Static Analysis and Runtime-Assertion Checking: Contribution to Security Counter-Measures

Dillon Pariente[1] and Julien Signoles[2]
dillon.pariente@dassault-aviation.com
julien.signoles@cea.fr

[1] Dassault Aviation
[2] CEA LIST, Software Reliabiliy and Security Laboratory

Abstract. This paper[3] presents a methodology which combines static analysis and runtime assertion checking in order to automatically generate counter-measures, and execute them whenever a flaw in the code which may compromise the security of an application is detected during execution. Static analysis pinpoints alarms that must be converted into runtime checks. Therefore the verifier is able to only monitor the security critical points of the application. This method allows to strengthen a security-critical source code in a cost-effective manner. We implemented it in the Frama-C framework and experimented it on a real use case based on Apache web server. The paper ends with preliminary considerations on potential perspectives for security evaluation and certification.

1 Introduction and context

Formal methods have been proved particularly powerful for runtime error search in safety context [3, 9, 26], and also – more recently – for security verification and validation purpose, as demonstrated by recent success stories and compliance with international security standards [27]. Applying static analysis tools on real use cases, however, may represent an unaffordable cost with respect to non-vital programs whose allocated budget for verification activities is usually small compared to highly critical applications. Actually this is one of the most recurrent criticisms regarding widely spreading static analysis over source code validation and verification industrial practices: proving security properties or ensuring the absence of safety alarms may need extra-efforts like complex interactive proofs, heavy code annotation activities or lots of tool parameterizations for which an important amount of time and expertise is often required.

[3] This work is done in the context of project VESSEDIA, which has received funding from the European Union's Horizon 2020 Research and Innovation Program under grant agreement No 731453.

Therefore, this paper presents a method – named CURSOR – applicable to non-critical source code. First it is well known that most attacks exploit at least one vulnerability (sometimes referenced as a CVE: Common Vulnerability and Exposure[4]), which itself may be caused by one or several Common Weakness Enumeration (CWE) entries[5] [6]. Based on this observation, CURSOR proposes to monitor some safety and security alarms raised by sound static analyzers (a relevant subset of the alarms detectable on C source code) in order to trigger predefined error-free counter-measures (CM) whenever alarms are really encountered at runtime. We have implemented this method within Frama-C [17], a framework for analysis of C source code which provides several sound static and dynamic analyzers, and experimented it on a real security-relevant use case: an Apache library. Indeed, Apache is one of the components used in an Aircraft e-Maintenance application (an ongoing development at Dassault Aviation) for which security analyses are required. The results demonstrate the usefulness of our approach, as well as its cost-efficiency.

Our **contributions** are therefore:

- **a new cost-effective method, CURSOR, 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 an Aircraft e-Maintenance application.

The paper is organized as follows. Section 2 presents Frama-C and its analyzers of interest for our study, namely its plug-ins Value and E-ACSL. Then Section 3 details the CURSOR method applied to an Apache library. Finally Section 4 briefly introduces practical considerations about security evaluation and certification concerns.

2 Tools: Static Analysis Contribution to Runtime-Assertion Checking

The CURSOR method, explained in Section 3, has been implemented within the Frama-C framework which we sum up here. More details can

[4] `http://cve.mitre.org`
[5] `http://cwe.mitre.org`
[6] See for instance `https://www.cvedetails.com/vulnerability-list/cweid-200/vulnerabilities.html`

be found in Frama-C-specific papers [17, 19]. Section 2.1 is a general overview of the framework while Section 2.2 and Section 2.3 describe the two most important plug-ins in our context, namely Value and E-ACSL, which respectively provide static analysis by abstract interpretation and runtime assertion checking.

2.1 Frama-C Overview

Frama-C is an open source platform[7] which aims at analyzing source code written in ISO C99. This code may be annotated in ACSL formal specification language [1] (briefly introduced later on in this section). Recently Frama-Clang has been released as a prototype Frama-C extension to handle C++ code. The platform gathers together code analyzers. It is written in OCaml [10] and based on a plug-in architecture [23]: each analyzer is a plug-in which is linked against the Frama-C kernel. This architecture allows users (including external ones) to easily extend the framework with extra features through new plug-ins [24]. This proved useful for the CURSOR method whose implementation partly relies on a dedicated plug-in, namely Gena-CWE which is introduced in Section 3.

