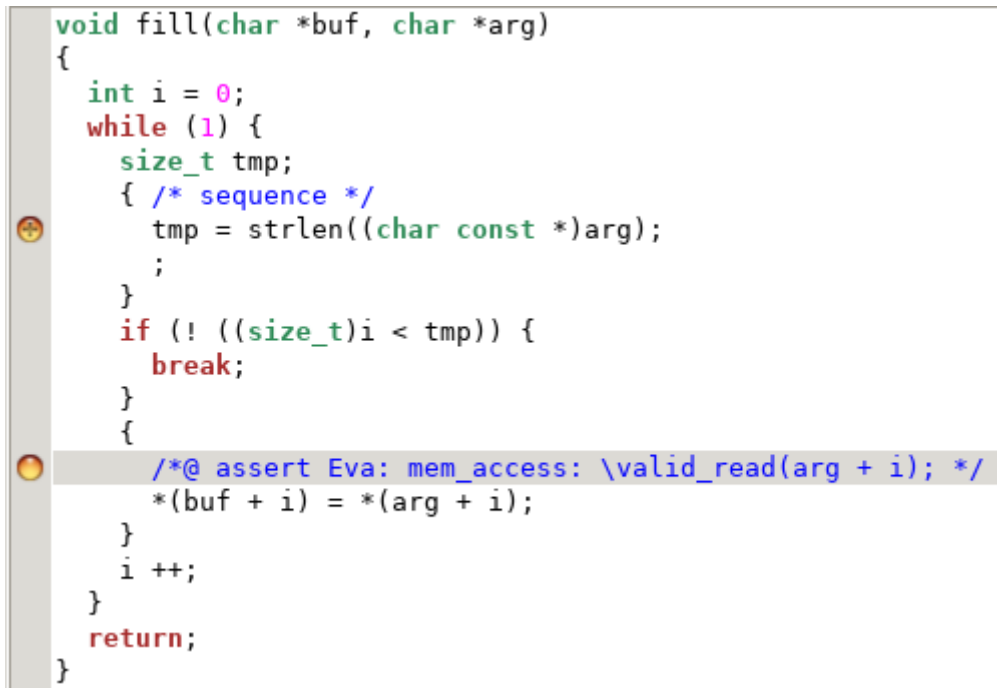
The next section presents each category of vulnerability. Each description starts by an excerpt of the Frama-C analysis results for basic samples. This simple test case shows the kind of alarms which an analyst should look for when reviewing the analysis output. The following subsection explains the changes applied to the complex sample's source code to let Frama-C analyse it successfully. Afterwards, the results obtained by each static analyser are given and then discussed.

Chapter 3 Presentation and analysis of the benchmarks' results

The [D4.3] document present all the details about the vulnerabilities we are looking for in the benchmarks. Therefore, only the vulnerabilities in the basic samples are kept in this section.

3.1 Vulnerability 1 – Stack Buffer Overflow

3.1.1 Basic sample



```
void fill(char *buf, char *arg)
{
    int i = 0;
    while (1) {
        size_t tmp;
        { /* sequence */
            tmp = strlen((char const *)arg);
            ;
        }
        if (! ((size_t)i < tmp)) {
            break;
        }
        {
            /*@ assert Eva: mem_access: \valid_read(arg + i); */
            *(buf + i) = *(arg + i);
        }
        i ++;
    }
    return;
}
```

Figure 1 - Frama-C's result for the basic stack overflow sample

```

int main(int argc, char **argv)
{
    int __retres;
    char name[16] = {(char)0};
    if (argc < 2) {
        {
            __retres = 1;
            goto return_label;
        }
    }
    /*@ assert Eva: mem_access: \valid_read(argv + 1); */
    fill(name, *(argv + 1));
    __retres = 0;
    return_label: return __retres;
}

```

Figure 2 - main function calling the vulnerable fill function

In this basic sample, the user's input is stored in the `arg` variable. The length of this input is computed by the `strlen` function and then this value is used as the size for the data, which will be copied into the destination buffer, the `buf` variable. Yet, the call to the `fill` function displayed in the Figure 2, `fill(name, *(argv + 1))` gives the `name` buffer as the first argument. The second argument is the first argument in the command line. Therefore, the `buf` variable in the `fill` function points to buffer of a static size of 16 bytes and the second argument `arg` is given by the user and can have any size.

Frama-C produces a warning at line 5 of `main.c` file related to the buffer overflow:

```
[eva:alarm] main.c:5: Warning: out of bounds read. assert \valid_read(arg + i);
```

3.1.2 Modifications made to the sample

The main part of the stack buffer overflow sample's source code is only one C file named `service.c`.

Its dependencies are few operations for manipulating and printing strings like `strcat` and `puts`.

Their signatures and semantics are equivalent to the POSIX compliant functions. So, the Frama-C custom libc has already suitable "stubs" for these functions and no extra work is required.

Only two non-standard functions are used: `itoa` and `receive_until`. This issue is fixed by copy-pasting their code in the `service.c` source file.

With these changes, Frama-C is able to parse and analyse this sample.

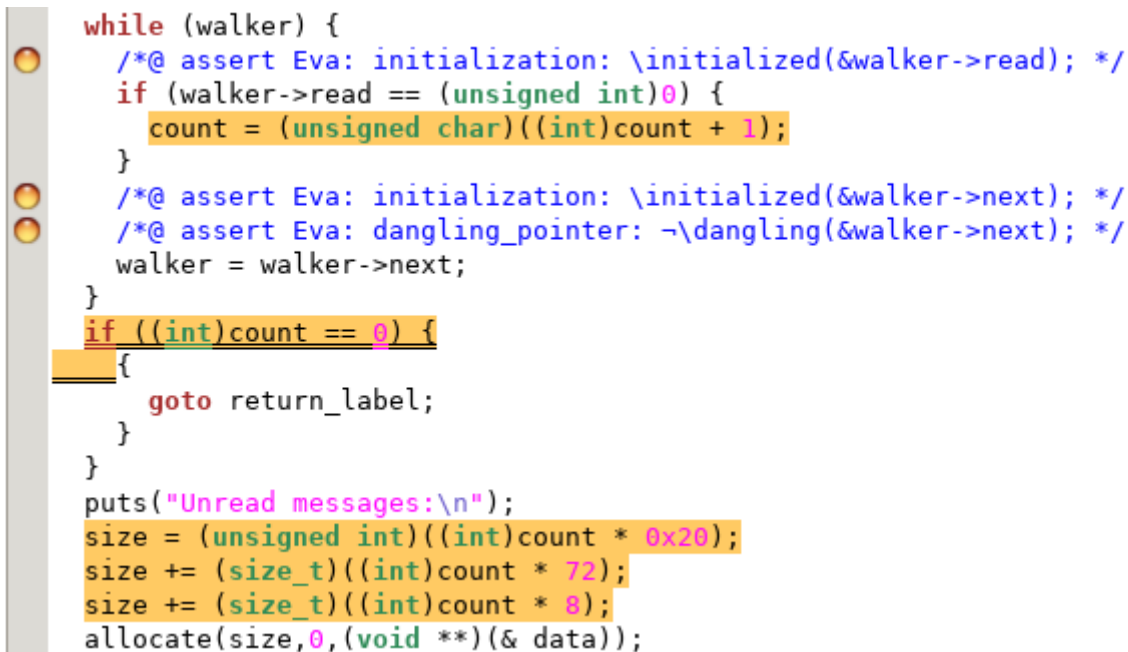
3.1.3 Frama-C's results

Frama-C is not able to produce a red alarm related to the known stack buffer overflow vulnerability, which occurs in the `list_unread_messages` function in the file `src/service.c`.

The `count` variable is an `unsigned char`. This variable is incremented in the loop displayed in the Figure 3. The loop can iterate more than 255 times overflowing the `count` variable. Frama-C does not warn about this behaviour. Thus an input of 256 characters will overflow the `count` variable and produce a call to the `allocate` function with a `size` of 0.

By default, Frama-C does not warn against unsigned integer overflow because it is a defined behaviour with regard to the ISO C specification. Frama-C possesses the option `-warn-unsigned-overflow` to force an alert about this specific case because it can induce a bug sometimes as in this

sample. The analyst analysed again this sample with the `-warn-unsigned-overflow` option. Frama-C raises a new red alarm related to an unsigned overflow on the `itoa` function, which is not related to the known vulnerability. Even with the `-warn-unsigned-overflow` option, Frama-C does not produce an alert on the `count = (unsigned char)((int) count + 1);` statement.



```

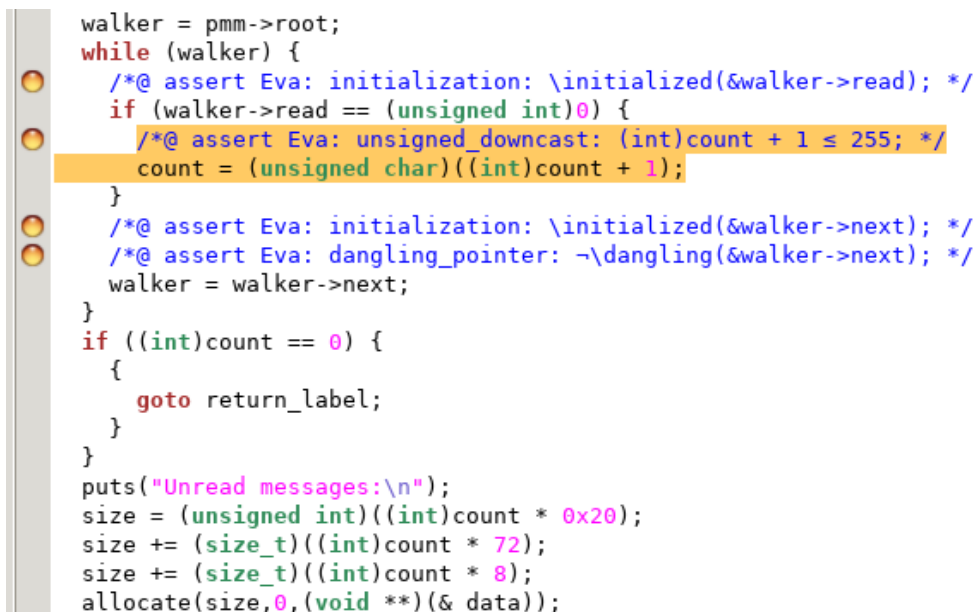
while (walker) {
    /*@ assert Eva: initialization: \initialized(&walker->read); */
    if (walker->read == (unsigned int)0) {
        count = (unsigned char)((int)count + 1);
    }
    /*@ assert Eva: initialization: \initialized(&walker->next); */
    /*@ assert Eva: dangling_pointer: -\dangling(&walker->next); */
    walker = walker->next;
}
if ((int)count == 0) {
    goto return_label;
}
puts("Unread messages:\n");
size = (unsigned int)((int)count * 0x20);
size += (size_t)((int)count * 72);
size += (size_t)((int)count * 8);
allocate(size,0,(void **>(& data));

```

Figure 3 - Missing alarms on the Frama-C's analysis output for the stack buffer overflow sample in the `src/service.c` source file

A thorough analysis of this sample reveals that the vulnerability defined as an integer overflow in this description could be better defined as an unsigned downcast. The expression `count + 1` which is an `int` of 4 bytes is cast to the smaller type `unsigned char` which has a size of only 1 byte.