The Frama-C kernel provides a normalized representation of C programs and ACSL specifications. In addition, it yields several general services for supporting plug-in development and providing convenient features to Frama-C's end-users. The kernel also allows plug-ins to collaborate together either sequentially or in parallel. Sequential collaboration consists in a chain of analyses that perform operations one after another, while parallel collaboration combines partial analysis results from several analyzers to complete a full program verification. The CURSOR method described in Section 3 exemplifies sequential collaboration (it is however worth mentioning that in Frama-C, parallel collaboration relies on kernel consolidation of analysis results detailed in [6]).

The ACSL specification language is a tool of choice when combining Frama-C analyzers. Indeed it is shared by the whole framework: any analyzer may both verify and generate ACSL annotations, which can in turn be verified by other analyzers. Generally, ACSL allows its users to specify functional properties of C programs similarly to Eiffel [22] and JML [21]. It is based on the notion of function contract which specifies the preconditions that are supposed to be true before a call of f (i.e. ensured by the caller), and

[7] `http://frama-c.com`

the postconditions that should be satisfied after the call of f (and should be thus established during the verification of f). Numerous examples may be found in the reference tutorial [5]. The most important feature of ACSL for this paper is assertions: predicates which must hold at a particular program point. It is worth noting that ACSL is powerful enough to be able to express most C99 undefined behaviors with such predicates as illustrated later on this paper. A Frama-C plug-in named RTE may even be used to generate ACSL assertions for every potential errors corresponding to certain families of runtime errors.

2.2 Value

The Value Analysis plug-in of Frama-C (Value for short) [11] automatically computes sets of possible values[8] for the variables of an analyzed program at each program point, by means of abstract interpretation [7]. It also warns about potential runtime errors through the generation of ACSL annotations similar to those of the RTE plug-in. Value aims at being directly usable on any C code in any applicative domain, from low-level system libraries to safety-critical applications. Indeed Value relies on a general-purpose efficient domain (*a.k.a.* lattice) in order to represent the possible values of memory locations at each program point. However this domain can be less precise than a custom domain designed for a particular code pattern. In order to circumvent this issue, Value has recently been improved by Eva (Evolved Value Analysis) [4] which aims at reconciling the efficiency of the Value's domain with the flexibility of collaborative custom domains. Another way to circumvent possible precision issues is provided by Frama-C itself: what cannot be proven by Value (or Eva)[9] may still be proven by another plug-in, possibly a dedicated one.

This Frama-C ecosystem leads to another Value functionality: the possibility to reuse its results. Indeed what Value has computed is (partly) available in the Frama-C GUI and helps the user better understand Value's results. An illustrative example for PolarSSL 1.1.7[10] is presented in Figure 1. It also allows derived analyses like slicing to be sound. In particular it helps these analyses to safely interpret function pointers and find out potential aliasing. In this way, several plug-ins have been developed by academic and industrial users for proving specific goals in a

[8] These sets are named *domains*, and can be represented in their simplest form as intervals of values.

[9] Value and Eva share the same user interface. The rest of this paper applies in the same way to both plug-ins. We will continue to use Value for the sake of consistency.

[10] See `https://tls.mbed.org/`.

safe way without spending too much time with pointer intricacies [2,8,13]. The CURSOR method introduced in this paper also collaborates with Value by inspecting its results in order to discover complementary CWE alarms (see Section 3).

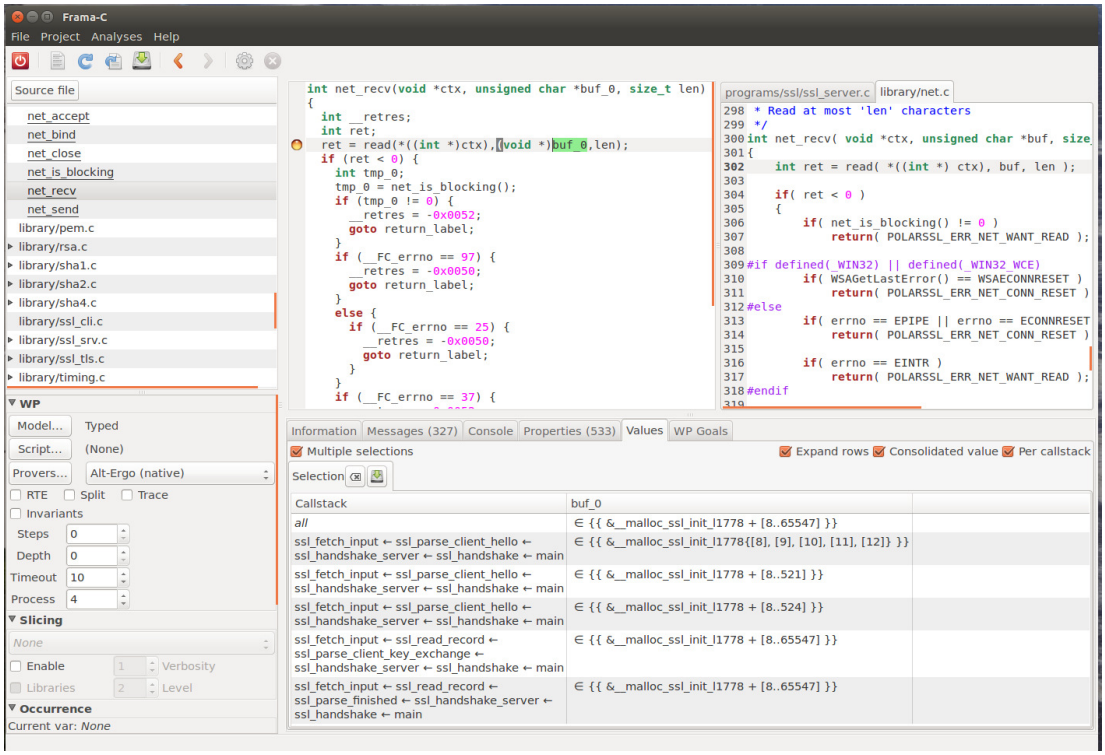

Fig. 1. Frama-C GUI with Value's results on PolarSSL's function `net_recv`. It displays the possible values of the function's parameter `buf` *per* callstack.

2.3 E-ACSL

Frama-C was originally oriented towards static verification. Consequently ACSL has the same bias. In particular, it contains several constructs (*e.g.* unbounded quantifications and axioms) with a well-defined mathematical meaning but no computational semantics because they cannot be executed. To circumvent this issue, an "executable" subset of ACSL has been identified in which all constructs are computationnally well-defined. This specification language is called E-ACSL ("E" stands for "executable") [12]. All annotations in this paper are actually both ACSL and E-ACSL annotations. In particular, all assertions corresponding to detectable CWEs and automatically generated by RTE and Value fall into this category.

A Frama-C plug-in also called E-ACSL is in charge of transforming E-ACSL annotations into executable code. More precisely, E-ACSL is a program transformation tool: it takes as input a C program p annotated with E-ACSL specifications and generates another C program which observationally behaves like p if each annotation is satisfied, or stops on the first failing annotation otherwise. In other words, E-ACSL generates an *inline* monitor [14] for a C program based on its formal specification. The translation scheme is described in Figure 2. It highlights the fact that E-ACSL benefits from Frama-C and uses static analyses to optimize the generated code. For instance, before generating the code, it implements a backward sound over-approximating analysis [16] in order to reduce the instrumentation of memory: if there is no annotation depending on some memory location l (*e.g.* corresponding to some program variable `x`), there is no need to track its allocation, initialization and de-allocation. During code generation, it also uses a dedicated type system to optimize operations over integer (which are mathematical integers in E-ACSL) [15]. It also takes care of not generating C99 undefined behaviors when translating annotations.

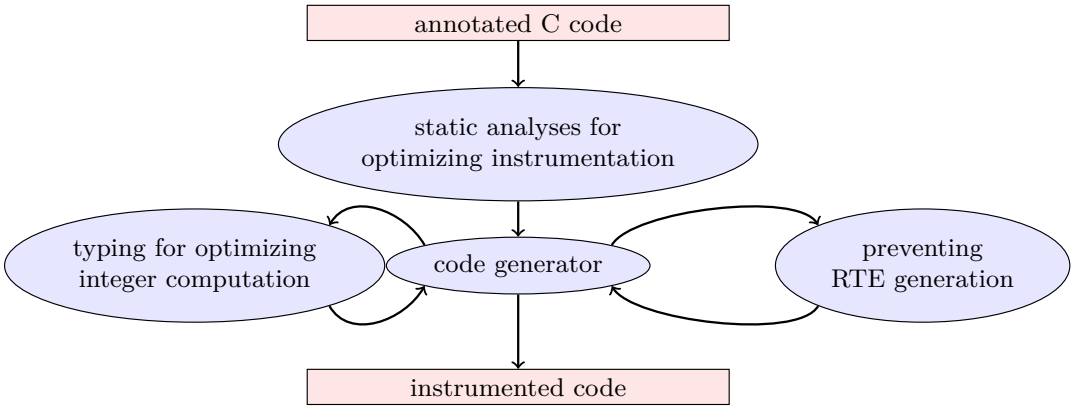

Fig. 2. E-ACSL Translation Scheme.