Therefore, the correct option to give to Frama-C is `-warn-unsigned-downcast`. This option lets Frama-C output a warning about the downcast that provokes the stack buffer overflow. The shows the alert in the Frama-C GUI when the switch `-warn-unsigned-downcast` is enabled.



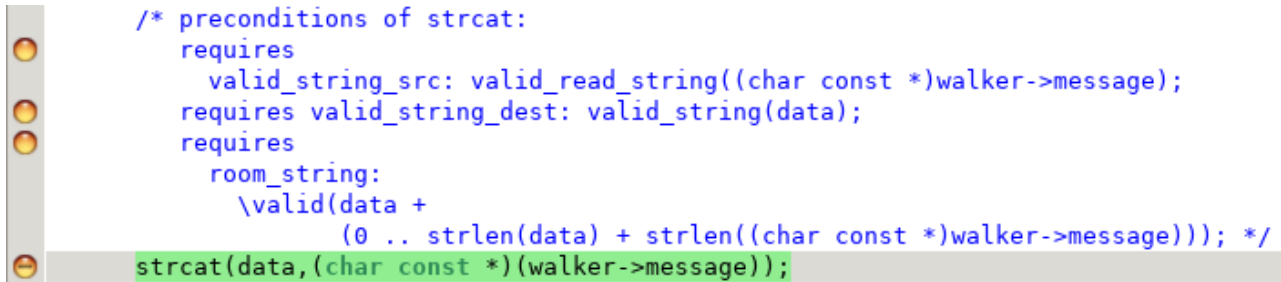
```

walker = pmm->root;
while (walker) {
    /*@ assert Eva: initialization: \initialized(&walker->read); */
    if (walker->read == (unsigned int)0) {
        /*@ assert Eva: unsigned_downcast: (int)count + 1 ≤ 255; */
        count = (unsigned char)((int)count + 1);
    }
    /*@ assert Eva: initialization: \initialized(&walker->next); */
    /*@ assert Eva: dangling_pointer: -\dangling(&walker->next); */
    walker = walker->next;
}
if ((int)count == 0) {
    goto return_label;
}
puts("Unread messages:\n");
size = (unsigned int)((int)count * 0x20);
size += (size_t)((int)count * 72);
size += (size_t)((int)count * 8);
allocate(size,0,(void **>(& data));

```

Figure 4 – Warning about the unsigned downcast when the switch `-warn-unsigned-downcast` is enabled

We expect a warning or even better a red alarm on the third call of `strcat` in the `list_unread_messages` function. Frama-C shows a warning because it cannot prove that the requirements for a safe use of `strcat` hold.



```

/* preconditions of strcat:
requires
  valid_string_src: valid_read_string((char const *)walker->message);
requires valid_string_dest: valid_string(data);
requires
  room_string:
    \valid(data +
      (0 .. strlen(data) + strlen((char const *)walker->message))); */
strcat(data, (char const *) (walker->message));

```

Figure 5 - Vulnerable call to `strcat` function in the stack buffer overflow sample

No Red Alarm is raised by Frama-C. Thus this alert seems as important as all the 102 other warnings in the eye of an analyst.

3.1.4 Clang static analyser's results

Clang static analyser does not find any vulnerability in the code.

```

> scan-build gcc "-I." libcgc.c src/service.c
[...]
src/service.c: In function 'read_message':
src/service.c:427:2: warning: 'strncat' specified bound 256 equals
destination size [-Wstringop-overflow=]
    strncat( buffer, "*****\n", 0x100 );
    ^~~~~~

src/service.c:435:4: warning: 'strncat' specified bound 256 equals
destination size [-Wstringop-overflow=]
    strncat(buffer, ": ", 0x100 );
    ^~~~~~

src/service.c:437:4: warning: 'strncat' specified bound 256 equals
destination size [-Wstringop-overflow=]
    strncat( buffer, "\n*****\n", 0x100 );
    ^~~~~~

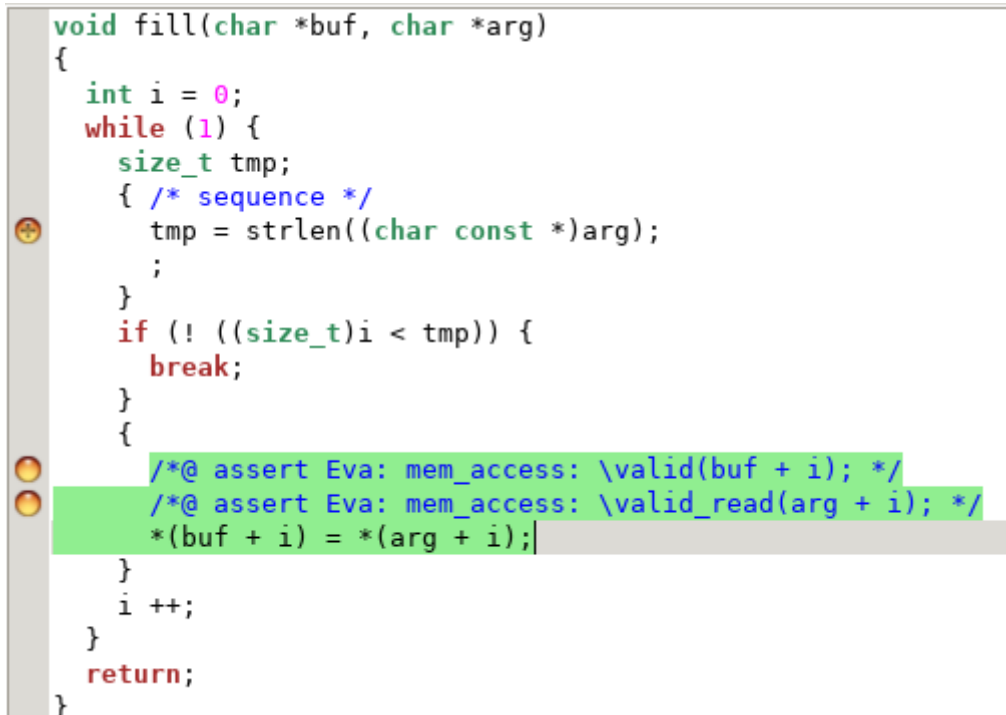
scan-build: Removing directory '/tmp/scan-build-2019-08-30-152326-30608-1'
because it contains no reports.
scan-build: No bugs found.

```

The three warnings raised by the GCC compiler are not related to the expected vulnerability. Even though, the `strncat` function is wrongly used and could be a hint for an unknown vulnerability.

3.2 Vulnerability 2 – Heap buffer overflow

3.2.1 Basic sample



```

void fill(char *buf, char *arg)
{
    int i = 0;
    while (1) {
        size_t tmp;
        { /* sequence */
            tmp = strlen((char const *)arg);
            ;
        }
        if (! ((size_t)i < tmp)) {
            break;
        }
        {
            /*@ assert Eva: mem_access: \valid(buf + i); */
            /*@ assert Eva: mem_access: \valid_read(arg + i); */
            *(buf + i) = *(arg + i);
        }
        i ++;
    }
    return;
}

```

Figure 6 – Frama-C’s result for the basic heap overflow sample

The only difference between this basic sample and the one described in the §3.1.1 is where the memory pointed by the `buf` variable is allocated. Here, the `buf` points to a buffer of 16 bytes in the heap of the process. In the previous sample, the memory was located in the stack of the process. Otherwise, the vulnerability is exactly the same.

The analysis of this basic sample by Frama-C raises two warnings related to the heap overflow at the line 6 of the `main.c` file.

```

[eva:alarm] main.c:6: Warning: out of bounds write. assert \valid(buf + i);
[eva:alarm] main.c:6: Warning: out of bounds read. assert \valid_read(arg + i);

```

3.2.2 Modifications made to the sample

The source code of this sample is particular, the `main.c` file is an already pre-processed form. It seems the file was compiled with `gcc -E` and then pushed to git repository.

To work on a sane base, we cleaned this file, removing all the useless lines and comments inserted by the compiler.

The sample dependencies are memory-related operations (`malloc/free`), strings manipulations and comparisons and conversions (`strchr`, `strcpy`, `strtol`). Again, the `libc` shipped with Frama-C can handle all these functions. So Frama-C is able to analyse this code base composed by the `main.c` and `io.c` files.

This sample presents another hurdle which hinders its analysis. The EVA plugin cannot handle recursive function and thus the plugin abort the analysis ended it with an error.

To circumvent this issue, Frama-C has the `-eva-ignore-recursive-calls` switch. This option makes the EVA plugin ignores all the recursive functions it could face making them unreachable. This solution alone is not enough to correctly analyse this sample because the vulnerable code paths go through the recursive functions.

The `-inline-call` option lets the analyst defines functions which will be inlined in the code analysed by Frama-C.