The E-ACSL plug-in does require E-ACSL-annotated code as input. However there is no need to write these annotations manually. Therefore E-ACSL may really be used as a fully automatic tool. For instance, if interested in certain classes of runtime errors, the user may first use the RTE plug-in to generate E-ACSL annotations and then use E-ACSL to monitor them at runtime. Used this way, E-ACSL is similar to memory debuggers like ASan and MemCheck (built on top of Valgrind). Recent

experiments on programs from SPEC CPU benchmark[11] show that runtime overheads of E-ACSL are on average 19 times the normal execution [28]. That is comparable to MemCheck, while the average overhead of ASan is about 2. It is worth noting that E-ACSL checks more properties than these memory debuggers. The implementation of the CURSOR method uses E-ACSL in this spirit but in a more cost-efficient way (see Section 3). E-ACSL also allows to modify its default behavior which is to stop the program execution whenever an annotation is violated. Once again, the implementation of the CURSOR method relies on this feature.

3 CURSOR: principles and results

The method presented in the following is straightforward for some categories of applications and security flaws. The process is intended to be as simple as possible. For a few families of CWEs, it is almost fully automated for cost-efficiency, and provides a sensitive improvement regarding security flaw robustness.

At first, this process consists of identifying source code libraries of functionalities worth analyzing in order to be strengthened. For instance in the context of web applications and servers, it may consist of sets of functions dealing with low-level file management, sanitizing functions on strings or URLs, etc. These functions may implement SFR (Security Functional Requirements) as defined in Common Criteria[12]. These SFR are intended to meet the security objectives, which means to counter threats in the assumed operating environment of the ToE.

Then, a static formal analysis is performed by applying Frama-C's Value plug-in: at this point, several categories of CWE may be found in the C source code. TrustInSoft Analyzer[13], which is based on the Frama-C platform and Value, was able for instance to identify automatically several CWEs (119 to 127: buffer-related weaknesses, 369: divide-by-zero, 415: double-free, 416: use-after-free, 457: use of uninitialized variable, 476: null pointer dereference, 562: return of stack variable address, 690: unchecked return value to null pointer dereference) on some commercial-of-the-shelf software [27]. When analyzing a source code with Value, the CWEs are expressed as ACSL annotations attached to the related statement. For instance, consider the following code snippet:

[11] `https://www.spec.org/cpu/`

[12] A standardized framework for the definition and validation of the security provided by a given product (*a.k.a.* Target of Evaluation, or *ToE*). `http://www.commoncriteriaportal.org`

[13] `https://trust-in-soft.com`

```
// ...
  j = i + 1;
  x = * (p + j);
// ...
```

Value may raise some alarms as ACSL assertions: for instance an integer overflow[14] on the first statement, and an invalid dereferencing in the second one, as follows.

```
// ...
  /*@ assert Value: signed_overflow: i+1 <= 2147483647; */
  j = i + 1;
  /*@ assert Value: mem_access: \valid_read(p+j); */
  x = * (p + j);
// ...
```

These alarms could be spurious as the runtime context might never lead to an error or actual exploitable security flaw. The false positive alarms may be due to static over-approximations inherent to sound analyses by abstract interpretation, or a too imprecise initial state (i.e. domains of value for each location in memory) at the entry point of the analysis.

Once Value analysis has been performed, categories of CWEs are identified in the source code under study (in our sample code: CWE-190 “Integer Overflow or Wraparound”, and CWE-125 “Out-of-bounds Read”). At this point, the CURSOR method benefits from the possibility to write its own Frama-C plug-ins [24]: CURSOR is enriched by some scripts gathered in a plug-in named Gena-CWE. This plug-in detects complementary categories of CWEs which are not directly displayed by Value but are however computed and can be obtained by adequately querying its abstract states. For instance, among others, it can detect and locate CWE-174 “Double Release of Resource”, CWE-457 “Uninitialized Variables”, CWE-570 “Always False conditions”, CWE-571 “Always True conditions”. If not relevant with regards to risk analysis, the search for some families of CWE can also be deactivated.

In the case of a library, Value is applied function-by-function: the C functions under analysis can potentially be called at runtime from any control point of the application. Thus, it is necessary to analyze these functions with the largest possible input domain for each of their arguments, independently from the calling context. As Value is a context-sensitive analysis, the functions are analyzed and thus annotated separately.

The code, once automatically annotated with ACSL alarms corresponding to potential CWEs, can now be analyzed by the E-ACSL plug-in

[14] It assumes a 32-bit architecture.

in order to translate all these annotations into executable statements. The previous code snippet is then transformed as the following instrumented code:

```
// ...
  /*@ assert Value: signed_overflow: i+1 <= 2147483647; */
  __store_block((void *)(& p),4U);
  e_acsl_assert(i + 1 <= 2147483647,
                (char *)"Assertion",(char *)"f_ABSCAL",
                (char *)"Value: signed_overflow: i+1 <= 2147483647",
                3056);
  j = i + 1;
  /*@ assert Value: mem_access: \valid_read(p+j); */
  {
    int __e_acsl_valid_read;
    __e_acsl_valid_read = __valid_read((void *)(p + j),sizeof(char))
        ;
    e_acsl_assert(__e_acsl_valid_read,(char *)"Assertion",(char *)"
        f_ABSCAL",
                  (char *)"Value: mem_access: \\valid_read(p+j)"
                      ,3058);
  }
  x = (char *)*(p + j);
  __delete_block((void *)(& p));
  __e_acsl_memory_clean();
// ...
```

So far, some code is added by E-ACSL which will be executed at runtime. It mainly contains memory management statements specific to E-ACSL, and condition evaluations corresponding to each alarm[15]. In case the conditions are met during execution (i.e., in the example code above, if the first argument of function `e_acsl_assert` is true), some extra-code will be triggered. Indeed, this extra-code constitutes the implementation of the related counter-measures (CM) in case the ACSL alarms are activated, making the application potentially more robust to attacks based on the targeted weaknesses in the source code. At the origin, `e_acsl_assert` was intended to stop the code execution – thus at runtime – whenever a violation was detected. But in CURSOR this mechanism is extended in such a way that the execution continues, activating some pre-defined counter-measure behaviors. All the arguments of `e_acsl_assert` can be used by the CM functions: of course the boolean condition, but also the kind of emitted alarm ("Assertion" in the example), the function in which this alarm is raised, the alarm expression itself, and finally the location (line number) in the source file.

[15] The interested reader may refer to the E-ACSL documentation [25] and related publications [16, 20, 28] for further details, in particular on how pointer validity is efficiently managed at runtime through the tracking of allocated memory blocks.

In practice, these CM may range from logging the information (storing *what* happened, *where* and *why*, for any further processing as in case of forensics for instance), to halting the service or the system (or even any other defensive or "retaliating" measures!). The choice of the adequate CM is left to the user, with regards to the corresponding security risk analysis – only the latter is able to help identifying efficiently the relevant parts of the software to be CURSOR-ed. It is worth noting that these CM functions can be either specific to the alarm location and its context, or definitely generic. This is possible thanks to the information passed as argument at the calling context generated for each alarm: annotation kind, file location, rationale, etc., on which any on-the-fly filtering will allow to trigger the most suitable counter-measure.

Of course, these CM functions must themselves be free of dreaded weaknesses. The last step of the process will therefore consist in formally verifying the absence of flaws in the implementation of these CM functions according to security functional requirements (which includes avoiding any disclosure of sensitive data related to the *triggered* weaknesses). This is done by means of the Frama-C platform, its static analyzers, and possibly a set of security functional ACSL properties annotated in the source code. The plug-ins mostly involved in these verifications are Value and Wp[16]. When required, they are also combined with dynamic analyses. Indeed, dynamic analyses may help static ones decide the validity of some complex alarms: for instance, dynamic approaches combining fuzzy testing and static techniques [18] can automatically generate executable scenarios that will compromise a given sought security property, and thus confirm the presence of a related vulnerability. Otherwise, the very same property could have required important efforts and expertise to be discharged when only using formal methods. Indeed, static and dynamic techniques are commonly seen as able to palliate their mutual limitations in numerous situations, and their coupling constitutes a promising field of investigation extending results presented in [18].

A more sophisticated – thus less push-button – version of the CURSOR method is currently defined in this regard, yielding some hints on static/dynamic possible combinations, but this approach still needs more experiments and consolidations at this stage.

Finally, the whole annotated code enriched with automatic CM calling contexts and CM implementations can be pentested for further detection of vulnerabilities not caught by static analysis (therefore by dynamic

[16] `http://frama-c.com/wp.html`

analysis means, like automatic security-oriented fuzzers – as AFL[17] –, applied on the instrumented program) and/or embedded into the system for operations. Figure 3 presents the whole process.