The combination of the two options forces Frama-C to analyse these functions at least once before ignoring them. Thus, with this trick, the tool is able to correctly analyse the samples with recursive and mutually recursive functions.

3.2.3 Frama-C's results

There is no red alarm pointing to the known heap buffer overflow in the Frama-C's output.

There are multiple warnings at the statement where the vulnerable `strcat` is called. However, the alarms raised do not give more information about the reason.



```

tmp = strlen((char const *)data);
;
}
if ((tmp + note->size) + (size_t)1 > note->cap) {
    note->cap *= (size_t)2;
    /*@ assert Eva: dangling_pointer: ~\dangling(&note->buf); */
    note->buf = (char *)realloc((void *)note->buf, note->cap);
    /*@ assert Eva: dangling_pointer: ~\dangling(&note->buf); */
    if (note->buf == (char *)0) {
        dprintf_va_5(2, "ERROR %s:%d:\tbad alloc\n", (char *)"src/main.c", 124);
        exit(1);
    }
}
/* preconditions of strcat:
requires valid_string_src: valid_read_string(data);
Non transposable: requires valid_string_dest: valid_string(dest);
Non transposable: requires room_string: \valid(dest + (0 .. strlen(dest) + strlen(src))); */
/*@ assert Eva: dangling_pointer: ~\dangling(&note->buf); */
strcat(note->buf, (char const *)data);
{ /* sequence */
    tmp_0 = strlen((char const *)data);
    note->size += tmp_0;
}
__retres = note;
return_label: return __retres;
}

```

Figure 7 - Excerpt of the Frama-C's analysis of the Heap Buffer overflow sample

Besides, there are 38 alarms related to invalid memory accesses and no indication can lead analyst to consider in priority this alarm rather than the others.

Also, the alarm about the use of a dangling pointer seems to be a false positive.

3.2.4 Clang static analyser's results

```
> scan-build gcc "-I." libcgc.c src/io.c src/main.c
[...]  
src/main.c:523:24: warning: Potential leak of memory pointed to by 'argv'  
    note_t* note = get_note(argv[0]);  
                        ^~~~~~  
src/main.c:536:9: warning: Branch condition evaluates to a garbage value  
    if (resp)  
        ^~~~  
2 warnings generated.  
scan-build: 2 bugs found.
```

Clang static analysers find two potential bugs not related to the one described by its author and expected by the benchmark.

3.3 Vulnerability 3 – Null pointer dereference

3.3.1 Basic sample

```

int main(int argc, char **argv)
{
    int __retres;
    int *num1;
    int *num2;
    int result;
    num1 = (int *)malloc(sizeof(int) * (unsigned int)1);
    if (num1 == (int *)0) {
        {
            __retres = 1;
            goto return_label;
        }
    }
    num2 = (int *)malloc(sizeof(int) * (unsigned int)1);
    if (num2 == (int *)0) {
        {
            __retres = 2;
            goto return_label;
        }
    }
    num1 = input_num(num1);
    num2 = input_num(num2);
    /*@ assert Eva: mem_access: \valid_read(num1); */
    /*@ assert Eva: mem_access: \valid_read(num2); */
    /*@ assert Eva: signed_overflow: -2147483648 ≤ *num1 + *num2; */
    /*@ assert Eva: signed_overflow: *num1 + *num2 ≤ 2147483647; */
    result = *num1 + *num2;
    printf_va_2("result: %d\n", result);
    __retres = 0;
    return_label: return __retres;
}

```

Figure 8 - Frama-C's result for the basic null pointer dereference sample

The function `input_num` can return null and so invalidate the pointers `num1` and `num2`. Thus, it is not a valid operation to try to dereference them.

Frama-C raises two alarms at the line 28 of the `main.c` file.

[eva:alarm]	main.c:28:	Warning:	out	of	bounds	read.	assert
\valid_read(num1);							
[eva:alarm]	main.c:28:	Warning:	out	of	bounds	read.	assert
\valid_read(num2);							

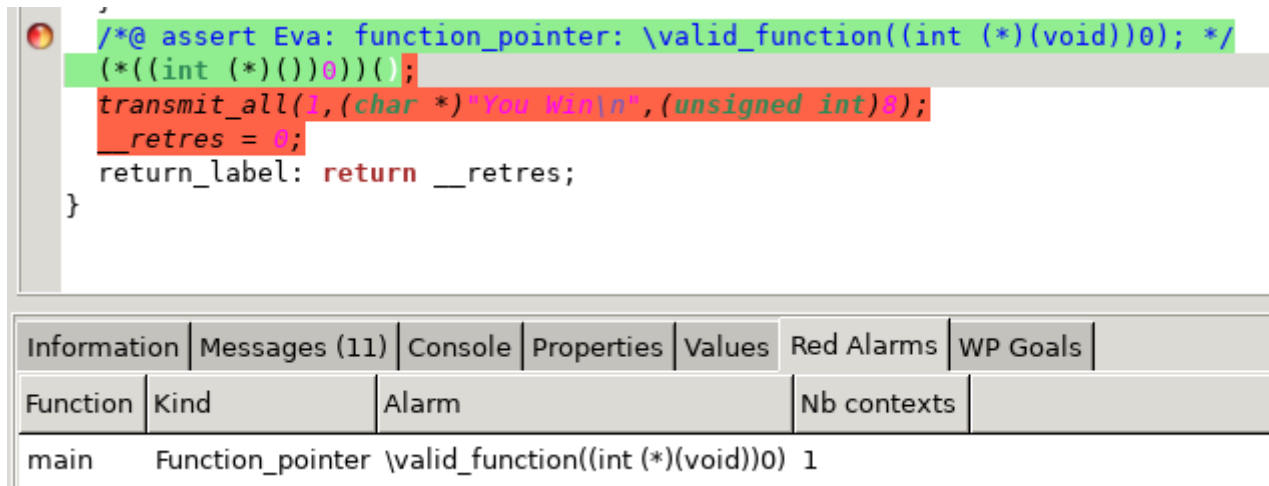
3.3.2 Modifications made to the sample

This sample is also easily handled by Frama-C. It is only composed of the `main.c` file. It does not have any dependencies to a custom `libc`. Instead, it uses directly the `transmit` and `receive` "syscalls". With our compatibility layer which defines those two functions over `read` and `write`, Frama-C is able to process it.

3.3.3 Frama-C's results

The EVA analysis made by Frama-C detects the known null pointer dereference as shown in the Figure 9. It raises a unique red alarm which points directly to the issue.

Frama-C also generates 8 other alarms related to out of bounds access to memory and integer overflows.



Information	Messages (11)	Console	Properties	Values	Red Alarms	WP Goals
Function	Kind	Alarm			Nb contexts	
main	Function_pointer	\valid_function((int (*)(void))0)			1	

Figure 9 - Red alarm for the null pointer dereference vulnerability

3.3.4 Clang static analyser's results

The following listing shows the output of the analysis of the sample by Clang static analyser.

```
> scan-build gcc "-I." libcgc.c src/main.c
[...]
src/main.c:207:3: warning: Called function pointer is null (null
dereference)
    ((int (*)(void))0)();
    ^~~~~~
1 warning generated.
scan-build: 1 bug found.
```

The analyser successfully identifies the expected vulnerability. It is also able to produce the code path, which leads to the vulnerability.

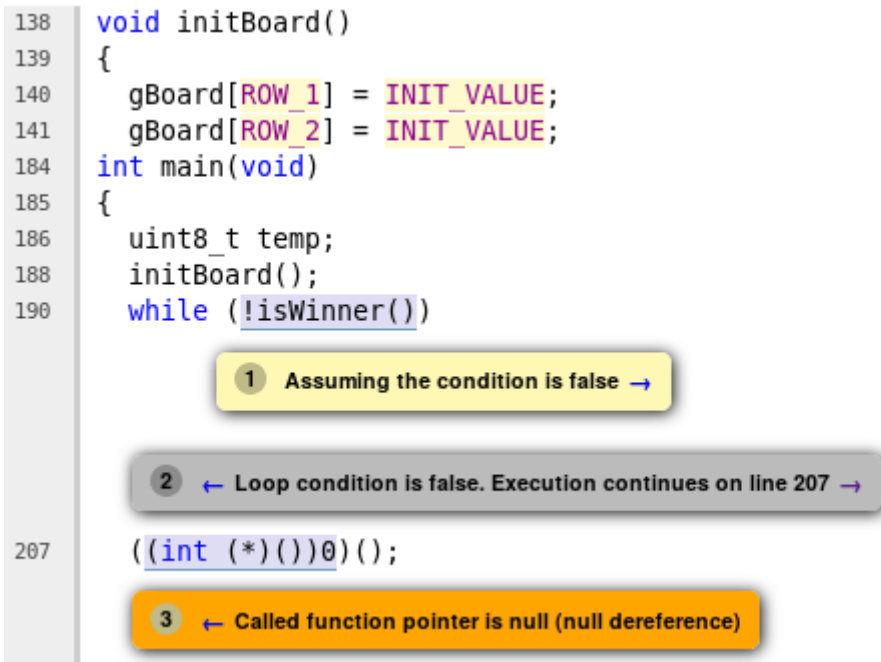


Figure 10 - Code path leading to the Null pointer dereference vulnerability

3.4 Vulnerability 4 – Use after free

3.4.1 Basic sample

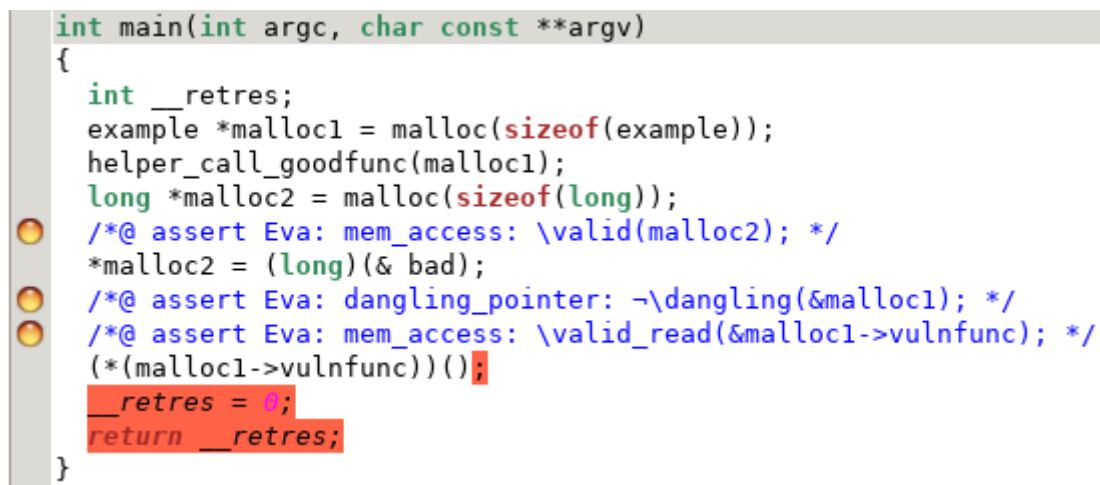


Figure 11 - Frama-C's result for the basic use-after-free sample

The call to the `helper_call_goodfunc` frees the memory pointed by the `malloc1` variable. So, when the call returns, the `malloc1` is effectively a dangling pointer and should not be used anymore.

Frama-C's analysis displays two warnings related to the use-after-free vulnerability at line 30 of the `main.c` file.

Frama-C detects that the `malloc1` pointer is a dangling pointer and that the access to the `vulnfunc` field is an out of bound access.

```

[eva:alarm] main.c:30: Warning:
    accessing left value that contains escaping addresses.
    assert -\dangling(&malloc1);
    
```

```
[eva:alarm] main.c:30: Warning:
    out of bounds read. assert \valid_read(&malloc1->vulnfunc);
```

3.4.2 Modifications made to the sample

The Use after free sample depends on standard functions relate memory and string handling. So there is no issue for Frama-C on this part.

However, two non-crashing off-by-one bugs hinder the analysis. These bugs put Frama-C in an invalid state and Frama-C cannot continue its analysis.

These bugs are not expected nor described in the sample description. Yet, they are genuine bugs discovered by Frama-C.

The two following patches were applied to fix theses bugs and obtain a better coverage for Frama-C's analysis.

```
1 //for (i = 0; i <= sizeof(g_password)/sizeof(g_password[0]); ++i)
2 // patch off-by-one
3 for (i = 0; i < 20; ++i)
4 {
5   cgc_random(&c, sizeof(c), NULL);
6   [...]
7 }
8 g_password[i] = '\0'
```

```
1
2 //for (i = 0; i < sizeof(default_movies)/sizeof(default_movies[0]); ++i)
3 for (i = 0; i < 10; ++i)
4 {
5   movie = (movie_t *) malloc(sizeof(movie_t));
6   [...]
7 }
```

3.4.3 Frama-C's result

The analysis raises 164 alarms but no red ones. There are 63 warnings about manipulation of dangling pointers which gives some hints about issues related to the handling of the dynamic memory.

The expected alarm should highlight the bad use of the global `movie_list_t movies_rented` variable. This list shares the same pointers as the ones contained in the `movie_list_t movies_full` variable. However, the program forgets to remove a movie object from both these lists when a movie is deleted. This behaviour yields some potential dangling pointers in the `movies_rented` list which could lead to a use-after-free vulnerability. Therefore, Frama-C raises alarms about the use of the pointer in the `movies_rented` list. Yet, it raises the same kinds of alarms when the program uses the pointers stored in the `movies_full` list. The excerpt of the `list_movies` function, Figure 12, shows these warnings for both of the list. The ones related to the `movies_full` list are false positives.

```

/*@ assert Eva: dangling_pointer: ~\dangling(&movies_full); */
node = movies_full;
while (node != (movie_node_t *)0) {
    /*@ assert Eva: signed_overflow: num_movies + 1 ≤ 2147483647; */
    num_movies++;
    /*@ assert Eva: initialization: \initialized(&node->movie); */
    /*@ assert
        Eva: initialization: \initialized(&(node->movie)->print_info);
    */
    /*@ assert
        Eva: function_pointer: \valid_function((node->movie)->print_info
    */
    /*@ assert Eva: dangling_pointer: ~\dangling(&node->movie); */
    ((*((node->movie)->print_info))(num_movies,node->movie);
}
/*@ assert Eva: initialization: \initialized(&node->next); */
/*@ assert Eva: dangling_pointer: ~\dangling(&node->next); */
node = node->next;
}

/*@ assert Eva: dangling_pointer: ~\dangling(&movies_rented); */
node = movies_rented;
while (node != (movie_node_t *)0) {
    /*@ assert Eva: signed_overflow: num_movies + 1 ≤ 2147483647; */
    num_movies++;
    /*@ assert Eva: initialization: \initialized(&node->movie); */
    /*@ assert
        Eva: initialization: \initialized(&(node->movie)->print_info);
    */
    /*@ assert
        Eva: function_pointer: \valid_function((node->movie)->print_info
    */
    /*@ assert Eva: dangling_pointer: ~\dangling(&node->movie); */
    ((*((node->movie)->print_info))(num_movies,node->movie);
}
/*@ assert Eva: initialization: \initialized(&node->next); */
/*@ assert Eva: dangling_pointer: ~\dangling(&node->next); */
node = node->next;
}

```

Figure 12 - Alarms related to dangling pointers in the use-after-free sample

3.4.4 Clang static analyser's results

```

> scan-build gcc "-I." libcgc.c src/main.c src/movie.c
[...]
src/main.c:431:9: warning: Branch condition evaluates to a garbage value
    if (movie->desc)
        ^~~~~~
src/main.c:657:7: warning: Branch condition evaluates to a garbage value
    if (new_title)

```



```

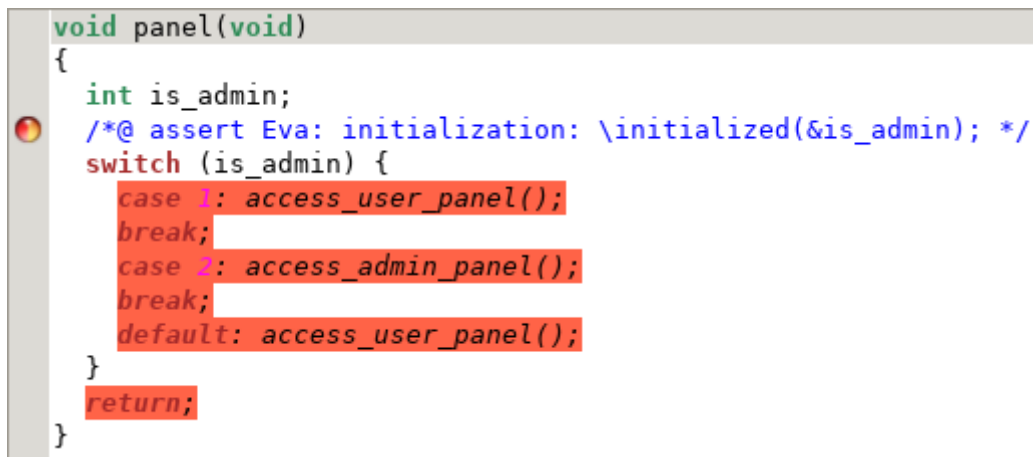
^~~~~~
src/main.c:659:7: warning: Branch condition evaluates to a garbage value
    if (new_desc)
    ^~~~~~
3 warnings generated.
src/movie.c:135:11: warning: Access to field 'next' results in a
dereference of a null pointer (loaded from variable 'prev')
    tmp = prev->next;
    ^~~~~~
1 warning generated.
scan-build: 4 bugs found.

```

The results obtained by Clang static analysers do not concern the expected vulnerability, nor the second vulnerability explained in the sample's description in the `README.md` file. It raises four alerts, which seem to be false positives.

3.5 Vulnerability 5 – Uninitialised variable

3.5.1 Basic sample



```

void panel(void)
{
    int is_admin;
    /*@ assert Eva: initialization: \initialized(&is_admin); */
    switch (is_admin) {
        case 1: access_user_panel();
        break;
        case 2: access_admin_panel();
        break;
        default: access_user_panel();
    }
    return;
}

```

Figure 13 – Frama-C's output for the basic uninitialised variable sample

The variable `is_admin` is defined at the beginning of the `panel` function but no value is given to it. When the switch statement reads its value, the behaviour of the process is undefined.

Frama-C detects the use of the uninitialised variable `is_admin` at the line of the `main.c` source file.

```

[eva:alarm] main.c:18: Warning:
    accessing uninitialized left value. assert \initialized(&is_admin);

```

3.5.2 Modifications made to the sample

This sample is only one `main.c` file. It uses the standard POSIX strings and memory-related functions, so Frama-C handles it without difficulty.

3.5.3 Frama-C's results

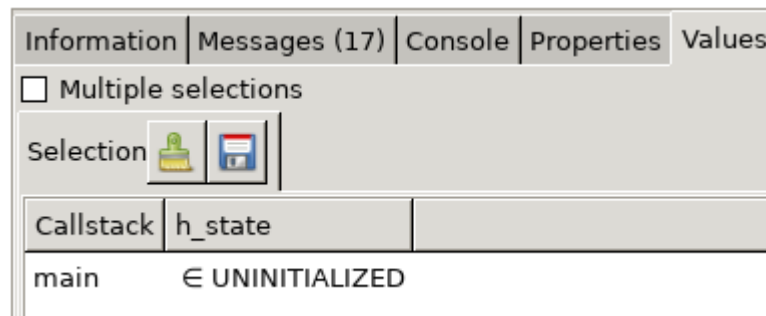
Frama-C's EVA plugin outputs a red alarm for the use of the `h_state` variable before its initialisation. The analysis highlights precisely a vulnerable statement related to the known vulnerability.

```
void play_game(void)
{
    hackman_state_t h_state;
    char buf[128];
    int i;
    int found;
    int error;
    while (1) {
        {
            char tmp_0;
            error = 0;
            if (win) {
                goto new_chal;
            }
            else {
                size_t tmp;
                /* preconditions of strlen:
                 requires
                 valid_string_s: valid_read_string((char const *)h_state.word); */
                tmp = strlen((char const *)h_state.word);
                if (tmp == (size_t)0) {
                    goto new_chal;
                }
            }
        }
    }
}
```

Information	Messages (17)	Console	Properties	Values	Red Alarms	WP Goals
Function	Kind	Alarm			Nb contexts	
strlen	property	requires valid_string_s: valid_read_string(s) 1				

Figure 14 - Red alarm spotting the uninitialised variable vulnerability

However, the description given in the alarm does not give any indication about the genuine issue. The description informs the user that the `h_state.word` field which is sent to the `strlen` function is not always a valid string. To understand why, the analyst has to check the abstract value computed by the EVA plugin for the `h_state` variable, which is `UNINITIALIZED`. With this information, the analyst can deduce that the underlying vulnerability is an uninitialised variable.