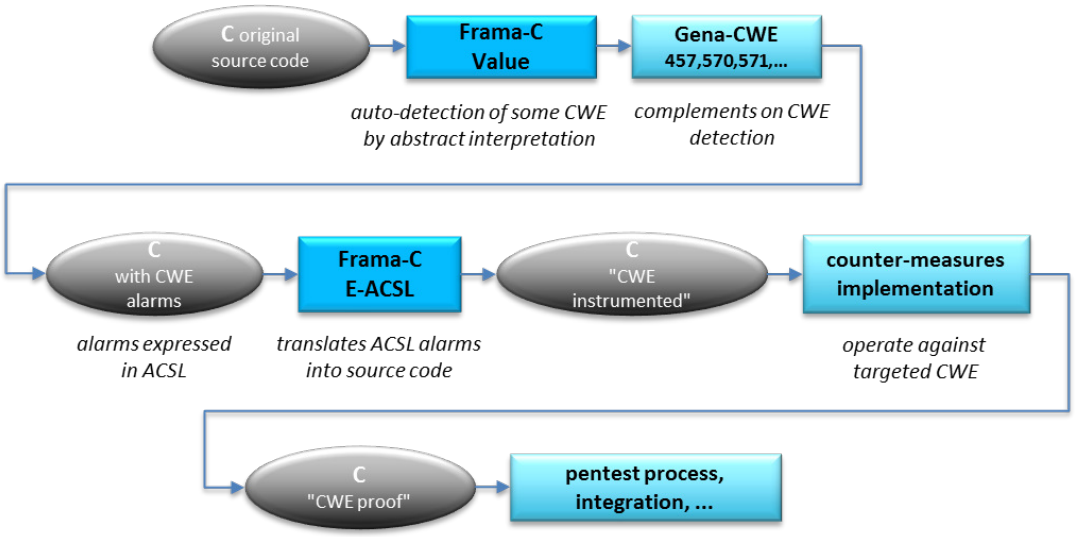

Fig. 3. CURSOR method process.

In this figure, squares represent either existing tools or complements implemented for the sake of the method. It is worth noting that generating, even automatically, as many CM calling contexts as alarms (spurious or not) may result in a lot of extra-code, and in some cases with potential impacts on the execution time and memory consumption at runtime. This point is addressed by several means. First, the method is not applied to the whole application, but only on security sensitive parts (typically some well-chosen libraries). Second, some strategies relying on security risk analysis might allow to get rid of some categories of alarms (or CWEs) emitted by Frama-C plug-ins: from a functional viewpoint, for a given function, the same alarm kind occurring on the same variable but in different control points should be kept only once in the source code, namely the first time it appears in the control flow. This heuristic has been implemented by a new Frama-C script, with no particular difficulty from a technical point of view. However, it may also break the soundness of the formal analyses, and thus should be used with care and duly justified.

Application to Apache libraries. The simplified process presented above was applied to several Apache libraries. As an illustration, it was used

[17] American Fuzzy Lop: `http://lcamtuf.coredump.cx/afl`

to strengthen the `httpd/server/util.c` source code, a representative library containing some file management and sanitizing functions, relevant for security analysis purpose. The snapshot of this (100+ functions) library callgraph is given in Figure 4. This code is originally 7,958 loc (before pre-processing by Frama-C, 14.8 kloc afterward in particular because of macro expansions and inclusion of external libraries). On this library, 340 ACSL annotations are generated by Value and Gena-CWE. E-ACSL plug-in is then applied, expanding instrumented source code to 9,759 loc (excluding counter-measures to be implemented on the user side). This instrumentation represents an increase of about 22% of the original source code size. This code can then be compiled and executed: any violation of a CWE alarm will thus trigger one of the implemented counter-measures, with negligible impact on memory consumption and execution time from a functional standpoint (for more issues about performance, see [28]).

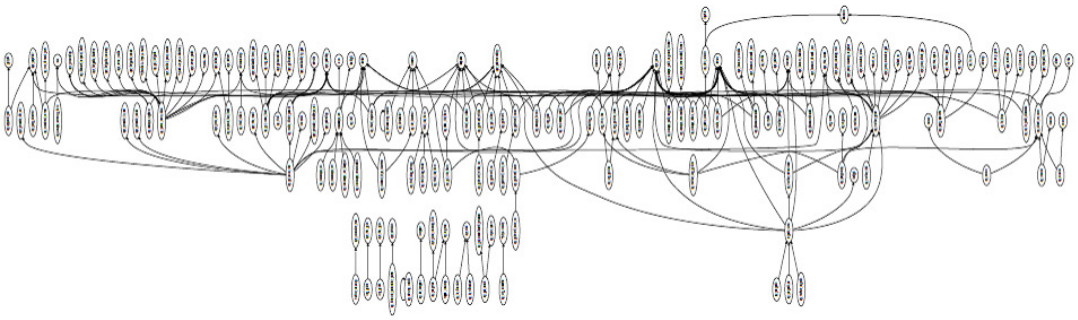

Fig. 4. An Apache library's callgraph.

4 Security Evaluation and Certification Considerations

(This section briefly presents some insights into CURSOR and Common Criteria certification process. For convenience, it can be skipped for the reader not concerned with certification issues.)

The CURSOR method has a clear impact on the attack surface and application behavior, which is definitely modified by inserting the CM calling contexts into the source code. These CM intend to drastically change the behavior of the application under analysis. In this respect, the pentester (and possibly the hacker) will face different reactions for the - apparently - same software according to whether CURSOR was applied or not. Namely, whenever a pentester might expect triggering an error from a given vulnerability in the code (for instance detected after a "fruitful"

white-box manual review of the source code), the CURSOR-ed version of the application will bifurcate at runtime to the corresponding CM which may be as stealthy as possible (at least, the CM should hide clues to the pentester about the original flaw in the code), and thus diverting from the expected original behavior.

Besides, in a Common Criteria (CC) certification context, the method presented in this paper comes with some clear benefits (in the following *ATE_** and *FAU_** identify some CC security requirements):

- Completing in some extent the security functional requirements (SFR) through automatic generation of potential alarms and their executable translation,
- Replacing some documentation effort by generating evidence: providing automated coverage testing (ATE_COV) marks for the new functional requirements,
- Security audit data generation (FAU_GEN): the executable code generated by the CURSOR method allows to log causes of crashes or attack "witnesses", then contributing to the FAU_GEN requirement,
- To some extent, security audit automatic response (FAU_ARP), analysis (FAU_SAA), and review (FAU_SAR) can also be addressed by CURSOR.

These considerations are not exhaustive, and still preliminary at this stage. Further analyses are necessary to assess the perimeter of use and the benefits which can be expected when applying the CURSOR method in a certification process. However, the approach seems already quite promising, and cost-efficient from security and affordability standpoints.

5 Conclusion and Perspectives

The CURSOR method presented in this paper permits to automatically counter some families of attacks based on CWEs. It first consists in generating as many ACSL alarms in the source code as the Frama-C sound formal static analyzers will find, and then translating them – even the spurious ones – as executable code for further pentesting/dynamic analyses and eventually operational use. Most of the process is fully automated, and scalable as the static formal analyses can be done function-by-function. It is also cost-efficient as no particular expertise is needed in formal methods, at least in the simplified version of the method: ACSL annotations corresponding to security alarms are not expected to be discharged during the process.

CURSOR remains a straightforward exploitation of Frama-C plug-ins, with some further developments depending on the complementary and relevant CWEs refined from application-dependant security risk analyses. Moreover, future developments will consist in developing a Frama-C plug-in to automatically classify Frama-C alarms with respect to their corresponding CWEs, while extending the scope of detectable CWEs. First experiments with CURSOR method, performed in an R&T context at this stage, are quite conclusive, even for open source applications for which design and development documentations are either unavailable, or only exploitable with a certain amount of investment not always affordable for applications with a low level of criticality. It is worth noting that this paper includes the very first publicly available R&T evaluation of E-ACSL. Yet the exact impact of the deployment of counter-measures in terms of time efficiency and memory consumption is still to be done, even if recent intensive benchmarking on an evolved version of E-ACSL [28] might suggest it is clearly acceptable. Also E-ACSL is still a young tool and lots of improvements are already planned to enhance memory footprint and computing time consumption of the generated monitors, as well as the scope of sought security alarms.

Some insights were also presented in this paper with regards to security evaluation and CC certification processes, which might be argued and deepened in order to obtain concrete recognition of CURSOR's benefits among security communities and certification bodies.

Acknowledgements: The authors would like to thank Marion Daubignard, Raphaël Rigo and the anonymous reviewers for their helpful and constructive comments that contributed to improving the final version of the paper.