Figure 15 - `h_state` abstract value computed by the EVA plugin

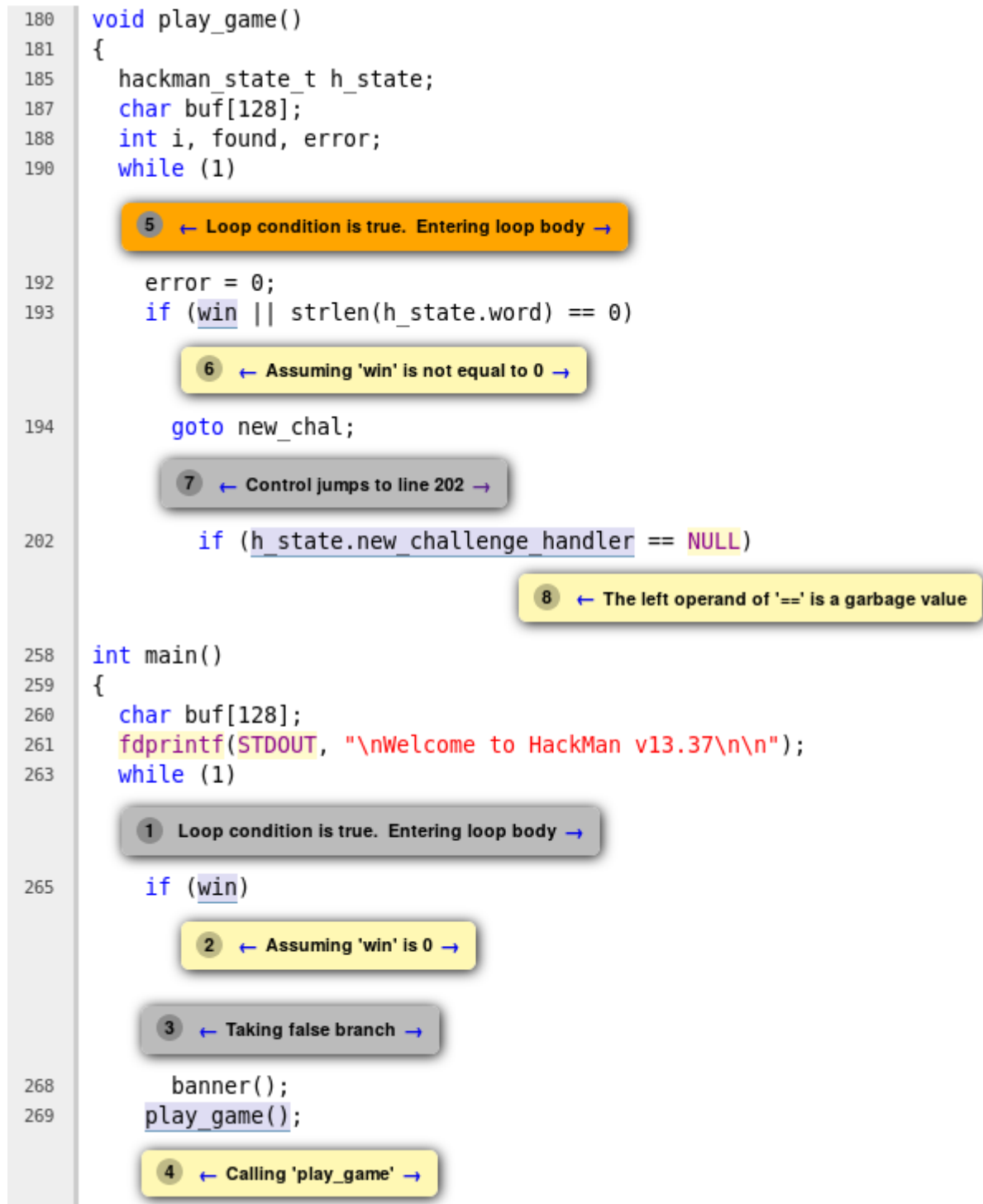
3.5.4 Clang static analyser's results

The next listing displays the output given by the analysis of the uninitialised variable sample by the Clang static analyser.

```
scan-build: Using '/home/fsl/Tools/clan+llvm-8.0.0/bin/clang-8' for static
analysis
src/main.c:202:43: warning: The left operand of '==' is a garbage value
    if (h_state.new_challenge_handler == NULL)
        ~~~~~~~~~~~~~~~~~~~~~~ ^
1 warning generated.
scan-build: 1 bug found.
```

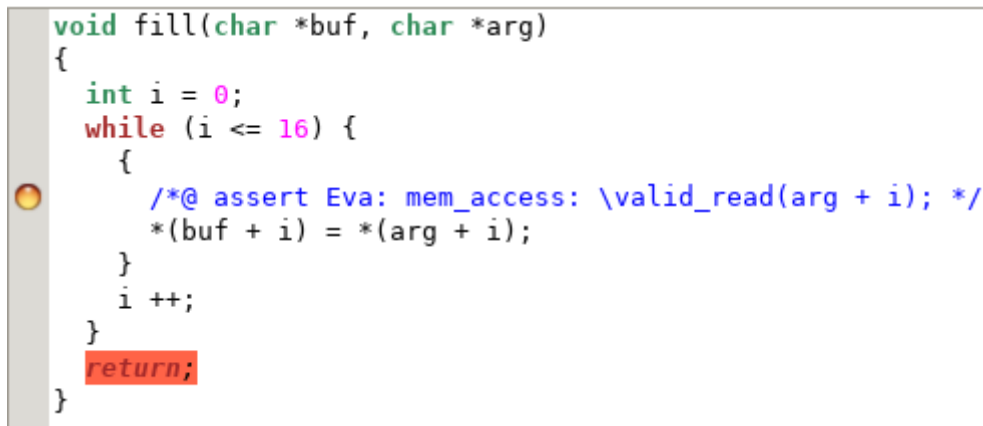
The bug reported by Clang static analyser seems related to the expected vulnerability. Indeed, the `h_state` variable can be used before being initialised. However, the line where the warning was found is different from the one found by Frama-C.

We studied the code path found by Clang static analyser to understand where this discrepancy comes from. The path is displayed in the Figure 16. Clang static analyser returns a spurious path. Actually, this path contains two checks against the global variable `win` at the steps n°2 and n°6. However, the assumptions made by the static analysers are incompatible. In the step n°2, the `win` variable must be equal to 0. Yet, in the step n°6, `win` must be different from 0. Therefore, the alarm raised by the Clang static analyser is wrong even though the warning highlights a genuine problem in the code source.

Figure 16 - Code path to trigger the use of uninitialized `h_state` variable

3.6 Vulnerability 6 – Off by one

3.6.1 Basic sample



```
void fill(char *buf, char *arg)
{
    int i = 0;
    while (i <= 16) {
        /*@ assert Eva: mem_access: \valid_read(arg + i); */
        *(buf + i) = *(arg + i);
    }
    i++;
}
return;
}
```

Figure 17 – Frama-C’s result for the basic off-by-one sample

Similarly to the stack buffer overflow vulnerability §3.1.1, the destination buffer, the `buf` variable, is 16 bytes long. However, the loop iterates 17 times. Thus, there is an overflow of only one byte on the stack of the process.

Frama-C alerts us about the line 7 of the `main.c` file where the off-by-one vulnerability is present.

```
[eva:alarm] main.c:8: Warning: out of bounds read. assert \valid_read(arg + i);
```

3.6.2 Modifications made to the sample

This sample is the one which required the most modifications. It implements an in-memory file system. Unfortunately, the name chosen for its internal structure and its API collide with the I/O stream API of the POSIX API: `FILE`, `fopen`, `fread`... To fix this, we added the prefix `fs_` or `FS` to each symbol, which raised an error.

The second issue raised by this sample concerns the implementation of the random generator in `lib/prng.c` file. The random generator uses source entropy given by the DECREE platform. This source is accessible through a hardcoded memory page, which starts at the address `0x4347C000`. Frama-C is not aware that this memory page is always mapped in the process memory. So it gets stuck in an invalid state when facing any statement trying to access this random page⁶.

The vulnerabilities expected in this sample are not affected by the code in `lib/prng.c`. Thus, we feel safe to modify the behaviour of this random generator for the sake of the experiment. Even if the modifications reduce its entropy.

We replaced the hardcoded access to the memory page to a global buffer with a hardcoded value in `src/service.c` file.

```
const char rand_page[] = "HARCODEDSECRET [...] FRAMA-CISOK!!";
```

⁶ It appears there is an option in Frama-C to declare a range in the process memory which is always valid to read or write. This option called `-absolute-valid-range` is the preferred solution to handle hardcoded pointer.

This simple change was enough to let Frama-C fully analyse the sample.

3.6.3 Frama-C's results

The vulnerable statement for this sample is a call to `strcpy` in the `PrependCommandHistory` function in the `shell.c` file. The Frama-C analysis displays three alerts for this statement because Frama-C cannot prove that the requirements needed to call the `strcpy` function hold here.