References

1. Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. http://frama-c.com/acsl.html.
2. Pascal Berthomé, Karinne Heydemann, Xavier Kauffmann-Tourkestansky, and Jean-François Lalande. Attack model for verification of interval security properties for smart card C codes. In *Programming Languages and Analysis for Security (PLAS 2010)*, pages 1–12. ACM, 2010.
3. Peter G. Bishop, Robin E. Bloomfield, and Lukasz Cyra. Combining testing and proof to gain high assurance in software: A case study. In *International Symposium on Software Reliability Engineering (ISSRE'13)*, pages 248–257, November 2013.
4. Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring abstract interpreters through state and value abstractions. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2017)*, January 2017.
5. Jochen Burghardt, Jens Gerlach, and Timon Lapawczyk. *ACSL by Example*. 2016. https://gitlab.fokus.fraunhofer.de/verification/open-acslbyexample/blob/master/ACSL-by-Example.pdf.
6. Loïc Correnson and Julien Signoles. Combining Analyses for C Program Verification. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2012)*, volume 7437 of *LNCS*, pages 108–130. Springer, 2012.
7. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL 1977)*, pages 238–252. ACM Press, 1977.
8. Pascal Cuoq, David Delmas, Stéphane Duprat, and Virginia Moya Lamiel. Fan-C, a Frama-C plug-in for data flow verification. In *Embedded Real-Time Software and Systems Congress (ERTS^2 2012)*, 2012.
9. Pascal Cuoq, Philippe Hilsenkopf, Florent Kirchner, Sébastien Labbé, Nguyen Thuy, and Boris Yakobowski. Formal verification of software important to safety using the Frama-C tool suite. In *International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies (NPIC & HMIT)*, 2012.
10. Pascal Cuoq and Julien Signoles. Experience report: Ocaml for an industrial-strength static analysis framework. In *International Confererence on Functional Programming (ICFP 2009)*, pages 281–286, September 2009.
11. Pascal Cuoq, Boris Yakobowski, and Virgile Prevosto. *Frama-C's value analysis plug-in*. `http://frama-c.com/download/value-analysis.pdf`.
12. Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. Common specification language for static and dynamic analysis of C programs. In *the 28th Annual ACM Symposium on Applied Computing (SAC 2013)*, pages 1230–1235. ACM, April 2013.
13. Jonathan-Christofer Demay, Éric Totel, and Frédéric Tronel. SIDAN: a tool dedicated to software instrumentation for detecting attacks on non-control-data. In *International Conference on Risks and Security of Internet and Systems (CRiSIS 2009)*, pages 51–58. IEEE, 2009.
14. Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 141–175. IOS Press, 2013.

15. Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. Rester statique pour devenir plus rapide, plus précis et plus mince. In David Baelde and Jade Alglave, editors, *Journées Francophones des Langages Applicatifs (JFLA'15)*, Le Val d'Ajol, France, January 2015. In French.
16. Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. Fast as a Shadow, Expressive as a Tree: Optimized Memory Monitoring for C. *Science of Computer Programming*, pages 226–246, oct 2016.
17. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspect of Computing*, 27(3):573–609, 2015.
18. Balázs Kiss, Nikolai Kosmatov, Dillon Pariente, and Armand Puccetti. Dynamic analyses for vulnerability detection: Illustration on heartbleed. In *Software Verification and Testing - 11th International Haifa Verification Conference, HVC'15, Haifa, Israel, November 17-19, 2015, Proceedings*. Springer Verlag, 2015.
19. Nikolai Kosmatov and Julien Signoles. Frama-C, a Collaborative Framework for C Code Verification. Tutorial Synopsis. In *International Conference on Runtime Verification (RV 2016)*, September 2016.
20. Nikolaï Kosmatov, Guillaume Petiot, and Julien Signoles. An Optimized Memory Monitoring for Runtime Assertion Checking of C Programs. In *International Conference on Runtime Verification (RV'13)*, sep 2013.
21. Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accomodates both runtime assertion checking and formal verification. In *International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS*, pages 262–284. Springer, 2002.
22. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1988.
23. Julien Signoles. Software Architecture of Code Analysis Frameworks Matters: The Frama-C Example. In *Workshop on Formal Integrated Development Environment (F-IDE 2015)*, pages 86–96, June 2015.
24. Julien Signoles, Loïc Correnson, Matthieu Lemerre, and Virgile Prevosto. *Frama-C Plug-in Development Guide*. `http://frama-c.com/download/plugin-development-guide.pdf`.
25. Julien Signoles and Kostyantyn Vorobyov. *E-ACSL User Manual*. `http://frama-c.com/download/e-acsl/e-acsl-manual.pdf`.
26. Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In *Formal Methods (FM'09)*, November 2009.
27. TrustInSoft. PolarSSL 1.1.8 verification kit, v1.0. Technical report, 2015.
28. Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. Shadow State Encoding for Efficient Monitoring of Block-level Properties. Submitted for publication.