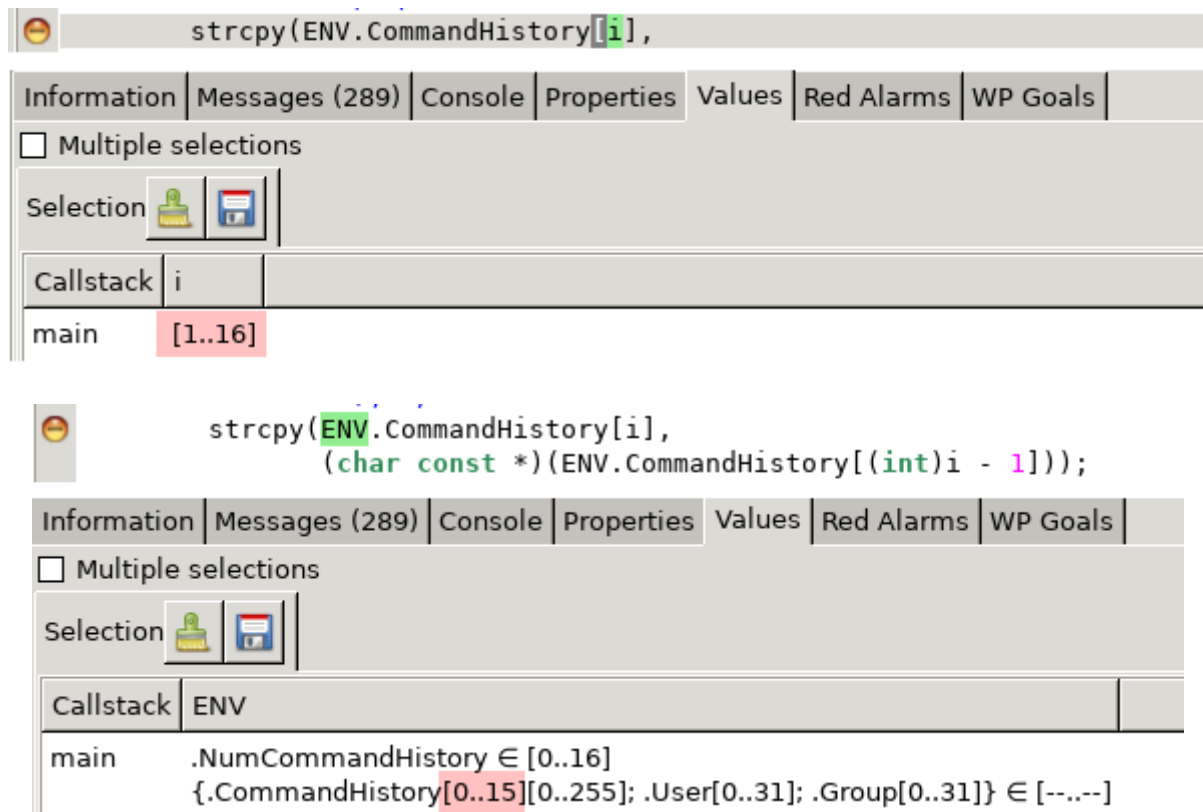
There are 32 alarms about non valid arguments for the `strcpy` function in the analysis's output. The truly vulnerable statement has to be extracted from all the other cases.

```
void PrependCommandHistory(char *buf)
{
    uint8_t i;
    ENV.NumCommandHistory = (unsigned char)0;
    i = (unsigned char)16;
    while ((int)i > 0) {
        {
            if ((int)ENV.CommandHistory[(int)i - 1][0] != '\000') {
                /* preconditions of strcpy:
                requires
                valid_string_src:
                valid_read_string((char const *)ENV.CommandHistory[(int)
                ((int)i - 1]);
                requires
                room_string:
                \valid((char *)ENV.CommandHistory[i] +
                (0 ..
                strlen((char const *)ENV.CommandHistory[(int)
                ((int)i - 1)]));
                requires
                separation:
                \separated(
                (char *)ENV.CommandHistory[i] +
                (0 ..
                strlen((char const *)ENV.CommandHistory[(int)((int)i - 1])),
                (char const *)ENV.CommandHistory[(int)((int)i - 1]) +
                (0 ..
                strlen((char const *)ENV.CommandHistory[(int)((int)i - 1]))
                ); */
                strcpy(ENV.CommandHistory[i],
                (char const *)ENV.CommandHistory[(int)i - 1]);
                if ((int)ENV.NumCommandHistory == 0) {
                    ENV.NumCommandHistory = i;
                }
            }
        }
    }
}
```

Figure 18 - Alarms raised for the call to the `strcpy` vulnerable to an off-by-one buffer overflow

Furthermore, no alert of type `out of bounds read` is raised for the `ENV.CommandHistory[i]` expression.

Yet, the variation domains computed by the EVA plugin give some hints about the off-by-one vulnerability. The index `i` ranges from 1 to 16 even though the `ENV.CommandHistory` field is an array of only 15 elements.

Figure 19 - EVA plugin outputs for the `ENV` and `i` variables in the off-by-one sample

3.6.4 Clang static analyser's results

The Clang static analyser does not find any bugs in this sample.

```
> scan-build gcc "-I." libcgc.c src/fs.c src/io.c src/screen.c
src/service.c src/shell.c src/user.c lib/prng.c
scan-build: Removing directory [...] because it contains no reports.
scan-build: No bugs found.
```

3.7 Vulnerability 7 – Double free

3.7.1 Basic sample

```
int main(void)
{
    int __retres;
    char *a;
    char *b;
    a = (char *)malloc((unsigned int)64);
    free((void *)a);
    b = (char *)malloc((unsigned int)64);
    /*@ assert Eva: dangling_pointer: -\dangling(&a); */
    free((void *)a);
    __retres = 0;
    return __retres;
}
```

Figure 20 – Frama-C's output for the basic double free sample

This sample calls the function `free` two times on the pointer `a`.

Frama-C shows an alert at the line 9 of the `main.c` file where the second call to `free` function on the pointer `a` is made.

```
[eva:alarm] main.c:9: Warning:
    accessing left value that contains escaping addresses.
    assert -\dangling(&a);
```

3.7.2 Modifications made to the sample

The samples source code is four C files: `main.c`, `ktv.c`, `hashtable.c` and `array.c`.

The only necessary modifications were to modify the include directives to use the standard `libc`.

This sample also contains some recursive functions. So, it is necessary to use the combination of the `-inline-calls` and `-eva-ignore-recursive-function` option to successfully analyse this sample with Frama-C.

3.7.3 Frama-C's results

The analysis of this sample took longer than the other ones. During our experiments, it took 269 seconds.


```

case (kty_type_t)KTY_OBJECT:
/*@ assert Eva: initialization: \initialized(&item->item.i_object); */
/*@ assert Eva: mem_access: \valid_read(&item->item.i_object); */
/*@ assert Eva: dangling_pointer: ~\dangling(&item->item.i_object); */
htbl_destroy(item->item.i_object);
break;
default: break;
}
/*@ assert Eva: dangling_pointer: ~\dangling(&item); */
free((void *)item);
}
return;
}

```

Figure 21 - Frama-C's result for the double free sample

A warning is displayed related to the double free vulnerability. Frama-C raises 424 alarms in total and five red alarms unrelated to the known vulnerability.

The sample's description reports a secondary vulnerability, which is a stack buffer overflow. Yet, Frama-C does not raise any warning for the guilty `strcpy` statement.

```

kty_item_t *item = array_get(parser->nyan_says,i);
strcpy(c,(char const *)item->item.i_string.s);
c += item->item.i_string.len;
}
i++;
}
dprintf_va_5(1,"NYAN SAYS...\n"\n%s\n"\n",buf);
return_label: return;
}

```

Figure 22 - Undetected stack buffer overflow in the heap buffer overflow sample

Besides, the statement is highlighted in red which means that for Frama-C the statement is unreachable. Thus, Frama-C did not analyse it.

The guard `parser->cats < 3` in the `nyan` function, displayed in the Figure 23, makes this code unreachable, Frama-C EVA's plugin computes a set of value of {0; 1} for `parser->cats` field. Therefore, the condition is always true and the `nyan` function returns directly without further processing.

```

/*@ assert Eva: initialization: \initialized(&parser->dumps); */
/*@ assert Eva: mem_access: \valid_read(&parser->dumps); */
(*(parser->dumps))(my_kty);
if (parser->cats < (unsigned int)3) {
{
goto return_label;
}
}
dprintf_va_4(1,"%s",nyan_cat);
c = buf;
i = 0;
while (1) {
int tmp_0;
{ /* sequence */
tmp_0 = array_length(parser->nyan_says);
}
}
}

```

Figure 23 - Guard making the stack buffer overflow unreachable in the `nyan` function

3.7.4 Clang static analyser's results

The Clang static analyser finds six bugs in this sample source code.

```
> scan-build gcc "-I." libcgc.c src/main.c src/kyt.c src/hashtable.c
src/array.c
[...]
src/kyt.c:46:20: warning: Value stored to 'tmp' during its initialization
is never read
    char c[2] = {0}, tmp[2] = {0};
                   ^~~      ~~~

src/kyt.c:228:8: warning: Value stored to 'decimal' during its
initialization is never read
    char decimal[3] = {0};
      ^~~~~~      ~~~

src/kyt.c:405:7: warning: Null pointer passed as an argument to a 'nonnull'
parameter
    if (strcmp("nyan_says", key) == 0 && new->type == KTY_STRING)
      ^~~~~~~~~~~~~~~~~~~~~~

src/kyt.c:437:9: warning: Null pointer passed as an argument to a 'nonnull'
parameter
    if (strcmp("nyan_says", key) == 0 && new->type == KTY_STRING)
      ^~~~~~~~~~~~~~~~~~~~~~

4 warnings generated.
src/hashtable.c:52:9: warning: Branch condition evaluates to a garbage
value
    if (table->table)
      ^~~~~~

1 warning generated.
src/array.c:47:9: warning: Branch condition evaluates to a garbage value
    if (arr->arr)
      ^~~~~~

1 warning generated.
scan-build: 6 bugs found.
```

None of the warnings raised by Clang static analyser are related to the expected “Double Free” vulnerability.

3.8 Vulnerability 8 – Format string

3.8.1 Basic sample

Frama-C's analysis displays multiple alerts, which are not related to the format string vulnerability.

The vulnerability occurs at the second call to `printf` that has as first argument `*(argv + 1)`. This pointer is the first argument in the command line given to the process. Thus, it is controllable by the user.

```
int main(int argc, char **argv)
{
    int __retres;
    int tmp;
    if (argc < 3) {
        puts("Please input username and password to authenticate yourself.");
        {
            __retres = 1;
            goto return_label;
        }
    }
    printf_va_1("Authenticating user ");
    {
        void *__va_args[1] = {(void *)0};
        /*@ assert Eva: mem_access: \valid_read(argv + 1); */
        printf((char const *)*(argv + 1), (void * const *)__va_args);
    }
    /*@ assert Eva: mem_access: \valid_read(argv + 2); */
    tmp = auth(*(argv + 1), *(argv + 2));
    if (tmp != 1) {
        puts("\n- Incorrect user or password");
        {
            __retres = 1;
            goto return_label;
        }
    }
    puts("\n+ Welcome to the admin panel.");
    __retres = 0;
    return_label: return __retres;
}
```

Figure 24 – Frama-C's output for the basic format string sample

The bad use of the format string parameter of the `printf` function is not handled by the EVA plugin but by the Variadic plugin. It seems that the Variadic plugin's alerts are not shown in the GUI but solely in the console output.

Yet, Frama-C generates a warning for the line 21 where the format string vulnerability happens.

```
[variadic] main.c:21: Warning:
  Call to function printf with non-static format argument:
  no specification will be generated.
```

Format string warnings are not displayed in the "Messages" windows of the Frama-C GUI. An analyst has to search through the console output generated by the analysis.

3.8.2 Modifications made to the sample

The samples source code is seven C files: `admin.c`, `cmdb_backend.c`, `main.c`, `readline.c` and `user.c`.

The only necessary modifications were to modify the include directives to use the standard `libc`.

3.8.3 Frama-C's results

The EVA plugin's analysis generates 68 alarms which are unrelated to the known format string vulnerability. However, for this experiment, only the variadic plugin's output is really interesting. It raises three warnings.

There is only one warning related to the call of the `printf` function with non-static format argument leading an analyst directly to the issue.

3.8.4 Clang static analyser's results

To analyse this sample with the Clang static analyser we used the following command line.

```
> scan-build gcc "-I." libcgc.c src/admin.c src/cmdb_backend.c src/cmdb.c
src/debug.c src/main.c src/readline.c src/user.c
[...]
src/cmdb_backend.c:134:12: warning: Potential leak of memory pointed to by
'row'
    return 0;
    ^
1 warning generated.
scan-build: 1 bug found.
```

The bug reported by Clang is not related to the "Format String" vulnerability.

Nowadays, the C compilers detect this kind of blatant format string, thus we recompiled the samples with a stricter set of compiler checks. We used the `-Wall` command line flag for this purpose.

Surprisingly, the GCC compiler, in this 8.3.0 version, does not raise any warning about the vulnerable `printf`.

```
> gcc -Wall "-I." libcgc.c src/admin.c src/cmdb_backend.c src/cmdb.c
src/debug.c src/main.c src/readline.c src/user.c
src/cmdb_backend.c: In function 'add_entry':
src/cmdb_backend.c:146:16: warning: unused variable 'i' [-Wunused-
variable]
    size_t i;
    ^
src/cmdb_backend.c: In function 'print_movies':
src/cmdb_backend.c:263:18: warning: format '%d' expects argument of type
'int', but argument 2 has type 'size_t' {aka 'long unsigned int'} [-
Wformat=]
    printf("%d movie(s)\n", g_list_length);
    ~^          ~~~~~~
    %ld
```

```
src/cmdb_backend.c:278:18: warning: format '%d' expects argument of type
'int', but argument 2 has type 'size_t' {aka 'long unsigned int'} [-Wformat=]
    printf("%d movie(s)\n", g_num_rented);
        ~^
        ~~~~~~

    %ld
```

However, the Clang 8.0.0 compiler shows a warning at the guilty `printf` statement.

```
>s clang -Wall "-I." libcgc.c src/admin.c src/cmdb_backend.c src/cmdb.c
src/debug.c src/main.c src/readline.c src/user.c
src/cmdb_backend.c:146:16: warning: unused variable 'i' [-Wunused-
variable]
    size_t i;
        ^

src/cmdb_backend.c:263:33: warning: format specifies type 'int' but the
argument has type 'size_t' (aka 'unsigned long') [-Wformat]
    printf("%d movie(s)\n", g_list_length);
        ~~
        ^~~~~~

    %zu

src/cmdb_backend.c:278:33: warning: format specifies type 'int' but the
argument has type 'size_t' (aka 'unsigned long') [-Wformat]
    printf("%d movie(s)\n", g_num_rented);
        ~~
        ^~~~~~

    %zu

src/cmdb_backend.c:288:16: warning: format string is not a string literal
(potentially insecure) [-Wformat-security]
    printf(g_all_genres[i]);
        ^~~~~~

src/cmdb_backend.c:288:16: note: treat the string as an argument to avoid
this
    printf(g_all_genres[i]);
        ^
    "%s",
```

Chapter 4 Experiment results summary

We used the `-eva-metrics-cover` option from the Frama-C command line tool to obtain the coverage associated of the EVA plugin analysis. This metric shows the percentage of code that Frama-C succeeded to analyse. It was a useful tool to troubleshoot the issue faced when we analysed the samples.

Unfortunately, only Frama-C is able to output the coverage of the analysed source code. Thus, we miss the data for the Clang static analyser tool.

Sample	EVA function coverage	EVA statements coverage in those functions (%)	# lines of code
Stack buffer overflow	50%	95%	447
Heap buffer overflow	32%	91%	579
Null pointer dereference	42%	90%	170
Use after free	37%	95%	753
Off by one	28%	93%	1468
Uninitialised variable	4%	50%	223
Double free	63%	98%	1398
Format String	68%	97%	759

Table 2 - Coverage obtained by Frama-C's analysis

Overall, the function coverage of the EVA plugin is pretty low, because the plugin computes the coverage over all the provided code to Frama-C, even if the code is unreachable. Nevertheless, Frama-C is able to produce an alarm related to each vulnerability in the benchmarks.

The uninitialised variable sample displays very low function coverage, as shown in Table 2. When Frama-C finds a red alarm, the code which follows is considered as unreachable. In this case, the vulnerability appears near the beginning of the main function. Thus, Frama-C finds it quickly and stops immediately. This behaviour explains this very low coverage. Yet, it does not hinder the ability of Frama-C to identify successfully the vulnerability in this case.

The [D4.3] document established three metrics to evaluate the static analysers:

1. **Detection:** a Boolean to check if the vulnerability is detected or not.
2. **Calculation time:** How long does the tool take to produce its results.
3. **False positive:** How many warnings raised are actually false positives.

At the end of the experiments, it appeared that the Calculation time was not relevant here. Once stripped down of their custom library, each sample are pretty small. Therefore, the analysis took a few seconds for both tools. One exception is the double free sample, which took 269 seconds for Frama-C to carry out its analysis. The reason that could explain this gap in analysis time could be

the use of the `–inline-call` option. In addition, the double free is the second biggest sample in terms of lines of code and has the better code coverage, as shown in Table 2.

We also reconsidered the false positive metric. It is a long, tedious, and error prone process to review each alarm raised by Frama-C to check if the alarm is spurious or not. Besides, if we only consider alarms directly related to the expected vulnerability, all the others alarms should be seen as false positives. Even though this method is much quicker, it is also unfair. Indeed, during the experiments Frama-C detected some unknown and genuine bugs in the samples. Therefore, without an exhaustive triaging of every alarm we could not compute a meaningful false positive metric. For all these reasons, we did not try to compute these metrics for the benchmarks.

This metric gives an indication about the precision and the usefulness of the static analyser. To fill the gap for the missing false positive metric, we introduced two new metrics in addition to ones described in the [D4.3] document. The first one simply tells if the static analyser is able to produce an alarm, which points to the vulnerability expected in each sample. We called it “Vulnerability found” and the metric is a boolean value. The alarm produced should stand out from among all the other alarms in a way that an evaluator cannot overlook it during an audit.

In the Frama-C static analyser’s vocabulary, the red alarms fits all the requirements. Thus, if there is a red alarm pointing to the guilty statement that induces the vulnerability we consider that Frama-C has found it.

We did not find the equivalent of the “red alarm” in the Clang static analyser. The warnings raised by the tool are not sorted. Yet, the Clang static analyser produces very few warnings. So we only checked the fact that a warning points to the portion of the code related to the expected vulnerability. If it is the case, we state that the vulnerability is found.

Sample	# Alarms (orange)	# RED ALARM	Vulnerability found by a RED ALARM	# Total alerts / # lines
Stack buffer overflow	75	0	No	16,78%
Heap buffer overflow	108	0	No	18,65%
Null pointer dereference	9	1	Yes	5,88%
Use after free	164	0	No	21,78%
Off by one	75	1	No	5,18%
Uninitialised variable	13	1	Yes	6,28%
Double free	424	5	No	30,69%
Format String	68	0	No (by the variadic plugin)	8,96%

Table 3 - Metrics about the alarms raised by Frama-C

Frama-C produces few red alarms. These alarms stand out the most and would be inspected first by an analyst. However, in only two cases, the Red were alarms raised by Frama-C were genuine ones. Furthermore, the Null pointer dereference sample and the uninitialised variable sample are also spotted by Clang static analyser. So, it seems that these cases are pretty simple to detect for a static analyser.

Because the variadic plugin's warnings are split from the alarms raised by the EVA plugin, Table 3 does not show that Frama-C successfully detects the Format string vulnerability.

The ratio between the number of alarms against the number of line of codes gives an idea about how useful Frama-C is during a code review. Too many alarms per line of code means that the manual triage would be too time consuming for an analyst in an evaluation scenario with restricted time budget.

Sample	# Alarms	Vulnerability found	# Total alerts / # lines
Stack buffer overflow	0	No	0,00%
Heap buffer overflow	2	No	0,35%
Null pointer dereference	1	Yes	0,59%
Use after free	4	No	0,53%
Off by one	0	No	0,00%
Uninitialised variable	1	Yes	0,00%
Double free	6	No	0,00%
Format String	1	No (by the compiler)	0,00%

Table 4 - Metrics about the alarms raised by the Clang static analyser

The metrics of the Clang static analyser are the total opposite of the Frama-C's one. Overall, the Clang static analyser is only able to detect the simplest samples: null pointer dereference and the uninitialised variable. Clang static analyser raised a lot less warnings but there are not relevant to the vulnerabilities we are looking for.

4.1 Experiments on the patched version of the source code

To obtain an idea about the relevance of the alarms raised by the static analysers, we made the same experiment on the patched version of each samples.

The expected result for these experiments is a decrease in the number of alarms and no alarms at all related to the patched vulnerabilities.

The DARPA CGC corpus contains the patched version of every samples. These code fixes are guarded by the use of the `PATCHED` macro.

In all the selected sample, the patch are straightforward and minimal most of the time, it only affects one statement in the sample's code base.

To build the version of each sample where the bug is fixed, we just enabled the `PATCHED` definition in the build chain. To do so, we added the command switch `-DPATCHED` to the command line used to analyse each sample with Clang static analyser. The equivalent switch for Frama-C is `-cpp-extra-args="-DPATCHED"`.

Patched Test cases	EVA function coverage	EVA statements coverage (%)	Alarms (orange)	RED ALARM	lines	Total alerts/ # lines
Stack buffer overflow	50%	95%	75	0	447	16,78%
Heap buffer overflow	32%	91%	109	0	594	18,35%
Null pointer dereference	46%	91%	8	0	169	4,73%
Use after free	36%	95%	165	0	756	21,83%
Off by one	28%	93%	75	1	1468	5,18%
Uninitialised variable	37%	93%	19	0	224	8,48%
Double free	57%	97%	424	5	1398	30,69%
Format String	68%	97%	68	0	756	8,99%

Table 5 - Coverage obtained by Frama-C's analysis on the patched source code

The patches applied to each sample only affects few lines in the source code. Therefore, there is no big difference in the coverage obtained by Frama-C in both experiments, when we compare Table 2 and Table 6.

Sample Patched	EVA function coverage	EVA statements coverage (%)	# lines
Stack buffer overflow	50%	95%	447
Heap buffer overflow	32%	91%	594
Null pointer dereference	46%	91%	169
Use after free	36%	95%	756
Off by one	28%	93%	1468
Uninitialised variable	37%	93%	224
Double free	57%	97%	1398
Format String	68%	97%	756

Table 6 - Coverage obtained by Frama-C on the patched version of the source code

The comparison between the Table 3 and the Table 7 highlights the fact that for the samples Stack buffer overflow, Heap buffer overflow, Off by one, Double free and Format String, the fix made to each sample did not affect the analysis made by Frama-C. Indeed, there is no difference between the two experiments. The number of alarms and red alarms are still the same. Going through all the alarms raised in both scenarios: vulnerable or patched source code, to confirm that each alarm is still the same is a lot of manual work. We compared the Frama-C's outputs for the Off by one, they were no difference in the outputs related to the alarms. An overview of others samples' outputs seems to confirm that most of the alarms are effectively the same: they target the same source code line and they have the same color level (orange or red).

Consequently, these outputs raise a doubt about the relevance of the subset of alarms that points to the vulnerable portion of the source code in each sample. These alarms are present in both cases. Thus, their origin seems to be a limitation of the EVA's analysis.

In two cases, Use after free and Uninitialised variable, the number of alarms increased, respectively by 6 and by 1. These results are counter-intuitive because they seem to indicate that these new alarms and the old ones are unrelated to the vulnerability we were looking for.

For the Uninitialised variable sample, this increase is explainable by the increase in function coverage of 33%, displayed in the Table 6. The EVA's analysis goes further because no red alarm is found and new potential defects are spotted by Frama-C.

For the two Null pointer dereference and Uninitialised variable samples, Frama-C's analysis does not raise any red alarm anymore. It was the expected results. The vulnerabilities are patched and there are neither blatant bugs nor vulnerabilities in their source code.

Sample Patched	# Alarms (orange)	# Red alarms	Red alarm related to the patched vulnerability (False positive)	# Total alerts/ # lines
Stack buffer overflow	75	0	No	16,78%
Heap buffer overflow	109	0	No	18,35%
Null pointer dereference	8	0	No	4,73%
Use after free	165	0	No	21,83%
Off by one	75	1	No	5,18%
Uninitialised variable	19	0	No	8,48%
Double free	424	5	No	30,69%
Format String	68	0	No (handled by the variadic plugin)	8,99%

Table 7 - Metrics about the alarms raised by Frama-C on the patched version of the samples

Table 8 shows the results obtained by the Clang static analysers on the patched samples. The tool does not produce any false positives related to the patched vulnerabilities. The two vulnerabilities found by Clang static analysers, the Null pointer dereference and Uninitialised variable samples, do not raise any alarms anymore. Therefore, the tool seems to handle those cases correctly.

However, the results for all the other samples are strictly the same. The Clang static analyser produced the same amount of alarms for each sample and the lines of the source code highlighted as buggy were still the same. Those results confirm the fact that those alarms were not related to the vulnerability we were trying to detect.

Sample	# Alarms	Alerts related to the patched vulnerability (False Positive)	# Total alerts / # lines
Stack buffer overflow	0	No	0,00%
Heap buffer overflow	2	No	0,35%
Null pointer dereference	0	No	0,59%
Use after free	4	No	0,53%
Off by one	0	No	0,00%
Uninitialised variable	0	No	0,00%
Double free	6	No	0,00%
Format String	1	No (handled by the compiler)	0,00%

Table 8 - Results of the Clang static analysers on the patched version of each samples

The experiments made on the patched samples seem to prove that the majority of the alarms raised by both of the static analysers are unrelated to the vulnerability we were trying to spot.

Both of the static analysers handle the Null pointer dereference and Uninitialised variable correctly. They are able to spot the vulnerability precisely through an alert or a red alarm. In addition, when the vulnerability is fixed no alarms is raised anymore about it.

Besides, thanks to its variadic plugin, Frama-C also handles the Format String sample the way we expect.

With regard to the two experiments, it seems that the both tools produce genuine alarms only for the two samples: Null dereference and Uninitialised variable. In addition, Frama-C with its Variadic plugin also handles correctly the Format string sample. The Clang static analyser for this part, does not produce any warnings related to this vulnerability. Yet, recent compilers are able to warn developers about this issue. Therefore, Clang static analyser could rely on this behaviour and does not even need to detect this class of vulnerability.

Summary and Conclusion

This document describes the experiments we made to test the capacity of two static analysers: Frama-C and Clang static analyser to handle and analyse vulnerable source code samples from the DARPA CGC corpus.

The main issue we faced when we try to analyse the benchmarks with Frama-C was the use of the non-standard DECREE platform. This is quite a rare situation to face for common software running on the prevalent platforms like Windows, Linux and macOS. Yet, this kind of situation is not unusual for software targeting embedded systems as the computing hardware platforms are often very different (e.g. DSP). The solution based upon a compatibility layer and modifying the source code if needed was not too expensive for our experiments, because the samples were small. This same approach does seem realistic for a bigger project with tens or hundreds of functions to “stub”. Moreover, it seems cumbersome to handle multiple versions of the same source code and maintain it only for Frama-C. A solution, which does not require to modify the source code but solely to configure Frama-C or its standard library, would be more convenient.

The benchmarks clearly show that Frama-C produces more genuine alarms than Clang static analysers. For each sample, the lines of code linked to the known vulnerability were highlighted. It also detects genuine bugs not planned by the benchmarks and bad practices. Yet, it also produces a lot more alarms in general which means that this false positives ratio is consequently higher. Furthermore, the alarms raised on the vulnerable samples are still displayed when we analyse the patched version these samples, except for the three most trivial samples: Null pointer dereference, Uninitialised variable and Format String. More investigation and expertise about the inner functioning of Frama-C are needed to explain this behaviour.

In the analyst’s point of view, the alarms raised by Frama-C are not always informative about the nature of the bug. Even though they point to the faulty lines in the code, Frama-C does not try to distinguish between a very large buffer overflow from an off-by-one, nor if the memory overflow is in the heap, the stack or the global data. With the data types of the variables and the variation domains computed by the EVA plugin, Frama-C could give more precise hints about the potential issue. These hints would be very valuable during the manual process of triaging the alerts.

With regard to the benchmarks proposed in the document [D4.3], once stripped-down of their custom dependencies, the DARPA CGC samples are small. Therefore, no significant difference was found in the time to analyse the samples by each static analysers.

The DARPA CGC competition’s purpose was to test binary analysers, static and dynamic. The analysers had to generate an input triggering the vulnerability in the sample. Consequently, some vulnerabilities are trivial to find statically like the null pointer dereference and the format string samples. The challenge resided in the capability of the analysers to forge the input reaching the vulnerable portion of code.

When it comes to the results, we used Frama-C and the previous knowledge about the vulnerabilities in the samples to identify quickly if Frama-C succeeds to detect each vulnerability. Even with this knowledge, it was not always straightforward to judge if the alarms raised by Frama-C were genuine or spurious. Therefore, our experiments do not represent the real process of a code review, where an analyst starts from the alarms to figure out if a genuine bug is present in the code.

The two major problems we faced that could hinder the use of Frama-C, during an evaluation, would be the time to build a compatibility layer if Frama-C is not able to analyse it by default. Besides, the quantity of alarms raised by Frama-C is still high considering the small size of the code base in the benchmarks.

Chapter 5 List of Abbreviations

Abbreviation	Translation
POSIX	The Portable Operating System Interface
API	Application programming interface
GUI	Graphical User Interface